Runtime Polymorphic Generic Programming—Mixing Objects and Concepts in ConceptC++

Mat Marcus[1], Jaakko Järvi[2], and Sean Parent[1]

[1] Adobe Systems Inc. {mmarcus|sparent}@adobe.com
[2] Texas A&M University jarvi@cs.tamu.edu

Abstract. A long-held goal of software engineering has been the ability to treat software libraries as reusbale components that can be composed with program-specific code to produce applications. The object-oriented programming paradigm offers mechanisms to write libraries that are open for extension, but it tends to impose intrusive interface requirements on the types that will be supplied to the library. The generic programming paradigm has seen much success in C++, partly due to the fact that libraries remain open to extension without imposing the need to intrusively inherit from particular abstract base classes. However, the static polymorphism that is a staple of programming with templates and overloads in C++, limits generic programming's applicability in application domains where more dynamic polymorphism is required. In this paper we present the **poly<>** library, a part of Adobe System's open source library ASL, that combines the object-oriented and generic programming paradigms to provide non-intrusive, transparent, value-based, runtime-polymorphism. Usage, impact on design, and implementation techniques are discussed.

1 Introduction

Successful development of robust large scale-software applications depends on upon the ability to combine application-specific functionality with independently developed library modules from a variety of sources, with a reasonable amount of application-specific glue code. To support this activity, modules must remain open for extension but closed for modification [18]. Object-oriented programming and generic programming are the two main paradigms available for creating such modules in C++.

In object-oriented programming, libraries typically specify that the types supplied to the library must be derived from a common abstract base class, providing implementations for a collection of pure virtual functions. The library knows only about the abstract base class interface, but can be "extended" to work with new user types derived from the abstract interface. That is, variability is achieved through differing implementations of the virtual functions in the derived classes. This is how object-oriented programming supports modules that are closed for modification, yet remain open for extension. One strength of this paradigm is its support for varying the types supplied to a module at runtime. Composability of modules is limited, however, since independently produced modules generally do not agree on common abstract interfaces from which supplied types must inherit.

The paradigm of generic programming, pioneered by Stepanov, Musser and their collaborators [16, 19], is based on the principle of decomposing software into efficient components which make only minimal assumptions about other components, allowing maximum flexibility in composition [7]. C++ libraries developed following the generic programming paradigm typically rely on templates for the parametric and ad-hoc polymorphism they offer. Composability is enhanced as use of a library does not require inheriting from a particular abstract interface. Interfaces of library components are specified using *concepts*—collections of requirements analogous to, say, Haskell type classes [28]. The key difference to abstract base classes and inheritance is that a type can be made to satisfy the constraints of a concept retroactively, independently of the definition of the type. Also, generic programming strives to make algorithms fully generic, while remaining as efficient as non-generic "hand-written" algorithms. Such an approach is not possible when the cost of any customization is a virtual function call.

Many successful software libraries have been designed and implemented following the generic programming paradigm [1,2,6,8,22,24,25,27]. We summarize the paradigm and supporting extensions to C++ in Section 2.

In combining elements of object-oriented programming with those of generic programming, we take generic programming as the starting point, retaining its central ideas. In particular, generic programming is built upon the notion of *value types* that are assignable, copy constructible, etc. [7] The behavior expected from value types reflects that of C++'s built-in types, like **int**, **double**, and so forth. This generally assumes that types encapsulate their memory and resource management into their constructors, copy-constructors, assignment operators, and destructors, so that objects can be copied, and passed as parameters by copy, etc., without worrying about references to their resources becoming aliased or becoming dangling. Value types simplify local reasoning about programs. Explicitly managing objects on the heap and using *pass-by-reference* as the parameter passing mode makes for complex object ownership management (and object lifetime management in languages that are not garbage collected). Instead, explicitly visible mechanisms—thin wrapper types like **reference_wrapper** in the (draft) C++ standard library [5]—are used when sharing is desired.

We would like to simultaneously enjoy the benefits of both the object-oriented and the generic programming paradigms: easily composable modules that can support dynamic polymorphism. Value semantics and dynamic polymorphism of abstract base classes are, however, not easily mixed. Indeed, libraries written following the generic programming paradigm tend to integrate with object-oriented programming only in trivial ways. Established library idioms of generic programming provide no way to use concepts to define rich interfaces that simultaneously supporting static and dynamic polymorphism. In what follows we present, in stages, the **poly<>** library supporting construction of applications from composable generic components and use of dynamic polymorphism at the components' generic interfaces. We combine the object-oriented and generic programming paradigms to provide non-intrusive, transparent, value-based, runtime-polymorphism. The **poly<>** library is part of Adobe System's open source *Adobe Source Library* (ASL) [1].

