1(menu)视图>>阅读视图

2双击表格

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *attribute-specifier-seq* | | | | | | | | |
| ( | *attribute-specifier* | | | | | | | )+ |
|  | ( | [ [ | *attribute-list* | ] ] | | | *alignment-specifier* | ) |  |
|  |  |  | (*attribute*( ...)?)?(,(*attribute*( ...)?)?)\* |  |  | (**alignas** ( *type-id*( ...)? )  |**alignas** ( *constant-expression*( ...)? )  ) |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *attribute* | | | | | | | | | | | | | |
| *attribute-token* | | | | | | | ( | *attribute-argument-clause* | | | | | )? |
| ( | *identifier* | | | *attribute-scoped-token* | | | ) |  | ( | *balanced-token-seq* | | | ) |  |
|  |  |  | *attribute-namespace* | :: | *identifier* |  |  |  | ( | *balanced-token* | )\* |  |  |
|  |  |  | *identifier* |  |  |  |  |  |  | (( *balanced-token-seq* )  |[ *balanced-token-seq* ]  |{ *balanced-token-seq* }  |any *token* other than a parenthesis, a bracket, or a brace  ) |  |  |  |

|  |
| --- |
| attribute-specifier-seq:  　　attribute-specifier-seq opt attribute-specifier  attribute-specifier:  　　[ [ attribute-list ] ]  　　alignment-specifier  alignment-specifier:  　　alignas ( type-id ... opt )  　　alignas ( constant-expression ... opt )  attribute-list:  　　attribute opt  　　attribute-list , attribute opt  　　attribute ...  　　attribute-list , attribute ...  attribute:  　　attribute-token attribute-argument-clause opt  attribute-token:  　　identifier  　　attribute-scoped-token  attribute-scoped-token:  　　attribute-namespace :: identifier  attribute-namespace:  　　identifier  attribute-argument-clause:  　　( balanced-token-seq )  balanced-token-seq:  　　balanced-token opt  　　balanced-token-seq balanced-token  balanced-token:  　　( balanced-token-seq )  　　[ balanced-token-seq ]  　　{ balanced-token-seq }  　　any token other than a parenthesis, a bracket, or a brace |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *init-declarator-list* | | | | | | | | | | |
| *init-declarator* | | | | | | | | | | (,*init-declarator*)\* |
| *declarator* | | | | | | | ( | *initializer* | )? |  |
| ( | *ptr-declarator* | | | *noptr-declarator* | *parameters-and-qualifiers* | *trailing-return-type* | ) |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *cv-qualifier-seq* | | | | | | |
| ( | *cv-qualifier* | | | | | )+ |
|  | ( | **const** | | | **volatile** | ) |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ref-qualifier* | | | | |
| ( | **&** | | | **&&** | ) |

|  |  |  |  |
| --- | --- | --- | --- |
| *ptr-declarator* | | | |
| ( | *ptr-operator* | )\* | *noptr-declarator* |
|  | (**\***( *attribute-specifier-seq*)?( *cv-qualifier-seq*)?  |**&**( *attribute-specifier-seq*)?  |**&&**( *attribute-specifier-seq*)?  |*nested-name-specifier* **\***( *attribute-specifier-seq*)?( *cv-qualifier-seq*)?  ) |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *noptr-declarator* | | | | | | |
| ( | *declarator-id* | ( *attribute-specifier-seq*)? | | | ( *ptr-declarator* ) | ) | (*parameters-and-qualifiers*  |[( *constant-expression*)? ]( *attribute-specifier-seq*)?  )\* |
|  | (...)? *id-expression* |  |  |  |  |  |

|  |
| --- |
| *parameters-and-qualifiers* |
| ( *parameter-declaration-clause* )( *cv-qualifier-seq*)?( *ref-qualifier*)?( *exception-specification*)?( *attribute-specifier-seq*)? |

|  |
| --- |
| *trailing-return-type* |
| -> *trailing-type-specifier-seq*( *abstract-declarator*)? |

