I. The C Language

NAME

Introduction - the C compiler

FUNCTION

The C compiler is a set of three programs that take as input, to the first of the programs, one or more files of C source code, and produce as output, from the last of the programs, assembler code that will perform the semantic intent of the source code. Output from the files may be separately compiled, then combined at load time to form an executable program; or C subroutines can be compiled for later inclusion with other programs. One can also look on the compiler as a vehicle for implementing an instance of an abstract C machine, i.e., a machine that executes statements in the language defined by some standard. That standard is generally accepted to be Appendix A of Kernighan and Ritchie, "The C Programming Language", Prentice-Hall 1978.

This section describes the current implementation, as succinctly, and, it is to be hoped, as precisely as it is defined in the language standard. It is organized into loosely coupled subsections, each covering a different aspect of the language. No serious attempt is made to be tutorial; the interested student is referred to Kernighan and Ritchie, then back to this section for a review of the differences.

The recommended order of reading is:

Introduction - this is it.

Syntax - how to spell the words and punctuate the statements.

Identifiers - the naming of things, the scope of names and their important attributes.

Declarations - how to introduce identifiers and associate attributes with them.

Initializers - how to specify the initial values of all sorts of data types.

Statements - how to specify the executable code that goes with a function name.

Expressions - the binding of operators, order of evaluation, and coercion of types.

Constants - the kinds of expressions that can be evaluated at compile time.

Preprocessor - #define and #include expansion.

Style - recommendations for what parts of the language to use, and what to avoid; how to format code.

Portability - techniques for writing C that is maximally portable.

Differences - comparative anatomy of this implementation, the language standard, and other implementations.

Diagnostics - things the compiler complains about.

SEE ALSO

Documentation for compiler operation proper is found in the various C Interface manuals, which also contain the descriptions of subroutines used to communicate with various operating systems. The standard library of C callable functions is documented in the remaining sections of this manual.

NAME

Syntax - syntax rules for C

FUNCTION

At the lowest level, a C program is represented as a text file, consisting of lines each terminated by a newline character. Any characters between /* and */ inclusive, including newlines, are comments and are replaced by a single blank character. A newline preceded by '\' is discarded, along with any leading whitespace on the next line, so that lines may be continued within character or string constants, discussed below. The compiler cannot deal with a text line larger than 512 characters, either before or after the processing of comments and continuations.

Text lines are broken into tokens, strings of characters possibly separated by whitespace. Whitespace consists of one or more non-printing characters, such as space, tab, or newline; its sole effect is to delimit tokens that might otherwise be merged. Tokens take several forms:

An identifier - consists of a letter or underscore, followed by zero or more letters, underscores, or digits. Uppercase letters are distinct from lowercase letters; no more than eight characters are significant in comparing identifiers. More severe restrictions may be placed on external identifiers by the world outside the compiler (see Differences). There are also a number of identifiers reserved for use as keywords:

auto	extern	short
break	float	sizeof
case	for	static
char	goto	struct
continue	if	switch
default	int	typedef
do	long	union
double	register	unsigned
else	return	while

A numeric constant - consists of a decimal digit, followed by zero or more letters and digits. If the leading characters are "0x" or "0X", the constant is a hexadecimal literal and may contain the letters 'a' through 'f', in either case, to represent the digit values 10 through 15, respectively. Otherwise a leading '0' implies an octal literal, which nevertheless may contain the digits '8' and '9'. A non-zero leading digit implies a decimal literal. Any of these forms may end in 'l' or 'L' to specify a long constant. The constant is also made long if a) a decimal literal cannot be properly represented as a signed integer, or b) any other literal constant cannot be properly represented as an unsigned integer. Overflow is not diagnosed.

A floating literal - consists of a decimal integer part, a decimal point '.', a decimal fraction part, and an exponent, where an exponent consists of an 'e' or 'E' and an optionally signed ('+' or '-') decimal power of ten by which the integer part plus fraction part must be multiplied. Either the decimal point or the exponent may be omitted, but not both; either the integer part or the fraction part may be

omitted, but not both. Any numeric constant that is not one of these literal forms is illegal, e.g., "5ax3". A floating literal is of type double. Overflow is not diagnosed.

A character constant - consists of a single quote, followed by zero or more character literals, followed by a second single quote. A character literal consists of a) any character except '\', newline, or single quote, the value being the ASCII representation of that character; b) a \ followed by any character except newline, the value being the ASCII representation of that character, except that characters in the sequence <'b', 't', 'v', 'f', 'n', 'r', '(', '!', ')', '^'> have the ASCII values for the corresponding members of the sequence <backspace, horizontal tab, vertical tab, form feed, newline, carriage return, '{', '|', '}', '~'>; or c) a '\' followed by one to three decimal digits, the value being the octal number represented by those digits. A newline is never permitted inside quotes, except when escaped with a '\' for line continuation. The value of the constant is an integer base 256, whose digits are the character literal values. The constant is long if it cannot be properly represented as an unsigned integer. Overflow is not diagnosed.

A string constant - is just like a character constant, except that double quotes '"' are used to delimit the string. The value is the (secret) name of a NUL terminated array of characters, the elements initialized to the character literals in the string.

Punctuation - consists of predefined strings of one to three characters. The complete set of punctuation for C is:

!	*	:	=+	>)	\)
!=	+	;	=-	>=	\^
%	++	<	=/	>>	^
&	,	<<	=<<	?	{
&&	-	<=	==	[	|
(	--	=	=>>	]	|)
(<	->	=%	=^	\!	||
(|	.	=&	=|	\!!	}
)	/	=*	>	\(	~

Punctuation in the sequence <'(<', '(|', '>)', '\!', '\!!', '\(', '\)', '\^', '|)'> is entirely equivalent to the corresponding member of the sequence <'{', '[', '}', '|', '||', '{', '}', '~', ']'>.

The longest possible punctuation string is matched, so that "==+", for example, is recognized as "==", "+", and never as "=", "=+".

Other characters - such as '@' or '\' alone are illegal outside of character or string constants, and are diagnosed.

NOTATION

Grammar, the rules by which the syntactic elements above are put together, permeates the remaining discussions. To avoid long-winded descriptions, some simple shorthand is used throughout this section to describe grammatical constructs.

A name enclosed in angle brackets, such as <statement>, is a "metanotion", i.e., some grammatical element defined elsewhere. Presumably any sequence of tokens that meets the grammatical rules for that metanotion can be used in its place, subject to any semantic limitations explicitly stated. Just about any other symbol stands for itself, i.e., it must appear literally, like the semicolon in

```
<expression> ;
```

Exceptions are the punctuation "[", "]", "]*", "{", "}", and "|"; these have special meanings unless made literal by being enclosed in single quotes.

Brackets surround an element that may occur zero or one time. The optional occurrence of a label, for instance, is specified by:

```
[ <identifier> : ] <statement>
```

This means that the metanotion <identifier> may but need not appear before <statement>, and if it does must be followed by a literal colon. To specify the optional, arbitrary repetition of an element, the notation "[]*" is used. A comma separated list of <id> metanotions, for example (i.e., one instance of <id> followed by zero or more repetitions), would be represented by:

```
<id> [, <id> ]*
```

Vertical bars are used to separate the elements in a list of alternatives, exactly one of which must be selected. The line:

```
char | short | long
```

requires the specification of any one of the three keywords listed. Such a list of alternatives is enclosed in braces in order to precisely delimit its scope. For instance:

```
{ <decl> [ = ] <expr> |
  <decl> [ = ] <elist> }
```

emphasizes that a data initializer has the format given by the entirety of one of the two lines specified.

