2016-10-16

Operator Dot Wording

Bjarne Stroustrup (bs@ms.com)
Gabriel Dos Reis (gdr@microsoft.com)

Abstract

This is the proposed wording for allowing a user-defined operator dot (**operator.()**) for specifying "smart references" similar to the way we provide "smart pointers." The gist of the proposal (P0416R0) is that if an **operator.()** is defined for a class **Ref** then by default *every* operation on a **Ref** object is forwarded to the result of **operator.()**. However, an operation explicitly declared as a member of **Ref** is applied to the **Ref** object without forwarding.

This wording is the result of EWG discussions based on

- Bjarne Stroustrup and Gabriel Dos Reis: *Operator Dot (R3)*. P0416R0
- Bjarne Stroustrup and Gabriel Dos Reis: *Operator Dot (R2)*. N4477
- Bjarne Stroustrup and Gabriel Dos Reis: *Operator Dot*. N4173

1. Wording

The proposed changes are against the Committee Draft document N4604.

1.1. Signature of a member access function

Modify definition of 'signature' (1.3.18) as follows:

> <function> name, return type if class member access function (13.5.6), parameter type list (8.3.5), and enclosing namespace (if any)

1.2. Definition of surrogate type

Add to paragraph 3.9/8:

> A *reference surrogate type* is a class type with at least a dot access function (13.5.6).

1.3. Semantics and value category of dot-access expressions

Modify bullet (5.1.1) of paragraph 8.5.3/5 as follows

is an lvalue (but is not a bit-field), and "*cv1* T1" is reference-compatible with "*cv2* T2", or T2 is a reference surrogate type (3.9) and overload resolution (13.3.1.8) finds at least one viable dot access function with return type reference-related to T1, or

Add a new bullet just before bullet (5.2) in paragraph 8.5.3/5:

If T2 is a reference surrogate type (3.9), then the best dot access function is selected through overload resolution as described in 13.3.1.8 with cv1 T1 as the target type.

1.4. Class member lookup in the presence of access functions

Add a new paragraph before 10.2/7:

If the lookup is being performed without considering dot access functions or if the declaration set of *S(f,***C***)* is non-empty, the result of name lookup for *f* in **C** is that set. Otherwise, given the dot access function *m*, if the declaration set of *S(m,***C***)* does not contain at least one declaration, the result of name lookup for *f* in **C** is that set. [Note: That means, the result is either empty or the invalid set.]

Otherwise, group the declarations in the declaration set of S(m,C) into partitions indexed by non-cv-qualified types R_i as follows: a class member access function is in a partition indexed by R_i if its return type a possibly cv-qualified R_i, or a reference to a possibly cv-qualified R_i. For each **R_i**, calculate the lookup sets *S(f,***R_i***)* considering dot access functions.

- If, for all **R_i**, the declaration set of *S(f,***R_i***)* is empty, the result of name lookup for *f* in **C** is the empty set.
- Otherwise, if the declaration set of at least one *S(f,***R_i***)* is the invalid set, the result of name lookup for *f* in **C** is the invalid set.
- Otherwise, if exactly one of the declaration sets of *S(f,***R_i***)* is non-empty, the result of name lookup for *f* in **C** is that set.
- Otherwise, the result of name lookup is the invalid set.

[Example:

```
struct A { int x; };
struct B { int x; int y; };
struct C {
    A& operator.();
    const A& operator.() const;
    B& operator.();
};
void f(C& c) {
     c.y = 42;     // OK.
     c.x = 7;      // ERROR: ambiguity.
}
```

```
struct D {
  A& operator.();
};
D d1 { };
D d2 { };
d1 = d2;  // OK: invokes A::operator=;
struct E {
   A& operator.();
   E& operator=(const E&) = default;
};
E e1 { };
E e2 { };
e1 = e2; // E::operator=
struct H { int x { }; };
struct J { int x { }; int y { }; };
struct K  {
   J& operator.();
 };
struct L : H, K { };
int main() {
   L d { };
    d.y = 42;    // OK
    d.x = 17;    // ERROR.

--end example]
```

Modify paragraph 10.2/7 as follows:

~~The result of name lookup for *f* in **C** is the declaration set of *S(f,* **C***)*. If it~~ If the result of name lookup for *f* in **C** is an invalid set, the program is ill-formed. […]

1.5. Dot-access conversion

Modify paragraph 12.3/1 as follows:

Type conversions of class objects can be specified by constructors ~~and by~~, conversion functions, and by dot-access conversions.

Modify paragraph 12.3/4 as follows:

At most one user-defined conversion (constructor, or conversion function, or dot-access conversion) is implicitly applied to a single value.

