WG21-P0252R0
2016-02-13

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

Wording

Add to paragraph 3.9/8:

> A reference *surrogate type* is a class type with at least one declaration of dot access function.

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 the overload resolution process as described in 13.5.6 with cv1 T1 as the target type.

Add new paragraph 9.3/10 to section 9.3:

> 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 *member access functions*. A member access function shall be a non-static member function and shall not be a template. A member access function shall take no parameter. The return type of a dot access function shall be a cv-qualified class type or a cv-qualified reference type.

Append to paragraph 10.2/2 the following:

> If *B* is an unambiguous base class of *D*, the notation *B-in-D* designates the *access path* from *D* to the base class subobject of type *B* in *D*. If the class *D* declares a member access function *mf* with return type *T*, then the notation *T-via-D-by-mf* denotes the access path from *D* to the object or reference of type *T* specified by *mf*. Applying the access path *T-via-D-by-mf* to an entity *obj* of type *D* means invoking the member function *mf* on *obj*.

Modify the introductory sentence of paragraph 10.2/3 as follows:

> The *lookup set* for **f** in **C**, called *S(f,C)*, consists of two component sets: the *declaration set*, a set of members f; and the *subobject set*, a set of ~~subobjects~~ access paths to the class scopes where declarations of these members (possibly including using-declarations) were found.

Modify the introductory sentence of paragraph 10.2/5 as follows:

> Otherwise, if *f* is being looked up in the context of a *postfix-expression* of the form *expr.***template**$_{opt}$ *f* (resp. *expr->***template**$_{opt}$ *f*) where *expr* is an expression of type "*cv* **C**" or an rvalue reference of type **C**, let mf_i denote each of the associated member access functions in **C** (replacing *using-declaration*s with the member access functions they designate) with return type $\mathbf{T}_i$. For each mf_i, compute the lookup set $S(f,\mathbf{T}_i)$ in the context of the expression $expr_i.f$ (resp. $expr_i$->f) where $expr_i$ represents the invocation of the member function mf_i on the expression *expr*. If at least one lookup set $S(,\mathbf{T}_i)$ has a non-empty declaration set, the calculation of $S(f,\mathbf{C})$ is complete and specified as follows:
>
> – If the declaration set of any $S(f,\mathbf{T}_i)$ is invalid, then the declaration set of $S(f,\mathbf{C})$ is invalid, and the subobject set of $S(f,\mathbf{C})$ is the union of all $S(f,\mathbf{T}_k)$ with each access path precomposed with $\mathbf{T}_k$-*via*-**C**-*by*-mf_k.
>
> – Perform overload resolution on the subset of the access member functions mf_k (given the implicit object designated by *expr*) with lookup sets $S(f,\mathbf{T}_k)$ that have non-empty declaration sets. If overload resolution succeeds, with best candidate mf_j as a result, then $S(f,\mathbf{C})$ is $S(f,\mathbf{T}_j)$ where every access path is replaced by precomposition by $\mathbf{T}_j$-*via*-**C**-*by*-mf_j.
>
> – Otherwise, the declaration set is invalid, and the subobject set is the union of the subobject sets of $S(f,\mathbf{T}_k)$ with all the access paths precomposed by $\mathbf{T}_k$-*via*-**C**-*by*-mf_k.
>
> [Example:
> ```
> struct A { int x; };
> struct B { int x; int y; };
> struct C {
>    A& operator.();
>    B& operator.();
> };
> void f(C& c) {
>     c.y = 42;     // OK.  S(y,C) = { { B::y }, { B-via-C-by-operator.-returning-B& }
> ```

```
    c.x = 7;    // ERROR: ambiguity.  S(x,C) = { invalid, { A-via-C-operator.-returnin-A&, B-via-C-by-operator.-returning-B&} }
}
```
--end example]

Otherwise (i.e. C does not contain a declaration of f, an access member function leading to f, or the resulting declaration set is empty), S(f, C) is initialy empty. If C has base classes, calculate the lookup set of f in each direct base subobject B_i, and merge each such lookup set S(f, B_i) in turn into S(f,C). If the subobject set of any $S(f,\mathbf{B}_i)$ has an access path that contains **T**-*via*-**B**-*by-mf*$_i$ then all other lookup sets must have empty declaration sets, and $S(f,\mathbf{C})$ is $S(f,\mathbf{B}_i)$; otherwise the declaration set of $S(f,\mathbf{C})$ is invalid and the subobject set of $S(f,\mathbf{C})$ is the union of all the subobject sets of the $S(f,\mathbf{B}_i)$.

[Example:

```
struct A { int x { }; };
struct B { int x { }; int y { }; };
struct C  {
   B& operator.();
};
struct D : A, C { };
int main() {
   D d { };
   d.y = 42;   // OK
   d.x = 17;   // ERROR.
}
```
--end example]

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

> Function declarations, other than member access function declarations, that differ only in the return type, the exception specification (15.4), or both 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 sequence of member access functions to an initializer expression. Overload resolution is used to select the member 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 access member function **operator.** is looked up in the class scope S (10.2). Those access member functions with return type "cv2 T2" (when initializing an lvalue reference) or "cv2 T2" or "rvalue reference to cv2 T2" (when initializing an rvalue reference), where "cv1 T" is reference-compatible (8.5.3) with "cv2 T2", 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.

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

operator. shall be a non-**static** member function taking no parameters. It 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) as ever, and not as (***p).operator.().m** – 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);         // calls a.p.f(0)
        a.f("asdf");    // call a.q.f string("asdf")

        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.()**.