Doc Number:	X3J16/93-0168R1 WG21/N0375R1
Date:	November 23, 1993
Project:	Programming Language C++
Ref Docs:	93-0062 / N0269
SMLI Doc Number:	SMLI93-0397

Unifying Built-In Operators and User-Defined Functions
Revision 1

Samuel C. Kendall
Sun Microsystems Laboratories, Inc.
Sam.Kendall@East.Sun.COM

1. Introduction

This is a proposal to describe the built-in operators using overloadings and conversions, much as we describe overloaded functions.

The built-in operators are currently described (§5) much as in ISO C. Unfortunately there is no "interface" between §5 and the extensions to the type system, and to conversions, that C++ makes over C. Tom Plum's e-mail message c++std-core-2587 (core issue 178) proposes to address this situation by adding some "master control" paragraphs to the beginning of §5. However, as I understand it, Tom's approach does not address conflicts between user-defined and built-in operators as in examples 2 and 3 below.

Here is a simple example of a use of a built-in operator that "interfaces" with a user-defined conversion:

Example 1

```
struct A {
      operator int();         // An A can become an int.
};
void f(A a) {
      a + a;                  // a.operator int() + a.operator int()
}
```

Example 1 should obviously use a built-in meaning for `operator+`. This is implied by §5/8 of the WP (working paper), but in a very imprecise way.

The next two examples are more problematic.

Example 2

```
struct B {};
struct A {
      operator int();         // An A can become an int
      operator B();           // or a B.
};
B operator+(B, B);
void f(A a) {
      a + a;                  // Should be ambiguous.
}
```

Example 3

```
struct A {
      operator int();          // An A can become an int
      operator long();         // or a long.
};
void f(A a) {
      a + a;                   // Ambiguous, but by what rule?
}
```

How are these uses of `operator+` interpreted? The WP provides no clear guide.

I propose instead a "Unified Model" for built-in operators and user-defined operators and functions (including template functions, although I don't talk much about them in this document). Sections 2, 3, 4 and 5 present the unified model. Section 6 discusses some subtle issues and presents some examples. Section 7 briefly discusses a related approach taken by Microsoft C++.

The unified model would be a big change to the WP. I'd prefer to avoid this big change. If anyone can think of a simpler way to define the behavior of the examples above, and the examples in section 6 below, please let me know.

This is *not* a proposal for language extensions. I've tried to avoid any changes to the language. I've had three goals in specifying things that are not specified in the WP:

- Try for simplicity.
- Make a doubtful expression (such as those in examples 2 and 3) ambiguous so that the compiler can catch more of the user's mistakes.
- Follow existing implementations, particularly Cfront 3.0, but also Microsoft Visual C++ 1.0 and Borland's Turbo C++ for Windows 3.1.

A point of notation: throughout this paper "§4.1" means section 4.1 of the WP, but "section 4.1" means section 4.1 of this document.

This paper has been influenced by countless core reflector messages, and particularly by discussions with Jan Gray, Jonathan Shopiro, Frank Buschmann, Eric Krohn, and Jim Welsch. Ron Guilmette and others have argued that a scheme like this is necessary. Of course, all the mistakes and omissions are mine.

2. The Unified Model

This section and the next three sections present the unified model. Section 3 revises the standard conversions, placing them in novel categories and in most cases expressing them more formally than §4 and §13.2. Section 4 shows a series of *argument contexts* that govern which conversions are allowed in various situations (argument to built-in operator, argument to user-defined function, argument to template function, etc.) and how conversion sequences are compared. Section 5 restates the built-in operators (§5) as *built-in templates.*

The major simplification to the language description is that the matching of operator and operands is defined by just one (admittedly complicated) mechanism, argument matching.

Some things are still missing. This paper is intended to compliment a redefinition of overloading resolution being worked on by Jon Shopiro and others. I hope a future revision of this paper can build on that foundation.

I have also left out definitions (in section 5) of some difficult built-in operators such as `.` and casts. In most cases the operators I've left out cannot be overloaded, so their definitions are not central to deciding whether the unified model can work at all. Two important and difficult exceptions are `operator->` and `operator()`; these still need definitions in this framework.

3. Revised Standard Conversions

Table 1 restates the standard conversions (§4 and §13.2/8) in novel categories.

