Core II WP Changes (London) -- troff

Note: the change of "null-pointer constant" to "null pointer constant" in 18.1p4 [lib.support.types] has been passed to the Clause 18 group and is not included here.

*** troff.orig/access Wed Jul 16 04:05:24 1997
***************
*** 377,391 ****
 is public, or
 .LI
 .I m
 as a member of
 .I N
! is private or protected, and the reference occurs in a member or friend of class
 .I N ,
or
 .LI
 there exists a base class
 .I B
 of
 .I N
 that is accessible at the point of reference, and
--- 377,408 ----
 is public, or
 .LI
 .I m
 as a member of
 .I N
! ." Canada _115/ L2953 Canada 7, core-752 class.access.base
! is private, and the reference occurs in a member or friend of class
 .I N ,
or
 .LI
 + .I m
 + as a member of
 + .I N
 + is protected, and the reference occurs in a member or friend of class
 + .I N ,
 + or in a member or friend of a class
 + .I P
 + derived from
 + .I N ,
 + where
 + .I m
 + as a member of
 + .I P
 + is private or protected, or
 + .LI
 there exists a base class
 .I B
 of
that is accessible at the point of reference, and

```c
void fr(B* pb, D1* p1, D2* p2)
{
    pb->i = 1; // ill-formed
    p1->i = 2; // ill-formed
    ! p2->i = 3; // ok (access through a D2)
    ! p2->B::i = 4; // ok (access through a D2, qualification ignored)
    int B::* pmi_B = &B::i; // ill-formed
    int B::* pmi_B2 = &D2::i; // ok (type of &D2::i is "int B::*")
    B::j = 5; // ok (because refers to static member)
    D2::j = 6; // ok (because refers to static member)
}
```

```c
--- 843,854 ----
void fr(B* pb, D1* p1, D2* p2)
{
    pb->i = 1; // ill-formed
    p1->i = 2; // ill-formed
    ! p2->i = 3; // ok (access through a D2)
    ! p2->B::i = 4; // ok (access through a D2, even though
    ! // naming class is B)
    int B::* pmi_B = &B::i; // ill-formed
    int B::* pmi_B2 = &D2::i; // ok (type of &D2::i is "int B::*")
    B::j = 5; // ok (because refers to static member)
    D2::j = 6; // ok (because refers to static member)
}
```

diff -c5 troff.orig/basic troff/basic
*** troff.orig/basic Wed Jul 16 04:05:29 1997
--- troff/basic Wed Jul 16 15:54:59 1997

***************
*** 310,320 ****
an expression is converted (either implicitly or explicitly) to type
.CW T
(_conv_, _expr.type.conv_, _expr.dynamic.cast_, _expr.static.cast_,
_expr.cast_), or
.LI
! an expression is converted to the type
.CW T
or reference to
.CW T
using an implicit conversion (_conv_), a
--- 310,324 ----
! an expression is converted (either implicitly or explicitly) to type
.CW T
(_conv_, _expr.type.conv_, _expr.dynamic.cast_, _expr.static.cast_,
_expr.cast_), or
.LI
! an expression that is not a null pointer constant, and has type
! other than
! .CW "void *",
! is converted to the type
.CW T
or reference to
.CW T
using an implicit conversion (_conv_), a

