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 representing 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 separate 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  constants | 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-prefix  s-char-sequence  s-char  raw-string  r-char-sequence  r-char  d-char-sequence  d-char | 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 specialization, 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. |
| 3.3.1 Declarative regions and scopes | 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 initialization 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. |
| 3.7.1 Static storage duration | static storage duration | All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have static storage duration. |
| 3.7.2 Thread storage duration | thread storage duration | All variables declared with the thread\_local keyword have thread storage duration. |
| 3.7.3 Automatic storage duration | automatic storage duration | Block-scope variables explicitly declared register or not explicitly declared static or extern have automatic storage duration. |
| 3.7.4 Dynamic storage duration |  | Objects can be created dynamically during program execution (1.9), using new-expressions (5.3.4), and destroyed using delete-expressions (5.3.5). |
| 3.7.4 Dynamic storage duration | allocation function  deallocation function | A C++ implementation provides access to, and management of, dynamic storage via the global allocation functions operator new and operator new[] and the global deallocation functions operator delete and operator delete[]. |
| 3.7.4 Dynamic storage duration | usual (non-placement) deallocation function. | The global operator delete with exactly one parameter is a usual (non-placement) deallocation function. |
| 3.7.4 Dynamic storage duration | usual deallocation function | The global operator delete with exactly two parameters, the second of which has type std::size\_t, is a usual deallocation function. |
| 3.7.4 Dynamic storage duration | usual deallocation function | Similarly, the global operator delete[] with exactly one parameter is a usual deallocation function. |
| 3.7.4 Dynamic storage duration | usual deallocation function  placement allocation function | The global operator delete[] with exactly two parameters, the second of which has type std::size\_t, is a usual deallocation function. 37  37) This deallocation function precludes use of an allocation function void operator new(std::size\_t, std::size\_t) as a placement allocation function (C.3.2). |
| 3.7.4 Dynamic storage duration | usual deallocation function | If a class T has a member deallocation function named operator delete with exactly one parameter, then that function is a usual deallocation function. |
| 3.7.4 Dynamic storage duration | usual deallocation function | If class T does not declare such an operator delete but does declare a member deallocation function named operator delete with exactly two parameters, the second of which has type std::size\_t, then this function is a usual deallocation function. |
| 3.7.4 Dynamic storage duration | usual (non-placement) deallocation function | Similarly, if a class T has a member deallocation function named operator delete[] with exactly one parameter, then that function is a usual (non-placement) deallocation function. |
| 3.7.4 Dynamic storage duration | usual deallocation function | If class T does not declare such an operator delete[] but does declare a member deallocation function named operator delete[] with exactly two parameters, the second of which has type std::size\_t, then this function is a usual deallocation function. |
| 3.7.4.3 Safely-derived pointers | traceable pointer object | A traceable pointer object is  — an object of an object pointer type (3.9.2), or  — an object of an integral type that is at least as large as std::intptr\_t, or  — a sequence of elements in an array of narrow character type (3.9.1), where the size and alignment of the sequence match those of some object pointer type. |
| 3.7.4.3 Safely-derived pointers | safely-derived pointer | A pointer value is a safely-derived pointer to a dynamic object only if it has an object pointer type and it is one of the following:  — the value returned by a call to the C++ standard library implementation of ::operator new(std::size\_t); 39  — the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from indirection through a safely-derived pointer value;  — the result of well-defined pointer arithmetic (5.7) using a safely-derived pointer value;  — the result of a well-defined pointer conversion (4.10, 5.4) of a safely-derived pointer value;  — the result of a reinterpret\_cast of a safely-derived pointer value;  — the result of a reinterpret\_cast of an integer representation of a safely-derived pointer value;  — the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value. |
| 3.7.4.3 Safely-derived pointers | integer representation of a safely-derived pointer | An integer value is an integer representation of a safely-derived pointer only if its type is at least as large as std::intptr\_t and it is one of the following:  — the result of a reinterpret\_cast of a safely-derived pointer value;  — the result of a valid conversion of an integer representation of a safely-derived pointer value;  — the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;  — the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value P, if that result converted by reinterpret\_cast<void\*> would compare equal to a safely-derived pointer computable from reinterpret\_cast<void\*>(P). |
| 3.7.4.3 Safely-derived pointers | relaxed pointer safety | An implementation may have relaxed pointer safety, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. |
| 3.7.4.3 Safely-derived pointers | strict pointer safety | Alternatively, an implementation may have strict pointer safety, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (20.7.4). [Note: the effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 3.7.4.2. This is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. —end note ] |
| 3.8 Object lifetime | lifetime | The lifetime of an object is a runtime property of the object. |
| 3.8 Object lifetime | non-trivial initialization | An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [Note: initialization by a trivial copy/move constructor is non-trivial initialization. —end note ] |
| 3.8 Object lifetime | ends the lifetime of an object of type T with static (3.7.1), thread (3.7.2), or automatic (3.7.3) storage duration and if T has a non-trivial destructor | If a program ends the lifetime of an object of type T with static (3.7.1), thread (3.7.2), or automatic (3.7.3) storage duration and if T has a non-trivial destructor, 41 the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception.  41) That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration. |
