definitions

|  |  |  |
| --- | --- | --- |
| Foreword |  | Unicode characters and strings (<uchar.h>) (originally specified in ISO/IEC TR 19769:2004) |
| Introduction |  | The Working Group responsible for this standard (WG 14) maintains a site on the World Wide Web at <http://www.open-std.org/JTC1/SC22/WG14/> containing additional information relevant to this standard such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses. |
| 2. Normative references |  | ISO/IEC 2382-1:1993, Information technology — Vocabulary — Part 1: Fundamental terms. |
| 3.1 | access | <execution-time action> to read or modify the value of an object  NOTE 1 Where only one of these two actions is meant, "read" or "modify" is used.  NOTE 2 "Modify" includes the case where the new value being stored is the same as the previous value.  NOTE 3 Expressions that are not evaluated do not access objects. |
| 3.2 | alignment | requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address |
| 3.3 | argument  actual argument  actual parameter (deprecated) | expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation |
| 3.4 | behavior | external appearance or action |
| 3.4.1 | implementation-defined behavior | unspecified behavior where each implementation documents how the choice is made  EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right. |
| 3.4.2 | locale-specific behavior | behavior that depends on local conventions of nationality, culture, and language that each implementation documents  EXAMPLE An example of locale-specific behavior is whether the islower function returns true for characters other than the 26 lowercase Latin letters. |
| 3.4.3 | undefined behavior | behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements  NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).  EXAMPLE An example of undefined behavior is the behavior on integer overflow. |
| 3.4.4 | unspecified behavior | use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance  EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated. |
| 3.5 | bit | unit of data storage in the execution environment large enough to hold an object that may have one of two values  NOTE It need not be possible to express the address of each individual bit of an object. |
| 3.6 | byte  low-order bit  high-order bit | addressable unit of data storage large enough to hold any member of the basic character set of the execution environment  NOTE 1 It is possible to express the address of each individual byte of an object uniquely.  NOTE 2 A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. |
| 3.7 | character | <abstract> member of a set of elements used for the organization, control, or representation of data |
| 3.7.1 | character | single-byte character  <C> bit representation that fits in a byte |
| 3.7.2 | multibyte character | sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment  NOTE The extended character set is a superset of the basic character set. |
| 3.7.3 | wide character | value representable by an object of type wchar\_t, capable of representing any character in the current locale |
| 3.8 | constraint | restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted |
| 3.9 | correctly rounded result | representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision |
| 3.10 | diagnostic message | message belonging to an implementation-defined subset of the implementation's message output |
| 3.11 | forward reference | reference to a later subclause of this International Standard that contains additional information relevant to this subclause |
| 3.12 | implementation | particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment |
| 3.13 | implementation limit | restriction imposed upon programs by the implementation |
| 3.14 | memory location | either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width  NOTE 1 Two threads of execution can update and access separate memory locations without interfering  with each other.  NOTE 2 A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (non-zero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.  EXAMPLE A structure declared as   |  | | --- | | struct {  char a;  int b:5, c:11, :0, d:8;  struct { int ee:8; } e;  } |   contains four separate memory locations: The member a, and bit-fields d and e.ee are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields b and c together constitute the fourth memory location. The bit-fields b and c cannot be concurrently modified, but b and a, for example, can be. |
| 3.15 | object | region of data storage in the execution environment, the contents of which can represent values  NOTE When referenced, an object may be interpreted as having a particular type; see 6.3.2.1. |
| 3.16 | parameter  formal parameter  formal argument (deprecated) | object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition |
| 3.17 | recommended practice | specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations |
| 3.18 | runtime-constraint | requirement on a program when calling a library function  NOTE 1 Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.  NOTE 2 Implementations that support the extensions in annex K are required to verify that the runtime-  constraints for a library function are not violated by the program; see K.3.1.4. |
| 3.19 | value | precise meaning of the contents of an object when interpreted as having a specific type |
| 3.19.1 | implementation-defined value | unspecified value where each implementation documents how the choice is made |
| 3.19.2 | indeterminate value | either an unspecified value or a trap representation |
| 3.19.3 | unspecified value | valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance  NOTE An unspecified value cannot be a trap representation. |
| 3.19.4 | trap representation | an object representation that need not represent a value of the object type |
| 3.19.5 | perform a trap | interrupt execution of the program such that no further operations are performed  NOTE In this International Standard, when the word "trap" is not immediately followed by "representation", this is the intended usage. 2) |
| 3.20 | ... | ceiling of x: the least integer greater than or equal to x  EXAMPLE ... |
| 3.21 | ... | floor of x: the greatest integer less than or equal to x  EXAMPLE ... |
| 4. Conformance | undefined behavior | If a "shall" or "shall not" requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined". |
| 4. Conformance | strictly conforming program | A strictly conforming program shall use only those features of the language and library specified in this International Standard. 3) It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit. |
| 4. Conformance | conforming implementation | The two forms of conforming implementation are hosted and freestanding. A conforming hosted implementation shall accept any strictly conforming program. A conforming freestanding implementation shall accept any strictly conforming program in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers <float.h>, <iso646.h>, <limits.h>, <stdalign.h>, <stdarg.h>, <stdbool.h>, <stddef.h>, <stdint.h>, and <stdnoreturn.h>. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program. 4)  4) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard. |
| 4. Conformance | conforming program | A conforming program is one that is acceptable to a conforming implementation. 5)  5) Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend upon nonportable features of a conforming implementation. |
| 5. Environment |  | An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the translation environment and the execution environment in this International Standard. |
| 5.1.1.1 Program structure | source files  preprocessing files | The text of the program is kept in units called source files, (or preprocessing files) in this International Standard. |
| 5.1.1.1 Program structure | preprocessing translation unit | A source file together with all the headers and source files included via the preprocessing directive #include is known as a preprocessing translation unit. |
| 5.1.1.1 Program structure | translation unit | After preprocessing, a preprocessing translation unit is called a translation unit. |
| 5.1.1.1 Program structure | translated translation unit |  |
| 5.1.1.1 Program structure | new-line character |  |
| 5.1.1.1 Program structure | end-of-line indicator |  |
| 5.1.1.2 Translation phases | physical source line  logical source line | Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. |
| 5.1.1.2 Translation phases | program image |  |
| 5.1.2 Execution environments |  | Two execution environments are defined: freestanding and hosted. |
| 5.1.2 Execution environments | program startup | In both cases, program startup occurs when a designated C function is called by the execution  environment. |
| 5.1.2 Execution environments | initialize | All objects with static storage duration shall be initialized (set to their initial values) before program startup. |
| 5.1.2 Execution environments | program termination | Program termination returns control to the execution environment. |
| 5.1.2.1 Freestanding environment | freestanding environment | In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. |
| 5.1.2.2.1 Program startup | main | The function called at program startup is named main. |
| 5.1.2.2.1 Program startup | program name | If the value of argc is greater than zero, the string pointed to by argv[0] represents the program name; argv[0][0] shall be the null character if the program name is not available from the host environment. |