We use an example from the domain of graphical user interfaces (GUI) to motivate and illustrate our techniques. Adobe's *Eve*, a component of ASL, is a layout engine that can be used to calculate positions for GUI widgets in a dialog. Our experimental version of Eve (relying on extensions to C++ described below) defines its interface in terms of concepts. In particular, we define the **Placeable** concept that requires a **measure** operation for obtaining the *extents* of a widget, and a **place** operation for effecting the placement of a widget into certain position in a window. Our layout engine can operate on different kinds and types of widgets, from the same or from different GUI frameworks, as long as the concrete widget types are adapted to conform to the **Placeable** interface. In some applications, the widget types might be known statically. In other applications, the layout can consist of varying types of widgets. Ideally, the engine itself does not need to concern itself with this distinction. Depending on the needs of the client of the layout engine, the same engine template can be instantiated either with a single specific widget type, or with a wrapper type that allows the engine to operate on an open ended set of widget types. Below, instantiating the engine with **HIViewRef** gives an engine statically bound to operate on Mac Carbon [3] widgets; we omit the definitions that make **HIViewRef** model **Placeable**. The second instantiation uses a **poly** template, explained in Section 3.4, to create a dynamically polymorphic value type, akin to an *existential type* [17].

```
template <Placeable P> struct layout_engine { ... }

layout_engine<HIViewRef> le;
layout_engine<poly<placeable>> le;
```

The code in the application assembling the components determines whether to use static or dynamic polymorphism. The library does not impose this decision on its clients.

Most of the techniques described here are part of ASL, some experimental features of it. The machinery presented is an extension of the ideas described in [21]. The techniques require the addition of some boilerplate code, but the crucial point is that this code is external to both to the algorithm library (corresponding to the layout engine in our example) and to the library types provided to it (the Mac OS X Carbon library in our example). We will illustrate how such transparent adaptation is possible in the sections which follow. Section 3.2 indicates where some intrusion to make libraries **poly<>**-aware can be used to gain more control when combining static and dynamic polymorphism.

We present our techniques using C++ extended with “concepts” [10], here referred to as *ConceptC++*. ConceptC++ adds constrained templates to C++, and is a likely extension to the next revision of standard C++. In this article, C++ refers to the language as specified in its current standard [14].

2 Background: Generic Programming and ConceptC++

Generic programming is a systematic approach to designing and organizing software. It focuses on finding the most general (or abstract) formulations of algorithms together with their efficient implementations [16]. Integral to the paradigm is the organization of constraints on type parameters into *concepts*, which describe the commonly occurring abstractions in a particular domain. The archetypal example of the results of the generic programming approach is the Standard Template Library (STL) [27] and its documentation [4,26].

A brief description of the established generic programming terminology is as follows: A *concept* is a collection of requirements on a type (types). We say that a type (or a tuple of types) *models* a concept whenever it satisfies all the requirements of that concept. A concept is said to *refine* another concept if its set of requirements includes all requirements of the other concept. The kinds of requirements in a concept are *valid expressions* (possibly expressed as function signatures), *associated types*, *semantic constraints*, and *complexity guarantees*. Function signatures specify the operations that must be defined for the modeling type(s). The associated types of a concept specify mappings from the modeling type(s) to other collaborating types (such as the mapping from a container to the type of its elements).

C++ templates are unconstrained. Generic C++ libraries therefore describe concepts only in the library documentation, and the constraints on type parameters of generic algorithms as part of algorithms' documentation. ConceptC++ makes concept descriptions and type parameter constraints known to the compiler, to enable modular type-checking of templates. Type-checking in ConceptC++ concerns only with the syntactic requirements of concepts.

The central language construct of ConceptC++ is **concept**, collecting a set of requirements on a type (or types). For example, the following concept **LessThanComparable** requires that the "less than" operator, **<**, is defined for any type that models the concept:

```
concept LessThanComparable<typename T> {
  bool operator<(T, T);
}
```

C++ allows defining the operator **<** as a member or as a non-member function; for some types it comes built-in. Concepts are not concerned with how the operator has been defined: any means is adequate to satisfy the requirement.