Table 1: Revised Standard Conversions

From	To	Constraints
Dereference Conversions		
`T&`	`T`	
`T[`*dimension*$_{opt}$`]`	`T*`	Footnote[a].
`T(&)(`*param-types*`)`	`T(*)(`*param-types*`)`	Footnote[b].
Enreference Conversions		
`T`	`T&`	
Qualification Conversions		
`T&`	`const T&`	Actually, these are the more complicated `const`/`volatile` pointer and reference conversions as in §4.6.
`T&`	`volatile T&`	
`T*`	`const T*`	
`T*`	`volatile T*`	
Integral Promotions		

`char`
`signed char`
`unsigned char`
`short`
`unsigned short`
	One of:
	`int`
	`unsigned int`
		Implementation-defined

`wchar_t`
`enum E` *(any enumeration type)*
	One of:
	`int`
	`unsigned int`
	`long`
	`unsigned long`
		Implementation-defined

Usual Integral Conversions		
`int`	`unsigned int` `long`	
`unsigned int`	*One of:* `long` *no conversion*	Conversion iff a `long` can represent all the values of an `unsigned int`

`int`
`unsigned int`
`long`
	`unsigned long`	

Table 1: Revised Standard Conversions (Continued)

From	To	Constraints
Usual Floating Conversions		
`float`	`double`	
`double`	`long double`	Footnote[c].

`int`
`long`
`unsigned int`
`unsigned long`
	`float`
	`double`
	`long double`	

Other Arithmetic Conversions		

`int`
`unsigned int`
`long`
`unsigned long`
`double`
`long double`
	Any arithmetic type	A conversion is an Other Arithmetic Conversion only if it is not a Usual Integral or Usual Floating Conversion; so exclude, eg, `int --> long`. Also, there are no conversions from a type to itself; so exclude, eg, `int --> int`.

Miscellaneous Conversions		

Constant 0 of type:
`int`
`unsigned int`
`long`
`unsigned long`
	`T*`
	`T C::*`	

`T(*)`(*param-types*)	*One of:* `void*` *no conversion*	Conversion iff a `void*` can represent all the values of a function pointer
cv `D&`	*cv* `B&`	`B` is an unambiguous, accessible base class of `D`
cv `D*`	*cv* `B*`	`B` is an unambiguous, accessible base class of `D`
cv `T*`	*cv* `void*`	
`T B::*`	`T D::*`	`B` is an unambiguous, accessible base class of `D`
Ellipsis Conversions		
`T`	`...`	There are restrictions on `T`, but these are not listed here.

a. It is perhaps unfortunate that an rvalue array implicitly converts to a pointer, but that's the way the language is now. For example:

```
// ary-rv1.C, fnc-rv1.C
struct A { int a[10]; };
A f();
int* ip = f().a;              // ok, int [10] --> int*
int (*ap)[10] = &f().a;       // bad, address of rvalue
```

Cfront 3.0 and Microsoft C++ agree. (Turbo C++ crashes.)
b. Only lvalue functions (that is, only non-member functions) can convert to pointer. This I express by putting a "reference to function" type on the left-hand side of the conversion.
c. Is there a conversion `float --> long double`? §4.3 is not clear. I don't think there should be such a conversion; instead, we can compose the `float --> double` and `double --> long double` conversions.
The question can be decided with this fragment:

```
// flt-ldb1.C
void f(double);
void f(long double);
f(1.0f);       // f(double(1.0f)) or ambiguous?
```

If the resolution is `f(double(1.0f))` then there is no `float --> long double` conversion. If the resolution is ambiguous then there is such a conversion.

3.1 Notation

When we write a conversion rule outside of table 1 we use "`-->`" to separate the source and destination type; for example, `int --> char` is one of the Other Arithmetic Conversions.

In these rules *`cv`* stands for any combination of cv-qualifiers, or none at all. If it appears on both sides of a rule it is identical on both sides. Thus a rule such as

```
cv D& --> cv B&
```

actually stands for four rules, one for each possible "value" of *`cv`*.

A rule with a list on the left and/or right actually stands for the cross-product of the lists. Thus many rules can be implied by one box in table 1.

3.2 Categories of Conversions

The new categories facilitate the argument contexts defined in section 4.

