Rationale for International Standard— Programming Languages— C

Revision 2
20 October 1999

0. Introduction

This Rationale summarizes the deliberations of NCITS J11 (formerly X3J11) and SC22 WG14, respectively the ANSI Technical Committee and ISO/IEC JTC 1 Working Group, charged with revising the International Standard for the C programming language; and it retains much of the text of the Rationale for the original ANSI Standard (ANSI X3.159-1989, the so-called "C89"). This document has been published along with the draft Standard to assist the process of formal public review.

There have been several changes to the Standard already. C89 was quickly adopted as an International Standard (ISO/IEC 9899:1990, commonly called "C90"), with changes to clause and subclause numbering to conform to ISO practices. Since then, there have been two Technical Corrigenda and one Amendment, AMD1; and those three documents, together with C90 itself, compose the current International Standard, ("C95"). The draft Standard is often called "C9X."

J11 represents a cross-section of the C community in the United States: it consists of about twenty or thirty members representing hardware manufacturers, vendors of compilers and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. WG14's participants are representatives of national standards bodies such as AFNOR, ANSI, BSI, DIN and DS. In this Rationale, the unqualified "Committee" refers to J11 and WG14 working together to create C9X.

Upon publication of the new Standard, the primary role of the Committee will be to offer interpretations of the Standard. It will consider and respond to all correspondence it receives.

The Committee's overall goal was to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.

The original X3J11 charter clearly mandated *codifying common existing practice*, and the C89 Committee held fast to precedent wherever that was clear and unambiguous. The vast majority of the language defined by C89 was precisely the same as defined in Appendix A of the first edition of *The C Programming Language* by Brian Kernighan and Dennis Ritchie, and as was implemented in almost all C translators of the time. (This document is hereinafter referred to as K&R.)

K&R was not the only source of "existing practice." Much work had been done over the years to improve the C language by addressing its weaknesses, and the C89 Committee formalized enhancements of proven value which had become part of the various dialects of C. This practice has continued in the present Committee.

Existing practice, however, has not always been consistent. Various dialects of C have approached problems in different and sometimes diametrically opposed ways. This divergence has happened for several reasons. First, K&R, which once served as the language specification for almost all C translators, is imprecise in some areas (thereby allowing divergent interpretations), and it does not

address some issues (such as a complete specification of a library) important for code portability. Second, as the language has matured over the years, various extensions have been added in different dialects to address limitations and weaknesses of the language; but these extensions have not been consistent across dialects.

One of the C89 Committee's goals was to consider such areas of divergence and to establish a set of clear, unambiguous rules consistent with the rest of the language. This effort included the consideration of extensions made in various C dialects, the specification of a complete set of required library functions, and the development of a complete, correct syntax for C.

Much of the Committee's work has always been in large part a balancing act. The C89 Committee tried to improve portability while retaining the definition of certain features of C as machine-dependent, it attempted to incorporate valuable new ideas without disrupting the basic structure and fabric of the language, and it tried to develop a clear and consistent language without invalidating existing programs. All of the goals were important and each decision was weighed in the light of sometimes contradictory requirements in an attempt to reach a workable compromise.

In specifying a standard language, the C89 Committee used several principles which continue to guide our deliberations today. The most important of these are:

Existing code is important, existing implementations are not. A large body of C code exists of considerable commercial value. Every attempt has been made to ensure that the bulk of this code will be acceptable to any implementation conforming to the Standard. The C89 Committee did not want to force most programmers to modify their C programs just to have them accepted by a conforming translator.

On the other hand, no one implementation was held up as the exemplar by which to define C. It was assumed that all existing implementations must change somewhat to conform to the Standard.

C code can be portable. Although the C language was originally born with the UNIX operating system on the DEC PDP-11, it has since been implemented on a wide variety of computers and operating systems. It has also seen considerable use in cross-compilation of code for embedded systems to be executed in a free-standing environment. The C89 Committee attempted to specify the language and the library to be as widely implementable as possible, while recognizing that a system must meet certain minimum criteria to be considered a viable host or target for the language.

C code can be non-portable. Although it strove to give programmers the opportunity to write truly portable programs, the C89 Committee did not want to *force* programmers into writing portably, to preclude the use of C as a "high-level assembler:" the ability to write machine-specific code is one of the strengths of C. It is this principle which largely motivates drawing the distinction between *strictly conforming program* and *conforming program* (§4).

Avoid "quiet changes." Any change to widespread practice altering the meaning of existing code causes problems. Changes that cause code to be so ill-formed as to require diagnostic messages are at least easy to detect. As much as seemed possible consistent with its other goals, the C89 Committee avoided changes that quietly alter one valid program to another with different semantics, that cause a working program to work differently without notice. In important places

where this principle is violated, both the C89 Rationale and this Rationale point out a QUIET CHANGE.

A standard is a treaty between implementor and programmer. Some numerical limits were added to the Standard to give both implementors and programmers a better understanding of what must be provided by an implementation, of what can be expected and depended upon to exist. These limits were, and still are, presented as *minimum maxima* (that is, lower limits placed on the values of upper limits specified by an implementation) with the understanding that any implementor is at liberty to provide higher limits than the Standard mandates. Any program that takes advantage of these more tolerant limits is not strictly conforming, however, since other implementations are at liberty to enforce the mandated limits.

Keep the spirit of C. The C89 Committee kept as a major goal to preserve the traditional *spirit* of C. There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles upon which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like

- *Trust the programmer.*
- *Don't prevent the programmer from doing what needs to be done.*
- *Keep the language small and simple.*
- *Provide only one way to do an operation.*
- *Make it fast, even if it is not guaranteed to be portable.*

The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined to be how the target machine's hardware does it rather than by a general abstract rule. An example of this willingness to live with *what the machine does* can be seen in the rules that govern the widening of **`char`** objects for use in expressions: whether the values of **`char`** objects widen to signed or unsigned quantities typically depends on which byte operation is more efficient on the target machine.

One of the goals of the C89 Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases the C89 Committee introduced features to improve the possible efficiency of the generated code; for instance, floating point operations may be performed in single-precision if both operands are **`float`** rather than **`double`**.

At the WG14 meeting in Tokyo, Japan, in July 1994, the original principles were re-endorsed and the following new ones were added:

Support international programming. During the initial standardization process, support for internationalization was something of an afterthought. Now that internationalization has become an important topic, it should have equal visibility. As a result, all revision proposals shall be reviewed with regard to their impact on internationalization as well as for other technical merit.

Codify existing practice to address evident deficiencies. Only those concepts that have some prior art should be accepted. (Prior art may come from implementations of languages other than C.)

Unless some proposed new feature addresses an evident deficiency that is actually felt by more than a few C programmers, *no new inventions should be entertained*.

Minimize incompatibilities with C90 (ISO/IEC 9899:1990). It should be possible for existing C implementations to gradually migrate to future conformance, rather than requiring a replacement of the environment. It should also be possible for the vast majority of existing conforming programs to run unchanged.

Minimize incompatibilities with C++. The Committee recognizes the need for a clear and defensible plan for addressing the compatibility issue with C++. The Committee endorses the principle of maintaining the largest common subset clearly and from the outset. Such a principle should satisfy the requirement to maximize overlap of the languages while maintaining a distinction between them and allowing them to evolve separately.

The Committee is content to let C++ be the big and ambitious language. While some features of C++ may well be embraced, it is not the Committee's intention that C become C++.

Maintain conceptual simplicity. The Committee prefers an economy of concepts that do the job. Members should identify the issues and prescribe the minimal amount of machinery that will solve the problems. The Committee recognizes the importance of being able to describe and teach new concepts in a straightforward and concise manner.

During the revision process, it was important to consider the following observations:

- Regarding the 11 principles, there is a tradeoff between them—none is absolute. However, the more the Committee deviates from them, the more rationale will be needed to explain the deviation.
- There had been a very positive reception of the standard from both the user and vendor communities.
- The standard was not considered to be broken. Rather, the revision was needed to track emerging and/or changing technologies and internationalization requirements.
- Most users of C view it as a general-purpose high-level language. While higher level constructs can be added, they should be done so only if they don't contradict the basic principles.
- There are a good number of useful suggestions to be found from the public comments and defect report processing.

Areas to which the Committee looked when revising the C Standard included:

- Incorporate AMD1.
- Incorporate all Technical Corrigenda and records of response.

- Current defect reports.
- Future directions in current standard.
- Features currently labeled obsolescent.
- Cross-language standards groups work.
- Requirements resulting from JTC 1/SC 2 (character sets).
- The evolution of C++.
- The evolution of other languages, particularly with regard to interlanguage communication issues.
- Other papers and proposals from member delegations, such as the numerical extensions Technical Report which was proposed by J11.
- Other comments from the public at large.
- Other prior art.

This Rationale focuses primarily on additions, clarifications, and changes made to the C language. It is *not* a rationale for the C language as a whole: the C89 Committee was charged with codifying an existing language, not designing a new one. No attempt is made in this Rationale to defend the pre-existing syntax of the language, such as the syntax of declarations or the binding of operators. The Standard is contrived as carefully as possible to permit a broad range of implementations, from direct interpreters to highly optimizing compilers with separate linkers, from ROM-based embedded microcomputers to multi-user multi-processing host systems. A certain amount of specialized terminology has therefore been chosen to minimize the bias toward compiler implementations shown in K&R.

This Rationale discusses some language or library features which were *not* adopted into the Standard. These are usually features which are popular in some C implementations, so that a user of those implementations might question why they do not appear in the Standard.

0.1 Organization of the document

This Rationale is organized to parallel the Standard as closely as possible to facilitate finding relevant discussions. Some subclauses of the Standard are absent from this Rationale: this indicates that the Committee thought no special comment was necessary. Where a given discussion touches on several areas, attempts have been made to include cross references within the text. Such references, unless they specify the Standard or the Rationale, are deliberately ambiguous.

This document has one appendix called MSE which brings together information on the Multibyte Support Extensions (MSE) that were added to C90 by AMD1. This is essentially the Rationale for

AMD1; and it was kept largely unchanged because it was thought that it would be clearer to have the MSE rationale in one place, as opposed to scattered throughout the document.

Just as the Standard proper excludes all examples, footnotes, references, and informative annexes, *this Rationale is not part of the Standard*. The C language is defined by the Standard alone. If any part of this Rationale is not in accord with that definition, the Committee would very much like to be so informed.

1. Scope

2. Normative References

3. Terms and definitions

The definitions of *object*, *bit*, *byte*, and *alignment* reflect a strong consensus, reached after considerable discussion, about the fundamental nature of the memory organization of a C environment:

- All objects in C must be representable as a contiguous sequence of bytes, each of which is at least 8 bits wide.
- A **`char`** whether signed or unsigned, occupies exactly one byte.

(Thus, for instance, on a machine with 36-bit *words*, a *byte* can be defined to consist of 9, 12, 18, or 36 bits, these numbers being all the exact divisors of 36 which are not less than 8.) These strictures codify the widespread presumption that any object can be treated as an array of characters, the size of which is given by the **`sizeof`** operator with that object's type as its operand.

These definitions do not preclude "holes" in **`struct`** objects. Such holes are in fact often mandated by alignment and packing requirements. The holes simply do not participate in representing the composite value of an object.

The definition of *object* does not employ the notion of type. Thus an object has no type in and of itself. However, since an object may only be designated by an *lvalue* (see §6.3.2.1), the phrase "the type of an object" is taken to mean, here and in the Standard, "the type of the lvalue designating this object," and "the value of an object" means "the contents of the object interpreted as a value of the type of the lvalue designating the object."

The concepts of *multibyte character*, *wide character*, *universal character*, and *extended character* have been added to C to support very large character sets (see §5.2.1 and §MSE.1).

The terms *unspecified behavior*, *undefined behavior*, and *implementation-defined behavior* are used

to categorize the result of writing programs whose properties the Standard does not, or cannot, completely describe. The goal of adopting this categorization is to allow a certain variety among implementations which permits *quality of implementation* to be an active force in the marketplace as well as to allow certain popular extensions, without removing the cachet of *conformance to the Standard*. Informative Annex G of the Standard catalogs those behaviors which fall into one of these three categories.

Unspecified behavior gives the implementor some latitude in translating programs. This latitude does not extend as far as failing to translate the program, however, because all possible behaviors are "correct" in the sense that they don't cause undefined behavior in *any* implementation.

Undefined behavior gives the implementor license not to catch certain program errors that are difficult to diagnose. It also identifies areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior.

Implementation-defined behavior gives an implementor the freedom to choose the appropriate approach, but requires that this choice be explained to the user. Behaviors designated as implementation-defined are generally those in which a user could make meaningful coding decisions based on the implementation's definition. Implementors should bear in mind this criterion when deciding how extensive an implementation definition ought to be. As with unspecified behavior, simply failing to translate the source containing the implementation-defined behavior is not an adequate response.

A new feature of C9X: While responding to a Defect Report filed against C89, the Committee came to realize that the term, "implementation-defined," was sometimes being used in the sense of "implementation must document" when dealing with locales. The term, "locale-specific behavior," already in C89, was then used extensively in C95 to distinguish those properties of locales which can appear in a strictly conforming program. Because the presence or absence of a specific locale is, with two exceptions, implementation-defined, some users of the Standard were confused as to whether locales could be used at all in strictly conforming programs.

A successful call to **`setlocale`** has side effects, known informally as "setting the contents of the current locale," which can alter the subsequent output of the program. A program whose output is altered only by such side effects—for example, because the decimal point character has changed—is still strictly conforming.

A program whose output is affected by the value returned by a call to **`setlocale`** might not be strictly conforming. If the only way in which the result affects the final output is by determining, directly or indirectly, whether to make another call to **`setlocale`**, then the program remains strictly conforming; but if the result affects the output in some other way, then it does not.

4. Conformance

The three-fold definition of conformance is used to broaden the population of conforming programs

and distinguish between conforming programs using a single implementation and portable conforming programs.

A *strictly conforming program* is another term for a maximally portable program. The goal is to give the programmer a *fighting chance* to make powerful C programs that are also highly portable, without seeming to demean perfectly useful C programs that happen not to be portable, thus the adverb *strictly*.

By defining conforming implementations in terms of the programs they accept, the Standard leaves open the door for a broad class of extensions as part of a conforming implementation. By defining both *conforming hosted* and *conforming freestanding* implementations, the Standard recognizes the use of C to write such programs as operating systems and ROM-based applications, as well as more conventional hosted applications. Beyond this two-level scheme, no additional subsetting is defined for C, since the C89 Committee felt strongly that too many levels dilutes the effectiveness of a standard.

Conforming program is thus the most tolerant of all categories, since only one conforming implementation need accept it. The primary limitation on this license is §5.1.1.3.

Diverse parts of the Standard comprise the "treaty" between programmers and implementors regarding various name spaces: if the programmer follows the rules of the Standard the implementation will not impose any further restrictions or surprises:

- A strictly conforming program can use only a restricted subset of the identifiers that begin with underscore (§7.1.3). Identifiers and keywords are distinct (§6.4.1). Otherwise, programmers can use whatever internal names they wish; a conforming implementation is guaranteed not to use conflicting names of the form reserved for the programmer. (Note, however, the class of identifiers which are identified in §7.26 as possible future library names.)

- The external functions defined in, or called within, a portable program can be named whatever the programmer wishes, as long as these names are distinct from the external names defined by the Standard library (§7). External names in a maximally portable program must be distinct within the first 31 characters (in C95, the first 6 characters mapped into one case) (see §5.2.4.1 and §6.4.2).

- A maximally portable program cannot, of course, assume any language keywords other than those defined in the Standard.

- Each function called within a maximally portable program must either be defined within some source file of the program or else be a function in the Standard library.

One proposal long entertained by the C89 Committee was to mandate that each implementation have a translation-time switch for turning off extensions and making a pure Standard-conforming implementation. It was pointed out, however, that virtually every translation-time switch setting effectively creates a different "implementation," however close may be the effect of translating with two different switch settings. Whether an implementor chooses to offer a family of conforming

implementations, or to offer an assortment of non-conforming implementations along with one that conforms, was not the business of the C89 Committee to mandate. The Standard therefore confines itself to describing conformance, and merely suggests areas where extensions will not compromise conformance.

Other proposals rejected more quickly were to provide a validation suite, and to provide the source code for an acceptable library. Both were recognized to be major undertakings, and both were seen to compromise the integrity of the Standard by giving concrete examples that might bear more weight than the Standard itself. The potential legal implications were also a concern.

Standardization of such tools as program consistency checkers and symbolic debuggers lies outside the mandate of the C89 Committee. However, the C89 Committee took pains to allow such programs to work with conforming programs and implementations.

5. Environment

Because C has seen widespread use as a cross-compiled cross-compilation language, a clear distinction must be made between translation and execution environments. The C89 preprocessor, for instance, is permitted to evaluate the expression in a **`#if`** directive using the long integer or unsigned long integer arithmetic native to the translation environment: these integers must comprise at least 32 bits, but need not match the number of bits in the execution environment. In C9X, this arithmetic must be done in **`intmax_t`** or **`uintmax_t`**, which must comprise at least 64 bits and must match the execution environment. Other translation time arithmetic, however, such as type casting and floating point arithmetic, must more closely model the execution environment regardless of translation environment.

5.1 Conceptual models

The *as if* principle is invoked repeatedly in this Rationale. The C89 Committee found that describing various aspects of the C language, library, and environment in terms of concrete models best serves discussion and presentation. Every attempt has been made to craft the models so that implementors are constrained only insofar as they must bring about the same result, *as if* they had implemented the presentation model; often enough the clearest model would make for the worst implementation.

5.1.1 Translation environment

5.1.1.1 Program structure

The terms *source file, external linkage, linked, libraries,* and *executable program* all imply a conventional compiler/linker combination. All of these concepts have shaped the semantics of C, however, and are inescapable even in an interpreted environment. Thus, while implementations are not required to support separate compilation and linking with libraries, in some ways they must behave *as if* they do.

5.1.1.2 Translation phases

Perhaps the greatest undesirable diversity among pre-C89 implementations can be found in preprocessing. Admittedly a distinct and primitive language superimposed upon C, the preprocessing commands accreted over time, with little central direction, and with even less precision in their documentation. This evolution has resulted in a variety of local features, each with its ardent adherents: K&R offers little clear basis for choosing one over the other.

The consensus of the C89 Committee is that preprocessing should be simple and *overt*, that it should sacrifice power for clarity. For instance, the macro invocation **`f(a,b)`** should assuredly have two actual arguments, even if **`b`** expands to **`c,d`**; and the formal definition of **`f`** must call for exactly two arguments. Above all, the preprocessing sub-language should be specified precisely enough to minimize or eliminate dialect formation. To clarify the nature of preprocessing, the translation from source text to tokens is spelled out as a number of separate phases. The separate phases need not actually be present in the translator, but the net effect must be *as if* they were. The phases need not be performed in a separate preprocessor, although the definition certainly permits this common practice. Since the preprocessor need not know anything about the specific properties of the target, a machine-independent implementation is permissible. The C89 Committee deemed that it was outside the scope of its mandate to require that the output of the preprocessing phases be available as a separate translator output file.

The *phases of translation* are spelled out to resolve questions raised about the precedence of different parses. Can a **`#define`** begin a comment? (No.) Is backslash/new-line permitted within a trigraph? (No.) Must a comment be contained within one **`#include`** file? (Yes.) And so on. The Rationale on preprocessing (§6.10) discusses the reasons for many of the decisions that shaped the specification of the phases of translation.

A backslash immediately before a newline has long been used to continue string literals, as well as preprocessing command lines. In the interest of easing machine generation of C, and of transporting code to machines with restrictive physical line lengths, the C89 Committee generalized this mechanism to permit *any* token to be continued by interposing a backslash/newline sequence.

In translation phase 4, the syntactic category preprocessing-file applies to each included file separately from the file it is included into. Thus an included file cannot contain, for example, unbalanced **`#else`** or **`#elif`** directives.

5.1.1.3 Diagnostics

By mandating some form of diagnostic message for any program containing a syntax error or constraint violation, the Standard performs two important services. First, it gives teeth to the concept of *erroneous program*, since a conforming implementation must distinguish such a program from a valid one. Second, it severely constrains the nature of extensions permissible to a conforming implementation.

The Standard says nothing about the nature of the diagnostic message, which could simply be

"syntax error", with no hint of where the error occurs. (An implementation must, of course, describe what translator output constitutes a diagnostic message, so that the user can recognize it as such.) The C89 Committee ultimately decided that any diagnostic activity beyond this level is an issue of *quality of implementation*, and that market forces would encourage more useful diagnostics. Nevertheless, the C89 Committee felt that at least some significant class of errors must be diagnosed, and the class specified should be recognizable by all translators.

The Standard does not forbid extensions provided that they do not invalidate strictly conforming programs, and the translator must allow extensions to be disabled as discussed in Rationale §4. Otherwise, extensions to a conforming implementation lie in such realms as defining semantics for syntax to which no semantics is ascribed by the Standard, or giving meaning to *undefined behavior*.

5.1.2 Execution environments

The definition of *program startup* in the Standard is designed to permit initialization of static storage by executable code, as well as by data translated into the program image.

5.1.2.1 Freestanding environment

As little as possible is said about freestanding environments, since little is served by constraining them.

5.1.2.2 Hosted environment

The properties required of a hosted environment are spelled out in a fair amount of detail in order to give programmers a reasonable chance of writing programs which are portable among such environments.

5.1.2.2.1 Program startup

The behavior of the arguments to **`main`**, and of the interaction of **`exit`**, **`main`** and **`atexit`** (see §7.20.4.2) has been codified to curb some unwanted variety in the representation of **`argv`** strings, and in the meaning of values returned by **`main`**.

The specification of **`argc`** and **`argv`** as arguments to **`main`** recognizes extensive prior practice. **`argv[argc]`** is required to be a null pointer to provide a redundant check for the end of the list, also on the basis of common practice.

`main` is the only function that may portably be declared either with zero or two arguments. (The number of other functions' arguments must match exactly between invocation and definition.) This special case simply recognizes the widespread practice of leaving off the arguments to **`main`** when the program does not access the program argument strings. While many implementations support more than two arguments to **`main`**, such practice is neither blessed nor forbidden by the Standard; a program that defines **`main`** with three arguments is not *strictly conforming* (see §K.5.1.).

Command line I/O redirection is not mandated by the Standard, as this was deemed to be a feature

of the underlying operating system rather than the C language.

5.1.2.3 Program execution

Because C expressions can contain side effects, issues of *sequencing* are important in expression evaluation (see §6.5 and Annexes C and D). Most operators impose no sequencing requirements, but a few operators impose *sequence points* upon their evaluation: comma, logical-AND, logical-OR, and conditional. In the expression **`(i = 1, a[i] = 0)`**, for example, the side effect (alteration to storage) specified by **`i = 1`** must be completed before the expression **`a[i] = 0`** is evaluated.

Other sequence points are imposed by statement execution and completion of evaluation of a *full expression* (see §6.8). Thus in **`fn(++a)`**, the incrementation of **`a`** must be completed before **`fn`** is called. In **`i = 1; a[i] = 0`**; the side effect of **`i = 1`** must be complete before **`a[i] = 0`** is evaluated.

The notion of *agreement* has to do with the relationship between the *abstract machine* defining the semantics and an actual implementation. An *agreement point* for some object or class of objects is a sequence point at which the value of the object(s) in the real implementation must agree with the value prescribed by the abstract semantics.

For example, compilers that hold variables in registers can sometimes drastically reduce execution times. In a loop like

```
sum = 0;
for (i = 0; i < N; ++i)
    sum += a[i];
```

both **`sum`** and **`i`** might be profitably kept in registers during the execution of the loop. Thus, the actual memory objects designated by **`sum`** and **`i`** would not change state during the loop.

Such behavior is, of course, too loose for hardware-oriented applications such as device drivers and memory-mapped I/O. The following loop looks almost identical to the previous example, but the specification of **`volatile`** ensures that each assignment to **`*ttyport`** takes place in the same sequence, and with the same values, as the abstract machine would have done.

```
volatile short *ttyport;
/* ... */
for (i = 0; i < N; ++i)
    *ttyport = a[i];
```

Another common optimization is to pre-compute common subexpressions. In this loop:

```
volatile short *ttyport;
short mask1, mask2;
/* ... */
for (i = 0; i < N; ++i)
```

```
*ttyport = a[i] & mask1 & mask2;
```

evaluation of the subexpression **mask1 & mask2** could be performed prior to the loop in the real implementation, assuming that neither **mask1** nor **mask2** appear as an operand of the address-of (**&**) operator anywhere in the function. In the abstract machine, of course, this subexpression is reevaluated at each loop iteration, but the real implementation is not required to mimic this repetitiveness, because the variables **mask1** and **mask2** are not **volatile** and the same results are obtained either way.

The previous example shows that a subexpression can be precomputed in the real implementation. A question sometimes asked regarding optimization is, "Is the rearrangement still conforming if the precomputed expression might raise a signal (such as division by zero)?" Fortunately for optimizers, the answer is "Yes," because any evaluation that raises a computational signal has fallen into an *undefined behavior* (§6.5), for which any action is allowable.

Behavior is described in terms of an *abstract machine* to underscore, once again, that the Standard mandates results *as if* certain mechanisms are used, without requiring those actual mechanisms in the implementation. The Standard specifies agreement points at which the value of an object or class of objects in an implementation must agree with the value ascribed by the abstract semantics.

Annex C to the Standard lists the sequence points specified in the body of the Standard.

The class of *interactive devices* is intended to include at least asynchronous terminals, or paired display screens and keyboards. An implementation may extend the definition to include other input and output devices, or even network inter-program connections, provided they obey the Standard's characterization of interactivity.

5.2 Environmental considerations

5.2.1 Character sets

The C89 Committee ultimately came to remarkable unanimity on the subject of character set requirements. There was strong sentiment that C should not be tied to ASCII, despite its heritage and despite the precedent of Ada being defined in terms of ASCII. Rather, an implementation is required to provide a unique character code for each of the printable graphics used by C, and for each of the control codes representable by an escape sequence. (No particular graphic representation for any character is prescribed; thus the common Japanese practice of using the glyph "¥" for the C character "\" is perfectly legitimate.) Translation and execution environments may have different character sets, but each must meet this requirement in its own way. The goal is to ensure that a conforming implementation can translate a C translator written in C.

For this reason, and for economy of description, source code is described *as if* it undergoes the same translation as text that is input by the standard library I/O routines: each line is terminated by some newline character regardless of its external representation.

A new feature of C9X: C9X adds the concept of *universal character name* (UCN) (see §6.4.3) in

order to allow the use of any character in a C source, not just English characters. The primary goal of the Committee was to enable the use of any "native" character in identifiers, string literals and character constants, while retaining the portability objective of C.

With the concept of multibyte characters, "native" characters could be used in string literals and character constants, but this use was very dependent on the implementation and did not usually work in heterogenous environments. Also, this did not encompass identifiers.

Both the C and C++ Committees studied this situation, and the adopted solution was to introduce a new notation for UCNs. Its general forms are **\u***nnnn* and **\U***nnnnnnnn*, to designate a given character according to its short name as described by ISO/IEC 10646. Thus, **\u***nnnn* can be used to designate a Unicode character. This way, programs that must be fully portable may use virtually any character from any script used in the world and still be portable, provided of course that if it prints the character, the execution character set has representation for it.

Of course the notation **\u***nnnn*, like trigraphs, is not very easy to use in everyday programming; so there is a mapping that links UCN and multibyte characters to enable source programs to stay readable by users while maintaining portability. Given the current state of multibyte encodings, this mapping is specified to be implementation-defined; but an implementation can provide the users with utility programs that do the conversion from UCNs to "native" multibytes or vice versa, thus providing a way to exchange source files between implementations using the UCN notation.

UCN models

Once this was adopted, there was still one problem, how to specify UCNs in the Standard. Both the C and C++ Committees studied this situation and the available solutions, and drafted three models:

A. Convert everything to UCNs in basic source characters as soon as possible, that is, in translation phase 1.

B. Use native encodings where possible, UCNs otherwise.

C. Convert everything to wide characters as soon as possible using an internal encoding that encompasses the entire source character set and all UCNs.

Furthermore, in any place where a program could tell which model was being used, the standard should try to label those corner cases as undefined behavior.

The C++ committee defined its Standard in terms of model A, just because that was the clearest to specify (used the fewest hypothetical constructs) because the basic source character set is a well-defined finite set.

The situation is not the same for C given the already existing text for the standard, which allows multibyte characters to appear almost anywhere (the most notable exception being in identifiers), and given the more low-level (or "close to the metal") nature of some uses of the language.

Therefore, the C Committee agreed in general that model B, keeping UCNs and native characters until as late as possible, is more in the "spirit of C" and, while probably more difficult to specify, is more able to encompass the existing diversity. The advantage of model B is also that it might encompass more programs and users' intents than the two others, particularly if shift states are significant in the source text as is often the case in East Asia.

In any case, translation phase 1 begins with an implementation-defined mapping; and such mapping can choose to implement model A or C (but the implementation must document it). As a by-product, a strictly conforming program cannot rely on the specifics handled differently by the three models: examples of non-strict conformance include handling of shift states inside strings and calls like **`fopen("\\ubeda\\file.txt","r")`** and **`#include "sys\udefault.h"`**. Shift states are guaranteed to be handled correctly, however, as long as the implementation performs no mapping at the beginning of phase 1; and the two specific examples given above can be made much more portable by rewriting these as **`fopen("\\" "ubeda\\file.txt", "r")`** and **`#include "sys/udefault.h"`**.

5.2.1.1 Trigraph sequences

Trigraph sequences were introduced in C89 as alternate spellings of some characters to allow the implementation of C in character sets which do not provide a sufficient number of non-alphabetic graphics.

Implementations are required to support these alternate spellings, even if the character set in use is ASCII, in order to allow transportation of code from systems which must use the trigraphs. AMD1 also added *digraphs* (see §6.4.6 and §MSE.4).

The C89 Committee faced a serious problem in trying to define a character set for C. Not all of the character sets in general use have the right number of characters, nor do they support the graphical symbols that C users expect to see. For instance, many character sets for languages other than English resemble ASCII except that codes used for graphic characters in ASCII are instead used for alphabetic characters or diacritical marks. C relies upon a richer set of graphic characters than most other programming languages, so the representation of programs in character sets other than ASCII is a greater problem than for most other programming languages.

ISO (the International Organization for Standardization) uses three technical terms to describe character sets: *repertoire*, *collating sequence*, and *codeset*. The *repertoire* is the set of distinct printable characters. The term abstracts the notion of printable character from any particular representation; the glyphs R, *R*, *R*, **R**, ***R***, R, and R, all represent the same element of the repertoire, "upper-case-R", which is distinct from "lower-case-r". Having decided on the repertoire to be used (C needs a repertoire of 91 characters plus whitespace), one can then pick a *collating sequence* which corresponds to the internal representation in a computer. The repertoire and collating sequence together form the *codeset*.

What is needed for C is to determine the necessary repertoire, ignore the collating sequence altogether (it is of no importance to the language), and then find ways of expressing the repertoire in a way that should give no problems with currently popular codesets.

C derived its repertoire from the ASCII codeset. Unfortunately, the ASCII repertoire is not a subset of all other commonly used character sets; and widespread practice in Europe is not to implement all of ASCII either, but to use some parts of its collating sequence for special national characters.

The solution is an internationally agreed-upon repertoire in terms of which an international representation of C can be defined. ISO has defined such a standard, ISO/IEC 646, which describes an *invariant subset* of ASCII.

The characters in the ASCII repertoire used by C and absent from the ISO/IEC 646 invariant repertoire are:

 # [ ] { } \ | ~ ^

Given this repertoire, the C89 Committee faced the problem of defining representations for the absent characters. The obvious idea of defining two-character escape sequences fails because C uses all the characters which *are* in the ISO/IEC 646 repertoire, so no single escape character is available. The best that can be done is to use a *trigraph*: an escape digraph followed by a distinguishing character.

?? was selected as the escape digraph because it is not used anywhere else in C except as noted below; it suggests that something unusual is going on. The third character was chosen with an eye to graphical similarity to the character being represented.

The sequence **??** cannot occur in a valid pre-C89 program except in strings, character constants, comments, or header names. The character escape sequence **'\?'** (see §6.4.4.4) was introduced to allow two adjacent question marks in such contexts to be represented as **?\?**, a form distinct from the escape digraph. The Committee makes no claims that a program written using trigraphs looks attractive. As a matter of style, it may be wise to surround trigraphs with white space, so that they stand out better in program text. Some users may wish to define preprocessing macros for some or all of the trigraph sequences.

QUIET CHANGE IN C89

> Programs with character sequences such as **??!** in string constants, character constants, or header names will produce different results in C89-conforming translators.

5.2.1.2 Multibyte characters

The “a byte is a character” orientation of C works well for text in Western alphabets, where the number of characters in the character set is under 256. The fit is rather uncomfortable for languages such as Japanese and Chinese, where the repertoire of ideograms numbers in the thousands or tens of thousands. Internally, such character sets can be represented as numeric codes, and it is merely necessary to choose the appropriate integer type to hold any such character. Externally, whether in the files manipulated by a program, or in the text of the source files themselves, a conversion

between these large codes and the various byte-oriented media is necessary.

The support in C of large character sets is based on these principles:

- Multibyte encodings of large character sets are necessary in I/O operations, in source text comments, in source text string and character literals, and beginning with C9X, in native language identifiers.

- No existing multibyte encoding is mandated in preference to any other; no widespread existing encoding should be precluded.

- The null character (**'\0'**) may not be used as part of a multibyte encoding, except for the one-byte null character itself. This allows existing functions which manipulate strings to work transparently with multibyte sequences.

- Shift encodings (which interpret byte sequences in part on the basis of some state information) must start out in a known (default) shift state under certain circumstances such as the start of string literals.

5.2.2 Character display semantics

The Standard defines a number of internal character codes for specifying "format-effecting actions on display devices," and provides printable escape sequences for each of them. These character codes are clearly modeled after ASCII control codes, and the mnemonic letters used to specify their escape sequences reflect this heritage. Nevertheless, they are *internal* codes for specifying the format of a display in an environment-independent manner; they must be written to a *text file* to effect formatting on a display device. The Standard states quite clearly that the external representation of a text file (or data stream) may well differ from the internal form, both in character codes and number of characters needed to represent a single internal code.

The distinction between internal and external codes most needs emphasis with respect to *new-line*. NCITS L2, Codes and Character Sets (and now also ISO/IEC JTC 1/SC2/WG1, 8 Bit Character Sets), uses the term to refer to an external code used for information interchange whose display semantics specify a move to the next line. Although ISO/IEC 646 deprecates the combination of the motion to the next line with a motion to the initial position on the line, the C Standard uses *new-line* to designate the end-of-line internal code represented by the escape sequence **'\n'**. While this ambiguity is perhaps unfortunate, use of the term in the latter sense is nearly universal within the C community. But the knowledge that this internal code has numerous external representations depending upon operating system and medium is equally widespread.

The alert sequence (**'\a'**) was added by popular demand to replace, for instance, the ASCII BEL code explicitly coded as **'\007'**.

Proposals to add **'\e'** for ASCII ESC (**'\033'**) were not adopted because other popular character sets have no obvious equivalent (see §6.4.4.4.)

The vertical tab sequence (**'\v'**) was added since many existing implementations support it, and since it is convenient to have a designation within the language for all the defined white space characters.

The semantics of the motion control escape sequences carefully avoid the Western language assumptions that printing advances left-to-right and top-to-bottom.

To avoid the issue of whether an implementation conforms if it cannot properly effect vertical tabs (for instance), the Standard emphasizes that the semantics merely describe *intent*.

5.2.3 Signals and interrupts

Signals are difficult to specify in a system-independent way. The C89 Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a **volatile static** variable which can be written uninterruptedly and promptly return. (The header **<signal.h>** specifies a type **sig_atomic_t** which can be so written.) It is further guaranteed that a signal handler will not corrupt the automatic storage of an instantiation of any executing function, even if that function is called within the signal handler. No such guarantees can be extended to library functions, with the explicit exceptions of **longjmp** (§7.13.2.1) and **signal** (§7.14.1.1), since the library functions may be arbitrarily interrelated and since some of them have profound effect on the environment.

Calls to **longjmp** are problematic, despite the assurances of §7.13.2.1. The signal could have occurred during the execution of some library function which was in the process of updating external state and/or static variables.

A second signal for the same handler could occur before the first is processed, and the Standard makes no guarantees as to what happens to the second signal.

5.2.4 Environmental limits

The C89 Committee agreed that the Standard must say something about certain capacities and limitations, but just how to enforce these treaty points was the topic of considerable debate.

5.2.4.1 Translation limits

The Standard requires that an implementation be able to translate and execute some program that meets each of the stated limits. This criterion was felt to give a useful latitude to the implementor in meeting these limits. While a deficient implementation could probably contrive a program that meets this requirement, yet still succeed in being useless, the C89 Committee felt that such ingenuity would probably require more work than making something useful. The sense of both the C89 and C9X Committees was that implementors should not construe the translation limits as the values of hard-wired parameters, but rather as a set of criteria by which an implementation will be judged.

Some of the limits chosen represent interesting compromises. The goal was to allow reasonably

large portable programs to be written, without placing excessive burdens on reasonably small implementations, some of which might run on machines with only 64K of memory. In C9X, the minimum amount of memory for the target machine was raised to 512K. In addition, the Committee recognized that smaller machines rarely serve as a host for a C compiler: programs for embedded systems or small machines are almost always developed using a cross compiler running on a personal computer or workstation. This allows for a great increase in some of the translation limits.

C89's minimum maximum limit of 257 cases in a switch statement allows coding of lexical routines which can branch on any character (one of at least 256 values) or on the value **`EOF`**. This has been extended to 1023 cases in C9X.

The requirement that a conforming implementation be able to translate and execute at least one program that reaches each of the stated limits is not meant to excuse the implementation from doing the best it can to translate and execute other programs. It was deemed infeasible to require successful translation and execution of *all* programs not exceeding those limits. Many of these limits require resources such as memory that a reasonable implementation might allocate from a shared pool; so there is no requirement that all the limits be attained *simultaneously*. Requiring just one acceptable program that attains each limit is simply meant to ensure conformance with these requirements.

The C9X Committee reviewed several proposed changes to strengthen or clarify the wording on conformance, especially with respect to translation limits. The belief was that it is simply not practical to provide a specification which is strong enough to be useful, but which still allows for real-world problems such as bugs. The Committee therefore chose to consider the matter a quality-of-implementation issue, and to leave translation limits in the standard to give guidance.

5.2.4.2 Numerical limits

5.2.4.2.1 Sizes of integer types `<limits.h>`

Such a large body of C code has been developed for 8-bit byte machines that the integer sizes in such environments must be considered normative. The prescribed limits are minima: an implementation on a machine with 9-bit bytes can be conforming, as can an implementation that defines **`int`** to be the same width as **`long`**. The negative limits have been chosen to accommodate one's-complement or sign-magnitude implementations, as well as the more usual two's-complement. The limits for the maxima and minima of unsigned types are specified as unsigned constants (e.g., **`65535u`**) to avoid surprising widening of expressions involving these extrema.

The macro **`CHAR_BIT`** makes available the number of bits in a **`char`** object. The C89 Committee saw little utility in adding such macros for other data types.

The names associated with the **`short int`** types (**`SHRT_MIN`**, etc., rather than **`SHORT_MIN`**, etc.) reflect prior art rather than obsessive abbreviation on the C89 Committee's part.

5.2.4.2.2 Characteristics of floating types `<float.h>`

The characterization of floating point follows, with minor changes, that of the Fortran standardization committee. The C89 Committee chose to follow the Fortran model in some part out of a concern for Fortran-to-C translation, and in large part out of deference to the Fortran committee's greater experience with fine points of floating point usage. Note that the floating point model adopted permits all common representations, including sign-magnitude and two's-complement, but precludes a logarithmic implementation.

The C89 Committee also endeavored to accommodate the IEEE 754 floating point standard by not adopting any constraints on floating point which were contrary to that standard. IEEE 754 is now an international standard, IEC 60559; and that is how it is referred to in C9X.

The term **FLT_MANT_DIG** stands for "float mantissa digits." The Standard now uses the more precise term *significand* rather than *mantissa*.

In C9X, all values except **FLT_ROUNDS** in **<float.h>** must be usable as static initializers.

The overflow and/or underflow thresholds may not be the same for all arithmetic operations. For example, there is at least one machine where the overflow threshold for addition is twice as big as for multiplication. Another implementation uses a pair of **double**s to represent a **long double**. In that implementation, the next representable **long double** value after **1.0L** is **1.0L + LDBL_MIN**, yet, the difference between those two numbers (**LDBL_MIN**) is not $b^{(1-p)}$, otherwise known as **LDBL_EPSILON**. Because of anomalies like these, there are few hard requirements on the **<float.h>** values. But, the values in **<float.h>** should be in terms of the hardware representation used to store floating point values in memory, not in terms of the effective accuracy of operations, nor in terms of registers, and should apply to all operations. The representation stored in memory may have padding bits and/or bytes that do not contribute to the value. The padding should not be included in the **<float.h>** values.

Because of the practical difficulty involved in defining a uniform metric that all vendors would be willing to follow (just computing the accuracy reliably could be a significant, and because the importance of floating point accuracy differs greatly among users, the standard allows a great deal of latitude in how an implementation documents the accuracy of the real and complex floating point operations and functions.

Here are some ways that an implementation might address the need to define the accuracy:

- digits correct
- digits wrong
  - maximum Units in the Last Place (ULPs) error
  - maximum absolute error
  - maximum relative error

[1] See X3J3 working document S8-112.

For complex values, some methods are:

error in terms of both real and imaginary parts

error in terms of Euclidean norm, $\|a + ib\| = \sqrt{(a*a + b*b)}$

There are two usages of the term ULP. One is in the context of differences between two numbers, that is, *f(x)* differs from *F(x)* by 3 ULPs. The other is the value of the ULP of a number, that is, an ULP of the value 1.0 is **`DBL_EPSILON`**. For this discussion, we are interested in the former; the difference between the computed value and the infinitely precise value.

The error between two floating-point numbers in ULPs depends on the radix and the precision used in representing the number, but not the exponent. With a decimal radix and 3 digits of precision, the computed value **`0.314e+1`** differs from the value **`0.31416e+1`** by 0.16 ULPs. If both numbers are scaled by the same power of the radix, for example, **`0.314e+49`** and **`0.31416e+49`**, they still differ by 0.16 ULPs.