ConceptC++ requires an explicit declaration to establish that a particular type (or a parametrized class of types) is a model of a concept. These declarations are called *concept maps*. The following two declarations state that the **int** and **complex<double>** types are models of the **LessThanComparable** concept:

```
concept_map LessThanComparable<int> { }

concept_map LessThanComparable<complex<double> > {
  bool operator<(complex<double> a, complex<double> b) { return abs(a) < abs(b); }
}
```

The two definitions differ in how **LessThanComparable**'s requirements are satisfied. For **int**, the built-in **<** operator for integers provides the required implementation; for **complex<double>**, the definition in the body of the concept map is used. For a concept map to type check, for each of its requirements an implementation must be present in the concept map's body, or found in the scope where the concept map is defined.

Concept maps can be templates. In the following, all instances of the standard **pair** template are declared to be **LessThanComparable**:

```
template <typename T, typename U>
requires LessThanComparable<T>, LessThanComparable<U>
```

```
concept_map LessThanComparable<pair<T, U> > {
  bool operator<(const pair<T, U>& a, const pair<T, U>& b) {
    return a.first < b.first || (!(b.first < a.first) && a.second < b.second);
  }
}
```

The simple generic function **min_element** below uses the **LessThanComparable** concept as a constraint. Constraints on type parameters are stated in the **requires** clauses of templates. Bodies of templates are type-checked assuming the constraints in the **requires** clauses are satisfied. Correspondingly, at template instantiation, the type-checker checks that the template arguments satisfy the constraints in the **requires** clauses.

```
template <typename Iter>
requires ForwardIterator<Iter>, LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last) {
  Iter best = first;
  while (first != last) { if (∗first < ∗best) best = first; ++first; }
  return best;
}
```

The **ForwardIterator** concept that appears in the constraints of **min_element** is shown in Figure 1. This concept provides basic iteration capabilities. The dereferencing operator $*$ gives the value that an iterator refers to. The **++** operator advances the iterator to the next element. Equality comparison is used to decide when the end of the sequence is reached. Requirements for the operators **==** and **!=** are not stated directly in the body of **ForwardIterator**, but are obtained through *refinement* of another concept **EqualityComparable** (not shown). Syntax of refinement is that of inheritance between classes. The associated type **value_type** denotes the type of the value that the iterator refers to. A **requires** clause in the body of a concept can be used to place additional constraints on the parameters or associated types of a concept. Here, **value_type** must model **CopyConstructible**, which is one of the (draft) standard concepts and has its expected meaning. Examples of models of **ForwardIterator** include all pointer types and the iterator types of standard containers.

```
concept ForwardIterator<typename Iter> : EqualityComparable<Iter>, CopyConstructible<Iter> {
  typename value_type;
  requires CopyConstructible<value_type>;

  value_type& operator∗(Iter);
  Iter& operator++(Iter&);
  Iter operator++(Iter&, int);
}
```

Fig. 1. The **ForwardIterator** concept (simplified from the one in the STL).

The **min_element** algorithm works for any sequence of values defined as a pair of iterators, as long as the iterator type is a model of the **ForwardIterator** concept and the iterator's value type is a model of **LessThanComparable**. Assuming the concept map definition for **complex<double>** we showed above, the following call satisfies the constraints of **min_element**. The invocation of operator **<** in the body of **min_element** then calls the definition given in the concept map.

```
vector<complex<double> > cd;
// fill cd with values
complex<double> smallest = min_element(cd.begin(), cd.end());
```

A concept definition can be preceded with the keyword **auto** to signify that no explicit concept map is necessary to establish a *models* relation between a type and a concept—structural conformance to the requirements suffices. Simple concepts with only a few requirements are typically defined as **auto**. Concept maps can be written explicitly for **auto** concepts as well.

ConceptC++ provides a syntactic shortcut for succinctly expressing constraints directly in the template parameter list: instead of the keyword **typename**, a concept name then precedes a template parameter. The following function signature uses the shortcut to constrain **Iter** to be a model of **ForwardIterator**:

```
template <ForwardIterator Iter> requires LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last);
```

Same shortcut works for associated types. The declaration **CopyConstructible value_type;** can replace the following lines in Figure 1:

```
typename value_type;
requires CopyConstructible<value_type>;
```

For a detailed description of the language features of ConceptC++, see [10, 11].

3 Development

In these sections, we present the **poly** library, comprising machinery for creating non-intrusive runtime-polymorphic value wrappers, parametrized by concept. We discuss several use cases, including non-intrusive heterogeneous containers and the retroactive addition of runtime polymorphism to a templated interface of a generic library. Except for code to fully leverage concept refinement, discussed in Section 3.2, the machinery does not intrude upon the generic library or upon the types to be used as inputs to the library routines.