***************
*** 3824,3836 ****
(_expr.cast_).
An expression of type
.CW void
shall be used only
as an expression statement (_stmt.expr_), as an operand
! of a comma expression (_expr.comma_), or as a second or third operand of
.CW ?:
! (_expr.cond_).
P
 even if the implementation defines two or more basic types to have the
same
value representation, they are nevertheless different types.
.N] e
--- 3828,3845 -----
(_expr.cast_).
An expression of type
.CW void
shall be used only
as an expression statement (_stmt.expr_), as an operand
! of a comma expression (_expr.comma_), as a second or third operand of
.CW ?:
! (_expr.cond_),
! or as the expression in a return statement (_stmt.return_) for a function
! with the return type
! .CW void .
P
 even if the implementation defines two or more basic types to have the
same
value representation, they are nevertheless different types.
.N] e
diff -c5 troff.orig/class troff/class
*** troff.orig/class Wed Jul 16 04:05:31 1997
--- troff/class Wed Jul 16 15:51:08 1997
***************
*** 1631,1641 ****
such a class is called a
.I local
class.
The name of a local class is local to its enclosing scope.
.ix "scope-of local-class
! The local class is in the scope of the enclosing scope.
Declarations in a local class can use only type names, static variables,
.CW extern
variables and functions,
and enumerators from the enclosing scope.
.E[--- 1631,1644 ----
such a class is called a
.I local
class.
The name of a local class is local to its enclosing scope.
.ix "scope-of local-class
! The local class is in the scope of the enclosing scope.
! and has the same access to names outside the function as does the
! enclosing function.
Declarations in a local class can use only type names, static variables,
.CW extern
variables and functions,
and enumerators from the enclosing scope.
An lvalue of type `array of N T('`

or `array of unknown bound of N T('

can be converted to an rvalue of type `pointer to T('`

The result is a pointer to the first element of the array.

A string literal (_lex.string_) that is not a wide string literal can be converted to an rvalue of type `pointer to char('`

a wide string literal can be converted to an rvalue of type `pointer to wchar_t('`. In either case, the result is a pointer to the first element of the array.

For the purpose of ranking in overload resolution (_over.ics.scs_), this conversion is considered an array-to-pointer conversion followed by a qualification.

--- 136,160 ----

this conversion is deprecated. See Annex _depr_.

For the purpose of ranking in overload resolution (_over.ics.scs_), this conversion is considered an array-to-pointer conversion followed by a qualification.

+ ." USA misc L9242 Editorials; core-773 conv.array
+ This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue.

+ ." this conversion is deprecated. See Annex _depr_.
+ For the purpose of ranking in overload resolution (_over.ics.scs_), this conversion is considered an array-to-pointer conversion followed by a qualification.
Each enumeration defines a type that is different from all other types.  

The type of an enumerator is its enumeration.  

The underlying type of its enumeration.  Prior to the closing brace, the type of each enumerator is the type of its initializing value.  If an initializer is specified for an enumerator, the initializing value has the same type as the expression.  Otherwise the type is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.  

The default argument expression has the same semantic constraints as the initializer expression in a declaration of a variable of the parameter type, using the copy-initialization semantics (_dcl.init_).  The names in the expression are bound, and the semantic constraints are checked, at the point of declaration.

In the following code, the parameter type.

In the following code, the parameter type.
A reference to type \(``\f2cv1\fP\ \f5T1\fP\`'

is initialized by an expression of type \(``\f2cv2\fP\ \f5T2\fP\`'

as follows:

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and

If the initializer expression is an lvalue (but not an lvalue for a

bit-field), and
resolution (_over.match_), then

. P
the reference is bound directly to the initializer expression lvalue in
the
first case, and the reference is bound to the lvalue result of the
conversion
in the second case.
! In these cases the reference is said to
. I "bind directly"
 .ix "direct-binding of reference"
to the initializer expression.
 .N[
the usual lvalue-to-rvalue (_conv.lval_), array-to-pointer
]
resolution (_over.match_), then

.P
the reference is bound directly to the initializer expression lvalue in
the
first case, and the reference is bound to the lvalue result of the
conversion
in the second case.
! In these cases the reference is said to
. I "bind directly"
 .ix "direct-binding of reference"
to the initializer expression.
 .N[
the usual lvalue-to-rvalue (_conv.lval_), array-to-pointer
]
resolution (_over.match_), then

.P
the reference is bound directly to the initializer expression lvalue in
the
first case, and the reference is bound to the lvalue result of the
conversion
in the second case.
! In these cases the reference is said to
. I "bind directly"
 .ix "direct-binding of reference"
to the initializer expression.
 .N[
the usual lvalue-to-rvalue (_conv.lval_), array-to-pointer
]
resolution (_over.match_), then