Here is a list of terms that designate groups of conversions:

- Standard conversions
  - Trivial conversions
    - Dereference conversions*
    - Enreference conversions*
    - Qualification conversions*
  - Integral promotions
  - Usual arithmetic conversions
    - Usual integral conversions*
    - Usual floating conversions*
  - Other arithmetic conversions*
  - Miscellaneous conversions*
  - Ellipsis conversions*
- User-defined conversions

* designates new categories.

One term is indented below another if it refers to a subset of the other term's conversions. For example, the usual integral conversions are a subset of the usual arithmetic conversions, which are in turn a subset of the standard conversions.

3.3 Reference-Normal Type

There is some confusion about what a conversion involving reference types means. For example, the conversion I write as `T& --> const T&` is written sometimes without the “reference” type modifiers (§13.2/9), sometimes with them (§13.2/12). Also, we are on the verge of deciding that expressions of reference type “decay” immediately to non-reference type. If there are (effectively) no expressions of reference type, how can we have implicit conversions involving reference types?

We can use the `T --> T&` and `T& --> T` conversions freely and ignore the problem; but this will not work in my approach, because I allow certain built-in operators to overload on both `T&` and `T`.

We can make sense of this mess by performing conversioons and argument matching on the *reference-normal type* of an expression. The reference-normal type of an expression of type `T` is simply `T&` if the expression is an lvalue, `T` otherwise.

The alternative is some cumbersome notation to indicate when a conversion takes and/or yields an lvalue, eg:

> `{ T,` *lvalue* `} --> { const T,` *lvalue* `}`
>
> `{ T (`*param-types*`),` *lvalue* `} --> T (*)(`*param-types*`)`

or to indicate the same thing in English, which is tricky to get right.

In the rest of this paper I assume the reference-normal type is used in argument matching. It is essential in allowing built-in operators such as `,` and `?:` to make fine distinctions between lvalues and rvalues.

3.4 Where is an Lvalue’s Value Extracted?

A side-benefit of the unified model is a solution to problem (G) posed in my 93-0134 / N0341. That problem is that the WP does not define where an lvalue’s value is extracted, and thus where it is ill-formed to use an expression of incomplete type. The solution in the unified model is that a lvalue’s value is extracted exactly when the `T& --> T` conversion is invoked: it is ill-formed to invoke that conversion if `T` is an incomplete type.

4. Argument Contexts and Ordering Rules

4.1 Argument Contexts

This section expands the conversion sequence ordering rules in §13.2/11-12 to handle not only user-defined functions, but also function templates (§14.4) and the built-in operators (§5). A function template or a built-in template (see section 5) is considered to be a set of overloadings.

There are a handful of *argument contexts*; these govern the conversions that are allowed and how they are ordered. The conversion contexts are:

- the *function context* (table 2), for arguments of functions and user-defined operators;
- the *template function context* (table 3), for arguments of template functions and (user-defined) template operators;
- the *built-in operator context* (table 4), for arguments of built-in operators; and
- the *restricted context* (table 5), for the left-hand argument of `= . -> .* ->*`.

All of these argument contexts except the built-in operator context are implied in the WP.

Table 2: Function Context

Match with
Exact (no conversions) Dereference conversions Enreference conversions
Qualification conversions
Integral Promotions
Usual integral conversions Usual floating conversions Other arithmetic conversions Miscellaneous conversions
User-defined conversions
Ellipsis conversions

Table 3: Template Function Context

Match with
Exact (no conversions) Dereference conversions

The dereference conversion is necessary in the template function context because of our use of reference-normal type; otherwise a variable of type `T` would not be a legal argument to a template function expecting a `T`.

The extensions group will probably add more conversions to the template function context.

Table 4: Built-in Operator Context

Match with
Exact (no conversions)
Dereference conversions
Enreference conversions
Qualification conversions
Integral Promotions
Usual integral conversions
Usual floating conversions
Miscellaneous conversions
User-defined conversions

Table 5: Restricted Context

Match with
Exact (no conversions) Dereference conversions Enreference conversions
Qualification conversions

4.2 Ordering Rules for Conversions[1]

Comparing two conversions in the same argument context is easy. For each context, the respective table defines which conversions are allowed, in decreasing order of *goodness*. If one box is above another, the conversions in the upper box are better than the conversions in the lower box. All the conversions in a box are pairwise unordered, with one exception. The exception to this unordered-in-a-box rule is for certain miscellaneous conversions such as `cv D* --> cv B*`; see §13.2/12 item [3]. This procedure is similar to that in §13.2/12.