EXAMPLE

Various ways of writing a ten:

```
integer  10, 012, 0Xa, '\n'
long     10L
double   10.0, 1e1, 1.0e+01
```

NAME

Identifiers - naming things in C

FUNCTION

Identifiers are used to give names to the objects created in a C program. Syntactically, an identifier is a sequence of letters, underscores, and digits, starting with a non-digit. For the sake of comparisons, only the first eight characters are significant, so "summation" and "summations" are the same identifier. Externally published identifiers are typically even more restricted (see Differences).

There are several name spaces in a C program, so that the same identifier may have different meanings in the same extent of program text, depending on usage. Things such as struct or union tags, members of struct or union, and labels will be considered separately later on. The bulk of this discussion concerns the name space inhabited by the names of objects that occupy storage at execution time.

Each such identifier acquires, from its usage in a C program, a precisely defined lexical scope, storage class, and type. The scope is the extent of program text over which the compiler knows that a given meaning holds for an identifier. The storage class determines both the lifetime of values assigned to an object and the extent of program text over which a given meaning holds for its identifier, whether the compiler knows it or not. The type determines what operations can be performed on an object and how its values are encoded. Needless to say, these important attributes often interact.

SCOPE

There are two basic contexts in a C program -- inside a program block and outside it. The program block can be the entire body of a defined function, including its argument declarations, or any contained region enclosed in braces "{}". In either case, the scope of an identifier depends strongly on what context it is first mentioned in.

If an identifier first appears outside a program block, its scope is from its first appearance to the end of the file, less any contained program blocks in which that identifier is explicitly declared to have a storage class other than extern, i.e., a local redeclaration. To legally appear outside a program block, an identifier must be a) explicitly declared to be extern or static, or b) used in an initializer for an object of type pointer. In the latter case, the identifier is implicitly declared to be extern, with type (tentatively) int.

If an identifier appears inside a program block and is explicitly declared to have a storage class other than extern, its scope is from that appearance to the end of that program block, less any local redeclarations.

The only other place an identifier may legally first appear is inside a program block, within an expression, where the name of a function is required. In this case the identifier is implicitly declared to be extern, with type function returning (tentatively) int.

In short, locals remain local while externals are made known as globally as possible, without requiring the compiler to back up over the text.

STORAGE CLASSES

Storage classes come in a variety of flavors, some with different seasoning depending on context.

extern - outside a program block, means that the name should be published for common use among any of the files composing the program that also publish the same name. The published name may be shortened to as few as six significant characters, and/or compressed to one case, depending on the target operating system; so while the compiler distinguishes "Counted" and "counter", subsequent processing of the compiled text may not. Inside a program block, extern merely emphasizes that an earlier definition in a containing block holds, if any. If none, then the name is published as above. The lifetime of extern objects is the duration of program execution.

static - outside a program block, means that the name should not be published outside the file. Inside a program block, static means that the identifier names an object known only within the program block, less any local redeclarations. The lifetime of static objects is the duration of program execution, so the value of a local static is retained between invocations of the program block that knows about it.

auto - can only be declared inside a program block and means that the identifier names an object known only within the program block, less any local redeclarations. The lifetime of auto objects is the time between each entry and exit of the program block, so the value of an auto is lost between invocations of the program block that knows about it. Multiple instances of the same auto may exist simultaneously, one instance for each dynamic activation of its program block.

register - can only be declared inside a program block, and means much the same as auto, except that a) efficient storage, such as the machine's fast registers, should be favored to hold the object, and b) the address of the object cannot be taken. It is not considered an error to declare more objects of class register than can be accommodated; excess ones are simply taken as auto. The lifetime of register objects is the same as auto objects. An argument declared to be of class register is copied into a fast register on entry to the function. Currently, all implementations support at least three simultaneous register declarations, none of which hold an object larger than int.

typedef - means that the name should be recognized as a type specifier, not associated with any object. Lifetime is hence irrelevant. Redeclaring a typedef in a contained program block is permissible but mildly perilous.

TYPE

All types in C must be built from a fixed set of basic types: the integer forms char, unsigned char, short, unsigned short, long, and unsigned long; and the floating types float and double. The type int is a synonym either for short or long integer, depending on the size of pointers on the target machine; char, short, int, and long are signed unless explicitly declared to be unsigned. From these are derived the composite forms struct, union, bitfield, pointer to, array of, and function returning. Recursive application of the rules for deriving composite types leads to a large, if not truly infinite, assortment of types.

[unsigned] char - is a byte integer, something just big enough to hold all of the characters in the machine's character set. It is promised that printable characters and common whitespace codes are small positive integers or zero.

[unsigned] short - is typically a two-byte integer, something just big enough to hold reasonable counts.

[unsigned] int - is either a short or a long, depending on the machine. It is promised to be big enough to count all the bytes in the machine's address space.

[unsigned] long - is typically a four-byte integer, something comfortably large.

float - is a floating number of short precision, typically four bytes.

double - is a floating number of longer precision than float, if possible, typically eight bytes. Also known as "long float".

struct - is a sequence of one or more member declarations, with holes as needed to keep everything on proper storage boundaries for the machine. There are contexts in which a struct may have unknown content. Members may be any types but function returning, array of unknown size, and struct of unknown content.

union - is an alternation of one or more members, the union being big enough and aligned well enough to accept any of its member types. Members may be any types but function returning, array of unknown size, and struct of unknown content. A union of unknown content is treated just like a struct of unknown content.

bitfield - is a contiguous subfield of an unsigned int, always declared as a member of struct. It participates in expressions much like an unsigned int, except that its address may not be taken.

pointer to - is an unsigned int that is used to hold the address of some object. It is promised that no C object will ever have an address of zero.

array of - is a repetition of some type, whose size is either a compile-time constant or unknown. Any type but function returning, array of unknown size, and struct of unknown content may be used in an array.

function returning - is a body of executable text whose invocation returns the value of some type. Only the basic types or pointer to may be returned by a function.

OTHER NAME SPACES

struct or union tags have a scope that extends from first appearance through the end of the program file; they may not be redefined. struct tags are a separate name space, and union tags are a separate name space.

Labels in a function body have a scope that extends from first appearance, in a goto or as a statement label, through the end of the function body; they may not be redefined within that scope. Labels are a separate name space.

Members of a struct or union have a scope that extends from first appearance, in the content definition of the struct or union, through the end of the program file. They may not be redefined within any struct or union, unless the new definition calls for the same type and offset. [N.B. As a compile time option, each struct or union may be given its own name space.]

NAME

Declarations - declaring names in C

FUNCTION

Declarations form the backbone of a C program. They are used to associate a scope, storage class, and type with most identifiers, to specify the initial values of objects named by identifiers, and to introduce the body of executable text associated with each function name. There are four types of declarations, external, structure, argument, and local. The cast operator uses an abbreviated form of declaration to specify type.

EXTERNAL DECLARATIONS

each having one of the forms:

```
[ <sc> ] [ <ty> ] <decl> <fn-body>
[ <sc> ] [ <ty> ] [ <decl> [ <dinit> ] [, <decl> [ <dinit> ] ]* ] ;
```

i.e., a storage class and type specifier, optionally followed by either a function body or a comma separated list of declarators <decl>, each optionally initialized, the list ending in a semicolon.

The storage class <sc> may be extern, static, or typedef; default is extern. The type <ty> may be a) a basic type, b) a struct or union declaration, described below, or c) an identifier earlier declared to be a typedef; default is int. The basic types may be written as:

```
{ [ unsigned ] [ char | short | long ] [ int ] |
  [ long ] float |
  double }
```

where long float is the same as double.