When the two numbers being compared span a power of the radix, the two possible ULP error calculations differ by a factor of the radix. For a decimal radix and 3 digits of precision, consider the two values **`9.99e2`** and **`1.01e3`**. These are the two values adjacent to the value **`1.00e3`**, a power of the radix, in this number system. If 999 is the correct value and 1010 is the computed value, the error is 11 ULPs; but, if 1010 is the correct value and 999 is the computed value, then the error is 1.1 ULPs.

Some math functions such as those that do argument reduction modulo an approximation of π have good accuracy for small arguments, but poor accuracy for large arguments. It is not unusual for an implementation of the trigonometric functions to have zero bits correct in the computed result for large arguments. For cases like this, an implementation might break the domain of the function into disjoint regions and specify the accuracy in each region.

If an implementation documents worst case error, there is no requirement that it be the minimum worst case error. That is, if a vendor believes that the worst case error for a function is around 5 ULPs, they could document it as 7 ULPs to be safe.

The Committee could not agree on upper limits on accuracy that all conforming implementations must meet, for example, “addition is no worse than 2 ULPs for all implementations.” This is a quality of implementation issue.

Implementations that conform to IEC 60559 have one half ULP accuracy in round-to-nearest mode, and one ULP accuracy in the other three rounding modes, for the basic arithmetic operations and square root. For other floating point arithmetics, it is a rare implementation that has worse than one ULP accuracy for the basic arithmetic operations.

The accuracy of decimal-to-binary conversions and format conversions are discussed elsewhere in the Standard.

For the math library functions, fast, correctly rounded 0.5 ULP accuracy remains a research problem. Some implementations provide two math libraries, one being faster but less accurate than the other.

The C9X Committee discussed the idea of allowing the programmer to find out the accuracy of floating point operations and math functions during compilation (say, via macros) or during execution (with a function call), but neither got enough support to warrant the change to the Standard. The use of macros would require over one hundred symbols to name every math function, for example, **`ULP_SINF`**, **`ULP_SIN`**, and **`ULP_SINL`** just for the real-valued **`sin`** function. One possible function implementation might be a function that takes the name of the operation or math function as a string, **`ulp_err("sin")`** for example, that would return a **`double`** such as 3.5 to indicate the worst case error, with –1.0 indicating unknown error. But such a simple scheme would likely be of very limited use given that so many functions have accuracies that differ significantly across their domains. Constrained to worst case error across the entire domain, most implementations would wind up reporting either unknown error or else a uselessly large error for a very large percentage of functions. This would be useless because most programs that care about accuracy are written in the first place to try to compensate for accuracy problems that typically arise when pushing domain boundaries; and implementing something more useful like the worst case error for a user-specified partition of the domain would be excessively difficult.

NaNs

C9X does not define the behavior of signaling NaNs, nor does it specify the interpretation of NaN significands.

The IEC 60559 floating-point standard specifies quiet and signaling NaNs, but these terms can be applied for some non-IEC 60559 implementations as well. For example, the VAX reserved operand and the CRAY indefinite qualify as signaling NaNs. In IEC 60559 standard arithmetic, operations that trigger a signaling NaN argument generally return a quiet NaN result provided no trap is taken. Full support for signaling NaNs implies restartable traps, such as the optional traps specified in the IEC 60559 floating-point standard.

The primary utility of quiet NaNs, as stated in IEC 60559, “to handle otherwise intractable situations, such as providing a default value for 0.0/0.0,” is supported by this specification.

Other applications of NaNs may prove useful. Available parts of NaNs have been used to encode auxiliary information, for example about the NaN’s origin. Signaling NaNs might be candidates for filling uninitialized storage; and their available parts could distinguish uninitialized floating objects. IEC 60559 signaling NaNs and trap handlers potentially provide hooks for maintaining diagnostic information or for implementing special arithmetics.

However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. The IEC 60559 floating-point standard recommends that NaNs propagate; but it does not require this and not all implementations do. And the floating-point standard fails to specify the contents of NaNs

through format conversion. Making signaling NaNs predictable imposes optimization restrictions that anticipated benefits don't justify. For these reasons this standard does not define the behavior of signaling NaNs nor specify the interpretation of NaN significands.

A draft version of the NCEG floating-point specification included signaling NaNs. It could serve as a guide for implementation extensions in support of signaling NaNs.

6. Language

While more formal methods of language definition were explored, the C89 Committee decided early on to employ the style of K&R: Backus-Naur Form for the syntax and prose for the constraints and semantics. Anything more ambitious was considered to be likely to delay the Standard, and to make it less accessible to its audience.

6.2 Concepts

6.2.1 Scopes of identifiers

C89 separated from the overloaded keywords for storage classes the various concepts of *scope*, *linkage*, *name space*, and *storage duration* (see §6.2.2, §6.2.3 and §6.2.4.). This has traditionally been a major area of confusion.

One source of dispute was whether identifiers with external linkage should have file scope even when introduced within a block. K&R was vague on this point, and has been interpreted differently by different pre-C89 implementations. For example, the following fragment would be valid in the file scope scheme, while invalid in the block scope scheme:

```
typedef struct data d_struct;

first()
{
    extern d_struct func();
    /* … */
}

second()
{
    d_struct n = func();
}
```

While it was generally agreed that it is poor practice to take advantage of an external declaration once it had gone out of scope, some argued that a translator had to remember the declaration for checking anyway, so why not acknowledge this? The compromise adopted was to decree essentially that block scope rules apply, but that a conforming implementation need not diagnose a failure to redeclare an external identifier that had gone out of scope (*undefined behavior*).

QUIET CHANGE IN C89

A program relying on file scope rules may be valid under block scope rules but behave differently, for instance, if **d_struct** were defined as type **float** rather than **struct data** in the example above.

Although the scope of an identifier in a function prototype begins at its declaration and ends at the end of that function's declarator, this scope is ignored by the preprocessor. Thus an identifier in a prototype having the same name as that of an existing macro is treated as an invocation of that macro. For example:

```
#define status 23
void exit(int status);
```

generates an error, since the prototype after preprocessing becomes

```
void exit(int 23);
```

Perhaps more surprising is what happens if **status** is defined

```
#define status []
```

Then the resulting prototype is

```
void exit(int []);
```

which is syntactically correct but semantically quite different from the intent.

To protect an implementation's header prototypes from such misinterpretation, the implementor must write them to avoid these surprises. Possible solutions include not using identifiers in prototypes, or using names (such as _ **_status** or **_Status**) in the reserved name space.

6.2.2 Linkages of identifiers

The first declaration of an identifier, including implicit declarations before C9X, must specify by the presence or absence of the keyword **static** whether the identifier has internal or external linkage. This requirement allows for one-pass compilation in an implementation which must treat internal linkage items differently from external linkage items. An example of such an implementation is one which produces intermediate assembler code, and which therefore must construct names for internal linkage items to circumvent identifier length and/or case restrictions in the target assembler.

Pre-C89 practice in this area was inconsistent. Some implementations avoided the renaming problem simply by restricting internal linkage names by the same rules as the ones used for external linkage. Others have disallowed a static declaration followed later by a defining instance, even

though such constructs are necessary to declare mutually-recursive static functions. The requirements adopted in C89 called for changes in some existing programs, but allowed for maximum flexibility.

The definition model to be used for objects with external linkage was a major C89 standardization issue. The basic problem was to decide which declarations of an object define storage for the object, and which merely reference an existing object. A related problem was whether multiple definitions of storage are allowed, or only one is acceptable. Pre-C89 implementations exhibit at least four different models, listed here in order of increasing restrictiveness:

Common Every object declaration with external linkage, regardless of whether the keyword **`extern`** appears in the declaration, creates a definition of storage. When all of the modules are combined together, each definition with the same name is located at the same address in memory. (The name is derived from *common storage* in Fortran.) This model was the intent of the original designer of C, Dennis Ritchie.

Relaxed Ref/Def The appearance of the keyword **`extern`** in a declaration, regardless of whether it is used inside or outside of the scope of a function, indicates a pure reference (ref), which does not define storage. Somewhere in all of the translation units, at least one definition (def) of the object must exist. An external definition is indicated by an object declaration in file scope containing no storage class indication. A reference without a corresponding definition is an error. Some implementations also will not generate a reference for items which are declared with the **`extern`** keyword but are never used in the code. The UNIX operating system C compiler and linker implement this model, which is recognized as a *common extension* to the C language (see §K.5.11). UNIX C programs which take advantage of this model are standard conforming in their environment, but are not maximally portable (not strictly conforming).

Strict Ref/Def This is the same as the relaxed ref/def model, save that only one definition is allowed. Again, some implementations may decide not to put out references to items that are not used. This is the model specified in K&R.

Initialization This model requires an explicit initialization to define storage. All other declarations are references.

Figure 6.1 demonstrates the differences between the models. The intent is that Figure 6.1 shows working programs in which the symbol **`i`** is neither undefined nor multiply defined.

The Standard model is a combination of features of the strict ref/def model and the initialization model. As in the strict ref/def model, only a single translation unit contains the definition of a given object because many environments cannot effectively or efficiently support the “distributed definition” inherent in the common or relaxed ref/def approaches. However, either an initialization, or an appropriate declaration without storage class specifier (see §6.9), serves as the external definition. This composite approach was chosen to accommodate as wide a range of environments and existing implementations as possible.

Figure 6.1: Comparison of identifier linkage models

Model	File 1	File 2

Common	`extern int i;`
	`int main( )`
	`{`
	`  i = 1;`
	`  second( );`
	`}`
		`extern int i;`
		`void second( )`
		`{`
		`  third( i );`
		`}`

Relaxed Ref/Def	`int i;`
	`int main( )`
	`{`
	`  i = 1;`
	`  second( );`
	`}`
		`int i;`
		`void second( )`
		`{`
		`  third( i );`
		`}`

Strict Ref/Def	`int i;`
	`int main( )`
	`{`
	`  i = 1;`
	`  second( );`
	`}`
		`extern int i;`
		`void second( )`
		`{`
		`  third( i );`
		`}`

Initializer	`int i = 0;`
	`int main( )`
	`{`
	`  i = 1;`
	`  second( );`
	`}`
		`int i;`
		`void second( )`
		`{`
		`  third( i );`
		`}`

6.2.3 Name spaces of identifiers

Pre-C89 implementations varied considerably in the number of separate name spaces maintained. The position adopted in the Standard is to permit as many separate name spaces as can be distinguished by context, except that all tags (**`struct`**, **`union`**, and **`enum`**) comprise a single name space.

6.2.4 Storage durations of objects

It was necessary to clarify the effect on automatic storage of jumping into a block that declares local storage (see §6.8.2.). While many implementations could traditionally allocate the maximum depth of automatic storage upon entry to a function, the addition to C9X of the variable length array feature (§6.7.5.2) forces the implementation to allocate some objects when the declaration is encountered.

A new feature of C9X: C89 requires all declarations in a block to occur before any statements. On the other hand, many languages similar to C (such as Algol 68 and C++) permit declarations and statements to be mixed in an arbitrary manner. This feature has been found to be useful and has been added to C9X.

Declarations that initialize variables can contain complex expressions and have arbitrary side-

effects, and it is necessary to define when these take place, particularly when the flow of control involves arbitrary jumps. There is a simple rule of thumb: the variable declared is created with an unspecified value when the block is entered, but the initializer is evaluated and the value placed in the variable when the declaration is reached in the normal course of execution. Thus a jump forward past a declaration leaves it uninitialized, while a jump backwards will cause it to be initialized more than once. If the declaration does not initialize the variable, it sets it to an unspecified value even if this is not the first time the declaration has been reached.

The scope of a variable starts at its declaration. Therefore, although the variable exists as soon as the block is entered, it cannot be referred to by name until its declaration is reached.

Example:

```
int j = 42;
{
    int i = 0;

loop:
    printf("I = %4d, ", i);
    printf("J1 = %4d, ", ++j);
    int j = i;
    printf("J2 = %4d, ", ++j);
    int k;
    printf("K1 = %4d, ", k);
    k = i * 10;
    printf("K2 = %4d, ", k);
    if (i % 2 == 0) goto skip;
    int m = i * 5;
skip:
    printf("M = %4d\n", m);

    if (++i < 5) goto loop;
}
```

will output:

```
I =    0, J1 =   43, J2 =    1, K1 = ????, K2 =    0, M = ????
I =    1, J1 =   44, J2 =    2, K1 = ????, K2 =   10, M =    5
I =    2, J1 =   45, J2 =    3, K1 = ????, K2 =   20, M =    5
I =    3, J1 =   46, J2 =    4, K1 = ????, K2 =   30, M =   15
I =    4, J1 =   47, J2 =    5, K1 = ????, K2 =   40, M =   15
```

where "**????**" indicates an indeterminate value (and any use of an indeterminate value is undefined behavior).

These rules have to be modified slightly for variable length arrays. The implementation will not know how much space is required for the array until its declaration is reached, and so cannot create

it until then. This has two implications for jumps:

A jump to a point after the declaration of a VLA is forbidden, because it would be possible to refer to the VLA without creating it. Such a jump requires a diagnostic.

A jump to a point before the declaration of a VLA destroys the VLA.

A number of other approaches were considered, but there were problems with all of them. In particular, this choice of rules ensures that VLAs can always be destroyed in the reverse order of their creation, which is essential if they are placed on the stack.

To effect true reentrancy for functions in the presence of signals raised asynchronously (see §5.2.3), an implementation must assure that the storage for function return values has automatic duration. For example, the caller could allocate automatic storage for the return value and communicate its location to the called function. (The typical case of return registers for small-sized types conforms to this requirement: the calling convention of the implementation implicitly communicates the return location to the called function.)

6.2.5 Types

Several new types were added in C89:

```
void
void*
signed char
unsigned char
unsigned short
unsigned long
long double
```

And new designations for existing types were added:

signed short	for	**short**
signed int	for	**int**
signed long	for	**long**

C9X also adds new types:

```
_Bool
long long
unsigned long long
float _Imaginary
float _Complex
double _Imaginary
double _Complex
long dobule _Imaginary
long double _Complex
```

C9X also allows extended integer types (see §7.8, **<inttypes.h>**, and §7.18, **<stdint.h>**) and a boolean type (see §7.16, **<stdbool.h>**).

void is used primarily as the typemark for a function that returns no result. It may also be used as the cast **(void)** to indicate explicitly that the value of an expression is to be discarded while retaining the expression's side effects. Finally, a function prototype list that has no arguments is written as **f(void)**, because **f()** retains its old meaning that nothing is said about the arguments. Note that there is no such thing as a "void object."

A "pointer to **void**," **void***, is a generic pointer capable of pointing to any object (except for bit-fields and objects declared with the **register** storage class) without loss of information. A pointer to **void** must have the same representation and alignment as a pointer to **char**; the intent of this rule is to allow existing programs that call library functions such as **memcpy** and **free** to continue to work. A pointer to **void** cannot be dereferenced, although such a pointer can be converted to a normal pointer type which can be dereferenced. Pointers to other types coerce silently to and from **void*** in assignments, function prototypes, comparisons, and conditional expressions, whereas other pointer type clashes are invalid. It is undefined what will happen if a pointer of some type is converted to **void***, and then the **void*** pointer is converted to a type with a stricter alignment requirement. Three types of **char** are specified: **signed**, plain, and **unsigned**. A plain **char** may be represented as either signed or unsigned depending upon the implementation, as in prior practice. The type **signed char** was introduced in C89 to make available a one-byte signed integer type on those systems which implement plain **char** as **unsigned char**. For reasons of symmetry, the keyword **signed** is allowed as part of the type name of other integer types. Two varieties of the integer types are specified: **signed** and **unsigned**. If neither specifier is used, **signed** is assumed. The only unsigned type in K&R is **unsigned int**.

The keyword **unsigned** is something of a misnomer, suggesting as it does in arithmetic that it is non-negative but capable of overflow. The semantics of the C type **unsigned** is that of modulus, or wrap-around, arithmetic for which overflow has no meaning. The result of an **unsigned** arithmetic operation is thus always defined, whereas the result of a signed operation may be undefined. In practice, on two's-complement machines, both types often give the same result for all operators except division, modulus, right shift, and comparisons. Hence there has been a lack of sensitivity in the C community to the differences between signed and unsigned arithmetic.

A new floating type, **long double**, was added in C89. The **long double** type must offer at least as much precision as the **double** type. Several architectures support more than two floating point types and thus can map a distinct machine type onto this additional C type. Several architectures which support only two floating point types can also take advantage of the three C types by mapping the less precise type onto both **float** and **double**, and designating the more precise type **long double**. Architectures in which this mapping might be desirable include those in which single-precision types offer at least as much precision as most other machines' double-precision, or those on which single-precision arithmetic is considerably more efficient than double-precision. Thus the common C floating types would map onto an efficient implementation type, but the more precise type would still be available to those programmers who require its use. See

Annex F for a discussion of IEC 60559 floating-point types.

To avoid confusion, **long float** as a synonym for **double** was retired in C89.

Floating types of different widths (**double** wider than **float** and **long double** wider than **double**) facilitate porting code that, intentionally or not, depends on differences in type widths. Many results are exact or correctly rounded when computed with twice the number of digits of precision as the data. For example, the calculation

```
float d, x, y, z, w;
/* ... */
d = (double) x * y - (double) z * w;
```

yields a correctly rounded determinant if **double** has twice the precision of **float** and the individual operations are correctly rounded. (The casts to **double** are unnecessary if the minimum evaluation format is **double** or **long double**.)

A new feature of C9X: Complex types were added to C as part of the effort to make C suitable and attractive for general numerical programming. Complex arithmetic is used heavily in certain important application areas.

The underlying implementation of the complex types is Cartesian, rather than polar, for overall efficiency and consistency with other programming languages. The implementation is explicitly stated so that characteristics and behaviors can be defined simply and unambiguously.

Enumerations permit the declaration of named constants in a more convenient and structured fashion than does **#define**. Both enumeration constants and variables behave like integer types for the sake of type checking, however.

The C89 Committee considered several alternatives for enumeration types in C:

1. leave them out;
2. include them as definitions of integer constants;
3. include them in the weakly typed form of the UNIX C compiler;
4. include them with strong typing as in Pascal.

The C89 Committee adopted the second alternative on the grounds that this approach most clearly reflects common practice. Doing away with enumerations altogether would invalidate a fair amount of existing code; stronger typing than integer creates problems, for example, with arrays indexed by enumerations.

6.2.6 Representations of types

6.2.6.2 Integer types

The C89 Committee explicitly required binary representation of integers on the grounds that this stricture was implicit in any case:

- Bit fields are specified by a number of bits, with no mention of "invalid integer" representation. The only reasonable encoding for such bit fields is binary.
- The integer formats for **`printf`** suggest no provision for "invalid integer" values, implying that any result of bitwise manipulation produces an integer result which can be printed by **`printf`**.
- All methods of specifying integer constants—decimal, hex, and octal—specify an integer value. No method independent of integers is defined for specifying "bit-string constants." Only a binary encoding provides a complete one-to-one mapping between bit strings and integer values.

The restriction to binary numeration systems rules out such curiosities as Gray code and makes possible arithmetic definitions of the bitwise operators on unsigned types.

6.2.7 Compatible type and composite type

The concepts of *compatible type* and *composite type* were introduced to allow C89 to discuss those situations in which type declarations need not be identical. These terms are especially useful in explaining the relationship between an incomplete type and a completed type. With the addition of variable length arrays (§6.7.5.2) in C9X, array type compatibility was extended so that variable length arrays are compatible with both an array of known constant size and an array with an incomplete type.

Structure, union, or enumeration type declarations in two different translation units do not formally declare the *same type*, even if the text of these declarations come from the same header file, since the translation units are themselves disjoint. The Standard thus specifies additional compatibility rules for such types so that two such declarations are compatible if they are sufficiently similar.

6.3 Conversions

6.3.1 Arithmetic operands

6.3.1.1 Booleans, characters and integers

Between the publication of K&R and the development of C89, a serious divergence had occurred among implementations in the evolution of integer promotion rules. Implementations fell into two major camps which may be characterized as *unsigned preserving* and *value preserving*. The difference between these approaches centered on the treatment of **`unsigned char`** and **`unsigned short`** when widened by the *integer promotions*, but the decision had an impact on

the typing of constants as well (see §6.4.4.1).

The *unsigned preserving* approach calls for promoting the two smaller unsigned types to **unsigned int**. This is a simple rule, and yields a type which is independent of execution environment.

The *value preserving* approach calls for promoting those types to **signed int** if that type can properly represent all the values of the original type, and otherwise for promoting those types to **unsigned int**. Thus, if the execution environment represents **short** as something smaller than **int**, **unsigned short** becomes **int**; otherwise it becomes **unsigned int**.

Both schemes give the same answer in the vast majority of cases, and both give the same effective result in even more cases in implementations with two's-complement arithmetic and quiet wraparound on signed overflow—that is, in most current implementations. In such implementations, differences between the two only appear when these two conditions are both true:

1. An expression involving an **unsigned char** or **unsigned short** produces an **int**-wide result in which the sign bit is set, that is, either a unary operation on such a type, or a binary operation in which the other operand is an **int** or "narrower" type.

2. The result of the preceding expression is used in a context in which its signedness is significant:

   - **sizeof(int) < sizeof(long)** and it is in a context where it must be widened to a **long** type, or

   - it is the left operand of the right-shift operator in an implementation where this shift is defined as arithmetic, or

   - it is either operand of **/**, **%**, **<**, **<=**, **>**, or **>=**.

In such circumstances a genuine ambiguity of interpretation arises. The result must be dubbed *questionably signed*, since a case can be made for either the signed or unsigned interpretation. Exactly the same ambiguity arises whenever an **unsigned int** confronts a **signed int** across an operator, and the **signed int** has a negative value. Neither scheme does any better, or any worse, in resolving the ambiguity of this confrontation. Suddenly, the negative **signed int** becomes a very large **unsigned int**, which may be surprising, or it may be exactly what is desired by a knowledgeable programmer. Of course, *all of these ambiguities can be avoided by a judicious use of casts*.

One of the important outcomes of exploring this problem is the understanding that high-quality compilers might do well to look for such questionable code and offer (optional) diagnostics, and that conscientious instructors might do well to warn programmers of the problems of implicit type conversions.

The unsigned preserving rules greatly increase the number of situations where **`unsigned int`** confronts **`signed int`** to yield a questionably signed result, whereas the value preserving rules minimize such confrontations. Thus, the value preserving rules were considered to be safer for the novice, or unwary, programmer. After much discussion, the C89 Committee decided in favor of value preserving rules, despite the fact that the UNIX C compilers had evolved in the direction of unsigned preserving.

QUIET CHANGE IN C89

> A program that depends upon unsigned preserving arithmetic conversions will behave differently, probably without complaint. This was considered the most serious semantic change made by the C89 Committee to a widespread current practice.

The Standard clarifies that the integer promotion rules also apply to bit fields.

6.3.1.2 Boolean type

Note that, although **`_Bool`** is technically an integer type, conversion to **`_Bool`** does not always work the same as conversion to other integer types. Consider, for example, that the expression **`(_Bool)0.5`** evaluates to 1, whereas **`(int)0.5`** evaluates to 0. The first result is correct: it simply says that 0.5 is non-zero; but it may be somewhat counter-intuitive unless a **`_Bool`** is thought of as a "truth value" rather than as a one-bit integer.

6.3.1.3 Signed and unsigned integers

Precise rules are now provided for converting to and from unsigned integers. On a two's-complement machine, the operation is still virtual (no change of representation is required), but the rules are now stated independent of representation.

6.3.1.4 Real floating and integer

There was strong agreement in the C89 Committee that floating values should truncate toward zero when converted to an integer type, the specification adopted in the Standard. Although K&R permitted negative floating values to truncate away from zero, no C89 Committee member knew of an implementation that functioned in such a manner.[2]

Note that conversion from integer to floating may indeed require rounding if the integer is too wide to represent exactly in the floating-point format.

6.3.1.5 Real floating types

C89, unlike K&R, did not require rounding in the **`double`** to **`float`** conversion. Some widely

[2] The Committee has since learned of one such implementation.

used IEC 60559 floating point processor chips control floating to integer conversion with the same mode bits as for double-precision to single-precision conversion. Since truncation-toward-zero is the appropriate setting for C in the former case, it would be expensive to require such implementations to round to **`float`**. In C9X, §F.7.3 requires round-to-nearest for conversions between floating formats as required by IEC 60559.

6.3.1.8 Usual arithmetic conversions

The rules in the Standard for these conversions are slight modifications of those in K&R: the modifications accommodate the added types and the value preserving rules. Explicit license was added to perform calculations in a "wider" type than absolutely necessary, since this can sometimes produce smaller and faster code, not to mention the correct answer more often. Calculations can also be performed in a "narrower" type by the *as if* rule so long as the same end result is obtained. *Explicit casting can always be used to obtain a value in a desired type.*

The C9X Committee relaxed the requirement that **`float`** operands be converted to **`double`**. An implementation may still choose to convert.

QUIET CHANGE IN C89

> Expressions with **`float`** operands may be computed at lower than double precision. K&R specified that all floating point operations be done in **`double`**.

Real and imaginary operands are not converted to complex because doing so would require extra computation, while producing undesirable results in certain cases involving infinities, NaNs and signed zeros. For example, with automatic conversion to complex,

$$2.0 \times (3.0 + i\infty) \Rightarrow (2.0 + i0.0) \times (3.0 + i\infty) \Rightarrow$$
$$(2.0 \times 3.0 - 0.0 \times \infty) + i(2.0 \times \infty + 0.0 \times 3.0) \Rightarrow \mathrm{NaN} + i\infty$$

rather than the desired result, $6.0 + i\infty$. Optimizers for implementations with infinities, including all IEC 60559 ones, would not be able to eliminate the operations with the zero imaginary part of the converted operand.

The following example illustrates the problem with signed zeros. With automatic conversion to complex,

$$2.0 \times (3.0 - i0.0) \Rightarrow (2.0 + i0.0) \times (3.0 - i0.0) \Rightarrow$$
$$(2.0 \times 3.0 + 0.0 \times 0.0) + i(-2.0 \times 0.0 + 0.0 \times 3.0) \Rightarrow 6.0 + i0.0$$

rather than the desired result, $6.0 - i0.0$.

The problems illustrated in the examples above have counterparts for imaginary operands. The mathematical product $i2.0 \times (\infty + i3.0)$ should yield $-6.0 + i\infty$, but with automatic conversion to complex,

$$i2.0 \times (\infty + i3.0) \Rightarrow (0.0 + i2.0) \times (\infty + i3.0) \Rightarrow$$
$$(0.0 \times \infty - 2.0 \times 3.0) + i(0.0 \times 3.0 + 2.0 \times \infty) \Rightarrow \mathrm{NaN} + i\infty$$

This also demonstrates the need for imaginary types specified in Annex G. Without them, $i2.0$ would have to be represented as $0.0 + i2.0$, implying that $\mathrm{NaN} + i\infty$ would be the semantically correct result regardless of conversion rules; and optimizers for implementations with infinities would not be able to eliminate the operations with the zero real part.

6.3.2 Other operands

6.3.2.1 Lvalues, arrays and function designators

A difference of opinion within the C community centered around the meaning of *lvalue*, one group considering an lvalue to be any kind of object locator, another group holding that an lvalue is meaningful on the left side of an assigning operator. The C89 Committee adopted the definition of lvalue as an object locator. The term *modifiable lvalue* is used for the second of the above concepts.

The role of array objects has been a classic source of confusion in C, in large part because of the numerous contexts in which an array reference is converted to a pointer to its first element. While this conversion neatly handles the semantics of subscripting, the fact that **`a[i]`** is a modifiable lvalue while **`a`** is not has puzzled many students of the language. A more precise description was incorporated in C89 in the hope of combatting this confusion.

6.3.2.2 `void`

The description of operators and expressions is simplified by saying that **`void`** yields a value, with the understanding that the value has no representation, and hence requires no storage.

6.3.2.3 Pointers

C has now been implemented on a wide range of architectures. While some of these architectures feature uniform pointers which are the size of some integer type, maximally portable code cannot assume any necessary correspondence between different pointer types and the integer types. On some implementations, pointers can even be wider than any integer type.

The use of **`void*`** (“pointer to **`void`**”) as a generic object pointer type is an invention of the C89 Committee. Adoption of this type was stimulated by the desire to specify function prototype arguments that either quietly convert arbitrary pointers (as in **`fread`**) or complain if the argument type does not exactly match (as in **`strcmp`**). Nothing is said about pointers to functions, which may be incommensurate with object pointers and/or integers.

Since pointers and integers are now considered incommensurate, the only integer value that can be safely converted to a pointer is a constant expression with the value 0. The result of converting any other integer value, including a non-constant expression with the value 0, to a pointer is implementation-defined.

Consequences of the treatment of pointer types in the Standard include:

- A pointer to **`void`** may be converted to a pointer to an object of any type.
- A pointer to any object of any type may be converted to a pointer to **`void`**.
- If a pointer to an object is converted to a pointer to **`void`** and back again to the original pointer type, the result compares equal to original pointer.
- It is invalid to convert a pointer to an object of any type to a pointer to an object of a different type without an explicit cast.
- Even with an explicit cast, it is invalid to convert a function pointer to an object pointer or a pointer to void, or vice versa.
- It is invalid to convert a pointer to a function of one type to a pointer to a function of a different type without a cast.
- Pointers to functions that have different parameter-type information (including the "old-style" absence of parameter-type information) are different types.

Implicit in the Standard is the notion of *invalid pointers*. In discussing pointers, the Standard typically refers to "a pointer to an object" or "a pointer to a function" or "a null pointer." A special case in address arithmetic allows for a pointer to just past the end of an array. Any other pointer is invalid.

An invalid pointer might be created in several ways. An arbitrary value can be assigned (via a cast) to a pointer variable. (This could even create a valid pointer, depending on the value.) A pointer to an object becomes invalid if the memory containing the object is deallocated or moved by **`realloc`**. Pointer arithmetic can produce pointers outside the range of an array.

Regardless how an invalid pointer is created, any use of it yields undefined behavior. Even assignment, comparison with a null pointer constant, or comparison with itself, might on some systems result in an exception.

Consider a hypothetical segmented architecture on which pointers comprise a segment descriptor and an offset. Suppose that segments are relatively small so that large arrays are allocated in multiple segments. While the segments are valid (allocated, mapped to real memory), the hardware, operating system, or C implementation can make these multiple segments behave like a single object: pointer arithmetic and relational operators use the defined mapping to impose the proper order on the elements of the array. Once the memory is deallocated, the mapping is no longer guaranteed to exist. Use of the segment descriptor might now cause an exception, or the hardware addressing logic might return meaningless data.

6.4 Lexical Elements

The Standard endeavors to bring preprocessing more closely into line with the token orientation of the language proper. To do so requires that at least some information about white space be retained through the early phases of translation (see §5.1.1.2). It also requires that an inverse mapping be defined from tokens back to source characters (see §6.10.3).

6.4.1 Keywords

Several keywords were added in C89: **`const`**, **`enum`**, **`signed`**, **`void`** and **`volatile`**. New in C9X are the keywords **`inline`**, **`restrict`**, **`_Bool`**, **`_Complex`** and **`_Imaginary`**.

Where possible, however, new features have been added by overloading existing keywords, as, for example, **`long double`** instead of **`extended`**. It is recognized that each added keyword will require some existing code that used it as an identifier to be rewritten. No meaningful programs are known to be quietly changed by adding the new keywords.

The keywords **`entry`**, **`fortran`**, and **`asm`** have not been included since they were either never used, or are not portable. Uses of **`fortran`** and **`asm`** as keywords are noted as *common extensions*.

`_Complex` and **`_Imaginary`**, not **`complex`** and **`imaginary`**, are keywords in order that freestanding implementations are not required to support complex. Old code using the names **`complex`** or **`imaginary`** will still work (assuming **`<complex.h>`** is not included), and combined C/C++ implementations will not have to finesse C-only public keywords.

6.4.2 Identifiers

6.4.2.1 General

Because of the linkers available at the time, the C89 Committee made the decision to restrict significance of identifiers with external linkage to six case-insensitive characters. This limit is increased in C9X to 31 case-sensitive characters.

While an implementation is not obliged to remember more than the first 63 characters of an identifier with internal linkage, or the first 31 characters of an identifier with external linkage, the programmer is effectively prohibited from intentionally creating two different identifiers that are the same within the appropriate length. Implementations may therefore store the full identifier; they are not obliged to truncate to 63 or 31.

QUIET CHANGE

> A program that depends on identifiers matching only in the first few characters may change to one with distinct objects for each variant spelling of the identifier.

6.4.2.2 Predefined identifiers

A new feature of C9X: C9X introduces *predefined identifiers*, which have block scope (as distinct from predefined macros which have file scope), and one such predefined identifier, `__func__`, which allows the function name to be used at execution time.

6.4.3 Universal character names

A new feature of C9X: Note that, to allow for Universal Character Names (UCNs), a new production has been added to the grammar that encompasses all forms of identifier elements (basic letter, UCN, or extended character). There was some discussion about the need to require an implementation to handle all digits, Arabic or otherwise, in a similar way. The general feeling was that detecting the "extended digits" might be an undesirable burden for many implementations and should be avoided if possible.

Note that a strictly conforming program may use in identifiers only the extended characters listed in Annex I, and may not begin an identifier with an extended digit.

6.4.4 Constants

In folding and converting constants, an implementation must use at least as much precision as is provided by the target environment. However, it is not required to use exactly the same precision as the target, since this would require a cross compiler to simulate target arithmetic at translation time.

The C89 Committee considered the introduction of structure constants. Although it agreed that structure literals would occasionally be useful, its policy was not to invent new features unless a strong need exists. Since then, such structure constants have been shown to be quite useful, so C9X introduces *compound literals* (see §6.5.2.5).

6.4.4.1 Integer constants

The C90 rule that the default type of a decimal integer constant is either **`int`**, **`long`**, or **`unsigned long`**, depending on which type is large enough to hold the value without overflow, simplifies the use of constants. The choices in C9X are **`int`**, **`long`** and **`long long`**.

C89 added the suffixes **`U`** and **`u`** to specify unsigned numbers. C9X adds **`LL`** to specify **`long long`**.

Unlike decimal constants, octal and hexadecimal constants too large to be **`int`**s are typed as **`unsigned int`** if within range of that type, since it is more likely that they represent bit patterns or masks, which are generally best treated as unsigned, rather than "real" numbers.

Little support was expressed for the old practice of permitting the digits 8 and 9 in an octal constant, so it was dropped in C89.

A proposal to add binary constants was rejected due to lack of precedent and insufficient utility.

Despite a concern that a “lower-case-l” could be taken for the numeral one at the end of a numeric literal, the C89 Committee rejected proposals to remove this usage, primarily on the grounds of sanctioning existing practice.

The rules given for typing integer constants were carefully worked out in accordance with the C89 Committee’s deliberations on integer promotion rules. In C9X, this is clarified and extended with the notion of “rank” (see §6.3.1.1).

QUIET CHANGE IN C89

Unsuffixed integer constants may have different types. In K&R, unsuffixed decimal constants greater than **INT_MAX**, and unsuffixed octal or hexadecimal constants greater than **UINT_MAX** are of type **long**.

QUIET CHANGE IN C9X

Unsuffixed integer constants may have different types in C9X than. Such constants greater than **LONG_MAX** are of type **unsigned long** in C89, but are of type **long long** in C9X.

6.4.4.2 Floating constants

Consistent with existing practice, a floating point constant is defined to have type **double**. Since C89 allows expressions that contain only **float** operands to be performed in **float** arithmetic rather than **double**, a method of expressing explicit **float** constants is desirable. The **long double** type raises similar issues.

The **F** and **L** suffixes have been added to convey type information with floating constants, much like the **L** suffix does for long integers. The default type of floating constants remains **double** for compatibility with prior practice. Lower-case **f** and **l** are also allowed as suffixes.

Note that the run-time selection of the decimal point character by **setlocale** (§7.11.1.1) has no effect on the syntax of C source text: the decimal point character is always period.

Since floating constants are converted to appropriate internal representations at translation time, default rounding direction and precision will be in effect and execution-time exceptions will not be raised, even under the effect of an enabling **FENV_ACCESS** pragma. Library functions such as **strtod** provide execution-time conversion of decimal strings.

A new feature of C9X: C9X adds hexadecimal notation because it more clearly expresses the significance of floating constants. The binary-exponent part is required, instead of optional as it is for decimal notation, to avoid ambiguity resulting from an **f** suffix being mistaken as a hexadecimal digit.

Constants of **long double** type are not generally portable, even among IEC 60559 implementations.

Unlike integers, floating values cannot all be represented directly by hexadecimal constant syntax. A sign can be prefixed for negative numbers and –0. Constant NaNs and infinities are provided through macros in **<math.h>**. Note that **0x1.FFFFFEp128f**, which might appear to be an IEC 60559 single-format NaN, in fact overflows to an infinity in that format.

An alternate approach might have been to represent bit patterns. For example

```
#define  FLT_MAX  0x.7F7FFFFF
```

This would have allowed representation of NaNs and infinities, however numerical values would have been more obscure owing to bias in the exponent and the implicit significand bit, and NaN representations would still not have been portable: even the determination of IEC 60559 quiet NaN vs. signaling NaN is implementation-defined.

The straightforward approach of denoting octal constants by a 0 prefix would have been inconsistent with allowing a leading 0 digit, a moot point as the need for octal floating constants was deemed insufficient.

6.4.4.3 Enumeration constants

Whereas an enumeration variable may have any integer type that correctly represents all its values when widened to **int**, an enumeration constant is only usable as the value of an expression. Hence its type is simply **int**.

6.4.4.4 Character constants

C89 removed the digits 8 and 9 from octal escape sequences (see §6.4.4.1).

The alert escape sequence was added in C89 (see §5.2.2).

Hexadecimal escape sequences beginning with **\x** were adopted in C89, with precedent in several existing implementations. There was little sentiment for providing **\X** as well. The escape sequence extends to the first non-hex-digit character, thus providing the capability of expressing any character constant no matter how large the type **char** is.

The C89 Committee chose to reserve all lower-case letters not currently used for future escape sequences (*undefined behavior*). C9X adds **\U** from Java. All other characters with no current meaning are left to the implementor for extensions (*implementation-defined behavior*). No portable meaning is assigned to multi-character constants or ones containing other than the mandated source character set (*implementation-defined behavior*).

The C89 Committee considered proposals to add the character constant **'\e'** to represent the

ASCII ESC (**'\033'**) character. This proposal was based upon the use of ESC as the initial character of most control sequences in common terminal driving disciplines such as ANSI X3.64. However, this usage has no obvious counterpart in other popular character codes such as EBCDIC.

A programmer merely wishing to avoid having to type **"\033"** to represent the ESC character in an ANSI X3.64 (ISO/IEC 6429) environment, instead of

```
printf("\033[10;10h%d\n", somevalue);
```

may write

```
#define  ESC  "\033"
printf( ESC "[10;10h%d\n", somevalue);
```

Notwithstanding the general rule that literal constants are non-negative[3], a character constant containing one character is effectively preceded with a **(char)** cast and hence may yield a negative value if plain **char** is represented the same as **signed char**. This simply reflects widespread past practice and was deemed too dangerous to change.

QUIET CHANGE IN C89

A constant of the form **'\078'** is valid, but now has different meaning. It now denotes a character constant whose value is the (implementation-defined) combination of the values of the two characters **'\07'** and **'8'**. In some implementations the old meaning is the character whose code is **078** ≡ **0100** ≡ 64.

QUIET CHANGE IN C89

A constant of the form **'\a'** or **'\x'** now may have different meaning. The old meaning, if any, was implementation dependent.

QUIET CHANGE IN C9X

Character literals of the form **'\unnnn'** and **'\Unnnnnnnn'** now have different meanings (see §6.4.3). Note that the escape sequence beginning with **\U** is reserved in C9X, but was not reserved in C89.

An **L** prefix distinguishes wide character constants.

6.4.5 String literals

String literals are not required to be modifiable. This specification allows implementations to share copies of strings with identical text, to place string literals in read-only memory, and to perform

[3] Note that -4 is an expression: unary minus with operand 4.

certain optimizations. However, string literals do not have the type *array of const char* in order to avoid the problems of pointer type checking, particularly with library functions, since assigning a *pointer to const char* to a plain *pointer to char* is not valid. Those members of the C89 Committee who insisted that string literals should be modifiable were content to have this practice designated a common extension (see §K.5.5).

Existing code which modifies string literals can be made strictly conforming by replacing the string literal with an initialized static character array. For instance,

```
char *p, *make_temp(char *str);
/* ... */
p = make_temp("tempXXX");
    // make_temp overwrites literal with unique name
```

can be changed to:

```
char *p, *make_temp(char *str);
/* ... */
{
    static char template[ ] = "tempXXX";
    p = make_temp( template );
}
```

A string can be continued across multiple lines by using the backslash–newline line continuation, but this requires that the continuation of the string start in the first position of the next line. To permit more flexible layout, and to solve some preprocessing problems (see §6.10.3), the C89 Committee introduced string literal concatenation. Two string literals in a row are pasted together, with no null character in the middle, to make one combined string literal. This addition to the C language allows a programmer to extend a string literal beyond the end of a physical line without having to use the backslash–newline mechanism and thereby destroying the indentation scheme of the program. An explicit concatenation operator was not introduced because the concatenation is a lexical construct rather than a run-time operation.

A new feature of C9X: In C89, attempting to concatenate a character string literal and a wide string literal resulted in undefined behavior, primarily because the C89 Committee saw little need to do so. However, there are a number of macros defined by the standard as expanding into character string literals which are frequently needed as wide strings instead (the format specifier macros in **<inttypes.h>** are particularly notable examples, as are the predefined macros __**FILE**__, __**DATE**__, and __**TIME**__). Rather than specifying two forms of each macro, one character string literal and one wide string literal, the Committee decided to go ahead and define concatenating a character string literal and a wide string literal as resulting in a wide string literal. This solves the problem not only for the library and predefined macros, but for similar user-defined macros as well.

Without concatenation:

```
// say the column is this wide
```

```
    alpha = "abcdefghijklm\
nopqrstuvwxyz" ;
```

With concatenation:

```
// say the column is this wide
alpha = "abcdefghijklm"
        "nopqrstuvwxyz";
```

String concatenation can be used to specify a hex-digit character following a hexadecimal escape sequence:

```
char a[] = "\xff" "f" ;
char b[] = {'\xff', 'f', '\0'};
```

These two initializations give **a** and **b** the same string value.

QUIET CHANGE IN C89

A string of the form **"\078"** is valid, but now has different meaning.

