Doc Number: X3J16/92-0053
WG21/N0130
Date: June 2, 1992
Project: Programming Language C++
Ref Doc: X3J16/92-0023
Reply to: Samuel C. Kendall
CenterLine Software, Inc.
10 Fawcett Street
Cambridge, MA 02138
kendall@centerline.com

Merging Lvalue and Reference; and Canonicalizing Types

Samuel C. Kendall

1. Introduction

This is a proposal to merge the now distinct concepts of *lvalue* and *expression of reference type*. My purpose is to clarify and simplify the working paper (WP) in an area that is now somewhat ambiguous. This is *not* a proposal for language extensions, though it does result in some minor changes to the language.

There is also an almost unrelated item, *Canonicalizing Types*, following the larger proposal.

In merging lvalue and expression of reference type, the language description is simplified as follows:

- Lvalues as distinct entities are eliminated. “Lvalue” becomes a synonym for “expression of reference type”. “Rvalue”, if desired, becomes a synonym for “expression of non-reference type”.

The changes necessary to achieve these simplifications are as follows:

- Bit-field of size N, an attribute which is currently outside the type system, is brought into it. However, we restrict the use of bit-field types so as not to extend the language (e.g., no pointers to bit-fields!).

The actual language changes necessitated by these changes are quite minor:

- We restrict the conditional operator in a minor case.

I have not proposed exact changes in wording to the WP to implement this proposal.

This proposal is part of a larger proposal, not yet complete, to describe built-in operators in terms of conversions and overloading. Thus the proposal you are reading may not provide sufficient benefits of simplicity to justify itself (although I think that it does). If that turns out to be the case, please treat it as a meditation on the nature of lvalues and reference types.

I am using Tom Plum’s informal notation for section and paragraph numbers; thus 522 is section

5.2.2, and 12_8p2 is section 12.8 paragraph 2.

2. Lvalues Are References

In the current WP, the following two entities are very similar:

- lvalue of type `T`
- expression of type `T&`

This section is a proposal to merge them: to define “lvalue” as shorthand for “expression of reference type”. If we want “rvalue” in the WP, define it to be “expression of non-reference type”. We should not say “lvalue of type `T`”; instead, say “expression of type `T&`”.

Let us first look at the proposal in detail; then at some seemingly odd consequences of eliminating lvalue; then at some parts of the WP that are clarified by this change.

2.1 The Proposal in Detail

To review lvalues and references, read 37, 5p7, 822, and 843. For more information, see 522p9, 532p2, 54p14, 5_17p1, and 5_17p7. See also the rules governing the initialization of references to base classes; they are stated once in terms of references (47) and once in terms of lvalues (843p4).

Reading these, we find that a reference “is-a” lvalue; that is, anywhere an lvalue of type `T` appears, any reference to `T` can be substituted.

But is the reverse true, that an lvalue “is-a” reference? Can any lvalue of type `T` be substituted anywhere a reference to `T` can appear? No; there are two exceptions. First, a bit-field lvalue cannot be converted to a reference; we solve this problem in section <> below. And second, references behave differently than lvalues as the second and third operands of the conditional operator.

This second exception we deal with by extending C++ very slightly. Here are the relevant sentences from the WP:

> ...[I]f both the second and third expressions are references, reference conversions are performed to bring them to common type.... The result is an lvalue if the second and the third operands are of the same type and both are lvalues. (5_16p3)

This implies the following:

```
// B is an accessible base class of D
B b;
D d;
cond ? (B&)b : (D&)d;     // references: legal, yields a B&
cond ? b : d;             // lvalues: illegal
```

Is this difference important? Considering that

```
b = d
```

is legal, it makes sense for

```
cond ? b : d
```

to be legal also (and to yield an lvalue of type `B`). I propose that it be made so.

2.2 Consequences

The consequences of eliminating lvalues in favor of references are described in three parts: first, in the meanings of identifiers; second, in the conversion rules; and third, in the meanings of operators that take or yield lvalues.

2.2.1 Meanings of identifiers in expressions

Currently the result of any of these (51; these are some but not all of the primary-expressions):

primary-expression:
    `::` *identifier*
    `::` *operator-function-id*
    `::` *qualified-id*
    id-expression

is described basically as having the type it was declared with, and it "is an lvalue if the identifier is" (51p4). Most identifiers are lvalues, obviously, but I was unable to find where in the WP this is stated. (Can someone help me out here?) The only non-lvalue identifiers (that are legal expressions) are enumerators (72p1) and `this` (51p3, 931p1).

