Member Functions in C Structures
SC22WG14 N652 — NCITS/J11 97-015

7 January 1997

Bill Seymour
wseymour@email.usps.gov

This paper suggests the changes that need to be made to C9X to allow C structures to contain member functions and to provide a predefined identifier, **this**, that can be used to reference other structure members within the body of a member function definition. Because they are controversial, the **public** and **private** keywords and the **::** operator are proposed in separate papers (N653 / 97-016 and N654 / 97-017, respectively) so that they can be accepted or rejected independently.

Throughout this proposal, it is assumed that a member function is a kind of function, not something fundamentally different from a function. Although this requires language to be added in several places to the effect that the text does not apply to member functions, the alternative would have been to describe two kinds of functions, member and non-member, and to say explicitly which kind is meant every place that the word "function" appears in C9X.

One significant difference between member functions and other functions is that there is no strictly conforming pointer to a member function because this would require a special pointer-to-member operator for C++ compatibility.

5.2.4.1 Translation limits

Change (from N590 adopted in Toronto):

— 31 significant initial characters in an external identifier

to:

— 31 significant initial characters in an external identifier, not counting any characters that the implementation may add to allow for member function linkage as required by 6.1.2.2

Rationale

This leaves the limit on the number of characters that a programmer may use at 31 as decided in Toronto. How vendors provide member function linkage is unspecified; but if they add characters to function names, then those additional characters must not count among the 31.

6.1.2.2 Linkages of identifiers

[1] Change the second sentence from:

There are three kinds of linkage: external, internal, and none.

to:

There are four kinds of linkage: external, internal, member function, and none.

[2] Change the first sentence from:

In the set of translation units and libraries that constitutes an entire program, each instance of a particular identifier with *external linkage* denotes the same object or function.

to:

In the set of translation units and libraries that constitutes an entire program, each instance of a particular identifier with *external linkage* denotes the same object or function, and each instance of a particular identifier declared with *member function linkage* within compatible structure types denotes the same function.

Rationale

The intent is to require that, given the declarations:

```
/* translation unit 1 */
struct A { int foo(void); } a;

/* translation unit 2 */
struct A { int foo(void); } b;

/* translation unit 3 */
typedef struct A { int foo(void); } C;
C c;

/* translation unit 4 */
typedef struct A { int foo(void); } D;
D d;
```

a.foo(), **b.foo()**, **c.foo()** and **d.foo()** all call the same function.

6.1.2.5 Types

[11] Change the second bullet from:

— A *structure type* describes a sequentially allocated nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

to:

— A *structure type* describes a nonempty set of member objects and/or member functions. The storage for each member object shall be allocated sequentially. Each member object has an optionally specified name and possibly distinct type. Each member function has a specified name and possibly distinct type.

And add between the function type and pointer type bullets the following language that mimics the function type language:

— A special case of function type is *member function type* which describes a function associated with a structure type and with specified return type. A member function type is characterized by the structure in which it is declared, its return type, and the number and types of its parameters. A member function type is said to be derived from its return type and enclosing structure, and if its return type is *T* and enclosing structure is *S*, the member function type is sometimes called “member function of *S* returning *T*.” The construction of a member function type from a structure and a return type is called “member function type derivation.”

6.1.2.7 Predefined identifiers (subclause added in Toronto)

Add to the list of predefined identifiers:

this

6.1.6 Punctuators

Add to the list of punctuators:

::

6.2.2.1 Lvalues and function designators

[1] In the last sentence, change "does not have any member" to "does not have any member object".

Rationale

This sentence defines a *modifiable lvalue* as an lvalue that is not const-qualified and that, if it is a structure or union, has no const-qualified member. This requirement should be limited to member objects because a member function can be const-qualified, too; but that is simply a promise that the member function does not modify the object for which it is called.

[4] Change from:

> A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator[36] or the unary **&** operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*."

to:

> A *function designator* is an expression that has function type. A special case of a function designator is a *member function designator* which is an expression that has member function type. Except when it is a member function designator or the operand of the **sizeof** operator[36] or the unary **&** operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*." There is no strictly conforming "pointer to member function of…" type.

6.3.1.2 The identifier this (new subclause after 6.3.1.1 which was added in Toronto)

Semantics

The identifier **this** is implicitly declared by the translator as if, immediately following the implicit declaration of **__func__** in a member function definition, the declaration

```
Q T * const this;
```

appeared, where *Q* is the possibly empty list of type qualifiers that appear in the member function declaration and *T* is the structure type of which the function is a member. The value of **this** shall be as specified in 6.3.2.4.

Within the body of a member function definition, any identifier that could denote a member of a structure of type *T*, is not already immediately preceded by **this->**, and is not otherwise in scope, shall be translated as if it were the primary expression, **(this->*identifier*)**.[footnote]

[*footnote*]. Thus, within a member function body, a structure member may be referenced without being preceded by **this->** unless it is hidden by another identifier.

6.3.2.2 Function calls

Change constraint **[1]** from:

> The expression that denotes the called function[41] shall have type pointer to function returning void or returning an object type other than an array type.

to:

> The expression that denotes the called function shall have type pointer to function[41], or it shall denote a member function, returning void or returning an object type other than an array type.

Rationale

The above language is needed because we don't have pointers to member functions. Note that the reference to footnote 41 which reads

> 41. Most often, this is the result of converting an identifier that is a function designator.

has been moved to make it apply, not to any expression that denotes a called function, but rather to such an expression that is a pointer.

[4] After "visible for this identifier" add the footnote:

> This implies that such a function is not a member function because identifiers that designate member functions must have visible declarations.

[5] Change the next to last sentence from:

> If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.6.6.4.

to:

> If the expression that denotes the called function has type pointer to function returning an object type, or designates a member function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.6.6.4.

[6] After "that does not include a prototype" add the footnote:

> This implies that such a function is not a member function because member functions must have prototypes.

6.3.2.3 Structure and union members

Change the subclause title to

Structure and union member objects

and add the sentence

This subclause does not apply to member functions which are covered in 6.3.2.4.

6.3.2.4 Postfix increment and decrement operators

Renumber to 6.3.2.5, then insert the new subclause:

6.3.2.4 Structure member functions

This subclause does not apply to member objects which are covered in 6.3.2.3.

Constraints

[1] The first operand of the **.** operator shall have a qualified or unqualified structure type, and the second operand shall name a member function of that type.

[2] The first operand of the **->** operator shall have type "pointer to qualified or unqualified structure," and the second operand shall name a member function of the type pointed to.

[3] In either case, in translation phase 7, the expression must be followed immediately (except for white space) by parentheses that enclose an optional argument expression list.

Semantics

[4] A postfix-expression followed by a dot **.**, an identifier that designates a member function, and parentheses that enclose an optional argument expression list, calls a member function of the structure object designated by the postfix-expression preceding the dot. Within the body of the member function definition, the predefined identifier, **this**, shall be a pointer to the object designated by the postfix-expression preceding the dot.

[5] A postfix-expression followed by an arrow **->**, an identifier that designates a member function, and parentheses that enclose an optional argument expression list, calls a member function of the structure object pointed to by the postfix-expression preceding the arrow. Within the body of the member function definition, the predefined identifier, **this**, shall have the value of the postfix-expression preceding the arrow.

Rationale

Member functions are described in a new subclause because it would be too difficult to describe both member objects and member functions in the same subclause. Constraint [3] is needed because there are no pointers to member functions.

6.5.2.1 Structure and union specifiers

[2] Change the first sentence from:

> A structure or union shall not contain a member with incomplete or function type.

to:

> A structure or union shall not contain a member with incomplete or function type, except that a structure declared at file scope may contain a member with member function type.

Rationale

A structure that contains one or more member functions may not be declared at block scope because the member function itself would have to be defined at file scope where its prototype would not be visible.