| 5.1.2.2.1 Program startup | program parameters | If the value of argc is greater than one, the strings pointed to by argv[1] through argv[argc-1] represent the program parameters. |
| 5.1.2.2.2 Program execution | （hosted environment） | In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (clause 7). |
| 5.1.2.3 Program execution | side effects | Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, 12) which are changes in the state of the execution environment. |
| 5.1.2.3 Program execution | evaluation | Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object. |
|  | sequenced before  sequenced after |  |
|  | unsequenced |  |
|  | indeterminately sequenced |  |
|  | sequence point |  |
| 5.1.2.3 Program execution | observable behavior | The least requirements on a conforming implementation are:  — Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.  — At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.  — The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.  This is the observable behavior of the program. |
|  | implicit spilling |  |
|  | explicit store and load |  |
|  | roundoff error |  |
| 5.1.2.4 Multi-threaded executions and data races | thread of execution (or thread)  The execution of each thread  The execution of the entire program | Under a hosted implementation, a program can have more than one thread of execution (or thread) running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. 14)  14) The execution can usually be viewed as an interleaving of all of the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described below. |
|  | conflict | Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. |
|  | atomic operations |  |
|  | synchronization operation |  |
|  | acquire operation  release operation  consume operation |  |
|  | fence |  |
|  | acquire fence  release fence |  |
|  | relaxed atomic operation |  |
|  | read-modify-write operation |  |
|  | ... |  |
| 5.2.1 Character sets | source character set  execution character set | Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the source character set), and the set interpreted in the execution environment (the execution character set). |
| 5.2.1 Character sets | basic character set  extended characters  extended character set | Each set is further divided into a basic character set, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called extended characters. The combined set is also called the extended character set. |
| 5.2.1 Character sets | escape sequence | In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by escape sequences consisting of the backslash \ followed by one or more characters. |
| 5.2.1 Character sets | null character | A byte with all bits set to 0, called the null character, shall exist in the basic execution character set; it is used to terminate a character string. |
| 5.2.1 Character sets | uppercase letter  lowercase letters  digits  graphic characters  control characters |  |
| 5.2.1 Character sets | end-of-line indicator | In source files, there shall be some way of indicating the end of each line of text; this International Standard treats such an end-of-line indicator as if it were a single new-line character. |
| 5.2.1 Character sets | letter | A letter is an uppercase letter or a lowercase letter as defined above; in this International Standard the term does not include other characters that are letters in other alphabets. |
| 5.2.1.1 Trigraph sequences | trigraph sequences | Before any other processing takes place, each occurrence of one of the following sequences of three characters (called trigraph sequences 17) ) is replaced with the corresponding single character.  ??= #  ??( [  ??/ \  ??) ]  ??' ^  ??< {  ??! |  ??> }  ??- ~  No other trigraph sequences exist. Each ? that does not begin one of the trigraphs listed above is not changed. |
| 5.2.1.2 Multibyte characters | multibyte character | The source character set may contain multibyte characters, used to represent members of the extended character set. |
| 5.2.1.2 Multibyte characters | state-dependent encoding  initial shift state  shift state | A multibyte character set may have a state-dependent encoding, wherein each sequence of multibyte characters begins in an initial shift state and enters other locale-specific shift states when specific multibyte characters are encountered in the sequence. |
| 5.2.2 Character display semantics | active position | The active position is that location on a display device where the next character output by the fputc function would appear. |
| 5.2.2 Character display semantics | alert  backspace  form feed  logical page  new line  carriage return  horizontal tab  vertical tab |  |
| 5.2.3 Signals and interrupts | function image | Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration. All such objects shall be maintained outside the function image (the instructions that compose the executable representation of a function) on a per-invocation basis. |
| 5.2.4.2.1 Sizes of integer types <limits.h> | magnitude | Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign. |
| 5.2.4.2.2 Characteristics of floating types <float.h> | sign  base or radix  exponent  precision  significand digit |  |
| 5.2.4.2.2 Characteristics of floating types <float.h> | floating-point number |  |
| 5.2.4.2.2 Characteristics of floating types <float.h> | normalized floating-point number  subnormal floating-point  number  unnormalized floating-point number | In addition to normalized floating-point numbers (f 1 > 0 if x ≠ 0), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers (x ≠ 0, e = e min , f 1 = 0) and unnormalized floating-point numbers (x ≠ 0, e > e min , f 1 = 0), and values that are not floating-point numbers, such as infinities and NaNs. |
| 5.2.4.2.2 Characteristics of floating types <float.h> | NaN  quiet NaN  signaling NaN | A NaN is an encoding signifying Not-a-Number. A quiet NaN propagates through almost every arithmetic operation without raising a floating-point exception; a signaling NaN generally raises a floating-point exception when occurring as an arithmetic operand. 22)  22) IEC 60559:1989 specifies quiet and signaling NaNs. For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior. |
| 5.2.4.2.2 Characteristics of floating types <float.h> | The minimum range of representable values for a floating type | The minimum range of representable values for a floating type is the most negative finite floating-point number representable in that type through the most positive finite floating-point number representable in that type. |
| 5.2.4.2.2 Characteristics of floating types <float.h> | minimal-width IEC 60559 double-extended format (64 bits of  precision) |  |
| 6.1 Notation | syntactic categories  nonterminal  literal words and character set members  terminal | In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by italic type, and literal words and character set members (terminals) by bold type. |
| 6.1 Notation |  | A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that  { expressionopt }  indicates an optional expression enclosed in braces. |
| 6.2.1 Scopes of identifiers | identifier | An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. |
| 6.2.1 Scopes of identifiers | enumeration constant | A member of an enumeration is called an enumeration constant. |
| 6.2.1 Scopes of identifiers | visible  scope | For each different entity that an identifier designates, the identifier is visible (i.e., can be used) only within a region of program text called its scope. |
| 6.2.1 Scopes of identifiers | function prototype | A function prototype is a declaration of a function that declares the types of its parameters. |
| 6.2.1 Scopes of identifiers | function scope | A label name is the only kind of identifier that has function scope. |
| 6.2.1 Scopes of identifiers | file scope | If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has file scope, which terminates at the end of the translation unit. |
| 6.2.1 Scopes of identifiers | block scope | If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which terminates at the end of the associated block. |
| 6.2.1 Scopes of identifiers | function prototype scope | If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has function prototype scope, which terminates at the end of the function declarator. |
| 6.2.1 Scopes of identifiers | inner scope  outer scope | If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the inner scope) will end strictly before the scope of the other entity (the outer scope). |
| 6.2.1 Scopes of identifiers | hidden | Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is hidden (and not visible) within the inner scope. |
| 6.2.1 Scopes of identifiers | same scope | Two identifiers have the same scope if and only if their scopes terminate at the same point. |
| 6.2.1 Scopes of identifiers | type name | As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted. |
| 6.2.2 Linkages of identifiers | linkage | An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called linkage. 29)  29) There is no linkage between different identifiers. |
| 6.2.2 Linkages of identifiers | external linkage | In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with external linkage denotes the same object or function. |
| 6.2.2 Linkages of identifiers | internal linkage | Within one translation unit, each declaration of an identifier with internal linkage denotes the same object or function. |