A <decl> is recursively defined as, in order of decreasing binding:

ident - ident is of type <ty>.

<decl> ([<id> [, <id>]*]) - <decl> is of type function returning <ty>. The comma separated list of identifiers is used only if the declaration is associated with a function body.

<decl> '[' <const> ']' - where '[' and ']' signify actual brackets. <decl> is of type array of <ty>. <const> is the unsigned repetition count.

<decl> '[' ']' - where '[' and ']' signify actual brackets. <decl> is of type array of <ty>, of unknown size.

***<decl>** - <decl> is of type pointer to <ty>.

(<decl>) - <ty> is redefined, inside the parentheses, as that type obtained for X if the entire declaration were rewritten with (<decl>) replaced by X.

The last rule has profound implications. It is intended, along with the rest of the <decl> notation, to permit declarators to be written much as they appear when used in expressions. Thus, "*" for "pointer to" corresponds to '*' for "indirect on", "()" for "function returning" corresponds to "()" for "called with", and "[]" for "array of" corresponds to "[]" for "subscripted with". Declarators must thus be read inside out, in the order in which the operators would be applied.

The two critical examples are:

```
int *fpi(); /* function returning pointer to int */
int (*pfi)();  /* pointer to function returning int */
```

Accepting these Truths is the first step on the path to Enlightenment.

Initializers come in two basic flavors, for objects of type function returning, and for everything else. The former is usually referred to as the definition of a function, i.e., the body of executable text associated with the function name. A typedef may not be initialized; a static must be initialized exactly once in the program file; an extern must be initialized exactly once among the entire set of files making up a C program.

Function bodies <fn-body> are described in Statements; data initializers <dinit> are described in Initializers. For now it will merely be observed that each function body begins with an argument declaration list, and each program block within the function body begins with a local declaration list. Functions may only be declared to return a basic type or a pointer to some other type.

STRUCTURE DECLARATIONS

If the type specifier in a declaration begins with struct or union, it must be followed by one of the forms:

```
{ <tag> '{' <dlist> '}' |
  <tag> |
  '{' <dlist> '}' }
```

where '{' and '}' signify actual braces. The first form defines the content of the structure as <dlist> and associates the definition with the identifier <tag>. The second form can be used to refer either to a structure of unknown content or as an abbreviation for an earlier instance of the first form. The last form is used to define content without defining a tag.

dlist is a sequence of one or more member declarations of the form:

```
[ <ty> ] <sudecl> [, <sudecl> ]* ;
```

where <sudecl> is one of the forms:

```
{ <decl> [ : <width> ] |
  : <width> }
```

<ty> and <decl> are the same as for external declarations, except that the types function returning, array of unknown size, and struct of unknown content may not be declared.

If the type is int or unsigned int, a bitfield specifier may follow <decl>, or stand alone. It consists of a colon ':' followed by a compile-time constant giving its width in bits. Adjacent <sudecl> declarators with bitfield specifiers are packed, as tightly as possible, into adjacent bitfields in an unsigned int; bitfield specifiers that stand alone call for unnamed padding. A new unsigned int is begun a) for the first field specifier in a declaration, b) for the first bitfield specifier following a non-bitfield specifier, c) for a bitfield specifier that will not fit in the remaining space in the current unsigned int, or d) for a stand alone field specifier whose width is zero, e.g., ": 0". Bitfields are packed right to left, i.e., the least significant bit is used first.

ARGUMENT DECLARATIONS

A function initializer begins with an argument declaration list, which is a sequence of zero or more declarations of the form:

```
[ register ] [ <ty> ] <decl> [, <decl> ]* ;
```

<ty> and <decl> are the same as for external declarations, except that the types char (and possibly short), function returning, array of, struct, and union are misleading if used. On a function call, any integer type shorter than int is widened to int; function returning and array of become pointers; and struct or union cannot be sent, so any such declaration is a (possibly dangerous) reinterpretation of the actual arguments sent.

The only storage class that may be declared is register; default is normal argument. The default type for undeclared arguments is int.

LOCAL DECLARATIONS

Each program block begins with a local declaration list, which is a sequence of zero or more declarations having one of the forms:

```
{ <lsc> [ <ty> ] [ <decls> ] ; |
  [ <lsc> ] <ty> [ <decls> ] ; }
```

where <decls> is

```
<decl> [ <linit> ] [, <decl> [ <linit> ] ]*
```

In other words, either the storage class <lsc> or type <ty> must be present.

The storage class <lsc> may be auto, register, extern, static, or typedef; default is auto. <ty> and <decl> are the same as for external declarations.

A static may be followed by a data initializer <linit>, just like the <dinit> of external declarations, described in Initializers. An auto or register may be followed by an linit of the form: optional '=', followed by any expression that may appear as the right operand of '=' in an expression in the same context. Such initializers for auto and register become code which is executed on each entry to the program block.

A register may hold only an object of size int (which includes unsigned and pointer). Anything larger than an unsigned int declared to be in a register is quietly made an auto; anything declared smaller than int is taken as register int.

CASTS

A cast is an operator that coerces a value to a specified type. It takes the form

```
( [ <ty> ] <a-decl> )
```

<ty> is the same as for external declarations, except that, in conjuncion with <a-decl>, only the basic types and pointer to may be specified. <a-decl> is an abstract declarator, much like the <decl> used for external declarations, but with the identifier omitted. Thus

```
(int *) /* coerces to pointer to int */
(struct x *(*)())   /* coerces to pointer to
                       function returning pointer to
                       struct x (!) */
```

To eliminate a lurking ambiguity before it bites, "()" is always taken as function returning, and never as (unnecessary) parentheses around the omitted identifier.

EXAMPLE

Some simple declarations:

```
char c;
int i, j;
long lo {37};
double df();
```

More elaborate:

```
int *fpi(); /* fpi is a function returning pointer to int */
typedef struct {
    double re, im;
    } COMPLEX;  /* COMPLEX is a synonym for the structure */
static COMPLEX *pc; /* pc is a static pointer to COMPLEX */
```

NAME

Initializers - giving values to data

FUNCTION

As part of the declaration process, a data object can be given an initial value. This value is established at load time, for objects with storage class extern or static, or on each entry to a program block, for objects with storage class auto or register. If no initializer is specified in a declaration then: an extern must be initialized in another declaration (not necessarily in the same file), a static outside a program block must be initialized in another declaration in the same file, a static inside a program block is set to all zeros, while an auto or register contains garbage.

The two basic formats for initializers are:

```
{ <decl> [ = ] <expr> |
  <decl> [ = ] <elist> }
```

where <elist> is

```
'{' [ <expr> | <elist> ] [, [ <expr> | <elist> ] ]* [, ] '}'
```

i.e., a comma-separated list of expressions and lists, each list enclosed in braces. Note that a trailing comma is explicitly permissible in an elist.

auto and register declarations with initializers behave much like assignment statements. Only scalar variables may be initialized, but the initializer may be any expression that can appear to the right of an assignment operator with that variable on the left. An <elist> is never acceptable in an auto or register initializer.

The remaining discussion concerns initializers for objects with storage class extern or static.

A scalar object is initialized with one <expr>. If it is an integer type (char, short, int, long, or bitfield), <expr> must be an expression reducible at compile time to an integer literal, i.e., a constant expression as described in Constants. If the object is a floating type (float or double), <expr> must be a floating literal or a constant expression; a constant expression is converted to a floating literal by the compiler. The compiler will not perform even obvious arithmetic involving floating literals, other than to apply unary '+' or '-' operators.

