could be restricted to those for which this promise can be kept. While the mechanisms described in the PT proposal are only tangentially relevant to this issue – the same problem would have arisen in defining a `BStack100` directly – they do widen the impact of this limitation.

A final similarity between customization and PT lies in the extent to which both stretch the C-based separate compilation model up to, and probably past, its limits. While it may be possible to maintain the C model, both proposals would be best implemented under more sophisticated compilation management environments.

5 Acknowledgements

Thanks to Michael Tiemann, Doug Schmidt, Rajendra Raj, and Marshall Cline for commenting on drafts of this paper. This work was facilitated by an equipment grant to the author by Sun Microsystems.

6 References

[1] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. "Interfaces for strongly-typed object-oriented programming," *Proc. OOPSLA*, October 1989.

[2] L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism," *Computing Surveys*, vol. 17, p. 471, 1985.

[3] C. Chambers and D. Ungar. "Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented language," *Proc. SIGPLAN*, June, 1989.

[4] C. Chambers, D. Ungar, and E. Lee. "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes," *Proc. OOPSLA*, October 1989.

[5] M. Cline and D. Lea. *Clearer, safer, and faster C++ through annotations,* Unpublished manuscript, 1989.

[6] W. R. Cook. "A Proposal for making Eiffel type-safe," *Proc. ECOOP*, July, 1989.

[7] J. Guttag. "Abstract data types and the development of data structures," *Communications of the ACM*, vol 20, p576, 1989.

[8] D. Lea. "Some storage management techniques for container classes," *The C++ Report*, January, 1990.

[9] O. Madsen and B. Moller-Pedersen. "Virtual classes: a powerful mechanism in object-oriented programming," *Proc. OOPSLA*, October 1989.

[10] R. Raj and H. Levy. "A compositional model for software reuse," *Proc. ECOOP*, July 1989.

[11] B. Stroustrup. "Parameterized types for C++," *Proc. Usenix C++ Conference*, October 1988.

```
template <class T> T sum(template const Matrix<T>& m)
{
  T s = 0;
  for (int i = 0; i < m.rows(); ++i)
    for (int j = 0; j < m.cols(); ++j)
      s += m.elem(i, j);
  return s;
}
```

This allows the code written for `sum` to be reused for matrices of any subclass of any Matrix class holding any numerical element type.

There may be ways to introduce both customization and parameterization into C++ that more seamlessly integrate the two concepts (as might possibly be done via an analog of BETA's "virtual classes" [9]). However, it is preferable to keep these mechanisms distinct: When you need one, having only the other doesn't help much. Since class templates generate several otherwise subtype-unrelated classes and/or subclasses simultaneously, they cannot be used to implement customization, or vice-versa, even both might rest upon the same internal compiler features.

There is another mesh point between PT and the present proposal. The class template mechanism does not improve one security aspect of homogeneous container classes. Consider the following example:

```
class B            { int x; /* ... */ };
class D : public B { int y; /* ... */ }

template <class T> class Stack100
{
private:
  int  sp;   T data[100];
public:
  void push(const T& a) { data[sp++] = a; }
  T&   top()            { return data[sp-1]; }
  //...
};

void f5()
{
  Stack100<B> s;  D d;  s.push(d);  // ...
}
```

This implementation fails to meet one of the most important specifications of a stack: Given any sensible definition of "equals", and ignoring the lack of error handling in the simplified implementation, it is desirable, for any legal argument `a`, that after `s.push(a)`, `s.top()` should equal `a`. This property cannot hold when `d` is pushed in `f5()`. The problem is solvable via the `exact` qualifier described above. If the member functions could be declared as `void push(const exact T& t)` and `exact T& top()`, then legal arguments

4 Customization and parametric types

Many readers will have noted that customization addresses similar issues, settled via similar implementation mechanisms (based on an internal macro-expander integrated into compilers), as were discussed in Stroustrup's [11] "Parameterized Types" (PT) proposal. While this is true, the two proposals are focused on two very different issues. Customization improves C++ support for emerging uses of inheritance, while PT extends C++ to overcome limitations that are logically distinct from subclassing.