|  |  |  |  |
| --- | --- | --- | --- |
| *type-id* | | | |
| *type-specifier-seq* | ( | *abstract-declarator* | )? |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *abstract-declarator* | | | | | | |
| ( | *ptr-abstract-declarator* | | | (*noptr-abstract-declarator*)? *parameters-and-qualifiers* *trailing-return-type* | | | *abstract-pack-declarator* | ) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *ptr-abstract-declarator* | | | | | |
| ( | *noptr-abstract-declarator* | | | | (*ptr-operator* )+(*noptr-abstract-declarator*)? | ) |
|  | (( *ptr-abstract-declarator* ))? | (*parameters-and-qualifiers*  |[( *constant-expression*)? ]( *attribute-specifier-seq*)?  )\* |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *abstract-pack-declarator* | | | | | | | | | |
| ( | *ptr-operator* | )\* | *noptr-abstract-pack-declarator* | | | | | | |
|  |  |  | ( | *parameters-and-qualifiers* | | | [( *constant-expression*)? ]( *attribute-specifier-seq*)? | | | ... | )\* |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *parameter-declaration-clause* | | | | | | | | | | | | |
| ( | ( | *parameter-declaration-list* | | | | )? | ( | ... | )? | | | parameter-declaration-list , ... | ) |
|  |  | *parameter-declaration* | ( | , *parameter-declaration* | )\* |  |  |  |  |  |  |  |
|  |  | (*attribute-specifier-seq*)? *decl-specifier-seq*  (*declarator*  |*abstract-declarator*  )  ( = *initializer-clause*)? |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *function-definition* | | | | | | | | | | |
| ( | *attribute-specifier-seq* | )? | ( | *decl-specifier-seq* | )? | *declarator* | ( | *virt-specifier-seq* | )? | *function-body* |
|  |  |  |  |  |  |  |  |  |  | ((*ctor-initializer*)? *compound-statement*  |*function-try-block*  |= **default** ;  |= **delete** ;  ) |