| 3.8 Object lifetime | before  after | In this section, "before" and "after" refer to the "happens before" relation (1.10). [Note: Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from another thread without adequate synchronization. —end note ] |
| 3.9 Types | object representation | The object representation of an object of type T is the sequence of N unsigned char objects taken up by the object of type T, where N equals sizeof(T). |
| 3.9 Types | value representation | The value representation of an object is the set of bits that hold the value of type T. |
| 3.9 Types | value | For trivially copyable types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values. 44  44) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C. |
| 3.9 Types | incompletely-defined object type | A class that has been declared but not defined, an enumeration type in certain contexts (7.2), or an array of unknown size or of incomplete element type, is an incompletely-defined object type. 45  45) The size and layout of an instance of an incompletely-defined object type is unknown. |
| 3.9 Types | incomplete type | Incompletely-defined object types and the void types are incomplete types (3.9.1). |
| 3.9 Types | object type | An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a void type. |
| 3.9 Types | scalar types | Arithmetic types (3.9.1), enumeration types, pointer types, pointer to member types (3.9.2), std::nullptr\_t, and cv-qualified versions of these types (3.9.3) are collectively called scalar types. |
| 3.9 Types | POD types | Scalar types, POD classes (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called POD types. |
| 3.9 Types | trivially copyable types | Cv-unqualified scalar types, trivially copyable class types (Clause 9), arrays of such types, and non-volatile const-qualified versions of these types (3.9.3) are collectively called trivially copyable types. |
| 3.9 Types | trivial types | Scalar types, trivial class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called trivial types. |
| 3.9 Types | standard-layout types | Scalar types, standard-layout class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called standard-layout types. |
| 3.9 Types | literal type | A type is a literal type if it is:  — void; or  — a scalar type; or  — a reference type; or  — an array of literal type; or  — a class type (Clause 9) that has all of the following properties:  — it has a trivial destructor,  — it is an aggregate type (8.5.1) or has at least one constexpr constructor or constructor template that is not a copy or move constructor, and  — all of its non-static data members and base classes are of non-volatile literal types. |
| 3.9 Types | layout-compatible | If two types T1 and T2 are the same type, then T1 and T2 are layout-compatible types. |
| 3.9.1 Fundamental types | characters  char | Objects declared as characters (char) shall be large enough to store any member of the implementation's basic character set. |
| 3.9.1 Fundamental types | narrow character types | Plain char, signed char, and unsigned char are three distinct types, collectively called narrow character types. |
| 3.9.1 Fundamental types | standard signed integer type | There are five standard signed integer types : "signed char", "short int", "int", "long int", and "long long int". |
| 3.9.1 Fundamental types | extended signed integer type | There may also be implementation-defined extended signed integer types. |
| 3.9.1 Fundamental types | signed integer types | The standard and extended signed integer types are collectively called signed integer types. |
| 3.9.1 Fundamental types | the natural size suggested by the architecture of the execution environment | Plain ints have the natural size suggested by the architecture of the execution environment 46 ; the other signed integer types are provided to meet special needs.  46) that is, large enough to contain any value in the range of INT\_MIN and INT\_MAX, as defined in the header <climits>. |
| 3.9.1 Fundamental types | standard unsigned integer type | For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type: "unsigned char", "unsigned short int", "unsigned int", "unsigned long int", and "unsigned long long int", each of which occupies the same amount of storage and has the same alignment requirements (3.11) as the corresponding signed integer type 47 ; that is, each signed integer type has the same object representation as its corresponding unsigned integer type.  47) See 7.1.6.2 regarding the correspondence between types and the sequences of type-specifiers that designate them. |
| 3.9.1 Fundamental types | extended unsigned integer type | Likewise, for each of the extended signed integer types there exists a corresponding extended unsigned integer type with the same amount of storage and alignment requirements. |
| 3.9.1 Fundamental types | unsigned integer types | The standard and extended unsigned integer types are collectively called unsigned integer types. |
| 3.9.1 Fundamental types | standard integer types  extended integer types | The standard signed integer types and standard unsigned integer types are collectively called the standard integer types, and the extended signed integer types and extended unsigned integer types are collectively called the extended integer types. |
| 3.9.1 Fundamental types | wchar\_t | Type wchar\_t is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.3.1). |
| 3.9.1 Fundamental types | underlying type | Type wchar\_t shall have the same size, signedness, and alignment requirements (3.11) as one of the other integral types, called its underlying type. |
| 3.9.1 Fundamental types | char16\_t  char32\_t  underlying type | Types char16\_t and char32\_t denote distinct types with the same size, signedness, and alignment as uint\_least16\_t and uint\_least32\_t, respectively, in <cstdint>, called the underlying types. |
| 3.9.1 Fundamental types | bool | Values of type bool are either true or false. 49 [Note: There are no signed, unsigned, short, or long bool types or values. —end note ]  49) Using a bool value in ways described by this International Standard as "undefined," such as by examining the value of an uninitialized automatic object, might cause it to behave as if it is neither true nor false. |
| 3.9.1 Fundamental types | integral types | Types bool, char, char16\_t, char32\_t, wchar\_t, and the signed and unsigned integer types are collectively called integral types. 50  50) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to integral types as specified in 4.5. |
| 3.9.1 Fundamental types | integer type | A synonym for integral type is integer type. |
| 3.9.1 Fundamental types | pure binary numeration system  2's complement  1's complement  signed magnitude | The representations of integral types shall define values by use of a pure binary numeration system. 51 [Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types. —end example ]  51) 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 power of 2, except perhaps for the bit with the highest position. (Adapted from the American National Dictionary for Information Processing Systems.) |
