A Modest Proposal for Curing the Public Field Phobia[*][†]

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece
email: dds@aueb.gr

Abstract

Field accessor methods have become a ubiquitous feature of object-oriented programming. The definition and use of such methods promote code bloat and an unnatural expression style. We propose a simple addition to the C++ language that can move the burden of providing abstraction support for fields from the programmer to the compiler.

Keywords

Field; object accessor method; getter methods; overloading; C++

1 Introduction

In the interest of abstraction, designers typically hide object and class fields from modules outside the class by declaring them as *private* [KM96, p. 37]. Subsequently, when writing in C++ or Java, programmers create public accessor methods [AG96, p. 40] for those fields with names such as *SetFieldName* and *GetFieldName*. Finally, in the interest of efficiency, the compiler (often nudged by the suitable application of the *inline* C++ keyword) tries to optimise those method calls into simple assignment or access operations. Some languages, systems, and style guidelines have even formalised this practice and mandated it for the implementation of interoperable components or for code conforming to specific standards.

In the following sections we outline problems associated with this practice and propose a modest language extension to support an alternative programming style.

[*] *SIGPLAN Notices*, 37(4):54–56, April 2002.

2 Problems

The practice of isolating all object fields by accessor methods leads to code bloat and an unnatural expression style. Both problems result in programs that are less readable and maintainable.

Consider a typical object definition:

```
class Point {
private:
    double x;        // X coordinate
    double y;        // Y coordinate
public:
    // Return X coordinate
    double
    GetX(void)
    {
        return (x);
    }

    // Set X coordinate
    void
    SetX(double val)
    {
        x = val;
    }
    // ...
}
```

Some code style formatting conventions dictate at least 13 lines of code for every object field; the style followed by the authors of Java would require 11 lines of code, while the compact style found in the canonical C++ description [Str97] would require two additional lines of code for every field. In C++, to make the code available for efficient inline expansion, the code is placed in a program area with a very high real-estate value: the class declaration. This is the place where programmers will look to find a class's interface and contractual obligations. The class declaration is also the code that will be included by all files that use this class. Regrettably, we populate this valuable real-estate with trivia that both programmers and compilers could do without. In large, carefully-crafted systems, the number of accessor methods can be significant. As an example, Rose and Rose [RR00] calculated that accessor methods used to obtain the value of a field (getter methods) account for about one fourth of the total number of methods in the standard Java "*java.**" package source classes.

To add insult to injury, the definition of these accessor methods, forces us to abandon the assignment and field access syntax provided by the language in favour of less natural forms of expression. Thus, instead of writing:

```
Point a;
a.x = a.y = 0;
screen.Moveto(a.x, a.y);
```

we have to write:

```
Point a;
a.SetX(0);
a.SetY(0);
screen.Moveto(a.GetX(), a.GetY());
```

Although in pure object-oriented languages, such as Smalltalk, the use of messages for accessing and modifying fields is natural, in languages supporting direct access to fields, such as C++ and Java, accessor methods just hinder code readability.

3 Proposal

The justification behind the creation of accessor methods—a callisthenic ritual followed daily by thousands of programmers—is abstraction. When a class directly exposes its fields as public, programmers that use the class gain insight into the class's implementation details. Future implementation or interface changes will be difficult in a body of code that contains specific references to fields. In our example, if the implementation of the *Point* class changes to use the polar notation, all direct references to *x* and *y* will have to be changed to calls to suitable accessor methods. Similarly, with public fields it is impossible to add some form of access control to a field, e.g. the enforcement of read-only access.

We believe that the amount of code bloat and the unnatural expression style that result from the definition and use of accessor methods justify a small language extension to provide the accessor method functionality in a more friendly fashion. A syntactic construct to express both a method and a field access in the same way is available in Ada, Eiffel, and Microsoft's C# [Lar01]. Components built around Microsoft's ActiveX technology [JGHJ96] also provide this functionality in the form of `set` and `get` methods that are then transparently mapped to field access in languages such as Visual Basic. In addition, the proposal in reference [RR00] integrates field access into the Java type system to eliminate the accessor methods used to enforce read-only access.

We propose an addition to the C++ language definition that transparently maps plain field access into accessor methods—when such methods are provided; a similar change could also be incorporated into Java when the language is extended to support some form of operator overloading. With the change we propose fields are typically created as public. When implementation or interface changes dictate a different realisation, suitable accessor methods can be written to bind the new implementation to the rest of the system without other code modifications. In addition, object properties that are typically accessed through a *get/set* interface can be made to appear as fields promoting a more consistent expression style.