However, in a call to a function or operator, one argument can appear in multiple contexts. In the most extreme case one argument can appear simultaneously in all four contexts! (I call these "four corners" examples; see section 6.)

How then do we compare two conversions using two *different* argument contexts? The problem is similar to analyzing one crime that spans two countries: which has jurisdiction? The answer is to try to make the comparison in both contexts. If one context cannot make the comparison (because one of the conversions is not in that context), then the other context has jurisdiction. Otherwise, if either context says "ambiguous", the result is "ambiguous". Otherwise both contexts agree on the result. (If they don't agree, there is an "internal error" in these rules.)

4.3 Ordering Rules for Conversion Sequences

A comparison of two conversion sequences yields "worse than", "better than", or "ambiguous". We compare conversion sequences by comparing the worst conversion in each sequence. There are additional comparison rules. In particular, something like the rules in §13.2/7-8 is necessary.

4.4 Ordering Rules for Overloadings

There are some rules for comparing overloadings that supplement the rules for comparing conversion sequences.

First, if a template function and an ordinary function are "ambiguous" by comparison of conversion sequences, the ordinary function is better. This is necessary because the ordinary function may be a specialization of the template function. This rule is implied by §14.4/3.

Second, for `operator,` and unary `operator&` only, if there is any user-defined overloading that is a legal match, it is better than any built-in match. This is necessary because there are exact built-in matches for all types; if this special rule were not present, only exact matches would invoke user-defined overloadings; thus, because calling an inherited member function requires a conversion, those operators could not be inherited. Cfront, Microsoft, and Borland seem to implement this rule.

4.5 The Built-in Operator Context

The key feature of this whole paper is the many preference levels in the built-in operator context. They allow the

1. The next three sections depend on Jon Shopiro's new overloading resolution scheme. In fact an understanding of that (unpublished) scheme is helpful in understanding these sections.

built-in operators to be overloaded on whether an argument is an lvalue, something the function context cannot do. They also bring the usual arithmetic conversions (a C concept) into the unified model. More specifically:

- Exact match is preferred to dereference conversions so that, given

  ```
  int i;
  ```

  The expression `0,i` is unambiguously an `int&` (an lvalue), while `0,5` is unambiguously an `int` (an rvalue).

- Dereference conversions are preferred to enreference conversions so that

  ```
  int e, i;
  e ? i : 5;        // e ? (int)i : 5
  ```

  won't be ambiguous, and will yield `int` rather than `int&` (an rvalue rather than an lvalue).

- The usual integral conversions are preferred to the usual floating conversions so that

  ```
  10L + 20U
  ```

  is unambiguous, and either a `long` or an `unsigned long` (the choice depends on the sizes of the types involved; see §4.5).

- The usual integral conversions are preferred to the miscellaneous conversions so that

  ```
  0 + 2L
  ```

  will be `(long)0 + 2L`; if the miscellaneous conversions were preferred to the usual integral conversions, then `int --> T*` would be preferred to `int --> long`; the arguments would match all instantiations of the built-in template `operator+(T*, I)` (see section 5); and the expression would be ambiguous.

- I don't think the miscellaneous conversions are ever compared to the usual floating conversions, so it doesn't matter which box is above the other.

- The other arithmetic conversions are not in the built-in operator context. There seem to be no cases where they could be unambiguously invoked.

5. The Built-in Operators

We express the built-in operators using *built-in templates*, written using the pseudo-keyword `builtin`.[1] A built-in template differs from a real operator template as follows:

- Remove the restriction that one parameter must be a class.
- Remove the restriction (if present for that operator) that the operator must be a member function.
- Definitions of built-in template type parameters are constrained by pseudo-keywords from the following table:

Table 6: Pseudo-keywords for builtin type parameters

Pseudo-keyword	Types indicated
`any`	Any non-reference type.
`class`	Any class type.
`nonclass`	Any non-class, non-reference type other than `void`.

1. These definitions are similar to Ada's package STANDARD.

Table 6: Pseudo-keywords for builtin type parameters (Continued)

Pseudo-keyword	Types indicated
`integral`	Any integral type.