|  |
| --- |
| init-declarator-list:  　　init-declarator  　　init-declarator-list , init-declarator  init-declarator:  　　declarator initializer opt  declarator:  　　ptr-declarator  　　noptr-declarator parameters-and-qualifiers trailing-return-type  ptr-declarator:  　　noptr-declarator  　　ptr-operator ptr-declarator  noptr-declarator:  　　declarator-id attribute-specifier-seq opt  　　noptr-declarator parameters-and-qualifiers  　　noptr-declarator [ constant-expression opt ] attribute-specifier-seq opt  　　( ptr-declarator )  parameters-and-qualifiers:  　　( parameter-declaration-clause ) cv-qualifier-seq opt ref-qualifier opt exception-specification opt attribute-specifier-seq opt  trailing-return-type:  　　-> trailing-type-specifier-seq abstract-declarator opt  ptr-operator:  　　\* attribute-specifier-seq opt cv-qualifier-seq opt  　　& attribute-specifier-seq opt  　　&& attribute-specifier-seq opt  　　nested-name-specifier \* attribute-specifier-seq opt cv-qualifier-seq opt  cv-qualifier-seq:  　　cv-qualifier cv-qualifier-seq opt  cv-qualifier:  　　const  　　volatile  ref-qualifier:  　　&  　　&&  declarator-id:  　　... opt id-expression  type-id:  　　type-specifier-seq abstract-declarator opt  abstract-declarator:  　　ptr-abstract-declarator  　　noptr-abstract-declarator opt parameters-and-qualifiers trailing-return-type  　　abstract-pack-declarator  ptr-abstract-declarator:  　　noptr-abstract-declarator  　　ptr-operator ptr-abstract-declarator opt  noptr-abstract-declarator:  　　noptr-abstract-declarator opt parameters-and-qualifiers  　　noptr-abstract-declarator opt [ constant-expression opt ] attribute-specifier-seq opt  　　( ptr-abstract-declarator )  abstract-pack-declarator:  　　noptr-abstract-pack-declarator  　　ptr-operator abstract-pack-declarator  noptr-abstract-pack-declarator:  　　noptr-abstract-pack-declarator parameters-and-qualifiers  　　noptr-abstract-pack-declarator [ constant-expression opt ] attribute-specifier-seq opt  parameter-declaration-clause:  　　parameter-declaration-list opt ... opt  　　parameter-declaration-list , ...  parameter-declaration-list:  　　parameter-declaration  　　parameter-declaration-list , parameter-declaration  parameter-declaration:  　　attribute-specifier-seq opt decl-specifier-seq declarator  　　attribute-specifier-seq opt decl-specifier-seq declarator = initializer-clause  　　attribute-specifier-seq opt decl-specifier-seq abstract-declarator opt  　　attribute-specifier-seq opt decl-specifier-seq abstract-declarator opt = initializer-clause  function-definition:  　　attribute-specifier-seq opt decl-specifier-seq opt declarator virt-specifier-seq opt function-body  function-body:  　　ctor-initializer opt compound-statement  　　function-try-block  　　= default ;  　　= delete ;  initializer:  　　brace-or-equal-initializer  　　( expression-list )  brace-or-equal-initializer:  　　= initializer-clause  　　braced-init-list  initializer-clause:  　　assignment-expression  　　braced-init-list  initializer-list:  initializer-clause ... opt  　　initializer-list , initializer-clause ... opt  braced-init-list:  　　{ initializer-list , opt }  　　{ } |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *initializer* | | | | | | | | | | | | | | | | | | | | |
| ( | *brace-or-equal-initializer* | | | | | | | | | | | | | | | | | | | ( *expression-list* ) | ) |
|  | ( | = | *initializer-clause* | | | | | | | *braced-init-list* | | | | | | | | ) |  |  |  |
|  |  |  | ( | *assignment-expression* | | | *braced-init-list* | ) |  | { | ( | *initializer-list* | ( | , | )? | )? | } |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | *initializer-clause*( ...)?  ( , *initializer-clause*( ...)?)\* |  |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| 5 Expressions | If an expression initially has the type “reference to T” (8.3.2, 8.5.3), the type is adjusted to T prior to  any further analysis. |
| 5 Expressions | If a prvalue initially has the type “cv T,” where T is a cv-unqualified non-class, non-array type, the type of  the expression is adjusted to T prior to any further analysis. |

