definitions

|  |  |  |
| --- | --- | --- |
| 1.1 Scope | C++  C standard | C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1999 Programming languages — C (hereinafter referred to as the C standard). |
| 1.3.1 argument | argument | argument  actual argument  actual parameter  <function call expression> expression in the comma-separated list bounded by the parentheses |
| 1.3.2 argument | argument | argument  actual argument  actual parameter  <function-like macro> sequence of preprocessing tokens in the comma-separated list bounded by the parentheses |
| 1.3.3 argument | argument | argument  actual argument  actual parameter  <throw expression> the operand of throw |
| 1.3.4 argument | argument | argument  actual argument  actual parameter  <template instantiation> expression, type-id or template-name in the comma-separated list bounded by the angle brackets |
| 1.3.5 conditionally-supported | conditionally-supported | conditionally-supported  program construct that an implementation is not required to support  [Note: Each implementation documents all conditionally-supported constructs that it does not support.—end note ] |
| 1.3.6 diagnostic message | diagnostic message | diagnostic message  message belonging to an implementation-defined subset of the implementation’s output messages |
| 1.3.7 dynamic type | dynamic type | dynamic type  <glvalue> type of the most derived object (1.8) to which the glvalue denoted by a glvalue expression refers  [Example: if a pointer (8.3.1) p whose static type is “pointer to class B” is pointing to an object of class D, derived from B (Clause 10), the dynamic type of the expression \*p is “D.” References (8.3.2) are treated similarly. —end example ] |
| 1.3.9 ill-formed program | ill-formed program | ill-formed program  program that is not well formed |
| 1.3.10 implementation-defined behavior | implementation-defined behavior | implementation-defined behavior  behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents |
| 1.3.11 implementation limits | implementation limits | implementation limits  restrictions imposed upon programs by the implementation |
| 1.3.12 locale-specific behavior | locale-specific behavior | locale-specific behavior  behavior that depends on local conventions of nationality, culture, and language that each implementation documents |
| 1.3.13 multibyte character | multibyte character | 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 (2.3). —end note ] |
| 1.3.14 parameter | parameter | parameter  formal argument  formal parameter  <function or catch clause> object or reference declared as part of a function declaration or definition or in the catch clause of an exception handler that acquires a value on entry to the function or handler |
| 1.3.15 parameter | parameter | parameter  formal argument  formal parameter  <function-like macro> identifier from the comma-separated list bounded by the parentheses immediately following the macro name |
| 1.3.16 parameter | parameter | parameter  formal argument  formal parameter  <template> template-parameter |
| 1.3.17 signature | signature | signature  <function> name, parameter type list (8.3.5), and enclosing namespace (if any)  [Note: Signatures are used as a basis for name mangling and linking.—end note ] |
| 1.3.18 signature | signature | signature  <function template> name, parameter type list (8.3.5), enclosing namespace (if any), return type, and template parameter list |
| 1.3.19 signature | signature | signature  <function template specialization> signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced) |
| 1.3.20 signature | signature | signature  <class member function> name, parameter type list (8.3.5), class of which the function is a member, cv-qualifiers (if any), and ref-qualifier (if any) |
| 1.3.21 signature | signature | signature  <class member function template> name, parameter type list (8.3.5), class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), return type, and template parameter list |
| 1.3.22 signature | signature | signature  <class member function template specialization> signature of the member function template of which it is a specialization and its template arguments (whether explicitly specified or deduced) |
| 1.3.23 static type | static type | static type  type of an expression (3.9) resulting from analysis of the program without considering execution semantics  [Note: The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing. —end note ] |
| 1.3.24 undefined behavior | undefined behavior | undefined behavior  behavior for which this International Standard imposes no requirements  [Note: Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible 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). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. —end note ] |
| 1.3.25 unspecified behavior | unspecified behavior | unspecified behavior  behavior, for a well-formed program construct and correct data, that depends on the implementation  [Note: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard. —end note ] |
| 1.3.26 well-formed program | well-formed program | C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2). |
| 1.4 Implementation compliance | diagnosable rules | The set of diagnosable rules consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that “no diagnostic is required” or which are described as resulting in "undefined behavior." |
| 1.4 Implementation compliance | conforming implementation |  |