| 6.2.2 Linkages of identifiers | no linkage | Each declaration of an identifier with no linkage denotes a unique entity. |
| 6.2.3 Name spaces of identifiers | name space  label name  tag  member  ordinary identifier | If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate name spaces for various categories of identifiers, as follows:  — label names (disambiguated by the syntax of the label declaration and use);  — the tags of structures, unions, and enumerations (disambiguated by following any 32) of the keywords struct, union, or enum);  — the members of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the . or -> operator);  — all other identifiers, called ordinary identifiers (declared in ordinary declarators or as enumeration constants). |
| 6.2.4 Storage durations of objects | storage duration | An object has a storage duration that determines its lifetime. |
| 6.2.4 Storage durations of objects | lifetime | The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. |
| 6.2.4 Storage durations of objects | static storage duration | An object whose identifier is declared without the storage-class specifier \_Thread\_local, and either with external or internal linkage or with the storage-class specifier static, has static storage duration. |
| 6.2.4 Storage durations of objects | thread storage duration | An object whose identifier is declared with the storage-class specifier \_Thread\_local has thread storage duration. |
| 6.2.4 Storage durations of objects | automatic storage duration | An object whose identifier is declared with no linkage and without the storage-class specifier static has automatic storage duration, as do some compound literals. |
| 6.2.4 Storage durations of objects | temporary lifetime | A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and temporary lifetime. 36)  36) The address of such an object is taken implicitly when an array member is accessed. |
| 6.2.5 Types | type | The meaning of a value stored in an object or returned by a function is determined by the type of the expression used to access it. |
| 6.2.5 Types | object type  function type | Types are partitioned into object types (types that describe objects) and function types (types that describe functions). |
| 6.2.5 Types | incomplete  complete | At various points within a translation unit an object type may be incomplete (lacking sufficient information to determine the size of objects of that type) or complete (having sufficient information). 37)  37) A type may be incomplete or complete throughout an entire translation unit, or it may change states at different points within a translation unit. |
| 6.2.5 Types | char | An object declared as type char is large enough to store any member of the basic execution character set. |
| 6.2.5 Types | standard signed integer type | There are five standard signed integer types, designated as signed char, short int, int, long int, and long long int. |
| 6.2.5 Types | extended signed integer type | There may also be implementation-defined extended signed integer types. 38)  38) Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3. |
| 6.2.5 Types | signed integer types | The standard and extended signed integer types are collectively called signed integer types. 39)  39) Therefore, any statement in this Standard about signed integer types also applies to the extended  signed integer types |
| 6.2.5 Types | a "plain" char object  A "plain" int object | An object declared as type signed char occupies the same amount of storage as a "plain" char object. A "plain" int object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range INT\_MIN to INT\_MAX as defined in the header <limits.h>). |
| 6.2.5 Types | standard unsigned integer type | The type \_Bool and the unsigned integer types that correspond to the standard signed integer types are the standard unsigned integer types. |
| 6.2.5 Types | extended unsigned integer type | The unsigned integer types that correspond to the extended signed integer types are the extended unsigned integer types. |
| 6.2.5 Types | unsigned integer types | The standard and extended unsigned integer types are collectively called unsigned integer types. 40)  40) Therefore, any statement in this Standard about unsigned integer types also applies to the extended unsigned integer types. |
| 6.2.5 Types | standard integer types  extended integer types | The standard signed integer types and standard unsigned integer types are collectively called the standard integer types; the extended signed integer types and extended unsigned integer types are collectively called the extended integer types. |
| 6.2.5 Types | signedness | For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type. |
| 6.2.5 Types | real floating type | There are three real floating types, designated as float, double, and long double. 42)  42) See "future language directions" (6.11.1). |
| 6.2.5 Types | complex type | There are three complex types, designated as float \_Complex, double \_Complex, and long double \_Complex. 43)  43) A specification for imaginary types is in annex G. |
| 6.2.5 Types | floating types | The real floating and complex types are collectively called the floating types. |
| 6.2.5 Types | corresponding real type | For each floating type there is a corresponding real type, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword \_Complex from the type name. |
| 6.2.5 Types | basic types | The type char, the signed and unsigned integer types, and the floating types are collectively called the basic types. |
| 6.2.5 Types | character types | The three types char, signed char, and unsigned char are collectively called the character types. |
| 6.2.5 Types | enumeration | An enumeration comprises a set of named integer constant values. |
| 6.2.5 Types | enumerated type | Each distinct enumeration constitutes a different enumerated type. |
| 6.2.5 Types | integer types | The type char, the signed and unsigned integer types, and the enumerated types are collectively called integer types. |
| 6.2.5 Types | real types | The integer and real floating types are collectively called real types. |
| 6.2.5 Types | arithmetic types | Integer and floating types are collectively called arithmetic types. |
| 6.2.5 Types | type domain  real type domain  complex type domain | Each arithmetic type belongs to one type domain: the real type domain comprises the real types, the complex type domain comprises the complex types. |
| 6.2.5 Types | void type | The void type comprises an empty set of values; it is an incomplete object type that cannot be completed. |
| 6.2.5 Types | derived type  array type  element type  array of T  array type derivation  structure type  union type  function type  return type  function returning T  function type derivation  pointer type  referenced type  pointer to T  pointer type derivation  atomic type | Any number of derived types can be constructed from the object and function types, as follows:  — An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the element type. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is T, the array type is sometimes called "array of T". The construction of an array type from an element type is called "array type derivation".  — A structure type describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.  — A union type describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.  — A function type describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is T, the function type is sometimes called "function returning T". The construction of a function type from a return type is called "function type derivation".  — A pointer type may be derived from a function type or an object type, called the referenced type. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type T is sometimes called "pointer to T". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.  — An atomic type describes the type designated by the construct \_Atomic ( type-name ). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.) |
| 6.2.5 Types | scalar types | Arithmetic types and pointer types are collectively called scalar types. |
| 6.2.5 Types | aggregate types | Array and structure types are collectively called aggregate types. 46)  46) Note that aggregate type does not include union type because an object with union type can only contain one member at a time. |
| 6.2.5 Types | known constant size | A type has known constant size if the type is not incomplete and is not a variable length array type. |
| 6.2.5 Types | derived declarator types | Array, function, and pointer types are collectively called derived declarator types. |
| 6.2.5 Types | declarator type derivation | A declarator type derivation from a type T is the construction of a derived declarator type from T by the application of an array-type, a function-type, or a pointer-type derivation to T. |
| 6.2.5 Types | type category | A type is characterized by its type category, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types. |
| 6.2.5 Types | unqualified type | Any type so far mentioned is an unqualified type. |
| 6.2.5 Types | qualified version | Each unqualified type has several qualified versions of its type, 47) corresponding to the combinations of one, two, or all three of the const, volatile, and restrict qualifiers.  47) See 6.7.3 regarding qualified array and function types. |
| 6.2.5 Types | atomic type | The presence of the \_Atomic qualifier designates an atomic type. |
| 6.2.5 Types | pointer to void | A pointer to void shall have the same representation and alignment requirements as a pointer to a character type. 48)  48) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions. |