A pointer is initialized with a constant expression or with the address of an external object, plus or minus a constant expression. Any constant other than zero is extremely machine dependent, hence this freedom should be exploited only by hardware interface code. If the address of an object appears in a pointer initializer, and the object has not yet been declared, it is implicitly declared to be (tentatively) an external int.

A union is initialized with one expression; the first member of the union is taken as the object to be initialized.

A struct is initialized with either an expression or a list. If an expression is used, the first named member of the struct is taken as the object to be initialized. Unnamed padding is implicitly initialized to zeros, in the presence of other initializers for the struct. If a list is used, the elements of the list are used to initialize corresponding named members of the struct. If there are more named members than initializers, excess members are initialized with zeros. It is an error for there to be more initializers in a list than named members in the struct.

An array of known size is initialized much like a struct: an expression initializes the first element only, while a list initializes elements starting with the first. If there are more elements than initializers, excess elements are initialized with zeros. It is an error for there to be more initializers in the list than there are elements in the array.

An array of unknown size, however, cannot have an excess of initializers, as its multiplicity is determined by the number of initializers provided. After initialization, therefore, an array always has a known size.

By special dispensation, an array of characters may be initialized by a string literal. Thus

```
char a[] {"help"};  /* is the same as */
char a[5] {'h', 'e', 'l', 'p', '\0'};
```

Elaborate composite types, such as arrays of structs, are naturally initialized with lists of sublists, whose structure reflects the structure of the creature being initialized. It is often permissible, however, to write an initializer by eliding braces around one or more sublists. In this case, a struct or array (sub)element uses only as many elements as it needs, leaving the rest for subsequent subelements.

In general, it is recommended that complex initializers either have a structure that exactly matches the object to be initialized, or have no internal structure at all. It is hard enough to get either of these extremes correct; intermediate forms frequently defy analysis.

EXAMPLE

```
char *p = "help";   /* p points at the string */
char a[] {"help"};  /* a contains five chars */
struct complex {
    float real, imag;
    } xx[3][3]
    { { {0, 0}, {0, 1}, {0, 2} },
      { {1, 0}, {1, 1}, {1, 2} },
      { {2, 0}, {2, 1}, {2, 2} } };
```

NAME

Statements - the executable code

FUNCTION

A C function definition consists of the function declaration proper, followed by any argument declarations, followed by a <program-block> which describes the action to be performed when the function is called. A <program-block> begins with a '{', optionally followed by a sequence of local declarations, optionally followed by a sequence of statements, and ends with a '}'. In addition to the <program-block> just described, which may be used recursively whenever a <statement> is permitted, the following are the legal <statement>s of a C program:

<expression>; - An <expression> terminated by a semicolon is a statement that causes the <expression> to be evaluated and the result discarded. Assignments and function calls are simply special cases of the expression statement. It is considered an error if the <expression> produces no useful side effect, i.e., "a = b;" is useful but "a + b;" is not.

; - A semicolon standing alone is a null statement. It does nothing.

if (<expression>) <statement> [else <statement>] - If <expression> evaluates to a non-zero of any type, the <statement> following it is evaluated and the else part, if present, is skipped; otherwise the <statement> following <expression> is skipped and the else part, if present, has its <statement> evaluated. As in all languages, each else part in a nested if <statement> is associated with the innermost "un-else'd" if.

switch (<expression>) <statement> - If <expression>, converted to an int, matches the value associated with any of the case labels in the statement following, execution resumes immediately following the matching label. Otherwise if the label "default" is present in the statement following, execution resumes immediately following it. Otherwise execution resumes with the statement following the switch statement. The statement controlled by a switch is typically a <program-block>, but doesn't have to be.

case <value>: <statement> - The case <statement> may only occur within a switch <statement>, as described above. The value must be an int computable at compile time and must not match any other case <value>s in the same switch.

default: <statement> - The default <statement> may only occur within a switch <statement>, as described above. It may occur at most once in any switch.

while (<expression>) <statement> - So long as <expression> evaluates to a non-zero of any type, <statement> is executed. <expression> is evaluated prior to each execution of the <statement>, plus one more time if it ever evaluates to zero. The <statement> may thus be executed zero or more times.

do <statement> while (<expression>); - The <statement> is executed and, so long as <expression> evaluates to non-zero of any type, the <statement> is repeated. The <statement> may be executed one or more times; <expression> is evaluated following each execution of the <statement>.

for (<ex1>; <ex2>; <ex3>) <statement> - <ex1>, <ex2>, and <ex3> are all <expression>s. <ex1> is evaluated exactly once, then so long as <ex2> evaluates to non-zero the <statement> is executed and <ex3> is evaluated. Thus the for behaves much like the sequence {<ex1>; while (<ex2>) {<statement> <ex3>; }}

break; - A break <statement> causes immediate exit from the innermost containing switch, while, do, or for <statement>, i.e., execution resumes with the <statement> following. A break <statement> may only occur inside a switch, while, do, or for.

continue; - A continue <statement> causes immediate exit from the <statement> part of the innermost containing while, do or for <statement>, i.e., execution resumes with the test part of a while or do, or with the <ex3> part of a for. A continue <statement> may only occur inside a while, do, or for.

goto <identifier>; - A goto <statement> causes execution to resume immediately following the <statement> labelled with the matching identifier contained within a common <program-block>. Such a labelled <statement> must be present.

<identifier>: <statement> - A label <statement> serves as a potential target for a goto, as described above. All labels within a given <program-block> must have unique <identifier>s.

return [<expression>]; - If the <expression> is present, it is evaluated and coerced to the type returned by the function, then the function returns with that value. If the <expression> is absent, the function returns with an undefined value. There is an implicit return statement (with no defined value) at the end of each <program-block> at the outermost level of a function definition.

NAME

Expressions - computing values in C

FUNCTION

C offers a rich collection of operators to specify actions on integers, floats, pointers and, occasionally, composite types. Operators can be classified as addressing, unary, or binary. Addressing operators bind most tightly, left to right from the basic term outward; then all unary operators are applied right to left, beginning with (at most one) postfix "++" or "--"; finally all binary operators are applied, binding either left to right or right to left and on a multi-level scale of precedence.

Parentheses may be used to override the default order of binding, without fear that redundant parentheses will alter the meaning of an expression, i.e., f(p) is the same as f((p)) or (f(p)). The language makes few promises about the order of evaluation, however, or even whether certain redundant computations occur at all. Expressions with multiple side effects can thus be fragile, e.g., *p++ = *++p can legitimately be evaluated in a number of incompatible ways.

Some operands must be in the class of "lvalues", i.e., things that make sense on the left side of an assignment operator. An identifier is the simplest lvalue, but any expression that evaluates to a recipe for locating declared objects can also be an lvalue. All scalar expressions also have an "rvalue", i.e., a thing that makes sense on the right side of an assignment operator. All lvalues are also rvalues.

Nearly all C operators deal only with scalar types, i.e., the basic types, bitfield, or pointer to. Where a scalar type is required and a composite type is present, the following implicit coercions are applied: array of ... is changed to pointer to ... with the same address value; function returning ... is changed to pointer to function returning ... with the same address value; structure or union is illegal.

ADDRESSING OPERATORS

func([expr [, expr]*]) - func must evaluate to type "function returning ..." and is the function to be called to obtain the rvalue of the expression, which is of type ... Any arguments are evaluated, in unspecified order, and fresh copies of their values are made for each function call (thus the function may freely alter its arguments with limited repercussions). char or short expressions are widened to int and float to double; all arguments must be scalar. No checking is made for mismatched arguments or an incorrect number of arguments, but no harm is done providing the highest numbered argument actually used and all its predecessors do correspond properly. Note that a function declared as returning anything smaller than an int actually returns int, while a function returning float actually returns double.

a[i] - is entirely equivalent to "*(a + i)", so the unary '*' and binary '+' should be examined for subtle implications. If a is of type array of ... and i is of integer type, however, the net effect is to deliver the ith element of a, having type ...

x.m - x must be an lvalue, which should be of type struct or union containing a member named m (m can never be an expression). The value is that of the m member of x, with type specified by m.

p->m - p must be coercible to a pointer, which should be a pointer to a struct or union containing a member named m (m can never be an expression). The value is that of the m member of the struct or union pointed at by p, with type specified by m.