.P
the reference is bound directly to the initializer expression lvalue in
the
first case, and the reference is bound to the lvalue result of the
conversion
in the second case.
! In these cases the reference is said to
. I "bind directly"
 .ix "direct-binding of reference"
to the initializer expression.
 .N[
the usual lvalue-to-rvalue (_conv.lval_), array-to-pointer
]
resolution (_over.match_), then

.P
the reference is bound directly to the initializer expression lvalue in
the
first case, and the reference is bound to the lvalue result of the
conversion
in the second case.
! In these cases the reference is said to
. I "bind directly"
 .ix "direct-binding of reference"
to the initializer expression.
 .N[
the usual lvalue-to-rvalue (_conv.lval_), array-to-pointer
]
is entered.
The order of evaluation of the postfix expression and the argument expression list is unspecified.

Recursive calls are permitted, except to the function named 
(CW main)

A function call is an lvalue if and only if the result type is a reference.

--- 785,795 ----

(CW T)

if

(CW T)

if

***************

--- 1671,1680 ----

(CW B)

need not contain the original member,
the dynamic type of the object on which the pointer to member is dereferenced
must contain the original member; see _expr.mptr.oper_.

+ ." USA 529/_10 L6877 USA core-774 expr.static.cast + .P
+ An rvalue of type `pointer to \f2cv\fP \f5void\fP\('' can be explicitly
+ converted to a pointer to object type. A value of type pointer to object
+ converted to `\('pointer to \f2cv\fP \f5void\fP\('' and back to
+ the original pointer type will have its original value.
The result of the expression
\[ \text{reinterpret\_cast}\langle \text{T}\rangle(v) \]

The \text{reinterpret\_cast} operator shall not cast away constness.

see \_\text{expr\_const\_cast\_} for the definition of ``casting away constness''.

Any expression may be cast to its own type using a
\text{reinterpret\_cast} operator.

The mapping performed by
\text{reinterpret\_cast} is implementation-defined.

---

Any expression may be cast to its own type using a
\text{const\_cast} operator.

The \text{const\_cast} operator shall not cast away constness.

see \_\text{expr\_const\_cast\_} for the definition of ``casting away constness''.

an expression may be cast to its own type using a
\text{const\_cast} operator.

---

For two pointer types
---

No other conversion shall be performed explicitly using
\text{const\_cast}.

---

an expression may be cast to its own type using a
\text{const\_cast} operator.

\text{const\_cast} are listed below.

No other conversion shall be performed explicitly using
\text{const\_cast}.

---

an expression may be cast to its own type using a
\text{const\_cast} operator.

\text{const\_cast} are listed below.

No other conversion shall be performed explicitly using
\text{const\_cast}.

---

an expression may be cast to its own type using a
\text{const\_cast} operator.
For two pointer types

***************
*** 2431,2449 ****
 Type shall not be defined in a
 .CW sizeof
 expression.

 P
 ! The result is a constant of an implementation-defined type
 ! which is the same type as that which is named
 .CW size_t
 ! .ix "implementation-defined type of [size_t]
 ! .ix "implementation-defined [sizeof] expression
 in the standard header
 .CW <cstddef> (_lib.support.types_).
 .ix "[cstddef]
 .ix "[size_t]
 .H3 "New" expr.new
 .P
 .ix "operator~[new];~see~[new]
 .ix "free~store;~see~also~[new],~[delete]
 .ix "memory~management;~see~also~[new],~[delete]

--- 2454,2474 -----
 Type shall not be defined in a
 .CW sizeof
 expression.