In order to see why both PT and customization are needed, consider the central concern of the PT proposal, improved support for the the definition and use of homogeneous container classes. Homogeneous container classes are structures (e.g., stacks, sets, matrices) that hold collections of other objects (e.g., ints, Complexes, Windows) but all of the exact same type per container class. (In contrast, heterogeneous container classes are those which hold collections of objects with the same base type, but possibly different exact types. Heterogeneous containers are readily implementable in C++, usually via pointer collections, without either PT or customization.) Homogeneous container classes are valuable because they enhance type-safety, because they generally support more efficient implementation than do heterogeneous containers, and because they provide mechanisms for ensuring that groups of objects possess coexistent lifetimes [8].

Customization in itself does not improve support for homogeneous containers because definitions of homogeneous container classes fall outside of the standard subtype/subclass hierarchy. For example, even though an intStack and a WindowStack share high-level specifications and code at some abstract level, they are not related via a common superclass. [4] This stems from the fact that while intStacks have member functions like `void push(const int&)`, WindowStacks have `void push(const Window&)`. Their specific argument types do not conform to each other, or to those of any other possible superclass. As discussed by Cook [6], in order to ensure complete type-safety, subclass virtual function arguments may not be restricted solely to argument types that are more specific than those of their parent classes. (Note that this problem is carefully avoided in section 3 above).

Stroustrup's solution to this problem was to introduce support for class templates, which serve as generators of full-fledged classes. Customization is fully compatible with the class templates in the common case where both are desirable. For example,

```
template <class T> class Matrix
{
  virtual       ~Matrix() {}
  virtual int    rows() const = 0;
  virtual int    cols() const = 0;
  virtual T      elem(int i, int j) const = 0;
  int            size() { return rows() * cols(); }
};
```

[4] Interestingly, in the singly-rooted class hierarchy of Smalltalk and related languages, a heterogeneous ObjectStack defined via root class Object is a *subclass* of all homogeneous Stacks. I know of no object-oriented language that supports easy expression of this fact.

in which constructors may be said to be inheritable (i.e., only via synthesis). Of course, these constructors may still be overridden in specific subclasses. Similar remarks hold for `X::operator = (X&)`.

3.7 Member variables

Consider,

```
class Z
{
private:
  template Matrix  a;
  template Matrix& b;
public:
      Z(template Matrix& p, template Matrix& q) :a(p), b(q) {}
//...
};
```

There are two problems encountered in supporting such custom member variables. The first is the value versus reference issue again: because the exact type of `a` differs across instances of `Z`, `sizeof(Z)` is not compile-time determinable across all `Z`s. This is remediable in the same manner as above, via internal fall-backs to freestore allocation and the internal use of a (fixed-size) pointer to the variably-sized part. Again, doing so results in no real cost to users who must otherwise simulate this effect manually.

The second problem is an inherent limitation of customization. Since customization revolves around classes, not individual objects, the present proposal does not fully enable compilers to hard-wire calls to either Matrix operations on either `a` or `b` inside `Z`'s member or user functions. Without heroic efforts, the customization benefits of `template` are lost; compilers are forced to disregard the specific object-by-object type information, and generate most-general-case dispatch-based code.

Neither of these issues is a compelling reason not to support the construct. However users must be made well-aware of its limitations.

3.8 Completely customized objects

Whenever a compiler can statically determine that *all* uses of an object have been customized (i.e., that no dispatch-based usages occur), then there is no need to allocate a dispatch table pointer ("vptr") for that object. This is only a significant issue for extremely lightweight objects, like instances of a `class Int` (not builtin type `int`). The advantages of programming with "simple" first-class objects without paying undue time or space penalties are highly desirable in many applications, making the corresponding sophisticated and challenging compilation analyses that would be required well worthwhile.

One could refuse to "unify" types in conditionals and related points of uncertainty when dealing with values, in which case this construct would be considered an ambiguity error.

