Doc No: X3J16 / 95-0192
WG21 / N0792
Date: Sep. 26, 1995
Project: Programming Language C++
Reply To: Ben Schreiber
BenSc@Microsoft.Com

Issues Regarding Function `Typedef`s

Overview

This paper intends to resolve the issues raised by Core Issue 479, and further issues raised on the Core reflector, starting with Core-6154.

Survey of Common Practice and Oppinions

For this survey, I have used Microsoft Visual C++ 4.0[1], Symantec C7.0 and BC4.0. Pieces of the example used are inserted to illustrate points. The entire example used is included at the end of this paper. Although the issues involved mainly function typedefs being used to declare member functions, this paper takes a quick look at all uses of function typedefs, including those that are valid C.

1. Can a typedef be used for a function type?

Yes. This is common practice and established C / C++.

2. Can a function be declared using that typedef?

Yes. This is common practice and established C / C++.
Even so, Sean Corfield, in core-6192 voiced a preference for this not being permitted.

3. Can a member function be declared using that typedef?

This is the first question raised by Core Issue 479:

```
typedef void fv(void);        // Legal.
struct S1 {
    fv mfuncv;                // Legal?
    static fv sfuncv;         // Legal?
};
```

Yes to both. `mfuncv` is a non-static member function, and `sfuncv` is a static member function. All compilers surveyed permit this, and a number of committee members have voiced an oppinion that this is intended to work.
Clearly, `sfuncv` should work, because the only difference between a static member function and a non-member function is scoping. `mfuncv` should work because it is common practice.
One problem that is encountered by allowing this is the analogous situation with templates. From Core-6185:

[1] I used pre-release 4.0, because (embarassingly enough), 2.0 suffered an internal error compiling this sample. No bugs were fixed in order to execute this test.

```
      template<typename T>
          class X {
              T member;
          };
          typedef void VF();
          X<VF>   x;      // Legal?
```

At the Monterey meeting it was decided to make this is ill-formed. It is my belief that this discrepancy is acceptable, as there is already precedent for requiring that it be possible to perform syntax analysis on a class template.

4. Can a pointer-to-member-function be declared using that typedef?

Example:

```
      struct S2 {
          void mfuncv(void);        // For reference
          void cmfuncv(void) const; // For reference
      };
      fv S2::*pms2fv = &S2::mfuncv; // Legal?
```

Yes. pms2fv has type “pointer-to-member function of struct S2.” In keeping with 3, this is supported by all compilers surveyed, and the sentiments voiced on the reflector are that this is legal. The “member-ness” of the function is acquired from the pointer-to-member.

5. Can such a typedef be declared at member scope?

Example:

```
      struct S3 {
          typedef void fv(void);    // Legal?
      };
```

Yes. `S3::fv` is the same type as `fv`: “function of (void) returning void”, i.e. the member context affects scoping but not meaning. That is how all compilers surveyed interpret this. The one problem with permitting this is that it is somewhat confusing. Especially in the presence of namespaces, there is little utility in doing this, so it may be reasonable to constrain this.

6. Can such a typedef be used with qualifiers?

This is the area that goes farthest out on a limb. None of the compilers surveyed gave reasonable results for all of these examples. When discussed, there were no strong feelings in favor of supporting these, although also no strong feelings in favor of disallowing them.

Declaring a member

Example:

```
      struct S4 {
          const fv cmfuncv;         // Legal?
      };
```

If this works, this clearly has to mean exactly the same thing as:

```
      struct S4 {
          void cmfuncv(void) const;
      };
```

All compilers surveyed handled this one OK.

Declaring a pointer-to-member

Example:

```
      struct S5 {
          void MF_CALL cmfuncv(void) const;
      };
      const fv S5::*pcms5fv = &S5::cmfuncv;     // Legal?
```

If allowed, this should mean the same thing as:

```
      void (S5::*pcms5fv)(void) const = &S5::cmfuncv; // Legal?
```

This works on SC7.0, but not on BC4.0 or VC4.0.

Declaring a non-member

Example:

```
      const fv func3;         // Surely not legal
      const fv *pfunc1;       // Legal?  Is this a
                              // pointer-to-member?
```

SC7.0 is the only one that diagnosed an error on the first example; BC4.0 ignored the qualifier, and VC4.0 got confused. None of the compilers surveyed did anything meaningful with the second example. Since these are not member function contexts, the qualifier can't have any meaning. I recommend that in these cases, the combination should be an error. A viable alternative would be to have the qualifier ignored, as when combined with a reference, but I think there is little utility in that. In particular, the main utility of permitting the qualifier to be ignored when applied to a reference is in templates. Use of function types as parameters are already disallowed in templates, so there is no utility in allowing the combination in this case.

Putting a qualifier in the typedef

It would seem nice to be able to include qualifers in a typedef used to define member functions. However, none of the compilers surveyed supports this (VC and SC disallowed the typedef declaration, BC ignored the const), and it seems pretty far out on a limb, so I do not recommend allowing it.

Recommendations

Since all of these issues require both rulings on their legality and on their meaning, each issue requires explicit wording to be added to the Working Paper, regardless of resolution. In order to expidite resolution of these issues, I will present wordings for each of the primary options for each issue. Note that the issues are listed such that for any issue that is resolved "disallow," all subsequent issues must be resolved "disallow."

Issues 1 & 2

Allow

This is the status quo. Some explicit examples would help, but are not needed.

Disallow

I do not consider this a viable option.