QUIET CHANGE IN C89

A string of the form **"\a"** or **"\x"** now has different meaning.

QUIET CHANGE IN C9X

Strings containing the character sequence **\unnnn** or **\Unnnnnnnn** now have different meanings (see §6.4.3). Note that the escape sequence beginning with **\U** is reserved in C9X, but was not reserved in C89.

QUIET CHANGE IN C89

It is neither required nor forbidden that identical string literals be represented by a single copy of the string in memory; a program depending upon either scheme may behave differently.

An **L** prefix distinguishes wide string literals. A prefix rather than a suffix notation was adopted so that a translator can know at the start of the processing of a string literal whether it is dealing with ordinary or wide characters.

6.4.6 Punctuators

C89 added the punctuator **...** (ellipsis) to denote a variable number of trailing arguments in a function prototype (see §6.7.5.3); and C9X extends this to function-like macros (see §6.10.3).

6.4.7 Header names

Header names in **#include** directives obey distinct tokenization rules; hence they are identified as distinct tokens. Attempting to treat quote-enclosed header names as string literals creates a contorted description of preprocessing, and the problems of treating angle bracket-enclosed header names as a sequence of C tokens is even more severe.

6.4.8 Preprocessing numbers

The notion of preprocessing numbers was introduced to simplify the description of preprocessing. It provides a means of talking about the tokenization of strings that look like numbers, or initial substrings of numbers, prior to their semantic interpretation. In the interests of keeping the description simple, occasional spurious forms are scanned as preprocessing numbers. For example, **0x123E+1** is a single token under the rules. The C89 Committee felt that it was better to tolerate such anomalies than burden the preprocessor with a more exact, and exacting, lexical specification. It felt that this anomaly was no worse than the principle under which the characters **a+++++b** are tokenized as **a ++ ++ + b** (an invalid expression), even though the tokenization **a ++ + ++ b** would yield a syntactically correct expression. In both cases, exercise of reasonable precaution in coding style avoids surprises.

A new feature of C9X: C9X replaces *nondigit* with *identifier-nondigit* in the grammar to allow the token pasting operator, **##**, to work as expected. Given the code

```
#define  mkident(s)  s ## 1m
/* ... */
int mkident(int) = 0;
```

if an identifier is passed to the **mkident** macro, then **1m** is parsed as a single pp-number, a valid single identifier is produced by the **##** operator, and nothing harmful happens. But consider a similar construction that might appear using Greek script:

```
#define  μk(p)  p ## 1μ
/* ... */
int μk(int) = 0;
```

For this code to work, **1μ** must be parsed as only one pp-token. Restricting pp-numbers to only the basic letters would break this.

6.4.9 Comments

The C89 Committee considered proposals to allow comments to nest. The main argument for nesting comments is that it would allow programmers to "comment out" code. The C89 Committee rejected this proposal on the grounds that comments should be used for adding documentation to a program, and that preferable mechanisms already exist for source code exclusion. For example,

```
#if 0
/* code to be excluded */
#endif
```

Preprocessing directives such as this prevent the enclosed code from being scanned by later translation phases. Bracketed material can include comments and other nested regions of bracketed code.

Another way of accomplishing these goals is with an **if** statement:

```
if (0) {
    /* code to be excluded */
}
```

Many modern compilers will generate no code for this **if** statement.

// comments were added for C9X due to their utility and widespread existing practice, especially in dual C/C++ translators.

QUIET CHANGE IN C9X

In certain unusual situations, code could have different semantics for C90 and C9X, for example

```
a = b //*divisor:*/ c
+ d;
```

In C90 this was equivalent to

```
a = b / c + d;
```

but in C9X it is equivalent to

```
a = b + d;
```

6.5 Expressions

Several closely related topics are involved in the precise specification of expression evaluation: *precedence*, *associativity*, *grouping*, *sequence points*, *agreement points*, *order of evaluation*, and *interleaving*.

The rules of *precedence* are encoded into the syntactic rules for each operator. For example, the syntax for *additive-expression* includes the rule

additive-expression + *multiplicative-expression*

which implies that **a+b*c** parses as **a+(b*c)**. The rules of *associativity* are similarly encoded into the syntactic rules. For example, the syntax for *assignment-expression* includes the rule

unary-expression assignment-operator assignment-expression

which implies that **a=b=c** parses as **a=(b=c)**.

With rules of precedence and associativity thus embodied in the syntax rules, the Standard specifies, in general, the *grouping* (association of operands with operators) in an expression.

K&R describes C as a language in which the operands of successive identical commutative associative operators can be regrouped. The C89 Committee decided to remove this license from the Standard, thus bringing C into accord with most other major high-level languages.

This change was motivated primarily by the desire to make C more suitable for floating point programming. Floating point arithmetic does not obey many of the mathematical rules that real arithmetic does. For instance, the two expressions **(a+b)+c** and **a+(b+c)** may well yield different results. Suppose that **b** is greater than 0, **a** equals **-b**, and **c** is positive but substantially smaller than **b**. (That is, suppose **c/b** is less than **DBL_EPSILON**.) Then **(a+b)+c** is **0+c**, or **c**, while **a+(b+c)** equals **a+b**, or 0. That is to say, floating point addition and multiplication are not associative.

K&R's rule imposes a high cost on translation of numerical code to C. Much numerical code is written in Fortran, which does provide a no-regrouping guarantee; indeed, this is the normal semantic interpretation in most high-level languages other than C. K&R's advice, "rewrite using explicit temporaries," is burdensome to those with tens or hundreds of thousands of lines of code to convert, a conversion which in most other respects could be done automatically.

Loss of the regrouping rule does not in fact prohibit much regrouping of integer expressions. The bitwise logical operators can be arbitrarily regrouped since any regrouping gives the same result *as if* the expression had not been regrouped. This is also true of integer addition and multiplication in implementations with two's-complement arithmetic and silent wraparound on overflow. Indeed, in any implementation, regroupings which do not introduce overflows behave *as if* no regrouping had occurred. (Results may also differ in such an implementation if the expression as written results in overflows; but in such a case the behavior is undefined, so any regrouping couldn't be any worse.)

The types of lvalues that may be used to access an object have been restricted so that an optimizer is not required to make worst-case aliasing assumptions (see also §6.7.3.1)

In practice, aliasing arises with the use of pointers. A contrived example to illustrate the issues is

```
int a;

void f(int * b)
{
```

```
    a = 1;
    *b = 2;
    g(a);
}
```

It is tempting to generate the call to **g** as if the source expression were **g(1)**, but **b** might point to **a**, so this optimization is not safe. On the other hand, consider

```
int a;
void f( double * b )
{
    a = 1;
    *b = 2.0;
    g(a);
}
```

Again the optimization is incorrect only if **b** points to **a**. However, this would only have come about if the address of **a** were somewhere cast to **double***. The C89 Committee has decided that such dubious possibilities need not be allowed for.

In principle, then, aliasing only need be allowed for when the lvalues all have the same type. In practice, the C89 Committee recognized certain prevalent exceptions:

- The lvalue types may differ in signedness. In the common range, a signed integer type and its unsigned variant have the same representation; and it was felt that an appreciable body of existing code is not "strictly typed" in this area.
- Character pointer types are often used in the bytewise manipulation of objects; a byte stored through such a character pointer may well end up in an object of any type.
- A qualified version of the object's type, though formally a different type, provides the same interpretation of the value of the object.

Structure and union types also have problematic aliasing properties:

```
struct fi{ float f; int i;};

void f( struct fi * fip, int * ip )
{
    static struct fi a = {2.0F, 1};
    *ip = 2;
    *fip = a;
    g(*ip);

    *fip = a;
    *ip = 2;
```

```
        g(fip->i);
    }
```

It is not safe to optimize the first call to **g** as **g(2)**, or the second as **g(1)**, since the call to **f** could quite legitimately have been

```
struct fi x;
f( &x, &x.i );
```

These observations explain the other exception to the same-type principle.

An implementation that is able to multiply two **double** operands and produce a **float** result in just one machine instruction might contract the multiplication and assignment in

```
float f;
double d1, d2;
/* ... */
f = d1 * d2;
```

Other examples of potential contraction operators include compound assignments (**+=**, **-=**, etc.), ternary add ($x + y + z$), and multiply-add ($xy + z$).

Contractions can lead to subtle anomalies even while increasing accuracy. The value of C expressions like **a * b + c * d** will depend on how the translator uses a contracted multiply-add. The Intel i860's multiply-add is slightly more problematic: since it keeps a wide but partial product, **a * b + z** may differ from **c * d + z** even though the exact mathematical products $a \times b$ and $c \times d$ are equal; the result depends not just on the mathematical result and the format, as ordinarily expected for error analysis, but also on the particular values of the operands.

The programmer can control the use of fused multiply-adds by disabling use of contractions with an **FP_CONTRACT** pragma and using the **fma** function where desired. The extra accuracy of a fused multiply-add, which produces a result with just one rounding, can be exploited for simpler and faster code.

6.5.1 Primary expressions

A primary expression may be **void** (parenthesized call to a function returning **void**), a function designator (identifier or parenthesized function designator), an lvalue (identifier or parenthesized lvalue), or simply a value expression. Constraints ensure that a **void** primary expression is no part of a further expression, except that a void expression may be cast to **void**, may be the second or third operand of a conditional operator, or may be an operand of a comma operator.

6.5.2 Postfix operators

6.5.2.1 Array subscripting

The C89 Committee found no reason to disallow the symmetry that permits **a[i]** to be written as **i[a]**.

The syntax and semantics of multidimensional arrays follow logically from the definition of arrays and the subscripting operation. The material in the Standard on multidimensional arrays introduces no new language features, but clarifies the C treatment of this important abstract data type.

6.5.2.2 Function calls

Pointers to functions may be used either as **(*pf)()** or as **pf()**. The latter construct, not sanctioned in K&R, appears in some present versions of C, is unambiguous, invalidates no old code, and can be an important shorthand. The shorthand is useful for packages that present only one external name, which designates a structure full of pointers to objects and functions: member functions can be called as **graphics.open(file)** instead of **(*graphics.open)(file)**.

The treatment of function designators can lead to some curious, but valid, syntactic forms. Given the declarations

```
int f(), (*pf)();
```

then all of the following expressions are valid function calls:

```
(&f)(); f(); (*f)(); (**f)(); (***f)();
pf();   (*pf)(); (**pf)(); (***pf)();
```

The first expression on each line was discussed in the previous paragraph. The second is conventional usage. All subsequent expressions take advantage of the implicit conversion of a function designator to a pointer value, in nearly all expression contexts. The C89 Committee saw no real harm in allowing these forms; outlawing forms like **(*f)()**, while still permitting ***a** for **a[]**, simply seemed more trouble than it was worth.

A new feature of C9X: The rule for implicit declaration of functions has been removed in C9X. The effect is to guarantee the production of a diagnostic that will catch an additional category of programming errors. After issuing the diagnostic, an implementation may choose to assume an implicit declaration and continue translation in order to support existing programs that exploited this feature.

For compatibility with past practice, all argument promotions occur as described in K&R in the absence of a prototype declaration, including the not always desirable promotion of **float** to **double**. A prototype gives the implementor explicit license to pass a **float** as a **float** rather than a **double**, or a **char** as a **char** rather than an **int**, or an argument in a special register, etc. If the definition of a function in the presence of a prototype would cause the function to expect other than the default promotion types, then clearly the calls to this function must also be made in the presence of a compatible prototype.

To clarify this and other relationships between function calls and function definitions, the Standard

describes the relationship between a function call or definition which does occur in the presence of a prototype and one that does not.

Thus a prototyped function with no "narrow" types and no variable argument list must be callable in the absence of a prototype, since the types actually passed in a call are equivalent to those in the explicit function definition prototype. This constraint is necessary to retain compatibility with past usage of library functions (see §7.1.4).

This provision constrains the latitude of an implementor because the parameter passing conventions of prototype and non-prototype function calls must be the same for functions accepting a fixed number of promoted arguments. Implementations in environments where efficient function calling mechanisms are available must, in effect, use the efficient calling sequence either in all "fixed argument list" calls or in none. Since efficient calling sequences often do not allow for variable argument functions, the fixed part of a variable argument list may be passed in a completely different fashion than in a fixed argument list with the same number and types of arguments.

The existing practice of omitting trailing parameters in a call if it is known that the parameters will not be used has consistently been discouraged. Since omission of such parameters creates a mismatch between the call and the declaration, the behavior in such cases is undefined, and a maximally portable program will avoid this usage. Hence an implementation is free to implement a function calling mechanism for fixed argument lists which would (perhaps fatally) fail if the wrong number or type of arguments were to be provided.

Strictly speaking then, calls to **`printf`** are obliged to be in the scope of a prototype (as by **`#include <stdio.h>`**), but implementations are not obliged to fail on such a lapse. (The behavior is *undefined.*)

6.5.2.3 Structure and union members

Since the language now permits structure parameters, structure assignment and functions returning structures, the concept of a *structure expression* is now part of the C language. A structure value can be produced by an assignment, by a function call, by a comma operator expression, by a conditional operator expression, or by a compound literal:

```
s1 = (s2 = s3)
sf(x)
(x, s1)
x ? s1 : s2
```

Except for the case of the compound literal, the result is *not* an lvalue; hence it cannot be assigned to nor can its address be taken.

Similarly, **`x.y`** is an lvalue only if **`x`** is an lvalue. Thus none of the following valid expressions are lvalues.

```
sf(3).a
(s1=s2).a
```

```
((i==6)?s1:s2).a
(x,s1).a
```

Even when **x.y** is an lvalue, it might not be modifiable:

```
const struct S s1;
s1.a = 3;       // invalid
```

The Standard requires that an implementation diagnose a *constraint error* in the case that the member of a structure or union designated by the identifier following a member selection operator (**.** or **->**) does not appear in the type of the structure or union designated by the first operand. K&R is unclear on this point.

6.5.2.4 Postfix increment and decrement operators

The C89 Committee did not endorse the practice in some implementations of considering post-increment and post-decrement operator expressions to be lvalues.

Increment and decrement operators are not defined for complex or imaginary types. Given the regular definition, they would be surprising for imaginary types, as the operators would have no effect. It is sometimes desirable to use the same source code with types being complex or imaginary depending on the implementation. In this scenario, increment or decrement of the complex 0+*iy* would differ from increment or decrement of the imaginary *iy*. Allowing increment and decrement of complex but not imaginary objects would not be helpful here either.

6.5.2.5 Compound literals

A new feature of C9X: Compound literals provide a mechanism for specifying constants of aggregate or union type. This eliminates the requirement for temporary variables when an aggregate or union value will only be needed once.

Compound literals integrate easily into the C grammar and do not impose any additional run-time overhead on a user's program. They also combine well with designated initializers (§6.7.8) to form an even more convenient aggregate or union constant notation. Their initial C implementation appeared in a compiler by Ken Thompson at AT&T Bell Laboratories.

6.5.3 Unary operators

6.5.3.1 Prefix increment and decrement operators

See §6.5.2.4.

6.5.3.2 Address and indirection operators

Some implementations have not allowed the **&** operator to be applied to an array or a function. (The construct was permitted in early versions of C, then later made optional.) The C89 Committee

endorsed the construct since it is unambiguous, and since data abstraction is enhanced by allowing the important **&** operator to apply uniformly to any addressable entity.

6.5.3.3 Unary arithmetic operators

Unary plus was adopted by the C89 Committee from several implementations, for symmetry with unary minus.

The bitwise complement operator **~**, and the other bitwise operators, have now been defined arithmetically for unsigned operands. Such operations are well-defined because of the restriction of integer representations to "binary numeration systems."

6.5.3.4 The `sizeof` operator

It is fundamental to the correct usage of functions such as **`malloc`** and **`fread`** that **`sizeof(char)`** be exactly one. In practice, this means that a *byte* in C terms is the smallest unit of storage, even if this unit is 36 bits wide; and all objects are composed of an integer number of these smallest units.

C89, like K&R, defined the result of the **`sizeof`** operator to be a constant of an unsigned integer type. Common implementations, and common usage, have often assumed that the resulting type is **`int`**. Old code that depends on this behavior has never been portable to implementations that define the result to be a type other than **`int`**. The C89 Committee did not feel it was proper to change the language to protect incorrect code.

The type of **`sizeof`**, whatever it is, is published (in the library header **`<stddef.h>`**) as **`size_t`**, since it is useful for the programmer to be able to refer to this type. This requirement implicitly restricts **`size_t`** to be a synonym for an existing unsigned integer type. Note also that, although **`size_t`** is an unsigned type, **`sizeof`** does not involve any arithmetic operations or conversions that would result in modulus behavior if the size is too large to represent as a **`size_t`**, thus quashing any notion that the largest declarable object might be too big to span even with an **`unsigned long`** in C89 or **`uintmax_t`** in C9X. This also restricts the maximum number of elements that may be declared in an array, since for any array **`a`** of **`N`** elements,

```
N == sizeof(a)/sizeof(a[0])
```

Thus **`size_t`** is also a convenient type for array sizes, and is so used in several library functions.

C89 specified that **`sizeof`**'s operand can be any value except a bit field, a void expression, or a function designator. This generality allows for interesting environmental inquiries. Given the declarations

```
int  *p, *q;
```

these expressions determine the size of the type used for:

```
sizeof(F(x))     // ... F's return value
sizeof(p-q)      // ... pointer difference
```

(The last type is available as **ptrdiff_t** in **<stddef.h>**.)

With the addition of variable length arrays (§6.7.5.2) in C9X, the **sizeof** operator is a constant expression only if the type of the operand is not a variable length array type. However, the notion of "size" is consistently maintained for important operations such as pointer increment, subscripting, and pointer difference. That is, it is still possible to determine the number of elements in a variable length array with

```
sizeof(vla) / sizeof(vla[0])
```

Finally, **sizeof** can still be used in an argument to the **malloc** function.

QUIET CHANGE IN C9X

With the introduction of the **long long** and extended integer types, the **sizeof** operator may yield a value that exceeds the range of a **long**.

6.5.4 Cast operators

A **(void)** cast is explicitly permitted, more for documentation than for utility.

Nothing portable can be said about casting integers to pointers, or vice versa, since the two are now incommensurate.

The definition of these conversions adopted in the Standard resembles that in K&R, but with several significant differences. K&R required that a pointer successfully converted to an integer must be guaranteed to be convertible back to the same pointer. This integer-to-pointer conversion is now specified as *implementation-defined*. While a high-quality implementation would preserve the same address value whenever possible, it was considered impractical to require that the identical representation be preserved. The C89 Committee noted that, on some current machine implementations, identical representations are required for efficient code generation for pointer comparisons and arithmetic operations.

The conversion of an integer constant expression with the value 0 to a pointer is defined similarly to K&R. The resulting pointer must not address any object, must appear to be equal to an integer value of 0, and may be assigned to or compared for equality with any other pointer. This definition does not necessarily imply a representation by a bit pattern of all zeros: an implementation could, for instance, use some address which causes a hardware trap when dereferenced.

The type **char** must have the least strict alignment of any type, so **char*** has often been used as a portable type for representing arbitrary object pointers. This usage creates an unfortunate confusion between the ideas of *arbitrary pointer* and *character or string pointer*. The new type **void***, which has the same representation as **char***, is therefore preferable for arbitrary pointers.

It is possible to cast a pointer of some qualified type (§6.7.3) to an unqualified version of that type. Since the qualifier defines some special access or aliasing property, however, any dereference of the cast pointer results in *undefined behavior*.

Because of the requirements of §6.3.1.5, a cast of an expression with a floating-point type to a smaller floating-point type (for example, **`double`** to **`float`**) cannot be optimized away.

6.5.5 Multiplicative operators

In C89, division of integers involving negative operands could round upward or downward in an implementation-defined manner; the intent was to avoid incurring overhead in run-time code to check for special cases and enforce specific behavior. In Fortran, however, the result will always truncate toward zero, and the overhead seems to be acceptable to the numeric programming community. Therefore, C9X now requires similar behavior, which should facilitate porting of code from Fortran to C. The table in §7.10.6.2 of this document illustrates the required semantics.

The C89 Committee rejected extending the **`%`** operator to work on floating types as such usage would duplicate the facility provided by **`fmod`** (see §7.12.10.1).

6.5.6 Additive operators

As with the **`sizeof`** operator (see §6.5.3.4), implementations have taken different approaches in defining a type for the difference between two pointers. It is important that this type be signed in order to obtain proper algebraic ordering when dealing with pointers within the same array. However, the magnitude of a pointer difference can be as large as the size of the largest object that can be declared; and since that is an unsigned type, the difference between two pointers can cause an overflow on some implementations.

QUIET CHANGE IN C9X

> With the introduction of the **`long long`** and extended integer types, the subtraction of pointers may return a value that exceeds the range of a **`long`**.

The C9X variable length array type (§6.7.5.2) does not affect the semantics of pointer difference. Similarly, incrementing a pointer to a variable length array increments according to the number of elements in the array just like a fixed length array.

```
void ptr_to_vla_incr(int n)
{
    int a[2][n];
    int (*p)[n] = a;

    p++;  // p == &a[1]
```

```
    // ...
}
```

If the declarations of **a** and **p** used an integer constant instead of the parameter **n**, then the increment of pointer **p** still results in **p** pointing to the second row of **a**. That is, **p** is incremented according to the number of elements in each row of **a**, and it doesn't matter whether **a** is a variable length array or a fixed length array. The expected behavior is preserved.

The type of *pointer minus pointer* is defined to be **int** in K&R. The Standard defines the result of this operation to be a signed integer, the size of which is implementation-defined. The type is published as **ptrdiff_t**, in the standard header **<stddef.h>**. Old code recompiled by a conforming compiler may no longer work if the implementation defines the result of such an operation to be a type other than **int** and if the program depended on the result to be of type **int**. This behavior was considered by the C89 Committee to be correctable. Overflow was considered not to break old code since it was undefined by K&R. Mismatch of types between actual and formal argument declarations is correctable by including a properly defined function prototype in the scope of the function invocation.

An important endorsement of widespread practice is the requirement that a pointer can always be incremented to *just past* the end of an array, with no fear of overflow or wraparound:

```
SOMETYPE array[SPAN];
// ...
for (p = &array[0]; p < &array[SPAN]; p++)
```

This stipulation merely requires that every object be followed by one byte whose address is representable. That byte can be the first byte of the next object declared for all but the last object located in a contiguous segment of memory. (In the example, the address **array + SPAN** must address a byte following the highest element of **array**.) Since the pointer expression **p+1** need not, and should not, be dereferenced, it is unnecessary to leave room for a complete object of size **sizeof(*p)**.

In the case of **p-1**, on the other hand, an entire object *would* have to be allocated prior to the array of objects that **p** traverses, so decrement loops that run off the bottom of an array can fail. This restriction allows segmented architectures, for instance, to place objects at the start of a range of addressable memory.

6.5.7 Bitwise shift operators

The description of shift operators in K&R suggests that shifting by a **long** count should force the left operand to be widened to **long** before being shifted. A more intuitive practice, endorsed by the C89 Committee, is that the type of the shift count has no bearing on the type of the result.

QUIET CHANGE IN C89

Shifting by a **long** count no longer coerces the shifted operand to **long**.

The C89 Committee affirmed the freedom in implementation granted by K&R in not requiring the signed right shift operation to sign extend, since such a requirement might slow down fast code and since the usefulness of sign extended shifts is marginal. (Shifting a negative two's-complement integer arithmetically right one place is *not* the same as dividing by two!)

6.5.8 Relational operators

For an explanation of why the pointer comparison of the object pointer **P** with the pointer expression **P+1** is always safe, see Rationale §6.5.6.

Some mathematical practice would be supported by defining the relational operators for complex operands so that **z1** *op* **z2** would be true if and only if both **creal(z1)** *op* **creal(z2)** and **cimag(z1) == cimag(z2)**. Believing such use to be uncommon, the C9X Committee voted against including this specification.

6.5.9 Equality operators

The C89 Committee considered, on more than one occasion, permitting comparison of structures for equality. Such proposals foundered on the problem of holes in structures. A byte-wise comparison of two structures would require that the holes assuredly be set to zero so that all holes would compare equal, a difficult task for automatic or dynamically allocated variables. The possibility of union-type elements in a structure raises insuperable problems with this approach. Without the assurance that all holes were set to zero, the implementation would have to be prepared to break a structure comparison into an arbitrary number of member comparisons; a seemingly simple expression could thus expand into a substantial stretch of code, which is contrary to the spirit of C.

In pointer comparisons, one of the operands may be of type **void***. In particular, this allows **NULL**, which can be defined as **(void*)0**, to be compared to any object pointer.

6.5.15 Conditional operator

The syntactic restrictions on the middle operand of the conditional operator have been relaxed to include more than just *logical-OR-expression*: several extant implementations have adopted this practice.

The type of a conditional operator expression can be **void**, a structure, or a union; most other operators do not deal with such types. The rules for balancing type between pointer and integer have, however, been tightened, since now only the constant 0 can portably be coerced to a pointer.

The Standard allows one of the second or third operands to be of type **void***, if the other is a pointer type. Since the result of such a conditional expression is **void***, an appropriate cast must be used.

6.5.16 Assignment operators

Certain syntactic forms of assignment operators have been discontinued, and others tightened up.

The storage assignment need not take place until the next sequence point. As a consequence, a straightforward syntactic test for ambiguous expressions can be stated. Some definitions: A *side effect* is a storage to any data object, or a read of a **`volatile`** object. An *ambiguous expression* is one whose value depends upon the order in which side effects are evaluated. A *pure function* is one with no side effects; an impure function is any other. A *sequenced expression* is one whose major operator defines a sequence point: comma, **`&&`**, **`||`**, or conditional operator; an *unsequenced expression* is any other. We can then say that an unsequenced expression may be ambiguous if more than one operand invokes any impure function, or if more than one operand contains an lvalue referencing the same object and one or more operands specify a side-effect to that object. Further, any expression containing an ambiguous sub-expression is ambiguous.

The optimization rules for factoring out assignments can also be stated. Let **`X(i,S)`** be an expression which contains no impure functions or sequenced operators, and suppose that **`X`** contains a storage **`S(i)`** to **`i`** which sets **`i`** to **`Snew(i)`** and returns **`Sval(i)`**. The possible storage expressions are

`S(i):`	`Sval(i):`	`Snew(i):`
`++i`	`i+1`	`i+1`
`i++`	`i`	`i+1`
`--i`	`i-1`	`i-1`
`i--`	`i`	`i-1`
`i = y`	`y`	`y`
`i` *`op`*`= y`	`i` *`op`* `y`	`i` *`op`* `y`

Then **`X(i,S)`** can be replaced by either

```
(T = i, i = Snew(i), X(T,Sval))
```

or

```
(T = X(i,Sval), i = Snew(i), T)
```

provided that neither **`i`** nor **`y`** have side effects themselves.

6.5.16.1 Simple assignment

Structure assignment was added: its use was foreshadowed even in K&R, and many existing implementations already support it.

The rules for type compatibility in assignment also apply to argument compatibility between actual argument expressions and their corresponding parameter types in a function prototype.

An implementation need not correctly perform an assignment between overlapping operands.

Overlapping operands occur most naturally in a union, where assigning one field to another is often desirable to effect a type conversion in place. The assignment may well work properly in all simple cases, but it is not maximally portable. Maximally portable code should use a temporary variable as an intermediate in such an assignment.

6.5.16.2 Compound assignment

The importance of requiring that the left operand lvalue be evaluated only once is not a question of efficiency, although that is one compelling reason for using the compound assignment operators. Rather, it is to assure that any side effects of evaluating the left operand are predictable.

Assignment operators of the form **=+**, described as *old fashioned* even in K&R, were dropped in C89. The form **+=** is now defined to be a single token, not two, so no white space is permitted within it. No compelling case could be made for permitting such white space.

QUIET CHANGE IN C89

Expressions of the form **x=-3** change meaning with the loss of the old-style assignment operators.

6.5.17 Comma operator

The left operand of a comma operator may be **void**, since only the right hand operator is relevant to the type of the expression.

6.6 Constant expressions

To clarify existing practice, several varieties of constant expression have been identified.

The expression following **#if** (§6.10.1) must expand to integer constants, character constants, the special operator **defined**, and operators with no side effects. Environmental inquiries can be made only using the macros defined in the standard headers, **<limits.h>**, **<stdint.h>**, etc.

Character constants, when evaluated in **#if** expressions, may be interpreted in the source character set, the execution character set, or some other implementation-defined character set. This latitude reflects the diversity of existing practice, especially in cross-compilers.

An *integer constant expression* must involve only numbers knowable at translation time, and operators with no side effects. Casts and the **sizeof** operator whose operand does not have a variable length array type (§6.7.5.2) may be used to interrogate the execution environment.

Static initializers include integer constant expressions, along with floating constants and simple addressing expressions. An implementation must accept arbitrary expressions involving floating and integer numbers and side effect-free operators in arithmetic initializers, but it is at liberty to turn such initializers into executable code which is invoked prior to program startup. This scheme

might impose some requirements on linkers or runtime library code in some implementations.

The translation environment must not produce a less accurate value for a floating-point initializer than the execution environment, but it is at liberty to do better. Thus a static initializer may well be slightly different from the same expression computed at execution time. However, while implementations are certainly *permitted* to produce exactly the same result in translation and execution environments, *requiring* this was deemed to be an intolerable burden on many cross-compilers.

QUIET CHANGE IN C89

A program that uses **`#if`** expressions to determine properties of the execution environment may now get different answers.

QUIET CHANGE IN C9X

Due to the introduction of new types, the preprocessor arithmetic must be performed using the semantics of either **`intmax_t`** or **`uintmax_t`** defined in **`<stdint.h>`**. This is a quiet change for cross-compilation implementations because C89 did not mandate that translation-time arithmetic have the properties of the execution environment, but C9X does.

6.7 Declarations

The C89 Committee decided that empty declarations are invalid, except for a special case with tags (see §6.7.2.3) and the case of enumerations such as **`enum {zero,one};`** (see §6.7.2.2). While many seemingly silly constructs are tolerated in other parts of the language in the interest of facilitating the machine generation of C, empty declarations were considered sufficiently easy to avoid.

6.7.1 Storage-class specifiers

Because the address of a register variable cannot be taken, objects of storage class **`register`** effectively exist in a space distinct from other objects. (Functions occupy yet a third address space.) This makes them candidates for optimal placement, the usual reason for declaring registers; but it also makes them candidates for more aggressive optimization.

The practice of representing register variables as wider types (as when **`register char`** is quietly changed to **`register int`**) is no longer acceptable.

6.7.2 Type specifiers

Several new type specifiers were added to C89: **`signed`**, **`enum`**, and **`void`**. **`long float`** was retired and **`long double`** was added, along with many integer types.

A new feature of C9X: several new type specifiers were added to C9X: **_Bool**, **_Complex** and **_Imaginary**, along with the related types.

A new feature of C9X: C9X adds a new integer data type, **long long**, as consolidation of prior art, whose impetus has been three hardware developments. First, disk density and capacity used to double every 3 years, but after 1989 has quadrupled every 3 years, yielding low-cost, physically small disks with large capacities. Although a fixed size for file pointers and file system structures is necessary for efficiency, eventually it is overtaken by disk growth, and limits need to be expanded. In the 1970s, 16-bit C (for the Digital PDP-11) first represented file information with 16-bit integers, which were rapidly obsoleted by disk progress. People switched to a 32-bit file system, first using **int[2]** constructs which were not only awkward, but also not efficiently portable to 32-bit hardware.

To solve the problem, the **long** type was added to the language, even though this required C on the PDP-11 to generate multiple operations to simulate 32-bit arithmetic. Even as 32-bit minicomputers became available alongside 16-bit systems, people still used **int** for efficiency, reserving **long** for cases where larger integers were truly needed, since **long** was noticeably less efficient on 16-bit systems. Both **short** and **long** were added to C, making **short** available for 16 bits, **long** for 32 bits, and **int** as convenient for performance. There was no desire to lock the numbers 16 or 32 into the language, as there existed C compilers for at least 24- and 36-bit CPUs, but rather to provide names that could be used for 32 bits as needed.

PDP-11 C might have been re-implemented with **int** as 32-bits, thus avoiding the need for **long**; but that would have made people change most uses of **int** to **short** or suffer serious performance degradation on PDP-11s. In addition to the potential impact on source code, the impact on existing object code and data files would have been worse, even in 1976. By the 1990s, with an immense installed base of software, and with widespread use of dynamic linked libraries, the impact of changing the size of a common data object in an existing environment is so high that few people would tolerate it, although it might be acceptable when creating a new environment. Hence, many vendors, to avoid namespace conflicts, have added a 64-bit integer to their 32-bit C environments using a new name, of which **long long** has been the most widely used.

C9X has therefore adopted **long long** as the name of an integer type with at least 64 bits of precision. People can and do argue about the particular choice of name, but it has been difficult to pick a clearly better name early enough, and by now it is fairly common practice, and may be viewed as one of the *least bad* choices.

To summarize this part: 32-bit CPUs are coming to need clean 64-bit integers, just as 16-bit CPUs came to need 32-bit integers, and the need for wider integers happens irrespective of other CPUs. Thus, 32-bit C has evolved from a common `ILP32` model (**int**, **long** and pointers are 32 bits) to `ILP32LL` (`ILP32` + 64-bit **long long**), and this still runs on 32-bit CPUs with sequences to emulate 64-bit arithmetic.

In the second and third interrelated trends, DRAM memories continue to quadruple in size every

3 years, and 64-bit microprocessors started to be widely used in 1992. By 1995, refrigerator-sized, microprocessor-based servers were being sold with 8GB to 16GB of memory, which required more than 32 bits for straightforward addressing. However, many 64-bit microprocessors are actually used in video games, X-Terminals, network routers, and other applications where pointer size is less important than performance for larger integers.

The memory trend encourages a C programming model in which pointers are enlarged to 64 bits (called `*P64`), of which the consensus choice seems to be `LP64` (**`long`**, pointers and **`long long`** are 64 bits; **`int`** is 32 bits), with **`long long`** in some sense redundant, just as **`long`** was on the 32-bit VAX. It is fairly difficult to mix this object code with the `ILP32` model, and so it is a new environment to which people must port code, but for which they receive noticeable benefits: they can address large memories, and file pointers automatically are enlarged to 64 bits. There do exist, of course, 32-bit CPUs with more-than-32-bit addressing, although C environments become much more straightforward on 64-bit CPUs with simple, flat addressing. In practice, people do not move from `ILP32LL` to `LP64` unless they have no choice or gain some clear benefit.

If people only consider `LP64` in isolation, **`long`** is 64 bits, and there seems no need for **`long long`**, just as the VAX 32-bit environment really did not need **`long`**. However, this view ignores the difficulty of getting compilers, linkers, debuggers, libraries, etc., to exist for `LP64`. In practice, these programs need to deal with 64-bit integers long before an `LP64` environment exists, in order to bootstrap, and later support, all these tools. Put another way, people must:

1. Using **`int[2]`**, upgrade compilers and a minimal set of tools to compile and debug code that uses **`long long`**.

2. Recode the compilers and all of the tools to actually use **`long long`**.

This ends up with a set of tools that run as `ILP32LL`, on existing 32-bit CPUs and new 64-bit CPUs, and can compile code to either `ILP32LL` or `LP64`. This is yet another reason where **`long long`** is important, not for the `LP64` model, but for the tools that support that model.

Most 64-bit micros can, and for commercial reasons must, continue to run existing `ILP32LL` object programs, alongside any new `LP64` programs. For example, database server processes often desire `LP64` to access large memory pools, but the implementers prefer to leave the client code as `ILP32` so that it can run on existing 32-bit CPUs as well, and where `LP64` provides no obvious value.

In mixed environments, it is of course very useful for programs to share data structures, and specifically for 32-bit programs to be able to cleanly describe aligned 64-bit integers, and in fact for it to be easy to write structure definitions whose size and alignment are identical between `ILP32LL` and `LP64`. This can be straightforwardly done using **`int`** and **`long long`**, just as it was doable in the 1970s via **`short`** and **`long`**.

Finally, one more important case occurs, in which people want performance benefits of 64-bit CPUs, while wishing to maintain source compatibility, but not necessarily binary compatibility,

with related 32-bit CPUs. In embedded control and consumer products, people have little interest in 64-bit pointers, but they often like 64-bit integer performance for bit manipulation, memory copies, encryption, and other tasks. They like `ILP32LL`, but with **`long long`** compiled to use 64-bit registers, rather than being simulated via 32-bit registers. While this is not binary-compatible with existing `ILP32LL` binaries, it is source-compatible; and it runs faster and uses less space than `LP64`, both of which are important in these markets. It is worth noting that of the many millions of 64-bit CPUs that exist, a very large majority are actually used in such applications rather than traditional computer systems.

Thus, there are 3 choices, all of which have been done already, and different customers choose different combinations:

`ILP32LL`, compiled 32-bit only, runs on 32- and 64-bit CPUs

- Needs **`long long`** to express 64-bit integers without breaking existing source and object code badly.

`LP64`, runs on 64-bit CPUs

- Does not need **`long long`** in isolation, but needed its earlier `ILP32LL` tools to have **`long long`** for sensible bootstrapping and later support.

`ILP32LL`, compiled to 64-bit registers, runs on 64-bit CPUs

- Wants **`long long`** to express 64-bit integers and get better performance, and still have source code that runs on related 32-bit CPUs.

A new integer data type is needed that can be used to express 64-bit integers efficiently and portably among 32- and 64-bit systems. It must be a new name to avoid a disastrous set of incompatibilities with existing 32-bit environments since one cannot safely change **`long`** to 64 bits and mix with existing object code. It is needed to deal with disk file size increases, but also to help bootstrap to 64-bit environments, and then wider, so that many programs can be compiled to exactly one binary that runs on both 32- and 64-bit CPUs.

While there is more argument about the specific syntax, nobody has seemed able to provide a compellingly better syntax than **`long long`**, which at least avoided gratuitous namespace pollution. Proposals like **`int64_t`** seem very awkward for 36-bit CPUs, for example.

Given the various complex interactions, **`long long`** seems a reasonable addition to C, as existing practice has shown the need for a larger integer, and **`long long`** syntax seems one of the *least bad* choices.

Implementation note: excluding the library, programs that do not use **`long long`** do not have any overhead caused by the addition of **`long long`** to C9X; however, certain standard library functions that are useful when the program does not otherwise use **`long long`** do have overhead that an implementation may wish to eliminate. The primary examples of such library functions are the **`printf`** and **`scanf`** families.

The only use of **`long long`** in some programs might be the support in **`printf`** to print **`long`**

long values even though the program never actually prints such a value. On systems where there is no hardware support for **long long** arithmetic, the **long long** support in **printf** might cause runtime functions to be loaded to emulate **long long** arithmetic. For embedded systems, where memory is precious, the code to support printing **long long** values and for doing **long long** arithmetic is a great burden if that code is never actually used by the program. (In contrast, large systems with shared libraries may have no overhead supporting **long long** all of the time.)

The situation with **long long** is very simiular to the situration with floating point in C for the PDP-11. Some models of the PDP-11 lacked floating point hardware and were limited to a 64K address space. (A similar situation exists with MS-DOS-based implementations.) Many C programs did not use floating point, but because of **printf** and **scanf**, wasted a significant part of their address space on floating point emulation code. The PDP-11 C solution was simple and effective: two versions of **printf** and **scanf** were provided. One version supported printing or scanning floating point values; the other version did not. By linking with the proper version of **printf** and **scanf**, the program could avoid the overhead of floating point code. This technique also works for avoiding any overhead from **long long**.

In the simplest form, this technique can be purely manual and require no changes to compilers or linkers: the user explicitly links the program with the proper versions of **printf** and **scanf**. Automatic solutions are also possible by having the compiler inform the linker if the program uses **long long**. A variety of sulutions are possible, and like the PDP-11 implementation, many embedded systems have already solved the overhead problem for floating point, and may find it useful to adopt a similar solution for **long long**.[4]

Since the technique discussed above allows an implementation to avoid the burden of linking in **long long** support when it is not needed, the committee saw no reason to make support for **long long** be optional.

QUIET CHANGE IN C9X

In some environments such as `LP64` and `ILP64`, **long long** and **long** are equivalent. In the others (`ILP32LL` and `LLP64`), **long long** is larger than **long**. If the system environment also changes standard definitions such as **ptrdiff_t** to become **long long** or **size_t** to become **unsigned long long**, then existing correct code can be broken. For example,

```
unsigned long x;
size_t y;
x = y;
```

silently truncates **y**.

[4] An implementation may be able to optimize more cases by inspecting the argument types in calls to printf and scanf family functions, but should be aware that vprintf and vscanf arguments types may not be available at compile time.

A new feature of C9X: In C89, all type specifiers could be omitted from the declaration specifiers in a declaration. In such a case **int** was implied. The Committee decided that the inherent danger of this feature outweighed its convenience, and so it was removed. The effect is to guarantee the production of a diagnostic that will catch an additional category of programming errors. After issuing the diagnostic, an implementation may choose to assume an implicit **int** and continue to translate the program in order to support existing source code that exploited this feature.

6.7.2.1 Structure and union specifiers

Three types of bit fields are now defined: plain **int** calls for *implementation-defined* signedness (as in K&R), **signed int** calls for assuredly signed fields, and **unsigned int** calls for unsigned fields. The old constraints on bit fields crossing *word* boundaries have been relaxed, since so many properties of bit fields are implementation dependent anyway.

The layout of structures is determined only to a limited extent:

- no hole may occur at the beginning.
- members occupy increasing storage addresses.
- if necessary, a hole is placed on the end to make the structure big enough to pack tightly into arrays and maintain proper alignment.

Since some existing implementations, in the interest of enhanced access time, leave internal holes larger than absolutely necessary, it is not clear that a portable deterministic method can be given for traversing a structure member by member.

To clarify what is meant by the notion that "all the members of a union occupy the same storage," the Standard specifies that a pointer to a union, when suitably cast, points to each member (or, in the case of a bit-field member, to the storage unit containing the bit field).

A new feature of C9X: There is a common idiom known as the "struct hack" for creating a structure containing a variable-size array:

```
struct s
{
    int n_items;
    /* possibly other fields */
    int items[1];
};
struct s *p;
size_t n, i;

/* code that sets n omitted */
p = malloc(sizeof(struct s) + (n - 1) * sizeof(int));
```

```
/* code to check for failure omitted */
p->n_items = n;
/* example usage */
for (i = 0; i < p->n_items; i++)
    p->items[i] = i;
```

The validity of this construct has always been questionable. In the response to one Defect Report, The Committee decided that it was undefined behavior because the array **p->items** contains only one item, irrespective of whether the space exists. An alternative construct was suggested: make the array size larger than the largest possible case (for example, using **int items[INT_MAX];**), but this approach is also undefined for other reasons.