A more aesthetically pleasing approach, that is in keeping with the idea that all differences between values and references should be based solely upon whether or not objects are copy-constructed, is to allow such compile-time uncertainties, and to resolve them at run-time. Doing so has some cost to the compiler. If the decision is delayed until run-time, then the size and initializer of `c` are not statically known, so its space must be allocated and deallocated from the freestore at run-time. However, there is no actual cost to the user. Without this support, if programmers need to create `c` in this fashion, then they must do so manually via pointers, as in

```
void f4()
{
  RowMajorMatrix a(10, 10);
  ColumnMajorMatrix b(10, 10);
  Matrix* c = (rand() % 2)? new RowMajorMatrix(a)
                          : new ColumnMajorMatrix(b);
 //...
 delete c;
}
```

So the the net result is merely added convenience, and one less opportunity to (mis)use pointers, at the expense of compiler complexity. This appears to be a small price to pay for such a gain in utility and conceptual simplicity.

There is also a more pragmatic reason for desiring this strategy. If customization is disabled by a user who is not interested in potential speedups, but only the other semantic properties of the `template` modifier (say, in the case of `operator +()`, above), then a compiler would need to fall back on run-time freestore allocation in order to support this anyway. (It would also need to include function pointers to constructors in vtables, which is not currently typical.)

3.6 Custom constructors

If `template` may modify `this` for member functions, then customized constructors are immediately definable. For example,

```
class DenseMatrix
{
  virtual void  copy(const Matrix&) = 0;
public:
                DenseMatrix(const Matrix& b) template { copy(b); }
//...
};
```

This allows a base class to specify the general form of subclass `X(X&)` constructors, thus generalizing the current rules for automatic constructor synthesis, and clarifying the sense

```
class DenseMatrix
{
  //...
  virtual template DenseMatrix& operator += (const Matrix& b) template
                                { /* ... */ return *this; }
};
```

The second use of `template` in the syntactic position (like `const`) that modifies the type of `this` is necessary here to properly propagate the type of `this` to the return value. Note also that a third use of `template`, to modify argument `b` might be applied here as well.

3.5 Custom values

While all examples so far have involved customization around references or pointers to objects, the entire proposal extends naturally to include customization around values. For example, for a `RowMajorMatrix a;` invoking a call-by-value version of `float sum(template Matrix m)`, via `float s = sum(a);` matches argument `m` as a RowMajorMatrix, created via the RowMajorMatrix `X(X&)` constructor. The most specific applicable `X(X&)` constructor must be chosen for value copies of `template` arguments.

Support for "constructive" functions with customized return types is among the most attractive assets of the entire proposal. For example,

```
template DenseMatrix operator + (const template DenseMatrix& a,
                                 const template Matrix& b)
{
  template DenseMatrix c(a); c += b; return c;
};
```

defines a version of `operator +()` applicable for any two appropriate arguments, without requiring case-by-case definition of all possible combinations of parameters, as would be required without customization. (Of course, in this example, it probably *would* be desirable to override this version for at least some argument pairs.)

There is, however, a significant problem encountered with compiling customized values that is not present for references or pointers. Consider,

```
void f3()
{
  RowMajorMatrix a(10, 10);
  ColumnMajorMatrix b(10, 10);
  template Matrix c = (rand() % 2)? a : b;
  //...
}
```

Here, the exact type of `c` is not known at compile time.

```
float sum(template const DiagonalMatrix& m)
{
  float s = 0;
  for (int i = 0; i < m.rows(); ++i) s += m.elem(i, i);
  return s;
}
```

The `template` here allows customization even if subclasses of `DiagonalMatrix` are introduced.

Existing argument-matching rules may be used in deciding when to call a synthesized version of `sum(DiagonalMatrix&)` rather than of `sum(Matrix&)`. As is true even now, when sufficiently precise type information is lacking, the most general versions of functions must be invoked/synthesized. This has important implications for the authors of specialized top-level or member functions: While such functions may be implemented arbitrarily differently than the base versions, they must preserve the same semantics, since authors cannot guarantee which version(s) will be called in a given program.