 P
 ! ." UK 533/6 L1969 UK Ed-488
 ! The result is a constant of type
 ! .CW size_t.
 ! .N[.
 .CW size_t
 ! is defined
 in the standard header
 .CW <cstddef> (_lib.support.types_).
 .ix "[cstddef]
 .ix "[size_t]
 + .N] e
 .H3 "New" expr.new
 .P
 .ix "operator~[new];~see~[new]
 .ix "free~store;~see~also~[new],~[delete]
 .ix "memory~management;~see~also~[new],~[delete]

***************
*** 2747,2759 ****
 ." UK issue 506
 Their value might vary from one invocation of
 .CW new
 to another.
 .E] e
 ." Storage allocation result
 .P
 ! The allocation function shall either return null or a pointer to a block
 ." UK issue 507
 of storage in which space for the object shall have been reserved.
 .N[ the block of storage is assumed to be appropriately aligned and of the
requested size. The address of the created object will not necessarily
--- 2772,2805 ---
." UK issue 506
Their value might vary from one invocation of
.CW new
to another.
.E] e
+ .P
+ ." USA _127/ L6873 USA public comment 28 expr.new
+ Unless an allocation function is declared with an empty
+ .I exception-specification
+ (_except.spec_)
+ .CW throw()
+ it shall indicate
+ failure to allocate storage by throwing a
+ .ix "[bad_alloc]"
+ .I bad_alloc
+ exception (_except_, _lib.bad.alloc_) and it shall return a non-null
+ pointer otherwise. If the allocation function is declared with an empty
+ .I exception-specification
+ .CW throw()
+ it shall return null to indicate failure to allocate storage and a
+ non-null pointer otherwise. When it returns null,
+ initialization shall not be done, the deallocation function shall not be
+ called, and the value of the
+ .I new-expression
+ shall be null.
." Storage allocation result
.P
! When the allocation function returns a value other than null, it shall be
! a pointer to a block
." UK issue 507
of storage in which space for the object shall have been reserved.
.N[ the block of storage is assumed to be appropriately aligned and of the
requested size. The address of the created object will not necessarily
***************
*** 2834,2849 ****
If the new expression creates an array of objects of class type,
access and ambiguity control are done for the destructor (_class.dtor_).
." Monterey motion 2.e, Jim Welch 95-0104 proposal (1)
." Exceptions
.P
- The allocation function can indicate failure by throwing a
- .CW bad_alloc
- .ix "[bad_alloc]"
- exception (_except_, _lib.bad.alloc_).
- In this case no initialization is done.
- .P
- .ix "exception-and [new]
If any part of the object initialization described above\*f
.Fs
This may include evaluating a
- .I new-initializer
--- 2880,2889 ----
***************
*** 3688,3697 ****
--- 3728,3751 ----
to a null pointer constant
and that any object pointer can be compared
to a pointer to (possibly cv-qualified)
.CW void .
.N] e
Pointers to objects or functions of the same type (after pointer conversions) can be compared,
with a result defined as follows:

- **3992, 4015**
  and is an rvalue.
  
  **N[**
  this includes the case where both operands are
  .I throw-expression s.
  **N] e**
  ! **P**

  Otherwise, if either the second or the third operand has
  ! (possibly cv-qualified)
  ! class or enumeration type, overload resolution is used to determine
  ! the conversions (if any) to be applied to the operands
  ! (_over.match.oper_, _over.built_). The conversions thus determined
  ! are applied, and the converted operands are used in place of the original
  ! operands for the remainder of this section.

  **P**

  If the second and third
  operands are lvalues and have the same type,
  the result is of that type and is an lvalue.

  **P**

  Otherwise, the result is an rvalue.

  Lvalue-to-rvalue (_conv.lval_), array-to-pointer (_conv.array_), and
  function-to-pointer (_conv.func_) standard conversions are performed on
  the
  second and third operands.

  After those conversions, one of the following shall hold:

  **LI**

  - **4046, 4155**
    and is an rvalue.
    **N[**
    this includes the case where both operands are
    .I throw-expression s.
    **N] e**

  ! **." USA _136/_28 L6899 USA core-756, 734, 682 expr.cond**
  ! Otherwise, if the second and third operand have different types, and
  ! either has (possibly cv-qualified) class type, an attempt is made to
  ! convert each of those operands to the type of the other. The process
  ! for determining whether an operand expression
  ! .CW E1
  ! of type
  ! .CW T1
  ! can be converted to match an operand expression
  ! .CW E2
  ! of type

  **E**
is defined as follows:
If 

is an lvalue:

can be converted to match 

if 

can be implicitly converted (_conv_) to the type

(subject to the constraint that in the conversion the reference must bind directly (_dcl.init.ref_) to 

If 

is an rvalue, or if the conversion above cannot be done:

RS 

if 

and 

have class type, and the underlying class types are the same or inheritance-related:

can be converted to match 

if the class of 

is the same type as, or a base class of, the class of 

and the cv-qualification of 

is the same cv-qualification as, or a greater cv-qualification than, 

the cv-qualification of 

If the conversion is applied, 

can be converted to match 

that still refers to the original source class object (or the appropriate subobject thereof).

that is, no copy is made.