UNARY OPERATORS

***p** - p must be of type pointer to ... The value is the value of the object currently pointed at by p, with type ...

&x - x must be an lvalue. The result is a pointer that points at x; the type is pointer to ... for x of type ...

+x - x must be of type integer or float. The result is an rvalue of the same value and type as x.

-x - x must be of type integer or float. The result is an rvalue which is the negative of x and the same type as x.

++x - x must be a scalar lvalue. x is incremented in place by one, following the rules of addition explained below. The result is an rvalue having the new value and the same type as x.

--x - x must be a scalar lvalue. x is decremented in place by one, following the rules of addition explained below. The result is an rvalue having the new value and the same type as x.

x++ - x must be a scalar lvalue other than floating. x is incremented in place by one, following the rules of addition explained below. The result is an rvalue having the old value and the same type as x.

x-- - x must be a scalar lvalue other than floating. x is decremented in place by one, following the rules of addition explained below. The result is an rvalue having the old value and the same type as x.

~x - x must be an integer. The result is the ones complement of x, having the same type as x.

!x - x must be scalar. The result is an integer 1 if x is zero; otherwise it is an integer 0.

(<a-type>) x - <a-type> is any scalar type declaration with the identifier omitted, e.g., (char *). The result is an rvalue obtained by coercing x to <a-type>. This operator is called a "cast" (see Declarations). Note that a cast to any type smaller than int is taken as (int), while (float) is taken as (double).

sizeof x, sizeof (<a-type>) - The result is an integer rvalue equal to the size in bytes of x or the size in bytes of an object of type <a-type>.

BINARY OPERATORS

There is an implicit "widening" order among the arithmetic types, to wit: char, unsigned char, short, bitfield (if int is equivalent to short), unsigned short, long, bitfield (if int is equivalent to long), unsigned long, float, and double; double is the widest type. In general, the type of a binary operator is the wider of the types of its two operands, the narrower operand being implicitly coerced to match the wider. If arithmetic is not done in place, as in i =+ j, then integer arithmetic is always performed on operands coerced by widening to at least int, and floating arithmetic is always performed on operands widened to double.

Coercions are made up from a series of transformations: A char or short becomes an int of the same value. Sign extension occurs for all int types not declared as unsigned; the latter are widened by zero fill on the left. An int is simply redefined as an unsigned, on twos complement machines at least, with no change in representation. Bitfields are unpacked into unsigned integers. An integer is converted to a double of the same numerical value, while a float is reformatted as a double of the same value, often simply by right fill with zeros.

Assignment may call for a narrowing coercion, which is performed by the following operations: A double is rounded to its nearest arithmetic equivalent in float format; conversion to integer involves discarding any fractional part, then truncating as need be on the left without regard to overflow. Similarly, integers are converted to narrower types by left truncation.

The binary operators are listed in descending order of binding, those with highest precedence first:

x*y - Both operands must be arithmetic (integer or float). The result is the product of x and y, with the type of the wider.

x/y - Both operands must be arithmetic. The result is the quotient of x divided by y, with the type of the wider. Precedence is the same as for *.

x%y - Both operands must be integer. The result is the remainder obtained by dividing x by y, with the type of the wider. Precedence is the same as for *.

x+y - If either operand is of type pointer to ..., the other operand must be of type integer, which is first multiplied by the size in bytes of the type ... then added to the pointer to produce a result of type pointer. Otherwise both operands must be arithmetic; the result is the sum of x and y, with the type of the wider.

x-y - If x is of type pointer to ... and y is of type integer, y is first multiplied by the size in bytes of the type ... then subtracted from the pointer to produce a result of type pointer. Otherwise if x is of type pointer to ... and y is of type pointer to ... and both point to types of the same size in bytes, then x is subtracted from y and the result divided by the size in bytes of ... to produce an integer result. Otherwise both x and y must be arithmetic; the result is y subtracted from x, with the type of the wider. Precedence is the same as for +.

x<<y - Both operands must be integer. The result is x left shifted y places, with the type of x. No promises are made if y is large (compared to the number of bits in x) or negative.

x>>y - Both operands must be integer. The result is x right shifted y places, with the type of x. If the result type is unsigned, no sign extension occurs on the shift; if it is signed, sign extension does occur. No promises are made if y is large or negative. Precedence is the same as for <<.

x<y, x<=y, x>y, x>=y - If either operand is of type pointer to ... and the other is of type integer, the integer is scaled as for addition before the comparison is made. Otherwise if both operands are of type pointer to ... the pointers are compared as unsigned integers. Otherwise both x and y must be arithmetic, and the narrower is widened to match the type of the wider before the comparison is made. The result is an integer 1 if the relation obtains; otherwise it is an integer 0.

x==y, x!=y - The operands are coerced as for <, then compared for equality (==) or inequality (!=). The result is an integer 1 if the relation obtains; otherwise it is an integer 0.

x&y - Both operands must be integer. The result is the bitwise and of x and y, with the type of the wider.

x^y - Both operands must be integer. The result is the bitwise exclusive or of x and y, with the type of the wider.

x|y - Both operands must be integer. The result is the bitwise inclusive or of x and y, with the type of the wider.

x&&y - Both operands must be scalar. If x is zero, the result is taken as integer 0 without evaluating y. Otherwise the result is integer 1 only if both x and y are nonzero.

x||y - Both operands must be scalar. If x is non-zero, the result is taken as integer 1 without evaluating y. Otherwise the result is in-

teger 1 if either x or y is nonzero.

t?x:y - If t, which must be scalar, is nonzero the result is x coerced to the final type; otherwise the result is y coerced to the final type; exactly one of the two operands x and y is evaluated. The final type is pointer to ... if either operand is pointer to ... and the other is integer (the integer is not scaled). Otherwise if both operands are of type pointer to ... the final type is the same as x. Otherwise both operands must be arithmetic and the final type is the wider of the two types.

x=y - Both operands must be scalar, and x must be an lvalue. y is coerced to the type of x and assigned to x. If x is a pointer to ..., y may be a pointer to ... or an integer (the integer is not scaled). Otherwise both operands must be of arithmetic type. The result is an rvalue equal to the value just assigned, having the type of x.

x*=y, x/=y, x%=y, x+=y, x-=y, x<<=y, x>>=y, x&=y, x^=y, x|=y - Each of the operations "x op= y" is equivalent to "x = x op y", except that x is evaluated only once and the type of "x op y" must be that of x, e.g., "x -= y" cannot be used if x and y are both pointers. The operators may also be written =op, for historical reasons, but in this form no whitespace may occur after the =. Precedence is the same as for =.

x,y - Both operands must be scalar. x is evaluated first, then y. The result is the value and type of y. Note that commas in an argument list to a function call are taken as argument separators, not comma operators. Thus f(a,b,c) represents three arguments, while f(a,(b,c)) represents two, the second one being c (after b has been evaluated).

NAME

Constants - compile time arithmetic

FUNCTION

There are four contexts in a C program where expressions must be evaluatable at compile time: the expression part of a #if preprocessor control line, the size of an array in a declaration, the width of a bit-field in a struct declaration, and the label of a case statement. In many other contexts the compiler endeavors to reduce expressions, but this is not mandatory except in the interest of efficiency.