| 3.9.1 Fundamental types | floating point types | There are three floating point types: float, double, and long double. |
| 3.9.1 Fundamental types | arithmetic types | Integral and floating types are collectively called arithmetic types. |
| 3.9.1 Fundamental types | void | The void type has an empty set of values. |
| 3.9.1 Fundamental types | std::nullptr\_t | A value of type std::nullptr\_t is a null pointer constant (4.10). |
| 3.9.2 Compound types | compound type  array  function  pointer  reference  classe  union  enumeration  pointer to non-static class member | Compound types can be constructed in the following ways:  — arrays of objects of a given type, 8.3.4;  — functions, which have parameters of given types and return void or references or objects of a given type, 8.3.5;  — pointers to void or objects or functions (including static members of classes) of a given type, 8.3.1;  — references to objects or functions of a given type, 8.3.2. There are two types of references:  — lvalue reference  — rvalue reference  — classes containing a sequence of objects of various types (Clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (Clause 11);  — unions, which are classes capable of containing objects of different types at different times, 9.5;  — enumerations, which comprise a set of named constant values. Each distinct enumeration constitutes a different enumerated type, 7.2;  — pointers to non-static 52 class members, which identify members of a given type within objects of a given class, 8.3.3.  52) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions. |
| 3.9.2 Compound types | object pointer type | The type of a pointer to void or a pointer to an object type is called an object pointer type. [Note: A pointer to void does not have a pointer-to-object type, however, because void is not an object type. —end note ] |
| 3.9.2 Compound types | function pointer type | The type of a pointer that can designate a function is called a function pointer type. |
| 3.9.2 Compound types | point to | If an object of type T is located at an address A, a pointer of type cv T\* whose value is the address A is said to point to that object, regardless of how the value was obtained. [Note: For instance, the address one past the end of an array (5.7) would be considered to point to an unrelated object of the array's element type that might be located at that address. There are further restrictions on pointers to objects with dynamic storage duration; see 3.7.4.3. —end note ] |
| 3.9.3 CV-qualifiers | cv-unqualified type | A type mentioned in 3.9.1 and 3.9.2 is a cv-unqualified type. |
| 3.9.3 CV-qualifiers | const-qualified  volatile-qualified  const-volatile-qualified | Each type which is a cv-unqualified complete or incomplete object type or is void (3.9) has three corresponding cv-qualified versions of its type: a const-qualified version, a volatile-qualified version, and a const-volatile-qualified version. |
| 3.9.3 CV-qualifiers | const object | A const object is an object of type const T or a non-mutable subobject of such an object. |
| 3.9.3 CV-qualifiers | volatile object | A volatile object is an object of type volatile T, a subobject of such an object, or a mutable subobject of a const volatile object. |
| 3.9.3 CV-qualifiers | const volatile object | A const volatile object is an object of type const volatile T, a non-mutable subobject of such an object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object. |
| 3.9.3 CV-qualifiers | more cv-qualified | There is a partial ordering on cv-qualifiers, so that a type can be said to be more cv-qualified than another. Table 9 shows the relations that constitute this ordering.  Table 9 — Relations on const and volatile   |  | | --- | | no cv-qualifier < const  no cv-qualifier < volatile  no cv-qualifier < const volatile  const < const volatile  volatile < const volatile | |
| 3.9.3 CV-qualifiers | cv (or cv1, cv2, etc.) | In this International Standard, the notation cv (or cv1, cv2, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of {const}, {volatile}, {const, volatile}, or the empty set. |
| 3.10 Lvalues and rvalues | lvalue | An lvalue (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. |
| 3.10 Lvalues and rvalues | xvalue | An xvalue (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). |
| 3.10 Lvalues and rvalues | glvalue | A glvalue ("generalized" lvalue) is an lvalue or an xvalue. |
| 3.10 Lvalues and rvalues | rvalue | An rvalue (so called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object. |
| 3.10 Lvalues and rvalues | prvalue | A prvalue ("pure" rvalue) is an rvalue that is not an xvalue. |
|  | value category | Every expression belongs to exactly one of the fundamental classifications in this taxonomy: lvalue, xvalue, or prvalue. This property of an expression is called its value category. [Note: The discussion of each built-in operator in Clause 5 indicates the category of the value it yields and the value categories of the operands it expects. For example, the built-in assignment operators expect that the left operand is an lvalue and that the right operand is a prvalue and yield an lvalue as the result. User-defined operators are functions, and the categories of values they expect and yield are determined by their parameter and return types. —end note ] |
| 3.10 Lvalues and rvalues | modifiable | If an expression can be used to modify the object to which it refers, the expression is called modifiable. |
| 3.11 Alignment | alignment requirement | Object types have alignment requirements (3.9.1, 3.9.2) which place restrictions on the addresses at which an object of that type may be allocated. |
| 3.11 Alignment | 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. |
| 3.11 Alignment | 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(std::max\_align\_t) (18.2). |
| 3.11 Alignment | extended alignment | An extended alignment is represented by an alignment greater than alignof(std::max\_align\_t). |
| 3.11 Alignment | over-aligned type | A type having an extended alignment requirement is an over-aligned type. [Note: every over-aligned type is or contains a class type to which extended alignment applies (possibly through a non-static data member). —end note ] |
| 3.11 Alignment | weaker  stronger  stricter | Alignments have an order from weaker to stronger or stricter alignments. |
| 4 Standard conversions | standard conversion | Standard conversions are implicit conversions with built-in meaning. |