Otherwise:

can be converted to match 

can be implicitly converted to the type that expression would have if 

were converted to an rvalue (or the type it has, if 

is an rvalue).
Using this process, it is determined whether the second operand can be
converted to match the third operand, and whether the third operand can
be
converted to match the second operand. If both can be converted, or one
can be converted but the conversion is ambiguous, the program is ill-
formed.
If neither can be converted, the operands are left unchanged and further
checking is performed as described below. If exactly one conversion
is possible, that conversion is applied to the chosen operand and
the converted operand is used in place of the original operand for the
remainder of this section.

If the second and third
operands are lvalues and have the same type,
the result is of that type and is an lvalue.

Otherwise, the result is an rvalue.

If the second and third operand
do not have the same type, and either has (possibly cv-qualified)
class type, overload resolution is used to determine the conversions
(if any) to be applied to the operands (_over.match.oper_, _over.built_).
If the overload resolution fails, the program is ill-formed.
Otherwise, the conversions thus determined are applied, and the
converted operands are used in place of the original operands for
the remainder of this section.

Lvalue-to-rvalue (_conv.lval_), array-to-pointer (_conv.array_), and
function-to-pointer (_conv.func_) standard conversions are performed on
the
second and third operands.

After those conversions, one of the following shall hold:

- copy-initialization (_dcl.init_) of a class object (_over.match.copy_);
- invocation of a conversion function for initialization of an object of a
  nonclass type from an expression of class type (_over.match.conv_); and
  
invocation of a conversion function for initialization of a temporary
to which a reference (_dcl.init.ref_) will be directly bound (_over.match.ref_).

Each of these contexts defines the set of candidate functions and
the list of arguments in its own unique way.
considered to be a member of the derived class for the purpose of defining
the type of the implicit object parameter.
For static member functions, the implicit object parameter is considered
to match any object (since if the function is selected, the object is discarded).

For static member functions, the implicit object parameter is considered
to match any object (since if the function is selected, the object is discarded).

\[ .\" USA _1333/_6886 core-778 over.match.funcs \]
\[ + .N[ \]
\[ + no actual type is established for the implicit object parameter \]
\[ + of a static member function, and no attempt will be made to determine a \]
\[ + conversion sequence for that parameter. See _over.match.best_. \]
\[ + .N] \]
\[ + P \]
\[ .ix "implicit~conversion~sequences implied~object~parameter \]
During overload resolution, the implied object argument is indistinguishable from other arguments.
The implicit object

***************
*** 1010,1022 ****
a + b;        // operator+(a,b) chosen over int(a) + int(b)
}
.Ce
.E]
.P
! If a built-in candidate is selected by overload resolution, any !
! class operands are first converted to the appropriate type for !
! the operator.
Then the operator is treated as the corresponding built-in operator and interpreted according to clause _expr_.
.P
The second operand of operator
.CW ->
--- 1017,1030 ----
a + b;        // operator+(a,b) chosen over int(a) + int(b)
}
.Ce
.E]
.P
\[ .\" USA _136/_28 L6899 USA core-756/734/682 over.match.oper \]
! If a built-in candidate is selected by overload resolution, the !
! operands are converted to the types of the corresponding parameters !
of the selected operation function.
Then the operator is treated as the corresponding built-in operator and interpreted according to clause _expr_.
.P
The second operand of operator
.CW ->
***************
*** 1043,1053 ****
.CW , ,
the unary operator
.CW & ,
or the operator
.CW -> ,
! and overload resolution is unsuccessful, then the operator is !
assumed to be the built-in operator and interpreted according to !
clause _expr_.
.P
.N[ the look up rules for operators in expressions are different than
--- 1051,1062 ----
.CW , ,
the unary operator
or the operator

! " USA misc L9242 Editorials; public comment 13 over.match.oper
!

and there are no viable functions, then the operator is
assumed to be the built-in operator and interpreted according to
clause _expr_.