`bigintegral`	Any of the following types:
	`int`
	`unsigned int`
	`long`
	`unsigned long`

`arithmetic`	Any arithmetic type.

`bigarithmetic`	Any of the following types:
	`int`
	`unsigned int`
	`long`
	`unsigned long`
	`float`
	`double`
	`long double`

`scalar`	Any arithmetic or pointer or pointer-to-member type.
`object`	Any non-cv-qualified, non-reference, non-void, non-function type.
`function`	Any function type.

- Remove the restriction that a parameter of reference type cannot accept a bit-field argument.
- The argument context (see section 5 above) is by default the built-in operator context rather than the template function context.
- The left-hand operand of `operator=` and a few others are in a restricted argument context. We express this by using the keyword

  ```
  restricted
  ```

  in declaring that argument. (`restricted` has nothing to do with the NCEG *restrict* keyword.)
- Evaluation order is unusual for some operators. For example, order of evaluation (and whether evaluation occurs at all) is unusual for arguments of `operator,` and `operator&&`.
- Certain operators such as `?:` have built-in templates even though they cannot be user-defined operators.

Here we go:

```
builtin <any T, bigintegral I> T* operator[](T*, I);
builtin <any T, bigintegral I> T* operator[](I, T*);

// function calling is a special case

// simple-type-specifier ( args ) is a special case

// dynamic_cast is a special case.

// . and -> are special cases.
```

```
// Prefix ++.
// For the first overloading, the result is a bit-field
// if the argument is.
builtin <integral I> I& operator++(restricted I&);
builtin <any T> T*& operator++(restricted T*&);

// Postfix ++.
builtin <integral I> I operator++(restricted I&, int);
builtin <any T> T* operator++(restricted T*&, int);

// The two forms of -- are just like the two forms of ++.
// Not written out here.

builtin <any T> T& operator*(T*);

// Unary & is a very special case.

builtin <any T> T* operator+(T*);
builtin <bigarithmetic A> A operator+(A);

builtin <bigarithmetic A> A operator-(A);

builtin <scalar B> int operator!(B);

builtin <bigintegral I> I operator~(I);

// sizeof.
// The result is Constant. Argument is not evaluated.
// We need the two overloadings because you can take the size of an
// lvalue or an rvalue.
builtin <any T> size_t sizeof(T&);
builtin <any T> size_t sizeof(T);

// sizeof (type-id) is a special case; it yields a Constant
// of type size_t.

// operator new and operator new[] is a special case.

builtin <any T> void operator delete(T*&); // May zero out arg
builtin <any T> void operator delete(T*);

builtin <any T> void operator delete[](T*&); // May zero out arg
builtin <any T> void operator delete[](T*);

// casts are a special case
```