| 1.4 Implementation compliance | hosted implementation  freestanding implementation | Two kinds of implementations are defined: a hosted implementation and a freestanding implementation. |
| 1.6 Syntax notation | syntactic categories  literal words and characters | In the syntax notation used in this International Standard, syntactic categories are indicated by italic type, and literal words and characters in constant width type. |
| 1.6 Syntax notation | terminal  non-terminal | Alternatives are listed on separate lines except in a few cases where a long set of alternatives is marked by the phrase "one of." If the text of an alternative is too long to fit on a line, the text is continued on subsequent lines indented from the first one. An optional terminal or non-terminal symbol is indicated by the subscript " opt ", so  { expressionopt }  indicates an optional expression enclosed in braces. |
| 1.6 Syntax notation | -name  -id  -seq  -list | Names for syntactic categories have generally been chosen according to the following rules:  — X-name is a use of an identifier in a context that determines its meaning (e.g., class-name, typedef-name).  — X-id is an identifier with no context-dependent meaning (e.g., qualified-id).  — X-seq is one or more X's without intervening delimiters (e.g., declaration-seq is a sequence of declarations).  — X-list is one or more X's separated by intervening commas (e.g., expression-list is a sequence of expressions separated by commas). |
| 1.7 The C++ memory model | byte | The fundamental storage unit in the C ++ memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. |
| 1.7 The C++ memory model | low-order bit  high-order bit | The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. |
| 1.7 The C++ memory model | memory location | A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [Note: Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. —end note ] |
| 1.8 The C++ object model | create  destroy  refer to  access  manipulate | The constructs in a C ++ program create, destroy, refer to, access, and manipulate objects. |
| 1.8 The C++ object model | object | An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. —end note ] |
| 1.8 The C++ object model | properties | The properties of an object are determined when the object is created. |
| 1.8 The C++ object model | object type | The term object type refers to the type with which the object is created. |
| 1.8 The C++ object model | subobject | Objects can contain other objects, called subobjects. |
| 1.8 The C++ object model | complete object | An object that is not a subobject of any other object is called a complete object. |
| 1.8 The C++ object model | complete object of x | For every object x, there is some object called the complete object of x, determined as follows:  — If x is a complete object, then x is the complete object of x.  — Otherwise, the complete object of x is the complete object of the (unique) object that contains x. |
| 1.8 The C++ object model | most derived class  most derived object | If a complete object, a data member (9.2), or an array element is of class type, its type is considered the most derived class, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a most derived object. |
| 1.9 Program execution | "as-if" rule | Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below. 5  5) This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced. |
| 1.9 Program execution | parameters of the abstract machine | Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, sizeof(int)). These constitute the parameters of the abstract machine. |
| 1.9 Program execution | corresponding instance | Each implementation shall include documentation describing its characteristics and behavior in these respects. 6 Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).  6) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.4. |
| 1.9 Program execution | allowable behaviors | Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. |
| 1.9 Program execution | conforming implementation | A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. |
| 1.9 Program execution | instance  suspended | An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal). |
| 1.9 Program execution | observable behavior | The least requirements on a conforming implementation are:  — Access 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 one of the possible results that execution of the program according to the abstract semantics would have produced.  — The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.  These collectively are referred to as the observable behavior of the program. [Note: More stringent correspondences between abstract and actual semantics may be defined by each implementation. —end note ] |
| 1.9 Program execution | full-expression  implicit full-expression | A full-expression is an expression that is not a subexpression of another expression. [Note: in some contexts, such as unevaluated operands, a syntactic subexpression is considered a full-expression (Clause 5). —end note ] If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression. |
| 1.9 Program execution | side effect | Accessing an object designated by a volatile glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. |
| 1.9 Program execution | evaluation  glvalue evaluation  prvalue evaluation | Evaluation of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. |
| 1.9 Program execution | sequenced before | Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. |
| 1.9 Program execution | unsequenced | If A is not sequenced before B and B is not sequenced before A, then A and B are unsequenced. [Note: The execution of unsequenced evaluations can overlap. —end note ] |
