

**Doc** <N 1816>  
**No:**  
**Date:** 2014-03-18  
**Reply**  
**to:** Max Abramson <aabramson221@ccsnh.edu>

## Adding methods() to structures in C

### Summary

This paper proposes the addition of C++ style methods to structures used in the C programming language. The benefit is shorter compile times than those for C++ programs, a reduced memory footprint, as well as better interoperability with C existing C code and libraries. The emphasis on speed and small size is recognized by this author as aligning itself with the growing use of the C language where performance, size, and real time applications is desired.

The intent, here, is to take the most conservative approach possible. I propose adding features already implemented by C++ compilers that are not likely to create problems for future expansions to the C language, to increase compile time, the difficulty in implementing dedicated C compilers, or to expand the size of binaries generated for existing projects. Additionally, this static, single-inheritance approach to Object-Oriented Programming is simpler than that used in C++ programs, while giving the programmer control over the means by which objects are initialized, if at all.

Members of a structure can be initialized using a C-style initialization list:

```
Date stValentinesDay = (2, 15);
```

or by calling a constructor:

```
Date thanksgiving.Date(11, 22);
```

At present, some compilers will not allow initialization lists to be used where a structure contains any objects, such as strings, nor can data members be skipped over:

```
Date birthday = ("February", , 1996);
```

This author considers this to be a major shortcoming of simple initialization which should be added as part of the standard, but will not be addressed here. Automatic constructors and destructors are among the most popular features in C++, and constructors allow the programmer to create custom methods that allow for static counters and other features, and Resource Allocation Is Initialization are some of the most used features in C++11.

```
struct Ostrich {
    Ostrich();           //no-argument default constructor provided by language
    Ostrich(double speedGuess) { //overloaded constructor taking one double
        (speedGuess) ? m_speedEst = speedGuess : m_speedEst = SPEED_LOOKUP;
        ++numRatites; //pre-increment is faster when working with objects
    }
    ~Ostrich() {
        --numRatites; //static int declared globally
    }
private:
    int m_weight, m_height, m_strideLength;
};
```

For reference, C++ handles backward compatibility with C programs by implementing a structure with a structure, though the default data members and methods remain public, by default, while structures are private by default. C++ handles methods by building a list of pointers to those methods.

In C, a `struct` is able to use nested structures in order to implement the “has a” aggregate or composition pattern. This has become the preferred pattern in object-oriented programming, wherever composition can be used. For reference, aggregation might be thought of as “a community college has students,” while composition may be thought of as “a car is composed of many parts.”

For reference, C++ also allows a class to *inherit from* many parent structures. However, this can result in unreadable or unmaintainable code, as well as the “deadly diamond of death,” where both parent classes share one or more grandparent classes. C++ also allows the use of virtual or dynamic functions, run time type inference, namespaces, and friend classes. This author hopes for the future implementation of namespaces, inheritance, and static, simple features commonly found in C++ code, but leaves questions such as multiple vs. single inheritance for future discussion.

Keeping with the conservative philosophy proposed, and only adding access specifiers and methods, the compiler would view the hierarchy of structures thusly:

- C++ style constructors and destructors are among the most popularly features in the language. They give the programmer more direct control of which actions to take when an object is first instantiated. It is the intent that these be added to the C language in in order to simplify code development.
- More fully support Resource Allocation Is Initialization, both for performance and maintainability of code.
- Access specifiers `protected` and `private` are added to the default of `public` for structures. For reference, C++ maintains backward compatibility with legacy C programs by implementing `struct` as a very simple structure, but with `public` as the default.
- Constructors can be overloaded allowing different versions to be called when a structure is instantiated.
- C++ style constructors and destructors simply use the name of the structure itself and behave differently from other methods. The standard practice in C has been to allow programmers to discover innate functionality in the language, putting a limited set of features to more novel use.
- Nested structures can be expanded with methods that apply only to objects instantiated for them to provide composition and aggregation for the development of useful design patterns in object-oriented programming. This also allows for the development of useful code that might otherwise use inherited data members and methods in OO languages like C++, Java, Objective-C, or Python.

The first three reasons motivate to expansion of the language. The second three are happy byproducts of these features and would help to avoid name collisions and other problems that develop with very large programs.