| 6.2.6.1 General | pure binary notation | Values stored in unsigned bit-fields and objects of type unsigned char shall be represented using a pure binary notation. 49)  49) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the American National Dictionary for Information Processing Systems.) A byte contains CHAR\_BIT bits, and the values of type unsigned char range from 0 to 2 ^ CHAR\_BIT - 1 . |
| 6.2.6.1 General | object representation | Values stored in non-bit-field objects of any other object type consist of n × CHAR\_BIT bits, where n is the size of an object of that type, in bytes. The value may be copied into an object of type unsigned char [n] (e.g., by memcpy); the resulting set of bytes is called the object representation of the value. |
| 6.2.6.1 General | trap representation | Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined. 50) Such a representation is called a trap representation.  50) Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it. |
| 6.2.6.2 Integer types | value bit  padding bit | For unsigned integer types other than unsigned char, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). |
| 6.2.6.2 Integer types | value representation | If there are N value bits, each bit shall represent a different power of 2 between 1 and 2 ^ (N - 1) , so that objects of that type shall be capable of representing values from 0 to 2 ^ N - 1 using a pure binary representation; this shall be known as the value representation. |
| 6.2.6.2 Integer types | sign bit | For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. |
| 6.2.6.2 Integer types | sign and magnitude  two's complement  ones's complement | If the sign bit is one, the value shall be modified in one of the following ways:  — the corresponding value with sign bit 0 is negated (sign and magnitude);  — the sign bit has the value -(2 ^ M ) (two's complement);  — the sign bit has the value -(2 ^ M - 1) (ones's complement). |
| 6.2.6.2 Integer types | negative zero | Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a negative zero. |
| 6.2.6.2 Integer types | valid | A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. |
| 6.2.6.2 Integer types | precision | The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. |
| 6.2.6.2 Integer types | width | The width of an integer type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision. |
| 6.2.7 Compatible type and composite type | compatible type | Two types have compatible type if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators. 55)  55) Tw o types need not be identical to be compatible. |
| 6.2.7 Compatible type and composite type | composite type | A composite type can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:  ... |
| 6.2.8 Alignment of objects | alignment requirement | Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. |
| 6.2.8 Alignment of objects | alignment | An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. |
| 6.2.8 Alignment of objects | fundamental alignment | A fundamental alignment is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to \_Alignof (max\_align\_t). |
| 6.2.8 Alignment of objects | extended alignment | An extended alignment is represented by an alignment greater than \_Alignof (max\_align\_t). |
| 6.2.8 Alignment of objects | over-aligned type | A type having an extended alignment requirement is an over-aligned type. 57)  57) Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied. |
| 6.2.8 Alignment of objects | weaker  stronger  stricter | Alignments have an order from weaker to stronger or stricter alignments. |
| 6.3 Conversions | implicit conversion  explicit conversion | Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an implicit conversion, as well as those that result from a cast operation (an explicit conversion). |
| 6.3.1.1 Boolean, characters, and integers | integer conversion rank | Every integer type has an integer conversion rank defined as follows:  ... |
| 6.3.1.1 Boolean, characters, and integers | integer promotion | If an int can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an int; otherwise, it is converted to an unsigned int. These are called the integer promotions. 58)  58) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, -, and ~ operators, and to both operands of the shift operators, as specified by their respective subclauses. |
| 6.3.1.7 Real and complex | positive zero  unsigned zero | When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero. |
| 6.3.1.8 Usual arithmetic conversions | common real type | Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. |
| 6.3.1.8 Usual arithmetic conversions | usual arithmetic conversions | This pattern is called the usual arithmetic conversions:  ... |
| 6.3.2.1 Lvalues, arrays, and function designators | lvalue | An lvalue is an expression (with an object type other than void) that potentially designates an object; 64) if an lvalue does not designate an object when it is evaluated, the behavior is undefined.  64) The name "lvalue" comes originally from the assignment expression E1 = E2, in which the left operand E1 is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called "rvalue" is in this International Standard described as the "value of an expression".  An obvious example of an lvalue is an identifier of an object. As a further example, if E is a unary expression that is a pointer to an object, \*E is an lvalue that designates the object to which E points. |
| 6.3.2.1 Lvalues, arrays, and function designators | modifiable lvalue | A modifiable lvalue is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type. |
| 6.3.2.1 Lvalues, arrays, and function designators | lvalue conversion | Except when it is the operand of the sizeof operator, the unary & operator, the ++ operator, the -- operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called lvalue conversion. |
| 6.3.2.1 Lvalues, arrays, and function designators | function designator | A function designator is an expression that has function type. |
| 6.3.2.2 void | void expression | The (nonexistent) value of a void expression (an expression that has type void) shall not be used in any way, and implicit or explicit conversions (except to void) shall not be applied to such an expression. |
| 6.3.2.3 Pointers | pointer to void | A pointer to void may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer. |
| 6.3.2.3 Pointers | null pointer constant | An integer constant expression with the value 0, or such an expression cast to type void \*, is called a null pointer constant. 66)  66) The macro NULL is defined in <stddef.h> (and other headers) as a null pointer constant; see 7.19. |
| 6.3.2.3 Pointers | null pointer | If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function. |
| 6.4 Lexical elements | token  preprocessing-token | token:  　　keyword  　　identifier  　　constant  　　string-literal  　　punctuator  preprocessing-token:  　　header-name  　　identifier  　　pp-number  　　character-constant  　　string-literal  　　punctuator  　　each non-white-space character that cannot be one of the above |
| 6.4 Lexical elements | token | A token is the minimal lexical element of the language in translation phases 7 and 8. |
| 6.4 Lexical elements | preprocessing token | A preprocessing token is the minimal lexical element of the language in translation  phases 3 through 6. |
| 6.4 Lexical elements | placemarker | 69) An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files. |
| 6.4 Lexical elements | white space  white-space characters | Preprocessing tokens can be separated by white space; this consists of comments (described later), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. |
| 6.4.1 Keywords | keyword | keyword: one of   |  |  |  | | --- | --- | --- | | auto  break  case  char  const  continue  default  do  double  else  enum  extern  float  for  goto | if  inline  int  long  register  restrict  return  short  signed  sizeof  static  struct  switch  typedef  union | unsigned  void  volatile  while  \_Alignas  \_Alignof  \_Atomic  \_Bool  \_Complex  \_Generic  \_Imaginary  \_Noreturn  \_Static\_assert  \_Thread\_local | |
| 6.4.2.1 General | identifier  identifier-nondigit  nondigit  digit | identifier:  　　identifier-nondigit  　　identifier identifier-nondigit  　　identifier digit  identifier-nondigit:  　　nondigit  　　universal-character-name  　　other implementation-defined characters  nondigit: one of  　　\_ a b c d e f g h i j k l m  　　n o p q r s t u v w x y z  　　A B C D E F G H I J K L M  　　N O P Q R S T U V W X Y Z  digit: one of  　　0 1 2 3 4 5 6 7 8 9 |
| 6.4.2.1 General | identifier | An identifier is a sequence of nondigit characters (including the underscore \_, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. |
| 6.4.2.1 General | universal character name | Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in D.1. 71)  71) On systems in which linkers cannot accept extended characters, an encoding of the universal character name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the \u in a universal character name. Extended characters may produce a long external identifier. |
| 6.4.2.1 General | multibyte character | An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; which characters and their correspondence to universal character names is implementation-defined. |
