Rationale for International Standard - Programming Language - C

1. SCOPE

This Rationale summarizes the deliberations of X3J11, the Technical Committee charged by ANSI with devising a standard for the C programming language. It has been published along with the draft Standard to assist the process of formal public review.

The X3J11 Committee represents a cross-section of the C community: it consists of about fifty active members representing hardware manufacturers, vendors of compilers and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. In the course of its deliberations, it has reviewed related American and international standards both published and in progress. It has attempted to be responsive to the concerns of the broader community: as of September 1988, it had received and reviewed almost 200 letters, including dozens of formal comments from the first public review, suggesting modifications and additions to the various preliminary drafts of the Standard.

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

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

K&R is not the only source of "existing practice." Much work has been done over the years to improve the C language by addressing its weaknesses. The Committee has formalized enhancements of proven value which have become part of the various dialects of C.

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 has 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; these extensions have not been consistent across dialects.

One of the 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.

The work of the Committee was in large part a balancing act. The Committee has 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. 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 Committee used several guiding principles, the most important of which 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 Committee did not want to force most programmers to modify their C programs just to have them accepted by conforming translator.

On the other hand, no one implementation was held up as the exemplar by which to define C it is 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 Committee has 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 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 Committee has 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, the Rationale points out a QUIET CHANGE.

A standard is a treaty between implementor and programmer. Some numerical limits have been 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 are presented as *minimum maxima* (i.e., 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 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 Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases the Committee has 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 proved to be an important topic, it should have equal visibility with other topics. As a result, ala revision proposals submitted 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 C programs to run unchanged.

Minimize incompatibilities with C++. The committee recognizes the need for a clear and defensible plan with regard to how it intends to address 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.

Regarding our relationship with C++, 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 them. 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 will be 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 needed to explain the deviation.

- There has been a very positive reception of the standard from both the user and vendor communities.

- The standard is not considered to be broken. Rather, the revision is 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 shall look when revising the C Standard include:

- Incorporate Amendment 1.

- 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 JTC1/SC2 (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 being proposed by J11.

- Other comments from the public at large.
- Other prior art.

Without a set of acceptance criteria, judging any technical proposal becomes a highly subjective, and definitely emotional, exercise. It also wastes a lot of time and energy. Therefore, submittors are encouraged tokeep all the guiding principles in mind when making submissions.

Guidelines for the submission of proposals will be provided. Each submission shall contain a cover page containing responses to a number of questions and further summary information enabling the essence of a submission to be distilled simply by reading that cover. The information requested will include such things as: title, author, author affiliation, date, document number, abstract, proposal category (e.g., editorial, correction, new feature), prior art, and target audience. Proposals that are not directly linked must be submitted separately, each with their own document number and cover page.

Submissions **must** be sponsored in the same way as defect reports; that is, either by the convener of WG14, WG14 itself, or by a WG14 national member body. This provides a filtering process and allows submissions to be rejected early in the process if they violate the revision principles. It also allows substantially incomplete or disjoint proposals to be returned for further refinement.

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 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 the Base Documents.

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.

1.2 Organization of the document

This Rationale is organized to parallel the Standard as closely as possible, to facilitate finding relevant discussions. Some subsections of the Rationale comprise just the subsection title from the Standard: 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. The Appendices were added as a repository for related material not included in the Standard itself, or to bring together in a single place information about a topic which was scattered throughout the Standard.

Just as the Standard proper excludes all examples, footnotes, references, and appendices, *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.

2. NORMATIVE REFERENCES

3. DEFINITIONS and CONVENTIONS

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` (or `signed char` or `unsigned char`) 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 §6.2.2.1), 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 concept of *multi-byte character* has been added to C to support very large character sets. (See §5.2.1.2.)

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*. Appendix K to 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.

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 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.

4. COMPLIANCE

The three-fold definition of compliance 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 demeaning 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 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 a program to rule it conforming. The primary limitation on this license is §5.1.1.3.

Diverse sections 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 (§6.1.1). Otherwise, programmers can use whatever internal names they wish; a conforming implementation is guaranteed not to use conflicting names of the form reserved to the programmer. (Note, however, the class of identifiers which are identified in §7.20 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 16 characters mapped into one case (§6.1.2).

- 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 Committee was to mandate that each implementation have a translate-time switch for turning off extensions and making a pure Standard-conforming implementation. It was pointed out, however, that virtually every translate-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 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 Committee. However, the Committee has taken 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 preprocessor, for instance, is permitted to evaluate the expression in a `#if` statement using the 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. Other translate-time arithmetic, however, such as type casting and floating 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 Committee has 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 Test

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 existing C implementations can be found in preprocessing. Admittedly a distinct and primitive language superimposed upon C, the preprocessing commands accreted over time, with little central direction, and with even less precision in their documentation. This evolution has resulted in a variety of local features, each with its ardent adherents: the Base Document offers little clear basis for choosing one over the other.

The consensus of the 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 Committee deemed that it was outside the scope of its mandate to require the output of the preprocessing phases be available as a separate translator output file.

The *phases of translation* are spelled out to resolve the numerous 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 section on preprocessing (§6.8) discusses the reasons for many of the articular decisions which shaped the specification of the phases of translation.

A backslash immediately before a new-line 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 Committee generalized this mechanism to permit *any* token to be continued by interposing a backslash/ new-line 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 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 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, but such extensions must not invalidate strictly conforming programs. The translator must diagnose the use of such extensions, or allow them to be disabled as discussed in (Rationale) §4. Otherwise, extensions to a conforming C 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.

The behavior of the arguments to `main`, and of the interaction of `exit`, `main` and `atexit` (see §7.14.4.2) 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 arguments must ordinarily 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 Standard Annex K.5.1.)

Command line I/O redirection is not mandated by the Standard; 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 §6.3.) Most operators impose no sequencing requirements, but a few operators impose *sequence points* upon their evaluation: comma, logical-AND, logical-OR, and conditional. For example, in the expression `(i = 1, a[i] = 0)` 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 §6.6). 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 (hypothetical) 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 re-evaluated 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 pre-computed in the real implementation. A question sometimes asked regarding optimization is, "Is the rearrangement still conforming if the pre-computed 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* (§6.3), 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.

Appendix 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 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 ¥ or 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 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 new-line character, regardless of its external representation.

5.2.1.1 Trigraph sequences

Trigraph sequences have been introduced 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.

The 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 extra 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.

The International Standards Organization (ISO) 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 96 characters), 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 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. The ISO has defined such a standard: ISO 646 describes an *invariant subset* of ASCII.

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

```
# [ ] { } \ | ~ ^
```

Given this repertoire, the 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 646 repertoire: 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 currently occur anywhere in a legal C program except in strings, character constants, comments, or header names. The character escape sequence '`\?`' (see §6.1.3.4) 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

> Programs with character sequences such as `??!` in string constants, character constants, or header names will now produce different results.

5.2.1.2 Multibyte characters

The "byte = character" orientation of C works well for text in Western alphabets, where the size of 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 integral 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 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, and in source text string and character literals.

- 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 transparently to work 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.

- The minimum number of absolutely necessary library functions is introduced. (See §7.14.7.)

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 modelled 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*. ANSI X3L2 (Codes and 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. Both ANSI X3L2 and ISO 646 deprecate the combination of the motion to the next line with a motion to the initial position on the line. The C Standard, on the other hand, 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'`) has been 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 such as EBCDIC have no obvious equivalent. (See §6.1.3.4.)

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 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` (§7.10.2.1) and `signal` (§7.11.1.1), 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 §7.10.2.1. 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 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 compile 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 Committee felt that such ingenuity would probably require more work than making something useful. The sense of the Committee is 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.

The 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.

5.2.4.2 Numerical limits

In addition to the discussion below, see §7.1.5.

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 ones-complement or sign-magnitude implementations, as well as the more usual twos-complement. The limits for the maxima and minima of unsigned types are specified as unsigned constants (e.g., `65535u`) to avoid surprising widenings of expressions involving these extrema.

The macro `CHAR_BIT` makes available the number of bits in a `char` object. The 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 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 (X3J3[1]). The 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 twos-complement, but precludes a logarithmic implementation.

Single precision (32-bit) floating point is considered adequate to support a conforming C implementation. Thus the minimum maxima constraining floating types are extremely permissive.

The Committee has also endeavored to accommodate the IEEE 754 floating point standard by not adopting any constraints on floating point which are contrary to this standard.

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

[1] See X3J3 working document S8-112.

6. LANGUAGE

While more formal methods of language definition were explored, the Committee decided early on to employ the style of the Base Document: 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.

6.1 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 §6.8.3).

6.1.1 Keywords

Several keywords have been added: `const`, `enum`, `signed`, `void`, and `volatile`. New for ***C9X*** are the keywords `restrict`, `inline`, `complex` and `imaginary`.

As much as 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*.

6.1.2 Identifiers

While an implementation is not obliged to remember more than the first 31 characters of an identifier for the purpose of name matching, the programmer is effectively prohibited from intentionally creating two different identifiers that are the same in the first 31 characters. Implementations may therefore store the full identifier; they are not obliged to truncate to 31.

The decision to extend significance to 31 characters for internal names was made with little opposition, but the decision to retain the old six-character case-insensitive restriction on significance of external names was most painful. While strong sentiment was expressed for making C "right" by requiring longer names everywhere, the Committee recognized that the language must, for years to come, coexist with other languages and with older assemblers and linkers. Rather than undermine support for the Standard, the severe restrictions have been retained.

The Committee has decided to label as *obsolescent* the practice of providing different identifier significance for internal and external identifiers, thereby signaling its intent that some future version of the C Standard require 31-character case-sensitive external name significance, and thereby encouraging new implementations to support such significance.

Three solutions to the external identifier length/case problem were explored, each with its own set of problems:

1. *Label any C implementation without at least 31-character, case-sensitive significance in external identifiers as non-standard.* This is unacceptable since the whole reason for a standard is portability, and many systems today simply do not provide such a name space.

2. *Require a C implementation which cannot provide 31-character, case-sensitive significance to map long identifiers into the identifier name space that it can provide.* This option quickly becomes very complex for large, multi-source programs, since a program-wide database has to be maintained for all modules to avoid giving two different identifiers the same actual external name. It also reduces the usefulness of source code debuggers and cross reference programs, which generally work with the short mapped names, since the source-code name used by the programmer would likely bear little resemblance to the name actually generated.

3. *Require a C implementation which cannot provide 31-character, case-sensitive significance to rewrite the linker, assembler, debugger, any other language translators which use the linker, etc.* This is not always practical, since the C implementor might not be providing the linker, etc. Indeed, on some systems only the manufacturer's linker can be used, either because the format of the resulting program file is not documented, or because the ability to create program files is restricted to secure programs.

Because of the decision to restrict significance of external identifiers to six case-insensitive characters, C programmers are faced with these choices when writing portable programs:

1. Make sure that external identifiers are unique within the first six characters, and use only one case within the name. A unique six-character prefix could be used, followed by an underscore, followed by a longer, more descriptive name:

```
extern int a_xvz_real_long_name;
extern int a_rwt_real_long_name2;
```

2. Use the prefix method described above, and then use `#define` statements to provide a longer, more descriptive name for the unique name, such as:

```
#define real_long_name  a_xvz_real_long_name
#define real_long_name2 a_rwt_real_long_name2
```

Note that overuse of this technique might result in exceeding the limit on the number of allowed `#define` macros, or some other implementation limit.

3. Use longer and/or multi-case external names, and limit the portability of the programs to systems that support the longer names.

4. Declare all exported items (or pointers thereto) in a single data structure and export that structure. The technique can reduce the number of external identifiers to one per translation unit; member names within the structure are internal identifiers, hence can have full

significance. The principal drawback of this technique is that functions can only be exported by reference, not by name; on many systems this entails a run-time overhead on each function call.