## Example

For reference, C++ treats structures as structures for backward compatibility, but with the default access specifier as `public`. Again, the paper proposes strict conformance to the C++ standard (WG 21/N3337) in handling structures for C. Code must operate in exactly the same way under C++, or the program must not compile, by default.

```
struct Run {
    const double SPEED_LOOKUP = 55;
    void estRunningSpeed(double hipHeight, double strideLength){...}
protected:
    double m_speedEst;
};

struct Ratites {
```

```

private:
    int m_numRatites;
public:
    Ostrich bob;          //this is the simplest way to nest structures
    Run runner;          //calls no-argument constructor
    void setNumRatites(int num) {
        m_numRatites = num;
    }
    int getNumRatites() {
        return m_numRatites;
    }
};

```

## Wording

The proposed wording draws heavily on the wording in WG21/N3337. Note: It is the intent of this proposal that constructors, destructors, and access specifiers conform to the specifications given in therein. Wording additions are included as four new keywords imported from C++, with identical meaning and syntax, and two new sections building a strict, conservative subset of the *access specifiers* and *constructors/destructors* defined in N3337.

### 6.4.1 Keywords

Add `this`, `public`, `private`, and `protected` to the list of reserved keywords.

Add a new sections 6.7.11, 6.7.12, and 6.7.13:

#### 6.7.11 Member Access

1 A member of a structure can be

- `private`; that is, its name can be used only by members of the structure in which it is declared.
- `protected`; that is, its name can be used only by members of the structure in which it is declared, by structures derived from that structure.
- `public`; that is, its name can be used anywhere without access restriction.

2 A member of a structure can also access all the names to which the structure itself has access. A local structure of a member function may access the same names that the member function itself may access. Access permissions are thus transitive and cumulative to nested and local structures.

3 Members of a structure defined with the keywords `struct` or `union` are `public` by default.

4 Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions. In the case of overloaded function names, access control is applied to the function selected by overload resolution. Because access control applies to names, access control is applied to a typedef name--not the entity referred to by the typedef.

5 It should be noted that it is access to members and base structures that is controlled, not their visibility. Names of members are still visible, and implicit conversions to base structures are still considered, when

those members and base structures are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base structures, the construct is not constructed according to the syntax rules, diagnosable semantic rules, or may have ambiguous meaning. Note: Base structures are not yet a concern for this specification as inheritance is not yet defined, but is included here for future reference.

- 6 All access controls in this section affect the ability to access a `struct` member name from the declaration of a particular entity, including parts of the declaration preceding the name of the entity being declared and, if the entity is a structure, the definitions of members of the `struct` appearing outside the structure's member specification. Note: this access also applies to implicit references to constructors, conversion functions, and destructors.
- 7 Here, all the uses of `A::I` are constructed according to the syntax rules, diagnosable semantic rules, and applies one definition because `A::f`, `A::x`, and `A::Q` are members of `struct A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of `struct A`. Similarly, the use of `A::B` as a base-specifier is constructed according to the syntax rules, diagnosable semantic rules, and applies one definition because `D` is derived from `A`, so checking of base-specifiers must be deferred until the entire base-specifier-list has been seen.
- 8 The names in a default argument (8.3.6) are bound at the point of declaration, and access is checked at that point rather than at any points of use of the default argument.

### 6.7.12 Access specifiers

1 Member declarations can be labeled by an access-specifier:

access specifier : member-specificationopt

An *access specifier* specifies the access rules for members following it until the end of the structure or until another access specifier is encountered.:

```
struct American {
    int boy;           // American::boy is public
    int girl;         // American::girl is public
private:
    float teen;       //American::teen is private
protected:
    float neighbor;  //American::neighbor is protected
public:
    char ism, s;      //American::ism and American::s are public
};
```

- 2 Any number of access specifiers is allowed and no particular order is required.
- 3 Non-static data members of a (non-union) structure with the same access control are allocated so that later members have higher addresses within a structure object. The order of allocation of non-static data members with different access control is unspecified. Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other.

4 When a member is redeclared within its structure definition, the access specified at its redeclaration shall be the same as at its initial declaration.

```
struct Ore {
    struct Iron;
    enum Wood : int;
private:
    struct Iron { };          //error: cannot change access
    enum Wood: int{e0};      //error: cannot change access
};
```

5 Note: In a derived structure, the lookup of a base structure name will find the injected-struct-name instead of the name of the base structure in the scope in which it was declared. The injected-struct-name might be less accessible than the name of the base struct in the scope in which it was declared.