// In the following definitions of `.*` and `->*`, `cv1` and `cv2` are either nothing, `const`, `volatile`,
// or `const volatile`. `cv12` is the union of `cv1` and `cv2` (combining the cv-qualifiers from both).

```
builtin <class C, object T> cv12 T& operator.*(cv1 C&, cv2 T C::*);
builtin <class C, object T> cv12 T operator.*(cv1 C, cv2 T C::*);
builtin <class C, function T> cv2 T operator.*(cv1 C&, cv2 T C::*);
builtin <class C, function T> cv2 T operator.*(cv1 C, cv2 T C::*);

builtin <class C, object T> cv12 T& operator->*(cv1 C*, cv2 T C::*);
builtin <class C, function T> cv2 T operator->*(cv1 C*, cv2 T C::*);

builtin <bigarithmetic A> A operator*(A, A);

builtin <bigarithmetic A> A operator/(A, A);

builtin <bigintegral I> I operator%(I, I);

builtin <bigarithmetic A> A operator+(A, A);
builtin <any T, bigintegral I> T* operator+(T*, I);
builtin <any T, bigintegral I> T* operator+(I, T*);

builtin <bigarithmetic A> A operator-(A, A);
builtin <any T, bigintegral I> T* operator-(T*, I);
builtin <any T> ptrdiff_t operator-(restricted T*, restricted T*);
```

```
builtin <bigintegral I1, bigintegral I2> I1 operator<<(I1, I2);

builtin <bigintegral I1, bigintegral I2> I1 operator>>(I1, I2);

builtin <scalar S> int operator<(S, S);

// Likewise for > <= >=.

builtin <nonclass A> int operator==(A, A);
builtin <any T> int operator==(T*, T*);
builtin <any T, class C> int operator==(T C::*, T C::*);

// Likewise for !=.

builtin <bigintegral I> I operator&(I, I);

builtin <bigintegral I> I operator|(I, I);

builtin <bigintegral I> I operator^(I, I);

builtin <scalar B1, scalar B2> int operator&&(B1, B2);
// 2nd arg may not be evaluated.

// likewise for ||

builtin <scalar B, any T> T& operator ?:(B, T&, T&);
builtin <scalar B, any T> T operator ?:(B, T, T);
builtin <scalar B, any T> T operator ?:(B, void, void);
// Only one of the 2nd and 3rd args is evaluated.
// We need both overloadings because the result is an lvalue iff both
// arguments are.
// For the T& overloading, the result is a bit-field if both arguments
// are.

builtin <nonclass T> T& operator=(restricted T&, T);
// The first arg is in the restricted context.
// class types have no builtin assignment operator. (Many have a
// default assignment operator, though; see 12.8.)

builtin <arithmetic A1, bigarithmetic A2>
A1& operator*=(restricted A1&, A2);

// The other op= operators are similar to *= and
// are not written out here.

builtin <any T> void operator throw(T);
builtin <any T> void operator throw(T&);

builtin <any T1, any T2> T2& operator,(T1, T2&);
builtin <any T1, any T2> T2 operator,(T1, T2);
// Need both overloadings because the result is an lvalue iff the
// second arg is.
// Unlike every other operator except unary &, any user-defined match
// is preferred to any built-in match.
```

The following subsections note additional rules. These may seem like too many special cases; but please remember that we are specifying a whole new area (user-defined conversions with built-in operators) that the WP does not deal with; in addition, §5 (which this document restates a good deal of) contains a host of special cases itself.

5.1 Attributes of (Sub)expressions

Please see N0365 = 93-0158 for more information on expression attributes. Attributes of the result have the default value unless otherwise indicated. For example, a comment next to the definition of operator++ indicates that the result is a bit-field (has the bit-field attribute) if the argument is a bit-field.

5.2 Conversions for the Relational, Equality, and Conditional Operators[1]

The last expression below is allowed by my overloading rules. It must be banned by a special-case rule.

```
// relop1.C
int* ip;
double* dp;
void* vp;

void f() {
      vp < dp;     // ok; vp < (void*)dp
      ip < vp;     // ok; (void*)ip < vp
      ip < dp;     // ill-formed
}
```

It's okay for one argument to be converted to `void*`; why isn't it okay for *both* arguments? Because if we allow it, there is no type-checking at all for pointer arguments to the relational and equality operators; almost any pointer type can be converted to `void*`.

Similar problems arise for the second and third arguments of `?:`. ISO C and §5.16 don't allow expressions like

```
condition ? ip : dp
```

I suggest a rule something like this for the operators `< > <= >= == !=`, and for the second and third arguments of the conditional operator:

> If the conversion sequences for both arguments includes the `cv T* --> cv void*` conversion, the resulting operator invocation is not well-formed.

For `?:`, some restriction on user-defined conversions may also be desired. Cfront 3.0 disallows all user-defined conversions; Microsoft C++ and Borland C++ allow a user-defined conversion for one argument, but not both.

5.3 Conversions and Pointer Subtraction

The argument contexts for pointer subtraction are `restricted` in order to disallow the following:

```
struct A { ... };
struct B { ... };
struct D : A, B { ... };
B* bp;
D* dp;

0 - dp;      // Bad. (D*)0 - dp would be incorrect.
bp - dp;     // Bad. bp - (B*)dp would be incorrect, since the array
             // which bp and dp point into is presumably an array
             // of D's, not of B's.
```

However, the `restricted` context also disallows user-defined conversions. This may be too drastic a restriction on pointer subtraction. If we want to allow user-defined conversions, we need to define a special context just for pointer subtraction.