QUIET CHANGE

A program that depends upon internal identifiers matching only in the first (say) eight characters may change to one with distinct objects for each variant spelling of the identifier.

6.1.2.1 Scopes of identifiers

The Standard has separated from the overloaded keywords for storage classes the various concepts of *scope*, *linkage*, *name space*, and *storage duration*. (See §6.1.2.2, §6.1.2.3, §6.1.2.4.) 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. The Base Document is vague on this point, and has been interpreted differently by different 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

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 of course 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.

6.1.2.2 Linkages of identifiers

The Standard requires that the first declaration, implicit or explicit, of an identifier 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 than 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.

Existing practice in this area is inconsistent. Some implementations have avoided the renaming problem simply by restricting internal linkage names by the same rules as 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 the Standard may call for changes in some existing programs, but allow for maximum flexibility.

The definition model to be used for objects with external linkage was a major 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. Existing implementations of C exhibit at least four different models, listed here in order of increasing restrictiveness:

Common Every object declaration with external linkage (whether or not 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` (whether it is used outside of the scope of a function or not) in a declaration 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 within the code. The UNIX operating system C compiler and linker implement this model, which is recognized as a *common extension* to the C language (K.5.11). UNIX C programs which take advantage of this model are standard conforming in their environment, but are not maximally portable.

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 and in the Base Document.

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 model adopted in the Standard 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 - 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 §6.7), serves as the external definition. This composite approach was chosen to accommodate as wide a range of environments and existing implementations as possible.

6.1.2.3 Name spaces of identifiers

Implementations have 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.

Figure 6.1: Comparison of identifier linkage models

<table>
<tr><th>Model</th><th>File 1</th><th>File 2</th></tr>
<tr><td>common</td><td><pre>extern int i:
main( ) {
       i = 1;
       second( );
}</pre></td><td><pre>extern int i;
second( ) {
         third(i);
}</pre></td></tr>
<tr><td>Relaxed Ref/Def</td><td><pre>int i;
main( ) {
       i = 1;
       second( );
}</pre></td><td><pre>int i;
second( ) {
         third(i);
}</pre></td></tr>
<tr><td>Strict Ref/Def</td><td><pre>int i;
main( ) {
       i = 1;
       second( );
}</pre></td><td><pre>extern int i;
second( ) {
         third(i);
}</pre></td></tr>
<tr><td>Initializer</td><td><pre>int i = 0;
main( ) {
       i = 1;
       second( );
}</pre></td><td><pre>int i;
second( ) {
         third(i);
}</pre></td></tr>
</table>

6.1.2.4 Storage durations of objects

It was necessary to clarify the effect on automatic storage of jumping into a block that declares local storage. (See §6.6.2.) While many implementations allocate the maximum depth of automatic storage upon entry to a function, some explicitly allocate and deallocate on block entry and exit. The latter are required to assure that local storage is allocated regardless of the path into the block (although initializers in automatic declarations are not executed unless the block is entered from the top).

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. This means that the caller must allocate automatic storage for the return value and communicate its location to the called function. (The typical case of return registers for small types conforms to this requirement: the calling convention of the implementation implicitly communicates the return location to the called function.)

6.1.2.5 Types

Several new types have been added:

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

New types added for C9X.

```
float complex
double complex
long double complex
long long
```

New designations for existing types have been added:

`signed short`	for	`short`
`signed int`	for	`int`
`signed long`	for	`long`

`void` is used primarily as the typemark for a function which returns no result. It may also be used, in any context where the value of an expression is to be discarded, to indicate explicitly that a value is ignored by writing the cast `(void)`. 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.

A "pointer to void," `void *`, is a generic pointer, capable of pointing to any (data) object except for bit-fields and objects declared with the `register` storage-class without truncation. A pointer to void must have the same representation and alignment as a pointer to character; the intent of this rule is to allow existing programs which call library functions (such as `memcpy` and `free`) to continue to work. A pointer to void may not be dereferenced, although such a pointer may be converted to a normal pointer type which may 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 to make available a one-byte signed integer type on those systems which implement plain `char` as unsigned. For reasons of symmetry, the keyword `signed` is allowed as part of the type name of other integral types. Two varieties of the integral types are specified: `signed` and `unsigned`. If neither specifier is used, signed is assumed. In the Base Document the only unsigned type 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 (in principle) be undefined. In practice, on twos-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 (see §6.2.1.1).

The Committee has explicitly restricted the C language to binary architectures, 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 "illegal 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 (see §6.3.3.3, §6.3.7, §6.3.10, §6.3.11, §6.3.12).

A new floating type `long double` has been added to C. The `long double` type must offer at least as much precision as the type `double`. Several architectures support more than two floating types and thus can map a distinct machine type onto this additional C type. Several architectures which only support two floating point types can also take advantage of the three C types by mapping the less precise type onto `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 floats offer at least as much precision as most other machines's double-precision, or those on which single-precision 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.

To avoid confusion, `long float` as a synonym for `double` has been retired.

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

The 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, for example, in Pascal.

The 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 instance, with arrays indexed by enumerations.

6.1.2.6 Compatible type and composite type

The notions of *compatible types* and *composite type* have been introduced 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 complete 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 include file, since the translation units are themselves disjoint. The Standard thus specifies additional compatibility rules for such types, so that if two such declarations are sufficiently similar they are compatible.

6.1.2.7 Predefined identifiers

6.1.2.8 Representations of types

6.1.2.8.1 General

6.1.2.8.2 Integer types

6.1.3 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 Committee considered the introduction of structure constants. Although it agreed that structure literals would occasionally be useful, its policy has been not to invent new features unless a strong need exists. Since the language already allows for initialized `const` structure objects, the need for inline anonymous structured constants seems less than pressing.

Several implementation difficulties beset structure constants. All other forms of constants are "self typing" - the type of the constant is evident from its lexical structure. Structure constants would require either an explicit type mark, or typing by context; either approach is considered to require increased complexity in the design of the translator, and either approach would also require as much, if not more, care on the part of the programmer as using an initialized structure object.

6.1.3.1 Floating constants

Consistent with existing practice, a floating point constant has been defined to have type `double`. Since the Standard now allows expressions that contain only `float` operands to be performed in `float` arithmetic (see §6.2.1.7) rather than `double`, a method of expressing explicit `float` constants is desirable. The new `long double` type raises similar issues.

Thus the `F` and `L` suffixes have been added to convey type information with floating constants, much like the `L` suffix 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` (§7.5.1) has no effect on the syntax of C source text: the decimal point character is always period.

6.1.3.2 Integer constants

The 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 suffixes `U` and `u` have been added to specify unsigned numbers.

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 has been dropped.

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 an integral (or floating) literal, the 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 Committee's deliberations on integral promotion rules (see §6.2.1.1).

QUIET CHANGE

> 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`.

6.1.3.3 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`. (See §6.1.2.5.)

6.1.3.4 Character constants

The digits 8 and 9 are no longer permitted in octal escape sequences. (Cf. octal constants, §6.1.3.2.)

The alert escape sequence has been added (see §5.2.2).

Hexadecimal escape sequences, beginning with `\x`, have been adopted, with precedent in several existing implementations. (Little sentiment was garnered 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. 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.

The Committee has chosen to reserve all lower case letters not currently used for future escape sequences (*undefined behavior*). 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 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 ASCII/X3.64 environment, may, instead of writing

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

write:

```
#define  ESC  "\033"

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

Notwithstanding the general rule that literal constants are non-negative [2], 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

> 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 $078 \equiv 0100 \equiv 64$.

QUIET CHANGE

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

An `L` prefix distinguishes wide character constants. (See §5.2.1.2.)

6.1.4 String literals

String literals are specified to be unmodifiable. This specification allows implementations to share copies of strings with identical text, to place string literals in read-only memory, and perform certain

[2] is an expression: unary minus with operand 3.

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 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 the literal */
     /* with a unique name */
```

can be changed to:

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

A long string can be continued across multiple lines by using the backslash-newline line continuation, but this practice 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 §6.8.3), the 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.

without concatenation:

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

with concatenation:

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

QUIET CHANGE

A string of the form "`078`" is valid, but now has different meaning. (See §6.1.3.)

QUIET CHANGE

A string of the form "`\a`" or "`\x`" now has different meaning. (See §6.1.3.)

QUIET CHANGE

> 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 (as opposed to suffix) notation was adopted so that a translator can know at the start of the processing of a long string literal whether it is dealing with ordinary or wide characters. (See §5.2.1.2.)

6.1.5 Operators

Assignment operators of the form `=+`, described as *old fashioned* even in K&R, have been dropped.

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

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

The operator `#` has been added in preprocessing statements: within a `#define` it causes the macro argument following to be converted to a string literal.

The operator `##` has also been added in preprocessing statements: within a `#define` it causes the tokens on either side to be *pasted* to make a single new token. See §6.8.3 for further discussion of these preprocessing operators.

6.1.6 Punctuators

The punctuator `...` (ellipsis) has been added to denote a variable number of trailing arguments in a function prototype. (See §6.5.5.3.)

The constraint that certain punctuators must occur in pairs (and the similar constraint on certain operators in §6.1.5) only applies after preprocessing. Syntactic constraints are checked during syntactic analysis, and this follows preprocessing.

6.1.7 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.

6.1.8 Preprocessing numbers

The notion of preprocessing numbers has been 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 - `0x123E+1` is a single token under the rules. The 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.

6.1.9 Comments

The 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 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
/* this code is bracketed out because ... */
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) {
      /* this code is bracketed out because ... */
      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/compilers[3].

QUIET CHANGE

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

3 The C++ programming language supports // comments.

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

but in C9X it is equivalent to

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

6.2 Conversions

6.2.1 Arithmetic operands

6.2.1.1 Characters and integers

Since the publication of K&R, a serious divergence has occurred among implementations of C in the evolution of integral promotion rules. Implementations fall into two major camps, which may be characterized as *unsigned preserving* and *value preserving*. The difference between these approaches centers on the treatment of `unsigned char` and `unsigned short`, when widened by the *integral promotions*, but the decision has an impact on the typing of constants as well (see §6.1.3.2).

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 twos-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: i.e., 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 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

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

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

6.2.1.2 Signed and unsigned integers

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

6.2.1.3 Real floating and integer

There was strong agreement that floating values should truncate toward zero when converted to an integral type, the specification adopted in the Standard. Although the Base Document permitted negative floating values to truncate away from zero, no Committee member knew of current hardware that functions in such a manner.[4]

6.2.1.4 Real floating types

The Standard, unlike the Base Document, does not require rounding in the `double` to `float` conversion. Some widely used IEEE floating point processor chips control floating to integral 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

[4] We have since been informed of one such implementation.

require such implementations to round to `float`.

6.2.1.5 Complex types

6.2.1.6 Real and complex

6.2.1.7 Usual arithmetic conversions

The rules in the Standard for these conversions are slight modifications of those in the Base Document: the modifications accommodate the added types and the value preserving rules (see §6.2.1.1). Explicit license has been 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 exactly the intermediate types required.*

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

QUIET CHANGE

> Expressions with `float` operands may now be computed at lower precision. The Base Document specified that all floating point operations be done in `double`.

6.2.2 Other operands

6.2.2.1 Lvalues and function designators

A difference of opinion within the C community has 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 Committee has 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 itself a modifiable lvalue while `a` is not has puzzled many students of the language. A more precise description has therefore been incorporated in the Standard, in the hopes of combatting this confusion.

6.2.2.2 `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, hence requires no storage.

6.2.2.3 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 may not assume any necessary correspondence between different pointer types and the integral types.

The use of `void *` ("pointer to void") as a generic object pointer type is an invention of the 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 that can be safely converted to a pointer is the constant 0. The result of converting any other integer to a pointer is machine dependent.

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. 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.

6.3 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 latter three terms are discussed in §5.1.2.3.

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.

The Base Document describes C as a language in which the operands of successive identical commutative associative operators can be regrouped. The Committee has 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) is not associative.