3.3 Custom return values

Type monitoring is more difficult for a compiler in the case of customized return values, since the exact return type is not known until *after* a customized function is synthesized. For the simplest example,

```
template Matrix& noop(template Matrix& a) { return a; }

void f2()
{
  RowMajorMatrix a(10,10);
  template Matrix& b = noop(a);
  float s = sum(b);
}
```

The most specific type of `b` is known to be `RowMajorMatrix` only after the specialized version, `noop(RowMajorMatrix&)` is synthesized.

3.4 Inheriting return value types

C++ currently possesses a serious, but removable limitation that often impedes type monitoring. For example, the declaration of `DenseMatrix& DenseMatrix::operator += (const Matrix& b)` above requires all subclass versions of `operator +=()` to return references to objects whose type is merely known to be some subclass of DenseMatrix. Thus any usage of this operator loses valuable type information for its result.

However, there is no convincing reason for this restriction. As discussed by Cook [6], it is entirely type-safe to allow subclass versions to return references to any subclass of DenseMatrix. With the `template` modifier, it becomes possible to express and support this,

In addition to an internal analog of a macro-expander, the major burden placed on a compiler supporting customization is the need for improved type-monitoring in order to ensure that the "best" versions of functions are synthesized and called. Two basic C++ constraints make this easier than it might be otherwise:

- Every object and expression has a most-specific type. For example, if `m` is declared as a RowMajorMatrix, then `m` has types RowMajorMatrix, DenseMatrix, and Matrix, with RowMajorMatrix being the most specific.
- All type-propagation is bottom-up. That is, the most specific types of the arguments to any operator or function fully determine the most specific type of the result.

Using these two constraints, customizing compilers can maintain the most specific types of all objects encountered in a program, across situations in which they are not currently required to do so.

There are many additional usage and implementation considerations. The most important ones are described here. A few details are omitted for the sake of brevity.

3.1 Inabilities to customize

As discussed above, there are cases where insufficient information is present to customize a call at compile time. Among the simplest examples is,

```
void f1()
{
  RowMajorMatrix a(10, 10);
  ColumnMajorMatrix b(10, 10);
  float s = sum( (rand() % 2) ? a : b );
}
```

The most specific type of the conditional is just `Matrix`, so a dispatch-based call must be made. Of course, more heroic intervention is certainly not disallowed: A compiler may, for example, safely convert this expression into `(rand() % 2) ? sum(a) : sum(b)`, thus allowing customization. However, these kinds of evasions are rarely possible in general.

3.2 Overriding automatic customization

It is not at all necessary for the compiler itself to synthesize specialized functions. As is already true in C++, users may override functions with more specific versions themselves. For example,

3 Implementing Customization

The most straightforward way to implement customization is by first introducing a type qualifier for objects that is roughly analogous to the use of `inline` for functions. In fact, simply extending the applicability of `inline` to objects would suffice, except that there are cases where the two usages would conflict. However, as discussed further below, the already-reserved word `template` fills this role nicely, and has the right intuitive meaning.

As a type qualifier, `template` should be legal wherever `const` may be used. It may modify any variable, function parameter (including the implicit `this` parameter to member functions), or function return value designating a value (i.e., distinct object), reference, or pointer. The semantics of `template` are that

- The corresponding given type is not necessarily the most specific run-time type possessed by the declared object;
- The compiler should determine the actual type as the most specific type knowable at compile time from the object's initializer;
- If possible in light of this type information, the compiler should create a special version of the function in which this declaration is embedded, based upon knowledge of the actual type.

For example, redeclaring `sum(template const Matrix& a)` instructs the compiler to construct a special version of `sum` for each actual Matrix subtype for which it is called.