Programmers often are faced with the need to maintain heterogeneous collections of related objects. In the object-oriented paradigm, a container of pointers to abstract base classes is the preferred implementation technique. However this requires that the objects to be stored must inherit from a particular base class. We maintain that the choice of whether an object will be used polymorphically—with a particular fixed interface specified by a base class—is not one which should be made at class definition time.

We expand on the generic layout engine example introduced in Section 1 to illustrate our use cases. Figure 2 shows the definition of the engine, as a class template that can be instantiated with any type that models the **Placeable** concept:

```
concept Placeable <typename T> : Copyable<T> {
  void measure(T& t, extents_t& result);
  void place(T& t, const place_data_t& place_data);
}
```

The **append** member function adds widgets to the a layout "problem". The **solve** member function uses the **measure** operation from **Placeable** to query the extents of each widget, calculates a solution satisfying the layout constraints (not shown), and finally invokes the **Placeable**'s **place** operation to inform each widget of its calculated location. The three vectors **placeable_m**, **extents_m**, **place_data** hold, respectively, the widgets to be placed, their extents, and ultimately the positioning information for the widgets, as computed by the layout engine.

```
template <Placeable P>
struct layout_engine {
  void append(P placeable) { placeables_m.push_back(placeable); }

  void solve() {
    extents_m.resize(placeables_m.size());

    for(int i = 0; i != placeables_m.size(); ++i) measure(placeables_m[i], extents_m[i]);

    // "solve" layout constraints and update place_data_m

    for(int i = 0; i != placeables_m.size(); ++i) place(placeables_m[i], place_data_m[i]);
  }

  vector<extents_t> extents_m;
  vector<P> placeables_m;
  vector<place_data_t> place_data_m;
}
```

Fig. 2. A simplified layout engine modeled after Eve.

The layout engine is a generic collection where the type of the values stored in the collection is parametrized. We can instantiate the engine in a straightforward manner to accept widgets of a particular type, e.g. as **layout_engine<HIViewRef>**, provided that a suitable concept map exists. In the code below, we omit the definitions of the **measure** and **place** functions that contain the Mac Carbon specific code for measuring of extents and placing of widgets:

```
void measure(HIViewRef& t, extents_t& result) { ... };
void place(HIViewRef& t, const place_data_t& place_data) { ... };

concept_map Placeable<HIViewRef> {}
```

In this case, the layout engine can be viewed as a homogeneous container of **HIViewRef**s. In some cases, however, it is desirable to use the engine with more than one (unrelated type). That is, we sometimes wish to view the engine as a *heterogeneous* container. To support this in a non-intrusive manner means that the code of Figure 2 and the **Placeable** concept must remain unchanged.

To use a template interface in a runtime polymorphic manner, an abstract class (and some boilerplate) must be created for each concept in the interface—no further adaptation is required at the individual class or function level. We introduce the **poly** library machinery in stages. We begin with a concrete type modeling the **Placeable** concept, then refactor it through a series of stages until all of our design goals are achieved. Finally, Section 3.4 presents an outline of the general purpose **poly<>** library that seeks to encapsulate the boilerplate code required when creating non-intrusive, runtime polymorphic, value classes. In its final form then, the instantiation of the heterogeneous layout engine is written as **layout_engine<poly<placeable>>**. The **placeable** type defines a member function for each function requirement of the **Placeable** concept, the **poly** template integrates this definition with the supporting machinery.

3.1 Implementing polymorphic value types

The ability to copy values is a prevalent requirement for type parameters of generic functions. specifically, this means that the types have a public copy constructor and an assignment operator. We capture these requirements in the **Copyable** concept:

```
concept Copyable <typename T> : std::CopyConstructible<T>, std::Assignable<T> {}
```

Users of the layout engine must instantiate it with a type that is a model of **Placeable**—and thus **Copyable** as well, since **Placeable** refines **Copyable**. We demonstrate with a minimal **Placeable** type, **Widget**, that we carry over to further examples. Here, the task of the concept map is to map non-member functions to member functions.

```
struct Widget {
  Widget& operator=(const Widget&) { ... }
  Widget(const Widget&) { ... }
  void measure(extents_t& result) { ... }
  void place(const place_data_t& place_data) { ... }
  ...
};

concept_map Placeable<Widget> {
  void measure(Widget& x, extents_t& result) { x.measure(result); }
  void place(Widget& x, const place_data_t& place_data) { x.place(place_data); }
}
```

With the concept map for **Widget**s in place, the instantiation **layout_engine<Widget>** is possible. *Static polymorphism* is the name of the mechanism that allows users to instantiate the engine with **Widget** or with **HIViewRef**. The widget type accepted by the layout engine's **append** function is fixed (either to **HIViewRef** or **Widget**), and cannot vary at runtime.