The Base Document'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. The Base Document'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.

Elimination 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 twos-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: 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.

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 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 Committee has recognized certain prevalent exceptions:

- The lvalue types may differ in signedness. In the common range, a signed integral type and its unsigned variant have the same representation; 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 = {2.0, 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.

6.3.1 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.

6.3.1.1 The identifier `__func__`

A new feature for ***C9X***

6.3.2 Postfix operators

6.3.2.1 Array subscripting

The 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.

6.3.2.2 Function calls

Pointers to functions may be used either as `(*pf)()` or as `pf()`. The latter construct, not sanctioned in the Base Document, 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 Committee saw no real harm in allowing these forms; outlawing forms like `(*f)()`, while still permitting `*a` (for `int a[]`), simply seemed more trouble than it was worth.

The rule for implicit declaration of functions has been removed. The effect is to guarantee that a diagnostic is produced, which 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 the Base Document 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 be made in the presence of a compatible prototype.

To clarify this and other relationships between function calls and function definitions, the Standard describes an equivalence 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 the explicit function definition prototype. This constraint is necessary to retain compatibility with past usage of library functions. (See §7.1.8.)

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 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 type 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 an inequivalence 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*).

6.3.2.3 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 or by a conditional operator expression:

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

In these cases, 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 may 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. The Base Document is unclear on this point.

6.3.2.4 Postfix increment and decrement operators

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

6.3.2.5 Compound literals

A new feature for ***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 (see §6.5.8) 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.

6.3.3 Unary operators

6.3.3.1 Prefix increment and decrement operators See §6.3.2.4.

6.3.3.2 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 Committee has 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.

6.3.3.3 Unary arithmetic operators

Unary plus was adopted by the 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 integral representations to "binary numeration systems." (See §6.1.2.8.2.)

6.3.3.4 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 comprised of an integral number of these smallest units. (See §1.6.)

The Standard, like the Base Document, defines the result of the `sizeof` operator to be a constant of an unsigned integral type. Common implementations, and common usage, have often presumed 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 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, thus quashing any notion that the largest declarable object might be too big to span even with an `unsigned long`. 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.

(See §7.13.8.1, §7.13.8.2, §7.14.3.1, etc.)

The Standard specifies that the argument to `sizeof` can be any value except a bit field, a void expression, or a function designator. This generality allows for interesting environmental enquiries; 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 of course available as `ptrdiff_t` in `<stddef.h>`.)

6.3.4 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 the Base Document, but with several significant differences. The Base Document 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 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 the integer constant 0 to a pointer is defined similarly to the Base Document. 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 (§6.5.3) 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*.

The Standard (§6.2.1.4) requires that a cast of one floating point type to another (e.g., `double` to `float`) results in an actual conversion.

6.3.5 Multiplicative operators

There was considerable sentiment for giving more portable semantics to division (and hence remainder) by specifying some way of giving less machine dependent results for negative operands. Few Committee members wanted to require this by default, lest existing fast code be gravely slowed. One suggestion was to make `signed int` a type distinct from plain `int`, and require better-defined semantics for `signed int` division and remainder. This suggestion was opposed on the grounds that effectively adding several types would have consequences out of proportion to the benefit to be obtained; the Committee twice rejected this approach. Instead the Committee has adopted new library functions `div` and `ldiv` which produce integral quotient and remainder with well-defined sign semantics. (See §7.14.6.2, §7.14.6.5.)

The Committee rejected extending the `%` operator to work on floating types; such usage would duplicate the facility provided by `fmod`. (See §7.7.10.1)

6.3.6 Additive operators

As with the `sizeof` operator, implementations have taken different approaches in defining a type for the difference between two pointers (see §6.3.3.4). 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 may cause an overflow.)

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 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 may fail. This restriction allows segmented architectures, for instance, to place objects at the start of a range of addressable memory.

6.3.7 Bitwise shift operators

See §6.3.3.3 for a discussion of the arithmetic definition of these 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 Committee, is that the type of the shift count has no bearing on the type of the result.

QUIET CHANGE

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

The Committee has affirmed the freedom in implementation granted by the Base Document 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 twos-complement integer arithmetically right one place is *not* the same as dividing by two!)

6.3.8 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 §6.3.6.

6.3.9 Equality operators

The 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.) Otherwise 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.

6.3.10 Bitwise AND operator

See §6.3.3.3 for a discussion of the arithmetic definition of the bitwise operators.

6.3.11 Bitwise exclusive OR operator

See §6.3.3.3.

6.3.12 Bitwise inclusive OR operator

See §6.3.3.3.

6.3.13 Logical AND operator

6.3.14 Logical OR operator

6.3.15 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 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.

6.3.16 Assignment operators

Certain syntactic forms of assignment operators have been discontinued, and others tightened up (see §6.1.5).

The storage assignment need not take place until the next sequence point. (A restriction in earlier drafts that the storage take place before the value of the expression is used has been removed.) 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 is 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 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`. The storage expressions, and related 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.

6.3.16.1 Simple assignment

Structure assignment has been 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 argument 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.

6.3.16.2 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.

6.3.17 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.

The example in the Standard clarifies that commas separating arguments "bind" tighter than the comma operator in expressions.

6.4 Constant Expressions

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

The expression following `#if` (§6.8.1) must expand to integer constants, character constants, the special operator `defined`, and operators with no side effects. No environmental inquiries can be made, since all arithmetic is done as translate-time (signed or unsigned) long integers, and casts are disallowed. The restriction to translate-time arithmetic frees an implementation from having to perform execution-environment arithmetic in the host environment. It does not preclude an implementation from doing so - the implementation may simply define "translate-time arithmetic" to be that of the target. Unsigned arithmetic is performed in these expressions (according to the default widening rules) when unsigned operands are involved; this rule allows for unsurprising arithmetic involving very large constants (i.e, those whose type is `unsigned long`) since they cannot be represented as `long` or constants explicitly marked as unsigned.

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 *integral constant expression* must involve only numbers knowable at translate time, and operators with no side effects. Casts and the `sizeof` operator may be used to interrogate the execution environment.

Static initializers include integral constant expressions, along with floating constants and simple addressing expressions. An implementation must accept arbitrary expressions involving floating and integral 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 (see §5.1.2.2); 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 than 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

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

6.5 Declarations

The Committee decided that empty declarations are invalid (except for a special case with tags, see §6.5.2.3, and the case of enumerations such as `enum {zero,one};`, see §6.5.2.2). 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.

The practice of placing the storage class specifier other than first in a declaration has been branded as *obsolescent* (See §6.9.2.) The Committee feels it desirable to rule out such constructs as

```
enum { aaa, aab,
     /* etc */
zzy, zzz } typedef a2z;
```

in some future standard.

6.5.1 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.

6.5.2 Type specifiers

Several new type specifiers have been added: `signed`, `enum`, and `void`. `long float` has been retired and `long double` has been added, along with a plethora of integer types. The Committee's reasons for each of these additions, and the one deletion, are given in section §6.1.2.5 of this document.

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 grow 2X every 3 years, but after 1989, has accelerated to 4X / 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 ints, which were rapidly obsoleted by disk progress. People switched to a 32-bit file system, first using int[2] constructs that were not only awkward, but also not efficiently portable to 32-bit hardware.

To solve the problem, `long` was added to the language, even though this required PDP-11 C 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 usedfor 32-bits as needed.

PDP-11 C might have been re-implemented with `int` as 32-bits, thus avoiding the need for `long`, but of course, making 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 impacton existing object code and data files would have been worse, even in1976. 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 have added a 64-bit integer to their 32-bit C environments, using a new name, of which `long long` has been the widestused, to avoid namespace conflicts. 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 longer integers happens irrespective of other CPUs. Thus, 32-bit C has evolved from a common `ILP32` model (integer, `long`, pointer 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 grow at 4X every 3 years, and 64-bit microprocessors are becoming widely used, starting in 1992. By 1995, refrigerator-sized, microprocessor-based servers were being sold with 8GB-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`s and pointers, and `long longs` are 64 bits, `ints` are 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 `ILP32`, 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-bit, 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 either to `ILP32LL` or to `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 reasons. 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, but 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.

`ILP64`, 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 has become 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, i.e., 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 longer, 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.

*A new feature for **C9X.*** In C90, 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 such a feature outweighed its convenience, and thus this feature was removed. The effect is to guarantee that a diagnostic is produced, which will catch an additional category of programming errors. Implementations may also choose to assume an implicit `int` and continue to translate the program, in order to support existing source code that exploited this feature.

6.5.2.1 Structure and union specifiers

Three types of `bit fields` are now defined: "plain" `int` calls for *implementation-defined* signedness (as in the Base Document), `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; and
- 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 field by field.

To clarify what is meant by the notion that "all the fields of 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).

6.5.2.2 Enumeration specifiers

6.5.2.3 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 has been given to the form:

```
struct y;
```

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

QUIET CHANGE

The empty declaration `struct x;` is no longer innocuous.

6.5.3 Type qualifiers

The Committee has added to C three *type qualifiers*: `const`, `volatile` and `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` is an invention of the Committee; it follows 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.)

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 a `nonconst` keyword instead of `const`, or `nonvolatile` instead of `volatile`. The senses of these concepts in the Standard were chosen to assure that the default, unqualified, case was the most common, and that it corresponded most clearly to traditional practice in the use of lvalue expressions.

Four combinations of the two qualifiers is possible; each defines a useful set of lvalue properties. The next several paragraphs describe typical uses of these qualifiers.

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. If a function parameter is declared `const`, then the referenced object is not changed (through that lvalue) in the body of the function - the parameter is read-only.

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; 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 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 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.

6.5.3.1 Formal definition of `restrict`

*A new feature for **C9X***. The `restrict` type qualifier allows programs to be written so that (at least some) 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 (i.e., 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` type 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 at 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, at the beginning of each execution of the block, the sole 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 `f3()`.

```
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 it 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*.

6.5.4 Function specifiers

6.5.5 Declarators

The function prototype syntax was adapted from C++. (See §6.3.2.2 and §6.5.5.3)

Some current implementations have a limit of six type modifiers (*function returning*, *array of*, *pointer to*, the limit used in Ritchie's original compiler. This limit has been raised to twelve 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.

6.5.5.1 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++, from which these constructs were adopted. (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.

6.5.5.2 Array declarators

The concept of *composite types* (§6.1.2.6) 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.

The declaration of 0-length arrays is invalid, under the general principle of not providing for 0-length objects. The only common use of this construct has been in the declaration of dynamically allocated variable-size arrays, such as

```
struct segment {
    short int   count;
    char    c[N];
};
```

```
struct segment * new_segment( const int length )
{
    struct segment * result;
    result = malloc( sizeof segment + (length-N) );
    result->count = length;
    return result;
}
```

In such usage, `N` would be `0` and `(length-N)` would be written as `length`. But this paradigm works just as well, as written, if `N` is `1`. (Note, by the by, an alternate way of specifying the size of `result`:

```
    result = malloc(offsetof(struct segment,c) + length);
```

This illustrates one of the uses of the `offsetof` macro.)

6.5.5.3 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 either occurred at runtime, or through the use of auxiliary software tools.

In function calls not in the scope of a function prototype, integral arguments have the *integral widening conversions* applied and `float` arguments are widened to `double`. It is thus impossible 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, and

- 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.

Optimizers can also take advantage of function prototype information. Consider this example:

```
extern int compare(const char * string1,
```

```
                        const char * string2) ;

        void func2(int x)
        {
                char * str1, * str2 ;
                    /* ... */
                x = compare(str1, str2) ;
                    /* ... */
        }
```

The optimizer knows that the pointers passed to `compare` are not used to assign new values of any objects that the pointers reference. Hence the optimizer can make less conservative assumptions about the side effects of `compare` than would otherwise be necessary. 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 extant 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);
        };
```

The structure `d_funct` has two fields, both of which hold pointers to functions taking two double arguments; the function types differ in their return type.