Issues 3 & 4

Allow

Allowing use of function typedefs for member functions and pointer-to-member-functions is the orthogonal thing to do, and is current practice.
To clarify the issue, add the following to the end of 9.4 [class.mfct]:

> A member function may be declared (but not defined) using a typedef for a function type. The resulting member function has exactly the same type as if the function declarator was explicitly provided. [For example:

```
typedef void fv(void);
struct S {
      fv    memfunc1;
      void  memfunc2(void);
};
fv S::* pmfv1 = &S::memfunc1;
fv S::* pmfv1 = &S::memfunc2;
```

-*end example*]

Since the pointer-to-member-function case is already implied by the combinatorics, this example serves to illustrate the intent that indeed this should work.

Disallow

The benefit of disallowing this is that it sidesteps the need to define how a type “function ...” becomes “member function ...”. It is probably the most consistant option.
Add a new paragraph to 9.2 [class.mem]:

> A non-`static` meber function must be declared using function declarator sytax.

Alternatively, the rule could be:

> A meber function must be declared using function declarator sytax.

The latter is more strict than needed to resolve this technical issue, but it also resolves the issue with templates, rendering the text added at the Monterey meeting redundant.
Also, add a new paragraph to 8.3.3 [dcl.mptr]:

> A pointer to member function must be declared using function declarator sytax.

Issue 5

Allow

Add to 9.10p1 [class.nested.type]:

> A type name for a function type denotes a plain function type, not a member function type. [Note: this means that the only effect of declaring it at class scope is that it is a nested type name.]

Disallow

Add to 9.10p1 [class.nested.type]:

> A type name for a function type is not allowed at class scope.

Issue 6

Combining qualifiers with function typedefs.

Allow

It is my recommendation that, if allowed, it should only be allowed in contexts where the function type will become a member-function type. To define what this means, add the following to the end of 8.3.5 [dcl.fct] p3:

> At all other times during the determination of a type, if a type of the form “ *cv-qualifier-seq* function returning T” is formed, it must be in a declaration of a nonstatic member function or of a pointer to member function; the *cv-qualifier-seq* modifies the `this` pointer, as if it had appeared in the function declarator.

Disallow

This is already partially disallowed by the end of 8.3.5p3 [dcl.fct]. To disallow other combinatorics, replace the last two sentences of p3 with:

> A *cv-qualifier-seq* can only be part of a declaration or definition of a nonstatic member function, and of a pointer to member function; it modifies the `this` pointer, see 9.4.2. It is part of the function parameter type list. At any time during the determination of a type, if a type of the form

"*cv-qualifier-seq* funcion returning T" is formed, other than from a *cv-qualifier-seq* in a function declarator, the program is ill-formed.

A Related Editorial Issue

Core Issue 479 also presents a question about pointer-to-member declarator sytax: (from Bill Gibbons):

```
struct T { void f(); };
typedef void (*PF)();
PF T::pmf = &T::f;  // pointer to member function ?
```

Bill Gibbons' proposed answer is No: "That is, once a declarator has been established to be a pointer to member, the distinction of pointer to data member or pointer to member function is entirely dependent on the type, not on the syntax."

I agree with this resolution, and propose the following editorial clarification to reslove the issue. Add to 8.3.3 [dcl.mptr] p3:

> [Note: the type "pointer-to-member" is distinct from the type "pointer", and cannot be built from a combination of "pointer" and "member" declarators (there is no "member" declarator). That is, once a declarator has been established to be a pointer to member, the distinction of pointer to data member or pointer to member function is entirely dependent on the type, not on the syntax.]

Note that this resolution is independent of the resolutions to the other issues in this paper.

Appendix: Example Used for Survey

```
//
// Compiler survey: use of typedefs to function types for
// declaring member functions.
//

#ifdef _MSC_VER
#define MF_CALL __cdecl
#else
#define MF_CALL
#endif

// Declare a typedef for type "function of (void) returning void"

typedef void fv(void);        // Legal

fv func1;               // Legal;
void func1(void);       // identical to this line


// Use it to declare a member:

struct S1 {
    fv mfuncv;                // Legal?
};
void (MF_CALL S1::* pms1fv)(void) = &S1::mfuncv;
                        // Confirm that we got a
                        // member function

// Use it to declare a pointer to member function

struct S2 {
    void MF_CALL mfuncv(void);            // For reference
    void MF_CALL cmfuncv(void) const;     // For reference
};
fv S2::*pms2fv = &S2::mfuncv; // Legal?

// Define the typedef inside a class -- typedef is expected
// to be a non-member function type

struct S3 {
    typedef void fv(void);    // Legal?
    fv mfuncv;                // Legal?
};

S3::fv func2;                 // Legal?
void (*pfv)(void) = &func2;   // Confirm we got a
                        // non-member function

void (MF_CALL S3::*pms3fv)(void) = &S3::mfuncv;
                        // Confirm that we got a
                        // member function

// Adding qualifiers

const fv func3;               // Surely not legal
const fv *pfunc1;       // Legal?  Is this a
                        // pointer-to-member?
void test1(void) {
    pfunc1 = &func1;          // OK if pfunc1 is non-member
    pfunc1 = &S2::cmfuncv;    // OK if pfunc1 is a member
}

// Adding qualifiers in member context

struct S4 {
    const fv cmfuncv;         // Legal?
};
void (MF_CALL S4::*pcms4fv)(void) const = &S4::cmfuncv;
                        // Confirm that we got a
                        // member function with
                        // const 'this'

// Adding qualifiers in pointer-to-member context

struct S5 {
    void MF_CALL cmfuncv(void) const;
};
const fv S5::*pcms5fv = &S5::cmfuncv;     // Legal?

// Declaring a function typedef with qualifers

typedef void cfv(void) const; // Legal?
cfv cfunc1;             // Legal?

// Use it

struct S6 {
    cfv cmfuncv;        // Legal?
};
void (MF_CALL S6::*pcms6fv)(void) const = &S6::cmfuncv;
                        // Confirm that we got a
                        // member function with
                        // const 'this'
```