P

the look up rules for operators in expressions are different than
***************

*** 1270,1289 ****
a reference to non-\f5\const\fP cannot be bound to an rvalue can affect
the viability of the function (see _over.ics.ref_).
.H3 "Best Viable Function" over.match.best
.P
.iix "overload resolution-and conversion

Let ICS\f2i\fP(\f5\fP)\fP denote the implicit conversion sequence that
converts
the \f2i\fP-th argument in the list to the type of the
.I i -th
parameter
of viable function
.CW F
._over.best.ics_ defines the implicit conversion sequences and
._over.ics.rank_
defines what it means for one implicit conversion sequence to be
a better conversion sequence or worse conversion sequence than
another.

Given these definitions, a viable function
.CW F1
is defined
to be a
.I better

--- 1279,1318 ----
a reference to non-\f5\const\fP cannot be bound to an rvalue can affect
the viability of the function (see _over.ics.ref_).
.H3 "Best Viable Function" over.match.best
.P
.iix "overload resolution-and conversion

Define ICS\f2i\fP(\f5\fP)\fP as follows:

.LI
! if
! .CW F
!
is a static member function, ICS\f21\fP(\f5\fP)\fP is defined such that
! ICS\f21\fP(\f5\fP)\fP is neither better nor worse that ICS\f21\fP(\f5\fP)\fP
! for any function
! .CW G
!

and, symmetrically, ICS\f21\fP(\f5\fP)\fP is neither better nor worse than
! ICS\f21\fP(\f5\fP)\fP

! .Fs
!

If a function is a static member function, this
! definition means that the first argument, the implied object parameter,
! has no effect in the determination of whether the function is better
! or worse than any other function.
! .Fe
!

otherwise,

.LI
! let ICS\f2i\fP(\f5\fP)\fP denote the implicit conversion sequence that
converts
the \f2i\fP-th argument in the list to the type of the
.I i -th
parameter
of viable function

_\texttt{\textunderscore over\textunderscore best.ics}_ defines the implicit conversion sequences and
_\texttt{\textunderscore over\textunderscore ics.rank}_

defines what it means for one implicit conversion sequence to be
a better conversion sequence or worse conversion sequence than
another.

Given these definitions, a viable function

_\texttt{\texttt{F}}_{1}

is defined
to be a
.I better

***************

*** 1306,1316 ****

is a template function specialization, or, if not that,
.II

_\texttt{\texttt{F}}_{1}

and

_\texttt{\texttt{F}}_{2}

are template functions with the same signature, and the function template

for

_\texttt{\texttt{F}}_{1}

is more specialized than the template for

_\texttt{\texttt{F}}_{2}

according to the partial ordering rules described in _\texttt{\texttt{\texttt{\textunderscore temp\textunderscore func\textunderscore order\textunderscore}}}_,

--- 1335,1346 ----

is a template function specialization, or, if not that,
.II

_\texttt{\texttt{F}}_{1}

and

_\texttt{\texttt{F}}_{2}

are template functions, and the function template

for

_\texttt{\texttt{F}}_{1}

is more specialized than the template for

_\texttt{\texttt{F}}_{2}

according to the partial ordering rules described in _\texttt{\texttt{\textunderscore temp\textunderscore func\textunderscore order\textunderscore}}_,

***************

*** 1473,1498 ****

contains no "`conversion``" from

_\texttt{\texttt{\text``const A``}}

to

_\texttt{\texttt{A}}_ .