6.5.6 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.

6.5.7 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 implementations of C have 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 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 §6.5.2.

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

6.5.8 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 Committee considered proposals for permitting automatic aggregate initializers to consist of a brace-enclosed series of arbitrary (execute-time) expressions, instead of just those usable for a translate-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 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.)

The Base Document 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 the Base Document. Although the construct is allowed, and its parse well defined, the Committee urges programmers to avoid partially elided initializers: such initializations can be quite confusing to read.

QUIET CHANGE

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

The 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 can now be said of unions, with the significant difference that only one union member (the first) can be initialized.
*A new feature for **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.

6.6 Statements

6.6.1 Labeled statements

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

6.6.2 Compound statement, or block

The 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 (see §6.1.2.4). The Committee rejected such a ban out of fear of invalidating working code (however undisciplined) and out of concern for those producing machine-generated C.

6.6.3 Expression and null statements

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

6.6.4 Selection statements

6.6.4.1 The `if` statement

See §6.6.2.

6.6.4.2 The `switch` statement

The controlling expression of a `switch` statement may now have any integral type, even `unsigned 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 `A` and that for `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 the switch more structured, as in Pascal, out of fear of invalidating working code.

QUIET CHANGE

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

6.6.5 Iteration statements

6.6.5.1 The `while` statement

6.6.5.2 The `do` statement

6.6.5.3 The `for` statement

6.6.6. Jump statements

6.6.6.1 The `goto` statement

See §6.6.2.

6.6.6.2 The `continue` statement

The 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.

6.6.6.3 The `break` statement
See §6.6.6.2.

6.6.6.4 The `return` statement

6.7 External definitions

6.7.1 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). (See also §6.1.2.2.)

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 sizeable 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 §6.5.5.3). 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 a (for instance) `char` parameter as if it were declared `int`, since the argument is known to be passed as an `int` (in the absence of prototypes). 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

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

Notes for implementors: the assignment conversion for argument passing often requires no executable code. In most twos-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 Committee decided, therefore, that arguments are effectively declared in the outermost block, and hence cannot be quietly redeclared in that block.

The Committee considered it important that a function taking a variable number of arguments, such as `printf`, be expressible portably in C. Hence, the Committee devoted much time to exploring methods of traversing variable argument lists. One proposal was to require arguments to be passed as a "brick" (i.e., 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 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 §7.12).

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 that a diagnostic is produced, which 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.

6.7.2 External object definitions

See §6.1.2.2.

6.8 Preprocessing directives

For an overview of the philosophy behind the preprocessor, see §5.1.1.2.

Different implementations have had different notions about whether white space is permissible before and/or after the `#` signalling a preprocessor line. The 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.

6.8.1 Conditional inclusion

For a discussion of evaluation of expressions following `#if`, see §6.4.

The operator `defined` has been added 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 (§6.8.8).

`#elif` has been 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` statement, even though if it *were* processed, a syntactic 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 easier matchup with their corresponding `#if` directives. The 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.

6.8.2 Source file inclusion

Specification of the `#include` directive raises distinctive grammatical problems because the file name is conventionally parsed quite differently than 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, if necessary, map a token sequence (back) into a sequence of characters.

- 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`) directive, 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 an include statement found within an include 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 Committee decided, in principle, in favor of the 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 include files.

Clause §5.2.4.1 on translation limits contains the required number of nesting levels for include 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 include 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.

6.8.3 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 as a separate (text-to-text) pre-pass or as a (token-oriented) portion of the compiler itself. Thus, the preprocessing grammar is specified in terms of tokens.

However, the new-line 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 the preservation of each white space character (which is easy for a text-to-text pre-pass) or 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 0
```

(in header2.h)

```
#define NULL_DEV 0
```

The 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.

Existing implementations have differed on whether keywords can be redefined by macro definitions.

The Committee has decided to allow this usage; it sees such redefinition as useful during the transition from existing to Standard-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 a signed-char implementation to one in which `char` is unsigned. The second case might be useful in adapting code which assumes that `sizeof` results in 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 for **C9X***. Function-like macro invocations may now have empty arguments, that is, an argument may consist of no preprocessing tokens. In C90, any argument that consisted of no preprocessing tokens had undefined behavior, but as mentioned in C90 Annex G.5.12, it was 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 is, 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.

6.8.3.1 Argument substitution

6.8.3.2 The # operator

Some implementations have decided to replace identifiers found within a string literal if they match a macro argument name. The replacement text is a "stringized" form of the actual argument token sequence. This practice appears to be contrary to the definition, in K&R, of preprocessing in terms of token sequences. The 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 Committee introduced a more tractable mechanism of comparable power.

The `#` operator has been introduced for stringizing. It may only be used in a `#define` expansion. 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 §6.1.4), 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 worth (over 500 characters) may have to be retained. As a compromise between token-based and character-based preprocessing disciplines, the 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, which might, for instance, convert a numeric literal to a canonical or internal representation, losing information about base, leading 0's, etc.) In the interest of simplicity, the Committee decided that each token should expand to just those characters used to specify it in the original source text.

QUIET CHANGE

> A macro that relies on formal parameter substitution within a string literal will produce different results.

6.8.3.3 The ## operator

Another facility relied on in much current practice but not specified in the Base Document is "token pasting," or building a new token by macro argument substitution. One existing implementation is to replace a comment within a macro expansion by zero characters, instead of the single space called for in K&R. The Committee considered this practice unacceptable.

As with "stringizing," the facility was considered desirable, but not the extant implementation of this facility, so the 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 is substituted for the formal, but the actual is not expanded:

```
#define a(n) aaa ## n
#define b    2
```

Given these definitions, 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.

6.8.3.4 Rescanning and further replacement

A problem faced by most current preprocessors is how to use a macro name in its expansion without suffering "recursive death." The 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 clearly was introduced during the expansion of the original `f`, so is not further expanded.

However, given the definitions

```
#define f(a)  a*g
#define g(a)  f(a)
```

the expansion rules allow the result to be either `2*f(9)` or `2*9*g` - it is unclear 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 Committee intentionally left this behavior ambiguous: it saw no useful purpose in specifying all the quirks of preprocessing for such questionably useful constructs.

6.8.3.5 Scope of macro definitions

Some pre-Standard implementations maintain a stack of `#define` instances for each identifier; `#undef` simply pops the stack. The 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 §7.1.8).

6.8.4 Line control

Aside from giving values to `__LINE__` and `__FILE__` (see §6.8.8), 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 recongnized as such inside of macros.

6.8.5 Error directive

The directive `#error` has been introduced to provide an explicit mechanism for forcing translation to fail under certain conditions. (Formally the Standard only requires, *can* only require, that a diagnostic be issued when the `#error` directive is effected. 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.) Traditionally such failure has had to be forced by inserting text so ill-formed that the translator gagged on it.

6.8.6 Pragma directive

The `#pragma` directive has been added as the universal method for extending the space of directives.

*A new feature for **C9X.*** Some directives have been standardized for C9X. Directives whose first preprocessing token is `STDC` are reserved for standardized directives.

6.8.7 Null directive

The existing practice of using empty # lines for spacing is supported in the Standard.

6.8.8 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; it recognizes that these macros have special built-in properties.

The macros `__DATE__` and `__TIME__` have been added to make available the time of translation. A particular format for the expansion of these macros has been specified to aid in parsing strings initialized by them.

The macros `__LINE__` and `__FILE__` have been added 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 or not. It is defined as having value 1; future versions of the Standard could define it as 2, 3, ..., to allow for conditional compilation on which version of the Standard a translator conforms to. This macro should be of use in the transition toward conformance to the Standard.

A new feature for ***C9X.*** The macros `__STDC_VERSION`, `__STDC_IEC_559` and `STDC_IEC_559_COMPLEX` were added.

6.8.9 Pragma operator

A new feature for ***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 pragma unary operator expressions and pragma directives should be treated consistently in whether they are preserved and in whether macro invocations within them are expended.

6.9 Future language directions

This section includes specific mention of the future direction in which the Committee intends to extend and/or restrict the language. The contents of this section 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 hosted or freestanding implementation. Users are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming program.

6.9.1 Character escape sequences

6.9.2 Storage-class specifiers

See §6.5.1.

6.9.3 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 §6.3.2.2 and §6.5.5.3. The gist of this case is that the new syntax addresses some of the most glaring weaknesses of the language defined in the Base Document, 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.

6.9.4 Function definitions

See §6.9.4.

6.9.5 Pragma directives

7. LIBRARY

7.1 Introduction

The Base Document for this section 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, ioctls, file access permissions and process control facilities. Those definitions were dropped from the Standard. Some other functions were excluded from the Standard because they were non-portable or were ill-defined.

Other facilities not in the library Base Document 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 the Base Document; 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 (e.g., `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 specification 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 if 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 control string pointed at by pointer `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 (i.e., `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 within a well-defined 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 because 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 of where the search terminated. Thus there is no possibility of copying take 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 (e.g., pointer to integer type can not 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 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 prior practice only certain library functions have been associated with header files, the Standard now mandates that *all* library functions have a header. Several headers have therefore been added, and the contents of a few old ones have been changed.

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 an un-hosted environment, where the only "file" support may be a network interface.

The Committee decided to make library headers "idempotent" - 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 typedefs. To ensure that such protective wrapping can be made to work, and to ensure proper scoping of typedefs, headers may only be included outside of any declaration.

Note to implementors: 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 Standard-conforming and alternate implementations is to add the extra, non-Standard, declarations to the header as in this illustration:

```
#ifdef __EXTENSIONS__
typedef int file_no;
extern int read(file_no _N, void * _Buffer, int _Nbytes);
/*...*/
#endif
```

The header is usable in an implementation of the Standard in the absence of a definition of `__EXTENSIONS__`, and the non-Standard implementation can provide the appropriate definitions to enable the extra declarations.

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 space of names 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 this issue of "name space pollution" is currently 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.

7.1.4 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 a static writable memory cell, which interferes with the construction of shareable libraries. Nevertheless, the Committee preferred to standardize this existing, however deficient, machinery rather than invent something more ambitious.

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`.

---

[5] See §6.1.2.1 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 <locals.h> must begin with an underscore, rather than following the pattern of other names in those structures.

[6] In earlier drafts of the Standard, errno and related macros were defined in <stddef.h>. When the Committee decided that the other definitions in this header were os such general utility that they should be required even in freestanding environments, it created <errno.h>.

7.1.5 Limits `<float.h>` and `<limits.h>`

Both `<float.h>` and `<limits.h>` are inventions. 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. Another possible method of determining some of this information is to evaluate arithmetic expressions in the preprocessing statements. 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 (possibly wildly different) arithmetic of the target machine on the host machine. (See §6.4.)

`<float.h>` makes available to programmers a set of useful quantities for numerical analysis. (See §5.2.4.2.) 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 Committee. The set was chosen so as not to prejudice an implementation's selection of floating-point representation.

Most of the limits in `<float.h>` are specified to be general `double` expressions rather than restricted constant expressions

- to allow use of values which cannot readily (or, in some cases, cannot possibly) be constructed as manifest constants, and

- to allow for run-time selection of floating-point properties, as is possible, for instance, in IEEE-854 implementations.

7.1.6 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` (see §6.3.6), `size_t` (see §6.3.3.4), `wchar_t` (see §6.1.3.4), 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 choose to define it as `(void *)0`; this latter form of definition is convenient on architectures where the pointer size(s) do(es) 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 has been 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 data-base 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.[7]

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.

Other implementations may choose to expand this macro as a call to a built-in function that interrogates the translator's symbol table.

7.1.7 Boolean type and values `<stdbool.h>`

7.1.8 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, i.e., 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 §7.13.7.5.)

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 §6.8.3).

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 (see §6.3.2.2), thus maintaining backwards compatibility with existing, pre-Standard, 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*.

---

[7] 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.

7.2 Diagnostics <assert.h>

7.2.1 Program diagnostics

7.2.1.1 The `assert` macro

Some implementations tolerate an arbitrary scalar expression as the argument to `assert`, but the 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);
#define assert(expr) \
   ((expr)? (void)0 : __gripe(#expr, __FILE__, __LINE__))
#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 which would conflict with the user's name space (see§6.1.2.1).

7.3 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.

The Standard (§7.20.2) 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>`.

7.3.1 Character testing 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`.

7.3.1.1 The `isalnum` function

7.3.1.2 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, §7.5) so long as

- `isupper(c)` implies `isalpha(c)`
- `islower(c)` implies `isalpha(c)`
- `isspace(c)`, `ispunct(c)`, `iscntrl(c)`, or `isdigit(c)` implies `!isalpha(c)`

7.3.1.3 The `isblank` function

*`isblank` is a new feature for **C9X***

7.3.1.4 The `iscntrl` function

7.3.1.5 The `isdigit` function

7.3.1.6 The `isgraph` function

7.3.1.7 The `islower` function

7.3.1.8 The `isprint` function

7.3.1.9 The `ispunct` function

7.3.1.10 The `isspace` function

`isspace` is widely used within the library as the working definition of white space.

7.3.1.11 The `isupper` function

7.3.1.12 The `isxdigit` function

7.3.2 Character case mapping functions

Earlier 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; 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.

7.3.2.1 The `tolower` function

7.3.2.2 The `toupper` function

7.4 Integer types `<inttypes.h>`

*`<inttypes.h>` is a new feature for **C9X*** The C Standard specifies that the language should support four, `signed` and `unsigned`, integer data types, `char`, `short`, `int` and `long`. However, the Standard places very little requirement on their size (number of bits) other than that `int` and `short` be at least 16-bits and `long` data type must 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 integral types, because Standard C's integral promotion rule can produce silent changes unexpectedly. The need for defining an *extended integer* type increased with the introduction of 64-bit based systems in the industry.

The purpose of `<inttypes.h>` header 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 to Standard C integer types or extensions that they support. Consistant use of this header will greatly increase the portability of a user's program across platforms.

7.4.1 Typedef names for integer types

7.4.1.1 Exact-width integer types

7.4.1.2 Minimum-width integer types

7.4.1.3 Fastest minimum-width integer types

7.4.1.4 Integer types capable of holding object pointers

7.4.1.5 Greatest-width integer types

7.4.2 Limits of specified-width integer types

7.4.2.1 Limits of exact-width integer types

7.4.2.2 Limits of minimum-width integer types

7.4.2.3 Limits of fastest minimum-width integer types

7.4.2.4 Limits of integer types capable of holding object pointers

7.4.2.5 Limits of greatest-width integer types

7.4.3 Macros for integer constants

7.4.3.1 Macros for minimum-width integer constants

7.4.3.2 Macros for greatest-width integer constants

7.4.4 Macros for format specifiers

7.4.5 Limits of other integer types

7.4.6 Conversion functions for greatest-width integer types

7.4.6.1 The `strtoimax` function

7.4.6.2 The `strtoumax` function

7.4.6.3 The `wcstoimax` function

7.4.6.4 The `wcstoumax` function

7.5 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. This set of letters suffices for English, Swahili, and Hawaiian; all 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. The German "sharp S", ß, occurs only in lower-case. European French usually omits diacriticals on upper-case letters. Some languages 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; 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; a period is common in Europe, and in some countries digits are not grouped by threes. In printing currency amounts, the currency symbol (which may be more than one character) may precede, follow, or be embedded in the digits.

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

1776-07-04	ISO Format
4.7.76	customary central European and British usage
7/4/76	customary U.S. usage
4.VII.76	Italian usage
76186	Julian date (YYDDD)
04JUL76	airline usage
Thursday, July 4, 1776	full U.S. format
Donnerstag, 4. Juli 1776	full German 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 Committee has 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 is not strictly conforming. (Use of national characters in comments is 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. Translate-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. 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 if a signal-handling function were to change the selected locale and return to a library function, that the return value from that library function must be completely correct with respect to the new locale.)

7.5.1 Locale control

7.5.1.1 The `setlocale` function

`setlocale` 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 - these files could then be added and modified without requiring any recompilation of a localizable program.

7.5.2 Numeric formatting convention inquiry

7.5.2.1 The `localeconv` function

The `localeconv` function gives a programmer access to information about how to format numeric quantities (monetary or otherwise). 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 very application-dependent.

7.6 Floating-point environment `<fenv.h>`

*`<fenv.h>` is a new feature for **C9X.***

7.6.1 The FENV_ACCESS pragma

*A new feature for **C9X***

7.6.2 Exceptions

7.6.2.1 The `feclearexcept` function

7.6.2.2 The `fegetexceptflag` function

7.6.2.3 The `feraiseexcept` function

7.6.2.4 The `fesetexceptflag` function

7.6.2.5 The `fetestexcept` function

7.6.3 Rounding

7.6.3.1 The `fegetround` function

7.6.3.2 The `fesetround` function

7.6.4 Environment

7.6.4.1 The `fegetenv` function

7.6.4.2 The `feholdexcept` function

7.6.4.3 The `fesetenv` function

7.6.4.4 The `feupdateenv` function

7.7 Mathematics `<math.h>`

For historical reasons, the math library is only defined for the floating type `double`. All the names formed by appending `f` or `l` to a name in `<math.h>` are reserved to allow for the definition of `float` and `long double` libraries.

The functions `ecvt`, `fcvt`, and `gcvt` have been dropped since their capability is available through `sprintf`.

Traditionally, `HUGE_VAL` has been 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 Committee decided to generalize `HUGE_VAL` to a positive double expression, 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.

Similarly, domain errors in the past were typically indicated by a zero return, which is not necessarily distinguishable from a valid result. The 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 IEEE P854 proposal on floating point representation and arithmetic.

7.7.1 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.

The Standard has been crafted to neither require nor preclude any popular implementation of floating point. 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".

The 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 Committee rejected this approach for several reasons:

- This style is incompatible with popular floating point implementations, such as IEEE 754 (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.

7.7.2 The `FP_CONTACT` pragma

7.7.3 Classification macros

7.7.3.1 The `fpclassify` macro

7.7.3.2 The `signbit` macro

7.7.3.3 The `isfinite` macro

7.7.3.4 The `isnormal` macro

7.7.3.5 The `isnan` macro

7.7.3.6 The `isinf` macro

7.7.4 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.

7.7.4.1 The `acos` function

7.7.4.2 The `asin` function

7.7.4.3 The `atan` function

7.7.4.4 The `atan2` function

The `atan2` function is modeled after FORTRAN's. It is described in terms of arctan y/x for simplicity; the 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 in this situation by various implements, 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 an IEEE *NaN* code.

7.7.4.5 The `cos` function

7.7.4.6 The `sin` function

7.7.4.7 The `tan` function

The tangent function has singularities at odd multiples of $\pi/2$, approaching $+\infty$ from one side and $-\infty$ from the other. Implementations commonly perform argument reduction using the best machine representation of π; 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 Committee has recognized that `tan` is an exception to the *range error* rule (§7.7.1) that an overflowing result produces `HUGE_VAL` properly signed.)

7.7.5 Hyperbolic functions

7.7.5.1 The `cosh` function

7.7.5.2 The `sinh` function

7.7.5.3 The `tanh` function

7.7.5.4 The `acosh` function

7.7.5.5 The `asinh` function

7.7.5.6 The `atanh` function

7.7.6 Exponential and logarithmic functions

7.7.6.1 The `exp` function

7.7.6.2 The `frexp` function

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 nonbinary architectures `ldexp` may lose precision, and `frexp` may be inefficient.

7.7.6.3 The `ldexp` function
See §7.7.6.2.

7.7.6.4 The `log` function

Whether `log(0.)` is a domain error or a range error is arguable. The choice in the Standard, *range error*, is for compatibility with IEEE P854. Some such implementations would represent the result as -∞, in which case no error is raised.

7.7.6.5 The `log10` function
See §7.7.6.4.

7.7.6.6 The `modf` function
See §7.7.6.2.

7.7.6.7 The `exp2` function

7.7.6.8 The `expm1` function

7.7.6.9 The `log1p` function

7.7.6.10 The `log2` function

7.7.6.11 The `logb` function

7.7.6.12 The `scalbn` function

7.7.6.13 The `scalbln` function

7.7.6.14 The `ilogb` function

7.7.7 Power and absolute value functions

7.7.7.1 The `fabs` function

Adding an absolute value operator was rejected by the Committee. An implementation can provide a built-in function for efficiency.

7.7.7.2 The `pow` function

7.7.7.3 The `sqrt` function

IEEE P854, 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: 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 IEEE P854 or vendor-specific floating point representations.

7.7.7.4 The `cbrt` function

7.7.7.5 The `hypot` function

7.7.8 Error and gamma functions

7.7.8.1 The `erf` function

7.7.8.2 The `erfc` function

7.7.8.3 The `gamma` function

7.7.8.4 The `lgamma` function

7.7.9 Nearest integer functions

7.7.9.1 The `ceil` function

Implementation note: The `ceil` function returns the smallest integral value in double format not less than `x`, even though that integer might not be representable in a C integral 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.

7.7.9.2 The `floor` function

7.7.9.3 The `nearbyint` function

7.7.9.4 The `rint` function

7.7.9.5 The `lrint` function

7.7.9.6 The `llrint` function

7.7.9.7 The `round` function

7.7.9.8 The `lround` function

7.7.9.9 The `llround` function

7.7.9.10 The `trunc` function

7.7.10 Remainder functions

7.7.10.1 The `fmod` function

`fmod` 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`-*i**`y`, where *i* 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 a formula 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` - efficiency considerations may dictate a different actual implementation.

The result of `fmod(x,0.0)` is either a domain error or 0.0; the result always lies between 0.0 and `y`, so specifying the non-erroneous result as 0.0 simply recognizes the limit case.

The Committee considered and rejected a proposal to use the remainder operator `%` for this function; the operators in general correspond to hardware facilities, and `fmod` is not supported in hardware on most machines.

7.7.10.2 The `remainder` function

7.7.10.3 The `remquo` function

7.7.11 Manipulation functions

7.7.11.1 The `copysign` function

7.7.11.2 The `nan` function

7.7.11.3 The `nextafter` function

7.7.11.4 The `nextafterx` function

7.7.12 Maximum, minimum, and positive difference functions

7.7.12.1 The `fdim` function

7.7.12.2 The `fmax` function

7.7.12.3 The `fmin` function

7.7.13 Floating multiply-add

7.7.13.1 The `fma` function

7.7.14 Comparison macros

7.7.14.1 The `isgreater` macro

7.7.14.2 The `isgreaterequal` macro

7.7.14.3 The `isless` macro

7.7.14.4 The `islessequal` macro

7.7.14.5 The `islessgreater` macro

7.7.14.6 The `isunordered` macro

7.8 Complex arithmetic `<complex.h>`

7.8.1 The `CX_LIMITED_RANGE` pragma

7.8.2 Complex functions

7.8.2.1 Branch cuts

7.8.2.2 The `cacos` function

7.8.2.3 The `casin` function

7.8.2.4 The `catan` function

7.8.2.5 The `ccos` function

7.8.2.6 The `csin` function

7.8.2.7 The `ctan` function

7.8.2.8 The `cacosh` function

7.8.2.9 The `casinh` function

7.8.2.10 The `catanh` function

7.8.2.11 The `ccosh` function

7.8.2.12 The `csinh` function

7.8.2.13 The `ctanh` function

7.8.2.14 The `cexp` function

7.8.2.15 The `clog` function

7.8.2.16 The `csqrt` function