| 4 Standard conversions | standard conversion sequence | A standard conversion sequence is a sequence of standard conversions in the following order:  — Zero or one conversion from the following set: lvalue-to-rvalue conversion, array-to-pointer conversion, and function-to-pointer conversion.  — Zero or one conversion from the following set: integral promotions, floating point promotion, integral conversions, floating point conversions, floating-integral conversions, pointer conversions, pointer to member conversions, and boolean conversions.  — Zero or one qualification conversion.  [Note: A standard conversion sequence can be empty, i.e., it can consist of no conversions. —end note ] |
| 4 Standard conversions | implicitly converted | An expression e can be implicitly converted to a type T if and only if the declaration T t=e; is well-formed, for some invented temporary variable t (8.5). |
| 4 Standard conversions | contextually converted to bool | Certain language constructs require that an expression be converted to a Boolean value. An expression e appearing in such a context is said to be contextually converted to bool and is well-formed if and only if the declaration bool t(e); is well-formed, for some invented temporary variable t (8.5). |
| 4 Standard conversions | ontextually implicitly converted to | Certain language constructs require conversion to a value having one of a specified set of types appropriate to the construct. An expression e of class type E appearing in such a context is said to be contextually implicitly converted to a specified type T and is well-formed if and only if e can be implicitly converted to a type T that is determined as follows: E is searched for conversion functions whose return type is cv T or reference to cv T such that T is allowed by the context. There shall be exactly one such T. |
| 4.1 Lvalue-to-rvalue conversion |  | A glvalue (3.10) of a non-function, non-array type T can be converted to a prvalue. 55 If T is an incomplete type, a program that necessitates this conversion is ill-formed. If T is a non-class type, the type of the prvalue is the cv-unqualified version of T. Otherwise, the type of the prvalue is T. 56  55) For historical reasons, this conversion is called the "lvalue-to-rvalue" conversion, even though that name does not accurately reflect the taxonomy of expressions described in 3.10.  56) In C++ class prvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types. |
| 4.2 Array-to-pointer conversion |  | An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be converted to a prvalue of type "pointer to T". |
| 4.3 Function-to-pointer conversion |  | An lvalue of function type T can be converted to a prvalue of type "pointer to T." |
| 4.4 Qualification conversions |  | 1 A prvalue of type "pointer to cv1 T" can be converted to a prvalue of type "pointer to cv2 T" if "cv2 T" is more cv-qualified than "cv1 T".  2 A prvalue of type "pointer to member of X of type cv1 T" can be converted to a prvalue of type "pointer to member of X of type cv2 T" if "cv2 T" is more cv-qualified than "cv1 T". |
| 4.4 Qualification conversions | similar | Two pointer types T1 and T2 are similar if there exists a type T and integer n > 0 such that:  T1 is cv1,0 pointer to cv1,1 pointer to ... cv1,n-1 pointer to cv1,n T  and  T2 is cv2,0 pointer to cv2,1 pointer to ... cv2,n-1 pointer to cv2,n T  where each cvi,j is const, volatile, const volatile, or nothing. |
| 4.4 Qualification conversions | cv-qualification signature | The n-tuple of cv-qualifiers after the first in a pointer type, e.g., cv1,1 , cv1,2 , ···, cv1,n in the pointer type T1, is called the cv-qualification signature of the pointer type. |
| 4.4 Qualification conversions | multi-level pointer to member type  multi-level mixed pointer and pointer to member type | A multi-level pointer to member type, or a multi-level mixed pointer and pointer to member type has the form:  cv0 P0 to cv1 P1 to ... cvn-1 Pn-1 to cvn T  where Pi is either a pointer or pointer to member and where T is not a pointer type or pointer to member type. |
| 4.4 Qualification conversions | similar | Two multi-level pointer to member types or two multi-level mixed pointer and pointer to member types T1 and T2 are similar if there exists a type T and integer n > 0 such that:  T1 is cv1,0 P0 to cv1,1 P1 to ... cv1,n-1 Pn-1 to cv1,n T  and  T2 is cv2,0 P0 to cv2,1 P1 to ... cv2,n-1 Pn-1 to cv2,n T |
| 4.5 Integral promotions | integral promotion | ...  These conversions are called integral promotions. |
| 4.6 Floating point promotion | floating point promotion | 1 A prvalue of type float can be converted to a prvalue of type double. The value is unchanged.  2 This conversion is called floating point promotion. |
| 4.7 Integral conversions |  | ... |
| 4.8 Floating point conversions |  | ... |
| 4.9 Floating-integral conversions |  | ... |
| 4.10 Pointer conversions | null pointer constant | A null pointer constant is an integer literal (2.14.2) with value zero or a prvalue of type std::nullptr\_t. |
| 4.10 Pointer conversions | null pointer value  null pointer conversion | A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of object pointer or function pointer type. Such a conversion is called a null pointer conversion. |
| 4.11 Pointer to member conversions | null member pointer value  null member pointer conversion | A null pointer constant (4.10) can be converted to a pointer to member type; the result is the null member pointer value of that type and is distinguishable from any pointer to member not created from a null pointer constant. Such a conversion is called a null member pointer conversion. |
| 4.12 Boolean conversions |  | ... |
| 4.13 Integer conversion rank | integer conversion rank | very integer type has an integer conversion rank defined as follows:  — No two signed integer types other than char and signed char (if char is signed) shall have the same rank, even if they have the same representation.  — The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.  — The rank of long long int shall be greater than the rank of long int, which shall be greater than the rank of int, which shall be greater than the rank of short int, which shall be greater than the rank of signed char.  — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.  — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.  — The rank of char shall equal the rank of signed char and unsigned char.  — The rank of bool shall be less than the rank of all other standard integer types.  — The ranks of char16\_t, char32\_t, and wchar\_t shall equal the ranks of their underlying types (3.9.1).  — The rank of any extended signed integer type relative to another extended signed integer type with the same size is implementation-defined, but still subject to the other rules for determining the integer conversion rank.  — For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 shall have greater rank than T3.  [Note: The integer conversion rank is used in the definition of the integral promotions (4.5) and the usual arithmetic conversions (Clause 5). —end note ] |
| 5 Expressions | expression | An expression is a sequence of operators and operands that specifies a computation. |