### 6.7.13 Nested structures

1 A nested structure is a member and as such has the same access rights as any other member. The members of an enclosing structure have no special access to members of a nested structure; the usual access rules for private, protected, and public shall be obeyed.

```
struct Eagle {
    int x;          //public, by default
    struct Bird { };
    struct Golden {
private:
    Bird b;        //OK: Eagle::Golden can access Eagle::Bird
    int y;

    void f(Eagle* ptrE, int i) {
        ptrE->x = i;    //OK: Eagle::Golden can access Eagle::x
    }
};

int g(Golden* ptrG) {
    return ptrG->y;    //error: Golden::y is private
}
};
```

Add a new section following 7.31, with the following text:

## 8. Constructors and Destructors

### 8.1 Concepts

- 1 The default constructor and destructor are special member functions. The implementation will implicitly declare these member functions for some structures when the program does not explicitly declare them. Programs shall not define implicitly-declared special member functions.
- 2 Programs may explicitly refer to implicitly-declared special member functions. For example: a program may explicitly call, take the address of or form a pointer to member to an implicitly-declared special member function.
- 3 Note: The special member functions affect the way objects of struct type are created and destroyed, and how values can be converted to values of other types. Such special member functions are called implicitly.
- 4 Special member functions obey the usual access specifiers. For example, declaring a constructor protected ensures that only derived structures can create objects using it.

## 8.2 Constructors

- 1 A special declarator syntax is used to declare or define the constructor. The syntax uses the constructor's struct name and an optional parameter list in that order. In such a declaration, optional parentheses around the constructor struct name are ignored.

```
struct Vector {  
    Vector();           // declares no-argument constructor  
    Vector(int index); // overloaded version that accepts an index value  
};
```

- 2 A constructor is used to initialize objects of its structure type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation will cause a constructor to be called to initialize an object. Note: For initialization of objects of struct type.
- 3 A typedef-name shall not be used as the struct-name in the declarator-id for a constructor declaration.
- 4 A constructor shall not be static. A constructor can be invoked for a const, volatile or const volatile object. A constructor shall not be declared const, volatile, or const volatile (9.3.2). const and volatile semantics are not applied on an object under construction. They come into effect when the constructor for the object ends. A constructor shall not be declared with a ref-qualifier.
- 5 A default constructor for a structure Zak is a constructor of `struct Zak` that can be called without an argument. If there is no user-declared constructor for `struct Zak`, a constructor having no parameters is implicitly declared as defaulted. An implicitly-declared default constructor is an inline public member of its structure. A default constructor for a structure is defined as deleted if:
  - it is a union that has a variant member with a non-trivial default constructor,
  - any non-static data member with no brace-or-equal-initializer is of reference type,
  - any non-variant non-static data member of type `const` (or array thereof) with no brace-equal-initializer does not have a user-provided default constructor,
  - it is a union and all of its variant members are of type `const` (or array thereof),
  - it is a structure, and all members of any anonymous union member are of type `const` or array thereof,
  - any non-static data member with no brace-or-equal-initializer, has `struct` type `M` or array thereof and

- either M has no default constructor or overload resolution as applied to M's default constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or
- any non-static data member has a type with a destructor that is delete or inaccessible from the defaulted default constructor.

A default constructor is *trivial* if it is not user-provided and if:

- it has no non-static data member of its structure that has a brace- or equal-initializer,
- all the direct base structures of its structure have trivial default constructors, and
- for all the non-static data members of its struct that are of struct type (or array thereof), each such structure has a trivial default constructor.

Otherwise, the default constructor is *non-trivial*.

6 A default constructor that is defaulted (and not defined as deleted) is *implicitly defined* when it is used to create an object of its structure type or when it is explicitly defaulted after its first declaration. The implicitly-defined default constructor performs the set of initializations of the structure that would be performed by a default constructor written by the user for that structure without an initializer and an empty compound statement.

7 Default constructors are called implicitly to create struct objects of static, thread, or automatic storage duration defined without an initializer (6.7.9), are called to create struct objects of dynamic storage duration created by a new-expression in which the new-initializer is omitted, or are called when the explicit type conversion syntax (H.2.4) is used. A program is ill-formed if the default constructor for an object is implicitly used and the constructor is not accessible.

8 Note: 12.6.2 describes the order in which constructors for base structures and non-static data members are called and describes how arguments can be specified for the calls to these constructors.

10 No return type (not even void) shall be specified for a constructor. A return statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

11 A functional notation type conversion (5.2.3) can be used to create new objects of its type. Note: The syntax looks like an explicit call of the constructor.

```
complex zz = complex(1,2.3);
cprint(complex(7.8,1.2));
```

12 An object created in this way is unnamed. Note: 12.2 describes the lifetime of temporary objects. Note: Explicit constructor calls do not yield lvalues.