| 6.4.2.1 General | external name  internal name | As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; the limit for an external name (an identifier that has external linkage) may be more restrictive than that for an internal name (a macro name or an identifier that does not have external linkage). |
| 6.4.2.2 Predefined identifiers | function-name | The identifier \_\_func\_\_ shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration  static const char \_\_func\_\_[] = "function-name";  appeared, where function-name is the name of the lexically-enclosing function. 72)  72) Since the name \_\_func\_\_ is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name \_\_func\_\_, the behavior is undefined. |
| 6.4.3 Universal character names | universal-character-name  hex-quad | universal-character-name:  　　\u hex-quad  　　\U hex-quad hex-quad  hex-quad:  　　hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit |
| 6.4.4 Constants | constant | constant:  　　integer-constant  　　floating-constant  　　enumeration-constant  　　character-constant |
| 6.4.4.1 Integer constants | integer-constant  decimal-constant  octal-constant  hexadecimal-constant  hexadecimal-prefix  nonzero-digit  octal-digit  hexadecimal-digit  integer-suffix  unsigned-suffix  long-suffix  long-long-suffix | integer-constant:  　　decimal-constant integer-suffixopt  　　octal-constant integer-suffixopt  　　hexadecimal-constant integer-suffixopt  decimal-constant:  　　nonzero-digit  　　decimal-constant digit  octal-constant:  　　0  　　octal-constant octal-digit  hexadecimal-constant:  　　hexadecimal-prefix hexadecimal-digit  　　hexadecimal-constant hexadecimal-digit  hexadecimal-prefix: one of  　　0x 0X  nonzero-digit: one of  　　1 2 3 4 5 6 7 8 9  octal-digit: one of  　　0 1 2 3 4 5 6 7  hexadecimal-digit: one of  　　0 1 2 3 4 5 6 7 8 9  　　a b c d e f  　　A B C D E F  integer-suffix:  　　unsigned-suffix long-suffixopt  　　unsigned-suffix long-long-suffix  　　long-suffix unsigned-suffixopt  　　long-long-suffix unsigned-suffixopt  unsigned-suffix: one of  　　u U  long-suffix: one of  　　l L  long-long-suffix: one of  　　ll LL |
| 6.4.4.2 Floating constants | floating-constant  decimal-floating-constant  hexadecimal-floating-constant  fractional-constant  exponent-part  sign  digit-sequence  hexadecimal-fractional-constant  binary-exponent-part  hexadecimal-digit-sequence  floating-suffix | floating-constant:  　　decimal-floating-constant  　　hexadecimal-floating-constant  decimal-floating-constant:  　　fractional-constant exponent-partopt floating-suffixopt  　　digit-sequence exponent-part floating-suffixopt  hexadecimal-floating-constant:  　　hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffixopt  　　hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-suffixopt  fractional-constant:  　　digit-sequenceopt . digit-sequence  　　digit-sequence .  exponent-part:  　　e signopt digit-sequence  　　E signopt digit-sequence  sign: one of  　　+ -  digit-sequence:  　　digit  　　digit-sequence digit  hexadecimal-fractional-constant:  　　hexadecimal-digit-sequenceopt . hexadecimal-digit-sequence  　　hexadecimal-digit-sequence .  binary-exponent-part:  　　p signopt digit-sequence  　　P signopt digit-sequence  hexadecimal-digit-sequence:  　　hexadecimal-digit  　　hexadecimal-digit-sequence hexadecimal-digit  floating-suffix: one of  　　f l F L |
| 6.4.4.2 Floating constants | significand part  exponent part | A floating constant has a significand part that may be followed by an exponent part and a suffix that specifies its type. |
| 6.4.4.3 Enumeration constants | enumeration-constant | enumeration-constant:  　　identifier |
| 6.4.4.4 Character constants | character-constant  c-char-sequence  c-char  escape-sequence  simple-escape-sequence  octal-escape-sequence  hexadecimal-escape-sequence | character-constant:  　　' c-char-sequence '  　　L' c-char-sequence '  　　u' c-char-sequence '  　　U' c-char-sequence '  c-char-sequence:  　　c-char  　　c-char-sequence c-char  c-char:  　　any member of the source character set except the single-quote ', backslash \, or new-line character  　　escape-sequence  escape-sequence:  　　simple-escape-sequence  　　octal-escape-sequence  　　hexadecimal-escape-sequence  　　universal-character-name  simple-escape-sequence: one of  　　\' \" \? \\  　　\a \b \f \n \r \t \v  octal-escape-sequence:  　　\ octal-digit  　　\ octal-digit octal-digit  　　\ octal-digit octal-digit octal-digit  hexadecimal-escape-sequence:  　　\x hexadecimal-digit  　　hexadecimal-escape-sequence hexadecimal-digit |
| 6.4.4.4 Character constants | integer character constant | An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in 'x'. |
| 6.4.4.4 Character constants | wide character constant | A wide character constant is the same, except prefixed by the letter L, u, or U. |
| 6.4.5 String literals | string-literal  encoding-prefix  s-char-sequence  s-char | string-literal:  　　encoding-prefixopt " s-char-sequenceopt "  encoding-prefix:  　　u8  　　u  　　U  　　L  s-char-sequence:  　　s-char  　　s-char-sequence s-char  s-char:  　　any member of the source character set except the double-quote ", backslash \, or new-line character  　　escape-sequence |
| 6.4.5 String literals | character string literal | A character string literal is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". |
| 6.4.5 String literals | UTF-8 string literal | A UTF-8 string literal is the same, except prefixed by u8. |
| 6.4.5 String literals | wide string literal | A wide string literal is the same, except prefixed by the letter L, u, or U. |
| 6.4.6 Punctuators | punctuator | punctuator: one of  　　[ ] ( ) { } . ->  　　++ -- & \* + - ~ !  　　/ % << >> < > <= >= == != ^ | && ||  　　? : ; ...  　　= \*= /= %= += -= <<= >>= &= ^= |=  　　, # ##  　　<: :> <% %> %: %:%: |
| 6.4.6 Punctuators | punctuator | A punctuator is a symbol that has independent syntactic and semantic significance. |
| 6.4.6 Punctuators | operator | Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an operator (other forms of operator also exist in some contexts). |
| 6.4.6 Punctuators | operand | An operand is an entity on which an operator acts. |
| 6.4.7 Header names | header-name  h-char-sequence  h-char  q-char-sequence  q-char | header-name:  　　< h-char-sequence >  　　" q-char-sequence "  h-char-sequence:  　　h-char  　　h-char-sequence h-char  h-char:  　　any member of the source character set except the new-line character and >  q-char-sequence:  　　q-char  　　q-char-sequence q-char  q-char:  　　any member of the source character set except the new-line character and " |
| 6.4.8 Preprocessing numbers | pp-number | pp-number:  　　digit  　　. digit  　　pp-number digit  　　pp-number identifier-nondigit  　　pp-number e sign  　　pp-number E sign  　　pp-number p sign  　　pp-number P sign  　　pp-number . |
| 6.4.9 Comments | comment | Except within a character constant, a string literal, or a comment, the characters /\* introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters \*/ that terminate it. 83)  83) Thus, /\* ... \*/ comments do not nest.  Except within a character constant, a string literal, or a comment, the characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. |
| 6.5 Expressions | expression | An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. |
| 6.5 Expressions | bitwise operators | Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as bitwise operators) are required to have operands that have integer type. |
| 6.5 Expressions | exceptional condition | If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined. |
| 6.5 Expressions | effective type | The effective type of an object for an access to its stored value is the declared type of the object, if any. 87) If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.  87) Allocated objects have no declared type. |
| 6.5 Expressions | contracted | A floating expression may be contracted, that is, evaluated as though it were a single operation, thereby omitting rounding errors implied by the source code and the expression evaluation method. 89)  89) The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the final operation is rounded to the format determined by the expression evaluation method. A contracted expression might also omit the raising of floating-point exceptions. |
| 6.5.1 Primary expressions | primary-expression | primary-expression:  　　identifier  　　constant  　　string-literal  　　( expression )  　　generic-selection |
| 6.5.1.1 Generic selection | generic-selection  generic-assoc-list  generic-association | generic-selection:  　　\_Generic ( assignment-expression , generic-assoc-list )  generic-assoc-list:  　　generic-association  　　generic-assoc-list , generic-association  generic-association:  　　type-name : assignment-expression  　　default : assignment-expression |
| 6.5.1.1 Generic selection | controlling expression | The controlling expression of a generic selection shall have type compatible with at most one of the types named in its generic association list. |
| 6.5.2 Postfix operators | postfix-expression  argument-expression-list | postfix-expression:  　　primary-expression  　　postfix-expression [ expression ]  　　postfix-expression ( argument-expression-listopt )  　　postfix-expression . identifier  　　postfix-expression -> identifier  　　postfix-expression ++  　　postfix-expression --  　　( type-name ) { initializer-list }  　　( type-name ) { initializer-list , }  argument-expression-list:  　　assignment-expression  　　argument-expression-list , assignment-expression |
| 6.5.2.1 Array subscripting | row-major order | It follows from this that arrays are stored in row-major order (last subscript varies fastest). |
| 6.5.2.2 Function calls | function call | A postfix expression followed by parentheses () containing a possibly empty, comma-separated list of expressions is a function call. |