The #if statement evaluates expressions using long integer arithmetic. No assignment operators, casts, or sizeof operators may be used. The result is compared against zero. There is no guarantee that large numbers will be treated the same across systems, due to the variation in operand size, but this variation is expected to be minimal among longs.

Bitfield widths, array sizes and case labels are also computed using long integer arithmetic, but only the integer part (if smaller than long) is retained. Moreover, the sizeof operator is permitted in such expressions.

In all other expressions, the compiler applies a number of reduction rules to simplify expressions at compile time. These include the following (assumed) identities:

```
x * 1 == x
x / 1 == x
x + 0 == x
x - 0 == x
x + y == y + x
(x + y) + z == x + (y + z)
x && true == x
x || false == x
false && x == <nothing>
true || x == <nothing>
(x + y) * z == x * z + y * z
```

plus a number of others. In other words, certain subexpressions may generate no code at all, if the operation is patently redundant. This is worth keeping in mind when writing I/O drivers and other machine dependent routines that are expected to produce useful side effects not obvious to the compiler.

The compiler does not perform common subexpression elimination, however, nor rearrange the order of computation between statements, so that a minimal determinism is assured.

To ensure that compile time reductions occur, on the other hand, it is best to group constant terms within an expression so the compiler does not have to guess the proper rearrangement to bring constants together.

NAME

Preprocessor - lines that begin with #

FUNCTION

A preprocessor is used by the C compiler to perform #define, #include, and other functions signalled by a control character, typically #, before actual compilation begins. A number of options can be specified at preprocess time by the use of flags whose effects are sometimes mentioned below, but which are more fully explained on the manual page for the pp command, described elsewhere in this manual.

Unless -c is specified, /* comments */ are replaced by a single space and any line that ends with a '\' is merged with its successor. If the first non-whitespace character on the resultant line matches either the preprocessor control character or the secondary preprocessor control character, the line is taken as a command to the preprocessor; all other lines are either skipped or expanded as described below.

The following command lines are recognized by the preprocessor:

#define <ident> <defn> - defines the identifier <ident> to be the definition string <defn> that occupies the remainder of the line. Identifiers consist of one or more letters, digits, and underscores '_', where the first character is a non-digit; only the first eight characters are used for comparing identifiers. A sequence of zero or more formal parameters, separated by commas and enclosed in parentheses, may be specified, provided that no whitespace occurs between the identifier and the opening parenthesis. The definition string begins with the first non-whitespace character following the identifier or its parameter list, and ends with the last non-whitespace character on the line. It may be empty. If an identifier is redefined, the new definition is pushed down on top of the older ones.

#undef <ident> - pops one level of definition for <ident>, if any. It is not considered an error to undef an undefined identifier.

#include <fname> - causes the contents of the file specified by <fname> to be lexically included in place of the command line. A filename can be a simple identifier, or an arbitrary string inside (literal) quotes "", or an arbitrary string inside (literal) angle brackets <>. In the last case, a series of standard prefixes is prepended to the filename, normally just "" unless otherwise specified at invocation time, to locate the file in one of several places. Included files may contain further includes.

#ifdef <ident> - commences skipping lines if the identifier <ident> is not defined, else processing proceeds normally for the range of control of the #ifdef. The range of control is up to and including a balancing #endif command. An #else command encountered in the range of control of the #ifdef causes skipping to cease if it was in effect, or normal processing to change to skipping if skipping was not in effect. It is permissible to nest #ifdef and other #if groups; entire groups will be skipped if skipping is in effect at the start.

Preprocessor commands such as #define and #include are not performed while skipping.

#ifndef <ident> - is the same as #ifdef, except that #ifndef commences skipping if the identifier is defined.

#if <expression> - is the same as #ifdef, except that the rest of the line is first expanded, then evaluated as a constant expression; if the expression is zero then skipping commences. An expression may contain parentheses, the unary operators +, -, !, and ~, the binary operators +, -, *, /, %, &, |, ^, <<, >>, <, ==, >, <=, =>, !=, && and ||, and the ternary operator ? :. The definitions and bindings of the operators match those for the C language, subject to the constraint that only integer constants may be used as operands.

#line <num> <fname> - causes the line number used for diagnostic printouts to be set to num and the corresponding filename to be set to fname, if present. If no filename is specified, the filename used for diagnostic printouts is left unchanged. num must be a decimal integer.

- is taken as an innocuous line, if empty. Anything else not recognizably a command causes a diagnostic.

Expansion of non-command lines causes each defined identifier to be replaced by its definition string, then rescanned for further expansion. If the definition has formal parameters, and the next token on the line is a left parenthesis, then a group of actual parameters, inside balanced parentheses, must occur on the line; formal parameters with no corresponding actual parameters are replaced by null strings.

Note that no attempt is made to add whitespace, before or after replacement text, to avoid blurring of token boundaries, just as no parentheses are added to avoid bizarre arithmetic binding in expressions. No expansion occurs within "" or '' strings.

BUGS

Circular definitions such as

```
#define x x
```

cause the preprocessor to blow up.

NAME

Style - rules for writing good C code

FUNCTION

C is too expressive a language to be used without discipline; it can rival APL in opacity or PL/I in variety. The following practices and restraints are recommended for writing good C code:

ORGANIZATION

If a C program totals more than about five hundred lines of code, it should be split into files each no bigger than that. As much as possible, related declarations should be packaged together, with as many of these declared static (LOCAL) as possible. Common definitions and type declarations should be grouped into one or more header files to be #included as need be with each file.

If any use is made of the standard library, its definitions should be included as well, as in:

```
/*  GENERAL HEADER FOR FILE
 * copyright (c) 1981 by Whitesmiths, Ltd.
 */
#include <std.h> #include "defs.h"

#define MAXN    100 /* definitions local to this file */ ...
```

For the contents of <std.h>, including the definitions of LOCAL, etc., see std.h in Section II of this manual.

Header files should be used to contain #defines, typedefs, and declarations that must be known to all source files making up a program. They should not include any initializers, as these would be repeated by multiple inclusion. A good convention is to use all caps for #define'd identifiers, as a warning to the reader that the language is being extended.

It is also good practice to explicitly import all external references needed in each function body by the use of extern (IMPORT) declarations. This not only documents any pathological connections, but also permits functions to be moved freely among files without creating problems.

Within a file, a good discipline is to put all data declarations first, then all function bodies in alphabetical order by function name. Data declarations are typically clumped into logical groups, e.g., all flags, all file control, etc.; an explanatory comment should precede each group of data and each function body, with a single blank line preceding the comment. If the body of a function is sufficiently complex, a good explanatory comment is the half dozen or so lines of pseudocode that best summarize the algorithm. There is seldom a need for additional comments, but if they are used they are best placed to the right of the line being explained, separated by one tab stop from the end of the statement.

If the types provided in std.h are not sufficient to describe all the objects used in a program, then all other types needed should be provided by #defines or typedefs. All declarations should be typed, preferably with these defined types, to improve readability.

RESTRICTIONS

The goto statement should never be used. The only case that can be made for it is to implement a multi-level break, which is not provided in C; but this seldom proves to be a prudent thing to do in the long run. If a function has no goto statements, it has no need for labels.

Other constructs to avoid are the do-while statement, which inevitably evolves into a safer while or for statement, and the continue statement, which is typically just a shoddy way of avoiding the proper use of else clauses inside loops.