In C++, *runtime polymorphism* is achieved through base class pointers or references, not values. A possible approach to allow the layout engine to work with both **HIViewRef**s and **Widget**s at the same time would to define an abstract class corresponding to the **Placeable** concept, and make pointers to that base class models of **Placeable**. In general, however, pointers go hand in hand with heap allocation, lifetime management, and shared reference issues. We can instead retain value semantics by separating concrete types into several components, following the Bridge pattern [9]. We begin by replacing the **Widget** class with the "handle" and "body" classes in Figure 3.

```
struct PlaceableInterface {
  virtual void measure(extents_t& result) = 0;
  virtual void place(const place_data_t& place_data) = 0;
  virtual void assign(const PlaceableInterface& x) = 0;
  virtual PlaceableInterface* clone() const = 0;
  virtual ~PlaceableInterface() {}
};

struct WidgetImplementation : PlaceableInterface {
  void measure(extents_t& result) { ... }
  void place(const place_data_t& place_data) { ... }

  void assign(const PlaceableInterface& x) { ... }
  PlaceableInterface* clone() const { ... }

  // data members
};

struct PlaceableHandle {
  void measure(extents_t& result) { interface_m->measure(result); }
  void place(const place_data_t& place_data) { interface_m->place(place_data); }

  PlaceableHandle(const PlaceableHandle& x) : interface_m(x.interface_m->clone()) {}
  PlaceableHandle(const WidgetImplementation& x)
    : interface_m(new WidgetImplementation(x)) {}
  PlaceableHandle& operator=(const PlaceableHandle& x)
    { interface_m->assign(*x.interface_m); return *this; }

  std::scoped_ptr<PlaceableInterface> interface_m;
};

concept_map Placeable<PlaceableHandle> {
  void measure(PlaceableHandle& x, extents_t& result) { x.measure(result); }
  void place(PlaceableHandle& x, const place_data_t& place_data) { x.place(place_data); }
}
```

Fig. 3. The **WidgetImplementation**, **PlaceableInterface**, and **PlaceableHandle** classes.

The body class consists of two parts: **PlaceableInterface** and **WidgetImplementation**. The former is the runtime polymorphism layer, the latter encapsulates code specific to the concrete widget type. The handle class **PlaceableHandle** behaves like a **Placeable** value, supporting copy construction and assignment. It delegates all calls to its member functions to **PlaceableInterface**. Copying and assignment of **PlaceableInterface** are implemented with **clone** and **assign** member functions.

With this use of the Bridge pattern we have achieved value-based runtime polymorphism, but with the cost of intruding upon the **Widget** type. We can rectify this with the help of templates. Instead of reimplementing **Widget**, we define a generic wrapper (**PlaceableImplementation**) for all **Placeable** widget types. We also change the handle's constructor to accept any **Placeable** type, and wrap it with **PlaceableImplementation**. The new configuration is shown in Figure 4.

```
template <Placeable T>
struct PlaceableImplementation : PlaceableInterface {
  void measure(extents_t& result) { Placeable<T>::measure(placeable_m, result); }
  void place(const place_data_t& place_data)
    { Placeable<T>::place(placeable_m, place_data); }

  void assign(const PlaceableInterface& x);

  PlaceableImplementation(const T& x) : placeable_m(x) {}
  PlaceableImplementation* clone() const
    { return new PlaceableImplementation<T>(placeable_m); }

  T placeable_m;
};

struct PlaceableHandle {
  ...
  template <typename T>
  PlaceableHandle(const T& x) : interface_m(new PlaceableImplementation<T>(x)) {}
  ...
};
```

Fig. 4. A generic "handle-body" class bundle for **Placeable** types. The only change to **PlaceableHandle** class from Figure 3 is making the constructor a template. The **PlaceableImplementation** template implements its member functions by delegating to an arbitrary wrapped concrete **Placeable** type. The definitions of **PlaceableInterface** and the concept map for **PlaceableHandle** are unchanged, and not shown.

With concept maps, any type (with suitable capabilities) can be made to model **Placeable**, and **PlaceableImplementation** can wrap any **Placeable** type. Adaptation of widget types first to the layout engine's interface and then to be usable as a run-time polymorphic value in that interface, is thus completely transparent and non-intrusive. The following code illustrates:

```
layout_engine<PlaceableHandle> le;
HWND x; Widget w;
  ...
le.append(x); le.append(w);
// equivalent, via implicit construction, to le(PlaceableHandle(x)), etc.
```

3.2 Refinement

So far we have achieved a form of non-intrusive value-based runtime-polymorphism: we can wrap different placeable types inside of an external **PlaceableHandle** template. In this way a single instance of a generic algorithm can support types unknown at compile time. This section extends our solution with the dynamic counterparts of the concept *refinement* relation and with overloading based on concepts.