The Committee felt that, although there was no way to implement the "struct hack" in C89, it was nonetheless a useful facility. Therefore the new feature of "flexible array members" was introduced. Apart from the empty brackets, and the removal of the "**-1**" in the **malloc** call, this is used in the same way as the struct hack, but is now explicitly valid code.

There are a few restrictions on flexible array members that ensure that code using them makes sense. For example, there must be at least one other member, and the flexible array must occur last. Similarly, structures containing flexible arrays can't occur in the middle of other structures or in arrays. Finally, **sizeof** applied to the structure ignores the array but counts any padding before it. This makes the **malloc** call as simple as possible.

6.7.2.2 Enumeration specifiers

A new feature of C9X: a common extension in many implementations allows a trailing comma after the list of enumeration constants. The Committee decided to adopt this feature as an innocuous extension that mirrors the trailing commas allowed in initializers.

6.7.2.3 Tags

As with all block-structured languages that also permit forward references, C has a problem with structure and union tags. If one wants to declare, within a block, two mutually-referencing structures, one must write something like

```
struct x { struct y *p; /*...*/ };
struct y { struct x *q; /*...*/ };
```

But if **struct y** is already defined in a containing block, the first field of **struct x** will refer to the older declaration.

Thus special semantics were given to the form

```
struct y;
```

which now hides the outer declaration of **y**, and "opens" a new instance in the current block.

QUIET CHANGE IN C89

The empty declaration **`struct x;`** is not innocuous.

6.7.3 Type qualifiers

The C89 Committee added to C two *type qualifiers*, **`const`** and **`volatile`**; and C9X adds a third, **`restrict`**. Individually and in combination they specify the assumptions a compiler can and must make when accessing an object through an lvalue.

The syntax and semantics of **`const`** were adapted from C++; the concept itself has appeared in other languages. **`volatile`** and **`restrict`** are inventions of the Committee; and both follow the syntactic model of **`const`**.

Type qualifiers were introduced in part to provide greater control over optimization. Several important optimization techniques are based on the principle of "cacheing": under certain circumstances the compiler can remember the last value accessed (read or written) from a location, and use this retained value the next time that location is read. (The memory, or "cache", is typically a hardware register.) If this memory is a machine register, for instance, the code can be smaller and faster using the register rather than accessing external memory.

The basic qualifiers can be characterized by the restrictions they impose on access and cacheing:

`const` No writes through this lvalue. In the absence of this qualifier, writes may occur through this lvalue.

`volatile` No cacheing through this lvalue: each operation in the abstract semantics must be performed (that is, no cacheing assumptions may be made, since the location is not guaranteed to contain any previous value). In the absence of this qualifier, the contents of the designated location may be assumed to be unchanged except for possible aliasing.

`restrict` Objects referenced through a **`restrict`**-qualified pointer have a special association with that pointer. All references to that object must directly or indirectly use the value of this pointer. In the absence of this qualifier, other pointers can alias this object. Cacheing the value in an object designated through a **`restrict`**-qualified pointer is safe at the beginning of the block in which the pointer is declared, because no pre-existing aliases may also be used to reference that object. The cached value must be restored to the object by the end of the block, where pre-existing aliases again become available. New aliases may be formed within the block, but these must all depend on the value of the **`restrict`**-qualified pointer, so that they can be identified and adjusted to refer to the cached value. For a **`restrict`**-qualified pointer at file scope, the block is the body of **`main`**.

A translator design with no cacheing optimizations can effectively ignore the type qualifiers, except insofar as they affect assignment compatibility.

It would have been possible, of course, to specify **nonconst** instead of **const**, etc. The senses of these concepts in the Standard were chosen to assure that the default, unqualified, case is the most common, and that it corresponds most clearly to traditional practice in the use of lvalue expressions.

Several combinations of the three qualifiers are possible and most define useful sets of lvalue properties. The next several paragraphs describe typical uses of the **const** and **volatile** qualifiers. The **restrict** qualifier is discussed in §6.7.3.1.

The translator may assume, for an unqualified lvalue, that it may read or write the referenced object, that the value of this object cannot be changed except by explicitly programmed actions in the current thread of control, but that other lvalue expressions could reference the same object.

const is specified in such a way that an implementation is at liberty to put **const** objects in read-only storage, and is encouraged to diagnose obvious attempts to modify them, but is not required to track down all the subtle ways that such checking can be subverted.

A **static volatile** object is an appropriate model for a memory-mapped I/O register. Implementors of C translators should take into account relevant hardware details on the target systems when implementing accesses to **volatile** objects. For instance, the hardware logic of a system may require that a two-byte memory-mapped register not be accessed with byte operations; and a compiler for such a system would have to assure that no such instructions were generated, even if the source code only accesses one byte of the register. Whether read-modify-write instructions can be used on such device registers must also be considered. Whatever decisions are adopted on such issues must be documented, as **volatile** access is implementation-defined. A **volatile** object is also an appropriate model for a variable shared among multiple processes.

A **static const volatile** object appropriately models a memory-mapped input port, such as a real-time clock. Similarly, a **const volatile** object models a variable which can be altered by another process but not by this one.

Although the type qualifiers are formally treated as defining new types, they actually serve as modifiers of declarators. Thus the declarations

```
const struct s {int a,b;} x;
struct s y;
```

declare **x** as a **const** object, but not **y**. The **const** property can be associated with the aggregate type by means of a type definition:

```
typedef const struct s {int a,b;} stype;
stype x;
```

```
stype y;
```

In these declarations the **const** property is associated with the declarator stype, so **x** and **y** are both **const** objects.

The C89 Committee considered making **const** and **volatile** storage classes, but this would have ruled out any number of desirable constructs, such as **const** members of structures and variable pointers to **const** types.

A cast of a value to a qualified type has no effect; the qualification (**volatile**, say) can have no effect on the access since it has occurred prior to the cast. If it is necessary to access a non-**volatile** object using **volatile** semantics, the technique is to cast the address of the object to the appropriate pointer-to-qualified type, then dereference that pointer.

6.7.3.1 Formal definition of `restrict`

A new feature of C9X: The **restrict** type qualifier allows programs to be written so that translators can produce significantly faster executables. Anyone for whom this is not a concern can safely ignore this feature of the language.

The problem that the **restrict** qualifier addresses is that potential aliasing can inhibit optimizations. Specifically, if a translator cannot determine that two different pointers are being used to reference different objects, then it cannot apply optimizations such as maintaining the values of the objects in registers rather than in memory, or reordering loads and stores of these values. This problem can have a significant effect on a program that, for example, performs arithmetic calculations on large arrays of numbers. The effect can be measured by comparing a program that uses pointers with a similar program that uses file scope arrays (or with a similar Fortran program). The array version can run faster by a factor of ten or more on a system with vector processors. Where such large performance gains are possible, implementations have of course offered their own solutions, usually in the form of compiler directives that specify particular optimizations. Differences in the spelling, scope, and precise meaning of these directives have made them troublesome to use in a program that must run on many different systems. This was the motivation for a standard solution.

The **restrict** qualifier was designed to express and extend two types of aliasing information already specified in the language.

First, if a single pointer is directly assigned the return value from an invocation of **malloc**, then that pointer is the sole initial means of access to the allocated object (that is, another pointer can gain access to that object only by being assigned a value that is based on the value of the first pointer). Declaring the pointer to be *restrict-qualified* expresses this information to a translator. Furthermore, the qualifier can be used to extend a translator's special treatment of such a pointer to more general situations. For example, an invocation of **malloc** might be hidden from the translator in another function, or a single invocation of **malloc** might be used to allocate several objects, each referenced through its own pointer.

Second, the library specifies two versions of an object copying function, because on many systems a faster copy is possible if it is known that the source and target arrays do not overlap. The **restrict** qualifier can be used to express the restriction on overlap in a new prototype that is compatible with the original version:

```
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
void *memmove(void * s1, const void * s2, size_t n);
```

With the restriction visible to a translator, a straightforward implementation of **memcpy** in C can now give a level of performance that previously required assembly language or other non-standard means. Thus the **restrict** qualifier provides a standard means with which to make, in the definition of any function, an aliasing assertion of a type that could previously be made only for library functions.

The complexity of the specification of the **restrict** qualifier reflects the fact that C has a rich set of types and a dynamic notion of the type of an object. Recall, for example, that an object does not have a fixed type, but acquires a type when referenced. Similarly, in some of the library functions, the extent of an array object referenced through a pointer parameter is dynamically determined, either by another parameter or by the contents of the array.

The full specification is necessary to determine the precise meaning of a qualifier in any context, and so must be understood by compiler implementors. Fortunately, most others will need to understand only a few simple patterns of usage explained in the following examples.

A translator can assume that a file scope **restrict**-qualified pointer is the sole initial means of access to an object, much as if it were the declared name of an array. This is useful for a dynamically allocated array whose size is not known until run time. Note in the example how a single block of storage is effectively subdivided into two disjoint objects.

```
float * restrict a1, * restrict a2;

void init(int n)
{
    float * t =  malloc(2 * n * sizeof(float));
    a1 = t;      // a1 refers to 1st half
    a2 = t + n;  // a2 refers to 2nd half
}
```

A translator can assume that a **restrict**-qualified pointer that is a function parameter is, at the beginning of each execution of the function, the sole means of access to an object. Note that this assumption expires with the end of each execution. In the following example, parameters **a1** and **a2** can be assumed to refer to disjoint array objects because both are **restrict**-qualified. This implies that each iteration of the loop is independent of the others, and so the loop can be aggressively optimized.

```
void f1(int n, float * restrict a1,
        const float * restrict a2)
{
    int i;
    for ( i = 0; i < n; i++ )
        a1[i] += a2[i];
}
```

A translator can assume that a **restrict**-qualified pointer declared with block scope is, during each execution of the block, the sole initial means of access to an object. An invocation of the macro shown in the following example is equivalent to an inline version of a call to the function **f1** above.

```
# define f2(N,A1,A2)                \
{   int n = (N);                    \
    float * restrict a1 = (A1);     \
    float * restrict a2 = (A2);     \
    int i;                          \
    for ( i = 0; i < n; i++ )       \
        a1[i] += a2[i];             \
}
```

The **restrict** qualifier can be used in the declaration of a structure member. A translator can assume, when an identifier is declared that provides a means of access to an object of that structure type, that the member provides the sole initial means of access to an object of the type specified in the member declaration. The duration of the assumption depends on the scope of the identifier, not on the scope of the declaration of the structure. Thus a translator can assume that **s1.a1** and **s1.a2** below are used to refer to disjoint objects for the duration of the whole program, but that **s2.a1** and **s2.a2** are used to refer to disjoint objects only for the duration of each invocation of the **f3** function.

```
struct t {
    int n;
    float * restrict a1, * restrict a2;
};

struct t s1;

void f3(struct t s2) { /* ... */ }
```

The meaning of the **restrict** qualifier for a union member or in a type definition is analogous. Just as an object with a declared name can be aliased by an unqualified pointer, so can the object associated with a **restrict**-qualified pointer. The **restrict** qualifier is therefore unlike the **register** storage class, which precludes such aliasing.

This allows the **restrict** qualifier to be introduced more easily into existing programs, and also allows **restrict** to be used in new programs that call functions from libraries that do not use

the qualifier. In particular, a **restrict**-qualified pointer can be the actual argument for a function parameter that is unqualified. On the other hand, it is easier for a translator to find opportunities for optimization if as many as possible of the pointers in a program are **restrict**-qualified.

6.7.4 Function specifiers

A new feature of C9X: The **inline** keyword, adapted from C++, is a *function-specifier* that can be used only in function declarations. It is useful for program optimizations that require the definition of a function to be visible at the site of a call. (Note that the Standard does not attempt to specify the nature of these optimizations.)

Visibility is assured if the function has internal linkage, or if it has external linkage and the call is in the same translation unit as the external definition. In these cases, the presence of the **inline** keyword in a declaration or definition of the function has no effect beyond indicating a preference that calls of that function should be optimized in preference to calls of other functions declared without the **inline** keyword.

Visibility is a problem for a call of a function with external linkage where the call is in a different translation unit from the function's definition. In this case, the **inline** keyword allows the translation unit containing the call to also contain a local, or inline, definition of the function.

A program can contain a translation unit with an external definition, a translation unit with an inline definition, and a translation unit with a declaration but no definition for a function. Calls in the latter translation unit will use the external definition as usual.

An inline definition of a function is considered to be a different definition than the external definition. If a call to some function **func** with external linkage occurs where an inline definition is visible, the behavior is the same as if the call were made to another function, say _ _**func**, with internal linkage. A conforming program must not depend on which function is called. This is the inline model in the Standard.

A conforming program must not rely on the implementation using the inline definition, nor may it rely on the implementation using the external definition. The address of a function is always the address corresponding to the external definition, but when this address is used to call the function, the inline definition might be used. Therefore, the following example might not behave as expected.

```
inline const char *saddr(void)
{
    static const char name[] = "saddr";
    return name;
}

int compare_name(void)
{
```

```
    return saddr() == saddr();  // unspecified behavior
}
```

Since the implementation might use the inline definition for one of the calls to **saddr** and use the external definition for the other, the equality operation is not guaranteed to evaluate to 1 (true). This shows that static objects defined within the inline definition are distinct from their corresponding object in the external definition. This motivated the constraint against even defining a non-**const** object of this type.

Inlining was added to the Standard in such a way that it can be implemented with existing linker technology, and a subset of C9X inlining is compatible with C++. This was achieved by requiring that exactly one translation unit containing the definition of an inline function be specified as the one that provides the external definition for the function. Because that specification consists simply of a declaration that either lacks the **inline** keyword, or contains both **inline** and **extern**, it will also be accepted by a C++ translator.

Inlining in C9X does extend the C++ specification in two ways. First, if a function is declared **inline** in one translation unit, it need not be declared **inline** in every other translation unit. This allows, for example, a library function that is to be inlined within the library but available only through an external definition elsewhere. The alternative of using a wrapper function for the external function requires an additional name; and it may also adversely impact performance if a translator does not actually do inline substitution.

Second, the requirement that all definitions of an inline function be “exactly the same” is replaced by the requirement that the behavior of the program should not depend on whether a call is implemented with a visible inline definition, or the external definition, of a function. This allows an inline definition to be specialized for its use within a particular translation unit. For example, the external definition of a library function might include some argument validation that is not needed for calls made from other functions in the same library. These extensions do offer some advantages; and programmers who are concerned about compatibility can simply abide by the stricter C++ rules.

Note that it is *not* appropriate for implementations to provide inline definitions of standard library functions in the standard headers because this can break some legacy code that redeclares standard library functions after including their headers. The **inline** keyword is intended only to provide users with a portable way to suggest inlining of functions. Because the standard headers need not be portable, implementations have other options along the lines of:

```
#define  abs(x)  __builtin_abs(x)
```

or other non-portable mechanisms for inlining standard library functions.

6.7.5 Declarators

The function prototype syntax was adapted from C++ (see §6.5.2.2 and §6.7.5.3).

Some pre-C89 implementations had a limit of six type modifiers (*function returning, array of, pointer to*), the limit used in Ritchie's original compiler. This limit was raised to twelve in C89 since the original limit has proven insufficient in some cases; in particular, it did not allow for Fortran-to-C translation, since Fortran allows for seven subscripts. (Some users have reported using nine or ten levels, particularly in machine-generated C code.)

6.7.5.1 Pointer declarators

A pointer declarator may have its own type qualifiers to specify the attributes of the pointer itself, as opposed to those of the reference type. The construct is adapted from C++.

`const int *` means *(variable) pointer to constant* **`int`**, and **`int * const`** means *constant pointer to (variable)* **`int`**, just as in C++. (And *mutatis mutandis* for the other type qualifiers.) As with other aspects of C type declarators, judicious use of **`typedef`** statements can clarify the code.

6.7.5.2 Array declarators

The concept of *composite types* (§6.2.7) was introduced to provide for the accretion of information from incomplete declarations, such as array declarations with missing size, and function declarations with missing prototype (argument declarations). Type declarators are therefore said to specify *compatible types* if they agree except for the fact that one provides less information of this sort than the other.

C9X adds a new array type called a variable length array type. The inability to declare arrays whose size is known only at execution time was often cited as a primary deterrent to using C as a numerical computing language. Adoption of some standard notion of execution time arrays was considered crucial for C's acceptance in the numerical computing world.

The number of elements specified in the declaration of a variable length array type is a runtime expression. Before C9X, this size expression was required to be an integer constant expression.

C9X makes a distinction between *variable length array types* and *variably modified types*, for example, a pointer to a variable length array. Variable length array types are a subset of all possible variably modified types.

All variably modified types must be declared at either block scope or function prototype scope. File scope identifiers cannot be declared with a variably modified type. Furthermore, array objects declared with either the **`static`** or **`extern`** storage class specifiers cannot be declared with a variable length array type, although block scope pointers declared with the **`static`** storage class specifier can be declared as pointers to variable length array types. Finally, if the identifier that is being declared has a variable length array type (as opposed to being a pointer to a variable length array), then it must be an ordinary identifier. This eliminates structure and union members.

Restricting variable length array declarators to identifiers with automatic storage duration is natural since "variableness" at file scope requires some notion of parameterized typing. There was

sentiment for allowing structure members to be variably modified; however allowing structure members to have a variable length array type introduces a host of problems such as the treatment when passing these objects, or even pointers to these objects, as parameters. In addition, the semantics of the **offsetof** macro would need to be extended and runtime semantics added. Finally, there was disagreement whether the size of a variable length array member could be determined using one of the other members. The Committee decided to limit variable length array types to declarations outside structures and unions.

Side effects in variable length array size expressions are guaranteed to be produced, except in one context. If a size expression is part of the operand of a **sizeof** operator, and the result of that **sizeof** operator does not depend on the value of the size expression, then it is unspecified whether side effects are produced. In the following example:

```
{
    int n = 5;
    int m = 7;
    size_t sz = sizeof(int (*)[n++]);
}
```

the value of the result of the sizeof operator is the same as in:

```
{
    int n = 5;
    int m = 7;
    size_t sz = sizeof(int (*)[m++]);
}
```

Since the value stored in **sz** does not depend on the size expression, the side effect in **n++** is not guaranteed to occur. Requiring the side effrect introduced a burden on some implementations. Since side effects in this context seemed to have limited utility and are not perceived to be a desired coding style, the Committee decided to make it unspecified whether these size expressions are actually evaluated.

A new feature of C9X: The **static** storage class specifier and any of the type-qualifiers, **restrict**, **const** or **volatile**, can appear inside the **[** and **]** that are used to declare an array type, but only in the outermost array type derivation of a function parameter.

The **static** keyword provides useful information about the intent of function parameters. Consider:

```
void fadd(double *a, const double *b)
{
    int i;

    for (i = 0; i < 10; i++) {
        if (a[i] < 0.0)
            return;
```

```
        a[i] += b[i];
    }

    return;
}
```

It would be a significant advantage on some systems for the translator to initiate, at the beginning of the function, prefetches or loads of the arrays that will be referenced through the parameters. There is no way in C89 for the user to provide information to the translator about how many elements are guaranteed to be available.

In C9X, the use of the **static** keyword in:

```
void fadd(double a[static 10], const double b[static 10])
{
    int i;

    for (i = 0; i < 10; i++) {
        if (a[i] < 0.0)
            return;

        a[i] += b[i];
    }

    return;
}
```

guarantees that both the pointers **a** and **b** provide access to the first element of an array containing at least ten elements. The **static** keyword also guarantees that the pointer is not **NULL** and points to an object of the appropriate effective type. It does not, however, guarantee that **a** and **b** point to unique, non-overlapping objects. The **restrict** keyword is used for that purpose as in:

```
void fadd(double a[static restrict 10],
    const double b[static restrict 10])
{
    int i;

    for (i = 0; i < 10; i++) {
        if (a[i] < 0.0)
            return;

        a[i] += b[i];
    }

    return;
}
```

This function definition specifies that the parameters **a** and **b** are restricted pointers. This is information that an optimizer can use, for example, to unroll the loop and reorder the loads and stores of the elements referenced through **a** and **b**.

The **const** keyword can be used to indicate that the pointer will always point to the same array object. The function declaration:

```
void f(double x[const], const double y[const]);
```

is another way of declaring:

```
void f(double * const x, const double * const y);
```

There does not appear to be much value in using **volatile** to qualify an array function parameter.

6.7.5.3 Function declarators (including prototypes)

The function prototype mechanism is one of the most useful additions to the C language. The feature, of course, has precedent in many of the Algol-derived languages of the past 25 years. The particular form adopted in the Standard is based in large part upon C++.

Function prototypes provide a powerful translation-time error detection capability. In traditional C practice without prototypes, it is extremely difficult for the translator to detect errors (wrong number or type of arguments) in calls to functions declared in another source file. Detection of such errors has occurred either at runtime or through the use of auxiliary software tools.

In function calls not in the scope of a function prototype, integer arguments have the *integer promotions* applied and **float** arguments are widened to **double**. It is not possible in such a call to pass an unconverted **char** or **float** argument. Function prototypes give the programmer explicit control over the function argument type conversions, so that the often inappropriate and sometimes inefficient default widening rules for arguments can be suppressed by the implementation.

Modifications of function interfaces are easier in cases where the actual arguments are still assignment compatible with the new formal parameter type: only the function definition and its prototype need to be rewritten in this case; no function calls need be rewritten. Allowing an optional identifier to appear in a function prototype serves two purposes:

- the programmer can associate a meaningful name with each argument position for documentation purposes.
- a function declarator and a function prototype can use the same syntax. The consistent syntax makes it easier for new users of C to learn the language. Automatic generation of function prototype declarators from function definitions is

also facilitated.

The Standard requires that calls to functions taking a variable number of arguments must occur in the presence of a prototype using the trailing ellipsis notation , (**...**). An implementation may thus assume that all other functions are called with a fixed argument list, and may therefore use possibly more efficient calling sequences. Programs using old-style headers in which the number of arguments in the calls and the definition differ may not work in implementations which take advantage of such optimizations. This is not a quiet change, strictly speaking, since the program does not conform to the Standard. A word of warning is in order, however, since the style is not uncommon in existing code, and since a conforming translator is not required to diagnose such mismatches when they occur in separate translation units. Such trouble spots can be made manifest (assuming an implementation provides reasonable diagnostics) by providing new-style function declarations in the translation units with the non-matching calls. Programmers who currently rely on being able to omit trailing arguments are advised to recode using the **<stdarg.h>** paradigm.

Function prototypes may be used to define function types as well:

```
typedef  double (*d_binop) (double A, double B);

struct d_funct {
    d_binop f1;
    int (*f2)(double, double);
};
```

struct d_funct has two members, both of which hold pointers to functions taking two **double** arguments; the function types differ in their return type.

A function prototype can have parameters that have variable length array types (§6.7.5.2) using a special syntax as in

```
int minimum(int, int [*][*]);
```

This is consistent with other C prototypes where the name of the parameter need not be specified.

There was considerable debate about whether to maintain the current lexical ordering rules for variable length array parameters in function definitions. For example, the following old-style declaration

```
void f(double a[*][*], int n);

void f(a, n)
    int n;
    double a[n][n];
{
    // ...
}
```

cannot be expressed with a definition that has a parameter type list as in

```
void f(double a[n][n], int n)  // error
{
    /* … */
}
```

Previously, programmers did not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. With Standard C's lexical ordering rules, the declaration of **a** would force **n** to be undefined or captured by an outside declaration. The possibility of allowing the scope of parameter **n** to extend to the beginning of the parameter-type-list was explored (relaxed lexical ordering), which would allow the size of parameter **a** to be defined in terms of parameter **n**, and could help convert a Fortran library routine into a C function. Such a change to the lexical ordering rules is not considered to be in the "Spirit of C," however. This is an unforeseen side effect of Standard C prototype syntax.

The following example demonstrates how to declare parameters in any order and avoid lexical ordering issues.

```
void g(double *ap, int n)
{
    double (*a)[n] = (double (*)[n]) ap;

    /* … */ a[1][2] /* … */
}
```

In this case, the parameter **ap** is assigned to a local pointer that is declared to be a pointer to a variable length array. The function **g** can be called as in

```
{
    double x[10][10];
    g(&x[0][0], 10);
}
```

which allows the array address to be passed as the first argument. The strict lexical ordering rules remain in place.

6.7.6 Type names

Empty parentheses within a type name are always taken as meaning *function with unspecified arguments* and never as unnecessary parentheses around the elided identifier. This specification avoids an ambiguity by fiat.

6.7.7 Type definitions

A **typedef** may only be redeclared in an inner block with a declaration that explicitly contains a type name. This rule avoids the ambiguity about whether to take the **typedef** as the type name or the candidate for redeclaration.

Some pre-C89 implementations allowed type specifiers to be added to a type defined using **typedef**. Thus

```
typedef short int small;
unsigned small x;
```

would give **x** the type **unsigned short int**. The C89 Committee decided that since this interpretation may be difficult to provide in many implementations, and since it defeats much of the utility of **typedef** as a data abstraction mechanism, such type modifications are invalid. This decision is incorporated in the rules of §6.7.2.

A proposed **typeof** operator was rejected on the grounds of insufficient utility.

In C89, a **typedef** could be redeclared in an inner block with a declaration that explicitly contained a type name. This rule avoided the ambiguity about whether to take the **typedef** as the type name or a candidate for redeclaration. In C9X, implicit **int** declarations are not allowed, so this anbiguity is not possible and the rule is no longer necessary.

Using a **typedef** to declare a variable length array object (see §6.7.5.2) could have two possible meanings. Either the size could be eagerly computed when the **typedef** is declared, or the size could be lazily computed when the *object* is declared. For example

```
{
    typedef VLA[n];
    n++;
    VLA object;

    // ...
}
```

The question arises whether **n** should be evaluated at the time the type definition itself is encountered or each time the type definition is used for some object declaration. The Committee decided that if the evaluation were to take place each time the **typedef** name is used, then a single type definition could yield variable length array types involving many different dimension sizes. This possibility seemed to violate the spirit of type definitions. The decision was made to force evaluation of the expression at the time the type definition itself is encountered.

6.7.8 Initialization

An implementation might conceivably have codes for floating zero and/or null pointer other than

all bits zero. In such a case, the implementation must fill out an incomplete initializer with the various appropriate representations of zero; it may not just fill the area with zero bytes.

The C89 Committee considered proposals for permitting automatic aggregate initializers to consist of a brace-enclosed series of arbitrary execution-time expressions, instead of just those usable for a translation-time static initializer. However, cases like this were troubling:

```
int x[2] = { f(x[1]), g(x[0]) };
```

Rather than determine a set of rules which would avoid pathological cases and yet not seem too arbitrary, the C89 Committee elected to permit only static initializers. Consequently, an implementation may choose to build a hidden static aggregate, using the same machinery as for other aggregate initializers, then copy that aggregate to the automatic variable upon block entry.

A structure expression, such as a call to a function returning the appropriate structure type, is permitted as an automatic structure initializer, since the usage seems unproblematic.

For programmer convenience, even though it is a minor irregularity in initializer semantics, the trailing null character in a string literal need not initialize an array element, as in

```
char mesg[5] = "help!";
```

Some widely used implementations provide precedent.

K&R allows a trailing comma in an initializer at the end of an initializer-list. The Standard has retained this syntax, since it provides flexibility in adding or deleting members from an initializer list, and simplifies machine generation of such lists.

Various implementations have parsed aggregate initializers with partially elided braces differently.

The Standard has reaffirmed the top-down parse described in K&R. Although the construct is allowed, and its parse well defined, the C89 Committee urges programmers to avoid partially elided initializers because such initializations can be quite confusing to read.

QUIET CHANGE IN C89

> Code which relies on a bottom-up parse of aggregate initializers with partially elided braces will not yield the expected initialized object.

The C89 Committee has adopted the rule (already used successfully in some implementations) that the first member of the union is the candidate for initialization. Other notations for union initialization were considered, but none seemed of sufficient merit to outweigh the lack of prior art.

This rule has a parallel with the initialization of structures. Members of structures are initialized in the sequence in which they are declared. The same could be said of C89 unions, with the significant difference that only one union member, the first, can be initialized.

A new feature of C9X: Designated initializers provide a mechanism for initializing sparse arrays, a practice common in numerical programming. They add useful functionality that already exists in Fortran so that programmers migrating to C need not suffer the loss of a program-text-saving notational feature.

This feature also allows initialization of sparse structures, common in systems programming, and allows initialization of unions via any member, regardless of whether or not it is the first member.

Designated initializers integrate easily into the C grammar and do not impose any additional run-time overhead on a user's program. Their initial C implementation appeared in a compiler by Ken Thompson at AT&T Bell Laboratories.

6.8 Statements and blocks

The C89 Committee considered proposals for forbidding a **`goto`** into a block from outside, since such a restriction would make possible much easier flow optimization and would avoid the whole issue of initializing auto storage; but it rejected such a ban out of fear of invalidating working code, however undisciplined, and out of concern for those producing machine-generated C.

A new feature of C9X: A common coding practice is always to use compound statements for every selection and iteration statement because this guards against inadvertent problems when changes are made in the future. Because this can lead to surprising behavior in connection with certain uses of compound literals (§6.5.2.5), the concept of a block has been expanded in C9X.

Given the following example involving three different compound literals:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    if (*(q = (int[2]){i, j}))
        fn(p = (int[5]){9, 8, 7, 6, 5}, q);
    else
        fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);

    return *p;
}
```

it seemed surprising that just introducing compound statements also introduced undefined behavior:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;
```

```
    if (*(q = (int[2]){i, j})) {
        fn(p = (int[5]){9, 8, 7, 6, 5}, q);
    } else {
        fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);
    }

    return *p;   // undefined—no guarantee *p designates an object
}
```

Therefore, the substatements associated with all selection and iteration statements are now defined to be blocks, even if they are not also compound statements. A compound statement remains a block, but is no longer the only kind of block. Furthermore, all selection and iteration statements themselves are also blocks, implying no guarantee that ***q** in the previous example designates an object, since the above example behaves as if written:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    {
        if (*(q = (int[2]){i, j})) {
            // *q is guaranteed to designate an object
            fn(p = (int[5]){9, 8, 7, 6, 5}, q);
        } else {
            // *q is guaranteed to designate an object
            fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);
        }
    }

    // *q is not guaranteed to designate an object

    return *p;   // *p is not guaranteed to designate an object
}
```

If compound literals are defined in selection or iteration statements, their lifetimes are limited to the implied enclosing block; therefore the definition of “block” has been moved to this section. This change is compatible with similar C++ rules.

QUIET CHANGE IN C9X

There are some pathological cases where program behavior changes quietly as demonstrated by the following example.

```
enum {a, b};
```

```
int different(void)
{
    if (sizeof(enum {b, a}) != sizeof(int))
        return a;  // a == 1

    return b;  // which b?
}
```

In C89, the declaration **enum {b, a}** persists after the **if** statement terminates; but in C9X, the implied block that encloses the entire **if** statement limits the scope of that declaration; therefore the **different** function returns different values in C89 and C9X. The Committee views such cases as unintended artifacts of allowing declarations as operands of cast and **sizeof** operators; and this change is not viewed as a serious problem.

6.8.1 Labeled statements

Since label definition and label reference are syntactically distinctive contexts, labels are established as a separate name space.

6.8.2 Compound statement

A new feature of C9X: declarations and statements may be mixed in an arbitrary manner (see §6.2.4).

6.8.3 Expression and null statements

The **void** cast is not needed in an expression statement, since any value is always discarded. Some compilers prefer this reassurance, however, for functions that return objects of types other than **void**.

6.8.4 Selection statements

A new feature of C9X: Unlike in C89, all selection statements and their associated substatements are blocks. See §6.8.2.

6.8.4.1 The `if` statement

See §6.8.

6.8.4.2 The `switch` statement

The controlling expression of a **switch** statement may now have any integer type, even **unsigned long long**. Floating types were rejected for **switch** statements since exact equality in floating point is not portable.

case labels are first converted to the type of the controlling expression of the **switch**, then checked for equality with other labels. No two may match after conversion.

Case ranges of the form, *lo* **..** *hi*, were seriously considered, but ultimately not adopted in the Standard on the grounds that it added no new capability, just a problematic coding convenience. The construct seems to promise more than it could be mandated to deliver:

- A great deal of code or jump table space might be generated for an innocent-looking case range such as **0 .. 65535**.
- The range **'A' .. 'Z'** would specify all the integers between the character code for "upper-case-A" and that for "upper-case-Z". In some common character sets this range would include non-alphabetic characters, and in others it might not include all the alphabetic characters, especially in non-English character sets.

No serious consideration was given to making **switch** more structured, as in Pascal, out of fear of invalidating working code.

QUIET CHANGE IN C89

long expressions and constants in **switch** statements are no longer truncated to **int**.

6.8.5 Iteration statements

A new feature of C9X: Unlike in C89, all iteration statements and their associated substatements are blocks. See §6.8.

6.8.5.3 The for statement

A new feature of C9X: It is common for a **for** loop to involve one or more counter variables which are initialized at the start of the loop and never used again. In C89 it was necessary to declare those variables at the start of the enclosing block with a subsequent risk of accidentally reusing them for some other purpose. It is now permitted to declare these variables as part of the **for** statement itself. Such a loop variable is in a new scope, so it does not affect any other variable with the same name and is destroyed at the end of the loop, which can lead to possible optimizations.

To simplify the syntax, each loop is limited to a single declaration (though this can declare several variables), and these must have **auto** or **register** storage class.

Example:

```
int i = 42;
for (int i = 5, j = 15; i < 10; i++, j--)
    printf("Loop %d %d\n", i, j);
```

```
printf("I = %d\n", i);  // there is no j in scope
```

will output:

```
Loop 5 15
Loop 6 14
Loop 7 13
Loop 8 12
Loop 9 11
I = 42
```

Note that the syntax allows loops like:

```
for (struct s *p = list, **q; p != NULL; p = *q)
    q = &(p->next);
```

A new feature of C9X: In C89, for loops were defined in terms of a syntactic rewrite into while loops. This introduced problems for the definition of the continue statement; and it also introduced problems when the operands of cast and sizeof operators contain declarations as in:

```
enum {a, b};

{
    int i, j = b;

    for (i = a; i < j; i += sizeof(enum {b, a}))
        j += b;
}
```

not being equivalent to:

```
enum {a, b};

{
    int i, j = b;

    i = a;
    while (i < j) {
        j += b;                         // which b?
        i += sizeof(enum {b, a});  // declaration of b moves
    }
}
```

because a different **b** is used to increment **i** in each case. For this reason, the syntactic rewrite has been replaced by words that describe the behavior.

6.8.6. Jump statements

6.8.6.1 The `goto` statement

With the combination of variable length arrays (see §6.7.5.2) and mixed code and declarations, situations can arise where a variable length array definition is skipped. In the following example

```
{
    int n = 1;
    goto label;

    int a[n];
label:
    // ...
}
```

it is problematic to allocate the array **a** because the **goto** statement causes a jump past the declaration. Therefore, it is forbidden to branch from outside the scope of a variably modified declaration to a point that is inside the scope, although it is permitted to jump from inside the scope to a point outside the scope. In the latter case the translator is expected to deallocate the memory associated with the variable length array. In the following example

```
{
    int n = 1;
label:
    int a[n];
    // ...
    if (n++ < 10) goto label;
}
```

the **goto** statement causes the array **a** to be deallocated. It is reallocated with a new size that is the value of **n** each time the declaration is encountered. Other automatic objects are not deallocated if a **goto** causes them to go out of scope.

See also §6.8.

6.8.6.2 The `continue` statement

The C89 Committee rejected proposed enhancements to **continue** and **break** which would allow specification of an iteration statement other than the immediately enclosing one on grounds of insufficient prior art.

6.8.6.3 The `break` statement

See §6.8.6.2.

6.9 External definitions

6.9.1 Function definitions

A *function definition* may have its old form and say nothing about arguments on calls, or it may be introduced by a *prototype* which affects argument checking and coercion on subsequent calls.

To avoid a nasty ambiguity, the Standard bans the use of **typedef** names as formal parameters. For instance, in translating the text

```
int f(size_t, a_t, b_t, c_t, d_t, e_t, f_t, g_t,
      h_t, i_t, j_t, k_t, l_t, m_t, n_t, o_t,
      p_t, q_t, r_t, s_t)
```

the translator determines that the construct can only be a prototype declaration as soon as it scans the first **size_t** and following comma. In the absence of this rule, it might be necessary to see the token following the right parenthesis that closes the parameter list, which would require a sizable look-ahead, before deciding whether the text under scrutiny is a prototype declaration or an old-style function header definition.

An argument list must be explicitly present in the declarator; it cannot be inherited from a **typedef** (see §6.7.5.3). That is to say, given the definition:

```
typedef int p(int q, int r);
```

the following fragment is invalid:

```
p funk    // weird
{ return q + r ; }
```

Some current implementations rewrite the type of, for instance, a **char** parameter as if it were declared **int**, since the argument is known to be passed as an **int** in the absence of a prototype. The Standard requires, however, that the received argument be converted *as if* by assignment upon function entry. Type rewriting is thus no longer permissible.

QUIET CHANGE IN C89

> Functions that depend on **char** or **short** parameter types being widened to **int**, or **float** widened to **double**, may behave differently.

Notes for implementors: the assignment conversion for argument passing often requires no executable code. In most two's-complement machines, a **short** or **char** is a contiguous subset of the bytes comprising the **int** actually passed for even the most unusual byte orderings, so that assignment conversion can be effected by adjusting the address of the argument if necessary.

For an argument declared **float**, however, an explicit conversion must usually be performed from the **double** actually passed to the **float** desired. Not many implementations can subset the bytes of a **double** to get a **float**. Even those that apparently permit simple truncation often get the wrong answer on certain negative numbers.

Some current implementations permit an argument to be masked by a declaration of the same identifier in the outermost block of a function. This usage is almost always an erroneous attempt by a novice C programmer to declare the argument; it is rarely the result of a deliberate attempt to render the argument unreachable. The C89 Committee decided, therefore, that arguments are effectively declared in the outermost block, and hence cannot be quietly redeclared in that block.

The C89 Committee considered it important that a function taking a variable number of arguments, **printf** for example, be expressible portably in C. Hence, the C89 Committee devoted much time to exploring methods of traversing variable argument lists. One proposal was to require arguments to be passed as a "brick," that is, a contiguous area of memory, the layout of which would be sufficiently well specified that a portable method of traversing the brick could be determined.

Several diverse implementations, however, can implement argument passing more efficiently if the arguments are not required to be contiguous. Thus, the C89 Committee decided to hide the implementation details of determining the location of successive elements of an argument list behind a standard set of macros (see §7.15).

The rule which caused undeclared parameters in an old-style function definition to be implicitly declared **int** has been removed: undeclared parameters are now a constraint violation. The effect is to guarantee production of a diagnostic that will catch an additional category of programming errors. After issuing the diagnostic, an implementation may choose to assume an implicit **int** declaration and continue translation in order to support existing programs that exploited this feature.

6.10 Preprocessing directives

Different implementations have had different notions about whether white space is permissible before and/or after the # signalling a preprocessor line. The C89 Committee decided to allow any white space before the #, and horizontal white space (spaces or tabs) between the # and the directive, since the white space introduces no ambiguity, causes no particular processing problems, and allows maximum flexibility in coding style. Note that similar considerations apply for comments, which are reduced to white space early in the phases of translation (§5.1.1.2):

```
 /* here a comment */ #if BLAH
#/* there a comment */ if BLAH
# if /* every-
     where a comment */ BLAH
```

The lines all illustrate legitimate placement of comments.

6.10.1 Conditional inclusion

For a discussion of evaluation of expressions following **#if**, see §6.6.

The operator **defined** was added to C89 to make possible writing boolean combinations of defined flags with one another and with other inclusion conditions. If the identifier **defined** were to be defined as a macro, **defined(X)** would mean the macro expansion in C text proper and the operator expression in a preprocessing directive (or else that the operator would no longer be available). To avoid this problem, such a definition is not permitted (§6.10.8).

#elif was added to minimize the stacking of **#endif** directives in multi-way conditionals.

Processing of skipped material is defined such that an implementation need only examine a logical line for the **#** and then for a directive name. Thus, assuming that **xxx** is undefined, in this example:

```
# ifndef xxx
# define xxx "abc"
# elif xxx > 0
    // ...
# endif
```

an implementation is not required to diagnose an error for the **#elif** directive, even though if it *were* processed, a syntax error would be detected.

Various proposals were considered for permitting text other than comments at the end of directives, particularly **#endif** and **#else**, presumably to label them for more easily matching their corresponding **#if** directives. The C89 Committee rejected all such proposals because of the difficulty of specifying exactly what would be permitted and how the translator would have to process it.

Various proposals were considered for permitting additional unary expressions to be used for the purpose of testing for the system type, testing for the presence of a file before **#include**, and other extensions to the preprocessing language. These proposals were all rejected on the grounds of insufficient prior art and/or insufficient utility.

6.10.2 Source file inclusion

Specification of the **#include** directive raises distinctive grammatical problems because the file name is conventionally parsed quite differently from an “ordinary” token sequence:

- The angle brackets are not operators, but delimiters.
- The double quotes do not delimit a string literal with all its defined escape sequences (in some systems, backslash is a legitimate character in a filename); the

construct just looks like a string literal.

- White space or characters not in the C repertoire may be permissible and significant within either or both forms.

These points in the description of phases of translation are of particular relevance to the parse of the **#include** directive:

- Any character otherwise unrecognized during tokenization is an instance of an "invalid token." As with valid tokens, the spelling is retained so that later phases can map a token sequence back into a sequence of characters if necessary.

- Preprocessing phases must maintain the spelling of preprocessing tokens; the filename is based on the original spelling of the tokens, not on any interpretation of escape sequences.

- The filename on the **#include** and **#line** directives, if it does not begin with **"** or **<**, is macro-expanded prior to execution of the directive. Allowing macros in the **#include** directive facilitates the parameterization of include file names, an important issue in transportability.

The file search rules used for the filename in the **#include** directive were left as implementation-defined. The Standard intends that the rules which are eventually provided by the implementor correspond as closely as possible to the original K&R rules. The primary reason that explicit rules were not included in the Standard is the infeasibility of describing a portable file system structure. It was considered unacceptable to include UNIX-like directory rules due to significant differences between this structure and other popular commercial file system structures.

Nested include files raise an issue of interpreting the file search rules. In UNIX C a **#include** directive found within an included file entails a search for the named file relative to the file system *directory* that holds the outer **#include**. Other implementations, including the earlier UNIX C described in K&R, always search relative to the same *current directory*. The C89 Committee decided in principle in favor of K&R approach, but was unable to provide explicit search rules as explained above.

The Standard specifies a set of include file names which must map onto distinct host file names. In the absence of such a requirement, it would be impossible to write portable programs using included files.

Subclause §5.2.4.1 on translation limits contains the required number of nesting levels for included files. The limits chosen were intended to reflect reasonable needs for users constrained by reasonable system resources available to implementors.

By defining a failure to read an included file as a syntax error, the Standard requires that the failure be diagnosed. More than one proposal was presented for some form of conditional include, or a directive such as **#ifincludable**, but none were accepted by the Committee due to lack of prior

art.

In C9X, the number of significant characters in header and source file names was raised from six to eight, and digits were allowed, in the belief that all implementations could support this, and because it could be of help to users.

6.10.3 Macro replacement

The specification of macro definition and replacement in the Standard was based on these principles:

- Interfere with existing code as little as possible.
- Keep the preprocessing model simple and uniform.
- Allow macros to be used wherever functions can be.
- Define macro expansion such that it produces the same token sequence whether the macro calls appear in open text, in macro arguments, or in macro definitions.

Preprocessing is specified in such a way that it can be implemented either as a separate text-to-text prepass or as a token-oriented portion of the compiler itself. Thus, the preprocessing grammar is specified in terms of tokens.

However the newline character must be a token during preprocessing because the preprocessing grammar is line-oriented. The presence or absence of white space is also important in several contexts, such as between the macro name and a following parenthesis in a **`#define`** directive. To avoid overly constraining the implementation, the Standard allows both the preservation of each white space character (which is easy for a text-to-text prepass) and the mapping of white space into a single “white space” token (which is easier for token-oriented translators).

The Committee desired to disallow “pernicious redefinitions” such as

(in header1.h)