More than five levels of indenting (see FORMATTING below) is a sure sign that a subfunction should be split out, as is the case with a function body that goes much over a page of listing or requires more than half a dozen local variables. Naturally there are exceptions to all these guidelines, but they are just that -- exceptions.

The use of the quasi-Boolean operators &&, ||, !, etc. to produce integer ones and zeros should not be indulged to perform cute arithmetic, as in

```
sum[i] = a[i] + b[i] + (10 <= sum[i - 1]);
```

Such practices, if used, should be commented, as should most tricky bit manipulations using &, | and ^.

Elaborate expressions involving ? and :, particularly multiple instances thereof, are often hard to read. Parenthesizing helps, but excessive use of parentheses is just as bad.

If the relational operators > and >= are avoided, then compound tests can be made to read like intervals along the number axis, as in

```
if ('0' <= c && c <= '9')
    ...
```

which is demonstrably true when c is a digit.

FORMATTING

While it may seem a trivial matter, the formatting of a C program can make all the difference between correct comprehension and repeated error. To get maximum benefit from support tools such as editors and cross referencers, one should apply formatting rules rigorously. The following high-handed dicta have proved their worth many times over:

Each external declaration should begin (with optional storage class and mandatory type) at the beginning of a line, immediately following its explanatory comment. All defining material, data initializers or function definitions, should be indented at least one tab stop, plus additional tab stops to reflect substructure. Tabs should be set uniformly every four to eight columns.

A function body, for instance, always looks like:

```
TYPE name(arg1, arg2, arg3)
    TYPE1 arg1;
    TYPE2 arg2, arg3;
    {
    <local declarations>

    <statements>
    }
```

This example assumes that arg2 and arg3 have the same type. If no <local declarations> are present, there is no empty line before <statements>.

<local declarations> consists of first extern (IMPORT), then register (FAST), then auto (no storage class specifier), then static (INTERN) declarations; these are sorted alphabetically by type within storage class and alphabetically by name within type. Comma separated lists may be used, so long as there are no initializers; an initialized variable should stand alone with its initializer. FAST and auto storage should be initialized at declaration time only if the value is not to change.

<statements> are formatted to emphasize control structure, according to the following basic patterns:

```
if (test)
    <statement>

if (test)
    {
    <statement>
    <statement>
    ...
    }
else
    <statement>

if (test1)
    <statement>
else if (test2)
    <statement>
else if (test3)
    ...
else
    <default statement>
```

```
switch (value)
    {
case A:
case B:
    <statement>
    ...
    break;
case C:
    <statement>
    ...
    break;
default:
    <statement>
    ...
    }

while (test)
    <statement>

for (init; test; incr)
    <statement>

for (; test; incr)
    <statement>

for (init; ; )
    <statement>

FOREVER
    <statement>

return (expr);
```

Note that, with the explicit exception of the else-if chain, each subordinate <statement> is indented one tab stop further to the right than its controlling statement. Without this rule, an else-if chain would be written:

```
if (test 1)
    <statement>
else
    if (test 2)
        <statement>
    else
        if (test 3)
            <statement>
        ...
            else
                <default statement>
```

Within statements, there should be no empty lines, nor any tabs or multiple spaces imbedded in a line. Each keyword should be followed by a single space, and each binary operator should have a single space on each side. No spaces should separate unary or addressing operators from their

operands. A possible exception to the operator rule is a composite constant, such as (GREEN|BLUE).

Parentheses should be used whenever there is a hint of ambiguity. Note in particular that & and | mix poorly with the relational operators, that the assigning operators are weaker than && and ||, and that << and >> are impossible to guess right. The worst offender is

```
if ((a & 030) != 030)
    ...
```

which does entirely the wrong thing if the parentheses are omitted.

If an expression is too long to fit on a line (of no more than 80 characters) it should be continued on the next line, indented one tab stop further than its start. A good rule is to continue only inside parentheses, or with a trailing operator on the preceding line, so that displaced fragments are more certain to cause diagnostics.

EXAMPLE

A typical library function looks like this:

```
#include <std.h>

/*  CONVERT LONG TO BUFFER
 *  copyright (c) 1978 by Whitesmiths, Ltd.
 */
BYTES ltob(is, ln, base)
    FAST TEXT *is;
    LONG ln;
    BYTES base;
    {
    FAST TEXT *s;
    ULONG lb;

    s = is;
    if (ln < 0 && base == 0)
        {
        ln = -ln;
        *s++ = '-';
        }
    if (base == 0)
        base = 10;
    else if (base < 0)
        base = -base;
    lb = base;
    if (ln < 0 || lb <= ln)
        s =+ ltob(s, ln / lb, base);
    *s = ln % lb + '0';
    if ('9' < *s)
        *s =+ ('a' - ('9' + 1));
    return (s - is + 1);
    }
```

NAME

Portability - writing portable code

FUNCTION

Writing highly portable C code is remarkably easy, most of the time. When machine dependencies creep in, however, they can be extremely difficult to dig out; and trying to keep them out of new code can often lead the programmer to paranoid extremes. Herewith a set of rules to follow that eliminate nearly all the ghastlies before they bite.

First and foremost, use the portable library. Pretend the C Interface Manuals don't exist, and fall out of love with the endearing peculiarities of your current host system. It will change.

If your program processes text files, assume that carriage returns, NULs, and other funny characters that don't print may disappear if written out and read back later. Assume that lseek will fail, even when it obviously should work on your system. Text lines may be as long as 512 characters, counting the terminating newline; but then they should never be longer than that. It is usually best not to depend on the presence of that trailing newline, if at all possible, in case the program is fed an unusually long line or a truncated last line.

If your program tries to process STDIN, STDOUT, or STDERR as a binary file, assume that the data will be corrupted; binary files must be opened by name on most systems. The third argument to open and create should always be present and for binary files should always be non-zero. A third argument of 1 is always acceptable, and will not lead to storage inefficiencies in the target file. Sadly, the set of functions fopen, fcreate, getfiles, etc. were frozen before the text/binary dichotomy became apparent, so they work smoothly only on text files; getbfiles is a later addition. Performing a binary open, followed by a finit with third argument READ or BWRITE, does make the buffered I/O mechanism safely available for sequential binary I/O, however.

Many operating systems will pad a binary file with NULs, which are hard to detect in the interface code. Consequently, any program that does binary reads must be prepared to deal with trailing NULs. Treating NUL as end of file is best, whenever possible. Another aspect of this problem is that the length of a binary file is poorly determined on many systems; consequently the ability to lseek relative to the end of a file is no longer supported in the portable specification.

The order in which bytes are stored for encoded arithmetic types varies all over the map. A long integer, '3210' for instance, can read out as '3210', '0123', or '2301' on three popular computers, where '0' is the least significant byte. The best rule is never to write a multi-byte datum, unless it is an array of chars or unless it is clearly understood that the resultant file will always be read into an identically declared datum on the same machine. Look for sizeof operators not connected with alloc calls; they are a sign of potential trouble. The library functions lstoi and itols are provided to ease transmission of two-byte integers among various implementations, while lstol and ltols provide a similar mechanism for four-byte quantities.

The portable specification says filenames should be no more elaborate than "xxxxxx.yy". Believe it. Better yet, use uname to build temp file names and avoid wiring any other file names into code. That's what the command line is for, and what getfiles is designed to help with. In the same vein, external identifiers should be chosen for the worst case, i.e., a system where only six characters in one case are retained. It is possible to have the compiler check the identifiers for conformance to this or similar constraints.

Declarations are designed to ease portability among machines with different data formats, but they must be used properly to do so. C is very tolerant of silly declarations for such things as: arguments to a function, values returned by a function, data held in registers, and casts. Since all of these items are essentially rvalues, they are never smaller than an int, or a double if floating point, no matter how they are declared. The difference matters only when some lvalueness creeps in, as when assigning to a register or argument, or taking the address of an argument.