|  |  |
| --- | --- |
| 2.14.7 Pointer literals | It is a prvalue of type std::nullptr\_t. |
| 4 Standard conversions | The result is an lvalue if T is an  lvalue reference type or an rvalue reference to function type (8.3.2), an xvalue if T is an rvalue reference to  object type, and a prvalue otherwise. |
| 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 |
| 4.1 Lvalue-to-rvalue conversion | In all other cases, the result of the conversion is determined according to the following  rules:  — If T is (possibly cv-qualified) std::nullptr\_t, the result is a null pointer constant (4.10).  — Otherwise, if T has a class type, the conversion copy-initializes a temporary of type T from the glvalue  and the result of the conversion is a prvalue for the temporary.  — Otherwise, if the object to which the glvalue refers contains an invalid pointer value (3.7.4.2, 3.7.4.3),  the behavior is implementation-defined.  — Otherwise, the value contained in the object indicated by the glvalue is the prvalue result. |
| 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”. The result is a pointer to the first element of the array. |
| 4.3 Function-to-pointer conversion | An lvalue of function type T can be converted to a prvalue of type “pointer to T.” The result is a pointer to  the function. 57  57) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function  cannot be obtained. |
| 5 Expressions | [Note: An expression is an xvalue if it is:  — the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference  to object type,  — a cast to an rvalue reference to object type,  — a class member access expression designating a non-static data member of non-reference type in which  the object expression is an xvalue, or  — a .\* pointer-to-member expression in which the first operand is an xvalue and the second operand is  a pointer to data member.  In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue  references to objects are treated as xvalues; rvalue references to functions are treated as lvalues whether  named or not. —end note ] |
| 5.1.1 General | A string literal is an lvalue; all other  literals are prvalues. |
| 5.1.1 General | If a declaration declares a member function or member function template of a class X, the expression this  is a prvalue of type “pointer to cv-qualifier-seq X” between the optional cv-qualifer-seq and the end of the  function-definition, member-declarator, or declarator. |
| 5.1.1 General | Otherwise, if a member-declarator declares a non-static data member (9.2) of a class X, the expression this  is a prvalue of type “pointer to X” within the optional brace-or-equal-initializer. It shall not appear elsewhere  in the member-declarator. |
| 5.1.1 General | The presence of parentheses does not affect whether the expression is an lvalue. |
| 5.1.1 General | The result is an lvalue if the entity is a function, variable,  or data member and a prvalue otherwise. |
| 5.1.1 General | The result is an lvalue if the member is a static  member function or a data member and a prvalue otherwise. |
| 5.1.1 General | The result is an lvalue if the member is a  function or a variable and a prvalue otherwise. |
| 5.1.1 General | The type of the  result is the type of the enumeration. The result is a prvalue. |
| 5.1.2 Lambda expressions | [Note: The cast ensures that the  transformed expression is a prvalue. —end note ] |
| 5.2.1 Subscripting | The  expression E1[E2] is identical (by definition) to \*((E1)+(E2)) [Note: see 5.3 and 5.7 for details of \* and  + and 8.3.4 for details of arrays. —end note ], except that in the case of an array operand, the result is an  lvalue if that operand is an lvalue and an xvalue otherwise. |
| 5.2.2 Function call | A function call is an lvalue if the result type is an lvalue reference type or an rvalue reference to function  type, an xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise. |
| 5.2.2 Function call | If a function call is a prvalue of object type:  — if the function call is either  — the operand of a decltype-specifier or  — the right operand of a comma operator that is the operand of a decltype-specifier,  a temporary object is not introduced for the prvalue. The type of the prvalue may be incomplete.  [Note: as a result, storage is not allocated for the prvalue and it is not destroyed; thus, a class type is  not instantiated as a result of being the type of a function call in this context. This is true regardless of  whether the expression uses function call notation or operator notation (13.3.1.2). —end note ] [Note:  unlike the rule for a decltype-specifier that considers whether an id-expression is parenthesized (7.1.6.2),  parentheses have no special meaning in this context. —end note ]  — otherwise, the type of the prvalue shall be complete. |
| 5.2.5 Class member access | If E2 is declared to have type “reference to T,” then E1.E2 is an lvalue; the type of E1.E2 is T. Otherwise,  one of the following rules applies.  — If E2 is a static data member and the type of E2 is T, then E1.E2 is an lvalue; the expression designates  the named member of the class. The type of E1.E2 is T.  — If E2 is a non-static data member and the type of E1 is “cq1 vq1 X”, and the type of E2 is “cq2 vq2  T”, the expression designates the named member of the object designated by the first expression. If  E1 is an lvalue, then E1.E2 is an lvalue; otherwise E1.E2 is an xvalue. Let the notation vq12 stand  for the “union” of vq1 and vq2; that is, if vq1 or vq2 is volatile, then vq12 is volatile. 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. If E2 is declared to be a mutable member, then the type of E1.E2 is “vq12 T”. If E2 is not  declared to be a mutable member, then the type of E1.E2 is “cq12 vq12 T”.  — If E2 is a (possibly overloaded) member function, function overload resolution (13.3) is used to deter-  mine whether E1.E2 refers to a static or a non-static member function.  　　— If it refers to a static member function and the type of E2 is “function of parameter-type-list  returning T”, then E1.E2 is an lvalue; the expression designates the static member function. The  type of E1.E2 is the same type as that of E2, namely “function of parameter-type-list returning  T”.  　　— Otherwise, if E1.E2 refers to a non-static member function and the type of E2 is “function of  parameter-type-list cv ref-qualifier opt returning T”, then E1.E2 is a prvalue. The expression  designates a non-static member function. The expression can be used only as the left-hand  operand of a member function call (9.3). [Note: Any redundant set of parentheses surrounding  the expression is ignored (5.1). —end note ] The type of E1.E2 is “function of parameter-type-list  cv returning T”.  — If E2 is a nested type, the expression E1.E2 is ill-formed.  — If E2 is a member enumerator and the type of E2 is T, the expression E1.E2 is a prvalue. The type of  E1.E2 is T. |
| 5.2.6 Increment and decrement | The result is a  prvalue. |
| 5.2.7 Dynamic cast | If T is a pointer type, v shall be a prvalue of a pointer to complete class type, and the result is a prvalue  of type T. If T is an lvalue reference type, v shall be an lvalue of a complete class type, and the result is  an lvalue of the type referred to by T. If T is an rvalue reference type, v shall be an expression having a  complete class type, and the result is an xvalue of the type referred to by T. |
| 5.2.7 Dynamic cast | Similarly, if T is “reference to cv1 B” and  v has type cv2 D such that B is a base class of D, the result is the unique B subobject of the D object referred  to by v.  67  The result is an lvalue if T is an lvalue reference, or an xvalue if T is an rvalue reference. |
| 5.2.8 Type identification | The result of a typeid expression is an lvalue of static type const std::type\_info (18.7.1) and dynamic type  const std::type\_info or const name where name is an implementation-defined class publicly derived from  std::type\_info which preserves the behavior described in 18.7.1. |
| 5.2.9 Static cast | If T  is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if T is an rvalue  reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The static\_cast operator  shall not cast away constness (5.2.11). |
| 5.2.10 Reinterpret cast | If T is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression v. |
| 5.2.11 Const cast | If T is an lvalue reference to object type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression v. |
| 5.3.1 Unary operators | 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 | The result of each of the following unary operators is a prvalue. |
| 5.3.1 Unary operators | If the operand is a qualified-id naming a non-static member m of some class C with type T, the result has  type “pointer to member of class C of type T” and is a prvalue designating C::m. Otherwise, if the type of  the expression is T, the result has type “pointer to T” and is a prvalue that is the address of the designated  object (1.7) or a pointer to the designated function. |
| 5.3.2 Increment and decrement | The result is the updated operand; it is an lvalue, and it is a bit-field if  the operand is a bit-field. |
| 5.3.7 noexcept operator | The result of the noexcept operator is a constant of type bool and is a prvalue. |
| 5.4 Explicit type conversion (cast notation) | The result is an lvalue if T is an lvalue reference  type or an rvalue reference to function type and an xvalue if T is an rvalue reference to object type; otherwise  the result is a prvalue. [Note: if T is a non-class type that is cv-qualified, the cv-qualifiers are discarded  when determining the type of the resulting prvalue; see Clause 5. —end note ] |
| 5.5 Pointer-to-member operators | In a .\* expression whose object expression is an rvalue, the program is ill-formed if the second  operand is a pointer to member function with ref-qualifier &. In a .\* expression whose object expression is  an lvalue, the program is ill-formed if the second operand is a pointer to member function with ref-qualifier  &&. The result of a .\* expression whose second operand is a pointer to a data member is an lvalue if the  first operand is an lvalue and an xvalue otherwise. The result of a .\* expression whose second operand is a  pointer to a member function is a prvalue. |
| 7.1.6.2 Simple type specifiers | For an expression e, the type denoted by decltype(e) is defined as follows:  — if e is an unparenthesized id-expression or an unparenthesized class member access (5.2.5), decltype(e)  is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded func-  tions, the program is ill-formed;  — otherwise, if e is an xvalue, decltype(e) is T&&, where T is the type of e;  — otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;  — otherwise, decltype(e) is the type of e. |
| 9.3.2 The this pointer | In the body of a non-static (9.3) member function, the keyword this is a prvalue expression whose value  is the address of the object for which the function is called. |
| 9.6 Bit-fields | [Note: If the initializer for a reference of type const T& is an lvalue that refers to a bit-field,  the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound  to the bit-field directly. See 8.5.3. —end note ] |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |