Document Number: WG14 N424/X3J11 95-025

C9X Revision Proposal
=====================

Title: Classes in C - Part 1: Basic Classes

Author: Robert Jervis
Author Affiliation: Sun Microsystems, Inc.
Postal Address: 2550 Garcia Ave., Mountain View, CA 94043 USA
E-mail Address: robert.jervis@eng.sun.com
Telephone Number: +1 415 3367964
Fax Number: +1 415 9640946
Sponsor: ____
Date: 1995-04-21
Proposal Category:
- __ Editorial change/non-normative contribution
- __ Correction
- X New feature
- __ Addition to obsolescent feature list
- __ Addition to Future Directions
- __ Other (please specify) ____

Area of Standard Affected:
- __ Environment
- X Language
- __ Preprocessor
- __ Library
  - __ Macro/typedef/tag name
  - __ Function
  - __ Header
- __ Other (please specify) ____

Prior Art: C++
Target Audience:

These features are useful to a wide range of programmers. The facilities help improve problems of name-space pollution by grouping member functions into classes. Public and private data members help control access to data structures so that the implementation can be more effectively separated from the interface of a piece of code.

Classes have proven to be especially helpful in writing windowing applications, graphics and data base applications.

Related Documents (if any): C++ Draft Standard

Proposal Attached: X Yes __ No, but what's your interest?
Abstract:

This proposal includes the exact wording changes needed to add class types to the C Standard. Class types in this proposal are a subset of C++ and are intended to be upward-compatible with it. Please advise me if I have accidentally introduced incompatibilities in this and the following proposals.

Proposal: ____

The wording changes are summarized in the following points.

* Both member data and functions can be declared.

* Calling member functions uses the same syntax as C++.

* Only prototypes may appear inside a class declaration. No inline member function definitions are allowed.

* Members are private by default, as in C++.

* Members can be declared to be public or private. Protected members may not be declared (these are only relevant if inheritance is also supported).

* The object mentioned in a member function call is passed as a hidden parameter. This hidden parameter can be referenced using the identifier this.

* Static members, including static member functions are not included in this proposal.

* Member functions can be qualified with const and volatile qualifiers.

* The proposal defines four new keywords:

```
class
private
public
this
```

* Inheritance, virtual functions, constructors and destructors are addressed in the next proposals. As a result they are not included here.

ISSUES:

These items are details of the proposal where the committee may wish to consider alternatives beside what is presented here.

* This proposal uses the term 'member function' for members of classes with function type. The much shorter and easier to use name 'method' is also widely used in programming literature. The word 'method' does not convey, however, the nature of the entity being named as well as 'member function'. A simple search and replace of one term for the other is all that is needed if the committee would prefer to use 'method'.

* The 'this' keyword could be defined as a normal identifier that has a special meaning inside member functions. In C++ it is a keyword, but can only be used meaningfully inside member functions. If we want to preserve existing C code that might use 'this' as a variable name, we could preserve that code and avoid one keyword. See sections 6.1.1 and 6.3.1 for details.

* The :: token proposed for member function definition syntax could be described in the grammar as two adjacent colon tokens. This would avoid adding a new token, but risks porting C code to C++ because we would allow white space between the colons and C++ would not. See sections 6.1.6, 6.5.2.4 and 6.5.4 for the details.

* This proposal makes no promises about common initial sequences of members in different classes, where the existing Standard does make promises for structure types. See section 6.3.2.3 for the details.

* This proposal makes class tags behave exactly like existing C structure tags. That is, in C++ tags can be used as normal identifiers in effect making them typedefs. I think that the issue of tags as typedefs is a more general issue, and not one limited to classes. As a result, I have done nothing to change C's treatment of tags. See section 6.5.2.3 for details.

The following are the specific changes to the Standard. Section numbers are in reference to the International Standard ISO/IEC 9899:1990 Programming Languages - C. Where relevant, changes affecting the TC1 defect reports are stated where they appear in the Standard. The TC2 Defect Reports have not been scanned for possible changes, nor has the normative addendum.

## 6.1.1 Keywords

ADD THE FOLLOWING KEYWORDS TO THE LIST:

```
class
private
public
this
```

## 6.1.2 Identifiers

Page 20, line 2, IS:

> ... a tag or a member of a structure, union, or enumeration; ...

SHOULD BE:

> ... a tag or a member of a class, structure, union, or enumeration; ...

RATIONALE:

> This simply lists the kinds of things an identifier can refer to.

### 6.1.2.1 Scopes of Identifiers

RATIONALE:

> Class scope is needed because you want to refer to the members of a class without having to say 'this->member' all the time. C++ allows this convenience, and it is widely used. It has no run-time overhead, since it is merely syntactic sugar for the full expression.

Page 20, line 22, IS:

> There are four kinds of scopes: function, file, block, and function prototype.

SHOULD BE:

> There are five kinds of scopes: function, file, class, block, and function prototype.

RATIONALE:

> Class member names can be used in member functions as follows:

```
        class A {
                int      foo;

                void     bar();
                };

        void A::bar()
        {
                foo = 3;        /* means this->foo = 3; */
```

}

A special class scope concept must be introduced so that the binding of class members is resolved properly.

Page 20, line 28-29, IS:

... appears outside of any block or list of parameters, ...

SHOULD BE:

... appears outside of any block, class declaration, member function definition or list of parameters, ...

RATIONALE:

This language restricts file scope identifiers to exclude class declarations. Those identifiers have class scope, not file scope.

Page 20, line 30, BEFORE:

If the declarator or type specifier ...

INSERT:

If the declarator or type specifier that declares the identifier appears inside a class declaration list, the identifer has class scope, which terminates at the } that closes the class specifier and resumes for the duration of each member function definition associated with the same class encountered later in the same translation unit.

RATIONALE:

This language specifies exactly what identifiers have class scope. It is a little complicated by the fact that member functions have to be prototyped in the class declaration, but defined outside it.

Page 20, line 39, IS:

Structure, union, and enumeration tags have scope ...

SHOULD BE:

Class, structure, union, and enumeration tags have scope ...

RATIONALE:

This just describes the scope of class tags as being the same as other tags. C++ allows tags to be used as if they were typedef names (more or less). This proposal does not do this. I believe that using tags as typedef names should be addressed as an independent proposal. Such a proposal logically includes all tag names, not just class names.

6.1.2.2 Linkages of Identifiers

RATIONALE:

Member functions have to be prototyped in headers that get included many times in a program's different translation units. They also need no more than one implementation somewhere. All references to the member function have to bind to the right class and function. Member functions in different classes have different names.

In practice, class linkage wll be accomplished by some form of name mangling: combining the class name and the function name into a single extern name.

If we want to support static objects inside classes (a feature of C++), they will have to have class linkage as well.

Page 21, line 8-9, IS:

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

SHOULD BE:

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

RATIONALE:

In order for member functions in distinct classes to bind to distinct definitions, there also needs to be a class linkage.

Page 21, line 13 BEFORE:

Identifiers with no linkage denote unique entities.

INSERT:

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

RATIONALE:

Each class has its own set of member functions. Moreover, calls in one translation unit must bind to definitions in another.

Page 21, line 20-21, IS:

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage class specifier extern.

SHOULD BE:

If the declaration of an identifier for a function has no storage-class specifier and the identifier does not have class scope, its linkage is determined exactly as if it were declared with the storage class specifier extern.

RATIONALE:

This change just excludes member functions from functions that get treated as extern.

Page 21, BEFORE line 24 INSERT:

An identifier with class scope declared to be a function has class linkage.

RATIONALE:

This defines the set of identifiers that have class linkage: member functions.

6.1.2.3 Name Spaces of Identifiers

RATIONALE:

This section presents something of a problem. The existing name spaces of C are not overlapping. If we add classes to C, class members actually participate in two name spaces: structure members (because they can appear as operands of the . and -> operators), and also ordinary identifiers (since they can appear as plain references inside member function definitions).

The language of this section does not seem to forbid this overlap, however, so I have simply listed class members in both name spaces.

Page 21, line 36-37, IS:

* the tags of structures, unions, and enumerations (disambiguated by following any of the keywords struct, union or enum).

SHOULD BE:

* the tags of classes, structures, unions, and enumerations (disambiguated by following any of the keywords class, struct, union or enum).

RATIONALE:

This simply indicates that class tags are in the tag name space.

Page 21, line 38-40, IS:

* the members of structures or unions; each structure or union has a separate name space ...

SHOULD BE:

* the members of classes, structures or unions; each class, structure or union has a separate name space ...

Page 22, line 1-2, IS:

* all other identifiers, called ordinary identifiers (declared in ordinary declarators or as enumeration constants).

SHOULD BE:

* all other identifiers, called ordinary identifiers (declared in ordinary declarators, as members and referenced within a member function or as enumeration constants).

6.1.2.5 Types

RATIONALE:

Classes are a new form of aggregate type closely related to structures. This section lays out the general attributes of a class.

✓

Page 22, line 28 IS:

Types are partitioned into object types (types that describe objects), function types (types that describe functions), and incomplete types ...

SHOULD BE:

Types are partitioned into object types (types that describe objects), function types (types that describe functions and member functions), and incomplete types ...

RATIONALE:

Member functions do not have types that can be converted to normal function types, but they clearly arenot object types. There is no special need to create a completely new category of type, but we do need to indicate clearly where member function types fit.

Page 23, line 28 BEFORE:

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

INSERT:

* A class type describes a sequentially allocated nonempty set of member objects and member functions. Each member object has an optionally specified name, possibly distinct type and visibility. Each member function has a specified name, possibly distinct type and visibility.

Page 23, line 37 BEFORE:

* A pointer type may ...

INSERT:

* A member function type describes a function associated with a class type and with specified return type. A member function type is characterized by the class in which it is declared, the 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 class, and if its return type is T and enclosing class is C, the member function type is sometimes called "member function of C returning T." The construction of a member function type from a class and a return type is called "member function type derivation."

RATIONALE:

Member functions have a distinct range of types, different from all function types. This distinction reflects the existence of the implied "this" parameter. This proposal does not establish pointers to member functions. Such pointers are not needed terribly often, so excluding them avoids the complexity of calls to member function pointers, which would require additional syntax.

Page 24, line 6-7 IS:

Array and structure types are collectively called aggregate types.

SHOULD BE:

Array, class and structure types are collectively called aggregate types.

RATIONALE:

What else can classes be?

Page 24, line 9-10 IS:

A structure or union type of unknown content (as described in 6.5.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

SHOULD BE:

A class, structure or union type of unknown content (as described in 6.5.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same class, structure or union tag with its defining content later in the same scope.

RATIONALE:

This defines classes tag to be incomplete and completable in the same was a structure types.

Page 24, line 13 IS:

Array, function, and pointer types ...

SHOULD BE:

Array, function, member function, and pointer types ...

RATIONALE:

This section makes a member function type a form of derived declarator type.

6.1.6 Punctuators

Page 32, line 4 ADD TO THE LIST OF PUNCTUATOR TOKENS:

::

RATIONALE:

This token is needed for member function definitions. C++ makes other uses of this token which are not included in this proposal. The next proposal that include inheritance add some syntax that makes use of this token as well.

6.3.1 Primary Expressions

Page 39, line 4 ADD TO THE LIST OF primariy-expression PRODUCTIONS:

this

RATIONALE:

The keyword 'this' is used in C++ expressions as if it were a variable name.

Page 39, line 11 BEFORE:

A constant is a primary expression.

INSERT:

The keyword 'this' is a primary expression, provided it is used within the block that is the body of a member function definition. The type of the operand is a constant pointer to the class of the member function being defined. If the member function type is qualified, the operand type is a constant pointer to the class qualified in the same way as the member function.

RATIONALE:

The object named in a member function call is passed as an implied parameter to the member function. The 'this' keyword is used to refer to the parameter.

The addition of qualified function types and their effect on 'this' allows the programmer to be able to declare const or volatile instances of a class object. Class objects so qualified can only be used to call similarly qualified member functions.

6.3.2 Postfix Operators

Page 39, line 28 AFTER:

```
        postfix-expression --
```

INSERT:

```
        postfix-expression . identifier (
                            argument-expression-list opt )
        postfix-expression -> identifier (
                            argument-expression-list opt )
```

RATIONALE:

The specification of member-function calls could be written so that the existing syntax is used without these productions. Technically the new productions actually introduce an ambiguity in the grammar. I think it is helpful, however, to emphasize the distinct nature of a member function call.

This way, a new separate section can be added that describes member function calls spearately from normal function calls and member references. Otherwise, a member function call would have to be identified as a function call whose first operand is a member reference that names a member function. This complicates those sections more than I like.

6.3.2.2 Function calls

Page 41, line 22-23 IS:

Recursive member function calls shall be permitted, both directly and indirectly through any chain of other functions.

SHOULD BE:

Recursive member function calls shall be permitted, both directly and indirectly through any chain of other functions or member functions.

RATIONALE:

This merely requires that recursion work regardless of the mix of function calls and member function calls.

Page 41, line 33 BEFORE:

An argument may be ...

INSERT:

If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and that identifier names a member function, the whole call expression is treated as a member function call whose initial tokens were 'this->' followed the tokens of the function call expression.

RATIONALE:

This allows a programmer to omit the 'this->' tokens within a member function definition. This is a significant convenience in writing code.

6.3.2.3 Structure and Union Members

CHANGE THE SECTION TITLE TO:

6.3.2.3 Class, Structure and Union Members

RATIONALE:

Classes behave like structures or unions with respect to . and -> operators. One minor exception is the paragraph beginning on page 42, line 5 that describes the behavior of overlapping objects and common initial sequences of members in structures. I have added no language to make any promises about memory layout in different classes.

Page 41, line 33-37, IS:

The first operand of the . operator shall have a qualfiied or unqualified structure or union type, and the second operand shall name a member of that type.

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

SHOULD BE:

The first operand of the . operator shall have a qualfiied or unqualified class, structure or union type, and the second operand shall name a member of that type.

The first operand of the -> operator shall have type "pointer to qualified or unqualified class", "pointer to qualified or unqualified structure" or "pointer to qualified or unqualified union", and the second operand shall name a member of the type pointed to.

The named member operand shall have object type and shall be visible.

RATIONALE:

This language describes the constraints on the . and -> operators. These operators have to be extended to include classes. The final constraint makes references to member functions or private members (outside their scope) off limits.

Page 41, line 39-40 IS:

A postfix expression followed by a dot . and an identifier designates a member of a structure or union object.

SHOULD BE:

A postfix expression followed by a dot . and an identifier designates a member of a class, structure or union object.

RATIONALE:

This just includes classes where structures and unions are mentioned.

Page 42, line 1-2 IS:

A postfix expression followed by an arrow -> and an identifier designates a member of a structure or union object.

SHOULD BE:

A postfix expression followed by an arrow -> and an identifier designates a member of a class, structure or union object.

RATIONALE:

This just includes classes where structures and unions are mentioned.

Page 42, line 34 ADD TO FORWARD REFERENCES:

class specifiers (6.5.2.4)

NEW SECTION:

Page 43, before line 11:

6.3.2.5 Member Function Calls

Constraints

The first operand of the . form of the member function call operator shall be an lvalue and have a qualfiied or unqualified class type, and the second operand shall name a member of that type.

The first operand of the -> form of the member function call operator shall have type "pointer to qualified or unqualified class" and the second operand shall name a member of the type pointed to.

The named member operand shall have member function type and shall be visible.

The number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

A postfix expression followed by a dot ., an identifier and followed by parentheses () containing a possibly empty, comma-separated list of expressions is a member function call. The value of 'this' in the called member function shall be the address of the object designated by the first operand.

A postfix expression followed by an arrow ->, an identifier and followed by parentheses () containing a possibly empty, comma-separated list of expressions is a member function call. The value of 'this' in the called member function shall be the value of the the first operand.

Arguments in a member function call have the same semantics as arguments in a function call.

Recursive member function calls shall be permitted, both directly and indirectly through any chain of other functions or member functions.

If the operand that names the called member function has type member function returning an object type, the member function call expression has the same type as the object type, and has the value determined as specified in 6.6.6.4. Otherwise, the member function call has type void.

RATIONALE:

This produces the core semantics of a call to a member function in the absence of inheritance in C++. The efficiency of a member function call is exactly the same as a call to a corresponding normal function with one additional paraemter: this.

6.5.2 Type Specifiers

Page 58, line 34 BEFORE:

struct-or-union-specifier

INSERT:

class-specifier

Page 59, line 18 BEFORE:

* struct-or-union specifier

INSERT:

* class specifier

Page 59, line 22 IS:

Specifiers for structures, unions, and enumerations are discussed in 6.5.2.1 through 6.5.2.3.

SHOULD BE:

Specifiers for classes, structures, unions, and enumerations are discussed in 6.5.2.1 through 6.5.2.4.

Page 59, line 27 IS:

Forward references: enumeration specifiers (6.5.2.2), structure and union specifiers (6.5.2.1), tags (6.5.2.3), type definitions (6.5.6).

SHOULD BE:

Forward references: enumeration specifiers (6.5.2.2), class specifiers (6.5.2.4), structure and union specifiers (6.5.2.1), tags (6.5.2.3), type definitions (6.5.6).

RATIONALE:

Classes are declared using the same grammatical unit, more or less, as structures and unions. To avoid renumbering sections, I have added a new section for class specifiers at the end of section 6.5.2, even though in other places classes are usually mentioned first. In the Draft, we may wish to reorder these sections in order to present class specifiers before structure and union specifiers.

6.5.2.3 Tags

Page 62, 16-23, IS:

A type specifier of the form

```
    struct-or-union identifier { struct-declaration-list }
```

or

```
    enum identifier { enumerator-list }
```

declares the identifier to be the tag of the structure, union or enumeration specified by the list. The list defines the structure content, union content, or enumeration content. If this declaration of the tag is visible, a subsequent declaration that uses the tag and that omits the bracketed list specifies the declared structure, union or enumerated type.

SHOULD BE:

A type specifier of the form

```
    class identifier { class-declaration-list }

    struct-or-union identifier { struct-declaration-list }
```

or

```
    enum identifier { enumerator-list }
```

declares the identifier to be the tag of the class, structure, union or enumeration specified by the list. The list defines the class content, structure content, union content, or enumeration content. If this declaration of the tag is visible, a subsequent declaration that uses the tag and that omits the bracketed list specifies the declared class, structure, union or enumerated type.

RATIONALE:

This wording simply makes class tags work exactly like structure, union or enumeration tags. Note that in C++ tags can be used as if they were typedefs.

```
    class foo { ... };

    class foo X;            /* keyword 'struct' needed in C */

        // in C++

    foo     X;              // The keyword can be omitted.
```

Of course, in C++, these declarations could be followed by:

```
        int     foo;            // Legal, foo is now an int.

        class foo Y;            // Now the keyword is needed because
                                // of the variable declaration.
```

If the committee wants the C++ behavior, then something must be done in section 6.5.2 to make that clear.

Page 62, line 25 IS:

If a type specifier of the form

    struct-or-union identifier

occurs prior to the declaration that defines the content, the structure or union is an incomplete type. It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed. If the type is to be completed, another declaration of the tag in the same scope (but not in an enclosed block, which declares a new type known only within that block) shall define the content. A declaration of the form

    struct-or-union identifier ;

specifies a structure or union type and declares a tag, both visible only within the scope in which the declaration occurs. It specifies a new type distinct from any type with the same tag in an enclosing scope (if any).

A type specifier of the form

    struct-or-union { structure-declaration-list }

or

    enum { enumerator-list }

specifies a new structure, union, or enumerated type, within the translation unit, that can only be referred to by the declaration of which it is a part.

INSERT:

If a type specifier of the form

    class identifier

or

    struct-or-union identifier

occurs prior to the declaration that defines the content, the class, structure, or union is an incomplete type. It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed. If the type is to be completed, another declaration of the tag in the same scope (but not in an enclosed block, which declares a new type known only within that block) shall define the content. A declaration of the form

    class identifier ;

or

    struct-or-union identifier ;

specifies a class, structure, or union type and declares a tag, both visible only within the scope in which the declaration occurs. It specifies a new type distinct from any type with the same tag in an enclosing scope (if any).

A type specifier of the form

```
class { class-declaration-list }

struct-or-union { structure-declaration-list }
```

or

```
enum { enumerator-list }
```

specifies a new class, structure, union, or enumerated type, within the translation unit, that can only be referred to by the declaration of which it is a part.

RATIONALE:

These rules deal with incomplete class declarations and forward declarations. Again, I have made classes behave exactly as existing C structures and unions.

NEW SECTION:

Page 64, before line 1:

6.5.2.4 Class Specifiers

Syntax

```
class-specifier:
        class identifier opt { class-declaration-list }
        class identifier

class-declaration-list:
        class-declaration
        class-declaration-list class-declaration

class-declaration:
        visibility-specifier opt
                class-specifier-qualifier-list
                        struct-declarator-list ;

class-specifier-qualifier-list:
        type-specifier class-specifier-qualifier-list opt
        type-qualifier class-specifier-qualifier-list opt
        visibility class-specifier-qualifier-list opt

visibility-specifier:
        visibility :

visibility:
        public
        private
```

Constraints

A class shall not contain a member with incomplete type. Hence it shall not contain an instance of itself (but may contain a pointer to an instance of itself).

The declarator of a member declared with member function type shall include a prototype.

Semantics

As discussed in 6.1.2.5, a class is a type consisting of a sequence of named members, whose storage is allocated in an ordered sequence.

The presence of a class-declaration-list in a class-specifier declares a new type, within a translation unit. The class-declaration-list is a sequence of declarations for the members of the class as well as optional visibility specifiers. If the class-delcaration-list contains no named members, the behavior is undefined. The type is incomplete until after the } that terminates the list.

A member of a class may have any object or member function type. In addition, a member may be a bit-field.

Each non-bit-field member of a class with object type is aligned in an implementation defined manner appropriate to its type.

Within a class 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. Whether a pointer to a class object, suitably converted, points to its initial member is unspecified. Therefore there may be unnamed padding anywhere within a class object, even at its beginning.

There also may be unnamed padding at the end of a class as necessary to achieve the appropriate alignment were the class to be an element of an array.

A member is visible (can be used as an operand of a dot . or arrow -> operator or in a member function call) within its scope or if it has public visibility.

A member has public visibility

* if the keyword public appears in the class-specifier-qualifier-list of the declaration of the member, or

* if the keyword private does not appear in the class-specifier-qualifier-list of its declaration and the nearest visibility-specifier appearing in the class' class-declaration-list in or before the member's declaration contains the keyword public.

All other members have private visibility.

Example

```
class point {
        int     x, y;

        public:

        void    set(int newx, int newy);

        int     getx(void);

        int     gety(void);
};

class point Cursor;
```

makes point the tag of a class, and then declares Cursor as an object that has this type.

The class contains five members: x, y, set, getx, and gety.

The members x and y have private visibility (which means that they can only be referenced within the member functions set, getx and gety. The member functions are all public, however, since they follow a public visibility specifier. Thus, they can be called from any function where this class declaration is visible, not just the member functions themselves.

RATIONALE:

The use of public and private members together with member functions are the mechanism that gives classes one of their great strengths: the ability to constrain users of a class from unrestricted manipulation of private data. This provides language support for an important principle of software design: the separation of implementation and interface.

One could add member functions to C structures and not include the visibility specifiers, but that would eliminate one of the important features of classes. I believe that the improved enforcement of interfaces that results from these features justifies the four new keywords needed to implement them.

Private data members mean that member functions, or some related feature, are also needed. Friend functions in C++ can be used, for example, to gain access to private data. With neither member functions nor friends, there is no way to access private data, thus making them useless features to have.

Friends do not need 'this' (since a friend is a normal function). Friends also do not introduce the problem of pointers to member functions (which are not defined in this proposal). There is still a new keyword ('friend') so including friends but not member functions doesn't reduce the number keywords needed.

Friends are not proposed here because they, being normal functions, share the global name space of C. THe name space pollution problem is significantly eased, however, by using member functions. Each member function need only have a unique name within the same class. Any number of other classes can use the same names for their own member functions.

In C++ programs, friends are not the preferred method of access to private data, member functions are. I think that using member functions would allow for a larger body of code to be portable and appropriate between C and C++.

Note that the specification of member function types mandates that prototypes must be used. Old-style function declarators are not allowed in a class declaration.

6.5.4 Declarators

Page 65, line 18 IS:

```
                    identifier
```

SHOULD BE:

```
                    identifier
                    identifier :: identifier
```

Page 65, line 21 IS:

direct-declarator ( parameter-type-list )

SHOULD BE:

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

RATIONALE:

Function declarators need to support const and volatile qualfication so that const and volatile qualified class objects can be properly manipulated with member functions.

Page 65, line 45 BEFORE:

In the following subsections, ...

INSERT:

A direct-declarator of the form identifier :: identifier declares the second identifier, and asserts that it is the name of a member with member function type and that the first identifier is the name of the class in which the member appears.

RATIONALE:

This describes the syntax needed to define member functions. In this proposal, the :: syntax only needed in member function definitions. The use of this token to qualify references in expressions is included in a later sub-proposal on inheritance.

The syntax is included here because the :: syntax must be embedded in a potentially complex declarator in the definition itself. Using this here simplifies the member function definition syntax below.

6.5.4.3 Function Declarators

Page 67, line 28 BEFORE:

Semantics

INSERT:

A function declarator with a non-empty type-qualifier-list shall appear as the outermost derivation of a member function type.

RATIONALE:

These qualifiers are only meaningful within classes. This sentence is a constraint, mandating a diagnostic if qualifiers are used anywhere else on a function declarator.

6.5.7 Initialization

Page 71, line 38 AS MODIFIED BY TC1#17Q17 IS:

Except where explciitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization. A union object containing only unnamed members has indeterminate value even after initialization.

SHOULD BE:

Except where explcitly stated otherwise, for the purposes of this subclause unnamed members of objects of class, structure, and union type and class members with private visibility or member function type do not participate in initialization. Unnamed members of class or structure objects have indeterminate value even after initialization. A union object containing only unnamed members has indeterminate value even after initialization. Class members with private visibility have the same value as an object with static storage duration and the same type as the member that was not explicitly initialized.

Page 72, line 9-11 IS:

The initializer for a structure or union object that has automatic storage duration either shall be an initializer list as described below, or shall be a single expression that has compatible structure or union type.

SHOULD BE:

The initializer for a class, structure, or union object that has automatic storage duration either shall be an initializer list as described below, or shall be a single expression that has compatible class, structure, or union type.

RATIONALE:

Classes are initialized like structures, except for the addition of private members. This proposal specifies that priovate members are initialized according to the rules for implciitly initialized statics. Unlike unnamed membersw, which can neven be manipulated at all, so that their content is irrelevant, private members can be used from within member functions. That means that a well defined value is much more useful and for static initializers no more expensive.

6.7 External Definitions

Page 81, line 8 IS:

declaration

SHOULD BE:

declaration
member-function-definition

Page 81, line 22-23 IS:

An external definition is an external declaration that is also a definition of a function or an object.

SHOULD BE:

An external definition is an external declaration that is also a definition of a function, an object, or a member function.

NEW SECTION:
Page 84, after line 10 ADD:

6.7.3 Member Function Definitions

Syntax

```
member-function-definition:
        declaration-specifiers opt declarator
                              compound-statement
```

Constraints

The identifier declared in a member function definition shall have a member function type, as specified by the declarator portion of the function definition.

The return type of a member function shall be void or an object type other than array.

The declaration-specifiers of a member function definition shall not include a storage-class specifier.

The function declarator that specifies the type of the member function shall include a parameter list. The declaration of each parameter shall include an identifier (except for the special case of a parameter list consisting of a single parameter of type void, in which there shall not be an identifier).

The class name mentioned in the declarator for a member function shall be visible and shall include a delcaration of a member with the same name and compatible type as that in the definition.

Semantics

The declarator in a member function definition specifies the name of the member function, its class and the identifiers and types of its parameters.

If a member function that accepts a variablenumber of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavipor is undefined.

On entry to the member function the value of each argument expression shall be converted to the type of its corresponding parameter, as if by assignment to the parameter. Array expressions and function designators as arguments are converted to pointers before the call. A declaration of a parameter as "array of type" shall be adjusted to "pointer to type," and a declaration of a parameter as "function returning type" shall be adjusted to "pointer to function returning type," as in 6.2.2.1. The resulting parameter type shall be an object type.

On entry to the member function the value of 'this' is set to the address of the appropriate class object as specified in 6.3.2.5.

Each parameter has automatic storage duration. Its identifier is an lvalue. The layout of storage for parameters is unspecified.

The object that 'this' refers to has automatic storage duration and it is an lvalue. The layout of storage for 'this' is unspecified.

RATIONALE:

This specification only allows member function definitions with prototypes. Since member functions are entirely new, there is no reason to support old-style definitions.

Otherwise, the specification here is the same as for normal functions (except for the bit about 'this').