Never assume that assigning to one of these creatures, or applying a cast such as (char), will perform any sort of truncation; it won't, at least not below integer. And while it is nice to be able to declare an argument to be char for the sake of documenting its likely range, it is easy to forget when taking its address that what you want is a pointer to int, not a pointer to char. Storing a char in part of an int may work a little bit right on some computers, but it will surely cause trouble sooner or later. Look for address of (unary &) operators applied to arguments, and expect problems.

The standard header provides a constellation of defined types to stylize proper usage. If you know how big you want a datum, regardless of what machine the program is run on, use the aliases for {char, short, long}: i.e., {TINY, COUNT, LONG} or the unsigned versions {UTINY, UCOUNT, ULONG}. If something must hold a pointer, declare it as such by all means; if it must span most or all of the address space of the target machine, as a subscript for example, declare it as unsigned int, or BYTES. Remember that case switch values are ints; hence, only short values are portable for case labels.

There are two other important malleable types besides BYTES. ARGINT is used when talking about an argument that is known to have been widened to an integer; it should serve as a red flag that something special is happening. TEXT, on the other hand, is used heavily to ensure efficient code; it is declared in the header as either char or unsigned char, depending on which is more easily handled by the target machine. When using TEXT variables, the programmer must be careful to mask possible sign extensions, using BYTMASK, should other than ASCII characters or small positive integers be stored in them. This usage parallels the standard uncertainty of char variables in other implementations of C.

To ensure maximum portability between 16 and 32-bit pointer machines, the best mind set is that an int is not equal to a short and it is not equal to a long, but it can and will be equal to either some time or other.

Avoid writing constants of fixed size, such as 0177777; instead write stretchable forms such as ~0 for the above. If you know a constant must be long on any machine, be sure to force it long; ~0 can be different from ~0L.

To end on a positive note, there are some things you can depend on across implementations. There will always be at least three registers available, for instance. These can hold anything up to an unsigned int and almost invariably offer substantial code space and execution time benefits if used for the most important variables. Pointers benefit particularly from being placed in registers. And the assigning operators, such as =+ or +=, frequently lead to better code production. This is particularly true for the smaller data types such as char, since C is obliged to compute (c1 + c2) to int precision, but may do (c1 =+ c2) as a char operation. And it is possible to parameterize code efficiently by writing expressions like:

```
if (sizeof (int) == sizeof (short) && <short test> ||
    sizeof (int) == sizeof (long) && <long test>)
```

Only <short test> or <long test> will actually be generated as runtime code; the rest is optimized out by the compiler.

NAME

Differences - comparative anatomy

FUNCTION

The definitive standard for C is Appendix A of Kernighan & Ritchie, as explained in the Introduction to this section. This implementation hews closely to that standard, save for minor changes in emphasis. There are also several available implementations of C that differ in more important respects. Herewith a summary of the things to look out for.

THE STANDARD

- The major deviation is that this compiler requires each external declaration to be explicitly initialized exactly once among all the files that comprise a C program; the standard permits external declarations to remain uninitialized.

- This implementation includes the types unsigned [char short long], which are not yet in the standard.

- Backslash is used to continue strings in the standard; its use is generalized here.

- Character constants with more than one character are defined here, but not in the standard.

- All struct and union tags share the name space of all members of struct and union, in the standard; each kind of tag has its own name space here.

- This implementation permits, as an option, separate name spaces for each struct or union and much more rigorous checking of . and -> operators.

- A union may be initialized in this implementation.

- A preprocessor macro invocation, e.g., swap(a, b), must be written all on one line in this implementation.

- The sizeof operator is explicitly disallowed in #if expressions, in this implementation.

UNIX/V6

- Not implemented in the UNIX/V6 compiler are: bitfields, short integers, unsigned integers, long integers, casts, unions, #if, #line, operators of the form op=, static external declarations (local to a file), or register arguments.

- UNIX/V6 initializes a structure as if it were an array of integers.

UNIX/V7

- Bitfields may not be initialized, in at least one of the UNIX/V7 compilers.
- Casts of the form (char) or (short) may actually truncate a value in the UNIX/V7 implementation; they have no effect on ints in this implementation.
- The address of an array cannot be taken in UNIX/V7 C.
- Enumerated types, structure assignment, and functions returning structs have been added in UNIX/V7 C.

SYSTEM DEPENDENCIES

Since this implementation produces assembler code for the target system, there is some variation in naming caused by assembler limitations. There may be as few as six, but never more than eight significant characters in external identifiers; often only one case of letters is significant. For specific differences, see the C Interface Manual for the relevant target machine.

NAME

Diagnostics - compiler complaints

FUNCTION

The first two passes of the compiler produce all user diagnostics, the initial (preprocessor) pass dealing with # control lines and lexical analysis, the next with everything else. If a pass produces diagnostics, later passes should not be run. Any compiler message containing an exclamation mark '!' or the word "panic" indicates problems with the compiler per se (they should "never happen") and hence should be reported to the maintainers. Here is a summary of the diagnostics that can be produced by erroneous C programs:

PREPROCESSOR DIAGNOSTICS:

bad #define - illegal define.
bad #define arguments - cannot parse #define line.
bad #include - illegal include.
bad #line - illegal #line.
bad #undef - illegal undef.
bad #xxx - unrecognizable # control line.
bad macro arguments - cannot parse macro definitions.
bad output file - cannot create output file.
can't #include xxx - cannot open file specified in #include.
can't open xxx - cannot open file specified as pp argument.
illegal #if expression
illegal #if syntax
illegal ? : in #if
illegal character: x - not a recognizable token in C.
illegal constant xxx - not a recognizable numeric form.
illegal float constant
illegal number in #if
illegal operator in #if
illegal unary op in #if
misplaced #xxx - preprocessor control line out of place.
missing #endif - unbalanced #if, #ifdef, or #ifndef.
missing) in #if
missing */ - unbalanced /* comment.
string too long - more than 512 characters.
truncated line - more than 512 characters.
unbalanced x - x is a delimiter: ', ", (, <, or {.

PASS 1 DIAGNOSTICS:

arithmetic type required - integer or floating.
array size unknown
bad (declaration) - stuff inside () unrecognizable.
bad external syntax
bad field width - negative or larger than word size.
bad output file - cannot create output file.
cannot initialize
constant required

declaration too complex - more than 5 modifiers.
external name conflict - when truncated for output
function redefined
function required - arguments declared, but no function body.
function size undefined
identifier not allowed
illegal %
illegal &
illegal /
illegal =+
illegal assignment
illegal bitfield
illegal break
illegal case
illegal cast
illegal comparison
illegal continue
illegal default
illegal double initializer
illegal field
illegal field initializer
illegal indirection - unary "*" operator.
illegal integer initializer
illegal member
illegal operand type
illegal pointer initializer
illegal postop
illegal return type
illegal selection
illegal statement
illegal storage class
illegal structure reference
illegal type modifier
illegal unsigned compare
illegal use of typedef
incomplete declaration
integer type required
lvalue required - see Expressions.
member conflict
member redefined
missing argument
missing expression
missing goto label
missing label
missing member name - identifier must follow . or ->.
no structure definition
not an argument
redeclared argument
redeclared external
redeclared local
redeclared typedef
redefined label
redefined tag
string initializer too long

```
structure size unknown
too many initializers
undeclared x - no declaration of x was encountered
undefined static
unexpected EOF
union size unknown
unknown member
useless expression - result unused, no side effect.
```