Some widgets can only provide accurate information on their extents with a two-pass measurement. Examples include text boxes where the justified height of the text depends on the width. To support multi-pass measurement, we define the following concept as a refinement of **Placeable**:

```
concept PlaceableTwopass <typename T> : Placeable<T> {
  void measure_vertical(T& t, extents_t& horizontal_result, const place_data_t& place_data);
}
```

ConceptC++ supports "concept-based" overloading, where the entailment relation between constraints is taken into consideration when selecting the best overload [15]. For example, we show two overloads for the **adjust_measurement** function below, first for **Placeable** types, the second for types that additionally model **PlaceableTwopass**. These functions are called from the layout engine's **solve** routine. In our simplified example, assume that **adjust_measurement** is empty for **Placeable** types, and that it obtains and exploits the second pass vertical measurement for **PlaceableTwopass** types. Further details of the implementations of the functions are not important here.

```
template <Placeable T>
void adjust_measurement(const T& t, extents_t& e, const place_data_t& m) {}

template <PlaceableTwopass T>
void adjust_measurement(const T& t, extents_t& e, const place_data_t& m) {
    ...
  measure_vertical(t, e, m);
    ...
}
```

When all types are statically known, the matching function with the most specific constraints is selected. Thus far, we have only defined a handle class for **Placeable** types, and if we wrap a widget supporting two-pass placement in a **Placeable** handle, the overloads for the more refined concept will not apply. If the bundle of classes in Figure 4 is extended according to the concept refinement relation, we can select a more refined handle class to wrap a client type.

Figure 5 shows the refined class bundle. The **PlaceableTwopassInterface** class inherits from the **PlaceableInterface** and adds the new virtual function **measure_vertical**. The **PlaceableTwopassImplementation** provides implementations for the pure virtual functions of **PlaceableTwopassInterface**. These implementations again delegate to the refined concept. Some repetition is necessary here: rather than only implementing the new member functions of **PlaceableTwopassInterface**, it is necessary to implement those, and all member functions that **PlaceableTwopassInterface** obtains via inheritance. One reason why the repetition is necessary is that it is not possible to parametrize over concept names in ConceptC++. The role of **PlaceableTwopassHandle** is exactly analogous to that of **PlaceableHandle**, and of course a concept map is necessary to make the new handle type a model of **PlaceableTwopass**.

This solution is sufficient if we can use either one of the handles for all widgets in a given layout task. If, however, some widgets do not support two-pass layout, the

```
struct PlaceableTwopassInterface : PlaceableInterface {
  virtual PlaceableTwopassInterface* clone() const = 0;
  virtual void measure_vertical(extents_t& horizontal_result,
                                const place_data_t& place_data) = 0;
};

template <typename T>
struct PlaceableTwopassImplementation : PlaceableTwopassInterface {
  void measure(extents_t& result)
    { PlaceableTwopass<T>::measure(placeable_twopass_m, result); }
  void place(const place_data_t& place_data)
    { PlaceableTwopass<T>::place(placeable_twopass_m, place_data); }
  void measure_vertical(extents_t& horizontal_result, const place_data_t& place_data)
    { PlaceableTwopass<T>::measure_vertical(placeable_twopass_m,
                                            horizontal_result, place_data); }

  void assign(const PlaceableInterface& x);
  PlaceableTwopassImplementation* clone() const;
  PlaceableTwopassImplementation(const T& x) : placeable_twopass_m(x) {}

  T placeable_twopass_m;
};

struct PlaceableTwopassHandle {
  void measure(extents_t& result) { interface_m->measure(result); }
  void place(const place_data_t& place_data)
    { interface_m->place(place_data); }
  void measure_vertical(extents_t& horizontal_result,
                        const place_data_t& place_data)
    { interface_m->measure_vertical(horizontal_result, place_data); }

  template <typename T>
  PlaceableTwopassHandle(const T& x)
    : interface_m(new PlaceableTwopassImplementation<T>(x)) {}

  PlaceableTwopassHandle& operator=(const PlaceableTwopassHandle& x)
    { interface_m.reset(x.interface_m->clone()); return *this; }
  PlaceableTwopassHandle(const PlaceableTwopassHandle& x)
    : interface_m(x.interface_m->clone()) {}

  std::scoped_ptr<PlaceableTwopassInterface> interface_m;
};

concept_map PlaceableTwopass<PlaceableTwopassHandle> {
  void measure(PlaceableTwopassHandle& x, extents_t& result);
  void place(PlaceableTwopassHandle& x, const place_data_t& place_data);
  void measure_vertical(PlaceableTwopassHandle& x, extents_t& horizontal_result,
                        const place_data_t& place_data);
}
```