| 5 Expressions | overload | Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 9) or enumeration type (7.2). |
| 5 Expressions | built-in operator | Operator overloading shall not modify the rules for the built-in operators, that is, for operators applied to types for which they are defined by this Standard. |
| 5 Expressions | unevaluated operand | In some contexts, unevaluated operands appear (5.2.8, 5.3.3, 5.3.7, 7.1.6.2). |
| 5 Expressions | usual arithmetic conversion | Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:  ... |
| 5 Expressions | discarded-value expression | In some contexts, an expression only appears for its side effects. Such an expression is called a discarded-value expression. |
| 5 Expressions | cv-combined type | The cv-combined type of two types T1 and T2 is a type T3 similar to T1 whose cv-qualification signature (4.4) is:  — for every j > 0, cv3,j is the union of cv1,j and cv2,j ;  — if the resulting cv3,j is different from cv1,j or cv2,j , then const is added to every cv3,k for 0 < k < j.  [Note: Given similar types T1 and T2, this construction ensures that both can be converted to T3. —end note ] |
| 5 Expressions | composite pointer type | The composite pointer type of two operands p1 and p2 having types T1 and T2, respectively, where at least one is a pointer or pointer to member type or std::nullptr\_t, is:  — if both p1 and p2 are null pointer constants, std::nullptr\_t;  — if either p1 or p2 is a null pointer constant, T2 or T1, respectively;  — if T1 or T2 is "pointer to cv1 void" and the other type is "pointer to cv2 T", "pointer to cv12 void", where cv12 is the union of cv1 and cv2;  — if T1 is "pointer to cv1 C1" and T2 is "pointer to cv2 C2", where C1 is reference-related to C2 or C2 is reference-related to C1 (8.5.3), the cv-combined type of T1 and T2 or the cv-combined type of T2 and T1, respectively;  — if T1 is "pointer to member of C1 of type cv1 U1" and T2 is "pointer to member of C2 of type cv2 U2" where C1 is reference-related to C2 or C2 is reference-related to C1 (8.5.3), the cv-combined type of T2 and T1 or the cv-combined type of T1 and T2, respectively;  — if T1 and T2 are similar multi-level mixed pointer and pointer to member types (4.4), the cv-combined type of T1 and T2;  — otherwise, a program that necessitates the determination of a composite pointer type is ill-formed. |
| 5.1.1 General | primary-expression  id-expression  unqualified-id | primary-expression:  　　literal  　　this  　　( expression )  　　id-expression  　　lambda-expression  id-expression:  　　unqualified-id  　　qualified-id  unqualified-id:  　　identifier  　　operator-function-id  　　conversion-function-id  　　literal-operator-id  　　~ class-name  　　~ decltype-specifier  　　template-id |
| 5.1.1 General | this | The keyword this names a pointer to the object for which a non-static member function (9.3.2) is invoked or a non-static data member's initializer (9.2) is evaluated. |
| 5.1.1 General | parenthesized expression | A parenthesized expression is a primary expression whose type and value are identical to those of the enclosed expression. |
| 5.1.1 General | identifier | An identifier is an id-expression provided it has been suitably declared (Clause 7). [Note: for operator-function-ids, see 13.5; for conversion-function-ids, see 12.3.2; for literal-operator-ids, see 13.5.8; for template-ids, see 14.2. A class-name or decltype-specifier prefixed by ~ denotes a destructor; see 12.4. Within the definition of a non-static member function, an identifier that names a non-static member is transformed to a class member access expression (9.3.1). —end note ] |
| 5.1.1 General | qualified-id  nested-name-specifier | qualified-id:  　　nested-name-specifier templateopt unqualified-id  nested-name-specifier:  　　::  　　type-name ::  　　namespace-name ::  　　decltype-specifier ::  　　nested-name-specifier identifier ::  　　nested-name-specifier templateopt simple-template-id :: |
| 5.1.2 Lambda expressions | lambda-expression  lambda-introducer  lambda-capture  capture-default  capture-list  capture  simple-capture  init-capture  lambda-declarator | lambda-expression:  　　lambda-introducer lambda-declaratoropt compound-statement  lambda-introducer:  　　[ lambda-captureopt ]  lambda-capture:  　　capture-default  　　capture-list  　　capture-default , capture-list  capture-default:  　　&  　　=  capture-list:  　　capture ...opt  　　capture-list , capture ...opt  capture:  　　simple-capture  　　init-capture  simple-capture:  　　identifier  　　& identifier  　　this  init-capture:  　　identifier initializer  　　& identifier initializer  lambda-declarator:  　　( parameter-declaration-clause ) mutableopt exception-specificationopt attribute-specifier-seqopt trailing-return-typeopt |
| 5.1.2 Lambda expressions | closure object | The evaluation of a lambda-expression results in a prvalue temporary (12.2). This temporary is called the closure object. |
| 5.1.2 Lambda expressions | closure type | The type of the lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type — called the closure type — whose properties are described below. |
| 5.1.2 Lambda expressions | local lambda expression | A lambda-expression whose smallest enclosing scope is a block scope (3.3.3) is a local lambda expression;  any other lambda-expression shall not have a capture-default or simple-capture in its lambda-introducer. |
| 5.1.2 Lambda expressions | reaching scope | The reaching scope of a local lambda expression is the set of enclosing scopes up to and including the  innermost enclosing function and its parameters. [Note: This reaching scope includes any intervening  lambda-expressions. —end note ] |
| 5.1.2 Lambda expressions | explicitly captured | An entity that is designated by a simple-capture is said to be explicitly  captured, and shall be this or a variable with automatic storage duration declared in the reaching scope of  the local lambda expression. |
| 5.1.2 Lambda expressions | implicitly capture | A lambda-expression with an associated capture-default that does not explicitly capture this or a variable  with automatic storage duration (this excludes any id-expression that has been found to refer to an init-  capture’s associated non-static data member), is said to implicitly capture the entity (i.e., this or a variable)  if the compound-statement:  — odr-uses (3.2) the entity, or  — names the entity in a potentially-evaluated expression (3.2) where the enclosing full-expression depends  on a generic lambda parameter declared within the reaching scope of the lambda-expression. |
| 5.1.2 Lambda expressions | captured | An entity is captured if it is captured explicitly or implicitly. |
| 5.1.2 Lambda expressions | captured by copy | An entity is captured by copy if it is implicitly captured and the capture-default is = or if it is explicitly  captured with a capture that is not of the form & identifier or & identifier initializer. |