Add the following to the example in 12.3/4:

```
struct A {
        int operator.();
};

struct B {
        A operator.();
};

int i = B{ }; // OK: B{}.B::operator.().A::operator.();
```

Add a new section titled **12.3.3 Dot-access conversion [class.conv.access]**

1. An *elemental access conversion* from a class X to a type T is an invocation of a dot-access function (13.5.6) declared in the class X with return type T. A *dot-access conversion* is either an elemental access conversion, or an elemental access conversion followed by one or more user-defined conversion sequences (13.3.3.1.2) where all user-defined conversions are elemental access conversions, followed by an elemental access conversion.
[*Example*:

```
        struct A {
                int operator.();
        };
        void f(int);   // #1
        void f(string); // #2;
        void g(int);   // #3;
        void g(A);     // #4;

        A a { };
        f(a);     // OK: calls #1 with a.operator.();
        A b { };
        g(b);     // OK: calls #4, Identity conversion
```

--end example]

2. A dot-access conversion is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it).

1.6. Overloading and overload resolution of member access functions

Modify bullet (2.1) of paragraph 13.1/2 as follows:

> Function declarations, other than class member access function declarations, that differ only in the return type, the exception specification (15.4), or both cannot be overloaded. Class member access functions that differ only in exception specifications cannot be overloaded.

Add a new bullet to paragraph 13.3/2 and modify the opening statement as follows:

Overload resolution selects the function to call in ~~seven~~ several distinct contexts in the language:

– Invocation of dot access function for access from an object of reference surrogate type

Modify paragraph 13.3.1/1 as follows:

The subclauses of 13.3.1 describe the set of candidate functions and the argument list submitted to overload resolution in each of the ~~seven~~ contexts in which overload resolution is used. […]

Modify the note in paragraph 13.3.1.2 as follows:

If not operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to Clause 5. [*Note*: Because ~~.~~ **.*** and **::** cannot be overloaded, these operators are always built-in operators interpreted according to Clause 5….]

Add the following row to the table 10 in paragraph 13.3.1.2/2 as follows

13.5.6	a.	(a).operator.()	

Modify the third bullet of paragraph 13.3.1.2/3 as follows:

For the operator **,**, the unary operator **&**, the operator **.**, or the operator ->, the built-in candidate set is empty. …

Modify paragraph 13.3.1.2/8 as follows:

The second operand of operator **.** (resp. **->**) is ignored in selecting ~~an operator ->~~ the corresponding operator function, and is not an argument when the ~~**operator ->**~~ operator function is called. When ~~**operator ->**~~ the operator function returns, the operator **.** (resp. ->) is applied to the value returned, with the original second operand.

Modify paragraph 13.3.1.2/9 as follows:

If the operator is the operator **,**, the unary operator **&**, operator **.**, or the operator ->, and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to Clause 5.

Add a new section **13.3.1.8** titled **Initialization by surrogate reference [over.match.dot]**

1. Under the conditions specified in 8.5.3, a reference can be bound directly to a glvalue that is the result of applying a dot-access conversion to an initializer expression. Overload resolution is used to select the dot access functions to be invoked. Assuming that "cv1 T" is the underlying type of the reference being initialized, and "cv S" is the type of the initializer expression, with S a surrogate reference type, the candidate functions are selected as follows:

– The dot access function **operator.** is looked up in the class scope S (10.2). Only those access member functions with return type reference-related to "cv2 T2" (when initializing an lvalue reference) or "rvalue reference to cv2 T2" (when initializing an rvalue reference), are candidate functions.

2. The argument list has one argument, which is the initializer expression.

Append **.** to the grammar production of *operator* in paragrtaph 13.5/1 and modify the note as follows:

> The last ~~two~~ three operators are function call (5.2.2), subscripting (5.2.1), and dot access. …

Remove **.** from the list in paragraph 13.5/3.

Modify the second sentence of paragraph 13.5.2/1 as follows:

> Thus, for any prefix unary operator **@**, **@x** can be interpreted as either **x.operator@()** or **operator@(x)** where the lookup of the member function operator@ considers user-declared dot access functions (10.2).

Modify the second sentence of paragraph 13.5.3/1 as follows:

> Thus, for any binary operator **@**, **x@y** can be interpreted as either **x.operator@(y)** or **operator@(x,y)** where the lookup of the member function **operator@** considers user-declared dot access functions (10.2).