```
#define NBUFS 10
```

(in header2.h)

```
#define NBUFS 12
```

which are clearly invitations to serious bugs in a program. There remained, however, the question of “benign redefinitions,” such as

(in header1.h)

```
#define NULL_DEV /* the first time */ 0
```

(in header2.h)

```
#define NULL_DEV /* the second time */ 0
```

The C89 Committee concluded that safe programming practice is better served by allowing benign redefinition where the definitions are the same. This allows independent headers to specify their understanding of the proper value for a symbol of interest to each, with diagnostics generated only if the definitions differ.

The definitions are considered "the same" if the identifier-lists, token sequences, and occurrences of white space (ignoring the spelling of white space) in the two definitions are identical.

Pre-C89 implementations differed on whether keywords could be redefined by macro definitions. The C89 Committee decided to allow this usage; it saw such redefinition as useful during the transition from existing to conforming translators.

These definitions illustrate possible uses:

```
# define char   signed char
# define sizeof (int) sizeof
# define const
```

The first case might be useful in moving extant code from an implementation in which plain **char** is signed to one in which it is unsigned. The second case might be useful in adapting code which assumes that the **sizeof** operator yields an **int** value. The redefinition of **const** could be useful in retrofitting more modern C code to an older implementation.

As with any other powerful language feature, keyword redefinition is subject to abuse. Users cannot expect any meaningful behavior to come about from source files starting with

```
#define int double
#include <stdio.h>
```

or similar subversions of common sense.

A new feature of C9X: C89 introduced a standard mechanism for defining functions with variable numbers of arguments, but did not allow any way of writing macros with the same property. For example, there is no way to write a macro that looks like a call to **printf**.

This facility is now available. The macro definition uses an ellipsis in the same way to indicate a variable argument list. However, since macro substitution is textual rather than run-time, a different mechanism is used to indicate where to substitute the arguments: the identifier **__VA_ARGS__**. This is replaced by all the arguments that match the ellipsis, including the commas between them.

For example, the following macro gives a “debugging **printf**”:

```
#ifdef DEBUG
#define dfprintf(stream, ...) \
    fprintf(stream, "DEBUG: " __VA_ARGS__)
#else
#define dfprintf(stream, ...) ((stream, __VA_ARGS__, 0))
#endif

#define dprintf(...) dfprintf(stderr, __VA_ARGS__)
```

For example,

```
dprintf("X = %d\n", x);
```

expands to

```
dfprintf(stderr, "X = %d\n", x);
```

and thus to one of

```
fprintf(stderr, "DEBUG: " "X = %d\n", x);
```

or

```
((stderr, "X = %d\n", x, 0));
```

If **DEBUG** is true, this calls **fprintf**, but first catenating **"DEBUG: "** to the format (which must therefore be a simple string). Otherwise it creates a comma expression (so that the arguments are still evaluated) with the value zero.

There must be at least one argument to match the ellipsis. This requirement avoids the problems that occur when the trailing arguments are included in a list of arguments to another macro or function. For example, if **dprintf** had been defined as

```
#define dprintf(format,...) \
    dfprintf(stderr, format, __VA_ARGS__)
```

and it were allowed for there to be only one argument, then there would be a trailing comma in the expanded form. While some implementations have used various notations or conventions to work around this problem, the Committee felt it better to avoid the problem altogether. Similarly, the **__VA_ARGS__** notation was preferred to other proposals for this syntax.

A new feature of C9X: Function-like macro invocations may also now have empty arguments, that is, an argument may consist of no preprocessing tokens. In C89, any argument that consisted of no preprocessing tokens had undefined behavior, but was noted as a common extension.

A function-like macro invocation **f()** has the form of either a call with no arguments or a call with one empty argument. Which form it actually takes is determined by the definition of **f**, which indicates the expected number of arguments.

The sequence

```
#define TENTH 0.1
#define  F f
#define  D     // expands into no preprocessing tokens
#define LD L
#define  glue(a, b) a ## b
#define xglue(a, b) glue(a, b)

float       f = xglue(TENTH,F) ;
double      d = xglue(TENTH,D) ;
long double ld = xglue(TENTH,LD);
```

results in

```
float       f = 0.1f ;
double      d = 0.1 ;
long double ld = 0.1L;
```

The expansion of **xglue(TENTH,D)** first expands into **glue(0.1,)** which is a macro invocation with an empty second argument, which then expands into **0.1**.

6.10.3.2 The # operator

Some pre-C89 implementations decided to replace identifiers found within a string literal if they matched a macro argument name. The replacement text is a "stringized" form of the actual argument token sequence. This practice appears to be contrary to K&R's definition of preprocessing in terms of token sequences. The C89 Committee declined to elaborate the syntax of string literals to the point where this practice could be condoned; however, since the facility provided by this mechanism seems to be widely used, the C89 Committee introduced a more tractable mechanism of comparable power.

The **#** operator, which may be used only in a **#define** expansion, was introduced for stringizing. It causes the formal parameter name following to be replaced by a string literal formed by stringizing the actual argument token sequence. In conjunction with string literal concatenation (see §6.4.5), use of this operator permits the construction of strings as effectively as by identifier replacement within a string. An example in the Standard illustrates this feature.

One problem with defining the effect of stringizing is the treatment of white space occurring in macro definitions. Where this could be discarded in the past, now upwards of one logical line may have to be retained. As a compromise between token-based and character-based preprocessing disciplines, the C89 Committee decided to permit white space to be retained as one bit of

information: none or one. Arbitrary white space is replaced in the string by one space character.

The remaining problem with stringizing was to associate a "spelling" with each token. The problem arises in token-based preprocessors that might, for instance, convert a numeric literal to a canonical or internal representation, losing information about base, leading zeros, etc. In the interest of simplicity, the C89 Committee decided that each token should expand to just those characters used to specify it in the original source text.

QUIET CHANGE IN C89

A macro that relies on formal parameter substitution within a string literal will produce different results.

6.10.3.3 The ## operator

Another facility relied on in much current practice but not specified in K&R is "token pasting," or building a new token by macro argument substitution. One pre-C89 implementation replaced a comment within a macro expansion by no characters instead of the single space called for in K&R. The C89 Committee considered this practice unacceptable.

As with "stringizing," the facility was considered desirable, but not the extant implementation of this facility, so the C89 Committee invented another preprocessing operator. The **##** operator within a macro expansion causes concatenation of the tokens on either side of it into a new composite token.

The specification of this pasting operator is based on these principles:

- Paste operations are explicit in the source.
- The **##** operator is associative.
- A formal parameter as an operand for **##** is not expanded before pasting. The actual parameter is substituted for the formal parameter; but the actual parameter is not expanded. Given, for example

```
#define a(n) aaa ## n
#define b    2
```

  the expansion of **a(b)** is **aaab**, not **aaa2** or **aaan**.
- A normal operand for **##** is not expanded before pasting.
- Pasting does not cross macro replacement boundaries.
- The token resulting from a paste operation is subject to further macro expansion.

These principles codify the essential features of prior art and are consistent with the specification of the stringizing operator.

6.10.3.4 Rescanning and further replacement

A problem faced by many pre-C89 preprocessors is how to use a macro name in its expansion without suffering "recursive death." The C89 Committee agreed simply to turn off the definition of a macro for the duration of the expansion of that macro. An example of this feature is included in the Standard.

The rescanning rules incorporate an ambiguity. Given the definitions

```
#define  f(a)  a*g
#define  g     f
```

it is clear (or at least unambiguous) that the expansion of **`f(2)(9)`** is **`2*f(9)`**, the **`f`** in the result being introduced during the expansion of the original **`f`**, and so is not further expanded.

However, given the definitions

```
#define f(a)  a*g
#define g(a)  f(a)
```

the expansion will to be either **`2*f(9)`** or **`2*9*g`**: there are no clear grounds for making a decision whether the **`f(9)`** token string resulting from the initial expansion of **`f`** and the examination of the rest of the source file should be considered as nested within the expansion of **`f`** or not. The C89 Committee intentionally left this behavior ambiguous as it saw no useful purpose in specifying all the quirks of preprocessing for such questionably useful constructs.

6.10.3.5 Scope of macro definitions

Some pre-C89 implementations maintained a stack of **`#define`** instances for each identifier, and **`#undef`** simply popped the stack. The C89 Committee agreed that more than one level of **`#define`** was more prone to error than utility.

It is explicitly permitted to **`#undef`** a macro that has no current definition. This capability is exploited in conjunction with the standard library (see §7.1.4).

6.10.4 Line control

Aside from giving values to **`__LINE__`** and **`__FILE__`** (see §6.10.8), the effect of **`#line`** is unspecified. A good implementation will presumably provide line and file information in conjunction with most diagnostics.

A new proposal for C9X to allow the **`#line`** directive to appear within macro invocations was considered. The Committe decided to not allow any preprocessor directives to be recognized as

such inside of macros.

6.10.5 Error directive

The **#error** directive was introduced in C89 to provide an explicit mechanism for forcing translation to fail under certain conditions. Formally, the Standard can require only that a diagnostic be issued when the **#error** directive is processed. It is the intent of the Committee, however, that translation cease immediately upon encountering this directive if this is feasible in the implementation. Further diagnostics on text beyond the directive are apt to be of little value.

6.10.6 Pragma directive

The **#pragma** directive was added in C89 as the universal method for extending the space of directives.

A new feature of C9X: Some **#pragma** directives have been standardized; and directives whose first preprocessing token is **STDC** are reserved for standardized directives.

6.10.7 Null directive

The existing practice of using empty # lines for spacing is supported in the Standard.

6.10.8 Predefined macro names

The rule that these macros may not be redefined or undefined reduces the complexity of the name space that the programmer and implementor must understand; and it recognizes that these macros have special built-in properties.

The macros __**DATE**__ and __**TIME**__ were added in C89 to make available the time of translation. A particular format for the expansion of these macros was specified to aid in parsing strings initialized by them.

The macros __**LINE**__ and __**FILE**__ were added in C89 to give programmers access to the source line number and file name.

The macro __**STDC**__ allows for conditional translation on whether the translator claims to be standard-conforming. It is defined as having the value 1. Future versions of the Standard could define it as 2, 3, etc., to allow for conditional compilation on which version of the Standard a translator conforms to. The C89 Committee felt that this macro would be of use in moving to a conforming implementation.

The macro __**STDC_VERSION**__ was added in C95.

A new feature of C9X: C9X adds two additional predefined macros: __**STDC_IEC_559**__ and

__STDC_IEC_559_COMPLEX__.

6.10.9 Pragma operator

A new feature of C9X: As an alternative syntax for a **#pragma** directive, the **_Pragma** operator has the advantage that it can be used in a macro replacement list. If a translator is directed to produce a preprocessed version of the source file, then expressions involving the unary **_Pragma** operator and **#pragma** directives should be treated consistently in whether they are preserved and in whether macro invocations within them are expanded.

6.11 Future language directions

This subclause includes specific mention of the future direction in which the Committee intends to extend and/or restrict the language. The contents of this subclause should be considered as quite likely to become a part of the next version of the Standard. Implementors are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming implementation. Users are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming program.

6.11.5 Storage-class specifiers

The practice of placing the storage class specifier other than first in a declaration was branded as *obsolescent*. The Committee felt it desirable to rule out such constructs as

```
enum { aaa, aab,
    /* etc. */
zzy, zzz } typedef a2z;
```

in some future standard.

6.11.6 Function declarators

The characterization as obsolescent of the use of the "old style" function declarations and definitions, that is, the traditional style not using prototypes, signals the Committee's intent that the new prototype style should eventually replace the old style.

The case for the prototype style is presented in §6.5.2.2 and §6.7.5.3. The gist of this case is that the new syntax addresses some of the most glaring weaknesses of the language defined in K&R, that the new style is superior to the old style on every count.

It was obviously out of the question to remove syntax used in the overwhelming majority of extant C code, so the Standard specifies two ways of writing function declarations and function definitions. Characterizing the old style as obsolescent is meant to discourage its use and to serve as a strong endorsement by the Committee of the new style. It confidently expects that approval and adoption of the prototype style will make it feasible for some future C Standard to remove the

old style syntax.

7. Library

7.1 Introduction

The base document for this part of the Standard was the *1984 /usr/group Standard*. The /usr/group document contains definitions of some facilities which were specific to the UNIX operating system and not relevant to other operating environments, such as pipes, **ioctl**, file access permissions and process control facilities. Those definitions were dropped from C89. Other functions were excluded as well because they were non-portable or were ill-defined.

Other facilities not in the /usr/group library but present in many UNIX implementations, such as the curses (terminal-independent screen handling) library were considered to be more complex and less essential than the facilities of /usr/group; these functions were not added to the Standard.

The prototypes for several library routines were changed in C9X and they now contain the new keyword **restrict** as part of some parameter declarations. The **restrict** keyword allows the prototype to express what was previously expressed by words.

The definition of certain C library routines such as **memcpy** contain the words:

> If copying takes place between objects that overlap, the behavior is undefined.

These words are present because copying between overlapping objects is quite rare, and this allowed vendors to provide efficient implementations of these library routines. Now that **restrict** allows users to express these same non-overlapping semantics, it is used in prototype declarations to demonstrate the utility of the keyword, and to act as guidance to those wishing to understand how to use it correctly.

In the case of **memcpy** above, the prototype is now declared as:

```
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
```

and the **restrict** keywords tell the translator that the first two parameters, **s1** and **s2**, are pointers that point to disjoint data objects. Essentially, this keyword provides the same information as the words that indicate copying between overlapping objects is not allowed.

Besides the library functions whose specifications state that copying between overlapping objects is not allowed, several others have also had their prototype adorned with the **restrict** keyword. For example:

```
int printf(const char * restrict format, ...);
```

A critical question that one asks when deciding if a pointer parameter should be **restrict**-

qualified or not is, if copying takes place between overlapping objects, will the function behave as expected. In the case of the **printf** function, unexpected behavior occurs in a call such as:

```
{
    int *p = malloc(n * sizeof(int));
    char *cp = (char *) p;
    strcpy(cp, "%s %n %s\n");

    printf(cp, "string1", p, "string2");
}
```

The unexpected behavior occurs because:

1. character pointers can alias other pointers to objects.
2. **p** and **cp** are aliases for the same dynamic object allocated by the call to the **malloc** function.
3. the **%n** specifier causes an integer value to overwrite the string pointed to by **cp** through **p**.

Remember that the **const** qualifier in the **printf** prototype only guarantees that the parameter pointing at the format string is read-only. Another alias, **p**, is allowed to modify the same format string.

Since the implementation costs are high if vendors are forced to cater to this extremely rare case, the **restrict** keyword is used to explicitly forbid situations like these.

Another library routine that uses **restrict** is:

```
char *fgets(char * restrict s, int n,
            FILE * restrict stream);
```

Again, since a character pointer can be a potential alias with other pointers, **restrict** is used to make it clear to the translator that parameter **s** is never an alias with parameter **stream** when the **fgets** function is called in a strictly conforming program.

Finally, the prototypes of certain library functions are adorned with **restrict** only if the pointer is used to access data. For example:

```
wchar_t *wcstok(wchar_t * restrict s1,
                const wchar_t * restrict s2,
                wchar_t ** restrict ptr);
```

The parameter **ptr** only has a **restrict** qualifier on the top-level pointer type. The reason the parameter declaration is not

```
wchar_t * restrict * restrict ptr
```

is that only the top-level pointer type is used to access an object. The lower-level pointer type is

only used to track the location in the wide character string where the search terminated. Thus there is no possibility of copying taking place between overlapping objects through the lower-level pointer.

In general, a **restrict**-qualified pointer provides useful information in the prototype of a library routine if more than one parameter with pointer type can alias each other. Sometimes the aliasing rules prevent this from happening (for example, a pointer to an integer type cannot alias a pointer to a floating-point type). When the aliasing rules allow two pointers to point at overlapping objects, then the **restrict** keyword can be used to indicate that this function should not be called with pointers to overlapping objects. This guideline also applies outside of the library if a parameter can alias a file-scope pointer.

7.1.1 Definitions of terms

The *decimal-point character* is the single character used in the input or output of floating point numbers, and may be changed by **setlocale**. This is a library construct; the decimal point in numeric literals in C source text is always a period.

7.1.2 Standard headers

Whereas in pre-C89 practice only certain library functions were associated with header files, C89 mandated that *each* library function be declared in some header. Several headers were therefore added, and the contents of a few old ones were changed, in each new Standard, C89, C95 and C9X.

In many implementations the names of headers are the names of files in special directories. This implementation technique is not required, however: the Standard makes no assumptions about the form that a file name may take on any system. Headers may thus have a special status if an implementation so chooses. Standard headers may even be built into a translator, provided that their contents do not become "known" until after they are explicitly included. One purpose of permitting these header "files" to be "built in" to the translator is to allow an implementation of the C language as an interpreter in a free-standing environment where the only "file" support may be a network interface.

The C89 Committee decided to make library headers "idempotent," that is, they should be includable any number of times, and includable in any order. This requirement, which reflects widespread existing practice, may necessitate some protective wrappers within the headers to avoid, for instance, redefinitions of **typedef**s. To ensure that such protective wrapping can be made to work, and to ensure proper scoping of **typedef**s, standard headers may only be included outside of any declaration.

A common way of providing this "protective wrapping" is

```
#ifndef __ERRNO_H
#define __ERRNO_H
/* body of <errno.h> */
```

```
/* ... */
#endif
```

where __**ERRNO_H** is an otherwise unused macro name.

Implementors often desire to provide implementations of C in addition to that prescribed by the Standard. For instance, an implementation may want to provide system-specific I/O facilities in **<stdio.h>**. A technique that allows the same header to be used in both the conforming and alternate implementations is to add the extra, non-Standard declarations to the header as in

```
#ifdef __EXTENSIONS__
typedef int file_no;
extern int read(file_no _N, void * _Buffer, int _Nbytes);
// ... */
#endif
```

The header is usable in a strictly conforming program in the absence of a definition of __**EXTENSIONS**__.

7.1.3 Reserved identifiers

To give implementors maximum latitude in packing library functions into files, all external identifiers defined by the library are reserved in a hosted environment. This means, in effect, that no user-supplied external names may match library names, *not even if the user function has the same specification*. Thus, for instance, **strtod** may be defined in the same object module as **printf**, with no fear that link-time conflicts will occur. Equally, **strtod** may call **printf**, or **printf** may call **strtod**, for whatever reason, with no fear that the wrong function will be called.

Also reserved for the implementor are *all* external identifiers beginning with an underscore, and all other identifiers beginning with an underscore followed by a capital letter or an underscore. This gives a name space for writing the numerous behind-the-scenes non-external macros and functions a library needs to do its job properly.

With these exceptions, the Standard assures the programmer that *all other* identifiers are available, with no fear of unexpected collisions when moving programs from one implementation to another[5]. Note, in particular, that part of the name space of internal identifiers beginning with underscore is available to the user: translator implementors have not been the only ones to find use for "hidden" names. C is such a portable language in many respects that the issue of "name space pollution" has been and is one of the principal barriers to writing completely portable code. Therefore the Standard assures that macro and **typedef** names are reserved only if the associated header is explicitly included.

[5] See §6.2.2.1 for a discussion of some of the precautions an implementor should take to keep this promise. Note also that any implementation-defined member names in structures defined in **<time.h>** and **<locale.h>** must begin with an underscore, rather than following the pattern of other names in those structures.

7.1.4 Use of library functions

To make usage more uniform for both implementor and programmer, the Standard requires that every library function, unless specifically noted otherwise, must be represented as an actual function, in case a program wishes to pass its address as a parameter to another function. On the other hand, every library function is now a candidate for redefinition in its associated header as a macro, provided that the macro performs a "safe" evaluation of its arguments, that is, it evaluates each of the arguments exactly once and parenthesizes them thoroughly; and provided that its top-level operator is such that the execution of the macro is not interleaved with other expressions. Two exceptions are the macros **`getc`** and **`putc`**, which may evaluate their arguments in an unsafe manner (see §7.19.7.5 and §7.19.7.8).

If a program requires that a library facility be implemented as an actual function, not as a macro, then the macro name, if any, may be erased by using the **`#undef`** preprocessing directive (see §6.10.3.5).

All library prototypes are specified in terms of the "widened" types: an argument formerly declared as **`char`** is now written as **`int`**. This ensures that most library functions can be called with or without a prototype in scope, thus maintaining backwards compatibility with pre-C89 code. Note, however, that since functions like **`printf`** and **`scanf`** use variable-length argument lists, they must be called in the scope of a prototype.

The Standard contains an example showing how certain library functions may be "built in" in an implementation that remains conforming.

Unlike in C89, some names are no longer unique in the first six characters. Such uniqueness is unnecessary because C9X no longer allows that minimum translation limit.

7.2 Diagnostics `<assert.h>`

7.2.1 Program diagnostics

7.2.1.1 The `assert` macro

Some pre-C89 implementations tolerated an arbitrary scalar expression as the argument to **`assert`**, but the C89 Committee decided to require correct operation only for **`int`** expressions. For the sake of implementors, no hard and fast format for the output of a failing assertion is required; but the Standard mandates enough machinery to replicate the form shown in the footnote.

It can be difficult or impossible to make **`assert`** a true function, so it is restricted to macro form only.

To minimize the number of different methods for program termination, **`assert`** is now defined in terms of the **`abort`** function.

Note that defining the macro **NDEBUG** to disable assertions may change the behavior of a program with no failing assertion if any argument expression to assert has side effects, because the expression is no longer evaluated.

It is possible to turn assertions off and on in different functions within a translation unit by defining or undefining **NDEBUG** and including **<assert.h>** again. The implementation of this behavior in **<assert.h>** is simple: undefine any previous definition of **assert** before providing the new one. Thus the header might look like

```
#undef assert
#ifdef NDEBUG
#define  assert(ignore)  ((void)0)
#else
extern void __gripe(char *_Expr, char *_File,
                    int _Line, const char *_Func);
#define assert(expr) \
    ((expr) ? (void)0 :\
    __gripe(#expr, __FILE__, __LINE__, __func__))
#endif
```

Note that **assert** must expand to a **void** expression, so the more obvious **if** statement does not suffice as a definition of **assert**. Note also the avoidance of names in a header that would conflict with the user's name space.

7.3 Complex arithmetic <complex.h>

A new feature of C9X.

The choice of **I** instead of **i** for the imaginary unit concedes to the widespread use of the identifier **i** for other purposes. The programmer can use a different identifier, say **j**, for the imaginary unit by following the inclusion of **<complex.h>** with

```
#undef I
#define j _Imaginary_I
```

An **I** suffix to designate imaginary constants is not required, as multiplication by **I** provides a sufficiently convenient and more generally useful notation for imaginary terms.

The corresponding real type for the imaginary unit is **float** so that use of **I** for algorithmic or notational convenience will not result in widening types.

On systems with imaginary types, the programmer has the ability to control whether use of the macro **I** introduces an imaginary type, by explicitly defining **I** to be **_Imaginary_I** or **_Complex_I**. Disallowing imaginary types is useful for some programs intended to run on implementations without support for such types.

The macro **_Imaginary_I** provides a test for whether imaginary types are supported (whether or not the implementation fully supports Annex G).

The **cis** function (**cos(x) + I*sin(x)**) was considered but rejected because its implementation is easy and straightforward, even though some implementations could compute sine and cosine more efficiently in tandem.

7.3.9 Manipulation functions

7.3.9.4 The cproj function

Two topologies are commonly used in complex mathematics: the complex plane with its continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. The complex types with their multiplicity of infinities provide a useful (though imperfect) model for the complex plane. The **cproj** function helps model the Riemann sphere by mapping all infinities to one, and should be used just before any operation, especially comparisons, that might give spurious results for any of the other infinities.

Note that a complex value with one infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is infinite, the complex value is infinite independent of the value of the other part. For the same reason, **cabs** returns an infinity if its argument has an infinite part and a NaN part.

7.4 Character Handling <ctype.h>

Pains were taken to eliminate any ASCII dependencies from the definition of the character handling functions. One notable result of this policy was the elimination of the function **isascii**, both because of the name and because its function was hard to generalize. Nevertheless, the character functions are often most clearly explained in concrete terms, so ASCII is used frequently to express examples.

Since these functions are often used primarily as macros, their domain is restricted to the small positive integers representable in an **unsigned char**, plus the value of **EOF**. **EOF** is traditionally **-1**, but may be any negative integer, and hence distinguishable from any valid character code. These macros may thus be efficiently implemented by using the argument as an index into a small array of attributes.

§7.26.2 warns that names beginning with **is** and **to**, when these are followed by lower-case letters, are subject to future use in adding items to **<ctype.h>**.

7.4.1 Character classification functions

The definitions of *printing character* and *control character* have been generalized from ASCII.

Note that none of these functions returns a nonzero value (true) for the argument value **EOF**.

7.4.1.2 The isalpha function

The Standard specifies that the set of letters, in the default *locale*, comprises the 26 upper-case and 26 lower-case letters of the Latin (English) alphabet. This set may vary in a *locale-specific* fashion (that is, under control of the **setlocale** function, see §7.11.1.1) so long as

- **isupper(c)** implies **isalpha(c)**
- **islower(c)** implies **isalpha(c)**
- **isspace(c)**, **ispunct(c)**, **iscntrl(c)**, and **isdigit(c)** all imply **!isalpha(c)**

7.4.1.3 The isblank function

A new feature of C9X: text processing applications often need to distinguish white space that can occur within lines from white space that also separates lines (for example, see §6.10 regarding use of whitespace in the preprocessor). This distinction is also a property of POSIX locale definition files.

7.4.1.10 The isspace function

isspace is widely used within the library as the working definition of white space.

7.4.2 Character case mapping functions

Pre-C89 libraries had almost equivalent macros, **_tolower** and **_toupper**, for these functions. The Standard now permits any library function to be additionally implemented as a macro provided that the underlying function must still be present. **_toupper** and **_tolower** are thus unnecessary and were dropped as part of the general standardization of library macros.

7.5 Errors <errno.h>

<errno.h> is a header invented to encapsulate the error handling mechanism used by many of the library routines in **<math.h>** and **<stdlib.h>**.[6]

The error reporting machinery centered about the setting of **errno** is generally regarded with tolerance at best. It requires a "pathological coupling" between library functions and makes use of

[6] In early drafts of C89, **errno** and related macros were defined in **<stddef.h>**. When the C89 Committee decided that the other definitions in this header were of such general utility that they should be required even in freestanding environments, it created **<errno.h>**.

a static writable memory cell, which interferes with the construction of shareable libraries. Nevertheless, the C89 Committee preferred to standardize this existing, however deficient, machinery rather than invent something more ambitious. In C9X, **errno** need no longer be set by math functions.

The definition of **errno** as an lvalue macro grants implementors the license to expand it to something like ***__errno_addr()**, where the function returns a pointer to the current modifiable copy of **errno**.

7.6 Floating-point environment <fenv.h>

A new feature of C9X.

The floating-point environment as defined here includes only execution-time modes, not the myriad of possible translation-time options that can affect a program's results. Each such option's deviation from this specification should be well documented.

Dynamic vs. static modes

Dynamic modes are potentially problematic because

1. the programmer may have to defend against undesirable mode settings, which imposes intellectual as well as time and space overhead.

2. the translator may not know which mode settings will be in effect or which functions change them at execution time, which inhibits optimization.

C9X addresses these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of static modes with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. The IEC 60559 standard requires dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, standard C has no facility, other than pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling **FENV_ACCESS** pragma would need such special treatment.

Translation

An implementation is not required to provide a facility for altering the modes for translation-time

arithmetic, or for making exception flags from the translation available to the executing program. The language and library provide facilities to cause floating-point operations to be done at execution time when they can be subjected to varying dynamic modes and their exceptions detected. The need does not seem sufficient to require similar facilities for translation.

The `fexcept_t` type

`fexcept_t` does not have to be an integer type. Its values must be obtained by a call to **`fegetexceptflag`**, and cannot be created by logical operations from the exception macros. An implementation might simply implement **`fexcept_t`** as an **`int`** and use the representations reflected by the exception macros, but isn't required to: other representations might contain extra information about the exceptions. **`fexcept_t`** might be a **`struct`** with a member for each exception (that might hold the address of the first or last floating-point instruction that caused that exception). C9X makes no claims about the internals of an **`fexcept_t`**, and so the user cannot inspect it.

Exception and rounding macros

Unsupported macros are not defined in order to assure that their use results in a translation error. A program might explicitly define such macros to allow translation of code (perhaps never executed) containing the macros. An unsupported exception macro should be defined to be 0, for example

```
#ifndef FE_INEXACT
#define FE_INEXACT 0
#endif
```

so that a bitwise OR of macros has a reasonable effect.

Exceptions

In previous drafts of this specification, several of the exception functions returned an **`int`** indicating whether the **`excepts`** argument represented supported exceptions. This facility was deemed unnecessary because **`excepts & ~FE_ALL_EXCEPT`** can be used to test invalidity of the **`excepts`** argument.

Rounding precision

The IEC 60559 floating-point standard prescribes rounding precision modes (in addition to the rounding direction modes covered by the functions in this section) as a means for systems whose results are always double or extended to mimic systems that deliver results to narrower formats. An implementation of C can meet this goal in any of the following ways:

1. By supporting the evaluation method indicated by **`FLT_EVAL_METHOD`** equal to 0.

2. By providing pragmas or compile options to shorten results by rounding to IEC 60559 single

or double precision.

3. By providing functions to dynamically set and get rounding precision modes which shorten results by rounding to IEC 60559 single or double precision. Recommended are functions **`fesetprec`** and **`fegetprec`** and macros **`FE_FLTPREC`**, **`FE_DBLPREC`**, and **`FE_LDBLPREC`**, analogous to the functions and macros for the rounding direction modes.

This specification does not include a portable interface for precision control because the IEC 60559 floating-point standard is ambivalent on whether it intends for precision control to be dynamic (like the rounding direction modes) or static. Indeed, some floating-point architectures provide control modes suitable for a dynamic mechanism, and others rely on instructions to deliver single- and double-format results suitable only for a static mechanism.

7.6.1 The `FENV_ACCESS` pragma

The performance of code under **`FENV_ACCESS ON`** may well be important; in fact an algorithm may access the floating-point environment specifically for performance. The implementation should optimize as aggressively as the **`FENV_ACCESS`** pragma allows. An implementation could also simply honor the floating-point environment in all cases and ignore the pragma.

The Committee's model is that the regions of code that are under **`FENV_ACCESS OFF`** do not have to maintain the exception flags, even if there are regions with **`FENV_ACCESS ON`** elsewhere in the program.

7.6.2 Floating-point exceptions

7.6.2.3 The `feraiseexcept` function

Raising overflow or underflow is allowed to also raise inexact because on some architectures the only practical way to raise an exception is to execute an instruction that has the exception as a side effect. Any IEC 60559 operation that raises either overflow or underflow raises inexact as well.

The function is not restricted to accept only valid coincident expressions for atomic operations, so the function can be used to raise exceptions accrued over several operations.

7.6.3 Rounding

7.6.3.2 The `fesetround` function

In previous drafts the function returned nonzero to indicate success. This was changed for consistency with other C functions that return a status indicator.

7.6.4 Environment

7.6.4.2 The `feholdexcept` function

In previous drafts the function returned nonzero to indicate success. This was changed for consistency with other C functions that return a status indicator.

`feholdexcept` should be effective on typical IEC 60559 implementations which have the default non-stop mode and at least one other mode for trap handling or aborting. If the implementation provides only the non-stop mode, then installing the non-stop mode is trivial.

A previous draft specified a **`feprocentry`** function, which was equivalent to

```
fegetenv(envp);
fesetenv(FE_DFL_ENV);
```

`feholdexcept` is more appropriate for the user model prescribed in §7.6.

7.7 Characteristics of floating types `<float.h>`

`<float.h>` makes available to programmers a set of useful quantities for numerical analysis (see §5.2.4.2.2). This set of quantities has seen widespread use for such analysis, in C and in other languages, and was recommended by the numerical analysts on the C89 Committee. The set was chosen so as not to prejudice an implementation's selection of floating-point representation. See also §7.10 for general remarks.

Regarding the **`FLT_DIG`** formula, note that the process of converting an arbitrary value in one floating-point format to a second floating-point format and then back again so as not to change the original value requires more precision than might be expected. In general, the formulas for number of digits needed for base conversions for integers do not work when applied to floating-point representations.

If the radix b is a power of 10, then it is obvious that all decimal numbers with $p \times \log_{10} b$ digits convert exactly to a floating-point representation and then convert exactly back to the same decimal number (as long as $p \times \log_{10} b$ is an integer).

When the radix b is not a power of 10, it can be difficult to find a case where a decimal number with $\lfloor p \times \log_{10} b \rfloor$ digits fails. Consider a four-bit mantissa system (that is, base b=2 and precision p=4) used to represent one-digit decimal numbers. While four bits are enough to represent one-digit numbers, they are not enough to support the conversions of decimal to binary and back to decimal in all cases (but they are enough for most cases). Consider a power of 2 that is just under **9.5e21**, for example, 2^{73} = **9.44e21**. For this number, the three consecutive one-digit numbers near that special value and their round-to-nearest representations are:

```
  9e21    1e22    2e22
0xFp69  0x8p70  0x8p71
```

No problems so far; but when these representations are converted back to decimal, the values as

three-digit numbers and the rounded one-digit numbers are:

```
8.85e21  9.44e21  1.89e22
   9e21     9e21     2e22
```

and we end up with two values the same. For this reason, four-bit mantissas are not enough to start with any one-digit decimal number, convert it to a binary floating-point representation, and then convert back to the same one-digit decimal number in all cases; and so p radix b digits are (just barely) not enough to allow any decimal numbers with $\lfloor p \times \log_{10} b \rfloor$ digits to do the round-trip conversion. p radix b digits *are* enough, however, for $\lfloor (p - 1) \times \log_{10} b \rfloor$ digits in all cases.

7.8 Format conversion of integer types <inttypes.h>

A new feature of C9X.

<inttypes.h> was derived from the header of the same name found on several existing 64-bit systems. The Committee debated other methods for specifying integer sizes and other characteristics, but in the end decided to standardize existing practice rather than innovate in this area. (See also §7.18 **<stdint.h>**.)

C89 specifies that the language should support four signed and unsigned integer data types, **char**, **short**, **int** and **long**, but places very little requirement on their size other than that **int** and **short** be at least 16 bits and **long** be at least as long as **int** and not smaller than 32 bits. For 16-bit systems, most implementations assign 8, 16, 16 and 32 bits to **char**, **short**, **int**, and **long**, respectively. For 32-bit systems, the common practice is to assign 8, 16, 32 and 32 bits to these types. This difference in **int** size can create some problems for users who migrate from one system to another which assigns different sizes to integer types, because Standard C's integer promotion rule can produce silent changes unexpectedly. The need for defining an *extended integer* type increased with the introduction of 64-bit systems.

The purpose of **<inttypes.h>** is to provide a set of integer types whose definitions are consistent across machines and independent of operating systems and other implementation idiosyncrasies. It defines, via **typedef**, integer types of various sizes. Implementations are free to **typedef** them as Standard C integer types or extensions that they support. Consistent use of this header will greatly increase the portability of a user's program across platforms.

7.9 Alternate spellings <iso646.h>

See §MSE.4.

7.10 Sizes of integer types <limits.h>

Both **<float.h>** and **<limits.h>** are inventions of the C89 Committee. Included in these headers are various parameters of the execution environment which are potentially useful at compile time, and which are difficult or impossible to determine by other means.

The availability of this information in headers provides a portable way of tuning a program to different environments. Requiring that preprocessing always yield the same results as run-time arithmetic, however, would cause problems for portable compilers (themselves written in C) or for cross-compilers, which would then be required to implement the target machine's arithmetic on the host machine.

7.11 Localization `<locale.h>`

C has become an international language. Users of the language outside the United States have been forced to deal with the various Americanisms built into the standard library routines. Areas affected by international considerations include:

Alphabet. The English language uses 26 letters derived from the Latin alphabet which suffice only for English and Swahili; other living languages use either the Latin alphabet *plus* other characters, or other non-Latin alphabets or syllabaries.

In English, each letter has an upper-case and lower-case form, but this is not generally the case. The German "sharp S", ß, for example, occurs only in lower case. European French usually omits diacritics on upper-case letters. Some scripts do not have the concept of two cases.

Collation. In both EBCDIC and ASCII the code for "z" is greater than the code for "a", and so on for other letters in the alphabet, so a "machine sort" gives not unreasonable results for ordering strings. In contrast, most European languages use a codeset resembling ASCII in which some of the codes used in ASCII for punctuation characters are used for alphabetic characters (see §5.2.1). The ordering of these codes is not alphabetic. In some languages letters with diacritics sort as separate letters; in others they should be collated just as the unmarked form. In Spanish, "ll" sorts as a single letter following "l"; in German, "ß" sorts like "ss".

Formatting of numbers and currency amounts. In the United States the period is invariably used for the decimal point, and this usage was built into the definitions of such functions as **`printf`** and **`scanf`**. Prevalent practice in several major European countries is to use a comma; a raised dot is employed in some locales. Similarly, in the United States a comma is used to separate groups of three digits to the left of the decimal point; but a period is common in Europe, and in some countries digits are not grouped by threes at all. In printing currency amounts, the currency symbol (which may be more than one character) may precede, follow, or be embedded in the digits. Note that the decimal point is a single character, not a multibyte string.

Date and time. The standard function **`asctime`** returns a string which includes abbreviations for month and weekday names, and returns the various elements in a format which might be considered unusual even in its country of origin.

Various common date formats include

1998-07-03	ISO Format
3.7.98	customary central European and British usage
7/3/98	customary U.S. usage
3.VII.98	Italian usage
98183	Julian date (YYDDD)
03JUL98	airline usage
Friday, July 3, 1998	full U.S. format
Freitag, 3. Juli 1998	full German format
den 3 juli 1998	full Swedish format

Time formats are also quite diverse:

3:30 PM	customary U.S. and British format
1530	U.S. military format
15h.30	Italian usage
15.30	German usage
15:30	common European usage

The C89 Committee introduced mechanisms into the C library to allow these and other issues to be treated in the appropriate *locale-specific* manner.

The localization features of the Standard are based on these principles:

English for C source. The C language proper is based on English. Keywords are based on English words. A program which uses "national characters" in identifiers was not strictly conforming through C95, but C9X allows identifiers to be written using the "universal character names" (UCNs) of ISO/IEC 10646. (Use of national characters in comments has always been strictly conforming, though what happens when such a program is printed in a different locale is unspecified.) The decimal point must be a period in C source, and no thousands delimiter may be used.

Runtime selectability. The locale must be selectable at runtime from an implementation-defined set of possibilities. Translation time selection does not offer sufficient flexibility. Software vendors do not want to supply different object forms of their programs in different locales. Users do not want to use different versions of a program just because they deal with several different locales.

Function interface. The locale is changed by calling a function, thus allowing the implementation to recognize the change, rather than by, say, changing a memory location that contains the decimal point character.

Immediate effect. When a new locale is selected, affected functions reflect the change immediately. (This is not meant to imply that, if a signal-handling function were to change the selected locale and return to a library function, the return value from that library function must be completely correct with respect to the new locale.)

7.11.1 Locale control

7.11.1.1 The `setlocale` function

The **`setlocale`** function provides the mechanism for controlling *locale-specific* features of the library. The **`category`** argument allows parts of the library to be localized as necessary without changing the entire locale-specific environment. Specifying the locale argument as a string gives an implementation maximum flexibility in providing a set of locales. For instance, an implementation could map the argument string into the name of a file containing appropriate localization parameters; and these files could then be added and modified without requiring any recompilation of a localizable program.

7.11.2 Numeric formatting convention inquiry

7.11.2.1 The `localeconv` function

The **`localeconv`** function gives a programmer access to information about how to format monetary and non-monetary numeric quantities. This sort of interface was considered preferable to defining conversion functions directly: even with a specified locale, the set of distinct formats that can be constructed from these elements is large; and the ones desired are very application-dependent.

A new feature of C9X: C9X extends the members in the **`lconv`** structure to cover long-standing POSIX practice and to permit additional flexibility for internationally formatted monetary amounts.

7.12 Mathematics `<math.h>`

Before C9X, the math library was defined only for the floating type **`double`**. All the names formed by appending **`f`** or **`l`** to a name in **`<math.h>`** were reserved to allow for the definition of **`float`** and **`long double`** libraries; and C9X provides for all three versions of math functions.

The functions **`ecvt`**, **`fcvt`**, and **`gcvt`** have been dropped since their capability is available through **`sprintf`**.

Before C89, **`HUGE_VAL`** was usually defined as a manifest constant that approximates the largest representable **`double`** value. As an approximation to *infinity* it is problematic. As a function return value indicating overflow, it can cause trouble if first assigned to a **`float`** before testing, since a **`float`** may not necessarily hold all values representable in a **`double`**.

After considering several alternatives, the C89 Committee decided to generalize **`HUGE_VAL`** to a positive expression of type **`double`** so that it could be expressed as an external identifier naming a location initialized precisely with the proper bit pattern. It can even be a special encoding for *machine infinity* on implementations that support such codes. It need not be representable as a **`float`** however. C9X adds **`HUGE_VALF`** and **`HUGE_VALL`**.

Similarly, domain errors before C89 were typically indicated by a zero return, which is not necessarily distinguishable from a valid result. The C89 Committee agreed to make the return value for domain errors *implementation-defined*, so that special machine codes can be used to advantage. This makes possible an implementation of the math library in accordance with the IEC 60559 proposal on floating point representation and arithmetic.

7.12.1 Treatment of error conditions

Whether underflow should be considered a range error and cause **`errno`** to be set is specified as *implementation-defined* since detection of underflow is inefficient on some systems. In C9X, **`errno`** is no longer required to be set to **`EDOM`** or **`ERANGE`** because that is an impediment to optimization.

The Standard has been crafted to neither require nor preclude any popular floating-point implementation. This principle affects the definition of *domain error*: an implementation may define extra domain errors to deal with floating-point arguments such as infinity or "not-a-number" (NaN).

The C89 Committee considered the adoption of the **`matherr`** capability from UNIX System V. In this feature of that system's math library, any error such as overflow or underflow results in a call from the library function to a user-defined exception handler named **`matherr`**. The C89 Committee rejected this approach for several reasons:

- This style is incompatible with popular floating point implementations such as IEC 60559, with its special return codes, or that of VAX/VMS.
- It conflicts with the error-handling style of Fortran, thus making it more difficult to translate useful bodies of mathematical code from that language to C.
- It requires the math library to be reentrant since math routines could be called from **`matherr`**, which may complicate some implementations.
- It introduces a new style of library interface: a user-defined library function with a library-defined name. Note, by way of comparison, the signal and exit handling mechanisms, which provide a way of "registering" user-defined functions.

7.12.2 The `FP_CONTRACT` pragma

A new feature of C9X. See §6.5 and §7.12.13.1.

7.12.3 Classification macros

New features of C9X.

Passing an integer, complex, or other non-floating type to a classification macro yields undefined

behavior.

7.12.3.3 The `isinf` macro

Note that **`isinf(x)`** cannot simply be defined as **`!isfinite(x)`**, because **`!isfinite(NAN)`** is true.

7.12.4 Trigonometric functions

Implementation note: trigonometric argument reduction should be performed by a method that causes no catastrophic discontinuities in the error of the computed result. In particular, methods based solely on naive application of a calculation like