.E]

! When the parameter has a class type and the argument expression is an
! rvalue of the same type, the implicit conversion sequence is an identity
! conversion.
! When the parameter has a class type and the argument expression is an
! lvalue of the
! same type, the implicit conversion sequence is an lvalue-to-rvalue
! conversion.
! When the parameter has a class type and the argument expression is an
! rvalue of
! a derived class type, the implicit conversion sequence is a
! .I derived-to-base
! .ix "derived-to-base conversion"
Conversion from the derived class to the base class.
! .N[ there is no such standard conversion; this derived-to-base Conversion
exists
only in the description of implicit conversion sequences.
! .N] e
- When the parameter has a class type and the argument expression is an lvalue of a derived class type, the implicit conversion sequence is an lvalue-to-rvalue conversion followed by a derived-to-base Conversion. A derived-to-base Conversion has Conversion rank (_over.ics.scs_).

P

In all contexts, when converting to the implicit object parameter or when converting to the left operand of an assignment operation only standard conversion sequences that create no temporary contains no "conversion" from

CW "const A"

to

CW A.

E]

! Canada _1332/ L3083 Canada 10, core-779, N1084 over.best.ics

! When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion.

! When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class.

N[ there is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences.

.N] e

A derived-to-base Conversion has Conversion rank (_over.ics.scs_).

P

In all contexts, when converting to the implicit object parameter or when converting to the left operand of an assignment operation only standard conversion sequences that create no temporary

***************

*** 1772,1782 ****

.ix "overloading subsequence-rule

.CW S1

is a proper subsequence of

.CW S2

(comparing the conversion sequences in the canonical form defined by _over.ics.scs_; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence)

or, if not that,

.LI

the rank of

.CW S1

--- 1798,1810 ----

.ix "overloading subsequence-rule

.CW S1

is a proper subsequence of

.CW S2

(comparing the conversion sequences in the canonical form defined by _over.ics.scs_, excluding any lvalue Transformation;

! the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence)

or, if not that,

.LI

the rank of

.CW S1

***************

*** 2692,2714 ****

.P

For every cv-qualified or cv-unqualified object type
there exist candidate operator functions of the form
  \[ f2T\fp* \text{ operator}(f2T\fp*, f2ptrdiff_t) \]
  \[ f2T\fp& \text{ operator}[](f2T\fp*, f2ptrdiff_t) \]
  \[ f2T\fp* \text{ operator}-(f2T\fp*, f2ptrdiff_t) \]
  \[ f2T\fp* \text{ operator}+(f2ptdiff_t, f2T\fp*) \]
  \[ f2T\fp& \text{ operator}[](f2ptdiff_t, f2T\fp*) \]

For every \textit{T}, where \textit{T} is a pointer to object type,
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]

For every pointer type

For every cv-qualified or cv-unqualified object type \textit{T},
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]

For every \textit{T}, where \textit{T} is a pointer to object type,
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]

For every pointer type

is either \textit{CW} volatile or empty,
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]

For every \textit{T}, where \textit{T} is a pointer to object type,
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]

For every pointer type

is either \textit{CW} volatile or empty,
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]

For every triple \textit{L}, (\textit{VQ},)

is either \textit{CW} volatile or empty,
there exist candidate operator functions of the form
  \[ \text{ptrdiff_t operator}(-f2T\fp, f2T\fp) \]
For every triple \( I L, (I VQ) \),

\[ \text{For every pair } (I T, (I CVQ)) \text{, where} \]

\( I T \) is a scalar, array, function, or class type, and

\( I CVQ \) is

\( \text{CW const} \),

\( \text{CW volatile} \),

\( \text{CW "const volatile"} \),

or empty, there exist candidate operator functions of the form

\( \text{Cb} \)

\[ \text{bool operator!} (bool); \]

\[ \text{bool operator&&} (bool, bool); \]

\[ \text{bool operator||} (bool, bool); \]

\( \text{Ce} \)

\[ \text{N[as with all these descriptions of candidate functions, this declaration serves} \]

\[ \text{only to describe the built-in operator for purposes of overload} \]

\[ \text{resolution.} \]

\( \text{The operator} \)