[4] Change:

> As discussed in 6.1.2.5, a structure is a type consisting of a sequence of named members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of named members, whose storage overlap.

to:

> As discussed in 6.1.2.5, the storage for member objects of a structure is allocated in an ordered sequence, whereas the storage for member objects of a union overlap.

[5] Change:

> Structure and union specifiers have the same form.

to:

> Structure and union specifiers have the same form, except that a structure specifier may declare member functions.

Add after **[5]**:

> The declaration of a member function in a struct-declaration-list is the member function's prototype.

6.5.2.1 **Structure and union specifiers** (continued)

[7] Change the first sentence from:

> A member of a structure or union may have any object type.

to:

> A member of a structure may have any object or member function type, and a member of a union may have any object type.

[11] Change:

> non-bit-field member

to:

> non-bit-field member object

[12] Change:

> Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may therefore be unnamed padding within a structure object, but not at its beginning, as necessary to achieve the appropriate alignment.

to:

> Within a structure object, the non-bit-field member objects and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member object (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may therefore be unnamed padding within a structure object, but not at its beginning (except in the special case described below of a structure that contains member functions only), as necessary to achieve the appropriate alignment.

And then add

> In the special case of a structure that contains one or more member functions but no member objects, the implementation shall treat such a structure as if it contained one unnamed member object of unspecified type.

And add the following footnote:

> This guarantees that the size of any structure is at least 1.

6.5.4 Declarators

Change:

direct-declarator **(** *parameter-type-list* **)**

to:

direct-declarator **(** *parameter-type-list* **)** *type-qualifier-list*$_{opt}$

And add the constraint:

A direct declarator of the form

direct-declarator **( )**

shall not be used in a member function prototype or member function definition, and a direct declarator of the form

direct-declarator **(** *parameter-type-list* **)** *type-qualifier-list*

shall be used only in a member function prototype or member function definition.

6.5.4.3 Function declarators (including prototypes)

[9] Change the first sentence from:

For two function types to be compatible, both shall specify compatible return types.

to:

For two function types to be compatible, both shall specify compatible return types; and if one is a member function type, then both shall be member function types declared in compatible structures.

6.5.7 Initialization

In **[7]**, change “member of that type.” to “member object of that type.”

Add between **[9]** and **[10]**:

Member functions of structure objects do not participate in initialization.

6.7.1 Function definitions

Change the grammar to:

function-definition:
  declaration-specifiers$_{opt}$ *declarator* *declaration-list*$_{opt}$ *compound-statement*
  declaration-specifiers$_{opt}$ *name-space-specifier* *declarator* *compound-statement*

name-space-specifier:
  struct$_{opt}$ *identifier* **::**

Add to constraints:

If a name-space-specifier is present, the declarator and any declaration-specifiers that are present shall agree with those in the member function prototype.

Add to semantics:

If a name-space-specifier is present, the function-definition defines a member function, otherwise the function-definition defines a function that is not a member function.

Renumber example 2 as 3, and insert example 2:

2. In the following:

```
int T::less(int other_value)
{
    return my_value < other_value;
}

int T::more(int other_value) const
{
    return this->my_value > other_value;
}
```

less and **more** are member functions of a structure of type *T*. As stated in 6.3.1.2, the translator shall implicitly declare the identifier **this** as:

```
T * const this;
```

in the body of **less**, and as:

```
const T * const this;
```

in the body of **more**. The identifier **my_value** shall be translated as **(this->my_value)** in the body of **less**, but not in the body of **more**.

6.7.1 Function definitions (continued)

In the example that was renumbered as 3, change:

> To pass one function to another,

to:

> To pass one function that is not a member function to another function,

7.1.6 Common definitions <stddef.h>

After “evaluates to an address constant.” in **[3]**, add the footnote:

> This implies that *member-designator* does not designate a member function because there are no pointers to member functions.

Rationale

The **offsetof** macro cannot find the offset of a member function.

G.1 Unspecified behavior

Add:

— The size of a structure that contains member functions only (6.5.2.1).