7.8.2.17 The `cabs` function

7.8.2.18 The `cpow` function

7.8.2.19 The `carg` function

7.8.2.20 The `conj` function

7.8.2.21 The `cimag` function

7.8.2.22 The `cproj` function

7.8.2.23 The `creal` function

7.9 Type-generic math `<tgmath.h>`

7.9.1 The Type-generic macros

7.10 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 struct type is unsuitable. Note that a one-element array of the appropriate type is a valid definition.

`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`.

7.10.1 Save calling environment

7.10.1.1 The `setjmp` macro

One proposed requirement on `setjmp` is that it be usable like any other function - 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 Committee is to require that implementations recognize that calling `setjmp` is a special case[8], 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.

7.10.2 Restore calling environment

7.10.2.1 The `longjmp` function

The Committee also considered requiring that a call to `longjmp` restore the (`setjmp`) calling environment fully - that upon execution of a `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.

Some implementations leave a process in a special state while a `signal` is being handled. An explicit reassurance must be given to the environment when the signal handler is done. To keep this job manageable, the Committee agreed to restrict `longjmp` to only one level of signal handling.

The `longjmp` function should not be called in an exit handler (i.e., a function registered with the `atexit` function (see§7.14.4.2)), since it might jump to some code which is no longer in scope.

7.11 Signal Handling `<signal.h>`

This facility has been retained from the Base Document since the 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, see §7.11.2.1) raising signals. For a discussion of the problems created by including signals, see §5.2.3.

[8] 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.

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 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>`.

7.11.1 Specify signal handling

7.11.1.1 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 re-entrant, they should not be called from a signal handler that returns. (See §5.2.3.) A specific exception to this rule has been 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*.

7.11.2 Send signal

7.11.2.1 The `raise` function

The function `raise` replaces the Base Document'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 Committee deemed it preferable to introduce a function which requires no (dummy) process argument. The Committee anticipates that IEEE 1003 will wish to standardize the `kill` function in the POSIX specification.

7.12 Variable Arguments `<stdarg.h>`

For a discussion of argument passing issues, see §6.7.1.

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 §7.13.6). 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 to state clearly that `va_list` is an object type.

7.12.1 Variable argument list access macros

7.12.1.1 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 `ap` to other functions to do the actual traversal. (It can, of course, traverse the list itself.)

The `parmN` argument to `va_start` is an aid to writing conforming ANSI C code for existing C implementations. Many implementations can use the second parameter within the structure of existing C language constructs to derive the address of the first variable argument. (Declaring `parmN` to be of storage class `register` would interfere with use of these constructs; hence the effect of such a declaration is *undefined behavior*. Other restrictions on the type of `parmN` are imposed for the same reason.) New implementations may choose to use hidden machinery that ignores the second argument to `va_start`, possibly even hiding a function call inside the macro.

Multiple `va_list` variables can be in use simultaneously in the same function; each requires its own calls to `va_start` and `va_end`.

7.12.1.2 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 §6.5.5.3.)

`va_arg` can only be used to access the value of an argument, not to obtain its address.

7.12.1.3 The `va_copy` macro

7.12.1.4 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.

7.13 Input/Output `<stdio.h>`

Many implementations of the C runtime environment (most notably the UNIX operating system) provide, aside from the standard I/O library (`fopen`, `fclose`, `fread`, `fwrite`, `fseek`), a set of unbuffered I/O services (`open`, `close`, `read`, `write`, `lseek`). The Committee has decided not to standardize the latter set of functions.

A suggested semantics for these functions in the UNIX world may be found in the emerging IEEE P1003 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 new-line character. Many hosts use conventions other than an in-stream new-line 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 new-line character translation would be lost (which breaks programs) or the implementor must be aware of the new-line 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 (1) file descriptors 0, 1, and 2 may not be available or may be reserved for another purpose, (2) the operating system may use a different set of services for terminal I/O than file I/O.

In summary, the 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.

7.13.1 Introduction

The macros `_IOFBF`, `_IOLBF`, `_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 the Base Document and not retained in the Standard.

`FOPEN_MAX` and `TMP_MAX` are added 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?) rather than something unusable like `USHRT_MAX`.

`fpos_t` wording has been changed 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.

7.13.2 Streams

C inherited its notion of text streams from the UNIX environment in which it was born. Having each line delimited by a single *new-line* 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 new-line 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 new-line 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 new-line 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 new-line 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 changes 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 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; 254 was chosen.

7.13.3 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.

Buffering, and unbuffering, is defined in a way suggesting 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).

7.13.4 Operations on files

7.13.4.1 The `remove` function

The Base Document provides the `unlink` system call to remove files. The UNIX-specific definition of this function prompted the Committee to replace it with a portable function.

7.13.4.2 The `rename` function

This function has been added to provide a system-independent atomic operation to change the name of an existing file; the Base Document 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 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*.

7.13.4.3 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.

7.13.4.4 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 this possibility.

7.13.5 File access functions

7.13.5.1 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.

7.13.5.2 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.

7.13.5.3 The `fopen` function

The `b` type modifier has been added to deal with the text/binary dichotomy (see §7.13.2). Because of the limited ability to seek within text files (see §7.13.9.1), an implementation is at liberty to treat the old update + modes as if `b` were also specified.

Table 4.1 tabulates the capabilities and actions associated with the various specified mode string arguments to `fopen`.

[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.

Table 4.1: 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 §7.13.5.6.) 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 way, on a system which provides record-oriented binary files, for an implementation to allow a programmer to specify record length.

A change of input/output direction on an update file is only allowed following a `fsetpos`, `fseek`, `rewind`, or `fflush` operation, since these are precisely the functions which assure that the I/O buffer has been flushed.

The Standard (§7.13.2) 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.

7.13.5.4 The `freopen` function

7.13.5.5 The `setbuf` function

`setbuf` is subsumed by `setvbuf`, but has been retained for compatibility with old code.

7.13.5.6 The `setvbuf` function

`setvbuf` has been 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. Further, 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 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.

7.13.6 Formatted input/output functions

7.13.6.1 The `fprintf` function

*A new feature for **C9X***. The modifier `m` has been added to the integer conversion specifiers, indicating that the corresponding argument has type `intmax_t` or `uintmax_t`.

Use of the `L` modifier with floating conversions has been added to deal with formatted output of the new type `long double`.

Note that the `%X` and `%x` formats expect a corresponding `int` argument; `%lX` or `%lx` must be supplied with a `long int` argument.

The conversion specification `%p` has been added for pointer conversion, since the size of a pointer is not necessarily the same as the size of an `int`. Because an implementation may support more than one size of pointer, the corresponding argument is expected to be a `(void *)` pointer.

The `%n` format has been added to permit ascertaining the number of characters converted up to that point in the current invocation of the formatter.

Some pre-Standard implementations switch formats for `%g` at an exponent of -3 instead of (the Standard's) -4: existing code which requires 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 Committee considered the latter notation preferable.

The Committee has reserved lower case conversion specifiers for future standardization.

The use of leading zero in field widths to specify zero padding has been superseded by a precision field. The older mechanism has been retained.

Some implementations have provided the format `%r` 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 §7.13.6.8.)

The printing formats for numbers is not entirely specified. The requirements of the Standard are loose enough to allow implementations to handle such cases as signed zero, *not-a-number*, and *infinity* in an appropriate fashion.

7.13.6.2 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` has been added, although it is obviously risky, for symmetry with `fprintf`. The `%i` format has been added to permit the scanner to determine the radix of the number in the input stream; the `%n` format has been added to make available the number of characters scanned thus far in the current invocation of the scanner.

White space is now defined by the `isspace` function. (See §7.3.1.10.)

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` calls, the pushback in `fscanf` must not affect the push-back 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 issued.

7.13.6.3 The `printf` function

See comments of section §7.13.6.1 above.

7.13.6.4 The `scanf` function

See comments in section §7.13.6.2 above.

7.13.6.5 The `sprintf` function

See §7.13.6.1 for comments on output formatting.

In the interests of minimizing redundancy, `sprintf` has subsumed the older, rather uncommon, `ecvt`, `fcvt`, and `gcvt`.

7.13.6.6 The `snprintf` function

*`snprintf` is a new feature for **C9X.*** The `sprintf` function is very useful, but can overrun the output buffer; 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.

7.13.6.7 The `sscanf` function

The behavior of `sscanf` on encountering end of string has been clarified. See also comments in section §7.13.6.2 above.

7.13.6.8 The `vfprintf` function

The functions `vfprintf`, `vprintf`, and `vsprintf` have been adopted from UNIX System V to facilitate writing special purpose formatted output functions.

7.13.6.9 The `vprintf` function

See §7.13.6.8.

7.13.6.10 The `vsprintf` function

See §7.13.6.8.

7.13.6.11 The `vsnprintf` function

See §7.13.6.6

7.13.6.12 The `vfscanf` function

7.13.6.13 The `vscanf` function

7.13.6.14 The `vsscanf` function

7.13.7 Character input/output functions

7.13.7.1 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.

7.13.7.2 The `fgets` function

This function subsumes `gets`, which has no limit to prevent storage overwrite on arbitrary input (see §7.13.7.7).

7.13.7.3 The `fputc` function

See §7.13.7.1.

7.13.7.4 The `fputs` function

7.13.7.5 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.

7.13.7.6 The `getchar` function

7.13.7.7 The `gets` function

See §7.13.7.2.

7.13.7.8 The `putc` function

See §7.13.7.5.

7.13.7.9 The `putchar` function

7.13.7.10 The `puts` function

`puts(s)` is not exactly equivalent to `fputs(stdout,s)`; `puts` also writes a new line after the argument string. This incompatibility reflects existing practice.

7.13.7.11 The `ungetc` function

The Base Document requires that at least one character be read before `ungetc` is called, in certain implementation-specific cases. The Committee has 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.

It is permissible to push back a different character than that which was read; this 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]

7.13.8 Direct input/output functions

7.13.8.1 The `fread` function

`size_t` is the appropriate type both for an object size and for an array bound (see §6.3.3.4), so this is the type of `size` and `nelem`.

7.13.8.2 The `fwrite` function

See §7.13.8.1.

7.13.9 File positioning functions

7.13.9.1 The `fgetpos` function

`fgetpos` and `fsetpos` have been added to allow random access operations on files which are too large to handle with `fseek` and `ftell`.

7.13.9.2 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 §7.13.2). 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`-`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` have been added to deal with files which 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, when it will be set.

7.13.9.3 The `fsetpos` function

7.13.9.4 The `ftell` function

[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.

`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; or
- the file may be positioned at a location not representable in a `long int`.

Thus a method for `ftell` to report failure has been specified. See also §7.13.9.1.

7.13.9.5 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.

7.13.10 Error-handling functions

7.13.10.1 The `clearerr` function

7.13.10.2 The `feof` function

7.13.10.3 The `ferror` function

7.13.10.4 The `perror` function

At various times, the 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 §7.15.6.1).

7.14 General Utilities `<stdlib.h>`

The header `<stdlib.h>` was invented by the Committee to hold an assortment of functions that were otherwise homeless.

7.14.1 String conversion functions

7.14.1.1 The `atof` function

`atof`, `atoi`, and `atol` are subsumed by `strtod` and `strtol`, but have been 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.

7.14.1.2 The `atoi` function
See §7.14.1.1.

7.14.1.3 The `atol` function
See §7.14.1.1.

7.14.1.4 The `atoll` function

7.14.1.5 The `strtod` function

`strtod` and `strtol` have been adopted (from UNIX System V) because they offer more control over the conversion process, and because they are required not to produce unexpected results on overflow during conversion.

7.14.1.6 The `strtof` function

7.14.1.7 The `strtold` function

7.14.1.8 The `strtol` function
See §7.14.1.5.

7.14.1.9 The `strtoll` function

7.14.1.10 The `strtoul` function

`strtoul` was introduced by the Committee to provide a facility like `strtol` for `unsigned long` values. Simply using `strtol` in such cases could result in overflow upon conversion.

7.14.1.11 The `strtoull` function

7.14.2 Pseudo-random sequence generation functions

7.14.2.1 The `rand` function

The 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 has published as an example in the Standard an algorithm that generates the same pseudo-random sequence in any conforming implementation, given the same seed.

7.14.2.2 The `srand` function

7.14.3 Memory management functions

The treatment of null pointers and 0-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 OBJ's */

/* initial allocation */
p = (OBJ *) calloc(0, sizeof(OBJ));
/* ... */

/* reallocations until size settles */
```

```
while(/* list changes size to c */) {
    p = (OBJ *) realloc((void *)p, c*sizeof(OBJ));
    /* ... */
}
```