4 C++ Realisation

The syntax we propose for implementing the language-supported accessor methods is based on overloading the `.` (dot) operator to provide pseudo-fields (or pseudo-data members in the C++ terminology). Pseudo-fields are field-like entities that are not declared as fields, but are read or written through language-supported accessor methods. Currently the `.` operator can not be overloaded; the justification for this is that the operator's second argument is a name (an *id-expression*) and not a value. For this reason the C++ operator overloading syntax needs to be slightly extended with one additional operator consisting of a `.` followed by an *id-expression* (the pseudo-field name):

operator-function-id:

`operator` *operator*
`operator.` *id-expression*

By the above definition the sequence `operator.`*id-expression* is now a legitimate *operator-function-id* that can be overloaded to create accessor methods for pseudo-fields. Following the declaration of such accessor methods the sequence `.`*id-expression* can be legitimately applied to a *postfix-expression* as a shortcut for calling the accessor methods defined with the given *operator-function-id* name.

To provide read access to a given pseudo-field *a* of type *T* a method with signature "*T* `operator.`*a*`()`" needs to be defined; correspondingly for write access to the pseudo-field *a* of type *T* a method with signature "*T* `operator.`*a*`(`*T*`)`" must be defined. To enforce read-only access, the second method is omitted; for write-only access the first method is omitted, while the return type of the second method becomes *void*. For reasons that have to do with the efficient implementation of pseudo-field pointers no other overloading based on these method arguments is allowed. In the form they are given they provide the default behaviour of a real object field.

Following these extensions we can define accessor methods for pseudo-fields as in the following example:

```
// Now defined in polar coordinates
class Point {
public:
// Public since we can define pseudo-fields
    double angle;
    double distance;
```

```
// Pseudo fields for Cartesian implementation
    // x pseudo-field
    double operator.x(double x);    // Set
    double operator.x();            // Get
    // y pseudo-field
    double operator.y(double y);    // Set
    double operator.y();            // Get
}

f()
{
    Point a;
    // Will map to (a).operator.x(0)
    a.x = 0;
    a.y = 0;
    // Will map to (a).operator.x(), ...
    screen.Moveto(a.x, a.y);
    //...
```

The semantics and the implementation of the new accessor methods are determined by performing the equivalent of the following source-to-source transformation:

1. The sequence `a.b` appearing in an lvalue context as `a.b`= *c* is transformed into

   `(a).operator.b(c)`

2. The sequence `a.b` appearing in all other contexts is transformed into

   `(a).operator.b()`

Field access through an object pointer using the dereferencing (`->`) operator is performed by transforming the construct *a*`->`*b* into the equivalent `(*a).b`. If the dereferencing operator is overloaded, the respective function shall be called before the above transformation takes place.

Pointers to fields can be accommodated at the expense of an additional level of memory indirection for reading or writing field values through pointers. Since the same field pointer can be used in both lvalue and other contexts, for a class *C*, a pointer *ap* to a pseudo-field *a* of type *T* must provide information for both accessor functions *aget* (pointer to *T* `operator.a()`) and *aset* (pointer to *T* `operator.a(T)`). In addition, the same pointer should also allow accessing fields that do not have any defined accessor methods. This second requirement can be trivially satisfied by having the compiler internally generate the accessor methods needed for all of the class's N fields. A two-dimensional table `V[`N`][2]` is then generated, with each row containing pointers to the respective field's *aget* and *aset* methods. A pointer to a field is the offset of that field's accessor method pointers within the table. Every time a field is accessed through that pointer as an lvalue, the pointer to *aget* is obtained from the first table column; in all other cases the pointer to *aset* is obtained from the second table column. Note that the field access overhead is only incurred when fields of classes containing overloaded accessor methods are accessed through field pointers.

5 Conclusions

Field accessor methods have become a ubiquitous feature of object-oriented programming. The resulting code bloat and unnatural expression style can be avoided without sacrificing abstraction by overloading the `.` (dot) operator to provide pseudo-fields. The syntax and semantics of such an extension for C++ are relatively straightforward. An implementation technique based on the use of a compile-time generated table can be used to retain the semantics of field access through pointers.

Acknowledgements

I wish to thank Sophia Drossopoulou and Sanjeev Sukumaran for insightful comments on earlier approaches that led towards the current proposal.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Languge*. Addison-Wesley, 1996.

[JGHJ96] Stephen Jakab, Darren Gill, Alex Homer, and Dave Jewell. *Visual Basic 5.0 ActiveX Control Creation*. Microsoft Press, Redmond, Washington, USA, 1996.

[KM96] Andrew Koenig and Barbara Moo. *Ruminations on C++: A Decade of Programming Insight and Experience*. Addison-Wesley, 1996.

[Lar01] Graig Larman. Protected variaton: The importance of being closed. *IEEE Software*, 18(3):89–91, May/June 2001.

[RR00] Eva Rose and Kristoffer Høgsbro Rose. Java access protection through typing. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, pages 136–142. Technical Report 269, Fernuniversität Hagen, 2000. Available online http://www.informatik.fernuni-hagen.de/pi5/publications.html.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.