Even disregarding the customization aspects of the `template` modifier, there is ample justification for its introduction. It syntactically distinguishes otherwise different roles of programmer-provided type information in C++. Currently, any base-class *reference* may be used to refer to a single object of any subclass, and thus has its type implicitly fixed upon initialization. A base-class *value* may refer only to a particular class, never a subclass. And a base-class *pointer* may be used ambiguously to denote objects of any subclass, perhaps including objects of different subclasses over time, except when designating elements of an array, in which case a pointer is considered to have an exact type. Under this proposal, `template` may modify all three to clarify and extend the role of type information now associated with references.

To push this point a little further, an argument could be made that an additional type qualifier is required, say, `exact`, that restricts particular pointers and references to denote objects of one exact type only, as do non-template values. In this manner, one could merely say that reference declarations default to non-customized `template`, values to `exact`, and pointers to their current dangerously indeterminate role. [3] There are several other desirable results, especially in terms of enhanced type-safety, that would stem from support of `exact` references and pointers. This issue is touched on later, but is not central to the remainder of the proposal.

[3] Except that it would confusingly overextend their meanings, the keyword choice of `public` versus `private` instead of `template` versus `exact` would be appropriate.

actual arguments supplied to it. This is roughly similar to how call-by-name functions are compiled in some languages.

Both of these approaches have advantages and disadvantages that reflect classic compilation versus execution and time versus space tradeoffs. The dispatch approach is much less space-intensive, and easier to implement, since all versions of a polymorphic function share the same machine code. The customization approach may generate more efficient implementations at the price of potential code explosion.

There is an overriding reason that C++ compilers must support the dispatch model. Dispatching may be used when the exact run-time type of an object is not known internally to the compiler. Customization is impossible in such situations. This includes, especially, C++ programs in which heterogeneous collections of objects, all of the same base type but different exact types, are used as arguments to functions. Such programming practices are typical, but by no means universal in C++. In many applications, it would be considered a design or implementation flaw to have *any* run-time type uncertainty in a C++ program.

Clearly, when the exact types of argument objects are known, customization can remedy the kinds of performance problems seen with Matrix classes. Since particular versions of the function are generated, each can integrate the appropriate inlines, and optimize accordingly. This strategy has the potential of resulting in machine code as fast as obtainable in any language.

Customization, or code synthesis, is not really a new idea in C++. All operations on builtin types (`int`, `float`, etc.) are open-coded, or customized for the sake of efficiency, as are non-virtual inlines. Also, `X(X&)` constructors and assignment operators are automatically synthesized if not otherwise defined by programmers.

It is possible to extend customization to apply to arbitrary classes and functions. Ungar and Chambers [3, 4] discovered that dispatching and customization can happily coexist in compilers for object-oriented languages. Their SELF compiler automatically customizes some code, especially for "lightweight" objects like small integers, partially customizes other functions by hard-coding cases for only the most typical arguments, and uses dispatching elsewhere. As they mention, these compilation strategies are adaptable to other object-oriented languages, including C++.

Given the overall structure of C++, *automatic* customization does not seem to be an attractive option. The need for customization is primarily a pragmatic concern of the intended *applications* of classes and their supporting functions, not the classes themselves, and certainly not the compiler per se. Many users are perfectly content with fully dispatched functions, especially for prototyping, debugging, and other non-time-critical use. More typically, customization should be applied selectively. For example, speedups via customization might be of overriding importance in operations that multiply matrices, but not in routines that print out their contents. This is the same pragmatics problem underlying C++ support for inlining. Class users ought to have at least as much to say about whether functions are inlined as do class authors. C++ compilers need to make such decisions easier to implement than requiring users to edit source code. Therefore, if customization is supported in C++, compilers should provide means for end-users to control the degree that both customization and inlining are exploited in particular programs. The logistics of doing so are neither simple nor impossible.

determine that all accesses within `sum` are legal, and thus optimize the checks away completely. For such reasons, this code is likely to execute around ten times more slowly on most machines than it would without the virtual calls. (Simple informal tests confirm the order of magnitude of this estimate.)