```
x - (2*pi) * (int)(x/(2*pi))
```

are ill-advised.

7.12.4.4 The `atan2` functions

The **`atan2`** function is modeled after Fortran's. It is described in terms of arctan (*y*/*x*) for simplicity. The C89 Committee did not wish to complicate the descriptions by specifying in detail how to determine the appropriate quadrant, since that should be obvious from normal mathematical convention. **`atan2(y,x)`** is well-defined and finite, even when **`x`** is 0; the one ambiguity occurs when both arguments are 0, because at that point any value in the range of the function could logically be selected. Since valid reasons can be advanced for all the different choices that have been made in this situation by various implementations, the Standard preserves the implementor's freedom to return an arbitrary well-defined value such as 0, to report a domain error, or to return a NaN.

7.12.4.7 The `tan` functions

The tangent function has singularities at odd multiples of $\pi/2$, approaching positive infinity from one side and negative infinity from the other. Implementations commonly perform argument reduction using the best machine representation of π; and for arguments to **`tan`** sufficiently close to a singularity, such reduction may yield a value on the wrong side of the singularity. In view of such problems, the C89 Committee recognized that **`tan`** is an exception to the *range error* rule (see §7.12.1) that an overflowing result produces **`HUGE_VAL`** properly signed.

7.12.6 Exponential and logarithmic functions

7.12.6.4 The `frexp` functions

The functions **`frexp`**, **`ldexp`**, and **`modf`** are primitives used by the remainder of the library. There was some sentiment for dropping them for the same reasons that **`ecvt`**, **`fcvt`**, and **`gcvt`** were dropped, but their adherents rescued them for general use. Their use is problematic: on non-

binary architectures, **ldexp** may lose precision and **frexp** may be inefficient.

7.12.6.6 The ldexp functions

See §7.12.6.4.

7.12.6.7 The log functions

Whether **log(0.0)** is a domain error or a range error is arguable. The choice in the Standard, *range error*, is for compatibility with IEC 60559. Some such implementations would represent the result as –∞, in which case no error is raised.

7.12.6.8 The log10 functions

See §7.12.6.7.

7.12.6.9 The log1p functions

See §7.12.6.7.

7.12.6.10 The log2 functions

See §7.12.6.7.

7.12.6.11 The logb functions

The treatment of subnormal *x* follows the recommendation in IEEE 854, which differs from IEEE 754 on this point. Even 754 implementations should follow this definition rather than the one recommended (not required) by 754.

7.12.6.12 The modf functions

See §6.12.6.4.

7.12.6.13 The scalbn and scalbln functions

In earlier versions of the specification, this function was called **scalb**. The name was changed to avoid conflicting with the Single Unix **scalb** function whose second argument is **double** instead of **int**. Single Unix's **scalb** was not included in C9X as its specification of certain special cases is inconsistent with the C9X approach and because the **scalbn** and **scalbln** functions were considered sufficient.

scalbln, whose second parameter has type **long int** is provided because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width **int** format.

7.12.7 Power and absolute-value functions

7.12.7.1 The `cbrt` functions

For some applications, a true cube root function, which returns negative results for negative arguments, is more appropriate than **`pow(x, 1.0/3.0)`**, which returns a NaN for **`x`** less than 0.

7.12.7.2 The `fabs` functions

Adding an absolute value operator was rejected by the C89 Committee. An implementation can provide a built-in function for efficiency.

7.12.7.5 The `sqrt` functions

IEC 60559, unlike the Standard, requires **`sqrt(-0.)`** to return a negatively signed magnitude-zero result. This is an issue on implementations that support a negative floating zero. The Standard specifies that taking the square root of a negative number (in the mathematical sense of less than 0) is a domain error which requires the function to return an *implementation-defined* value. This rule permits implementations to support either the IEC 60559 or vendor-specific floating point representations.

7.12.8 Error and gamma functions

7.12.8.3 The `lgamma` functions

Since the mathematical gamma function increases in value so quickly (it is around 10^{306} for an argument of only 170), the logarithm of gamma extends the useful domain. Also, for computing combinations and permutations, it is the quotient of the (potentially large) gammas that is needed; taking differences of the **`lgamma`**s instead allows for calculations without overflow.

In Single Unix, a call to **`lgamma`** sets an external variable, **`signgam`**, to the sign of **`gamma(x)`**, which is -1 if **`x < 0 && remainder(floor(x), 2) != 0`**.

Note that this specification does not remove the external identifier **`signgam`** from the user's name space. An implementation that supports **`lgamma`**'s setting of **`signgam`** as an extension must still protect the external identifier **`signgam`** if defined by the user.

7.12.8.4 The `tgamma` functions

In many other standards, the meaning of **`gamma`** has changed over the years. Originally, it computed the logarithm of the absolute value of the mathematical gamma function, with an external **`int`**, **`signgam`**, being set to the sign of the gamma function.

Then **gamma** was replaced with **lgamma**, and **gamma** was slated to be withdrawn. About that time, NCEG changed **gamma** to compute the mathematical gamma function, and that is what was adopted into C9X CD1; but it appears that the old meaning of **gamma** has not yet been withdrawn, so there would have been a conflict between C9X and current industry practice. C9X therefore changed the name in the FCD to **tgamma**, meaning "true gamma," to avoid this conflict.

7.12.9 Nearest integer functions

7.12.9.1 The ceil functions

Implementation note: the **ceil** function returns the smallest integer value in double format not less than **x**, even though that integer might not be representable in a C integer type. **ceil(x)** equals **x** for all **x** sufficiently large in magnitude. An implementation that calculates **ceil(x)** as

```
(double)(int)x
```

is ill-advised.

7.12.9.5 The lrint and llrint functions

Previous drafts specified

```
long rinttol(long double);
long long rinttoll(long double);
long roundtol(long double);
long long roundtoll(long double);
```

instead of

```
long lrint(double);
long long llrint(double);
long lround(double);
long long llround(double);
```

together with the **float** and **long double** versions.

There were two changes here. First, the parameter type changed to **double** to match other functions which, like these, return an integer-type result; this makes the interface style more consistent. Second, the names changed to make way for **f**- and **l**-suffixed versions of the functions, which become needed because of the first change (otherwise **rinttoll** could be either the **double** version of the **long long** function or the **long double** version of the **long** function).

For functions with a floating argument and an integer return type, the previous specification took the approach of declaring the parameter to be **long double**. The rationale was to avoid unnecessary multiple versions of the function in the interface. The implementation need not

actually promote a **float** or **double** argument to **long double**, so any potential inefficiency could be avoided.

With the previous interface, however, a programmer would be left to worry about the risk of incurring a costly promotion to **long double**. Also, the current specification seems more consistent with the rest of the interface where all the other functions come in three sizes. (A programmer might initially be surprised not to find **float** and **double** versions.)

7.12.10 Remainder functions

7.12.10.1 The fmod functions

The **fmod** function is defined even if the quotient x/y is not representable. This function is properly implemented by scaled subtraction rather than by division. The Standard defines the result in terms of the formula $x - n \times y$, where n is some integer. This integer need not be representable, and need not even be explicitly computed. Thus implementations are advised not to compute the result using code like

```
x - y * (int)(x/y)
```

Instead, the result can be computed in principle by subtracting **ldexp(y,n)** from x, for appropriately chosen decreasing n, until the remainder is between 0 and x, although efficiency considerations may dictate a different actual implementation.

The result of **fmod(y, 0.0)** is either a domain error or 0; the result always lies between 0 and **y**, so specifying the non-erroneous result as 0 simply recognizes the limit case.

The C89 Committee considered a proposal to use the remainder operator **%** for this function; but it was rejected because the operators in general correspond to hardware facilities, and **fmod** is not supported in hardware on most machines.

7.12.10.3 The remquo functions

The **remquo** functions are intended for implementing argument reductions which can exploit a few low-order bits of the quotient. Note that x may be so large in magnitude relative to y that an exact representation of the quotient is not practical.

7.12.11 Manipulation functions

7.12.11.1 The copysign functions

copysign and **signbit** need not be consistent with each other if the arithmetic is not consistent in its treatment of zeros. For example, the IBM S/370 has instructions to flip the sign bit making it possible to create a negative zero, but $\pm 0.0 \times \pm 1.0$ is always +0.0. In this case, **copysign** will treat –0.0 as positive, while **signbit** will treat it as negative.

7.12.11.3 The `nextafter` functions

It is sometimes desirable to find the next representation after a value in the direction of a previously computed value, maybe smaller, maybe larger. The **`nextafter`** functions have a second floating argument so that the program will not have to include floating-point tests for determining the direction in such situations. (On some machines, these tests may fail due to overflow, underflow or roundoff.)

For the case $x = y$, IEC 60559 recommends that x be returned. This specification differs so that **`nextafter(-0.0, +0.0)`** returns +0.0 and **`nextafter(+0.0, -0.0)`** returns –0.0.

The **`nextafter`** functions can be employed to obtain next values in a particular format. For example, **`nextafterf(x, y)`** will return the next float value after **`(float)x`** in the direction of **`(float)y`** regardless of the evaluation method.

An alternate proposal was to rename the **`double`** version of **`nextafter`** to **`nextafterd`**, retaining **`nextafterf`** and **`nextafterl`** (these three did not have a generic macro), and using the name **`nextafter`** for what is here named **`nexttoward`**. The current specification has a number of advantages:

1. **`nextafter`** and **`nexttoward`** conform to the usual rules for suffixes and type-generic macros. Before, **`nextafterd`** and **`nextafter`** were exceptional on both counts.

2. Without the change, **`nextafterf`** is not the **`float`** version of **`nextafter`**, which is potentially surprising.

3. It better matches prior art, which typically has a **`nextafter`** function with two **`double`** parameters.

7.12.11.4 The `nexttoward` functions

The second parameter of the **`nexttoward`** function has type **`long double`** so that the uncoerced value of the second argument can be used to determine the direction.

7.12.12 Maximum, minimum, and positive difference functions

The names for **`fmax`**, **`fmin`** and **`fdim`** have **`f`** prefixes to allow for extension integer versions following the example of **`fabs`** and **`abs`**.

7.12.13 Floating multiply-add

7.12.13.1 The `fma` functions

In many cases, clever use of floating (fused) multiply-add leads to much improved code; but its

unexpected use by the compiler can undermine carefully written code. The **FP_CONTRACT** macro can be used to disallow use of floating multiply-add; and the **fma** function guarantees its use where desired. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

7.13 Nonlocal jumps <setjmp.h>

jmp_buf must be an array type for compatibility with existing practice: programs typically omit the address operator before a **jmp_buf** argument, even though a pointer to the argument is desired, not the value of the argument itself. Thus, a scalar or structure type is unsuitable. Note that a one-element array of the appropriate type is a valid definition.

7.13.1 Save calling environment

7.13.1.1 The setjmp macro

setjmp is constrained to be a macro only: in some implementations the information necessary to restore context is only available while executing the function making the call to **setjmp**.

One proposed requirement on **setjmp** is that it be usable like any other function, that is, that it be callable in any expression context, and that the expression evaluate correctly whether the return from **setjmp** is direct or via a call to **longjmp**. Unfortunately, any implementation of **setjmp** as a conventional called function cannot know enough about the calling environment to save any temporary registers or dynamic stack locations used part way through an expression evaluation. (A **setjmp** macro seems to help only if it expands to inline assembly code or a call to a special built-in function.) The temporaries may be correct on the initial call to **setjmp**, but are not likely to be on any return initiated by a corresponding call to **longjmp**. These considerations dictated the constraint that **setjmp** be called only from within fairly simple expressions, ones not likely to need temporary storage.

An alternative proposal considered by the C89 Committee was to require that implementations recognize that calling **setjmp** is a special case[7], and hence that they take whatever precautions are necessary to restore the **setjmp** environment properly upon a **longjmp** call. This proposal was rejected on grounds of consistency: implementations are currently *allowed* to implement library functions specially, but no other situations *require* special treatment.

7.13.2 Restore calling environment

7.13.2.1 The longjmp function

The C89 Committee also considered requiring that a call to **longjmp** restore the calling

[7] This proposal was considered prior to the adoption of the stricture that **setjmp** be a macro. It can be considered as equivalent to proposing that the **setjmp** macro expand to a call to a special built-in compiler function.

environment fully, that is, that upon execution of **longjmp**, all local variables in the environment of **setjmp** have the values they did at the time of the **longjmp** call. Register variables create problems with this idea. Unfortunately, the best that many implementations attempt with register variables is to save them in **jmp_buf** at the time of the initial **setjmp** call, then restore them to that state on each return initiated by a **longjmp** call. Since compilers are certainly at liberty to change register variables to automatic, it is not obvious that a register declaration will indeed be rolled back. And since compilers are at liberty to change automatic variables to register if their addresses are never taken, it is not obvious that an automatic declaration will *not* be rolled back, hence the vague wording. In fact, the only reliable way to ensure that a local variable retain the value it had at the time of the call to **longjmp** is to define it with the **volatile** attribute. Note this does not apply to the floating-point environment (status flags and control modes) which is part of the global state just as **f** is.

Some implementations leave a process in a special state while a signal is being handled. Explicit reassurance must be given to the environment when the signal handler returns. To keep this job manageable, the C89 Committee agreed to restrict **longjmp** to only one level of signal handling.

The **longjmp** function should not be called in an exit handler, that is, a function registered with the **atexit** function (see §7.20.4.2), since it might jump to code that is no longer in scope.

7.14 Signal handling <signal.h>

This facility was retained from /usr/group since the C89 Committee felt it important to provide some standard mechanism for dealing with exceptional program conditions. Thus a subset of the signals defined in UNIX were retained in the Standard, along with the basic mechanisms of declaring signal handlers and, with adaptations, raising signals (see §7.14.2.1). For a discussion of the problems created by including signals, see §5.2.3.

The signal machinery contains many misnomers: **SIGFPE**, **SIGILL**, and **SIGSEGV** have their roots in PDP-11 hardware terminology, but the names are too entrenched to change. The occurrence of **SIGFPE**, for instance, does not necessarily indicate a floating-point error. A conforming implementation is not required to field *any* hardware interrupts.

The C89 Committee has reserved the space of names beginning with **SIG** to permit implementations to add local names to **<signal.h>**. This implies that such names should not be otherwise used in a C source file which includes **<signal.h>**.

7.14.1 Specify signal handling

7.14.1.1 The signal function

When a signal occurs, the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. This arrangement could cause problems if the signal handler invokes a library function that was being executed at the time of the signal. Since

library functions are not guaranteed to be reentrant, they should not be called from a signal handler that returns (see §5.2.3). A specific exception to this rule was granted for calls to **signal** from within the signal handler; otherwise, the handler could not reliably reset the signal.

The specification that some signals may be effectively set to **SIG_IGN** instead of **SIG_DFL** at program startup allows programs under UNIX systems to inherit this effective setting from parent processes.

For performance reasons, UNIX does not reset **SIGILL** to default handling when the handler is called (usually to emulate missing instructions). This treatment is sanctioned by specifying that whether reset occurs for **SIGILL** is *implementation-defined*.

7.14.2 Send signal

7.14.2.1 The raise function

The **raise** function replaces /usr/group's **kill** function. The latter has an extra argument which refers to the "process ID" affected by the signal. Since the execution model of the Standard does not deal with multi-processing, the C89 Committee deemed it preferable to introduce a function which requires no process argument. The **kill** function has been standardized in the POSIX specification.

7.15 Variable arguments <stdarg.h>

For a discussion of argument passing issues, see §6.9.1.

These macros, modeled, after the UNIX **<varargs.h>** macros, have been added to enable the portable implementation in C of library functions such as **printf** and **scanf** (see §7.19.6). Such implementation could otherwise be difficult, considering newer machines that may pass arguments in machine registers rather than using the more traditional stack-oriented methods.

The definitions of these macros in the Standard differ from their forebears: they have been extended to support argument lists that have a fixed set of arguments preceding the variable list.

va_start and **va_arg** must exist as macros, since **va_start** uses an argument that is passed by name and **va_arg** uses an argument which is the name of a data type. Using **#undef** on these names leads to *undefined behavior*.

The **va_list** type is not necessarily assignable, however a function can pass a pointer to its initialized argument list object as noted below. The wording has been changed in C9X to state clearly that **va_list** is an object type.

7.15.1 Variable argument list access macros

7.15.1.1 The va_arg macro

Changing an arbitrary type name into a type name which is a pointer to that type could require sophisticated rewriting. To allow the implementation of **va_arg** as a macro, **va_arg** need only correctly handle those type names that can be transformed into the appropriate pointer type by appending a *****, which handles most simple cases. Typedefs can be defined to reduce more complicated types to a tractable form. When using these macros, it is important to remember that the type of an argument in a variable argument list will never be an integer type smaller than **int**, nor will it ever be **float** (see §6.7.5.3).

va_arg can only be used to access the value of an argument, not to obtain its address.

7.15.1.2 The va_copy macro

A new feature of C9X.

When processing variable argument lists in a function, it is occasionally useful to backtrack and examine one or more arguments a second time. In C89, the only way to do this was to start again and exactly recreate the sequence of calls to the **va_arg** macro leading up to that argument; but when these calls are controlled in a complicated manner (such as a **printf** format) this can be difficult.

A much simpler approach is to copy the **va_list** object used to represent processing of the arguments. However, there is no safe way to do this in C89 because the object may include pointers to memory allocated by the **va_start** macro and destroyed by the **va_end** macro. The new **va_copy** macro provides this safe mechanism.

Calling the **va_copy** macro exactly duplicates the state of a **va_list** object; therefore an identical call to the **va_arg** macro on the two objects will produce the same results, and both objects must be cleaned up with separate calls to the **va_end** macro.

7.15.1.3 The va_end macro

va_end must also be called from within the body of the function having the variable argument list. In many implementations, this is a do-nothing operation; but those implementations that need it probably need it badly.

7.15.1.4 The va_start macro

va_start must be called within the body of the function whose argument list is to be traversed. That function can then pass a pointer to its **va_list** object to other functions to do the actual traversal, or it can traverse the list itself.

The **parmN** argument to **va_start** was intended to be an aid to implementors writing the definition of a conforming **va_start** macro entirely in C, even using pre-C89 compilers (for example, by taking the address of the parameter). The restrictions on the declaration of the **parmN**

parameter follow from the intent to allow this kind of implementation, as applying the **&** operator to a parameter name might not produce the intended result if the parameter's declaration did not meet these restrictions.

In practice, many current implementations have "hidden machinery" that is used by the **va_start** macro to diagnose incorrect usage (for example, to verify that **parmN** actually is the name of the last fixed parameter) or to handle more complex argument passing mechanisms. Such machinery would be capable of handling any kind of parameter without restriction, but the C89 Committee saw no compelling reason to lift these restrictions, as that would require all implementations to have such machinery.

Multiple **va_list** variables can be in use simultaneously in the same function; each requires its own calls to **va_start** and **va_end**.

7.16 Boolean type and values <stdbool.h>

A new feature of C9X.

7.17 Common definitions <stddef.h>

<stddef.h> is a header invented to provide definitions of several types and macros used widely in conjunction with the library: **ptrdiff_t**, **size_t**, **wchar_t**, and **NULL**. Including any header that references one of these macros will also define it, an exception to the usual library rule that each macro or function belongs to exactly one header.

NULL can be defined as any *null pointer constant.* Thus existing code can retain definitions of **NULL** as **0** or **0L**, but an implementation may also choose to define it as **(void*)0**. This latter form of definition is convenient on architectures where **sizeof(void*)** does not equal the size of any integer type. It has never been wise to use **NULL** in place of an arbitrary pointer as a function argument, however, since pointers to different types need not be the same size. The library avoids this problem by providing special macros for the arguments to **signal**, the one library function that might see a null function pointer.

The **offsetof** macro was added to provide a portable means of determining the offset, in bytes, of a member within its structure. This capability is useful in programs, such as are typical in database implementations, which declare a large number of different data structures: it is desirable to provide "generic" routines that work from descriptions of the structures, rather than from the structure declarations themselves.[8]

[8] Consider, for instance, a set of nodes (structures) which are to be dynamically allocated and garbage-collected, and which can contain pointers to other such nodes. A possible implementation is to have the first field in each node point to a descriptor for that node. The descriptor includes a table of the offsets of fields which are pointers to other nodes. A garbage-collector "mark" routine needs no further information about the content of the node (except, of course, where to put the mark). New node types can be added to the program without requiring the mark routine to be rewritten or even recomplied.

In many implementations, **offsetof** could be defined as one of

```
(size_t)&(((s_name*)0)->m_name)
```

or

```
(size_t)(char*)&(((s_name*)0)->m_name)
```

or, where **X** is some predeclared address (or 0) and **A(Z)** is defined as **((char*)&Z)**,

```
(size_t)(A((s_name*)X->m_name ) - A(X))
```

It was not feasible, however, to mandate any single one of these forms as a construct guaranteed to be portable. Some implementations may choose to expand this macro as a call to a built-in function that interrogates the translator's symbol table.

7.18 Integer types <stdint.h>

A new feature of C9X.

<stdint.h> is a subset of **<inttypes.h>** (see §7.8) more suitable for use in freestanding environments, which might not support the formatted I/O functions. In hosted environments, if the formatted conversion support is not wanted, using this header instead of **<inttypes.h>** avoids defining such a large number of macros.

It was observed that macros for minimum and maximum limits for other integer **typedef**s in standard headers would be similarly useful, so these were added.

7.18.1 Integer types

7.18.1.5 Greatest-width integer types

Note that these can be implementation-defined types that are wider than long **long**.

7.19 Input/output <stdio.h>

(See also §MSE.9 in regards to wide streams and files.)

Many implementations of the C runtime environment, most notably the UNIX operating system, provide, aside from the standard I/O library's **fopen**, **fclose**, **fread**, **fwrite**, and **fseek**, a set of unbuffered I/O services, **open**, **close**, **read**, **write**, and **lseek**. The C89 Committee decided not to standardize the latter set of functions.

Additional semantics for these functions may be found in the POSIX standard. The standard I/O library functions use a *file pointer* for referring to the desired I/O stream. The unbuffered I/O

services use a *file descriptor* (a small integer) to refer to the desired I/O stream.

Due to weak implementations of the standard I/O library, many implementors have assumed that the standard I/O library was used for small records and that the unbuffered I/O library was used for large records. However, a good implementation of the standard I/O library can match the performance of the unbuffered services on large records. The user also has the capability of tuning the performance of the standard I/O library (with **setvbuf**) to suit the application.

Some subtle differences between the two sets of services can make the implementation of the unbuffered I/O services difficult:

- The model of a file used in the unbuffered I/O services is an array of characters. Many C environments do not support this file model.
- Difficulties arise when handling the newline character. Many hosts use conventions other than an in-stream newline character to mark the end of a line. The unbuffered I/O services assume that no translation occurs between the program's data and the file data when performing I/O, so either the newline character translation would be lost (which breaks programs) or the implementor must be aware of the newline translation (which results in non-portable programs).
- On UNIX systems, file descriptors 0, 1, and 2 correspond to the standard input, output, and error streams. This convention may be problematic for other systems in that file descriptors 0, 1, and 2 may not be available or may be reserved for another purpose; and the operating system may use a different set of services for terminal and file I/O.

In summary, the C89 Committee chose not to standardize the unbuffered I/O services because

- They duplicate the facilities provided by the standard I/O services.
- The performance of the standard I/O services can be the same or better than the unbuffered I/O services.
- The unbuffered I/O file model may not be appropriate for many C language environments.

7.19.1 Introduction

The macros **_IOFBF**, **_IOLBF**, and **_IONBF** are enumerations of the third argument to **setvbuf**, a function adopted from UNIX System V.

SEEK_CUR, **SEEK_END**, and **SEEK_SET** have been moved to **<stdio.h>** from a header specified in /usr/group and not retained in the Standard.

FOPEN_MAX and **TMP_MAX** were added as environmental limits of some interest to programs that

manipulate multiple temporary files.

FILENAME_MAX is provided so that buffers to hold file names can be conveniently declared. If the target system supports arbitrarily long filenames, the implementor should provide some reasonable value (80, 255, 509, etc.) rather than something unusable like **USHRT_MAX**.

The **fpos_t** wording has been changed in C9X to exclude array type objects. If **fpos_t** were an array, then a function would not be able to handle **fpos_t** parameters in the same manner as other **fpos_t** variables.

7.19.2 Streams

C inherited its notion of text streams from the UNIX environment in which it was born. Having each line delimited by a single *newline* character, regardless of the characteristics of the actual terminal, supported a simple model of text as a sort of arbitrary length scroll or "galley." Having a channel that is "transparent" (no file structure or reserved data encodings) eliminated the need for a distinction between text and binary streams.

Many other environments have different properties, however. If a program written in C is to produce a text file digestible by other programs, by text editors in particular, it must conform to the text formatting conventions of that environment.

The I/O facilities defined by the Standard are both more complex and more restrictive than the ancestral I/O facilities of UNIX. This is justified on pragmatic grounds: most of the differences, restrictions and omissions exist to permit C I/O implementations in environments which differ from the UNIX I/O model.

Troublesome aspects of the stream concept include:

The definition of lines. In the UNIX model, division of a file into lines is effected by newline characters. Different techniques are used by other systems: lines may be separated by CR-LF (carriage return, line feed) or by unrecorded areas on the recording medium; or each line may be prefixed by its length. The Standard addresses this diversity by specifying that newline be used as a line separator at the program level, but then permitting an implementation to transform the data read or written to conform to the conventions of the environment.

Some environments represent text lines as blank-filled fixed-length records. Thus the Standard specifies that it is *implementation-defined* whether trailing blanks are removed from a line on input. (This specification also addresses the problems of environments which represent text as variable-length records, but do not allow a record length of 0: an empty line may be written as a one-character record containing a blank, and the blank is stripped on input.)

Transparency. Some programs require access to external data without modification. For instance, transformation of CR-LF to a newline character is usually not desirable when object code is

processed. The Standard defines two stream types, *text* and *binary*, to allow a program to define, when a file is opened, whether the preservation of its exact contents or of its line structure is more important in an environment which cannot accurately reflect both.

Random access. The UNIX I/O model features random access to data in a file, indexed by character number. On systems where a newline character processed by the program represents an unknown number of physically recorded characters, this simple mechanism cannot be consistently supported for text streams. The Standard abstracts the significant properties of random access for text streams: the ability to determine the current file position and then later reposition the file to the same location. **`ftell`** returns a *file position indicator*, which has no necessary interpretation except that an **`fseek`** operation with that indicator value will position the file to the same place. Thus an implementation may encode whatever file positioning information is most appropriate for a text file, subject only to the constraint that the encoding be representable as a **`long`**. Use of **`fgetpos`** and **`fsetpos`** removes even this constraint.

Buffering. UNIX allows the program to control the extent and type of buffering for various purposes. For example, a program can provide its own large I/O buffer to improve efficiency, or can request unbuffered terminal I/O to process each input character as it is entered. Other systems do not necessarily support this generality. Some systems provide only line-at-a-time access to terminal input; some systems support program-allocated buffers only by copying data to and from system-allocated buffers for processing. Buffering is addressed in the Standard by specifying UNIX-like **`setbuf`** and **`setvbuf`** functions, but permitting great latitude in their implementation. A conforming library need neither attempt the impossible nor respond to a program attempt to improve efficiency by introducing additional overhead.

Thus, the Standard imposes a clear distinction between *text streams*, which must be mapped to suit local custom, and *binary streams*, for which no mapping takes place. Local custom on UNIX and related systems is of course to treat the two sorts of streams identically, and nothing in the Standard requires any change to this practice.

Even the specification of binary streams requires some changes to accommodate a wide range of systems. Because many systems do not keep track of the length of a file to the nearest byte, an arbitrary number of characters may appear on the end of a binary stream directed to a file. The Standard cannot forbid this implementation, but does require that this padding consist only of null characters. The alternative would be to restrict C to producing binary files digestible only by other C programs; this alternative runs counter to the spirit of C.

The set of characters required to be preserved in text stream I/O are those needed for writing C programs; the intent is that the Standard should permit a C translator to be written in a maximally portable fashion. Control characters such as backspace are not required for this purpose, so their handling in text streams is not mandated.

It was agreed that some minimum maximum line length must be mandated, and 254 was chosen for C89. C9X increases this limit to 4095.

7.19.3 Files

The *as if* principle is once again invoked to define the nature of input and output in terms of just two functions, **fgetc** and **fputc**. The actual primitives in a given system may be quite different.

The distinction between buffered and unbuffered streams suggests the desired interactive behavior; but an implementation may still be conforming even if delays in a network or terminal controller prevent output from appearing in time. It is the *intent* that matters here.

No constraints are imposed upon file names except that they must be representable as strings with no embedded null characters.

7.19.4 Operations on files

7.19.4.1 The remove function

/usr/group provides the **unlink** system call to remove files. The UNIX-specific definition of this function prompted the C89 Committee to replace it with a portable function.

7.19.4.2 The rename function

This function was added to provide a system-independent atomic operation to change the name of an existing file; /usr/group only provided the link system call, which gives the file a new name without removing the old one, and which is extremely system-dependent.

The C89 Committee considered a proposal that **rename** should quietly copy a file if simple renaming couldn't be performed in some context, but rejected this as potentially too expensive at execution time.

rename is meant to give access to an underlying facility of the execution environment's operating system. When the new name is the name of an existing file, some systems allow the renaming and delete the old file or make it inaccessible by that name, while others prohibit the operation. The effect of **rename** is thus *implementation-defined*.

7.19.4.3 The tmpfile function

The **tmpfile** function is intended to allow users to create binary "scratch" files. The *as if* principle implies that the information in such a file need never actually be stored on a file-structured device.

The temporary file is created in binary update mode because it will presumably be first written and then read as transparently as possible. Trailing null-character padding may cause problems for some existing programs.

7.19.4.4 The `tmpnam` function

This function allows for more control than **`tmpfile`**: a file can be opened in binary mode or text mode, and files are not erased at completion.

There is always some time between the call to **`tmpnam`** and the use in **`fopen`** of the returned name. Hence it is conceivable that in some implementations the name, which named no file at the call to **`tmpnam`**, has been used as a filename by the time of the call to **`fopen`**. Implementations should devise name generation strategies which minimize this possibility, but users should allow for it.

A new feature of C9X: the C9X committee recognized that the C89 specification had a serious flaw: if **`tmpnam`** were called fewer than **`TMP_MAX`** times but was unable to generate a suitable string because every potential string named an existing file, there was no way to report failure and no undefined behavior, so there was no option other than to never return. C9X resolved the problem by allowing **`tmpnam`** to return a null pointer when it cannot generate a suitable string and by specifying that **`TMP_MAX`** is the number of potential strings, any or all of which may name existing files and thus not be suitable return values.

QUIET CHANGE IN C9X

> Code that calls **`tmpnam`** and doesn't check for a null return value may produce undefined behavior.

7.19.5 File access functions

7.19.5.1 The `fclose` function

On some operating systems, it is difficult or impossible to create a file unless something is written to the file. A maximally portable program which relies on a file being created must write something to the associated stream before closing it.

7.19.5.2 The `fflush` function

The **`fflush`** function ensures that output has been forced out of internal I/O buffers for a specified stream. Occasionally, however, it is necessary to ensure that *all* output is forced out, and the programmer may not conveniently be able to specify all the currently open streams, perhaps because some streams are manipulated within library packages.[9] To provide an implementation-independent method of flushing all output buffers, the Standard specifies that this is the result of calling **`fflush`** with a **`NULL`** argument.

7.19.5.3 The `fopen` function

[9] For instance, on a system (such as UNIX) which supports process forks, it is usually necessary to flush all output buffers just prior to the fork.

The **b** type modifier was added to deal with the text/binary dichotomy (see §7.19.2). Because of the limited ability to seek within text files (see §7.19.9.1), an implementation is at liberty to treat the old update **+** modes *as if* **b** were also specified.

Table 7.1 tabulates the capabilities and actions associated with the various specified mode string arguments to **fopen**.

Table 7.1: File and stream properties of **fopen** modes

	r	w	a	r+	w+	a+
file must exist before open	✔			✔		
old file contents discarded on open		✔			✔	
stream can be read	✔			✔	✔	✔
stream can be written		✔	✔	✔	✔	✔
stream can be written only at end			✔			✔

Other specifications for files, such as record length and block size, are not specified in the Standard due to their widely varying characteristics in different operating environments. Changes to file access modes and buffer sizes may be specified using the **setvbuf** function (see §7.19.5.6). An implementation may choose to allow additional file specifications as part of the mode string argument. For instance,

```
file1 = fopen(file1name, "wb,reclen=80");
```

might be a reasonable extension on a system that provides record-oriented binary files and allows a programmer to specify record length.

A change of input/output direction on an update file is only allowed following a successful **fsetpos**, **fseek**, **rewind**, or **fflush** operation, since these are precisely the functions which assure that the I/O buffer has been flushed.

§7.19.2 imposes the requirement that binary files not be truncated when they are updated. This rule does not preclude an implementation from supporting additional file types that do truncate when written to, even when they are opened with the same sort of **fopen** call. Magnetic tape files are an example of a file type that must be handled this way. (On most tape hardware it is impossible to write to a tape without destroying immediately following data.) Hence tape files are not "binary files" within the meaning of the Standard. A conforming hosted implementation must provide and document at least one file type (on disk, most likely) that behaves exactly as specified in the Standard.

7.19.5.5 The setbuf function

setbuf is subsumed by **setvbuf**; but it has been retained for compatibility with old code.

7.19.5.6 The setvbuf function

setvbuf was adopted from UNIX System V, both to control the nature of stream buffering and to specify the size of I/O buffers. An implementation is not required to make actual use of a buffer

provided for a stream, so a program must never expect the buffer's contents to reflect I/O operations. Furthermore, the Standard does not require that the requested buffering be implemented; it merely mandates a standard mechanism for requesting whatever buffering services might be provided.

Although three types of buffering are defined, an implementation may choose to make one or more of them equivalent. For example, a library may choose to implement line buffering for binary files as equivalent to unbuffered I/O, or it may choose to always implement full buffering as equivalent to line buffering.

The general principle is to provide portable code with a means of requesting the most appropriate popular buffering style, but not to require an implementation to support these styles.

A new feature of C9X: C90 was not clear about what, if anything, the **`size`** argument means when **`buf`** is a null pointer. Existing practice is mixed: some implementations ignore it completely, other implementations use it as guidance for determining the size of the buffer allocated by **`setvbuf`**. C9X gives warning that **`size`** might not be ignored in this case, so portable programs must be sure to supply a reasonable value.

7.19.6 Formatted input/output functions

7.19.6.1 The `fprintf` function

The **`%hh`**, **`%ll`** and **`%lf`** length modifiers were added in C9X (see §7.19.6.2).

Use of the **`L`** modifier with floating conversions was added in C89 to deal with formatted output of the **`long double`** type.

Note that the **`%X`** and **`%x`** formats expect a corresponding **`unsigned int`** argument, and **`%lX`** and **`%lx`** must be supplied with an **`unsigned long`** argument.

The **`%i`** conversion specifier was added in C89 for programmer convenience to provide symmetry with **`fscanf`**'s **`%i`** conversion specifier, even though it has exactly the same meaning as the **`%d`** conversion specifier when used with **`fprintf`**.

The **`%p`** conversion specifier was added to C89 for pointer conversion since the size of a pointer is not necessarily the same as the size of any integer type. Because an implementation may support more than one size of pointer, the corresponding argument is expected to be a pointer to **`void`**.

The **`%n`** format was added to C89 to permit ascertaining the number of characters converted up to that point in the current invocation of the formatter.

Some pre-C89 implementations switch formats for **`%g`** at an exponent of –3 instead of the Standard's –4: existing code which requires that the format switch at –3 will have to be changed.

Some existing implementations provide **%D** and **%O** as synonyms or replacements for **%ld** and **%lo**. The C89 Committee considered the latter notation preferable.

The C89 Committee reserved lower case conversion specifiers for future standardization.

The use of leading zero in field widths to specify zero padding is superseded by a precision field. The older mechanism was retained.

Some implementations have provided the **%r** format as a means of indirectly passing a variable-length argument list. The functions **vfprintf**, etc., are considered to be a more controlled method of effecting this indirection, so **%r** was not adopted in the Standard (see §7.19.6.8).

New features of C9X.

The C89 translation limit of 509 characters produced from a single conversion specifier was increased to 4095 in C9X (approximately an eight-fold increase) to reflect the increase in the minimum amount of memory available in the C9X target machine (see §5.2.4.1).

The printing formats for numbers are not entirely specified. The requirements of the Standard are loose enough to allow implementations to handle such cases as signed zero, NaN, and infinity in an appropriate fashion. These have been improved in C9X.

Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble boundaries. For example, the next value greater than one in the common IEC 60559 80-bit extended format could be

```
0x8.000000000000001p-3
```

The next value less than one in IEC 60559 double could be

```
0x1.fffffffffffp-1
```

Note that if the precision is missing, trailing zeros may be omitted. For example, the value positive zero might be