\( \text{CW ?} \ (\text{\`\`}) \)

\( \text{cannot be overloaded.} \)

\( \text{N]} \)

\[ \text{e} \]

\[ \text{P} \]

\[ \text{For every pair of promoted arithmetic types} \]

\( I L \)

\[ \text{and} \]

\( I R \),

--- 2876,2886 ----

\[ \text{Cb} \]

\[ \text{bool operator!} (bool); \]

\[ \text{bool operator&&} (bool, bool); \]

\[ \text{bool operator||} (bool, bool); \]

\( \text{Ce} \)

\[ \text{\" USA _136/_28 L6899 USA core-756,734,682 over.built} \]

\[ \text{P} \]

\[ \text{For every pair of promoted arithmetic types} \]

\( I L \)

\[ \text{and} \]

\( I R \),

***************

*** 2886,2916 ****

\[ \text{I LR} \]

\[ \text{is the result of the usual arithmetic conversions between types} \]

\( I L \)

\[ \text{and} \]

\( I R . \)

\[ \text{P} \]
For every type
 .IT ,
 where
 .IT is a pointer or pointer-to-member type, there exist candidate operator
 functions of the form
 .Cb
 \f2T\fP operator?(bool, \f2T\fP, \f2T\fP);
 - .Ce
 - .P
 - For every pair
 - .IT , ( .ICVQ ),
 - where
 - .IT
 - is a class type and
 - .ICVQ
 - is
 - .CW const ,
 - .CW volatile ,
 - .CW "const volatile" ,
 - or empty, there exist candidate operator functions of the form
 - .Cb
 - \f2CVQ T\fP operator?(bool, \f2CVQ T\fP, \f2CVQ T\fP);
 .Ce
 --- 2892,2913 ----
 .ILR
 is the result of the usual arithmetic conversions between types
 .IL
 and
 .IR .
+ .N[ as with all these descriptions of candidate functions, this declaration
+ only to describe the built-in operator for purposes of overload
+ resolution.
+ The operator
+ .CW ? \(`` \(``
+ cannot be overloaded.
+ .N] e
 .P
 For every type
 .IT ,
 where
 .IT is a pointer or pointer-to-member type, there exist candidate operator
 functions of the form
 .Cb
 \f2T\fP operator?(bool, \f2T\fP, \f2T\fP);
 .Ce
diff -c5 troff.orig/stmt troff/stmt
 *** troff.orig/stmt Wed Jul 16 04:05:55 1997
 --- troff/stmt Thu Jul 17 07:04:16 1997
 ***************
 *** 642,657 ***
 .CW return
 statement.
 .P
 A return statement without an expression can be used only in functions
 that
do not return a value,
! that is, a function with the return value type
 .CW void ,
a constructor (_class.ctor_), or a destructor (_class.dtor_).
A return statement with an expression can be used only in functions returning
a value; the value of the expression is returned to the caller.

A return statement without an expression can be used only in functions
that do not return a value, that is, a function with the return type
`void`, a constructor (_class.ctor_), or a destructor (_class.dtor_).

A return statement with an expression of non-void type can be used only in functions returning
a value; the value of the expression is returned to the caller.

Flowing off the end of a function is equivalent to a return with no value;
this results in undefined behavior in a value-returning function.

A return statement with an expression of type `void` can be used
only in functions with a return type of `cv void;` the expression is
evaluated just before the function returns to its caller.

struct T2 { T2(int){ } };  
int a, (*(*b)(T2))(int), c, d;

void f() {
    // disambiguation requires this to be parsed
    // as a declaration
    T1(a) = 3, T2(4),  // T2 will be declared as
     (*(*b)(T2(c)))(int(d));  // a variable of type T1
    // but this will not allow
struct T2 { T2(int){ } };  
int a, (*(b)(T2))(int), c, d;  

void f() {  
    // disambiguation requires this to be parsed  
    // as a declaration  
    T1(a) = 3,  
    T2(4),  
    (*(b)(T2(c)))(int(d));  
    // T2 will be declared as  
    // a variable of type T1  
    // but this will not allow