| 1.9 Program execution | indeterminately sequenced | Evaluations A and B are indeterminately sequenced when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [Note: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. —end note ] |
| 1.9 Program execution | 没有出现sequence point |  |
| 1.9 Program execution | sequencing constraint | The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be. |
| 1.10 Multi-threaded executions and data races | thread of execution  thread | A thread of execution (also known as a thread) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [Note: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. —end note ] |
|  | execution | The execution of the entire program consists of an execution of all of its threads. [Note: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. —end note ] |
|  | lock-free execution | Executions of atomic functions that are either defined to be lock-free (29.7) or indicated as lock-free (29.4) are lock-free executions. |
|  | obstruction-free | — If there is only one unblocked thread, a lock-free execution in that thread shall complete. [Note: Concurrently executing threads may prevent progress of a lock-free execution. For example, this situation can occur with load-locked store-conditional implementations. This property is sometimes termed obstruction-free. —end note ] |
|  | lock-free | — When one or more lock-free executions run concurrently, at least one should complete. [Note: It is difficult for some implementations to provide absolute guarantees to this effect, since repeated and particularly inopportune interference from other threads may prevent forward progress, e.g., by repeatedly stealing a cache line for unrelated purposes between load-locked and store-conditional instructions. Implementations should ensure that such effects cannot indefinitely delay progress under expected operating conditions, and that such anomalies can therefore safely be ignored by programmers. Outside this International Standard, this property is sometimes termed lock-free. —end note ] |
|  | conflict | Two expression evaluations conflict if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location. |
|  | synchronization operation | The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. |
|  | consume operation  acquire operation  release operation |  |
|  | acquire fence  release fence |  |
|  | relaxed atomic operation  atomic read-modify-write operation |  |
|  | modification order |  |
|  | release sequence |  |
|  | synchronize with |  |
|  | carries a dependency |  |
|  | dependency-ordered before |  |
|  | inter-thread happens before |  |
|  | happens before |  |
|  | visible side effect |  |
|  | ... |  |
| 2.1 Separate translation | source file | The text of the program is kept in units called source files in this International Standard. |
| 2.1 Separate translation | translation unit | A source file  together with all the headers (17.6.1.2) and source files included (16.2) via the preprocessing directive  #include, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is  called a translation unit. [Note: A C ++ program need not all be translated at the same time. —end note ] |
| 2.1 Separate translation | translated translation unit  instantiation unit |  |
| 2.2 Phases of translation | 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. |
| 2.2 Phases of translation | instantiation unit | All the required  instantiations are performed to produce instantiation units. [Note: These are similar to translated  translation units, but contain no references to uninstantiated templates and no template definitions.  —end note ] |
| 2.3 Character sets | basic source character set  space character  control character  graphical character | The basic source character set consists of 96 characters: the space character, the control characters repre-  senting horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters: 14  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  0 1 2 3 4 5 6 7 8 9  \_ { } [ ] # ( ) < > % : ; . ? \* + - / ^ & | ∼ ! = , \ " '  14) The glyphs for the members of the basic source character set are intended to identify characters from the subset of  ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the  source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to  document how the basic source characters are represented in source files. |
| 2.3 Character sets | universal-character-name  hex-quad | The universal-character-name construct provides a way to name other characters.  hex-quad:  　　hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit  universal-character-name:  　　\u hex-quad  　　\U hex-quad hex-quad |
| 2.3 Character sets | basic execution character set  basic execution wide-character set  control character  null character  null wide character | The basic execution character set and the basic execution wide-character set shall each contain all the  members of the basic source character set, plus control characters representing alert, backspace, and carriage  return, plus a null character (respectively, null wide character), whose representation has all zero bits. |
| 2.3 Character sets | execution character set  execution wide-character set | The execution character set  and the execution wide-character set are implementation-defined supersets of the basic execution character  set and the basic execution wide-character set, respectively. |
| 2.4 Trigraph sequences | trigraph sequence | Before any other processing takes place, each occurrence of one of the following sequences of three characters  (“trigraph sequences”) is replaced by the single character indicated in Table 1.  Table 1 — Trigraph sequences   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Trigraph | Replacement | Trigraph | Replacement | Trigraph | Replacement | | ??= | # | ??( | [ | ??< | { | | ??/ | \ | ??) | ] | ??> | } | | ??' | ˆ | ??! | | | ??- | ∼ | |
| 2.5 Preprocessing tokens | preprocessing-token | preprocessing-token:  　　header-name  　　identifier  　　pp-number  　　character-literal  　　user-defined-character-literal  　　string-literal  　　user-defined-string-literal  　　preprocessing-op-or-punc  　　each non-white-space character that cannot be one of the above |
| 2.5 Preprocessing tokens | preprocessing token | A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. |
| 2.5 Preprocessing tokens | white-space character | Preprocessing tokens can be separated by white space; this consists of comments (2.8), or white-space  characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. |
| 2.5 Preprocessing tokens | raw string literal | The raw string literal is defined as  the shortest sequence of characters that matches the raw-string pattern  encoding-prefixopt R raw-string |
| 2.6 Alternative tokens | alternative token  primary token | In all respects of the language, each alternative token behaves the same, respectively, as its primary token,  except for its spelling. 17 The set of alternative tokens is defined in Table 2.  Table 2 — Alternative tokens   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Alternative | Primary | Alternative | Primary | Alternative | Primary | | <% | { | and | && | and\_eq | &= | | %> | } | bitor | | | or\_eq | |= | | <: | [ | or | || | xor\_eq | ˆ= | | :> | ] | xor | ˆ | not | ! | | %: | # | compl | ∼ | not\_eq | != | | %:%: | ## | bitand | & |  |  |   17) Thus the “stringized” values (16.3.2) of [ and <: will be different, maintaining the source spelling, but the tokens can  otherwise be freely interchanged. |
| 2.7 Tokens | token | token:  　　identifier  　　keyword  　　literal  　　operator  　　punctuator |
| 2.7 Tokens | literal | There are five kinds of tokens: identifiers, keywords, literals, 18 operators, and other separators.  18) Literals include strings and character and numeric literals. |
| 2.7 Tokens | blank  white space | Blanks,  horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, “white space”), as described  below, are ignored except as they serve to separate tokens. [Note: Some white space is required to sepa-  rate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic  characters. —end note ] |
| 2.8 Comments |  | The characters /\* start a comment, which terminates with the characters \*/. |
| 2.8 Comments |  | The characters // start a comment, which terminates with the next new-line character. |
| 2.9 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 new-line and >  q-char-sequence:  　　q-char  　　q-char-sequence q-char  q-char:  　　any member of the source character set except new-line and " |
| 2.10 Preprocessing numbers | pp-number | pp-number:  　　digit  　　. digit  　　pp-number digit  　　pp-number ' digit  　　pp-number ' nondigit  　　pp-number identifier-nondigit  　　pp-number e sign  　　pp-number E sign  　　pp-number . |
| 2.11 Identifiers | 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 |
| 2.11 Identifiers | identifier | An identifier is an arbitrarily long sequence of letters and digits. |
| 2.11 Identifiers | identifiers with special meaning | The identifiers in Table 3 have a special meaning when appearing in a certain context.  Table 3 — Identifiers with special meaning   |  |  | | --- | --- | | override | final | |
| 2.11 Identifiers | regular identifier | Unless otherwise specified, any ambiguity as to whether a given identifier has a special meaning is resolved  to interpret the token as a regular identifier. |
| 2.12 Keywords | keyword | The identifiers shown in Table 4 are reserved for use as keywords (that is, they are unconditionally treated  as keywords in phase 7) except in an attribute-token (7.6.1) [Note: The export keyword is unused but is  reserved for future use.—end note ]:  Table 4 — Keywords   |  |  |  |  |  | | --- | --- | --- | --- | --- | | alignas  alignof  asm  auto  bool  break  case  catch  char  char16\_t  char32\_t  class  const  constexpr  const\_cast | continue  decltype  default  delete  do  double  dynamic\_cast  else  enum  explicit  export  extern  false  float  for | friend  goto  if  inline  int  long  mutable  namespace  new  noexcept  nullptr  operator  private  protected  public | register  reinterpret\_cast  return  short  signed  sizeof  static  static\_assert  static\_cast  struct  switch  template  this  thread\_local  throw | true  try  typedef  typeid  typename  union  unsigned  using  virtual  void  volatile  wchar\_t  while | |
| 2.12 Keywords | alternative representation | Furthermore, the alternative representations shown in Table 5 for certain operators and punctuators (2.6)  are reserved and shall not be used otherwise:  Table 5 — Alternative representations   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | and | and\_eq | bitand | bitor | compl | not | | not\_eq | or | or\_eq | xor | xor\_eq |  | |
| 2.13 Operators and punctuators | preprocessing-op-or-punc | preprocessing-op-or-punc: one of  　　{ } [ ] # ## ( )  　　<: :> <% %> %: %:%: ; : ...  　　new delete ? :: . .\*  　　+ - \* / % ˆ & |  　　~  　　! = < > += -= \*= /= %=  　　ˆ= &= |= << >> >>= <<= == !=  　　<= >= && || ++ -- , ->\* ->  　　and and\_eq bitand bitor compl not not\_eq  　　or or\_eq xor xor\_eq |
| 2.14.1 Kinds of literals | literal | There are several kinds of literals. 21  literal:  　　integer-literal  　　character-literal  　　floating-literal  　　string-literal  　　boolean-literal  　　pointer-literal  　　user-defined-literal  21) The term “literal” generally designates, in this International Standard, those tokens that are called “constants” in ISO C. |
| 2.14.2 Integer literals | integer-literal  decimal-literal  octal-literal  hexadecimal-literal  binary-literal  nonzero-digit  octal-digit  hexadecimal-digit  binary-digit  integer-suffix  unsigned-suffix  long-suffix  long-long-suffix | integer-literal:  　　decimal-literal integer-suffixopt  　　octal-literal integer-suffixopt  　　hexadecimal-literal integer-suffixopt  　　binary-literal integer-suffixopt  decimal-literal:  　　nonzero-digit  　　decimal-literal 'opt digit  octal-literal:  　　0  　　octal-literal 'opt octal-digit  hexadecimal-literal:  　　0x hexadecimal-digit  　　0X hexadecimal-digit  　　hexadecimal-literal 'opt hexadecimal-digit  binary-literal:  　　0b binary-digit  　　0B binary-digit  　　binary-literal 'opt binary-digit  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  binary-digit:  　　0  　　1  integer-suffix:  　　unsigned-suffix long-suffixopt  　　unsigned-suffix long-long-suffixopt  　　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 |
| 2.14.2 Integer literals | integer literal | An integer literal is a sequence of digits that has no period or exponent part, with optional separating single  quotes that are ignored when determining its value. |
| 2.14.2 Integer literals | decimal integer literal | A decimal integer literal (base ten) begins with a digit other than 0 and consists of a sequence  of decimal digits. |
| 2.14.2 Integer literals | octal integer literal | An octal integer literal (base eight) begins with the digit 0 and consists of a sequence of  octal digits. 22  22) The digits 8 and 9 are not octal digits. |
| 2.14.2 Integer literals | hexadecimal integer literal | A hexadecimal integer literal (base sixteen) begins with 0x or 0X and consists of a sequence  of hexadecimal digits, which include the decimal digits and the letters a through f and A through F with  decimal values ten through fifteen. |
| 2.14.2 Integer literals | binary integer literal | A binary integer literal (base two) begins with 0b or 0B and consists of  a sequence of binary digits. |
| 2.14.3 Character literals | character-literal  c-char-sequence  c-char  escape-sequence  simple-escape-sequence  octal-escape-sequence  hexadecimal-escape-sequence | character-literal:  　　' c-char-sequence '  　　u' c-char-sequence '  　　U' c-char-sequence '  　　L' 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  　　universal-character-name  escape-sequence:  　　simple-escape-sequence  　　octal-escape-sequence  　　hexadecimal-escape-sequence  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 |
| 2.14.3 Character literals | character literal | A character literal is one or more characters enclosed in single quotes, as in 'x', optionally preceded by  one of the letters u, U, or L, as in u'y', U'z', or L'x', respectively. |
| 2.14.3 Character literals | ordinary character literal  narrow-character literal | A character literal that does not begin  with u, U, or L is an ordinary character literal, also referred to as a narrow-character literal. |
| 2.14.3 Character literals | multicharacter literal | An  ordinary character literal that contains more than one c-char is a multicharacter literal. |
| 2.14.3 Character literals | wide-character literal | A  character literal that begins with the letter L, such as L'x', is a wide-character literal. |
| 2.14.3 Character literals | nongraphic character  escape sequence | Certain nongraphic characters, the single quote ', the double quote ", the question mark ?, 24 and the  backslash \, can be represented according to Table 7.  Table 7 — Escape sequences   |  |  |  | | --- | --- | --- | | new-line  horizontal tab  vertical tab  backspace  carriage return  form feed  alert  backslash  question mark  single quote  double quote  octal number  hex number | NL(LF)  HT  VT  BS  CR  FF  BEL  \  ?  '  "  ooo  hhh | \n  \t  \v  \b  \r  \f  \a  \\  \?  \'  \"  \ooo  \xhhh | |
| 2.14.4 Floating literals | floating-literal  fractional-constant  exponent-part  sign  digit-sequence  floating-suffix | floating-literal:  　　fractional-constant exponent-partopt floating-suffixopt  　　digit-sequence 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 'opt digit  floating-suffix: one of  　　f l F L |
| 2.14.4 Floating literals | significant part | The integer part, the optional decimal  point and the optional fraction part form the significant part of the floating literal. |
| 2.14.5 String literals |  | string-literal:  　　encoding-prefixopt " s-char-sequenceopt "  　　encoding-prefixopt R raw-string  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  　　universal-character-name  raw-string:  　　" d-char-sequenceopt ( r-char-sequenceopt ) d-char-sequenceopt "  r-char-sequence:  　　r-char  　　r-char-sequence r-char  r-char:  　　any member of the source character set, except a right parenthesis ) followed by the initial d-char-sequence (which may be empty) followed by a double quote ".  d-char-sequence:  　　d-char  　　d-char-sequence d-char  d-char:  　　any member of the basic source character set except: space, the left parenthesis (, the right parenthesis ), the backslash \, and the control characters representing horizontal tab, vertical tab, form feed, and newline. |
| 2.14.5 String literals | string literal | A string literal is a sequence of characters (as defined in 2.14.3) surrounded by double quotes, optionally  prefixed by R, u8, u8R, u, uR, U, UR, L, or LR, as in "...", R"(...)", u8"...", u8R"\*\*(...)\*\*", u"...",  uR"\*~(...)\*~", U"...", UR"zzz(...)zzz", L"...", or LR"(...)", respectively. |
| 2.14.5 String literals | raw string literal | A string literal that has an R in the prefix is a raw string literal. |
| 2.14.5 String literals | ordinary string literal | After translation phase 6, a string literal that does not begin with an encoding-prefix is an ordinary string  literal, and is initialized with the given characters. |
| 2.14.5 String literals | UTF-8 string literal | A string literal that begins with u8, such as u8"asdf", is a UTF-8 string literal. |
| 2.14.5 String literals | narrow string literal | Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. |
| 2.14.5 String literals | char16\_t string literal | A string literal that begins with u, such as u"asdf", is a char16\_t string literal. |
| 2.14.5 String literals | char32\_t string literal | A string literal that begins with U, such as U"asdf", is a char32\_t string literal. |
| 2.14.5 String literals | wide string literal | A string literal that begins with L, such as L"asdf", is a wide string literal. |
| 2.14.5 String literals | multibyte encoding | In a narrow string literal, a universal-character-  name may map to more than one char element due to multibyte encoding. |
| 2.14.6 Boolean literals | boolean-literal | boolean-literal:  　　false  　　true |
| 2.14.6 Boolean literals | Boolean literals | The Boolean literals are the keywords false and true. |
| 2.14.7 Pointer literals | pointer-literal | pointer-literal:  　　nullptr |
| 2.14.7 Pointer literals | pointer literal | The pointer literal is the keyword nullptr. |
| 2.14.8 User-defined literals | user-defined-literal  user-defined-integer-literal  user-defined-floating-literal  user-defined-string-literal  user-defined-character-literal  ud-suffix | user-defined-literal:  　　user-defined-integer-literal  　　user-defined-floating-literal  　　user-defined-string-literal  　　user-defined-character-literal  user-defined-integer-literal:  　　decimal-literal ud-suffix  　　octal-literal ud-suffix  　　hexadecimal-literal ud-suffix  　　binary-literal ud-suffix  user-defined-floating-literal:  　　fractional-constant exponent-part opt ud-suffix  　　digit-sequence exponent-part ud-suffix  user-defined-string-literal:  　　string-literal ud-suffix  user-defined-character-literal:  　　character-literal ud-suffix  ud-suffix:  　　identifier |
| 3 Basic concepts | entity | An entity is a value, object, reference, function, enumerator, type, class member, template, template spe-  cialization, namespace, parameter pack, or this. |
| 3 Basic concepts | name | A name is a use of an identifier (2.11), operator-function-id (13.5), literal-operator-id (13.5.8), conversion-  function-id (12.3.2), or template-id (14.2) that denotes an entity or label (6.6.4, 6.1). |
| 3 Basic concepts | declaration | Every name that denotes an entity is introduced by a declaration. |
| 3 Basic concepts | variable | A variable is introduced by the declaration of a reference other than a non-static data member or of an  object. |
| 3 Basic concepts | name lookup | Some names denote types or templates. In general, whenever a name is encountered it is necessary to  determine whether that name denotes one of these entities before continuing to parse the program that  contains it. The process that determines this is called name lookup (3.4). |
| 3 Basic concepts | same | Two names are the same if  — they are identifiers composed of the same character sequence, or  — they are operator-function-ids formed with the same operator, or  — they are conversion-function-ids formed with the same type, or  — they are template-ids that refer to the same class or function (14.4), or  — they are the names of literal operators (13.5.8) formed with the same literal suffix identifier. |
| 3.1 Declarations and definitions | definition | A declaration is a definition unless it declares a function without specifying the function’s body (8.4), it  contains the extern specifier (7.1.1) or a linkage-specification 25 (7.5) and neither an initializer nor a function-  body, it declares a static data member in a class definition (9.2, 9.4), it is a class name declaration (9.1), it is  an opaque-enum-declaration (7.2), it is a template-parameter (14.1), it is a parameter-declaration (8.3.5) in a  function declarator that is not the declarator of a function-definition, or it is a typedef declaration (7.1.3),  an alias-declaration (7.1.3), a using-declaration (7.3.3), a static\_assert-declaration (Clause 7), an attribute-  declaration (Clause 7), an empty-declaration (Clause 7), or a using-directive (7.3.4).  25) Appearing inside the braced-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a  definition. |
| 3.2 One definition rule |  | No translation unit shall contain more than one definition of any variable, function, class type, enumeration  type, or template. |
| 3.2 One definition rule | potentially evaluated | An expression is potentially evaluated unless it is an unevaluated operand (Clause 5) or a subexpression  thereof. |
| 3.2 One definition rule | potential results | The set of potential results of an expression e is defined as follows:  — If e is an id-expression (5.1.1), the set contains only e.  — If e is a class member access expression (5.2.5), the set contains the potential results of the object  expression.  — If e is a pointer-to-member expression (5.5) whose second operand is a constant expression, the set  contains the potential results of the object expression.  — If e has the form (e1), the set contains the potential results of e1.  — If e is a glvalue conditional expression (5.16), the set is the union of the sets of potential results of the  second and third operands.  — If e is a comma expression (5.18), the set contains the potential results of the right operand.  — Otherwise, the set is empty. |
| 3.2 One definition rule | odr-used | A variable x whose name appears as a potentially-evaluated expression ex is odr-used unless applying the  lvalue-to-rvalue conversion (4.1) to x yields a constant expression (5.19) that does not invoke any non-trivial  functions and, if x is an object, ex is an element of the set of potential results of an expression e, where  either the lvalue-to-rvalue conversion (4.1) is applied to e, or e is a discarded-value expression (Clause 5).  this is odr-used if it appears as a potentially-evaluated expression (including as the result of the implicit  transformation in the body of a non-static member function (9.3.1)). A virtual member function is odr-used  if it is not pure. A function whose name appears as a potentially-evaluated expression is odr-used if it is  the unique lookup result or the selected member of a set of overloaded functions (3.4, 13.3, 13.4), unless  it is a pure virtual function and its name is not explicitly qualified. [Note: This covers calls to named  functions (5.2.2), operator overloading (Clause 13), user-defined conversions (12.3.2), allocation function for  placement new (5.3.4), as well as non-default initialization (8.5). A constructor selected to copy or move an  object of class type is odr-used even if the call is actually elided by the implementation (12.8). —end note ]  An allocation or deallocation function for a class is odr-used by a new expression appearing in a potentially-  evaluated expression as specified in 5.3.4 and 12.5. A deallocation function for a class is odr-used by a delete  expression appearing in a potentially-evaluated expression as specified in 5.3.5 and 12.5. A non-placement  allocation or deallocation function for a class is odr-used by the definition of a constructor of that class. A  non-placement deallocation function for a class is odr-used by the definition of the destructor of that class,  or by being selected by the lookup at the point of definition of a virtual destructor (12.4). 26 An assignment  operator function in a class is odr-used by an implicitly-defined copy-assignment or move-assignment function  for another class as specified in 12.8. A default constructor for a class is odr-used by default initialization or  value initialization as specified in 8.5. A constructor for a class is odr-used as specified in 8.5. A destructor  for a class is odr-used if it is potentially invoked (12.4).  26) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however,  this is a permissible implementation technique. |
| 3.3.1 Declarative regions and scopes | declarative region  valid | Every name is introduced in some portion of program text called a declarative region, which is the largest part  of the program in which that name is valid, that is, in which that name may be used as an unqualified name  to refer to the same entity. |
|  | scope  potential scope | In general, each particular name is valid only within some possibly discontiguous  portion of program text called its scope. To determine the scope of a declaration, it is sometimes convenient  to refer to the potential scope of a declaration. The scope of a declaration is the same as its potential scope  unless the potential scope contains another declaration of the same name. In that case, the potential scope  of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration  in the outer (containing) declarative region. |
| 3.3.2 Point of declaration | point of declaration | The point of declaration for a name is immediately after its complete declarator (Clause 8) and before its  initializer (if any), except as noted below. |
| 3.3.3 Block scope | block scope | A name declared in a block (6.3) is local to that block; it has block scope. |
| 3.3.3 Block scope | local variable | A variable declared at block scope is a local  variable. |
| 3.3.4 Function prototype scope | function prototype scope | In a function declaration, or in any function declarator except the declarator of a function definition (8.4),  names of parameters (if supplied) have function prototype scope, which terminates at the end of the nearest  enclosing function declarator. |
| 3.3.5 Function scope | function scope | Labels (6.1) have function scope and may be used anywhere in the function in which they are declared. |
| 3.3.6 Namespace scope | namespace-body | The declarative region of a namespace-definition is its namespace-body. |
| 3.3.6 Namespace scope | member  member name | Entities declared  in a namespace-body are said to be members of the namespace, and names introduced by these declarations  into the declarative region of the namespace are said to be member names of the namespace. |
| 3.3.6 Namespace scope | namespace scope | A namespace  member name has namespace scope. |
| 3.3.6 Namespace scope | global namespace | The outermost declarative region of a translation unit is also a namespace, called the global namespace. |
| 3.3.6 Namespace scope | global namespace scope  global scope | A  name declared in the global namespace has global namespace scope (also called global scope). |