I propose instead that these primary expressions (of the syntaxes listed above) have the following types:

Table 1: Types of identifiers in expressions

Primary Expression	Type
enumerator	"`E`", where `E` is the enumeration type to which the enumerator belongs
`this`	"pointer to possibly qualified `C`", where `C` is the class type of the member function definition in which `this` appears, and the qualifiers (if any) are the same as those of the member function
declared as type "reference to `T`"	"reference to `T`"; in other words, an identifier declared as a reference type yields that reference type
all other	"reference to T", where T is the type the identifier was declared as; in other words, an identifier declared as a non-reference type "T" yields a value of type "T&"

This last rule may seem a bit strange. For example:

```
int i;
int& ir;
... i ...;      // yields an 'int&'
... ir ...;     // also yields an 'int&'
```

There is a precedent for this rule in Algol-68.

We have avoided dealing with member names in this section. See section <<>> for a treatment of member names.

2.2.2 Conversion Rules

The conversion rules are simplified in that all rules specifying conversions between lvalues and references go away (5p7, 522p9, and 54p14).

The trivial conversions (13_2p8) become Table 2:.

Table 2: Revised Trivial Conversions

	From	To	Constraints
a	*qualifiers*$_{opt}$ `T&`	`T`	`T` cannot be a function type, array type, or qualified type
b	`T(&)[`*dimension*$_{opt}$`]`	`T*`	
c	`T(&)(`*args*`)`	`T(*)(`*args*`)`	
d	`T`	`const T&`	
e	`T&`	`const T&`	
f	`T&`	`volatile T&`	
g	`T*`	`const T*`	
h	`T*`	`volatile T*`	

Rule a has had constraints and qualifier-stripping added, to avoid the generation of anomalous expression types: bare (rvalue) function, array, const, or volatile.

- We made the left-hand sides of rules b and c reference types. In the old list they only made sense for lvalues, although that was not stated.
- We changed T —> T& into rule d. This corresponds to a change already made in the language; for example:

```
void f(int&);
void g(const int&);
f(5);      // now illegal
g(6);      // legal
```

- For clarity, we added an optional dimension to the array type of rule b.
- We added rules e and f; they were simply missing from the list (although they were mentioned in 13_2p11).

2.2.3 Operators That Take Or Yield Lvalues

Built-in assignment (5_17p1) is defined to require a modifiable reference instead of a modifiable lvalue (37) on the lhs.

The type signatures of unary * and unary & become attractively simple:

```
unary *:    T* —> T&
unary &:    T& —> T*
```

(For unary `&` we have omitted discussion of the version which yields a pointer-to-member.)

The class member access operators (`.` and `->`, 524) yield results similarly to primary-expressions. If the member is of type “reference to `T`”, the operator yields a “reference to `T`”. If the member is of type “`T`”, where “`T`” is not a reference type, the operator yields a reference to possibly qualified “`T`”; `const` or `volatile` qualifiers are appropriated from the left-hand operand.

2.3 Minor Clarifications

Certain functions, the result of `e1.member_func` or `e2->member_func`, cannot have their address taken. Explicit destructors are also in this category. Right now the fact that you can’t take the address of these functions is a special case.

We can encode this fact by saying that these function expressions yield type `T(args)`, where `T` is not a reference type. Every other function expression yields type `T(&)(args)`, and so can have its address taken (explicitly or implicitly).

2.4 The Major Benefit

In the current WP, there are two nearly identical entities, lvalues and expressions of reference type. In every context we must consider the effect of either of these entities. Cutting the two down to one simply means that we have less to think about.

Unfortunately, bit-field types rear their ugly head instead. I think that they (unlike lvalues) can be confined to a few sections of the WP, thus limiting their impact. More detailed analysis is necessary here.

3. Folding Bit-Fields into the Type System

3.1 Current WP

In the current WP, the one important difference between lvalues and references is the status of bit-field expressions:

```
struct A { int i:4; } a;
... a.i ...;         // a (bit-field) lvalue of type int
```

Bit-field lvalues such as `a.i` cannot be made into references (96p3); they are the only such lvalues. The intent of this restriction is to prevent bit-fields from bleeding out to the rest of the language (we don’t want to allow pointers to bit-fields!).