Modify the third sentence of paragraph 13.5.4/1:

> Thus, a call **x(arg1,...)** is interpreted as **x.operator()(arg1, ...)** for a class object **x** of type **T** if ~~**T::operator()(T1, T2, T3)** exists~~ the lookup of the member function **operator()** in the scope of **T** considering dot access functions (10.2) is successful and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

Modify the third sentence of paragraph 13.5.5/1:

> Thus, a subscripting expression **x[y]** is interpreted as **x.operator[](y)** for a class object x of type **T** if ~~**T::operator[](T1)** exists~~ the lookup of the member function **operator[]** in the scope of T considering dot access functions (10.2) is successful and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

Modify the first two sentences of paragraph 13.5.6/1 as follows:

> operator-> ~~shall be a non-static member function taking no parameters. It~~ implements the class member access syntax that uses ->.

Modify the third sentence of 13.5.6/1 as follows:

> An expression x->m is interpreted as (x.operator->())->m for a class object x of type T if ~~T::operator >() exists~~ the lookup of the member function operator-> in the scope of T considering dot access functions (10.2) is successful and if the operator is selected as the best match function by the overload resolution mechanism (13.3).

Add the following paragraph before paragraph 13.5.6/1:

> A class may declare a member function **operator.** called a *dot access function*. Similarly, a class may declare a member function **operator->** called an *arrow access function*. The dot access function and the arrow access function are collectively called *class member access functions*. A class member access function shall be a non-static member function and shall take no parameter.
> [Example:
> struct A { int x; };
> struct B {
> A& operator.(); // OK
> };
> struct C {
> A operator.(); // OK
> };
> struct D {
> int operator.() { return a; }
> int a;
> };
> --end example]

Add to the end of section 13.5.6 Class member access [over.ref]:

> **operator.** implements the class member access that is not through a pointer, whether the syntax explicitly uses **.** or not.
> *postfix-expression* . *template*$_{opt}$ *id-expression*
> Unless **m** is explicitly declared to be a **public** member of **x**'s class or the destructor, the expression **x.m** is interpreted as **(x.operator.()).m** for a class object **x** of type **T** if **T::operator.()** exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3). [Note: If **p** is a pointer, **p->m** is interpreted as **(*p).m** (5.2.5) where dot access functions are not considered during the member lookup – end note]. [Note: It is "**public**" rather than "accessible" to prevent **x.m** to have different meanings in different contexts for the same **x**. – end note]
> [Example:
>
> ```
> template<typename T>
> class Ref {
> public:
>     T& operator.() { return *p; }
> ```

```
        void bind(S*);
        // …
    };

    struct S {
        int m;
        void f();
        Enum E { e1 };
    };

    Void use(Ref<S> r)
    {
        r.bind(new S);   // call r.bind(), not r.operator().bind()
        r.m = 1;         // r.operator.().m = 1
        r.f();           // r.operator.().f();
        r.E x;           // error: a type name cannot appear after dot
        S* p = &r;       // p = &r.operator.()
        Ref<S>* p = &r; // error: cannot assign an S* to a Ref<S>*
        p->m = 2;        // error: Ref has no member m
    } -- End example]
```

Multiple **operator.()**s can be defined for a class. If **operator.()** is selected to be called and there is more than one **operator.()** declared, selection among the **operator.()**s is done by looking at the specified member and the return types of the **operator.()**s. An expression **x.m** is valid if there is a unique **operator.()** with a return type **T** (or optionally cv qualified reference to **T**) for which **T** has a member **public** member **m**. [Example:

```
    struct T1 {
        void f1()
        void f(int);
        void g();
        int m1;
        int m;
    };

    struct T2 {
        void f2()
        void f(const string&);
        void g();
        int m2;
        int m;
    };

    struct S3 {
        T1& operator.() { return p; }  // use if the name after . is a member of T1
        T2& operator.() { return q; }  // use if the name after . is a member of T2
        // …
    private:
```

```
        T1& p;
        T2& q;
};

void (S3& a)
{
        a.g();          // error: ambiguous
        a.f1();         // calls a.p.f1()
        a.f2();         // call a.q.f2()
        a.f(0);         // error: ambiguous 'f'
        a.f("asdf");    // error: ambiguous 'f'

        auto x0 = a.m;      // error: ambiguous
        auto x1 = a.m1;     // a.p.m1
        auto x2 = a.m2;     // a.q.m2
}  -- end example]
```

Unary and binary operators are interpreted as calls of their appropriate operator functions (13.5) so that the previous rule apply [Note: for example, **x=y** is interpreted as **x.operator=(y)** which is interpreted as **x.operator.().operator=(y)** and **++x** is interpreted as **x.operator++()** – end note]. Implicit or explicit destructor invocations do not invoke **operator.()**.