Array processing code represents an extreme, but very important case in which allowing a compiler to perform inline substitution of a few critical "lightweight" member functions, and then to procedurally integrate and optimize the resulting code represents the difference between merely attractive class designs and usable packages. The problem is not that these calls are virtual, or even that there is procedure call overhead at all. It is the failure to procedurally integrate code that most hampers performance: In the ideal case, the executable code produced for `sum` should be so well integrated that few machine operations could be said to "belong" to `sum` or the invoked member functions per se. [2] The failure of current C++ compilers to make powerful optimization strategies even conceivable is surely unacceptable to many potential users of classes like Matrix that are targeted for use in high-performance applications. Without accommodations, authors of such packages must sacrifice object-oriented design methodologies in favor of less reusable, reliable, and coherent ad hoc coding strategies.

The remainder of this paper proposes one solution to this mismatch between positive design benefits and negative implementation costs of ABCs, and inheritance in general.

2 Compiling polymorphic functions

Any class member function or top-level client function that *invokes* a virtual class function is polymorphic (or more precisely, "subtype polymorphic" [2]). Any such function, like `sum(const Matrix&)`, may be thought of as a stand-in for an unknown number of specific functions `sum(const RowMajorMatrix&)`, `sum(const TriDiagonalMatrix&)`, and so on.

There are two approaches for implementing functions that share the exact same "high-level" C++ source code.

In the standard "dispatch" approach normally employed by C++ compilers, all versions share the same actual machine code, but produce different run-time consequences because the indirect calls generated from the member function dispatch tables ("vtables") invoke different implementation code for different objects passed in as arguments (including `this` as an implicit argument for member functions).

A second approach is to synthesize, or *customize* as many different specific versions of a function as are actually needed in a given user program. Each customized version can then hard-wire code specifically geared for the given arguments by directly calling and/or procedurally integrating invoked member functions, rather than dispatching them. Thus, even though such polymorphic functions share high-level source code, they are implemented via different machine code. A compiler may implement this strategy via analogs of macro-expansion: The high-level source code may be recompiled as a new function in light of the

[2] The C underpinnings of C++ present several additional well-known obstacles (e.g., uncontrolled aliasing) for compilers attempting to perform powerful Fortran-like optimization strategies for numerical and array processing. These problems also require careful attention, but are not considered here.

This last point is the most important one for a large number of applications. Consider the need for an integrated linear algebra package based in part on class Matrix. In order to be maximally useful, one would need a large number of Matrix subclasses, including RowMajorMatrix, ColumnMajorMatrix, TridiagonalMatrix, RowSparseMatrix, and dozens of others, as well as additional intermediate abstract subclasses, to support the range of algorithms that have been developed for dealing with such special (in terms of either representation details or element structure) Matrix types. For example,

```
class DenseMatrix : public Matrix
{
public:
  float&               operator () (int i, int j) = 0;
  virtual DenseMatrix& operator += (const Matrix& b)
                       { /* ... */ return *this; }
};

class RowMajorMatrix: public DenseMatrix
{
private:
  int r; int c; float* d;
public:
         RowMajorMatrix(int m, int n);
         RowMajorMatrix(const RowMajorMatrix& b);
        ~RowMajorMatrix();
  int    rows() const { return r; }
  int    cols() const { return c; }
  float& operator () (int i, int j)
         {
           if (i < 0 || i >= rows() || j < 0 || j >= cols())
             abort(); // or, someday, raise an exception
           return d[i * cols() + j];
         }
  float  elem(int i, int j) const { return (*this)(i, j); }
};
```

There is a great deal of design elegance to be had in this approach. Unfortunately, there is also a great deal of inefficiency in the corresponding code generated by current C++ compilers.