There is one place where bit-field lvalues do interact strangely with another feature, the conditional operator. Because `a.i` in the example above is an lvalue of type `int`, the following expression is legal:

```
int cond, j, f();
(cond ? a.i : j) = f();   // legal
&(cond ? a.i : j);        // unspecified; should be illegal
```

There are two problems with conditional expressions involving bit-fields. First, generating code for them is tricky (no C++ compiler I know of implements them). And second, they must themselves be considered bit-fields, so that the last line of the above example becomes illegal.

3.2 My Proposal

We create a new kind of derived type (362):

bit-field of length `N` and type `T`.

A bit-field of length N and type T is written `T :N`, as though `:N` were a declarator. There are references to bit-fields (in limited contexts), and there are classes, structures and unions containing bit-fields, but there are no arrays of bit-fields, no functions taking or returning bit-fields, no pointers to bit-fields, no constants of bit-field type, and no pointers to class members of bit-field type.

We add a trivial conversion to the list above::

Table 3: Addition to Revised Trivial Conversions

	From	To	Constraints
i	$qualifiers_{opt}$ `T(&):N`	`T`	`T` cannot be a qualified type

The type on the left is "reference to bit-field of length `N` and type possibly-qualified `T`". To avoid changing the language, we disallow the user from writing this type explicitly or forming it using typedefs. We also modify the constraint of trivial conversion a:

Table 4: Change in Rule a of Revised Trivial Conversions

	From	To	Constraints
a	$qualifiers_{opt}$ `T&`	`T`	`T` cannot be a function type, array type, qualified type, *or bit-field type*

There are no other conversions involving references to bit-fields. In particular, a "reference to bit-field of type `T`" cannot be converted to a "reference to `T`".

Declarators (8) for bit-fields can only appear in the declaration of a data member of bit-field type. No other bit-field declarators are allowed, not even in a typedef. No references to bit-fields can be declared (although expressions of that type are created by the use of a bit-field data member in an expression).

The conditional operator (5_16) does not accept references to bit-fields as its second or third operand; any second or third operand of type "reference to bit-field of type possibly-qualified `T`" is converted to `a T` (using trivial conversion i). This eliminates a difficult-to-implement feature of the current language.

The address-of operator does not accept references to bit-fields. The other operators that accept references (++, −−, assignment operators, comma operator) accept references to bit-fields. Since there are no bit-field rvalues, most operators are not affected.

4. Canonicalizing Types.

Rvalues of const and volatile type have no special meaning. Accordingly, we eliminate them with the following rules for type canonicalization. Tom Plum suggested these in London; see 92-0041 p. 27.

> A type declared as “function returning qualified `T`” is adjusted to read “function returning `T`”.
>
> A type declared as “function of qualified `T`” (any argument) is adjusted to read “function of `T`” (with the other argument types preserved; how do we say this?).

For example, a function defined as

```
void f(const int i) { ... }
```

has type `void (int)`, although its formal argument `i` still has type `const int`.

Several other rules of this kind are necessary. Here are a few. Note that you must use typedefs to construct an example for any of these rules.

> A type declared as “`const` reference to `T`” is adjusted to read “reference to `T`”. (Or it could be made illegal.)
>
> A type declared as “`volatile` reference to `T`” is *not* adjusted to read “reference to `T`”. (If volatile references are allowed,they have special semantics. I have not allowed for them in this paper.)
>
> A type declared as “`const` array of `T`” is adjusted to read “array of `const T`”.
>
> A type declared as “`volatile` array of `T`” is adjusted to read “array of `volatile T`”.
>
> A redundant qualifier is eliminated, eg:

```
typedef const int CI;
const CI ci = 5;          // ci is a const int, not
                          // a const const int.
```

See also trivial conversion rule a in section 2.2.2 above; it strips qualifiers from a type during dereferencing.

The WP probably needs an explicit phase of type canonicalization during the parsing of types.

This phase probably comes just after the interpolation of typedef names; again, see 92-0041 pp. 26-27.

Note that ANSI C needs a type canonicalization phase as well. For ANSI C this phase also includes a rewrite of `float` to `double` for old-style function definitions.