| 6.5.2.2 Function calls | called function | The postfix expression denotes the called function. |
| 6.5.2.2 Function calls | arguments | The list of expressions specifies the arguments to the function. |
| 6.5.2.2 Function calls | default argument promotion | If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type float are promoted to double. These are called the default argument promotions. |
| 6.5.2.3 Structure and union members | common initial sequence | Tw o structures share a common initial sequence if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members. |
| 6.5.2.4 Postfix increment and decrement operators | increment | As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). |
| 6.5.2.4 Postfix increment and decrement operators | single evaluation | With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation. |
| 6.5.2.4 Postfix increment and decrement operators | decrement | The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it). |
| 6.5.2.5 Compound literals | compound literal | A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a compound literal. |
| 6.5.3 Unary operators | unary-expression  unary-operator | unary-expression:  　　postfix-expression  　　++ unary-expression  　　-- unary-expression  　　unary-operator cast-expression  　　sizeof unary-expression  　　sizeof ( type-name )  　　\_Alignof ( type-name )  unary-operator: one of  　　& \* + - ~ ! |
| 6.5.3.3 Unary arithmetic operators | bitwise complement | The result of the ~ operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). |
| 6.5.4 Cast operators | cast-expression | cast-expression:  　　unary-expression  　　( type-name ) cast-expression |
| 6.5.4 Cast operators | cast | Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a cast. 104)  104) A cast does not yield an lvalue. Thus, a cast to a qualified type has the same effect as a cast to the unqualified version of the type. |
| 6.5.5 Multiplicative operators | multiplicative-expression | multiplicative-expression:  　　cast-expression  　　multiplicative-expression \* cast-expression  　　multiplicative-expression / cast-expression  　　multiplicative-expression % cast-expression |
| 6.5.6 Additive operators | additive-expression | additive-expression:  　　multiplicative-expression  　　additive-expression + multiplicative-expression  　　additive-expression - multiplicative-expression |
| 6.5.7 Bitwise shift operators | shift-expression | shift-expression:  　　additive-expression  　　shift-expression << additive-expression  　　shift-expression >> additive-expression |
| 6.5.8 Relational operators | relational-expression | relational-expression:  　　shift-expression  　　relational-expression < shift-expression  　　relational-expression > shift-expression  　　relational-expression <= shift-expression  　　relational-expression >= shift-expression |
| 6.5.9 Equality operators | equality-expression | equality-expression:  　　relational-expression  　　equality-expression == relational-expression  　　equality-expression != relational-expression |
| 6.5.10 Bitwise AND operator | AND-expression | AND-expression:  　　equality-expression  　　AND-expression & equality-expression |
| 6.5.10 Bitwise AND operator | bitwise AND | The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set). |
| 6.5.11 Bitwise exclusive OR operator | exclusive-OR-expression | exclusive-OR-expression:  　　AND-expression  　　exclusive-OR-expression ^ AND-expression |
| 6.5.11 Bitwise exclusive OR operator | bitwise exclusive OR | The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set). |
| 6.5.12 Bitwise inclusive OR operator | inclusive-OR-expression | inclusive-OR-expression:  　　exclusive-OR-expression  　　inclusive-OR-expression | exclusive-OR-expression |
| 6.5.12 Bitwise inclusive OR operator | bitwise inclusive OR | The result of the | operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set). |
| 6.5.13 Logical AND operator | logical-AND-expression | logical-AND-expression:  　　inclusive-OR-expression  　　logical-AND-expression && inclusive-OR-expression |
| 6.5.13 Logical AND operator | left-to-right evaluation | Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated. |
| 6.5.14 Logical OR operator | logical-OR-expression | logical-OR-expression:  　　logical-AND-expression  　　logical-OR-expression || logical-AND-expression |
| 6.5.14 Logical OR operator | left-to-right evaluation | Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated. |
| 6.5.15 Conditional operator | conditional-expression | conditional-expression:  　　logical-OR-expression  　　logical-OR-expression ? expression : conditional-expression |
| 6.5.16 Assignment operators | assignment-expression  assignment-operator | assignment-expression:  　　conditional-expression  　　unary-expression assignment-operator assignment-expression  assignment-operator: one of  　　= \*= /= %= += -= <<= >>= &= ^= |= |
| 6.5.16.1 Simple assignment | simple assignment | In simple assignment (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand. |
| 6.5.16.2 Compound assignment | compound assignment  single evaluation | A compound assignment of the form E1 op= E2 is equivalent to the simple assignment expression E1 = E1 op (E2), except that the lvalue E1 is evaluated only once, and with respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. |
| 6.5.17 Comma operator | expression | expression:  　　assignment-expression  　　expression , assignment-expression |
| 6.6 Constant expressions | constant-expression | constant-expression:  　　conditional-expression |
| 6.6 Constant expressions | integer constant expression | An integer constant expression 117) shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, \_Alignof expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the sizeof or \_Alignof operator.  117) An integer constant expression is required in a number of contexts such as the size of a bit-field member of a structure, the value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1. |
| 6.6 Constant expressions | arithmetic constant expression | An arithmetic constant expression shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, and \_Alignof expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a sizeof or \_Alignof operator. |
| 6.6 Constant expressions | address constant | An address constant is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary & operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. The array-subscript [] and member-access . and -> operators, the address & and indirection \* unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators. |
| 6.7 Declarations | declaration  declaration-specifiers  init-declarator-list  init-declarator | declaration:  　　declaration-specifiers init-declarator-listopt ;  　　static\_assert-declaration  declaration-specifiers:  　　storage-class-specifier declaration-specifiersopt  　　type-specifier declaration-specifiersopt  　　type-qualifier declaration-specifiersopt  　　function-specifier declaration-specifiersopt  　　alignment-specifier declaration-specifiersopt  init-declarator-list:  　　init-declarator  　　init-declarator-list , init-declarator  init-declarator:  　　declarator  　　declarator = initializer |
| 6.7 Declarations | definition | A declaration specifies the interpretation and attributes of a set of identifiers. A definition of an identifier is a declaration for that identifier that:  — for an object, causes storage to be reserved for that object;  — for a function, includes the function body; 119)  — for an enumeration constant, is the (only) declaration of the identifier;  — for a typedef name, is the first (or only) declaration of the identifier.  119) Function definitions have a different syntax, described in 6.9.1. |
| 6.7.1 Storage-class specifiers | storage-class-specifier | storage-class-specifier:  　　typedef  　　extern  　　static  　　\_Thread\_local  　　auto  　　register |
| 6.7.2 Type specifiers | type-specifier | type-specifier:  　　void  　　char  　　short  　　int  　　long  　　float  　　double  　　signed  　　unsigned  　　\_Bool  　　\_Complex  　　atomic-type-specifier  　　struct-or-union-specifier  　　enum-specifier  　　typedef-name |
| 6.7.2.1 Structure and union specifiers | struct-or-union-specifier  struct-or-union  struct-declaration-list  struct-declaration  specifier-qualifier-list  struct-declarator-list  struct-declarator | struct-or-union-specifier:  　　struct-or-union identifieropt { struct-declaration-list }  　　struct-or-union identifier  struct-or-union:  　　struct  　　union  struct-declaration-list:  　　struct-declaration  　　struct-declaration-list struct-declaration  struct-declaration:  　　specifier-qualifier-list struct-declarator-listopt ;  　　static\_assert-declaration  specifier-qualifier-list:  　　type-specifier specifier-qualifier-listopt  　　type-qualifier specifier-qualifier-listopt  struct-declarator-list:  　　struct-declarator  　　struct-declarator-list , struct-declarator  struct-declarator:  　　declarator  　　declaratoropt : constant-expression |
| 6.7.2.1 Structure and union specifiers | bit-field | A member of a structure or union may have any complete object type other than a variably modified type. 123) In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field; 124) its width is preceded by a colon.  123) A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.  124) The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects. |
| 6.7.2.1 Structure and union specifiers | unnamed bit-field | A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field. 126)  126) An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts. |