14 During the construction of a const object, if the value of the object or any of its *subobjects* is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor's this pointer, the value of the object or *subobject* thus obtained is unspecified.

```
struct Oscar;
void no_opt(Oscar*);
```

```
struct Oscar {
    int c;
    Oscar() : c(0) { no_opt(this); } // "this" references this structure
};
```

```

const Oscar oscObj;
void no_opt(Oscar* oPtr) {
int i = oscObj.c * 100;    //value of oscObj.c is unspecified
    oPtr->oscObj = 1;
    cout << oscObj.c * 100 << '\n'; //value of oscObj.c is unspecified
}

```

## 8.3 Destructors

- 1 Destructors are specially declared using an optional function-specifier followed by a tilde `~`, followed by the struct name of the destructor with no parameters. If the `(~StructName())` is declared within parentheses, the parentheses are ignored. A typedef name shall not be used as the struct name following the `~` in the declaration.
- 2 A destructor is used to destroy objects of its struct type. The destructor takes no parameters, and no return type (`void`, `int`, `double`, `string`, etc) may be specified. A destructor can be invoked for a `const` or `volatile` object, but it cannot be declared `const` nor `volatile`. `const` and `volatile` semantics are not applied on an object under destruction. They stop being in effect when the destructor for the object starts. A destructor shall not be declared with a ref-qualifier.
- 3 A declaration of a destructor that does not have an exception-specification is implicitly considered to have the same exception-specification as an implicit declaration.
- 4 If a struct has no user-declared destructor, a destructor is implicitly declared as defaulted. An implicitly declared destructor is an `inline public` member of its struct.
- 5 A defaulted destructor for a struct is defined as *deleted*:
  - —if it is a union-like struct that has a variant member with a non-trivial destructor,
  - —if any of the non-static data members has struct type A (or array thereof) and A has a deleted destructor or a destructor that is inaccessible from the defaulted destructor,
  - —if any direct struct has a deleted destructor or a destructor that is inaccessible from the defaulted destructor.
  - A destructor is defined as *trivial* if it is provided by the compiler and:
    - —if all of the direct base structures of its struct have *trivial* destructors, and
    - —if for all of the non-static data members of its struct that are of struct type (or array thereof), each such struct has a trivial destructor.
- Any other destructor is defined as *nontrivial*.
- 6 A *default destructor* that is not defined as *deleted* is implicitly defined when it is used to destroy an object of its struct type or when it is explicitly defaulted after its first declaration.
- 7 Before the defaulted destructor for a struct is implicitly defined, all the default destructors for its base structures and its non-static data members shall have been implicitly defined.
- 8 After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for a structure calls the destructors for its direct non-variant non-static data members, that destructors's direct base structures and, if it is the type of the structure, its destructor calls the destructors for the structure's base structures. All destructors are called as if they were referenced with a qualified name. Bases and members are destroyed in the reverse order of the completion of their

constructor(A B C, ~C ~B ~A). See item 6. A return statement in a destructor might not directly return to the caller; before transferring control to the caller, the destructors for the members and bases are called. Destructors for elements of an array are called in reverse order of their construction. see item 6. Note: inheritance and base structures are not proposed in this paper, but this section is included for reference and completeness.

9 Note: some language constructs have special semantics when used during destruction. See item 7.

10 Destructors are invoked implicitly under any of the following circumstances:

- for constructed objects with static storage duration (3.7.1) at program termination (3.6.3),
- for constructed objects with thread storage duration (3.7.2) at thread exit,
- for constructed objects with automatic storage duration (3.7.3) when the block in which an object is created exits (6.7),
- for constructed temporary objects when the lifetime of a temporary object ends.

A program is ill-formed if an object of struct type or array thereof is declared, and the destructor for the structure is not accessible at the point of the declaration. Destructors can also be invoked explicitly.

11 In an explicit destructor call, the destructor name appears as a `~` followed by a type-name that denotes the destructor's struct type. If the object is not of the destructor's struct type and not of a structure derived from the destructor's struct type, the program has undefined behavior when a destructor is invoked. An explicit destructor call must always be written using a member access operator or a qualified-id; In particular, the *unary-expression* `~StructName()` in a member function is not an explicit destructor call.

12 Note: explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a new-expression with the placement option. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities.

13 Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (3.8). For example: if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined.

14 Note: the notation for explicit call of a destructor can be used for any scalar type name (5.2.4). Allowing this makes it possible to write code without having to know if a destructor exists for a given type.