Consider a call to `sum(a)` for a `RowMajorMatrix a`. Because all of the invoked operations are virtual, actual computation is swamped by virtual function calls. Worse, because the implementation code lying inside the virtual functions is opaque to the C++ compiler, none of the standard optimization strategies (like strength reduction, common subexpression elimination, and even automatic vectorization) that can be applied to array processing code inside good optimizing compilers are applicable. Perhaps the easiest (but *not* the most important) example of the kinds of problems encountered is illustrated in the index checking occurring inside `RowMajorMatrix::operator()`, where the checks must be performed on each `elem` call since the compiler cannot "see" the bounds checking code and statically

```
float sum(const Matrix& m)
{
  float s = 0;
  for (int i = 0; i < m.rows(); ++i)
    for (int j = 0; j < m.cols(); ++j)
      s += m.elem(i, j);
  return s;
}
```

ABCs are closely related to the concept of "Abstract Data Types", or ADTs (see [7]), in that they completely hide from clients of the class all representation and implementation details (as defined in any given implementation subclass). They are not completely isomorphic to ADTs, since there is no C++ support for semantic specification of such classes (but see [5]), and since ABCs are declared and used in an object-oriented, rather than algebraic fashion.

Support for ABCs completes the possibilities implicit in the notion that protocol inheritance (also known as "specification inheritance", "interface inheritance", or just "subtyping") is a logically distinct concept from implementation/representation inheritance. Discussions of the importance of separating these two roles of inheritance may be found in [1, 10]. In C++, one can now obtain protocol-only inheritance via ABCs, implementation-only inheritance via "private" subclassing or simple composition, or both, via non-abstract "public" subclassing.

The availability of ABCs encourages a style of programming in which every different set of protocols gets its own ABC, and every implementation strategy gets its own subclass. Classes that define both protocol and implementation are needed only when specifications are narrow enough to dictate particular representations. Often, specifications only partially restrict implementations, leading to partially abstract classes. (For example, it might be defensible to force all matrices to possess member variables `int r, c;` to hold the numbers of rows and columns, and to rewrite `rows()` and `cols()` accordingly.)

C++ support for ABCs considerably enhances creation of both domain-independent and domain-specific reusable packages and libraries:

- ABCs support the design of class hierarchies without forcing premature commitments to implementation details.
- ABCs improve support for guaranteed "plug-compatible" implementations. Clients may replace one implementation subclass with another, without any other program changes.
- ABCs allow simpler integration of classes and libraries from different sources, by allowing integrators to subsume two different implementations under a common ABC.
- ABCs improve prospects for standardization efforts, since they address only specification, not implementation issues.
- ABCs are vital for development of packages in which subclasses with common protocols require vastly different implementations.

Customization in C++

Douglas Lea
SUNY at Oswego / NY CASE Center / GNU Project
(dl@oswego.edu)

Abstract

Extensions to C++ are proposed that would enable the directed synthesis of code as an alternative to dispatch-based code generation. The strategies are similar to those used in the SELF compiler, which uses customization techniques to impressive effect. The language extensions required to support customization also enhance support for other uses of inheritance in C++, and are useful in conjunction with proposed additions of parametric types.

1 Introduction

One of the smallest, but most important features introduced in C++ 2.0 is improved support for abstract base classes (ABCs) via the use of "pure virtual" member functions. For example, [1]

```
class Matrix
{
public:
  virtual       ~Matrix() {}
  virtual int    rows() const = 0;
  virtual int    cols() const = 0;
  virtual float  elem(int i, int j) const = 0;
  int            size() { return rows() * cols(); }
};
```

This declares a Matrix class in terms of its protocol or interface (i.e., the signatures of member functions), but not in terms of member function implementations or the data representations upon which they operate. Subclasses of Matrix are required to define the implementations of these functions themselves. As is often useful, this ABC also declares a non-pure member function (`size`) that operates exclusively via the pure virtual functions. Of course, it is also possible to declare ordinary "client" functions that operate on any Matrix via these member functions, regardless of implementation. For example,

[1] I will use Matrix-based examples throughout the paper, since they serve well to illustrate various points. The examples are mostly realistic, but substantially stripped-down from those suitable for serious application. Just to stave off controversy, I will note a few of the most egregious simplifications. But in all other respects, this paper has little to do with Matrix classes per se.