| 6.7.2.1 Structure and union specifiers | anonymous structure  anonymous union | An unnamed member whose type specifier is a structure specifier with no tag is called an anonymous structure; an unnamed member whose type specifier is a union specifier with no tag is called an anonymous union. |
| 6.7.2.1 Structure and union specifiers | flexible array member | As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a flexible array member. |
| 6.7.2.2 Enumeration specifiers | enum-specifier  enumerator-list  enumerator | enum-specifier:  　　enum identifieropt { enumerator-list }  　　enum identifieropt { enumerator-list , }  　　enum identifier  enumerator-list:  　　enumerator  　　enumerator-list , enumerator  enumerator:  　　enumeration-constant  　　enumeration-constant = constant-expression |
| 6.7.2.2 Enumeration specifiers | member | The enumerators of an enumeration are also known as its members. |
| 6.7.2.3 Tags | structure content  union content  enumeration content | A type specifier of the form  struct-or-union identifieropt { struct-declaration-list }  or  enum identifieropt { enumerator-list }  or  enum identifieropt { enumerator-list , }  declares a structure, union, or enumerated type. The list defines the structure content, union content, or enumeration content. |
| 6.7.2.4 Atomic type specifiers | atomic-type-specifier | atomic-type-specifier:  　　\_Atomic ( type-name ) |
| 6.7.3 Type qualifiers | type-qualifier | type-qualifier:  　　const  　　restrict  　　volatile  　　\_Atomic |
| 6.7.3 Type qualifiers | meaning | The intended use of the restrict qualifier (like the register storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior). |
| 6.7.3.1 Formal definition of restrict | restrict | ... |
| 6.7.3.1 Formal definition of restrict | based | In what follows, a pointer expression E is said to be based on object P if (at some sequence point in the execution of B prior to the evaluation of E) modifying P to point to a copy of the array object into which it formerly pointed would change the value of E. 137)  137) In other words, E depends on the value of P itself rather than on the value of an object referenced indirectly through P. For example, if identifier p has type (int \*\*restrict), then the pointer expressions p and p+1 are based on the restricted pointer object designated by p, but the pointer expressions \*p and p[1] are not. |
| 6.7.3.1 Formal definition of restrict | execution | Here an execution of B means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with B. |
| 6.7.3.1 Formal definition of restrict | effective dependence analysis |  |
| 6.7.3.1 Formal definition of restrict | equivalent nested block |  |
| 6.7.4 Function specifiers | function-specifier | function-specifier:  　　inline  　　\_Noreturn |
| 6.7.4 Function specifiers | inline function | A function declared with an inline function specifier is an inline function. |
| 6.7.4 Function specifiers | inline substitution |  |
| 6.7.4 Function specifiers | inline definition | If all of the file scope declarations for a function in a translation unit include the inline function specifier without extern, then the definition in that translation unit is an inline definition. |
| 6.7.5 Alignment specifier | alignment-specifier | alignment-specifier:  　　\_Alignas ( type-name )  　　\_Alignas ( constant-expression ) |
| 6.7.6 Declarators | declarator  direct-declarator  pointer  type-qualifier-list  parameter-type-list  parameter-list  parameter-declaration  identifier-list | declarator:  　　pointeropt direct-declarator  direct-declarator:  　　identifier  　　( declarator )  　　direct-declarator [ type-qualifier-listopt assignment-expressionopt ]  　　direct-declarator [ static type-qualifier-listopt assignment-expression ]  　　direct-declarator [ type-qualifier-list static assignment-expression ]  　　direct-declarator [ type-qualifier-listopt \* ]  　　direct-declarator ( parameter-type-list )  　　direct-declarator ( identifier-listopt )  pointer:  　　\* type-qualifier-listopt  　　\* type-qualifier-listopt pointer  type-qualifier-list:  　　type-qualifier  　　type-qualifier-list type-qualifier  parameter-type-list:  　　parameter-list  　　parameter-list , ...  parameter-list:  　　parameter-declaration  　　parameter-list , parameter-declaration  parameter-declaration:  　　declaration-specifiers declarator  　　declaration-specifiers abstract-declaratoropt  identifier-list:  　　identifier  　　identifier-list , identifier |
| 6.7.6 Declarators | declarator | Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers. |
| 6.7.6 Declarators | full declarator | A full declarator is a declarator that is not part of another declarator. |
| 6.7.6 Declarators | variably modified | If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be variably modified. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified. |
| 6.7.6.2 Array declarators | variable length array | If the size is \* instead of being an expression, the array type is a variable length array type of unspecified size, which can only be used in declarations or type names with function prototype scope; 143) such arrays are nonetheless complete types. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a variable length array type.  143) Thus, \* can be used only in function declarations that are not definitions (see 6.7.6.3). |
| 6.7.6.2 Array declarators | size specifier |  |
| 6.7.6.2 Array declarators | variably modified (VM) type |  |
| 6.7.7 Type names | type-name  abstract-declarator  direct-abstract-declarator | type-name:  　　specifier-qualifier-list abstract-declaratoropt  abstract-declarator:  　　pointer  　　pointeropt direct-abstract-declarator  direct-abstract-declarator:  　　( abstract-declarator )  　　direct-abstract-declaratoropt [ type-qualifier-listopt assignment-expressionopt ]  　　direct-abstract-declaratoropt [ static type-qualifier-listopt assignment-expression ]  　　direct-abstract-declaratoropt [ type-qualifier-list static assignment-expression ]  　　direct-abstract-declaratoropt [ \* ]  　　direct-abstract-declaratoropt ( parameter-type-listopt ) |
| 6.7.7 Type names | type name | In several contexts, it is necessary to specify a type. This is accomplished using a type name, which is syntactically a declaration for a function or an object of that type that omits the identifier. 147)  147) As indicated by the syntax, empty parentheses in a type name are interpreted as "function with no parameter specification", rather than redundant parentheses around the omitted identifier. |
| 6.7.8 Type definitions | typedef-name | typedef-name:  　　identifier |
|  | typedef name | In a declaration whose storage-class specifier is typedef, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.6. |
| 6.7.9 Initialization | initializer  initializer-list  designation  designator-list  designator | initializer:  　　assignment-expression  　　{ initializer-list }  　　{ initializer-list , }  initializer-list:  　　designationopt initializer  　　initializer-list , designationopt initializer  designation:  　　designator-list =  designator-list:  　　designator  　　designator-list designator  designator:  　　[ constant-expression ]  　　. identifier |
| 6.7.9 Initialization | current object | Each brace-enclosed initializer list has an associated current object. |
| 6.7.9 Initialization | fully bracketed initialization |  |
| 6.7.9 Initialization | inconsistently bracketed initialization |  |
| 6.7.9 Initialization | incompletely but consistently bracketed initialization |  |
| 6.7.9 Initialization | fully bracketed form |  |
| 6.7.9 Initialization | minimally bracketed form |  |
| 6.7.9 Initialization | unadorned initializer list |  |
| 6.7.10 Static assertions | static\_assert-declaration | static\_assert-declaration:  　　\_Static\_assert ( constant-expression , string-literal ) ; |
| 6.8 Statements and blocks | statement | statement:  　　labeled-statement  　　compound-statement  　　expression-statement  　　selection-statement  　　iteration-statement  　　jump-statement |
| 6.8 Statements and blocks | statement | A statement specifies an action to be performed. |
| 6.8 Statements and blocks | block | A block allows a set of declarations and statements to be grouped into one syntactic unit. |
| 6.8 Statements and blocks | full expression | A full expression is an expression that is not part of another expression or of a declarator. |
| 6.8.1 Labeled statements | labeled-statement | labeled-statement:  　　identifier : statement  　　case constant-expression : statement  　　default : statement |
| 6.8.2 Compound statement | compound-statement  block-item-list  block-item | compound-statement:  　　{ block-item-listopt }  block-item-list:  　　block-item  　　block-item-list block-item  block-item:  　　declaration  　　statement |
| 6.8.2 Compound statement | compound statement | A compound statement is a block. |
| 6.8.3 Expression and null statements | expression-statement | expression-statement:  　　expressionopt ; |
| 6.8.3 Expression and null statements | null statement | A null statement (consisting of just a semicolon) performs no operations. |
| 6.8.3 Expression and null statements | empty loop body |  |
| 6.8.4 Selection statements | selection-statement | selection-statement:  　　if ( expression ) statement  　　if ( expression ) statement else statement  　　switch ( expression ) statement |
| 6.8.4 Selection statements | controlling expression | A selection statement selects among a set of statements depending on the value of a controlling expression. |
| 6.8.4 Selection statements | substatement |  |