| 5.1.2 Lambda expressions | captured by reference | An entity is captured by reference if it is implicitly or explicitly captured but not captured by copy. |
| 5.2 Postfix expressions | postfix-expression  expression-list  pseudo-destructor-name | postfix-expression:  　　primary-expression  　　postfix-expression [ expression ]  　　postfix-expression [ braced-init-list ]  　　postfix-expression ( expression-listopt )  　　simple-type-specifier ( expression-listopt )  　　typename-specifier ( expression-listopt )  　　simple-type-specifier braced-init-list  　　typename-specifier braced-init-list  　　postfix-expression . templateopt id-expression  　　postfix-expression -> templateopt id-expression  　　postfix-expression . pseudo-destructor-name  　　postfix-expression -> pseudo-destructor-name  　　postfix-expression ++  　　postfix-expression --  　　dynamic\_cast < type-id > ( expression )  　　static\_cast < type-id > ( expression )  　　reinterpret\_cast < type-id > ( expression )  　　const\_cast < type-id > ( expression )  　　typeid ( expression )  　　typeid ( type-id )  expression-list:  　　initializer-list  pseudo-destructor-name:  　　nested-name-specifieropt type-name :: ~ type-name  　　nested-name-specifier template simple-template-id :: ~ type-name  　　nested-name-specifieropt ~ type-name  　　~ decltype-specifier |
| 5.2.2 Function call | function call | A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated  list of initializer-clauses which constitute the arguments to the function. |
| 5.2.2 Function call | virtual function call | If the selected function is non-virtual, or if the  id-expression in the class member access expression is a qualified-id, that function is called. Otherwise, its  final overrider (10.3) in the dynamic type of the object expression is called; such a call is referred to as a  virtual function call. [Note: the dynamic type is the type of the object referred to by the current value of  the object expression. 12.7 describes the behavior of virtual function calls when the object expression refers  to an object under construction or destruction. —end note ] |
| 5.2.2 Function call | nonconstant object | In addition, it is possible to modify the values of nonconstant objects  through pointer parameters. |
| 5.2.2 Function call | default argument promotion | ...  These promotions are referred to as the default argument promotions. |
| 5.2.5 Class member access | object expression | Abbreviating postfix-expression.id-expression as E1.E2, E1 is called the object expression. |
| 5.2.5 Class member access | cq  vq | In the remainder of 5.2.5, cq represents either const  or the absence of const and vq represents either volatile or the absence of volatile. |
| 5.2.5 Class member access | vq12 | Let the notation vq12 stand  for the “union” of vq1 and vq2; that is, if vq1 or vq2 is volatile, then vq12 is volatile. |
| 5.2.5 Class member access | cq12 | Similarly,  let the notation cq12 stand for the “union” of cq1 and cq2; that is, if cq1 or cq2 is const, then cq12  is const. |
| 5.2.7 Dynamic cast | fail | If C is the class type to which T points or refers, the run-time check logically executes as follows:  — If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class  subobject of a C object, and if only one object of type C is derived from the subobject pointed (referred)  to by v the result points (refers) to that C object.  — Otherwise, if v points (refers) to a public base class subobject of the most derived object, and the  type of the most derived object has a base class, of type C, that is unambiguous and public, the result  points (refers) to the C subobject of the most derived object.  — Otherwise, the run-time check fails. |
| 5.2.10 Reinterpret cast | type pun | A glvalue expression of type T1 can be cast to the type “reference to T2” if an expression of type “pointer  to T1” can be explicitly converted to the type “pointer to T2” using a reinterpret\_cast. The result refers  to the same object as the source glvalue, but with the specified type. [Note: That is, for lvalues, a reference  cast reinterpret\_cast<T&>(x) has the same effect as the conversion \*reinterpret\_cast<T\*>(&x) with  the built-in & and \* operators (and similarly for reinterpret\_cast<T&&>(x)). —end note ] No temporary  is created, no copy is made, and constructors (12.1) or conversion functions (12.3) are not called. 73  73) This is sometimes referred to as a type pun. |
| 5.2.11 Const cast | casting away constness | The following rules define the process known as casting away constness. In these rules Tn and Xn represent  types. For two pointer types:  X1 is T1cv1,1 \* ··· cv1,N \* where T1 is not a pointer type  X2 is T2cv2,1 \* ··· cv2,M \* where T2 is not a pointer type  K is min(N,M)  casting from X1 to X2 casts away constness if, for a non-pointer type T there does not exist an implicit  conversion (Clause 4) from:  Tcv1,(N-K+1) \* cv1,(N-K+2) \* ··· cv1,N \*  to  Tcv2,(M-K+1) \* cv2,(M-K+2) \* ··· cv2,M \* |
| 5.3 Unary expressions | unary-expression  unary-operator | unary-expression:  　　postfix-expression  　　++ cast-expression  　　-- cast-expression  　　unary-operator cast-expression  　　sizeof unary-expression  　　sizeof ( type-id )  　　sizeof ... ( identifier )  　　alignof ( type-id )  　　noexcept-expression  　　new-expression  　　delete-expression  unary-operator: one of  　　\* & + - ! ~ |
| 5.3.1 Unary operators | indirection | The unary \* operator performs indirection: the expression to which it is applied shall be a pointer to an  object type, or a pointer to a function type and the result is an lvalue referring to the object or function  to which the expression points. |
| 5.3.1 Unary operators | negative of an unsigned quantity | The  negative of an unsigned quantity is computed by subtracting its value from 2n , where n is the number of  bits in the promoted operand. |
| 5.3.3 Sizeof | sizeof operator | The sizeof operator yields the number of bytes in the object representation of its operand. |
| 5.3.4 New | new-expression  allocated type | The new-expression attempts to create an object of the type-id (8.1) or new-type-id to which it is applied.  The type of that object is the allocated type. |
| 5.3.4 New | new-expression  new-placement  new-type-id  new-declarator  noptr-new-declarator  new-initializer | new-expression:  　　::opt new new-placementopt new-type-id new-initializeropt  　　::opt new new-placementopt ( type-id ) new-initializeropt  new-placement:  　　( expression-list )  new-type-id:  　　type-specifier-seq new-declaratoropt  new-declarator:  　　ptr-operator new-declaratoropt  　　noptr-new-declarator  noptr-new-declarator:  　　[ expression ] attribute-specifier-seqopt  　　noptr-new-declarator [ constant-expression ] attribute-specifier-seqopt  new-initializer:  　　( expression-listopt )  　　braced-init-list |