Fig. 5. Refining the “handle-body” class bundle.

layout engine must be instantiated with **PlaceableHandle**, and no overloads for two-pass layout will be found. To summarize, we have presented a non-intrusive way to make a “static” library interface support run-time polymorphism, in the sense that multiple widget types can be supported simultaneously. The static type information is also used in interfaces within the library itself, between different components of the library. The runtime polymorphism layer that we provided for a particular “entry point” is not in effect after that entry point. A single handle provides a runtime dispatching layer from a single concept to its models—the concept is still selected statically and thus refinement between concepts is based purely on static type information.

As ConceptC++’s overload resolution is based on the static type of the arguments, it is not possible to non-intrusively inject runtime polymorphism to internal library interfaces that use overloading, such as the **adjust_measurement** functions called from **solve**. Hence, if we wish to support a form of refinement where the dynamic type affects which overloaded function to select, the dynamic type information has to be recovered with some form of “type casing”. This means that the solution becomes intrusive to the generic library. The generic library, **layout_engine** in this case, must be written to support this behavior explicitly. The library must know the handle class that corresponds to each concept it uses.

We demonstrate with the **adjust_measurement** function. We provide a new overload that matches handle types, detects the dynamic type, and dispatches based on it. This function forwards to the previously shown **adjust_measurement** functions. The forwarding call is qualified with the namespace of the layout library; the new overload is placed in a different namespace, together with the handle and body classes. Unqualified calls (e.g. from **solve**) to **adjust_measurement** see the entire overload set because of C++’s argument dependent lookup. This arrangement is one way to avoid the new overload to match again leading to infinite recursion.

```
void adjust_measurement(PlaceableHandle& t, extents_t& e, const place_data_t& m) {
  if (PlaceableTwopassHandle* p = PlaceableTwopass_cast<PlaceableTwopassInterface*>(&t))
    layout::adjust_meaurement(*p, e, m);
  else
    layout::adjust_meaurement(t, e, m);
}


PlaceableTwopassHandle& PlaceableTwopass_cast(PlaceableHandle& x) {
  dynamic_cast<PlaceableTwopassInterface&>(*x.interface_m);
  return reinterpret_cast<PlaceableTwopassHandle&>(x);
}
```

The **PlaceableTwopass_cast** function behaves like **dynamic_cast**: it returns a null pointer if casting is not successful. When casting from a refined handle to a base handle, we rely on the fact that a handle class consists exactly of a pointer to a class derived from the corresponding interface class. For example, **PlaceableHandle**’s **place()** operation is implemented in terms of **PlaceableInterface** by jumping to the fourth entry in the vtable. Since **PlaceableTwopassInterface** inherits from **PlaceableInterface**, the vtable slots will align, and we can treat **PlaceableTwopassHandle** as fully substitutable for **PlaceableHandle**. Of course a static cast will not work for this purpose, since the type system cannot verify this layout compatibility; here, we know by construction that the cast is safe.

A cast from a base handle to a refined handle will only succeed if the dynamic type of the base handle's Interface object is a derived from the refined handle's Interface type. Thanks to the requirement that refined interfaces must inherit from base interfaces, we can employ **dynamic_cast** to determine whether this type requirement is satisfied. We implement multiple forms of **poly_cast**, corresponding tho those offered by to **dynamic_cast**: a reference form which throws when attempting a downcast to an inappropriate type, and a pointer form which returns **NULL** upon such failure.

3.3 The small-object optimization

In the sections above, we demonstrated how to construct polymorphic value objects, with the help of the **PlaceableHandle**, **PlaceableInterface**, and **PlaceableImplementation** classes. Under the hood, we employed inheritance techniques from object-oriented design in a traditional manner. In particular, **PlaceableHandle** allocates the body class on the heap using operator **new**. This section presents an optimization that avoids heap allocations entirely when certain conditions are met. We refer to this optimization as the small-object optimization. For earlier work in this area see [23].

In small-object optimization we maintain a small buffer in the **PlaceableHandle** itself, in which small **PlaceableImplementation<T>** objects can be stored directly. Space is allocated from the free store only for objects too large to fit into this buffer. Also, types that do not have a non-throwing default constructor must use free store.