| 6.8.4.2 The switch statement | switch body | A switch statement causes control to jump to, into, or past the statement that is the switch body, depending on the value of a controlling expression, and on the presence of a default label and the values of any case labels on or in the switch body. |
| 6.8.5 Iteration statements | iteration-statement | iteration-statement:  　　while ( expression ) statement  　　do statement while ( expression ) ;  　　for ( expressionopt ; expressionopt ; expressionopt ) statement  　　for ( declaration expressionopt ; expressionopt ) statement |
| 6.8.5 Iteration statements | loop body | An iteration statement causes a statement called the loop body to be executed repeatedly until the controlling expression compares equal to 0. |
| 6.8.5 Iteration statements | empty loop |  |
| 6.8.6 Jump statements | jump-statement | jump-statement:  　　goto identifier ;  　　continue ;  　　break ;  　　return expressionopt ; |
| 6.8.6 Jump statements | unconditional jump |  |
| 6.8.6.2 The continue statement | smallest enclosing iteration statement |  |
| 6.8.6.3 The break statement | smallest enclosing switch statement |  |
| 6.9 External definitions | translation-unit  external-declaration | translation-unit:  　　external-declaration  　　translation-unit external-declaration  external-declaration:  　　function-definition  　　declaration |
| 6.9 External definitions | external declaration | As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). |
| 6.9 External definitions | external definition | An external definition is an external declaration that is also a definition of a function (other than an inline definition) or an object. |
| 6.9.1 Function definitions | function-definition  declaration-list | function-definition:  　　declaration-specifiers declarator declaration-listopt compound-statement  declaration-list:  　　declaration  　　declaration-list declaration |
| 6.9.1 Function definitions | parameter identifier |  |
| 6.9.1 Function definitions | function body |  |
| 6.9.2 External object definitions | external definition | If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier. |
| 6.9.2 External object definitions | tentative definition | A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier static, constitutes a tentative definition. |
| 6.10 Preprocessing directives | preprocessing-file  group  group-part  if-section  if-group  elif-groups  elif-group  else-group  endif-line  control-line  text-line  non-directive  lparen  replacement-list  pp-tokens  new-line | preprocessing-file:  　　groupopt  group:  　　group-part  　　group group-part  group-part:  　　if-section  　　control-line  　　text-line  　　# non-directive  if-section:  　　if-group elif-groupsopt else-groupopt endif-line  if-group:  　　# if constant-expression new-line groupopt  　　# ifdef identifier new-line groupopt  　　# ifndef identifier new-line groupopt  elif-groups:  　　elif-group  　　elif-groups elif-group  elif-group:  　　# elif constant-expression new-line groupopt  else-group:  　　# else new-line groupopt  endif-line:  　　# endif new-line  control-line:  　　# include pp-tokens new-line  　　# define identifier replacement-list new-line  　　# define identifier lparen identifier-listopt ) replacement-list new-line  　　# define identifier lparen ... ) replacement-list new-line  　　# define identifier lparen identifier-list , ... ) replacement-list new-line  　　# undef identifier new-line  　　# line pp-tokens new-line  　　# error pp-tokensopt new-line  　　# pragma pp-tokensopt new-line  　　# new-line  text-line:  　　pp-tokensopt new-line  non-directive:  　　pp-tokens new-line  lparen:  　　a ( character not immediately preceded by white-space  replacement-list:  　　pp-tokensopt  pp-tokens:  　　preprocessing-token  　　pp-tokens preprocessing-token  new-line:  　　the new-line character |
| 6.10 Preprocessing directives | preprocessing directive | A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence. 165) A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.  165) Thus, preprocessing directives are commonly called "lines". These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 6.10.3.2, for example). |
| 6.10 Preprocessing directives | character string literal creation operator |  |
| 6.10 Preprocessing directives | directive name |  |
| 6.10 Preprocessing directives | preprocessing | The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called preprocessing, because conceptually they occur before translation of the resulting translation unit. |
| 6.10.1 Conditional inclusion | defined | The expression that controls conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below; 166) and it may contain unary operator expressions of the form  defined identifier  or  defined ( identifier )  which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a #define preprocessing directive without an intervening #undef directive with the same subject identifier), 0 if it is not.  166) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc. |
| 6.10.1 Conditional inclusion | false | Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. |
| 6.10.1 Conditional inclusion | true | Only the first group whose control condition evaluates to true (nonzero) is processed. |
| 6.10.3 Macro replacement | replacement list |  |
| 6.10.3 Macro replacement | white-space separation |  |
| 6.10.3 Macro replacement | parameter identifier |  |
| 6.10.3 Macro replacement | macro name | The identifier immediately following the define is called the macro name. |
| 6.10.3 Macro replacement | object-like macro | A preprocessing directive of the form  # define identifier replacement-list new-line  defines an object-like macro that causes each subsequent instance of the macro name 171) to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.  171) Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters. |
| 6.10.3 Macro replacement | function-like macro | A preprocessing directive of the form  # define identifier lparen identifier-list opt ) replacement-list new-line  # define identifier lparen ... ) replacement-list new-line  # define identifier lparen identifier-list , ... ) replacement-list new-line  defines a function-like macro with parameters, whose use is similar syntactically to a function call. |
| 6.10.3 Macro replacement | an invocation of the macro | Each subsequent instance of the function-like macro name followed by a ( as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). |
| 6.10.3 Macro replacement | variable arguments | If there is a ... in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the variable arguments. |
| 6.10.3.1 Argument substitution | \_\_VA\_ARGS\_\_ | An identifier \_\_VA\_ARGS\_\_ that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it. |
| 6.10.3.3 The ## operator | placemarker | If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead. 173)  173) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4. |
| 6.10.3.5 Scope of macro definitions | manifest constant |  |
| 6.10.3.5 Scope of macro definitions | in-line code |  |
| 6.10.4 Line control | line number | The line number of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token. |
| 6.10.7 Null directive |  |  |
| 6.10.8 Predefined macro names | \_\_cplusplus | The implementation shall not predefine the macro \_\_cplusplus, nor shall it define it in any standard header. |
| 6.10.8.1 Mandatory macros | \_\_DATE\_\_  \_\_FILE\_ \_  \_\_LINE\_ \_  \_\_STDC\_ \_  \_\_STDC\_HOSTED\_\_  \_\_STDC\_VERSION\_\_  \_\_TIME\_\_ | ... |
| 6.10.8.2 Environment macros | \_\_STDC\_ISO\_10646\_\_ | ... |
| 6.10.8.2 Environment macros | Unicode required set | The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. |
| 6.10.8.2 Environment macros | \_\_STDC\_MB\_MIGHT\_NEQ\_WC\_\_  \_\_STDC\_UTF\_16\_\_  \_\_STDC\_UTF\_32\_\_ |  |
| 6.10.8.3 Conditional feature macros | \_\_STDC\_ANALYZABLE\_\_  \_\_STDC\_IEC\_559\_\_  \_\_STDC\_IEC\_559\_COMPLEX\_\_  \_\_STDC\_LIB\_EXT1\_\_  \_\_STDC\_NO\_ATOMICS\_\_  \_\_STDC\_NO\_COMPLEX\_\_  \_\_STDC\_NO\_THREADS\_\_  \_\_STDC\_NO\_VLA\_\_ |  |
| 6.10.9 Pragma operator | \_Pragma  destringized | A unary operator expression of the form:  \_Pragma ( string-literal )  is processed as follows: The string literal is destringized by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \\ by a single backslash. |

errors

|  |
| --- |
| 5.2.1.1 Trigraph sequences |
| Each ? that does not begin one of the trigraphs listed above is not changed. |

|  |  |
| --- | --- |
| 6.3.2.1 Lvalues, arrays, and function designators | 6.5.1 Primary expressions |
| Except when it is the operand of the sizeof operator, the unary & operator, the ++ operator, the -- operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called lvalue conversion. | A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression. |

|  |
| --- |
| 6.7.2.1 Structure and union specifiers |
| There may be unnamed padding within a structure object, but not at its beginning. |

|  |
| --- |
| 6.7.6 Declarators |
| A full declarator is a declarator that is not part of another declarator. The end of a full declarator is a sequence point. If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be variably modified. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified. |

|  |
| --- |
| 6.10.3 Macro replacement |
| If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the ...). There shall exist a ) preprocessing token that terminates the invocation. |

|  |
| --- |
| 6.10.3.3 The ## operator |
| The resulting token is available for further macro replacement. |

implementation-defined behaviours