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 0 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 Committee has 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

> A program which relies on size-0 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; 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.

7.14.3.1 The `calloc` function

Both `nelem` and `elsize` must be of type `size_t`, for reasons similar to those for `fread` (see §7.13.8.1).

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.

7.14.3.2 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.

7.14.3.3 The `malloc` function

7.14.3.4 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-size objects.

7.14.4 Communication with the environment

7.14.4.1 The `abort` function

The Committee vacillated over whether a call to `abort` should return if the `signal SIGABRT` is caught or ignored. To minimize astonishment, the final decision was that `abort` never returns.

7.14.4.2 The `atexit` function

`atexit` provides a program with a convenient way to clean up the environment before it exits. It is 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.

7.14.4.3 The `exit` function

The argument to `exit` is a status indication returned to the invoking environment. In the UNIX operating system, a value of 0 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 then implementation-defined. The macro `EXIT_FAILURE` has been added to provide such a capability. (`EXIT_SUCCESS` has been 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`).

7.14.4.4 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` is 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.

7.14.4.5 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; assuming more than one such level is obviously dangerous.

7.14.5 Searching and sorting utilities

7.14.5.2 The `bsearch` function

7.14.5.1 The `qsort` function

7.14.6 Integer arithmetic 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.

7.14.6.1 The `abs` function

The Committee rejected proposals to add an absolute value operator to the language. An implementation can provide a built-in function for efficiency.

7.14.6.2 The `div` function

`div` and `ldiv` provide a well-specified semantics for signed integral 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 modelling 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` and `ldiv`

number	denom	quot	rem
7	3	2	1
-7	3	-2	-1
7	-3	-2	1
-7	-3	2	-1

Divide-by-zero is described as *undefined behavior* rather than as setting `errno` to `EDOM`. The program can 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.

7.14.6.3 The `labs` function

7.14.6.4 The `llabs` function

7.14.6.5 The `ldiv` function

7.14.6.6 The `lldiv` function

7.14.7 Multibyte character functions

See §5.2.1.2 for an overall discussion of multibyte character representations and widecharacters.

7.14.7.1 The `mblen` function

7.14.7.2 The `mbtowc` function

7.14.7.3 The `wctomb` function

7.14.8 Multibyte string functions

See §5.2.1.2 for an overall discussion of multibyte character representations and wide characters.

7.14.8.1 The `mbstowcs` function

7.14.8.2 The `wcstombs` function

7.15 String Handling `<string.h>`

The Committee felt that the functions in this section 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.

7.15.1 String function conventions

`memcpy`, `memset`, `memcmp`, and `memchr` have been adopted 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-`), and
- transparent data of specified length (`mem-`).

7.15.2 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 presume nonoverlapping operands and be optimized accordingly.

7.15.2.1 The `memcpy` function

7.15.2.2 The `memmove` function

7.15.2.3 The `strcpy` function

7.15.2.4 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 has preferred to recognize existing practice rather than alter the function to better suit it to such use.

7.15.3 Concatenation functions

7.15.3.1 The `strcat` function

7.15.3.2 The `strncat` function

Note that this function may add `n`+1 characters to the string.

7.15.4 Comparison functions

7.15.4.1 The `memcmp` function
See §7.15.1.

7.15.4.2 The `strcmp` function

7.15.4.3 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 - the cost of transformation is then only paid once.

7.15.4.4 The `strncmp` function

7.15.4.5 The `strxfrm` function
See §7.15.4.3.

7.15.5 Search functions

7.15.5.1 The `memchr` function
See §7.15.1.

7.15.5.2 The `strchr` function

7.15.5.3 The `strcspn` function

7.15.5.4 The `strpbrk` function

7.15.5.5 The `strrchr` function

7.15.5.6 The `strspn` function

7.15.5.7 The `strstr` function

The `strstr` function is an invention of the Committee. It is included as a hook for efficient substring algorithms, or for built-in substring instructions.

7.15.5.8 The `strtok` function

This function has been included to provide a convenient solution to many simple problems of lexical analysis, such as scanning command line arguments.

7.15.6 Miscellaneous functions

7.15.6.1 The `memset` function
See §7.15.1, and §7.14.3.1.

7.15.6.2 The `strerror` function

This function is a descendant of `perror` (see §7.13.10.4). 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.

7.15.6.3 The `strlen` function

This function is now specified as returning a value of type `size_t`. (See §6.3.3.4.)

7.16. DATE AND TIME `<time.h>`

7.16.1 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 integral type.

Many C environments do not support the Base Document library concepts of daylight savings 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. The Base Document function `tzset`, which would require dealing with time zones, has been 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 displacement between Universal Time (*née* GMT) and local time is not available.

7.16.2 Time manipulation functions

7.16.2.1 The `clock` function

The 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.

7.16.2.2 The `difftime` function

`difftime` is an invention of the 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.

7.16.2.3 The `mktime` function

`mktime` was invented by the 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, i.e., 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 savings 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 savings time. As DST is currently legislated in the USA, the hour 0100-0159 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)`).

7.16.2.4 The `mkxtime` function

7.16.2.5 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.

7.16.2.6 Normalization of broken-down times

7.16.3 Time conversion functions

7.16.3.1 The `asctime` function

`asctime` Although the name of this function suggests a conflict with the principle of removing ASCII dependencies from the Standard, the name has been 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 (§7.16.3.6) provides appropriate facilities for locale-specific date and time strings.

7.16.3.2 The `ctime` function

7.16.3.3 The `gmtime` function

This function has been retained, despite objections that GMT - that is, Coordinated Universal Time (UTC) - is not available in some implementations, since UTC is a useful and widespread standard representation of time. If UTC is not available, a null pointer may be returned.

7.16.3.4 The `localtime` function

7.16.3.5 The `zonetime` function

7.16.3.6 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.

7.16.3.7 The `strfxtime` function

7.17 Alternative spellings `<iso646.h>`

7.18 Wide-chatacer classification and mapping utilities `<wctype.h>`

7.18.1 Introduction

7.18.2 Wide-character classification utilities

7.18.2.1 Wide-character classification functions

7.18.2.1.1 The `iswalnum` function

7.18.2.1.2 The `iswalpha` function

7.18.2.1.3 The `iswblank` function

7.18.2.1.4 The `iswcntrl` function

7.18.2.1.5 The `iswdigit` function

7.18.2.1.6 The `iswgraph` function

7.18.2.1.7 The `iswlower` function

7.18.2.1.8 The `iswprint` function

7.18.2.1.9 The `iswpunct` function

7.18.2.1.10 The `iswspace` function

7.18.2.1.11 The `iswupper` function

7.18.2.1.12 The `iswxdigit` function

7.18.2.2 Extensible wide-character classification functions

7.18.2.2.1 The `wctype` function

7.18.2.2.2 The `iswctype` function

7.18.3 Wide-character mapping utilities

7.18.3.1 Wide-character case-mapping functions

7.18.3.1.1 The `towlower` function

7.18.3.1.2 The `towupper` function

7.18.3.2 Extensible wide-character mapping functions

7.18.3.2.1 The `wctrans` function

7.18.3.2.2 The `towctrans` function

Extended multibyte and wide-character utilities `<wchar.h>`

7.19.1 Introduction

7.19.2 Formatted wide-character input-output functions

7.19.2.1 The `fwprintf` function

7.19.2.2 The `fwscanf` function

7.19.2.3 The `wprintf` function

7.19.2.4 The `wscanf` function

7.19.2.5 The `swprintf` function

7.19.2.6 The `swscanf` function

7.19.2.7 The `vfwprintf` function

7.19.2.8 The `vwprintf` function

7.19.2.9 The `vswprintf` function

7.19.2.10 The `vfwscanf` function

7.19.2.11 The `vwscanf` function

7.19.2.12 The `vswscanf` function

7.19.3 Wide-character input-output functions

7.19.3.1 The `fgetwc` function

7.19.3.2 The `fgetws` function

7.19.3.3 The `fputwc` function

7.19.3.4 The `fputws` function

7.19.3.5 The `getwc` function

7.19.3.6 The `getwchar` function

7.19.3.7 The `putwc` function

7.19.3.8 The `putwchar` function

7.19.3.9 The `ungetwc` function

7.19.3.10 The `fwide` function

7.19.4 General wide-string utilities

7.19.4.1 Wide-string numeric conversion functions

7.19.4.1.1 The `wcstod` function

7.19.4.1.2 The `wcstof` function

7.19.4.1.3 The `wcstold` function

7.19.4.1.4 The `wcstol` function

7.19.4.1.5 The `wcstoll` function

7.19.4.1.6 The `wcstoul` function

7.19.4.1.7 The `wcstoull` function

7.19.4.2 Wide-string copying functions

7.19.4.2.1 The `wcscpy` function

7.19.4.2.2 The `wcsncpy` function

7.19.4.3 Wide-string concatenation functions

7.19.4.3.1 The `wcscat` function

7.19.4.3.2 The `wcsncat` function

7.19.4.4 Wide-string comparison functions

7.19.4.4.1 The `wcscmp` function

7.19.4.4.2 The `wcscoll` function

7.19.4.4.3 The `wcsncmp` function

7.19.4.4.4 The `wcsxfrm` function

7.19.4.5 Wide-string search functions

7.19.4.5.1 The `wcschr` function

7.19.4.5.2 The `wcscspn` function

7.19.4.5.3 The `wcspbrk` function

7.19.4.5.4 The `wcsrchr` function

7.19.4.5.5 The `wcsspn` function

7.19.4.5.6 The `wcsstr` function

7.19.4.5.7 The `wcstok` function

7.19.4.5.8 The `wcslen` function

7.19.4.6 Wide-character array functions

7.19.4.6.1 The `wmemchr` function

7.19.4.6.2 The `wmemcmp` function

7.19.4.6.3 The `wmemcpy` function

7.19.4.6.4 The `wmemmove` function

7.19.4.6.5 The `wmemset` function

7.19.5 The `wcsftime` function

7.19.6 The `wcsfxtime` function

7.19.7 Extended multibyte and wide-character conversion utilities

7.19.7.1 Single-byte wide-character conversion functions

7.19.7.1.1 The `btowc` function

7.19.7.1.2 The `wctob` function

7.19.7.2 The `mbsinit` function

7.19.7.3 Restartable multibyte-wide-character conversion functions

7.19.7.3.1 The `mbrlen` function

7.19.7.3.2 The `mbrtowc` function

7.19.7.3.3 The `wcrtomb` function

7.19.7.4 Restartable multibyte-wide-string conversion functions

7.19.7.4.1 The `mbsrtowcs` function

7.19.7.4.2 The `wcsrtombs` function

7.20 Future library directions

7.20. Errors `<errno.h>`

7.20.2 Character handling `<ctype.h>`

7.20.3 Integral types `<inttypes.h>`

7.20.4 Localization `<locale.h>`

7.20.5 Signal handling `<signal.h>`

7.20.6 Input/output `<stdio.h>`

7.20.7 General utilities `<stdlib.h>`

7.20.8 Complex arithmetic `<complex.h>`

7.20.9 String handling `<string.h>`

7.20.10 Wide-character classification and mapping utilities `<wctype.h>`

7.20.11 Extended multibyte and wide-character utilities `<wchar.h>`

8. ANNEXES

Most of the material in the appendices 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 section 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.

