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 | 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 Tw o 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  digit  graphic character  control 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. |
| 6.2.1 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 | Tw o 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. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

implementation-defined behaviours