| 3.3.6 Namespace scope | global name | A name with global namespace scope is said to be a global name. |
| 3.3.7 Class scope |  | The following rules describe the scope of names declared in classes.  ... |
| 3.3.8 Enumeration scope | enumeration scope | The name of a scoped enumerator (7.2) has enumeration scope. |
| 3.3.9 Template parameter scope |  |  |
| 3.3.10 Name hiding |  | A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived  class (10.2). |
| 3.3.10 Name hiding | visible | If a name is in scope and is not hidden it is said to be visible. |
| 3.4 Name lookup | looked up in the context of an expression | A name “looked up in the context of an expression” is looked up as an unqualified name in the scope where  the expression is found. |
|  | ... |  |
| 3.5 Program and linkage | program | A program consists of one or more translation units (Clause 2) linked together. |
| 3.5 Program and linkage | translation unit | A translation unit consists  of a sequence of declarations. |
| 3.5 Program and linkage | translation-unit | translation-unit:  　　declaration-seqopt |
| 3.5 Program and linkage | linkage  external linkage  internal linkage  no linkage | A name is said to have linkage when it might denote the same object, reference, function, type, template,  namespace or value as a name introduced by a declaration in another scope:  — When a name has external linkage , the entity it denotes can be referred to by names from scopes of  other translation units or from other scopes of the same translation unit.  — When a name has internal linkage , the entity it denotes can be referred to by names from other scopes  in the same translation unit.  — When a name has no linkage , the entity it denotes cannot be referred to by names from other scopes. |
| 3.6.1 Main function | main | A program shall contain a global function called main, which is the designated start of the program. |
| 3.6.1 Main function | start-up  termination | In a freestanding environment, start-up and termination is implementation-defined; start-  up contains the execution of constructors for objects of namespace scope with static storage duration;  termination contains the execution of destructors for objects with static storage duration. |
| 3.6.1 Main function | ntmbs | If argc is nonzero these arguments  shall be supplied in argv[0] through argv[argc-1] as pointers to the initial characters of null-terminated  multibyte strings (ntmbs s) (17.5.2.1.4.2) and argv[0] shall be the pointer to the initial character of a  ntmbs that represents the name used to invoke the program or "". |
| 3.6.2 Initialization of non-local variables | named non-local variables | There are two broad classes of named non-local variables: those with static storage duration (3.7.1) and  those with thread storage duration (3.7.2). |
| 3.6.2 Initialization of non-local variables | constant initializer | A constant initializer for an object o is an expression that is a  constant expression, except that it may also invoke constexpr constructors for o and its subobjects even  if those objects are of non-literal class types [Note: such a class may have a non-trivial destructor —end  note ]. |
| 3.6.2 Initialization of non-local variables | constant initialization | Constant initialization is performed:  — if each full-expression (including implicit conversions) that appears in the initializer of a reference with  static or thread storage duration is a constant expression (5.19) and the reference is bound to an lvalue  designating an object with static storage duration, to a temporary (see 12.2), or to a function;  — if an object with static or thread storage duration is initialized by a constructor call, and if the  initialization full-expression is a constant initializer for the object;  — if an object with static or thread storage duration is not initialized by a constructor call and if either the  object is value-initialized or every full-expression that appears in its initializer is a constant expression. |
| 3.6.2 Initialization of non-local variables | static initialization  dynamic initialization | Together, zero-initialization and constant initialization are called static initialization; all other initial-  ization is dynamic initialization. |
| 3.6.2 Initialization of non-local variables | ordered  unordered | Dynamic initialization of a non-local variable with static storage duration is either ordered or  unordered. |
| 3.6.2 Initialization of non-local variables | fully initialized  zero-initialized | As a consequence, if the initialization of an object obj1 refers to an object obj2 of namespace scope  potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified  whether the value of obj2 used will be the value of the fully initialized obj2 (because obj2 was statically  initialized) or will be the value of obj2 merely zero-initialized. |
| 3.7 Storage duration | storage duration | Storage duration is the property of an object that defines the minimum potential lifetime of the storage  containing the object. |
| 3.7 Storage duration | lifetime | The lifetime of a reference is its storage duration. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

errors

|  |
| --- |
| 2.4 Trigraph sequences |
| No other trigraph sequence exists. Each ? that does not begin one of the trigraphs listed above is not changed. |

|  |
| --- |
| 2.14.5 String literals |
| A single c-char may produce more than one char16\_t  character in the form of surrogate pairs. |