| 5.3.4 New | dynamic storage duration | Entities created by a new-expression have dynamic storage duration (3.7.4). [Note: the lifetime of such  an entity is not necessarily restricted to the scope in which it is created. —end note ] |
| 5.3.5 Delete | delete-expression | The delete-expression operator destroys a most derived object (1.8) or array created by a new-expression. |
| 5.3.5 Delete | delete-expression | delete-expression:  　　::opt delete cast-expression  　　::opt delete [ ] cast-expression |
| 5.3.5 Delete | delete object | In the first alternative (delete object), the value of the operand of delete may be a null pointer  value, a pointer to a non-array object created by a previous new-expression, or a pointer to a subobject (1.8)  representing a base class of such an object (Clause 10). |
| 5.3.5 Delete | delete array | In the second  alternative (delete array), the value of the operand of delete may be a null pointer value or a pointer  value that resulted from a previous array new-expression. 81  81) For non-zero-length arrays, this is the same as a pointer to the first element of the array created by that new-expression.  Zero-length arrays do not have a first element. |
| 5.3.6 Alignof | alignof expression | An alignof expression yields the alignment requirement of its operand type. |
| 5.3.7 noexcept operator | noexcept operator | The noexcept operator determines whether the evaluation of its operand, which is an unevaluated operand  (Clause 5), can throw an exception (15.1). |
| 5.3.7 noexcept operator | noexcept-expression | noexcept-expression:  　　noexcept ( expression ) |
| 5.4 Explicit type conversion (cast notation) | cast notation | An explicit type conversion can be expressed using functional notation (5.2.3), a type conversion operator  (dynamic\_cast, static\_cast, reinterpret\_cast, const\_cast), or the cast notation. |
| 5.4 Explicit type conversion (cast notation) | cast-expression | cast-expression:  　　unary-expression  　　( type-id ) cast-expression |
| 5.5 Pointer-to-member operators | pm-expression | pm-expression:  　　cast-expression  　　pm-expression .\* cast-expression  　　pm-expression ->\* cast-expression |
| 5.5 Pointer-to-member operators | object expression | Abbreviating pm-expression.\*cast-expression as E1.\*E2, E1 is called the object expression. |
| 5.6 Multiplicative operators | multiplicative-expression | multiplicative-expression:  　　pm-expression  　　multiplicative-expression \* pm-expression  　　multiplicative-expression / pm-expression  　　multiplicative-expression % pm-expression |
| 5.7 Additive operators | additive-expression | additive-expression:  　　multiplicative-expression  　　additive-expression + multiplicative-expression  　　additive-expression - multiplicative-expression |
| 5.8 Shift operators | shift-expression | shift-expression:  　　additive-expression  　　shift-expression << additive-expression  　　shift-expression >> additive-expression |
| 5.9 Relational operators | relational-expression | relational-expression:  　　shift-expression  　　relational-expression < shift-expression  　　relational-expression > shift-expression  　　relational-expression <= shift-expression  　　relational-expression >= shift-expression |
| 5.10 Equality operators | equality-expression | equality-expression:  　　relational-expression  　　equality-expression == relational-expression  　　equality-expression != relational-expression |
| 5.11 Bitwise AND operator | and-expression | and-expression:  　　equality-expression  　　and-expression & equality-expression |
| 5.12 Bitwise exclusive OR operator | exclusive-or-expression | exclusive-or-expression:  　　and-expression  　　exclusive-or-expression ˆ and-expression |
| 5.13 Bitwise inclusive OR operator | inclusive-or-expression | inclusive-or-expression:  　　exclusive-or-expression  　　inclusive-or-expression | exclusive-or-expression |
| 5.14 Logical AND operator | logical-and-expression | logical-and-expression:  　　inclusive-or-expression  　　logical-and-expression && inclusive-or-expression |
| 5.14 Logical AND operator | left-to-right evaluation | Unlike &, && guarantees left-to-right  evaluation: the second operand is not evaluated if the first operand is false. |
| 5.15 Logical OR operator | logical-or-expression | logical-or-expression:  　　logical-and-expression  　　logical-or-expression || logical-and-expression |
| 5.15 Logical OR operator | left-to-right evaluation | Unlike |, || guarantees left-to-right  evaluation; moreover, the second operand is not evaluated if the first operand evaluates to true. |
| 5.16 Conditional operator | conditional-expression | conditional-expression:  　　logical-or-expression  　　logical-or-expression ? expression : assignment-expression |
| 5.17 Assignment and compound assignment operators | assignment-expression  assignment-operator | assignment-expression:  　　conditional-expression  　　logical-or-expression assignment-operator initializer-clause  　　throw-expression  assignment-operator: one of  　　= \*= /= %= += -= >>= <<= &= ˆ= |= |
| 5.18 Comma operator | expression | expression:  　　assignment-expression  　　expression , assignment-expression |
| 5.19 Constant expressions | constant expressions | Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause; other  contexts have different semantics depending on whether or not an expression satisfies these requirements.  Expressions that satisfy these requirements are called constant expressions. [Note: Constant expressions  can be evaluated during translation.—end note ] |
| 5.19 Constant expressions | constant-expression | constant-expression:  　　conditional-expression |
| 5.19 Constant expressions | core constant expression | A conditional-expression e is a core constant expression unless the evaluation of e, following the rules of the  abstract machine (1.9), would evaluate one of the following expressions:  ... |
| 5.19 Constant expressions | integral constant expression | An integral constant expression is an expression of integral or unscoped enumeration type, implicitly con-  verted to a prvalue, where the converted expression is a core constant expression. [Note: Such expressions  may be used as array bounds (8.3.4, 5.3.4), as bit-field lengths (9.6), as enumerator initializers if the un-  derlying type is not fixed (7.2), and as alignments (7.6.2). —end note ] |
| 5.19 Constant expressions | converted constant expression | A converted constant expression of  type T is an expression, implicitly converted to a prvalue of type T, where the converted expression is a core  constant expression and the implicit conversion sequence contains only user-defined conversions, lvalue-to-  rvalue conversions (4.1), integral promotions (4.5), and integral conversions (4.7) other than narrowing con-  versions (8.5.4). [Note: such expressions may be used in new expressions (5.3.4), as case expressions (6.4.2),  as enumerator initializers if the underlying type is fixed (7.2), as array bounds (8.3.4), and as integral or  enumeration non-type template arguments (14.3). —end note ] |