A. MSE ANNEX *(new text)*

*This annex text was taken from the rationale furnished with the AMENDMENT ISO 9899:1990/DAM 1 document. The references to the AMENDMENT document are correct for the AMENDMENT ISO 9899:1990/DAM 1 document but are not correct for the **C9X** working paper.*

A.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 ISO/IEC 9899:1990 § 6.13. Even in the Latin world, a multibyte coded character set may appear in the near future. This conflict is called a *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 ISO/IEC 9899:1990:

- 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, these 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.

ISO/IEC 9899:1990 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 committee ISO JTC1/SC22/WG14 was to study the various existing implementations and, with care, develop this first amendment to ISO/IEC 9899:1990. The set of developed library functions is commonly called the *MSE* (Multibyte Support Extension).

Similarly, ISO/IEC 9899:1990 deliberately chose not to address in detail the problem of writing C source code with character sets such as the national variants of ISO 646. These variants often redefine several of the punctuation characters used to write a number of C tokens. The (admittedly 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.

This amendment 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, this amendment knowingly introduces two classes of quiet changes:

- new tokens — The tokens `%:` and `%:%:` are just sequences of preprocessing tokens in ISO/IEC 9899:1990 but become single preprocessing tokens with specific meanings in this amendment. An existing program that uses either of these tokens in a macro argument can behave differently as a result of this amendment.

- new function names — Several names (with external linkage) not reserved to the implementation in ISO/IEC 9899:1990, such as `btowc`, are now so reserved if *any* translation unit in the program includes either of the headers `<wctype.h>` or `<wchar.h>`, even though none of the translation units using the name include the new header. An existing program that uses any of these names can behave differently as a result of this amendment.

A.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 `mbtowc` function.

After all necessary multibyte characters are read and converted, the `wchar_t` objects are processed in memory by the *MSE* functions, such as `iswxxx`, `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 `wctomb` 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 § A.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;
}
```

A.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 this amendment is similar to the corresponding single-byte function in ISO/IEC 9899:1990. 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 is relatively easy.

On other hand, "improvement" means that a function interface in this amendment is changed from the corresponding single-byte functions in ISO/IEC 9899:1990 in order to resolve problems potentially contained in the existing functions. Or, a corresponding function is not introduced in this amendment 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. The current amendment is the result of this process.

The following is a list of the functions that manipulate characters in parallel:

ISO/IEC 9899:1990	Amendment
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
fprintf	fwprintf
fscanf	fwscanf
printf	wprintf
scanf	wscanf
sprintf	swprintf
sscanf	swscanf
vfprintf	vwfprintf
vprintf	vwprintf
vsprintf	vswprintf
fgetc	fgetwc
fgets	fgetws
fputc	fputwc
fputs	fputws
getc	getwc
getchar	getwchar
putc	putwc
putchar	putwchar
ungetc	ungetwc
strtod	wcstod
strtol	wcstol
strtoul	wcstoul
memcpy	wmemcpy
memmove	wmemmove
strcpy	wcscpy
strncpy	wcsncpy
strcat	wcscat
strncat	wcsncat
memcmp	wmemcmp
strcmp	wcscmp
strcoll	wcscoll
strncmp	wcsncmp
strxfrm	wcsxfrm
memchr	wmemchr
strchr	wcschr
strcspn	wcscspn

```
strpbrk              wcspbrk
strrchr              wcsrchr
strspn               wcsspn
strstr               wcsstr
memset               wmemset
strlen               wcslen

strftime             wcsftime
```

Note that there may still be subtle differences, see for example A.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 repair the security hole that the existing functions carry. 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 ISO/IEC 9899:1990 that do not have corresponding partners in the amendment 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:

```
gets
atof
puts
atoi
perror
atol
strerror
```

Finally, the following is a list of the functions in this amendment that do not have corresponding partners in ISO/IEC 9899:1990. They were introduced to achieve better control over the conversion between multibyte characters and wide character, or to give character handling programs greater flexibility and simplicity:

```
wctype
iswctype
wctrans
towctrans
fwide
btowc
wctob
```

```
mbsinit
mbrlen
mbrtowc
wcrtomb
mbsrtowcs
wcsrtombs
```

A.4 Support for invariant ISO 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 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.

ISO/IEC 9899:1990 addresses the problem in a different way. It provides replacements at the level of individual characters using three-character sequences called *trigraphs*. For example, `??<` is entirely equivalent to `{`, even within a character constant or string literal. While this approach provides a complete solution for the known limitations of EBCDIC and ISO 646, the result is arguably not highly readable.

Thus, this amendment provides a set of more readable *digraphs*. These are two-character alternate spellings for several of the operators and punctuators that can be hard to read with ISO 646 national variants. Trigraphs are still required within character constants and string literals, but at least the commoner 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 amendment is incomplete and involves a mixture of approaches, but nevertheless believes that it can help make Standard C programs more readable.

A.5 Headers

A.5.1 <wchar.h>

A.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 ISO/IEC 9899:1990, 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 ISO/IEC 9899:1990 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.

And the committee strongly opposed adding any identifiers to existing headers.

The drawback to idea 3 is that it introduces an asymmetry between existing headers and the new header.

The drawback to idea 4 is that the committee strongly opposed adding any 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 new header <wctype.h>, as described in § A.5.2.)

A.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`. (How these types are defined in <stdio.h> may need to be revised so that suitable synonyms, with reserved names, can be used within <wchar.h>.)

A.5.2 `<wctype.h>`

The committee originally intended to place all *MSE* functionality in a single header, `<wchar.h>`, as explained in § A.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 seemed to form a separate group. (These are functions that typically have names of the form `iswxxx` or `towxxx`.) 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.

Hence, the committee 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.

A.6 Wide-character classification functions

Eleven `iswxxx` functions have been introduced to correspond to the character-testing functions defined in ISO/IEC 9899:1990. Each wide-character testing function is specified in parallel with the matching single-byte character handling function. However, the following changes were also introduced.

A.6.1 Locale dependency of iswxxx functions

The behavior of character-testing functions in ISO/IEC 9899:1990 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 ISO/IEC 9899:1990 has been replaced with a supersetting constraint for wide-character testing functions. There is no special description of "C" locale behavior for the `iswxxx` functions. Instead, the following rule is applied to any locale. When a character `c` causes `isxxx(c)` to return true, the corresponding wide character `wc` shall cause the corresponding function call `iswxxx(wc)` to return true.

```
isxxx(c) != 0 ==> iswxxx(wc) !=0
```

where `c == wctob(wc)`. Note that the converse relationship does not necessarily hold.

A.6.2 Changed space-character handling

The space character (' ') is treated specially in `isprint`, `isgraph`, and `ispunct`. Space-character handling in the corresponding wide-character functions differs from that specified in ISO/IEC 9899:1990. 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 ISO/IEC 9899:1990 in this regard. (See the footnote concerning iswgraph in this amendment).

A.7 Extensible classification and mapping functions

There are eleven standard character-testing functions defined in ISO/IEC 9899:1990. 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 `isxxx` functions, such as `iskanji`, `ishanzi`, etc.

However, this approach 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 this amendment, 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.

A.8 Generalized multibyte characters

ISO/IEC 9899:1990 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. Hence, the one-byte sequence `'a'` should always represent `L'a'` (at least initially) and `'\0'` should always represent `L'\0'`, to put matters most simply.

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 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 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.)

A.9 Streams and files

A.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 § 4.5.3.2 of this amendment, 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 hence 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.

A.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.

A.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 a long object 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.

A.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.

A.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 encoding-rule information 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

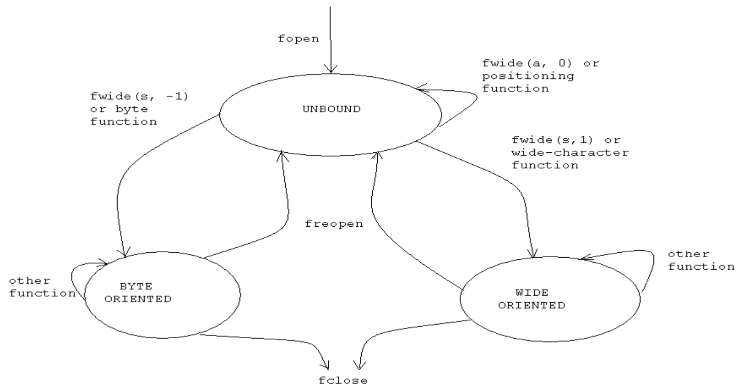
A.9.3 Byte versus wide-character input/output

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 § A.9.2, however, there is a constraint on mixed usage of the two type of input/output functions. That is, if a wide-character input/output functions is executed for a FILE object, its stream becomes wide-oriented and no byte input/output functions shall 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 (or 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



A.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 specifies the following usage of the wide-character input/output functions. 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.

A.10 Formatted input/output functions

A.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 consisting of characters only (such as strings) 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. 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 (in § 7.20.6 of ISO/IEC 9899:1990), 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. These parse or generate multibyte sequences "restartably." Thus, the functions described in § 4.6.5.1, § 4.6.5.2, § 4.6.5.3, and § 4.6.5.4 of this amendment were introduced.

Because these new conversions are pure extensions to ISO/IEC 9899:1990, they have several essential restrictions on their expected, therefore, 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. And 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 the examples in § 4.6.2.3.2 of this amendment.

- `fprintf` function — In a state-dependent encoding, redundant shift sequences may be written.

A.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.

A.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.) Hence, the committee introduced the function `fwide` 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 § A.9.2.3.)

A.12 Single-byte wide-character conversion functions

Two single-byte wide-character conversion functions, `btowc` and `wctob`, have been introduced in this amendment. These functions simplify mappings between a single-byte character and its corresponding wide character (if any).

ISO/IEC 9899:1990 specifies the rule `L'x' == 'x'` for a member `x` of the basic character set. The committee discussed whether to relax or tighten this rule. In this amendment, this rule is preserved without any changes. Applying the rule to all single-byte characters, however, imposes an unnecessary constraint on implementation 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 function `wctob` can be used to test safely and quickly whether a wide character corresponds to some single-byte character. For example, when the format string on a `scanf` function call is parsed and searched for a white space character, the `wctob` function can be used in conjunction with the `isspace` function. (See the specification of the `iswxxx` functions in § 4.5.2.1 of this amendment.)

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 in-line expression.

A.13 Extended conversion utilities

Although ISO/IEC 9899:1990 allows multibyte characters to have state-dependent encoding (§ 5.2.1.2), the original functions are not always sufficient to efficiently support state-dependent encodings, due to the following limitations of the multibyte character conversion functions (§ 7.14.7):

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. The multibyte-string conversion functions (§ 7.14.8) have an inconvenient shortcoming, regardless of state dependency of the encoding. When an encoding error occurs, these functions return 1961 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 this amendment.

A.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 to wide character, the conversion state stores information such as the position within the current multibyte character (as a sequence of characters or a wide-character accumulator). And 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. (It is not necessarily the *only* way to encode the initial conversion state, however.) Before any operations are performed on it, such a zero-valued 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.

A.13.2 Conversion utilities

Once the `mbstate_t` object was introduced, the committee discussed the need for additional functions to manipulate such objects.

A.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 this amendment. 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.

A.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.

A.13.2.3 Testing for initial shift state

The function `mbsinit` 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 the amendment treat the initial shift state as a special condition.

A.13.2.4 Restartable multibyte functions

Regarding problems 2 and 3 described at the beginning of § A.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 1962 instead of 1961, 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 argument. As part of this enhancement, the functions also have a parameter of type pointer to pointer to the source to the position where the conversion stopped.

A.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 a fixed width of 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 were another set of functions that return the column width for any kind of display devices for a given wide character or wide-character string; but these seemed to be beyond the scope of C language. Thus all proposals regarding column width were withdrew.

If an implementor needs this kind of functionality, there are a few ways to extend wide-character output functions and still remain conforming to this amendment. For example, the field width prefixed with a `#` can 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