6. Examples

Example 4

```
struct A {
      operator void*();
};
```

1. Thanks to Jan Gray for this example.

```
void f(A a) {
      return a == 0;     // a.operator void*() == (void*)0
}
```

6.1 The `?:` Operator and `D&` `-->` `B&` Conversions

Example 5

```
// quecol1.C
#include <stdio.h>

struct B {
      B() {}
      virtual char* type() { return "B"; }
};
struct D : B {
      D() {}
      virtual char* type() { return "D"; }
};

main() {
      B b;
      D d;
      B& br = b;
      D& dr = d;
      printf("Lvalues: %s %s\n",
            (1 ? b : d).type(),
            (0 ? b : d).type());
      printf("References: %s %s\n",
            (1 ? br : dr).type(),
            (0 ? br : dr).type());
      printf("Rvalues: %s %s\n",
            (1 ? B() : D()).type(),
            (0 ? B() : D()).type());
      return 0;
}
```

For this one the results of different compilers are worth looking at.

Cfront 3.0 prints:

```
Lvalues: B D
References: B D
Rvalues: B D
```

Microsoft C++ prints:

```
Lvalues: B B
References: B B
Rvalues: B B
```

Borland C++ prints (folding in compile-time errors):

`Lvalues:` *Compile-time error, two operands must evaluate to the same type*
`References: B D`
`Rvalues:` *Compile-time error, two operands must evaluate to the same type*

(I think Borland is the only one that implements §5.16 as written.)

The unified model produces the following result:

`Lvalues: B D`
`References: B D`
`Rvalues:` *Ill-formed*

The rvalue case is ill-formed because the second and third operands undergo conversions as follows:

```
B --> B&
D --> D& --> B&
```

They satisfy the `operator?:(int, B&, B&)` overloading. However, rvalues have been used to initialize non-const references, so the expression is ill-formed.

6.2 Incomplete Types

Some built-in overloadings are errors if actually matched. For example:

```
struct A;
ptrdiff_t f(A* p) {
      return p - p;// Error.
}
```

For binary − (and perhaps other operators), it is an error for the argument to be of type pointer-to-incomplete-type or pointer-to-function-type

But there is a subtle question remaining. Does an argument of that type match the builtin template or not? Here is how we check:

Example 6

```
// incompl2.C
struct Incomplete;
struct Complete {};
struct A {
      operator Incomplete*();
      operator Complete*();
};
void f(A a) {
      a - a;      // Ambiguous or
                  // a.operator Complete*() - a.operator Complete*()
}
```

Cfront, Microsoft C++, and Borland C++ say this is ambiguous; so I conclude that (quite properly) the builtin template for unary `operator*` accepts pointer-to-incomplete-type in type matching. (Confusingly, this example is not allowed at all unless we allow user-defined conversions for pointer subtraction, as suggested in section 5.3.)

6.3 A Simple "Four Corners" Example

Example 7

```
// 4corner2.C
// Template function, member function, regular function,
// and built-in operator all compete for the same operands.

struct A { void operator-(A); };
struct B {};
void operator-(B, B);
template <class T> void operator-(T, T);
struct C {};

void f(A a, B b, C c) {
      a - a;      // A::operator-(A)
      b - b;      // operator-(B, B)
      c - c;      // template instantiation operator-(C, C)
      5 - 5;      // built-in operator-(int, int)
}
```

There are no ambiguities in this example. But all examples of operator ambiguities are essentially perturbations of this example (introducing user-defined conversions, introducing inheritance relationships, using a different operator, using pointer types instead of class types, etc.). I leave these an exercises for future papers.

7. A Related Approach: Microsoft C++

As I understand it, Microsoft C++ implements the built-in operators using overloadings, but they do not have a separate argument context for built-in operators. Instead, they use a large number of overloadings, for example defining overloadings for binary + on all 144 combinations of arithmetic types. Although this approach works, and it has the advantage of relying a bit less on intricate conversion behavior, I think it is not suitable for a standard; there are too many overloadings. They also convey less information. For example, the expression

```
s + 5       // s is a short
```

(assuming 16-bit `shorts` and 32-bit `ints`) results in `s` being sign-extended to 32 bits before the addition takes place. My scheme conveys this fact by invoking a conversion from `short` to `int` and matching the `int+int` overloading. But Microsoft C++ has an overloading `short+int`; so the sign-extending behavior must be expressed as part of the definition of that overloading.