| 5.19 Constant expressions | constant expression | A constant expression is either a glvalue core constant expression whose value refers to an object with static  storage duration or to a function, or a prvalue core constant expression whose value is an object where, for  that object and its subobjects:  — each non-static data member of reference type refers to an object with static storage duration or to a  function, and  — if the object or subobject is of pointer type, it contains the address of an object with static storage  duration, the address past the end of such an object (5.7), the address of a function, or a null pointer  value. |
| 6 Statements | statement | statement:  　　labeled-statement  　　attribute-specifier-seqopt expression-statement  　　attribute-specifier-seqopt compound-statement  　　attribute-specifier-seqopt selection-statement  　　attribute-specifier-seqopt iteration-statement  　　attribute-specifier-seqopt jump-statement  　　declaration-statement  　　attribute-specifier-seqopt try-block |
| 6.1 Labeled statement | labeled-statement | labeled-statement:  　　attribute-specifier-seqopt identifier : statement  　　attribute-specifier-seqopt case constant-expression : statement  　　attribute-specifier-seqopt default : statement |
| 6.1 Labeled statement | identifier label | An identifier label declares the identifier. |
| 6.2 Expression statement | expression-statement | expression-statement:  　　expressionopt ; |
| 6.2 Expression statement | null statement | An expression statement with the expression missing  is called a null statement. [Note: Most statements are expression statements — usually assignments or  function calls. A null statement is useful to carry a label just before the } of a compound statement and to  supply a null body to an iteration statement such as a while statement (6.5.1). —end note ] |
| 6.3 Compound statement or block | compound statement  block | So that several statements can be used where one is expected, the compound statement (also, and equiva-  lently, called “block”) is provided. |
| 6.3 Compound statement or block | compound-statement  statement-seq | compound-statement:  　　{ statement-seqopt }  statement-seq:  　　statement  　　statement-seq statement |
| 6.4 Selection statements | selection statement | Selection statements choose one of several flows of control. |
| 6.4 Selection statements | selection-statement  condition | selection-statement:  　　if ( condition ) statement  　　if ( condition ) statement else statement  　　switch ( condition ) statement  condition:  　　expression  　　attribute-specifier-seqopt decl-specifier-seq declarator = initializer-clause  　　attribute-specifier-seqopt decl-specifier-seq declarator braced-init-list |
| 6.4 Selection statements | substatement | In Clause 6, the term substatement refers to  the contained statement or statements that appear in the syntax notation. |
| 6.4 Selection statements | the condition | The value of the condition will be referred to as simply “the condition” where the  usage is unambiguous. |
| 6.4.1 The if statement | un-elsed if |  |
| 6.4.2 The switch statement | switch statement | The switch statement causes control to be transferred to one of several statements depending on the value  of a condition. |
| 6.5 Iteration statements | iteration statement | Iteration statements specify looping. |
| 6.5 Iteration statements | iteration-statement  for-init-statement  for-range-declaration  for-range-initializer | iteration-statement:  　　while ( condition ) statement  　　do statement while ( expression ) ;  　　for ( for-init-statement conditionopt ; expressionopt ) statement  　　for ( for-range-declaration : for-range-initializer ) statement  for-init-statement:  　　expression-statement  　　simple-declaration  for-range-declaration:  　　attribute-specifier-seqopt decl-specifier-seq declarator  for-range-initializer:  　　expression  　　braced-init-list |
| 6.5.1 The while statement | while statement | In the while statement the substatement is executed repeatedly until the value of the condition (6.4) becomes  false. |
| 6.5.2 The do statement | do statement | In the do statement the substatement is executed repeatedly until the value of the expression becomes false. |
| 6.5.4 The range-based for statement | range-init | For a range-based for statement of the form  for ( for-range-declaration : expression ) statement  let range-init be equivalent to the expression surrounded by parentheses 89  ( expression )  and for a range-based for statement of the form  for ( for-range-declaration : braced-init-list ) statement  let range-init be equivalent to the braced-init-list.  89) this ensures that a top-level comma operator cannot be reinterpreted as a delimiter between init-declarators in the decla-  ration of \_\_range. |
| 6.6 Jump statements | jump statement | Jump statements unconditionally transfer control. |
| 6.6 Jump statements | jump-statement | jump-statement:  　　break ;  　　continue ;  　　return expressionopt ;  　　return braced-init-list ;  　　goto identifier ; |
| 6.6.1 The break statement | break statement | The break statement shall occur only in an iteration-statement or a switch statement and causes termination  of the smallest enclosing iteration-statement or switch statement; control passes to the statement following  the terminated statement, if any. |
| 6.6.2 The continue statement | continue statement | The continue statement shall occur only in an iteration-statement and causes control to pass to the loop-  continuation portion of the smallest enclosing iteration-statement, that is, to the end of the loop. |
| 6.6.3 The return statement | return statement | A function returns to its caller by the return statement. |
| 6.6.4 The goto statement | goto statement | The goto statement unconditionally transfers control to the statement labeled by the identifier. |
| 6.7 Declaration statement | declaration statement  declaration-statement | A declaration statement introduces one or more new identifiers into a block; it has the form  declaration-statement:  　　block-declaration |
| 6.7 Declaration statement | jump | A  program that jumps 90 from a point where a variable with automatic storage duration is not in scope to a  point where it is in scope is ill-formed unless the variable has scalar type, class type with a trivial default  constructor and a trivial destructor, a cv-qualified version of one of these types, or an array of one of the  preceding types and is declared without an initializer (8.5).  90) The transfer from the condition of a switch statement to a case label is considered a jump in this respect. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| 25.2.4 For each | nonconstant function | If the type of first satisfies the requirements of a  mutable iterator, f may apply nonconstant functions through the dereferenced iterator. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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. |