```
0x0.p+0
```

The more suggestive conversion specifiers for hexadecimal formatting, namely **x** and **h**, were unavailable; and since **h** was taken, **H** was ruled out in favor of a lower/upper case option. Possibilities other than **a** included: **b j k m q r t v w y z**. The optional **h** to indicate hexadecimal floating, as in **%he**, was deemed a less natural fit with the established scheme for specifiers and options.

QUIET CHANGE IN C95

Use of the **A** and **F** format specifiers constitutes a minor extension to C89 which does

not reserve them.

For binary-to-decimal conversion, the infinitely precise result is just the source value, and the destination format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

7.19.6.2 The `fscanf` function

The specification of **`fscanf`** is based in part on these principles:

- As soon as one specified conversion fails, the whole function invocation fails.
- One-character pushback is sufficient for the implementation of **`fscanf`**. Given the invalid field **`"-.x"`**, the characters **`"-."`** are not pushed back.
- If a "flawed field" is detected, no value is stored for the corresponding argument.
- The conversions performed by **`fscanf`** are compatible with those performed by **`strtod`** and **`strtol`**.

Input pointer conversion with **`%p`** was added to C89, although it is obviously risky, for symmetry with **`fprintf`**. The **`%i`** format was added to permit the scanner to determine the radix of the number in the input stream; the **`%n`** format was added to make available the number of characters scanned thus far in the current invocation of the scanner. C9X adds **`%a`** and **`%A`**.

White space is defined by the **`isspace`** function (see §7.4.1.9).

An implementation must not use the **`ungetc`** function to perform the necessary one-character pushback. In particular, since the unmatched text is left "unread," the file position indicator as reported by the **`ftell`** function must be the position of the character remaining to be read. Furthermore, if the unread characters were themselves pushed back via **`ungetc`**, the pushback in **`fscanf`** could not affect the pushback stack in **`ungetc`**. A **`scanf`** call that matches N characters from a stream must leave the stream in the same state *as if* N consecutive **`getc`** calls had been made.

A new feature of C9X: The **`hh`** and **`ll`** length modifiers were added in C9X. **`ll`** supports the new **`long long int`** type. **`hh`** adds the ability to treat character types the same as all other integer types; this can be useful in implementing macros such as **`SCNd8`** in **`<inttypes.h>`** (see 7.18).

7.19.6.3 The `printf` function

See comments in §7.19.6.1 above.

7.19.6.4 The `scanf` function

See comments in §7.19.6.2 above.

7.19.6.5 The `snprintf` function

A new feature of C9X: The **`sprintf`** function is very useful, but can overrun the output buffer; and that has been exploited in attacks on computer and network security. C9X addresses this problem by adding the **`snprintf`** function, modeled after the 4.4BSD version, which performs bounds checking on the output array.

7.19.6.6 The `sprintf` function

See §7.19.6.1 for comments on output formatting.

In the interests of minimizing redundancy, **`sprintf`** has subsumed the older, rather uncommon, **`ecvt`**, **`fcvt`**, and **`gcvt`**.

7.19.6.7 The `sscanf` function

The behavior of **`sscanf`** on encountering end of string has been clarified. See also comments in §7.19.6.2 above.

7.19.6.8 The `vfprintf` function

The functions **`vfprintf`**, **`vprintf`**, and **`vsprintf`** were adopted from UNIX System V to facilitate writing special purpose formatted output functions.

7.19.6.9 The `vfscanf` function

The functions **`vfscanf`**, **`vscanf`**, and **`vsscanf`** were adopted analogously to the **`vfprintf`** functions to facilitate writing special-purpose formatted intput functions.

7.19.6.10 The `vprintf` function

See §7.19.6.8.

7.19.6.11 The `vscanf` function

See §7.19.6.9.

7.19.6.12 The `vsnprintf` function

See §7.19.6.5.

7.19.6.13 The `vsprintf` function

See §7.19.6.8.

7.19.6.14 The `vsscanf` function

See §7.19.6.9.

7.19.7 Character input/output functions

7.19.7.1 The `fgetc` function

Because much existing code assumes that **`fgetc`** and **`fputc`** are the actual functions equivalent to the macros **`getc`** and **`putc`**, the Standard requires that they not be implemented as macros.

7.19.7.2 The `fgets` function

This function subsumes **`gets`** which has no limit to prevent storage overwrite on arbitrary input (see §7.19.7.7).

7.19.7.3 The `fputc` function

See §7.19.7.1.

7.19.7.5 The `getc` function

`getc` and **`putc`** have often been implemented as unsafe macros, since it is difficult in such a macro to touch the stream argument only once. Since this danger is common in prior art, these two functions are explicitly permitted to evaluate **`stream`** more than once.

7.19.7.7 The `gets` function

Because **`gets`** does not check for buffer overrun, it is generally unsafe to use when its input is not under the programmer's control. This has caused some to question whether it should appear in the Standard at all. The Committee decided that **`gets`** was useful and convenient in those special circumstances when the programmer does have adequate control over the input, and as longstanding existing practice, it needed a standard specification. In general, however, the preferred function is **`fgets`** (see §7.19.7.2).

7.19.7.8 The `putc` function

See §7.19.7.5.

7.19.7.10 The `puts` function

`puts(s)` is not exactly equivalent to **`fputs(stdout,s)`**; and **`puts`** also writes a newline after

the argument string. This incompatibility reflects existing practice.

7.19.7.11 The `ungetc` function

/usr/group requires that at least one character be read before **`ungetc`** is called in certain implementation-specific cases. The C89 Committee removed this requirement, thus obliging a **`FILE`** structure to have room to store one character of pushback regardless of the state of the buffer. It felt that this degree of generality makes clearer the ways in which the function may be used. The C9X Committee decided to deprecate the use of **`ungetc`** on a binary file at the beginning of the file because of the impossibility of distinguishing between successful and error returns from the **`ftell`** function, both of which would be **`-1L`**.

It is permissible to push back a different character than that which was read, which accords with common existing practice. The last-in, first-out nature of **`ungetc`** has been clarified.

`ungetc` is typically used to handle algorithms such as tokenization which involve one-character lookahead in text files. **`fseek`** and **`ftell`** are used for random access, typically in binary files. So that these disparate file-handling disciplines are not unnecessarily linked, the value of a text file's file position indicator immediately after **`ungetc`** has been specified as indeterminate.

Existing practice relies on two different models of the effect of **`ungetc`**. One model can be characterized as writing the pushed-back character "on top of" the previous character. This model implies an implementation in which the pushed-back characters are stored within the file buffer and bookkeeping is performed by setting the file position indicator to the previous character position. (Care must be taken in this model to recover the overwritten character values when the pushed-back characters are discarded as a result of other operations on the stream.) The other model can be characterized as pushing the character "between" the current character and the previous character. This implies an implementation in which the pushed-back characters are specially buffered (within the **`FILE`** structure, say) and accounted for by a flag or count. In this model it is natural *not* to move the file position indicator. The indeterminacy of the file position indicator while pushed-back characters exist accommodates both models.

Mandating either model by specifying the effect of **`ungetc`** on a text file's file position indicator creates problems with implementations that have assumed the other model. Requiring the file position indicator not to change after **`ungetc`** would necessitate changes in programs which combine random access and tokenization on text files, and rely on the file position indicator marking the end of a token even after pushback. Requiring the file position indicator to back up would create severe implementation problems in certain environments, since in some file organizations it can be impossible to find the previous input character position without having read the file sequentially to the point in question.[10]

[10] Consider, for instance, a sequential file of variable-length records in which a line is represented as a count field followed by the characters in the line. The file position indicator must encode a character position as the position of the count field plus an offset into the line; from the position of the count field and the length of the line, the next count field can be found. Insufficient information is available for finding the *previous* count field, so backing up from the first character of a line necessitates, in the general case, a sequential read from the start of the file.

7.19.8 Direct input/output functions

7.19.8.1 The `fread` function

`size_t` is the appropriate type both for an object size and for an array bound (see §6.5.3.4), so this is the type of both `size` and `nelem`.

7.19.8.2 The `fwrite` function

See §7.19.8.1.

7.19.9 File positioning functions

7.19.9.1 The `fgetpos` function

`fgetpos` and `fsetpos` were added to C89 to allow random access operations on files that are too large to handle with `fseek` and `ftell`.

7.19.9.2 The `fseek` function

Whereas a binary file can be treated as an ordered sequence of bytes counting from zero, a text file need not map one-to-one to its internal representation (see §7.19.2). Thus, only seeks to an earlier reported position are permitted for text files. The need to encode both record position and position within a record in a `long` value may constrain the size of text files upon which `fseek` and `ftell` can be used to be considerably smaller than the size of binary files.

Given these restrictions, the Committee still felt that this function has enough utility, and is used in sufficient existing code, to warrant its retention in the Standard. `fgetpos` and `fsetpos` were added to deal with files that are too large to handle with `fseek` and `ftell`.

The `fseek` function will reset the end-of-file flag for the stream; the error flag is not changed unless an error occurs, in which case it will be set.

7.19.9.4 The `ftell` function

`ftell` can fail for at least two reasons:

- the stream is associated with a terminal or some other file type for which *file position indicator* is meaningless.
- the file may be positioned at a location not representable in a `long`.

Thus a method for `ftell` to report failure was specified (see also §7.19.9.1).

7.19.9.5 The `rewind` function

Resetting the end-of-file and error indicators was added to the specification of **rewind** to make the specification more logically consistent.

7.19.10 Error-handling functions

7.19.10.4 The perror function

At various times, the C89 Committee considered providing a form of **perror** that delivers up an error string version of **errno** without performing any output. It ultimately decided to provide this capability in a separate function, **strerror** (see §7.21.6.2).

7.20 General Utilities <stdlib.h>

The header **<stdlib.h>** was invented by the C89 Committee to hold an assortment of functions that were otherwise homeless.

7.20.1 Numeric conversion functions

7.20.1.1 The atof function

atof, **atoi**, and **atol** are subsumed by **strtod** and **strtol**, but were retained because they are used extensively in existing code. They are less reliable, but may be faster if the argument is known to be in a valid range.

This specification does not require **float** and **long double** versions of **atof**, but instead encourages the use of **strtof** and **strtold** which have a more generally useful interface.

7.20.1.2 The atoi, atol, and atoll functions

See §7.20.1.1.

7.20.1.3 The strtod, strtof, and strtold functions

strtod was adopted for C89 from UNIX System V because it offers more control over the conversion process, and because it is required not to produce unexpected results on overflow during conversion. **strtol** (§7.20.1.4) was adopted for the same reason. C9X adds **strtof** and **strtold**.

So much regarding NaN significands is unspecified because so little is portable. Attaching meaning to NaN significands is problematic, even for one implementation, even an IEC 60559 one. For example, the IEC 60559 floating-point standard does not specify the effect of format conversions on NaN significands. Conversions, perhaps generated by the compiler, may alter NaN significands in obscure ways.

Requiring a sign for NaN or infinity input was considered as a way of minimizing the chance of mistakenly accepting nonnumeric input. The need for this was deemed insufficient, partly on the basis of prior art.

QUIET CHANGE IN C9X

For simplicity, the infinity and NaN representations are provided through straightforward enhancements to C89 rather than through a new locale. Note also that standard C locale categories do not affect the representations of infinities and NaNs.

A previous specification that **`strtod`** return a NaN for invalid numeric input as recommended by IEEE 854 was withdrawn because of the incompatibility with C89, which demands that **`strtod`** return zero for invalid numeric input.

7.20.1.4 The `strtol`, `strtoll`, `strtoul`, and `strtoull` functions

`strtol` was adopted for C89 as was **`strtod`** (§7.20.1.3); C9X adds **`strtoll`** and **`strtoull`**.

`strtoul` was introduced by the C89 Committee to provide a facility like **`strtol`** for **`unsigned long`** values. Simply using **`strtol`** in such cases could result in overflow upon conversion.

7.20.2 Pseudo-random sequence generation functions

7.20.2.1 The `rand` function

The C89 Committee decided that an implementation should be allowed to provide a **`rand`** function which generates the best random sequence possible in that implementation, and therefore mandated no standard algorithm. It recognized the value, however, of being able to generate the same pseudo-random sequence in different implementations, and so it published as an example in the Standard an algorithm that generates the same pseudo-random sequence in any conforming implementation, given the same seed.

The **`rand`** and **`srand`** functions were based on existing practice; indeed the example implementation was actually used in some versions of UNIX. Pseudo-random numbers have many uses; and it should be noted that the example generator, while adequate for casual purposes, is insufficiently random for demanding applications such as Monte-Carlo sampling and cryptography. Also, only 32,768 distinct values are returned, which may be insufficiently fine resolution for some purposes. Implementations may substitute improved algorithms and wider ranges of values; it is incumbent on the programmer to ensure that the particular generator has appropriate statistical properties for the intended application.

7.20.2.2 The `srand` function

See §7.20.2.1.

7.20.3 Memory management functions

The treatment of null pointers and zero-length allocation requests in the definition of these functions was in part guided by a desire to support this paradigm:

```
OBJ * p; // pointer to a variable list of OBJs

/* initial allocation */
p = (OBJ *) calloc(0, sizeof(OBJ));
/* ... */

/* reallocations until size settles */
while(1) {
    p = (OBJ *) realloc((void *)p, c * sizeof(OBJ));
    /* change value of c or break out of loop */
}
```

This coding style, not necessarily endorsed by the Committee, is reported to be in widespread use.

Some implementations have returned non-null values for allocation requests of zero bytes. Although this strategy has the theoretical advantage of distinguishing between "nothing" and "zero" (an unallocated pointer vs. a pointer to zero-length space), it has the more compelling theoretical disadvantage of requiring the concept of a zero-length object. Since such objects cannot be declared, the only way they could come into existence would be through such allocation requests.

The C89 Committee decided not to accept the idea of zero-length objects. The allocation functions may therefore return a null pointer for an allocation request of zero bytes. Note that this treatment does not preclude the paradigm outlined above.

QUIET CHANGE IN C89

A program which relies on size-zero allocation requests returning a non-null pointer will behave differently.

Some implementations provide a function, often called **alloca**, which allocates the requested object from automatic storage; and the object is automatically freed when the calling function exits. Such a function is not efficiently implementable in a variety of environments, so it was not adopted in the Standard.

7.20.3.1 The calloc function

Both **nelem** and **elsize** must be of type **size_t** for reasons similar to those for **fread** (see §7.19.8.1).

If a scalar with all bits zero is not interpreted as a zero value by an implementation, then **calloc**

may have astonishing results in existing programs transported there.

7.20.3.2 The `free` function

The Standard makes clear that a program may only free that which has been allocated, that an allocation may only be freed once, and that a region may not be accessed once it is freed. Some implementations allow more dangerous license. The null pointer is specified as a valid argument to this function to reduce the need for special-case coding.

7.20.3.4 The `realloc` function

A null first argument is permissible. If the first argument is not null, and the second argument is 0, then the call frees the memory pointed to by the first argument, and a null argument may be returned; this specification is consistent with the policy of not allowing zero-sized objects.

A new feature of C9X: the **`realloc`** function was changed to make it clear that the pointed-to object is deallocated, a new object is allocated, and the content of the new object is the same as that of the old object up to the lesser of the two sizes. C89 attempted to specify that the new object was the same object as the old object but might have a different address. This conflicts with other parts of the Standard that assume that the address of an object is constant during its lifetime. Also, implementations that support an actual allocation when the size is zero do not necessarily return a null pointer for this case. C89 appeared to require a null return value, and the Committee felt that this was too restrictive.

7.20.4 Communication with the environment

7.20.4.1 The `abort` function

The C89 Committee vacillated over whether a call to **`abort`** should return if the **`SIGABRT`** signal is caught or ignored. To minimize astonishment, the final decision was that **`abort`** never returns.

7.20.4.2 The `atexit` function

`atexit` provides a program with a convenient way to clean up the environment before it exits. It was adapted from the Whitesmiths C run-time library function **`onexit`**.

A suggested alternative was to use the **`SIGTERM`** facility of the **`signal`**/**`raise`** machinery, but that would not give the last-in-first-out stacking of multiple functions so useful with **`atexit`**.

It is the responsibility of the library to maintain the chain of registered functions so that they are invoked in the correct sequence upon program exit.

7.20.4.3 The `exit` function

The argument to **`exit`** is a status indication returned to the invoking environment. In the UNIX operating system, a value of zero is the successful return code from a program. As usage of C has

spread beyond UNIX, **exit(0)** has often been retained as an idiom indicating successful termination, even on operating systems with different systems of return codes. This usage is thus recognized as standard. There has never been a portable way of indicating a non-successful termination, since the arguments to exit are implementation-defined. The **EXIT_FAILURE** macro was added to C89 to provide such a capability. **EXIT_SUCCESS** was added as well.

Aside from calls explicitly coded by a programmer, **exit** is invoked on return from **main**. Thus in at least this case, the body of **exit** cannot assume the existence of any objects with automatic storage duration except those declared in **exit**.

The Committee considered the addition of **_exit**, but rejected it based on concerns of incompatible with the POSIX specification upon which it is based. For example, one concern expressed is that **_exit** was specified as a way to get out of a signal handler without triggering another signal, but that is not actually the way **_exit** behaves in POSIX environments. The Committee did not wish to give programmers this kind of false hope. (But see §7.20.4.4 for C9X.)

7.20.4.4 The _Exit function

A new feature of C9X: the C9X Committee considered it desirable to have an **_exit**-like function that would result in immediate program termination without triggering signals or **atexit**-registered functions, but chose the name, **_Exit**, rather than **_exit**, because of the potential conflict with existing practice mentioned above.

7.20.4.5 The getenv function

The definition of **getenv** is designed to accommodate both implementations that have all in-memory read-only environment strings and those that may have to read an environment string into a static buffer. Hence the pointer returned by the **getenv** function points to a string not modifiable by the caller. If an attempt is made to change this string, the behavior of future calls to **getenv** are undefined.

A corresponding **putenv** function was omitted from the Standard, since its utility outside a multi-process environment is questionable, and since its definition is properly the domain of an operating system standard.

7.20.4.6 The system function

The **system** function allows a program to suspend its execution temporarily in order to run another program to completion.

Information may be passed to the called program in three ways: through command-line argument strings, through the environment, and (most portably) through data files. Before calling the **system** function, the calling program should close all such data files.

Information may be returned from the called program in two ways: through the implementation-

defined return value (In many implementations, the termination status code which is the argument to the **exit** function is returned by the implementation to the caller as the value returned by the **system** function.), and (most portably) through data files.

If the environment is interactive, information may also be exchanged with users of interactive devices. Some implementations offer built-in programs called "commands" (for example, "date") which may provide useful information to an application program via the **system** function. The Standard does not attempt to characterize such commands, and their use is not portable.

On the other hand, the use of the **system** function *is* portable, provided the implementation supports the capability. The Standard permits the application to ascertain this by calling the **system** function with a null pointer argument. Whether more levels of nesting are supported can also be ascertained this way; but assuming more than one such level is obviously dangerous.

7.20.5 Searching and sorting utilities

C9X clarifies requirements and usage of the comparison functions.

7.20.6 Integer arithmetic functions

7.20.6.1 The abs, labs, and llabs functions

abs was moved from **<math.h>** as it was the only function in that library which did not involve **double** arithmetic. Some programs have included **<math.h>** solely to gain access to **abs**, but in some implementations this results in unused floating-point run-time routines becoming part of the translated program.

The C89 Committee rejected proposals to add an absolute value operator to the language. An implementation can provide a built-in function for efficiency.

7.20.6.2 The div, ldiv, and lldiv functions

Because C89 had implementation-defined semantics for division of signed integers when negative operands were involved, **div** and **ldiv**, and **lldiv** in C9X, were invented to provide well-specified semantics for signed integer division and remainder operations. The semantics were adopted to be the same as in Fortran. Since these functions return both the quotient and the remainder, they also serve as a convenient way of efficiently modeling underlying hardware that computes both results as part of the same operation. Table 7.2 summarizes the semantics of these functions.

Table 7.2: Results of **div**, **ldiv** and **lldiv**

numer	denom	quot	rem
7	3	2	1

-7	3	-2	-1
7	-3	-2	1
-7	-3	2	-1

Division by zero is described as *undefined behavior* rather than as setting **errno** to **EDOM**. The program can just as easily check for a zero divisor before a division as for an error code afterwards, and the adopted scheme reduces the burden on the function.

Now that C9X requires similar semantics for the division operator, the main reason for new programs to use **div**, **ldiv** or **lldiv** is to simultaneously obtain quotient and remainder.

7.20.7 Multibyte/wide character conversion functions

See §5.2.1.2, §MSE.8 and §MSE.9.1 for an overall discussion of multibyte character representations and wide characters.

Implementation note: although these functions are described as having a conversion state for consistency with the restartable equivalents in §7.24.6.3 and §7.24.6.4, the only piece of the conversion state that is required for these functions is the current shift state, if any.

7.20.8 Multibyte/wide string conversion functions

See §7.20.7.

7.21 String handling <string.h>

The C89 Committee felt that the functions in this subclause were all excellent candidates for replacement by high-performance built-in operations. Hence many simple functions have been retained, and several added, just to leave the door open for better implementations of these common operations.

The Standard reserves function names beginning with **str** or **mem** for possible future use.

7.21.1 String function conventions

memcpy, **memset**, **memcmp**, and **memchr** were adopted in C89 from several existing implementations. The general goal was to provide equivalent capabilities for three types of byte sequences:

- null-terminated strings (**str**-).
- null-terminated strings with a maximum length (**strn**-).

- transparent data of specified length (**mem**-).

7.21.2 Copying functions

A block copy routine should be “right”: it should work correctly even if the blocks being copied overlap. Otherwise it is more difficult to correctly code such overlapping copy operations, and portability suffers because the optimal C-coded algorithm on one machine may be horribly slow on another.

A block copy routine should be “fast”: it should be implementable as a few inline instructions which take maximum advantage of any block copy provisions of the hardware. Checking for overlapping copies produces too much code for convenient inlining in many implementations. The programmer knows in a great many cases that the two blocks cannot possibly overlap, so the space and time overhead are for naught.

These arguments are contradictory but each is compelling. Therefore the Standard mandates two block copy functions: **memmove** is required to work correctly even if the source and destination overlap, while **memcpy** can assume non-overlapping operands and be optimized accordingly.

7.21.2.4 The `strncpy` function

strncpy was initially introduced into the C library to deal with fixed-length name fields in structures such as directory entries. Such fields are not used in the same way as strings: the trailing null is unnecessary for a maximum-length field, and setting trailing bytes for shorter names to null assures efficient field-wise comparisons. **strncpy** is not by origin a “bounded strcpy,” and the Committee preferred to recognize existing practice rather than alter the function to better suit it to such use.

7.21.3 Concatenation functions

7.21.3.2 The `strncat` function

Note that this function may add **n+1** characters to the string.

7.21.4 Comparison functions

7.21.4.1 The `memcmp` function

See §7.21.1.

7.21.4.3 The `strcoll` function

strcoll and **strxfrm** provide for *locale-specific* string sorting. **strcoll** is intended for applications in which the number of comparisons is small; **strxfrm** is more appropriate when

items are to be compared a number of times and the cost of transformation is paid only once.

7.21.4.5 The `strxfrm` function

See §7.21.4.3.

7.21.5 Search functions

7.21.5.1 The `memchr` function

See §7.21.1.

7.21.5.7 The `strstr` function

The **`strstr`** function is an invention of the C89 Committee. It is included as a hook for efficient substring algorithms, or for built-in substring instructions.

7.21.5.8 The `strtok` function

This function was included in C89 to provide a convenient solution to many simple problems of lexical analysis, such as scanning command line arguments.

The **`strsep`** function was proposed as an enhanced replacement for the **`strtok`** function. While this is a common extension, it is easy enough for a user to provide this functionality, and it is unclear that an implementor can do a substantially better job; so, there was not sufficient support for adding this feature.

7.21.6 Miscellaneous functions

7.21.6.1 The `memset` function

See §7.21.1 and §7.20.3.1.

7.21.6.2 The `strerror` function

This function is a descendant of **`perror`** (see §7.19.10.4). It is defined such that it can return a pointer to an in-memory read-only string, or can copy a string into a static buffer on each call.

7.21.6.3 The `strlen` function

This function is now specified as returning a value of type **`size_t`** (see §6.5.3.4).

7.22 Type-generic math `<tgmath.h>`

A new feature of C9X.

Type-generic macros allow calling a function whose type is determined by the argument type, as is the case for C operators such as **+** and *****. For example, with a type-generic **cos** macro, the expression **cos((float)x)** will have type **float**. This feature enables writing more portably efficient code and alleviates need for awkward casting and suffixing in the process of porting or adjusting precision. Generic math functions are a widely appreciated feature of Fortran.

The only arguments that affect the type resolution are the arguments corresponding to the parameters that have type **double** in the synopsis. Hence the type of a type-generic call to **nexttoward**, whose second parameter is **long double** in the synopsis, is determined solely by the type of the first argument.

The term type-generic was chosen over the proposed alternatives of intrinsic and overloading. The term is more specific than intrinsic, which already is widely used with a more general meaning, and reflects a closer match to Fortran's generic functions than to C++ overloading.

The macros are placed in their own header in order not to silently break old programs that include **<math.h>**, for example with **printf("%e", sin(x))**.

modf(double, double *) is excluded because no way was seen to make it safe without complicating the type resolution.

This specification differs from an earlier proposal in that the type is determined solely by the argument, and may be narrower than the type for expression evaluation. This change was made because the performance costs for computing functions with narrow arguments to wide range and precision might be too high, even if the implementation efficiently evaluates basic operations to wider format.

Also, this differs from earlier proposals in that integer-type arguments are converted to **double** instead of **float**. Although converting to **float** would have been more consistent with the usual arithmetic conversions, converting to **double** has the advantages of preserving the value more often on many systems, and of being more compatible with C89 where unsuffixed calls to math functions with integer arguments were calls to **double** functions.

Having a **g** suffix for the generic macros was considered but thought unnecessary.

The implementation might, as an extension, endow appropriate ones of the macros that this standard specifies only for real arguments with the ability to invoke the complex functions.

This specification does not prescribe any particular implementation mechanism for generic macros. It could be implemented simply with built-in macros. The generic macro for **sqrt**, for example, could be implemented with

```
#undef sqrt
#define sqrt(x) __BUILTIN_GENERIC_sqrt(x)
```

Generic macros are designed for a useful level of consistency with C++ overloaded math functions.

The great majority of existing C programs are expected to be unaffected when **<tgmath.h>** is included instead of **<math.h>** or **<complex.h>**. Generic macros are similar to the C89 library masking macros, though the semantic types of return values differ.

The ability to overload on integer as well as floating types would have been useful for some functions, for example **copysign**. Overloading with different numbers of arguments would have allowed reusing names, for example **remainder** for **remquo**. However, these facilities would have complicated the specification; and their natural consistent use, such as for a floating **abs** or a two-argument **atan**, would have introduced further inconsistencies with C89 for insufficient benefit.

This specification in no way limits the implementation's options for efficiency, including inlining library functions.

7.23 Date and time <time.h>

7.23.1 Components of time

The types **clock_t** and **time_t** are arithmetic because values of these types must, in accordance with existing practice, on occasion be compared with –1 (a "don't-know" indication), suitably cast. No arithmetic properties of these types are defined by the Standard, however, in order to allow implementations the maximum flexibility in choosing ranges, precisions, and representations most appropriate to their intended application. The representation need not be a count of some basic unit; an implementation might conceivably represent different components of a temporal value as subfields of an integer type.

Many C environments do not support /usr/group library concepts of daylight saving time (DST, also called summer time) or time zones. Both notions are defined geographically and politically, and thus may require more knowledge about the real world than an implementation can support. Hence the Standard specifies the date and time functions such that information about DST and time zones is not required. /usr/group's **tzset** function, which would require dealing with time zones, was excluded altogether. An implementation reports that information about DST is not available by setting the **tm_isdst** field in a broken-down time to a negative value. An implementation may return a null pointer from a call to **gmtime** if information about the offset between Coordinated Universal Time (UTC, *née* GMT) and local time is not available.

7.23.2 Time manipulation functions

7.23.2.1 The clock function

This function is intended for measuring intervals of execution time in whatever units an implementation desires. The conflicting goals of high resolution, long interval capacity, and low timer overhead must be balanced carefully in the light of this intended use.

7.23.2.2 The `difftime` function

`difftime` is an invention of the C89 Committee. It is provided so that an implementation can store an indication of the date/time value in the most efficient format possible and still provide a method of calculating the difference between two times.

7.23.2.3 The `mktime` function

`mktime` was invented by the C89 Committee to complete the set of time functions. With this function it becomes possible to perform portable calculations involving clock times and broken-down times.

The rules on the ranges of the fields within the **`*timeptr`** record are crafted to permit useful arithmetic to be done. For instance, here is a paradigm for continuing some loop for an hour:

```
#include <time.h>
struct tm when;
time_t    now;
time_t    deadline;

/* ... */
now = time(0);
when = *localtime(&now);
when.tm_hour += 1;    // result is in the range [1,24]
deadline = mktime(&when);

printf("Loop will finish: %s\n", asctime(&when));
while (difftime(deadline, time(0)) > 0) whatever();
```

The specification of **`mktime`** guarantees that the addition to the **`tm_hour`** field produces the correct result even when the new value of **`tm_hour`** is 24, that is, a value outside the range ever returned by a library function in a **`struct tm`** object.

One of the reasons for adding this function is to replace the capability to do such arithmetic which is lost when a programmer cannot depend on **`time_t`** being an integral multiple of some known time unit.

Several readers of earlier versions of this Rationale have pointed out apparent problems in this example if **`now`** is just before a transition into or out of daylight saving time. However, **`when.tm_isdst`** indicates what sort of time was the basis of the calculation. Implementors, take heed. If this field is set to –1 on input, one truly ambiguous case involves the transition out of daylight saving time. As DST is currently legislated in the United States, the hour from 0100 to

0159 occurs twice, first as DST and then as standard time. Hence an unlabeled 0130 on this date is problematic. An implementation may choose to take this as DST or standard time, marking its decision in the **tm_isdst** field. It may also legitimately take this as invalid input and return **(time_t)(-1)**.

7.23.2.4 The time function

Since no measure is given for how precise an implementation's *best approximation* to the current time must be, an implementation could always return the same date instead of a more honest –1. This is, of course, not the intent.

7.23.3 Time conversion functions

7.23.3.1 The asctime function

Although the name of this function suggests a conflict with the principle of removing ASCII dependencies from the Standard, the name was retained due to prior art. For the same reason of existing practice, a proposal to remove the newline character from the string format was not adopted. Proposals to allow for the use of languages other than English in naming weekdays and months met with objections on grounds of prior art, and on grounds that a truly international version of this function was difficult to specify: three-letter abbreviation of weekday and month names is not universally conventional, for instance. The **strftime** function (see §7.23.3.5) provides appropriate facilities for locale-specific date and time strings.

7.23.3.3 The gmtime function

Despite objections that GMT, that is, Coordinated Universal Time (UTC), is not available in some implementations, this function was retained because UTC is a useful and widespread standard representation of time. If UTC is not available, a null pointer may be returned.

7.23.3.5 The strftime function

strftime provides a way of formatting the date and time in the appropriate locale-specific fashion using the **%c**, **%x**, and **%X** format specifiers. More generally, it allows the programmer to tailor whatever date and time format is appropriate for a given application. The facility is based on the UNIX system date command. See §7.5 for further discussion of locale specification. For the field controlled by **%P**, an implementation may wish to provide special symbols to mark noon and midnight.

A new feature of C9X: C9X extends the **strftime** specifiers, introducing **%C**, **%D**, **%e**, **%F**, **%g**, **%G**, **%h**, **%n**, **%r**, **%R**, **%t**, **%T**, **%u** and **%V**, as well as the **E** and **O** modifiers. These specifiers were chosen according to existing practice to cover long-standing POSIX practice and to allow all the formatting available in ISO 8601.

7.26 Future library directions

This subclause includes specific mention of the future direction in which the Committee intends to extend and/or restrict the library. The contents of this subclause should be considered as quite likely to become a part of the next version of the Standard. Implementors are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming implementation. Users are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming program.

8. Annexes

Most of the material in the annexes is not new. It is simply a summary of information in the Standard, collated for the convenience of users of the Standard.

New (advisory) information is found in Annex J (Common warnings) and in Annex K.5 (Common extensions). The subclause on common extensions is provided in part to give programmers even further information which may be useful in avoiding features of local dialects of C.

Annex F IEC 60559 floating-point arithmetic (normative)

A new feature of C9X.

Vagaries of floating-point arithmetic have plagued programmers and users since its inception; and they still do, even though hardware floating-point is now largely standardized. When IEEE binary floating-point standard 754 became an official standard in July 1985, 26 months before the radix-independent standard 854, several IEEE implementations were already shipping. In 1993, IEEE 754 was published as international standard IEC 559, now IEC 60559. Now virtually all new floating-point implementations conform to IEC 60559, at least in format if not to the last detail. Although these standards have been enormously successful in influencing hardware implementation, many of their features, including predictability, remain impractical or unavailable for use by programmers. IEC 60559 does not include language bindings, a cost of delivering the basic standard in a timely fashion. The C89 Committee attempted to remove conflicts with IEEE arithmetic, but did not specify IEEE support. Expediencies of programming language implementation and optimization can deny the features offered by modern hardware. In the meantime, particular companies have defined their own IEEE language extensions and libraries; and not surprisingly, lack of portability has impeded programming for these interfaces.

The Numerical C Extensions Group, NCEG, at its initial meeting in May 1989, identified support for IEEE floating-point arithmetic as one of its focus areas and organized a subgroup to produce a technical report. The subgroup benefited from the considerable C language and IEEE floating-point expertise associated with NCEG. It included individuals with substantial experience with language extensions (albeit proprietary) for IEEE floating-point. And, following after the standardization of C, it had a stable, well defined base for its extensions. Thus NCEG had a unique opportunity to solve this problem. The floating-point part of NCEG's technical report

published in 1995 was the basis for the C9X floating-point specification.

F.2 Types

Minimal conformance to the IEC 60559 floating-point standards does not require a format wider than single. The narrowest C **double** type allowed by standard C is wider than IEC 60559 single, and wider than the minimum IEC 60559 single-extended format. (IEC 60559 single-extended is an optional format intended only for those implementations that don't support double; it has at least 32 bits of precision.) Both standard C and the IEC 60559 standards would be satisfied if **float** were IEC 60559 single and **double** were an IEC 60559 single-extended format with at least 35 bits of precision. However, this specification goes slightly further by requiring **double** to be IEC 60559 double rather than just a wide IEC 60559 single-extended.

The primary objective of the IEC 60559 part of this specification is to facilitate writing portable code that exploits the floating-point standard, including its standardized single and double data formats. Bringing the C data types and the IEC 60559 standard formats into line advances this objective.

This specification accommodates what are expected to be the most important IEC 60559 floating-point architectures for general C implementations.

Because of standard C's bias toward **double**, extended-based architectures might appear to be better served by associating the C **double** type with IEC 60559 extended. However, such an approach would not allow standard C types for both IEC 60559 double and single and would go against current industry naming, in addition to undermining this specification's portability goal. Other features in the Standard, for example the type definitions **float_t** and **double_t** (defined in **<math.h>**), are intended to allow effective use of architectures with more efficient, wider formats.

The **long double** type is not required to be IEC 60559 extended because

1. some of the major IEC 60559 floating-point architectures for C implementations do not support extended.

2. double precision is adequate for a broad assortment of numerical applications.

3. extended is less standard than single or double in that only bounds for its range and precision are specified in IEC 60559.

For implementations without extended in hardware, non-IEC 60559 extended arithmetic written in software, exploiting double in hardware, provides some of the advantages of IEC 60559 extended but with significantly better performance than true IEC 60559 extended in software.

Specification for a variable-length extended type, one whose width could be changed by the user, was deemed premature. However, not unduly encumbering experimentation and future extensions, for example for variable length extended, is a goal of this specification.

Narrow-format implementations

Some C implementations, namely ones for digital signal processing, provide only the IEC 60559 single format, possibly augmented by single-extended, which may be narrower than IEC 60559 double or standard C **double**, and possibly further augmented by double in software. These non-conforming implementations might generally adopt this specification, though not matching its requirements for types.

One approach would be to match standard C **float** with single, match standard C **double** with single-extended or single; and match standard C **long double** with double, single-extended, or single. Then most of this specification could be applied straightforwardly. Users should be clearly warned that the types may not meet expectations.

Another approach would be to refer to a single-extended format as **long float** and then not recognize any C types not truly supported. This would provide ample warning for programs requiring double. The translation part of porting programs could be accomplished easily with the help of type definitions. In the absence of a double type, most of this specification for double could be adopted for the **long float** type. Having distinct types for **long float** and **double**, previously synonyms, requires more imagination.

F.5 Binary-decimal conversion

The IEC 60559 floating-point standard requires perfect rounding for a large though incomplete subset of decimal conversions. This specification goes beyond the IEC 60559 floating-point standard by requiring perfect rounding for all decimal conversions involving **DECIMAL_DIG** or fewer decimal digits and a supported IEC 60559 format, because practical methods are now available. Although not requiring correct rounding for arbitrarily wide decimal numbers, this specification is sufficient in the sense that it ensures that every internal numeric value in an IEC 60559 format can be determined as a decimal constant.

F.7 Environment

F.7.4 Constant expressions

An early version of this specification allowed translation-time constant arithmetic, but empowered the unary **+** operator, when applied to an operand, to inhibit translation-time evaluation of constant expressions. Introducing special semantics for the unary **+** operator did not seem necessary, as translation-time evaluation can be achieved by using static declarations.

F.7.5 Initialization

C89 did not specify when aggregate and union initialization is done. Otherwise, this section is merely a clarification. Note that, under the effect of an enabling **FENV_ACCESS** pragma, any exception resulting from execution-time initialization must be raised at execution time.

The specification for constant expressions and initialization does not suit C++, whose static and aggregate initializers need not be constant. Specifying all floating-point constant arithmetic and initialization to be *as if* at execution time would be suitable for C++, and given the **FENV_ACCESS** mechanism, still would allow the bulk of constant arithmetic to be done, in actuality, at translation time.

F.9 Mathematics <math.h>

HUGE_VAL cannot be implemented as

```
#define HUGE_VAL (1.0/0.0)
```

whose use may raise the divide-by-zero exception. Similarly, **INFINITY** and **NAN** cannot be implemented as **((float)(1.0/0.0))** and **((float)(0.0/0.0)**.

Special cases

The goals of the specification for special cases are to

1. define special-case results so that programs will run correctly for the widest range of inputs.

2. assure predictable special-case behavior the programmer can exploit for simpler, more efficient code.

3. allow implementations enough flexibility to provide needed performance.

Compatibility with IEC 60559 is a foremost strategy. The C9X annexes adopt the IEC 60559 specification for the functions covered by that standard, such as **sqrt** and **rint**, and follows the spirit of IEC 60559 for other functions. This means the special values (infinities, NaNs, and signed zeros) and the floating-point exceptions have a consistent meaning throughout the basic arithmetic and the libraries. At a higher level, C9X shares the IEC 60559 goal to enhance robustness through predictable behavior. For special cases, this behavior is chosen to be useful for most applications wherever possible, recognizing that it is in the nature of exceptional cases that one behavior is not best in all situations.

Typically, the tradeoff is between a numeric result that is useful in only some applications and a more pessimistic NaN result. As in IEC 60559, choosing utility over conservatism exacts a cost in specification complexity. For example, regarding NaNs only as error indicators and rules like "NaN in, NaN out" are simple but not always most useful. A NaN argument is often better interpreted as an indeterminate value. This supports the programming practice of initializing with NaNs those variables whose true values are yet to be determined, and permits returning the obvious numeric value for functions that are independent of one of their arguments. Thus **hypot(∞, NAN)** is infinity, as this would be the result regardless of the numeric value of the second argument. The C9X **fmax** and **fmin** functions return the maximum or minimum of their numerical arguments, hence **fmax(NAN, 1.2)** is 1.2, which is the desired behavior for

determining the maximum value in a set of partially initialized data.

Although a definition of **fmax** implying a NaN result for **fmax(NAN, 1.2)** might be equally useful, choosing one of the viable specifications instead of leaving the choice to the implementation has the inherent value of suiting some portable code instead of none. In other special cases, however, choices are left to the implementation because of existing practice (for example, the return value of **ilogb**) or performance issues (for example, whether certain rounding functions raise the inexact exception).

Generally, C9X eschews a NaN result where a numerical value is useful. IEC 60559 follows the same approach, as in defining overflow results to be infinite (given default rounding), which is neither mathematically correct nor useful in all cases. The results of **pow(**∞**,0)** and **pow(0,0)** are both 1, because there are applications that can exploit this definition. For example, if $x(p)$ and $y(p)$ are any analytic functions that become zero at $p = a$, then **pow(x,y)**, which equals **exp(y*log(x))**, approaches 1 as p approaches a. The result of **pow(**–2**,**∞**)** is +∞, because all large positive floating-point values are even integers. The result of **atan2(+0,+0)**, which is equivalent to **carg(**+0+i0**)**, is defined to be +0. A significant benefit is supporting a **clog** that is equivalent to **log** on the nonnegative real axis.

The choice for special-case behavior, which typically is arbitrary to some degree, was made in favor of preserving identities (involving numeric, not NaN, values), specification consistency among functions, and efficiency in implementation. The functions **hypot(***x***,***y***)** and **cabs(***x*+*y*i**)** are equivalent, as are **atan2(***y***,***x***)** and **carg(***x*+*y*i**)**, and these behave so as to be useful building blocks for other complex functions.

In certain respects, C9X is less demanding than might be expected, in order to give some flexibility in implementation, especially where the loss of utility is believed to be negligible or the cost is not justifiable. For example, C9X leaves to the implementation to decide whether functions (like **sin**) that are essentially always inexact raise the inexact flag, as there doesn't seem to be significant utility in testing an expression involving such a function for exactness. Functions in **<math.h>** are allowed to raise undeserved inexact and underflow exceptions, because determination may be difficult. C9X allows complex multiply and divide to raise spurious exceptions because of the performance cost of avoiding them.

The cost for exception behavior is intended to be modest enough for most purposes. And, as the exceptions are accessible only in code under the effect of an enabling **FENV_ACCESS** pragma, an implementation could invoke, perhaps even by default, routines that didn't have the specified exception behavior. (The pragma does not exempt the implementation from having to return specified result values.)

Underflow

The IEC 60559 floating-point standard offers the implementation multiple definitions of underflow. All resulting in the same values, the options differ only in that the thresholds when the exception is raised may differ by a rounding error. It is not intended that library functions necessarily use the same definition of underflow as the arithmetic, because the difference so

rarely matters.

Exactness

For some functions, **pow** for example, determining exactness in all cases may be too costly.

Functions have certain restrictions against raising spurious exceptions detectable by the user. For example, the implementation must hide an underflow generated by an intermediate computation of a non-tiny result.

F.9.1 Trigonometric functions

F.9.1.4 The atan2 functions

The more contentious cases are *y* and *x* both infinite or both zero. These deliver numeric results instead of NaNs in order to preserve more identities and for better utility. The specification of **atan2(0,0)** to be 0 facilitates the definition of **carg(***x*+*y*i**)** as **atan2(***x***,***y***)** and **clog(***z***)** as **log(**|*x*|**)** + **I*carg(***z***)** so that **clog(***z***)** agrees with **log(***x***)** on the real axis.

The specification of **atan2(**∞**,**∞**)** as π/4 indicates the reasonable quadrant, preserving some information in preference to none.

F.9.4 Power and absolute value functions

F.9.4.3 The hypot functions

Note that **hypot(INFINITY,NAN)** returns **+INFINITY**, under the justification that **hypot(INFINITY,***y***)** is +∞ for any numeric value *y*.

F.9.4.4 The pow functions

pow(*x***,0)** is specified to return 1 for any *x*, because there are significant applications where 1 is more useful than NaN. **pow(**f(*t*)**,**g(*t*)**)** approaches 1 in all cases where f and g are analytic functions and g(*t*) approaches zero. The result 1 better supports applications where the second argument is integral. **pow(NAN,0)** returns 1.0 on the general principle that if a result is independent of the numerical value of an argument, then that result is appropriate if that argument is a NaN.

F.9.9 Maximum, minimum, and positive difference functions

F.9.9.2 The fmax functions

Some applications may be better served by a **max** function that would return a NaN if one of its arguments were a NaN:

```
{ return (isgreaterequal(x, y) || isnan(x)) ? x : y; }
```

Note that two branches still are required for symmetry in NaN cases.

Annex G IEC 60559-compatible complex arithmetic (informative)

A new feature of C9X.

Although the specification in Annex G is fundamental for IEC 60559 style complex arithmetic, the annex is designated informative because of insufficient prior art for normative status.

G.2 Types

Although not present in older complex arithmetic facilities such as Fortran's, the imaginary types naturally model the imaginary axis of complex analysis, promote computational and storage efficiency, and capture the completeness and consistency of IEC 60559 arithmetic for the complex domain. See also rationale for §G.5.

The representation and alignment requirements of imaginary types are intended to allow imaginary arguments to **`fprintf`** and **`fscanf`**; however, technically speaking, this invokes undefined behavior because corresponding real and imaginary types are not compatible types. The recommended practice is that implementations promote arguments with **`float imaginary`** types to **`double imaginary`**, and treat the arguments as if they had the corresponding real type. When no prototype is in scope, function calls involving arguments with imaginary types should behave in a similar manner.

G.5 Binary operators

G.5.1 Multiplicative operators

Text book formulas for complex arithmetic tend to turn infinite inputs into NaNs, often losing useful information unnecessarily. For example,

$$(1+i0)(\infty+i\infty) \Rightarrow (1\times\infty - 0\times\infty) + i(0\times\infty+1\times\infty) \Rightarrow \text{NaN+iNaN}$$

and

$$\text{cexp}(\infty+i\text{NaN}) \Rightarrow \exp(\infty)\times(\text{cis}(\text{NaN})) \Rightarrow \text{NaN+iNaN}$$

but for applications modeling the Riemann sphere, result values of infinite magnitude would be more useful (even though their phase angles may be meaningless). In order to support the one-infinity model, C9X regards any complex value with at least one infinite part as a complex infinity (even if the other part is a NaN), and guarantees that operations and functions honor

basic properties of infinities, and provides the **`cproj`** function to map all infinities to a canonical one. For example, a finite non-zero value times an infinity must be an infinity, hence (1+i0)*(∞+i∞) must be an infinity. In the same spirit, cexp(∞+iNaN) is an infinity and cexp(-∞+iNaN) is a complex zero, which preserve cabs(cexp(*x*+i*y*)) = exp(*x*).

C9X treats multiple infinities so as to preserve directional information where possible, despite the inherent limitations of the ordered-pair (Cartesian) representation. The product of the imaginary unit and a real infinity is a correctly signed imaginary infinity, $i \times \infty = i\infty$. And

$$i \times (\infty - i\infty) = \infty + i\infty$$

which at least indicates the reasonable quadrant.

C9X allows complex multiply and divide to raise spurious exceptions because of the performance cost of avoiding them.

G.6 Complex arithmetic <complex.h>

See also rationale for §F.9 and §G.4.1.

Positing the imaginary unit constant is a natural analog to the mathematical notion of augmenting the reals with the imaginary unit. It allows writing imaginary and complex expressions in common mathematical style, for example **`x + I*y`**. Note that the multiplication here affects translated code, but does not necessitate an actual floating-point multiply, nor does the addition necessitate a floating-point add.

IEC 60559 compatibility is a primary rationale for the imaginary types. Without them the traditional complex arithmetic programming facilities prove fundamentally incompatible with IEC 60559 in the treatment of special values; with them compatibility comes surprisingly naturally. Very little special-case specification is required for imaginary types.

The imaginary types, together with the usual arithmetic conversion rules and operator specifications (see §G.4), allow substantially more efficient code. For example, multiplication of an imaginary by a complex can be implemented straightforwardly with two multiplications, instead of four multiplications and two additions.

In the absence of imaginary types, macros would be required in order to create certain special values. For example, 0+∞i could be created by **`CMPLX(0.0,INFINITY)`**. With the imaginary types, imaginary infinity is simply the value of **`I*INFINITY`**. (If imaginary types are not supported and **`I`** is **`_Complex_I`**, then **`INFINITY*I`** would result in a real part of NaN and an invalid exception.) With imaginary types, values of **`I*y`** and **`x + I*y`**, where **`x`** and **`y`** are real floating values, cover all values of the imaginary and complex types, hence eliminating this need for the complex macros.

Some programs are expected to use the imaginary types implicitly in constructions with the

imaginary unit **I**, such as **x + I*y**, and not explicitly in declarations. This suggests making the imaginary types private to the implementation and not available for explicit program declarations. However, such an approach was rejected as being less in the open spirit of C, and not much simpler. For the same reasons, the approach of treating imaginariness as an attribute of certain complex expressions, rather than as additional types, was rejected.

Another proposal was to regard the special values (infinities, NaNs, and signed zeros) as outside the model. This would allow any behavior when special values occur, including much that is prescribed by this specification. However, this approach would not serve the growing majority of implementations, including all IEC 60559 ones, that support the special values. These implementations would require additional specification in order to provide a consistent extension of their treatment of special cases in the real domain. On the other hand, implementations not supporting special values should have little additional trouble implementing imaginary types as proposed here.

The efficiency benefits of the imaginary types goes beyond what the implementation provides. In many cases programmers have foregone a programming language's complex arithmetic facilities, which, lacking an imaginary type, required contiguous storage of both real and imaginary parts; programmers could store and manipulate complex values more efficiently using real arithmetic directly. The imaginary types enable programmers to exploit the efficiency of the real formats without having to give up support for complex arithmetic semantics.

Care is taken throughout so that the sign of zero is available for distinguishing the sides of a branch cut along the axes, even at infinities. Therefore

$$\overline{\sqrt{-\infty+i0}} = 0+i\infty$$

and by conjugation

$$\overline{\sqrt{-\infty-i0}} = 0-i\infty$$

G.7 Type-generic math <tgmath.h>

Exploiting the fact that some functions map the imaginary axis onto the real or imaginary axis gains more efficient calculation involving imaginaries, and better meets user expectations in some cases. However, dropping out of the complex domain may lead to surprises as subsequent operations may be done with real functions, which generally are more restrictive than their complex counterparts. For example, **sqrt(-cos(I))** invokes the real **sqrt** function, which is invalid for the negative real value **-cos(I)**, whereas the complex **sqrt** is valid everywhere.

Annex H Language independent arithmetic (informative)

A new feature of C9X.

LIA-1 was not made a normative part of C9X for three reasons: implementation vendors saw no

need to add LIA-1 support because customers are not asking for it, LIA-1 may change now that work on LIA-2 is finishing and work on LIA-3 is starting, and the Committee did not wish to rush a possibly incomplete specification into C9X at the last moment. A proposed binding between C and LIA-1 was produced a few months before C9X was frozen. That binding was a compromise between those who believe LIA-1 should be forgotten and those who wanted full LIA-1 and more (for example, C signal handlers that could patch up exceptions on the fly and restart the exceptional floating-point instruction). It took several years for the NCEG floating-point specification to settle down, so it was assumed that it would take a similar timeframe to get the LIA-1 binding stable. The Committee did not wish to delay C9X for this one item. An informative LIA-1 annex was added, however, because all programming languages covered by ISO/IEC JTC1 SC22 standards are expected to review LIA-1 and incorporate and further define the binding between LIA-1 and each programming language.

C9X's binding between C and LIA-1 differs from LIA-1's C binding in three cases in which the Committee believes that LIA-1 is incorrect. First, LIA-2 and LIA-1 have different behaviors for mathematical pole exceptions (similar to finite non-zero divided by zero and log(0)). The Committee believes that LIA-2 is better and that LIA-1 will be changed to match LIA-2. Second, the existing hardware that many computers use for conversion from floating-point type to integer type raises the undefined exception, instead of the required integer overflow, for values that are out of bounds. Third, requiring support for signaling NaNs on IEC 60559 implementations should be optional because existing hardware support for signaling NaNs is inconsistent.

H.3.1.2 Traps

The math library is required by both C89 and C9X to be atomic in that no exceptions (raise of a signal and invocation of a user's signal handler) may be visible in the user's program. Because of that requirement, C9X cannot meet LIA's requirement of either trap-and-terminate or trap-and-resume for math library errors. On the other hand, both kinds of traps are allowed for the arithmetic operations.

Annex I Universal character names for identifiers (normative)

A new feature of C9X.

MSE. Multibyte Support Extensions Rationale

This text was taken from the rationale furnished with the amendment, ISO/IEC 9899/AMD1:1995, called simply AMD1 in this Annex.

MSE.1 MSE Background

Most traditional computer systems and computer languages, including traditional C, have an assumption, sometimes undocumented, that a "character" can be handled as an atomic quantity associated with a single memory storage unit — a "byte" or something similar. This is not true in general. For example, a Japanese, Chinese or Korean character usually requires a representation of two or three bytes; this is a *multibyte character* as defined by §3.72 and §5.2.1.2. Even in the Latin world, a multibyte coded character set appears. This conflict is called the *byte and character problem*.

A related concern in this area is how to address having at least two different meanings for string length: number of bytes and number of characters.

To cope with these problems, many technical experts, particularly in Japan, have developed their own sets of additional multibyte character functions, sometimes independently and sometimes cooperatively. Fortunately, the developed extensions are actually quite similar. It can be said that in the process they have found common features for multibyte character support. Moreover, the industry currently has many good implementations of such support.

The above in no way denigrates the important groundwork in multibyte- and wide-character programming provided by C90:

- Both the source and execution character sets can contain multibyte characters with possibly different encodings, even in the "C" locale.
- Multibyte characters are permitted in comments, string literals, character constants, and header names.
- The language supports wide-character constants and strings.
- The library has five basic functions that convert between multibyte and wide characters.

However, the five functions are often too restrictive and too primitive to develop portable international programs that manage characters. Consider a simple program that wants to count the number of characters, not bytes, in its input.

The prototypical program,

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

```
int main(void)
{
   int c, n = 0;
   while ((c = getchar()) != EOF)
        n++;
   printf("Count = %d\n", n);
   return 0;
}
```

does not work as expected if the input contains multibyte characters; it always counts the number of bytes. It is certainly possible to rewrite this program using just some of the five basic conversion functions, but the simplicity and elegance of the above are lost.

C90 deliberately chose not to invent a more complete multibyte- and wide-character library, choosing instead to await their natural development as the C community acquired more experience with wide characters. The task of the Committee was to study the various existing implementations and, with care, develop the first amendment to C90. The set of developed library functions is commonly called the *MSE* (Multibyte Support Extension).

Similarly, C90 deliberately chose not to address in detail the problem of writing C source code with character sets such as the national variants of ISO/IEC 646. These variants often redefine several of the punctuation characters used to write a number of C tokens. The partial solution adopted was to add *trigraphs* to the language. Thus, for example, **??<** can appear anywhere in a C program that **{** can appear, even within a character constant or a string literal.

AMD1 responds to an international sentiment that more readable alternatives should also be provided wherever possible. Thus, it adds to the language alternate spellings of several tokens. It also adds a library header, **<iso646.h>**, that defines a number of macros that expand to still other tokens which are less readable when spelled with trigraphs. Note, however, that trigraphs are still the only alternative to writing certain characters within a character constant or a string literal.

An important goal of any amendment to an international standard is to minimize quiet changes — changes in the definition of a programming language that transform a previously valid program into another valid program, or into an invalid program that need not generate a diagnostic message, with different behavior. (By contrast, changes that invalidate a previously valid program are generally considered palatable if they generate an obligatory diagnostic message at translation time.) Nevertheless, AMD1 knowingly introduces two classes of quiet changes:

- digraphs — The tokens **%:** and **%:%:** are just sequences of preprocessing tokens in C90 but become single preprocessing tokens with specific meanings in AMD1. An existing program that uses either of these tokens in a macro argument can behave differently as a result of AMD1.

- new function names — Several names with external linkage, such as **btowc**, not reserved to the implementation in C90, are now so reserved. An existing program that uses any of these names can behave differently as a result of AMD1.

MSE.2 Programming model based on wide characters

Using the MSE functions, a multibyte-character-handling program can be written as easily and in the same style as a traditional single-byte-based program. A programming model based on MSE function is as follows. First, a multibyte character or a multibyte string is read from an external file into a **wchar_t** object or a **wchar_t** array object by the **fgetwc** function, or other input functions based on the **fgetwc** function such as **getwchar**, **getwc**, or **fgetws**. During this read operation, a code conversion occurs: the input function converts the multibyte character to the corresponding wide character *as if* by a call to the **mbrtowc** function.

After all necessary multibyte characters are read and converted, the **wchar_t** objects are processed in memory by the MSE functions such as **isw***xxx*, **wcstod**, **wcscpy**, **wmemcmp**, and so on. Finally, the resulting **wchar_t** objects are written to an external file as a sequence of multibyte characters by the **fputwc** function or other output functions based on the **fputwc** function such as **putwchar**, **putwc**, or **fputws**. During this write operation, a code conversion occurs: the output function converts the wide character to the corresponding multibyte character *as if* by a call to the **wcrtomb** function.

In the case of the formatted input/output functions, a similar programming style can be applied, except that the character code conversion may also be done through extended conversion specifiers such as **%ls** and **%lc**. For example, the wide-character-based program corresponding to that shown in §MSE.1 can be written as follows:

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

int main(void)
{
   wint_t wc;
   int n = 0;

   while ((wc = getwchar()) != WEOF)
        n++;
   wprintf(L"Count = %d\n", n);
   return 0;
}
```

MSE.3 Parallelism versus improvement

When defining the MSE library functions, the Committee could have chosen a design policy based either on *parallelism* or on *improvement*. "Parallelism" means that a function interface defined in AMD1 is similar to the corresponding single-byte function in C90. The number of parameters in corresponding functions are exactly same, and the types of parameters and the types of return values have a simple correspondence:

char ↔ wchar_t **int ↔ wint_t**

An approach using this policy would have been relatively easy.

On the other hand, “improvement” means that a function interface in AMD1 is changed from the corresponding single-byte functions in C90 in order to resolve problems potentially contained in the existing functions. Or, a corresponding function is not introduced in AMD1 when the functionality can be better attained through other functions.

In an attempt to achieve improvement, there were numerous collisions of viewpoints on how to get the most appropriate interface. Moreover, much careful consideration and discussion among various experts in this area was necessary to decide which policy should be taken for each function. AMD1 is the result of this process.

The following is a list of the corresponding functions that manipulate characters:

C90	AMD1
isalnum	**iswalnum**
isalpha	**iswalpha**
iscntrl	**iswcntrl**
isdigit	**iswdigit**
isgraph	**iswgraph**
islower	**iswlower**
isprint	**iswprint**
ispunct	**iswpunct**
isspace	**iswspace**
isupper	**iswupper**
isxdigit	**iswxdigit**
tolower	**towlower**
toupper	**towupper**
fgetc	**fgetwc**
fgets	**fgetws**
fputc	**fputwc**
fputs	**fputws**
fprintf	**fwprintf**
fscanf	**fwscanf**
getc	**getwc**
getchar	**getwchar**
printf	**wprintf**
putc	**putwc**
putchar	**putwchar**
scanf	**wscanf**
sprintf	**swprintf**
sscanf	**swscanf**
ungetc	**ungetwc**
vfprintf	**vwfprintf**

vprintf	**vwprintf**
vsprintf	**vswprintf**
memchr	**wmemchr**
memcmp	**wmemcmp**
memcpy	**wmemcpy**
memmove	**wmemmove**
memset	**wmemset**
strcat	**wcscat**
strcmp	**wcscmp**
strcoll	**wcscoll**
strcpy	**wcscpy**
strchr	**wcschr**
strcspn	**wcscspn**
strftime	**wcsftime**
strlen	**wcslen**
strncat	**wcsncat**
strncmp	**wcsncmp**
strncpy	**wcsncpy**
strpbrk	**wcspbrk**
strrchr	**wcsrchr**
strspn	**wcsspn**
strstr	**wcsstr**
strtod	**wcstod**
strtok	**wcstok**
strtol	**wcstol**
strtoul	**wcstoul**
strxfrm	**wcsxfrm**

Additional parallel functions were added in C9X

isblank	**iswblank**
snprintf	(**swprintf**)
vfscanf	**vfwscanf**
vscanf	**vwscanf**
vsnprintf	(**vswprintf**)
vsscanf	**vswscanf**
strtof	**wcstof**
strtold	**wcstold**
strtoll	**wcstoll**
strtoull	**wcstoull**

Note that there may still be subtle differences (see for example §MSE.6.2).

The following functions have different interfaces between single-byte and wide-character versions:

- Members of the **sprintf** family based on wide characters all have an extra **size_t** parameter in order to avoid overflowing the buffer. Compare:

```
int sprintf(char *s, const char *format, ...);
int swprintf(wchar_t *s, size_t n,
             const wchar_t *format, ...);
int vsprintf(char *s, const char *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format,
              va_list arg);
```

- **wcstok**, the wide-character version of **strtok**, has an extra **wchar_t **** parameter in order to eliminate the internal memory that the **strtok** function has to maintain. Compare:

```
char *strtok(char *s1, const char *s2);
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2,
                wchar_t **ptr);
```

The following is a list of the functions in C90, with **atoll** added in C9X, that do not have parallel functions in AMD1 for any of several reasons such as redundancy, dangerous behavior, or a lack of need in a wide-character-based program. Most of these can be rather directly replaced by other functions:

```
atof
atoi
atol
atoll
gets
perror
puts
strerror
```

Finally, the following is a list of the functions in AMD1 that do not have parallel functions in C90. They were introduced either to achieve better control over the conversion between multibyte characters and wide characters, or to give character handling programs greater flexibility and simplicity:

```
btowc
fwide
iswctype
mbrlen
mbrtowc
mbsinit
mbsrtowcs
towctrans
wcrtomb
wcsrtombs
```

```
wctob
wctrans
wctype
```

MSE.4 Support for invariant ISO/IEC 646

With its rich set of operators and punctuators, the C language makes heavy demands on the ASCII character set. Even before the language was standardized, it presented problems to those who would move C to EBCDIC machines. More than one vendor provided alternate spellings for some of the tokens that used characters with no EBCDIC equivalent. With the spread of C throughout the world, such representation problems have only grown worse.

ISO/IEC 646, the international standard corresponding to ASCII, permits national variants of a number of the characters used by C. Strictly speaking, this is not a problem in representing C programs, since the necessary characters exist in all such variants: they just print oddly. Displaying C programs for human edification suffers, however, since the operators and punctuators can be hard to recognize in their various altered forms.

C90 addresses the problem in a different way. It provides replacements at the level of individual characters using three-character sequences called *trigraphs* (see §5.2.1.1). For example, **??<** is entirely equivalent to **{**, even within a character constant or string literal. While this approach provides a solution for the known limitations of EBCDIC (except for the exclamation mark) and ISO/IEC 646, the result is arguably not highly readable.

Thus, AMD1 provides a set of more readable *digraphs* (see §6.4.6). These are two-character alternate spellings for several of the operators and punctuators that can be hard to read with ISO/IEC 646 national variants. Trigraphs are still required within character constants and string literals, but at least the more common operators and punctuators can have more suggestive spellings using digraphs.

The added digraphs were intentionally kept to a minimum. Wherever possible, the Committee instead provided alternate spellings for operators in the form of macros defined in the new header **<iso646.h>**. Alternate spellings are provided for the preprocessing operators **#** and **##** because they cannot be replaced by macro names. Digraphs are also provided for the punctuators **[**, **]**, **{**, and **}** because macro names proved to be a less readable alternative. The Committee recognizes that the solution offered in this header is incomplete and involves a mixture of approaches, but nevertheless believes that it can help make Standard C programs more readable.

MSE.5 Headers

MSE.5.1 <wchar.h>

MSE.5.1.1 Prototypes in <wchar.h>

Function prototypes for the MSE library functions had to be included in some header. The

Committee considered following ideas:

1. Introduce new headers such as **<wctype.h>**, **<wstdio.h>**, and **<wstring.h>**, corresponding to the existing headers specified in C90 such as **<ctype.h>**, **<stdio.h>**, and **<string.h>**.

2. Declare all the MSE function prototypes in **<stdlib.h>** where **wchar_t** is already defined.

3. Introduce a new header and declare all the MSE function prototypes in the new header.

4. Declare the MSE function prototypes in the existing headers specified in C90 which are most closely related to these functions.

The drawback to idea 1 is that the relationship between new headers and existing ones becomes complicated. For example, there may be dependencies between the old and the new headers, so one or more headers may have to be included prior to including **<wstdio.h>**, as in:

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

The drawback to idea 2 is that the program has to include many prototype declarations even if the program does not need declarations in **<stdlib.h>** other than existing ones. At the time, the Committee strongly opposed adding several identifiers to existing headers for this purpose.

The drawback to idea 3 is that it introduces an asymmetry between existing headers and the new headers.

The drawback to idea 4, as with idea 2, is that the Committee strongly opposed adding many identifiers to existing headers.

So the Committee decided to introduce a new header, **<wchar.h>**, as the least objectionable way to declare all MSE function prototypes. Later, the Committee split off the functions analogous to those in **<ctype.h>** and placed their declarations in the header, **<wctype.h>**, as described in §MSE.5.2.

MSE.5.1.2 Types and macros in <wchar.h>

The Committee was concerned that the definitions of types and macros in **<wchar.h>** be specified efficiently. One goal was to require that only the header **<wchar.h>** need be included to use the MSE library functions; but there were strong objections to declaring existing types such as **FILE** in the new header.

The definitions in **<wchar.h>** are thus limited to those types and macros that are largely independent of the existing library. The existing header **<stdio.h>** must also be included along with **<wchar.h>** when the program needs explicit definitions of either of the types **FILE** and

fpos_t.

MSE.5.2 <wctype.h>

The Committee originally intended to place all MSE functionality in a single header, **<wchar.h>**, as explained in §MSE.5.1.1. It found, however, that this header was excessively large, even compared to the existing large headers, **<stdio.h>** and **<stdlib.h>**. The Committee also observed that the wide-character classification and mapping functions, typically have names of the form **isw***xxx* or **tow***xxx*, seemed to form a separate group. A translation unit could well make use of most of the functionality of the MSE without using this separate group. Equally, a translation unit might need the wide-character classification and mapping functions without needing the other MSE functions.

The Committee therefore decided to form a separate header, **<wctype.h>**, closely analogous to the existing **<ctype.h>**. That division also reduced the size of **<wchar.h>** to more manageable proportions.

MSE.6 Wide-character classification functions

Eleven **isw***xxx* functions were introduced to correspond to the character-testing functions defined in C90. Each wide-character testing function is specified in parallel with the matching single-byte character handling function, however the following changes were also introduced.

MSE.6.1 Locale dependency of **isw***xxx* functions

The behavior of character-testing functions in C90 is affected by the current locale, and some of the functions have implementation-defined aspects only when not in the "C" locale. For example, in the "C" locale, **islower** returns true (nonzero) only for lower case letters as defined in §5.2.1.

This existing "C" locale restriction for character testing functions in C90 has been replaced with a superseding constraint for wide-character-testing functions. There is no special description of "C" locale behavior for the **isw***xxx* functions. Instead, the following rule is applied to any locale: when a character **c** causes **is***xxx***(c)** to return true, the corresponding wide character **wc** shall cause the corresponding **isw***xxx***(wc)** to return true.

is*xxx***(c) != 0** ⇒ **isw***xxx***(wc) != 0**

where **c == wctob(wc)**. Note that the converse relationship does not necessarily hold.

MSE.6.2 Changed space character handling

The space character, **' '**, is treated specially in **isprint**, **isgraph**, and **ispunct**. Handling of the space character in the corresponding wide-character functions differs from that specified in C90. The corresponding wide-character functions return true for all wide characters for which

iswspace returns true, instead of just the single space character; therefore the behaviors of the **iswgraph** and **iswpunct** functions may differ from their matching functions in C90 in this regard (see the footnote concerning **iswgraph** in §7.25.2.1.6).

MSE.7 Extensible classification and mapping functions

There are eleven standard character-testing functions defined in C90. As the number of supported locales increases, the requirements for additional character classifications grows, and varies from locale to locale. To satisfy this requirement, many existing implementations, especially for non-English-speaking countries, have been defining new **is***xxx* functions, such as **iskanji**, **ishanja**, and so forth.

This approach, however, adds to the global namespace clutter (although the names have been reserved) and is not flexible at all in supporting additional classification requirements. Therefore, in AMD1, a pair of extensible wide character classification functions, **wctype** and **iswctype**, are introduced to satisfy the open-ended requirements for character classification. Since the name of a character classification is passed as an argument to the **wctype** function, it does not add to problem of global namespace pollution; and these generic interfaces allow a program to test if the classification is available in the current locale, and to test for locale-specific character classifications, such as Kanji or Hiragana in Japanese.

In the same way, a pair of wide character mapping functions, **wctrans** and **towctrans**, are introduced to support locale-specific character mappings. One of the example of applying this functionality is the mappings between Hiragana and Katakana in a Japanese character set.

MSE.8 Generalized multibyte characters

C90 intentionally restricted the class of acceptable encodings for multibyte characters. One goal was to ensure that, at least in the initial shift state, the characters in the basic C character set have multibyte representations that are single characters with the same code as the single-byte representation. The other was to ensure that the null byte should always be available as an end-of-string indicator. Hence, it should never appear as the second or subsequent byte of any multibyte code. For example, the one-byte sequence **'a'** should always represent **L'a'**, at least initially, and **'\0'** should always represent **L'\0'**.

While these may be reasonable restrictions within a C program, they hamper the ability of the MSE functions to read arbitrary wide-oriented files. For example, a system may wish to represent files as sequences of ISO/IEC 10646 characters. Reading or writing such a file as a wide-oriented stream should be an easy matter. At most, the library may have to map between native and some canonical byte order in the file. In fact, it is easy to think of an ISO/IEC 10646 file as being some form of multibyte file except that it violates both restrictions described above: the code for **'a'** can look like the four-byte sequence **\0\0\0a** for example.)

Thus, the MSE introduces the notion of a *generalized multibyte encoding*. It subsumes all the ways the Committee can currently imagine that operating systems will represent files containing

characters from a large character set. (Such encodings are valid only in files; they are still not permitted as internal multibyte encodings.)

MSE.9 Streams and files

MSE.9.1 Conversion state

It is necessary to convert between multibyte characters and wide characters within wide character input/output functions. The conversion state introduced in §7.24.6 is used to help perform this conversion. Every wide character input/output function makes use of, and updates, the conversion state held in the **`FILE`** object controlling the wide-oriented stream.

The conversion state in the **`FILE`** object augments the file position within the corresponding multibyte character stream with the parse state for the next multibyte character to be obtained from that stream. For state-dependent encodings, the remembered shift state is a part of this parse state, and therefore a part of the conversion state. (Note that a multibyte encoding that has *any* characters requiring two or more bytes needs a nontrivial conversion state even if it is not a state-dependent encoding.)

The wide character input/output functions behave *as if*:

- a **`FILE`** object includes a hidden **`mbstate_t`** object.
- the wide character input/output functions use this hidden object as the state argument to the **`mbrtowc`** or **`wcrtomb`** functions that perform the conversion between multibyte characters in the file and wide characters inside the program.

MSE.9.2 Implementation

The Committee assumed that only wide character input/output functions can maintain consistency between the conversion state information and the stream. The byte input/output functions do nothing with the conversion state information in the **`FILE`** object. The wide character input/output functions are designed on the premise that they always begin executing with the stream positioned at the boundary between two multibyte characters.

The Committee felt that it would be intolerable to require implementors to implement these functions without such a guarantee. Since executing a byte input/output function on a wide-oriented stream may well leave the file position indicator at other than the boundary between two multibyte characters, the Committee decided to prohibit such use of the byte input/output functions.

MSE.9.2.1 Seek operations

An **`fpos_t`** object for a stream in a state-dependent encoding includes the shift state information for the corresponding stream. In order to ensure the behavior of subsequent wide character

input/output functions in a state-dependent encoding environment, a seek operation should reset the conversion state corresponding to the file position as well as restoring the file position.

The traditional seek functions, **fseek** and **ftell**, may not be adequate in such an environment because even an object of type **long int** may be too small to hold both the conversion state information and the file position indicator. Thus, the newer **fsetpos** and **fgetpos** are preferred, since they can store as much information as necessary in an **fpos_t** object.

MSE.9.2.2 State-dependent encodings

With state-dependent encodings, a **FILE** object must include the conversion state for the stream. The Committee felt strongly that programmers should not have to handle the tedious details of keeping track of conversion states for wide-character input/output. There is no means, however, for programmers to access the internal shift state or conversion state in a **FILE** object.

MSE.9.2.3 Multiple encoding environments

A *multiple encoding environment* has two or more different encoding schemes for files. In such an environment, some programmers will want to handle two or more multibyte character encodings on a single platform, possibly within a single program. There is, for example, an environment in Japan that has two or more encoding rules for a single character set. Most implementations for Japanese environments should provide for such multiple encodings.

During program execution, the wide character input/output functions get information about the current encodings from the **LC_CTYPE** category of the current locale when the conversion state is bound, as described immediately below. When writing a program for a multiple encoding environment, the programmer should be aware of the proper **LC_CTYPE** category when opening a file and establishing its orientation. During subsequent accesses to the file, the **LC_CTYPE** category need not be restored by the program.

The encoding rule information is effectively a part of the conversion state. Thus, the information about the encoding rule should be stored with the hidden **mbstate_t** object within the **FILE** object. Some implementations may even choose to store the encoding rule as part of the value of an **fpos_t** object.

The conversion state just created when a file is opened is said to have *unbound* state because it has no relations to any of the encoding rules. Just after the first wide character input/output operation, the conversion state is *bound* to the encoding rule which corresponds to the **LC_CTYPE** category of the current locale. The following is a summary of the relations between various objects, the shift state, and the encoding rules:

	fpos_t	**FILE**
shift state	included	included

encoding rule	maybe	included
changing **LC_CTYPE** (unbound)	no effect	affected
changing **LC_CTYPE** (bound)	no effect	no effect

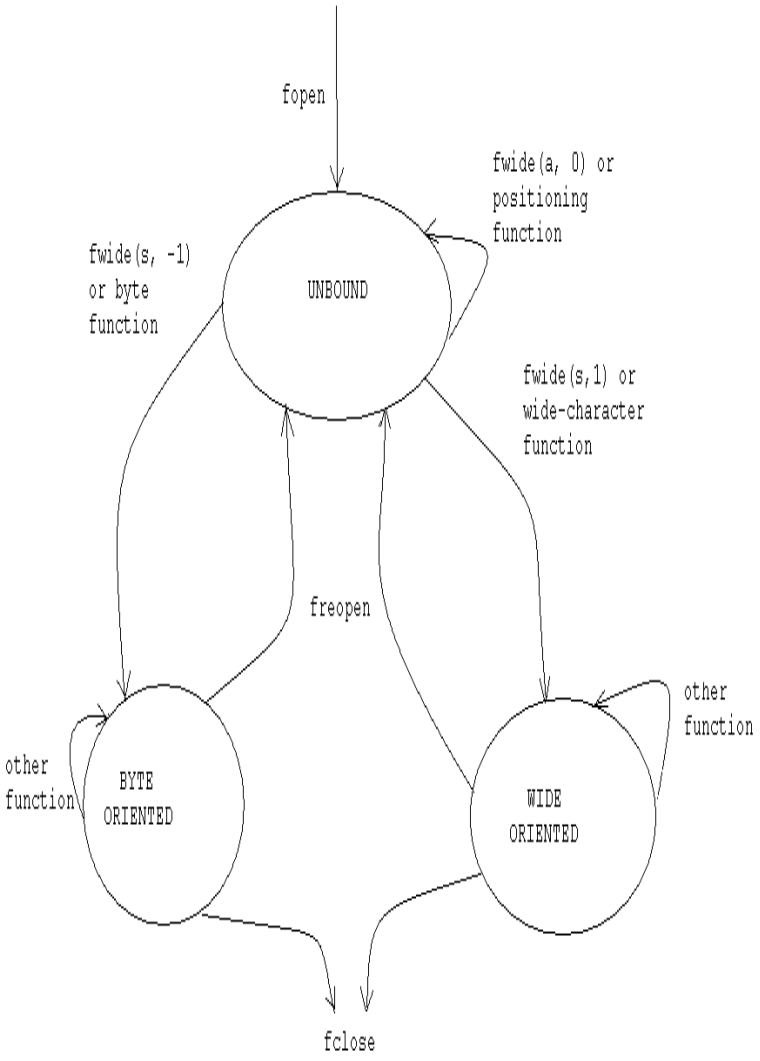
MSE.9.3 Byte versus wide-character input/output

Both the wide character input/output functions and the byte input/output functions refer the same type of object, a **FILE** object. As described in §MSE.9.2, however, there is a constraint on mixed usage of the two types of input/output functions. That is, if a wide character input/output function is executed for a **FILE** object, its stream becomes wide-oriented and no byte input/output function may be applied later, and conversely.

The reason for this constraint is to ensure consistency between the current file position and the current conversion state in the **FILE** object. Executing one of the byte input/output functions for a wide-oriented stream breaks this consistency because the byte input/output functions may, and should, ignore the conversion state information in the **FILE** object.

The diagram A1 shows the state transitions of a stream in response to various input/output functions.

Diagram A1


fopen
fwide(a, 0) or
positioning
function
fwide(s, -1)
or byte
function
UNBOUND
fwide(s,1) or
wide-character
function
freopen
other
function
other
function
BYTE
ORIENTED
WIDE
ORIENTED
fclose

MSE.9.4 Text versus binary input/output

In some implementations such as UNIX, there are streams which look the same whether read or written as text or binary. For example, arbitrary file positioning operations are supported even in text mode. In such an implementation, the Committee specified a file opened as a binary stream should obey the usage constraints placed upon text streams when accessed as a wide-oriented stream (for example, the restrictions on file positioning operations should be obeyed).

So an implementation of the wide character input/output functions can rely on the premise that programmers use the wide character input/output functions with a binary stream under the same constraints as for a text stream. An implementation may also provide wide character input/output functions that behave correctly on an unconstrained binary stream, however the behavior of the wide character input/output functions on such an unconstrained binary stream cannot be ensured by all implementations.

MSE.10 Formatted input/output functions

MSE.10.1 Enhancing existing formatted input/output functions

The simplest extension for wide character input/output is to use existing formatted input/output functions with existing byte-oriented streams. In this case, data such as strings that consist of characters only are treated as sequences of wide characters, and other data such as numerical values are treated as sequences of single-byte characters. Though this is not a complete model for wide character processing, it is a common extension among some existing implementations in Japan, and so the Committee decided to include a similar extension.

The original intent was to add the new conversion specifiers **`%S`** and **`%C`** to the existing formatted input and output functions to handle a wide character string and a wide character respectively. After long discussions about the actual implementation and future library directions, these specifiers were withdrawn. They were replaced with the qualified conversion specifiers, **`%ls`** and **`%lc`**, with the addition of **`%l[`**...**`]`** in the formatted input functions. Note that even though the new qualifier is introduced as an extension for wide character processing, the field width and the precision still specify the number of *bytes* in the multibyte representation in the stream.

To implement these new conversion specifiers efficiently, a new set of functions is required to parse or generate multibyte sequences “restartably.” Thus, the functions described in §7.24.6.4 were introduced.

Because these new conversions are pure extensions to C90, they have several essential restrictions on them, and so it is expected that they will be most useful in implementations that are not state-dependent. The restrictions are:

- **`fscanf`** function — In a state-dependent encoding, one or more shift sequences may be included in the format to be matched as part of an ordinary multibyte character literal text directive. Shift sequences may also be included in an input string. Because the **`fscanf`**

function treats these shift sequences in exactly the same way as for single byte characters, an unexpected match may occur or an expected match might not occur (see §4.6.2.3.2 of AMD1 for some examples).

- **`fprintf`** function — In a state-dependent encoding, redundant shift sequences may be written.

MSE.10.2 Formatted wide-character input/output functions

In the early MSE, formatted wide character input/output functions were not introduced because an extension to existing formatted input/output functions seemed to be sufficient. After considering the complete model for wide character handling, the need for formatted wide character input/output functions was recognized.

Formatted wide character input/output functions have much the same conversion specifiers and qualifiers as existing formatted input/output functions, even including the qualified conversion specifiers, **`%lc`**, **`%ls`**, and **`%l[`**...**`]`**, but because the format string consists of wide characters and the field width and precision specify the number of wide characters, some of the restrictions on existing functions are removed in the new functions. This means that wide characters are read and written under tighter control of the format string.

MSE.11 Adding the `fwide` function

While the Committee believes that the MSE provides reasonably complete functionality for manipulating wide-oriented files, it noticed that no reliable mechanism existed for testing or setting the orientation of a stream. The program can try certain operations to see if they fail, but that is risky and still not a complete strategy. The Committee therefore introduced the **`fwide`** function as a means of forcing a newly opened stream into the desired orientation without attempting any input/output on the stream. The function also serves as a passive means of testing the orientation of a stream, either before or after the orientation has been fixed; and it serves as a way to bind an encoding rule to a wide-oriented stream under more controlled circumstances (see §MSE.9.2.3).

MSE.12 Single-byte wide-character conversion functions

Two single-byte wide character conversion functions, **`btowc`** and **`wctob`**, were introduced in AMD1. These functions simplify mappings between a single-byte character and its corresponding wide character, if any.

C90 specifies the rule that **`L'x' == 'x'`** for any member **`x`** of the basic character set. The Committee discussed whether to relax or tighten this rule. In AMD1, this rule is preserved without any changes. Applying the rule to all single-byte characters, however, imposes an unnecessary constraint on implementations with regard to wide-character encodings. It prohibits an implementation from having a common wide-character encoding for multiple multibyte encodings.

On the other hand, relaxing or removing the rule was considered to be inappropriate in terms of

practical implementations. The new **wctob** function can be used to test safely and quickly whether a wide character corresponds to some single-byte character. For example, when the format string passed to **scanf** is parsed and searched for a white space character, the **wctob** function can be used in conjunction with the **isspace** function.

Similarly, there are frequent occasions in wide-character processing code, especially in the wide character handling library functions, where it is necessary to test quickly and efficiently whether a single-byte character is the first and only character of a valid multibyte character. This is the reason for introducing the **btowc** function. Note that, for some encodings, **btowc** can be reduced to a simple inline expression.

MSE.13 Extended conversion utilities

Although C90 allows multibyte characters to have state-dependent encoding (§5.2.1.2), the original functions are not always sufficient to efficiently support such encodings due to the following limitations of the multibyte character conversion functions (§7.20.7):

1. Since the functions maintain shift state information internally, they cannot handle multiple strings at the same time.

2. The formatted output functions may write redundant shift sequences, and the formatted input functions cannot reliably parse input with arbitrary or redundant shift sequences.

3. The multibyte string conversion functions (§7.20.8) have an inconvenient shortcoming regardless of state dependency of the encoding: when an encoding error occurs, these functions return **(size_t)(–1)** without any information on the location where the conversion stopped.

For all these reasons, the Committee felt it necessary to augment the set of conversion functions in AMD1.

MSE.13.1 Conversion state

To handle multiple strings with a state-dependent encoding, the Committee introduced the concept of conversion state. The conversion state determines the behavior of a conversion between multibyte and wide-character encodings. For conversion from multibyte characters to wide characters, the conversion state stores information such as the position within the current multibyte character (as a sequence of characters or a wide character accumulator). For conversions in either direction, the conversion state stores the current shift state, if any, and possibly the encoding rule.

The non-array object type **mbstate_t** is defined to encode the conversion state. A zero-valued **mbstate_t** object is assumed to describe the initial conversion state. (This is not necessarily the *only* way to encode the initial conversion state, however.) Before any operations are performed on it, such a zero-valued **mbstate_t** object is *unbound*. Once a multibyte or wide-character conversion function executes with the **mbstate_t** object as an argument, however, the object

becomes *bound* and holds the above information.

The conversion functions maintain the conversion state in an **mbstate_t** object according to the encoding specified in the **LC_CTYPE** category of the current locale. Once the conversion starts, the functions will work *as if* the encoding scheme were not changed provided all three of the following conditions obtain:

- the function is applied to the same string as when the **mbstate_t** object was first bound.
- the **LC_CTYPE** category setting is the same as when the **mbstate_t** object was first bound.
- the conversion direction (multibyte to wide character, or wide character to multibyte) is the same as when the **mbstate_t** object was first bound.

MSE.13.2 Conversion utilities

Once the **mbstate_t** object was introduced, the Committee discussed the need for additional functions to manipulate such objects.

MSE.13.2.1 Initializing conversion states

Though a method to initialize the object is needed, the Committee decided that it would be better not to define too many functions in AMD1. Thus the Committee decided to specify only one way to make an **mbstate_t** object represent the initial conversion state, by initializing it with zero. No initializing function is supplied.

MSE.13.2.2 Comparing conversion states

The Committee reached the conclusion that it may be impossible to define the equality between two conversion states. If two **mbstate_t** objects have the same values for all attributes, they might be the same. However, they might also have different values and still represent the same conversion state. No comparison function is supplied.

MSE.13.2.3 Testing for initial shift state

The **mbsinit** function was added to test whether an **mbstate_t** object describes the initial conversion state or not, because this state does not always correspond to a certain set of component values (and the components cannot be portably compared anyway). The function is necessary because many functions in AMD1 treat the initial shift state as a special condition.

MSE.13.2.4 Restartable multibyte functions

Regarding problems 2 and 3 described at the beginning of §MSE.13, the Committee introduced a method to distinguish between an invalid sequence of bytes and a valid prefix to a still incomplete multibyte character. When encountering such an incomplete multibyte sequence, the **mbrlen** and

mbrtowc functions return **(size_t)(-2)** instead of **(size_t)(-1)**, and the character accumulator in the **mbstate_t** object stores the partial character information. Thus, the user can resume the pending conversion later, and can even convert a sequence one byte at a time.

The new multibyte/wide-string conversion utilities are thus made *restartable* by using the character accumulator and shift state information stored in an **mbstate_t** object. As part of this enhancement, the functions also have a parameter that is a pointer to a pointer to the source of the position where the conversion stopped.

MSE.14 Column width

The number of characters to be read or written can be specified in existing formatted input/output functions. On a traditional display device that displays characters with fixed pitch, the number of characters is directly proportional to the width occupied by these characters; so the display format can be specified through the field width and/or the precision.

In formatted wide character input/output functions, the field width and the precision specify the number of wide characters to be read or written. The number of wide characters is not always directly proportional to the width of their display. For example, with Japanese traditional display devices, a single-byte character such as an ASCII character has half the width of a Kanji character, even though each of them is treated as one wide character. To control the display format for wide characters, a set of formatted wide character input/output functions were proposed whose metric was the column width instead of the character count.

This proposal was supported only by Japan. Critics observed that the proposal was based on such traditional display devices with fixed-width characters, while many modern display devices support a broad assortment of proportional pitch type faces. Hence, it was questioned whether the extra input/output functions in this proposal were really needed or were sufficiently general. Also considered was another set of functions that return the column width for any kind of display device for a given wide character or wide-character string; but these seemed to be beyond the scope of C. Thus all proposals regarding column width were withdrawn.

If an implementor needs this kind of functionality, there are a few ways to extend wide character output functions and still remain conforming to AMD1. For example, the new conversion specifier can be used to specify the column width as shown below:

%#N — set the counting mode to “printing positions” and reset the **%n** counter.

%N — set the counting mode back to “wide characters” and reset the **%n** counter.

Index

Index