We encapsulate the "local" (small objects stored in the handle) and "remote" (for heap-allocated objects) storage policies in the class templates shown in Figure 6—the former in **PlaceableStateLocal**, the latter in **PlaceableStateRemote**. We modify the **PlaceableImplementation** class to inherit from one of these classes according to whether the criteria for the small-object optimization are met or not. The **choose_storage** metafunction, in Figure 7, carries out this selection based on the size of the stored object.

In the case of a small object, we do not need the pointer (**interface_m** member in Figure 3) from the handle class to the implementation class, since the implementation object resides in the handle's buffer (the **data_m** member). Alternatively, for large objects stored on the heap, the **data_m** buffer can store the pointer to the heap allocated implementation object. To uniformly access the interface object regardless of how it is stored we encapsulate access in **PlaceableHandle** as shown below. (The rest of the changes to **PlaceableHandle** are straightforward, see Figure 8).

```
typedef double storage_t[2];

struct PlaceableHandle {
  ...
  PlaceableInterface& interface_ref() { return static_cast<PlaceableInterface>(storage()); }
  const PlaceableInterface& interface_ref() const
    { return static_cast<const PlaceableInterface >(storage()); }
  void storage() { return &data_m; }
  const void storage() const { return &data_m; }
  storage_t data_m;
  ...
```

```
template <typename T>
struct PlaceableStateLocal : PlaceableInterface {
  typedef T value_type;

  void assign(const PlaceableInterface& x) ;

  const value_type& get() const { return value_m; }
  value_type& get() { return value_m; }

  PlaceableStateLocal() {}
  explicit PlaceableStateLocal(const value_type& x) : value_m(x) {}

  T value_m;
};

template <typename T>
struct PlaceableStateRemote : PlaceableInterface {
  typedef T value_type;

  void assign(const PlaceableInterface& x);

  const value_type& get() const { return *value_ptr_m; }
  value_type& get() { return *value_ptr_m; }

  PlaceableStateRemote() : value_ptr_m(NULL) {}
  explicit PlaceableStateRemote(const value_type& x) { value_ptr_m = new value_type(x); }

  ~PlaceableStateRemote() { delete value_ptr_m; }

  T* value_ptr_m;
};
```

Fig. 6. The small-object optimization: storage policy classes.

```
};
```

By a the same token, we can no longer assume that our concrete **Placeable** object is accessible as **placeable_m** inside of **PlaceableImplementation**. Instead we modify **PlaceableImplementation** as in Figure 7 to use use the **get** member function inherited from its base storage class. Then, for example, the **measure** function becomes:

```
template <typename T>
struct PlaceableImplementation : choose_storage_type<T>::type {
  ...
  void measure(extents_t& result) { Placeable<T>::measure(this−>get(), result); }
  ...
};
```

The constructors, destructor, and clone function are also impacted by the two different storage policies. For example, the **clone** call is modified to use the *placement* form of the **new** operator, placing the result in the storage provided by the **PlaceableHandle** class.

After applying the small-object optimization, we are left with the arrangement of classes shown in UML in Figure 9.

3.4 Commonality/Variability Analysis

The final stage of our development is to perform a commonality/variability analysis, then to factor the common portions of our code away from the specific details of **Placeable**, to produce the outline of a generic non-intrusive, value-based runtime polymorphic library, **poly<>**.

The starting point is the arrangement of classes in Figure 9. We transform the **Placeable**-specific artifacts into the outline of the **poly<>** library, by splitting the *interface*, *implementation*, and *handle* classes into **Placeable**-independent and **Placeable**-specific pieces, parameterizing classes and introducing classes as needed. We begin with **PlaceableInterface**.

The **PlaceableInterface** specified the **measure**, **place**, **clone**, **assign**, and destructor operations. Only the first two of those are **Placeable**-specific, leading to a new version of **PlaceableInterface** with only those two functions. We introduce a new class template, **poly_interface**, which inherits from two classes: from its template parameter, in our case **PlaceableInterface**, and from a new interface, **value_interface**, containing the **Placeable**-independent operations **clone(), assign()**, and the virtual destructor, as shown below:

```
struct PlaceableInterface {
  virtual void measure(extents_t& result) = 0;
  virtual void place(const place_data_t& place_data) = 0;
};

struct value_interface {
  virtual value_interface∗ clone(void∗) const = 0;
  virtual void assign(const value_interface& x) = 0;
  virtual ~value_interface() {}
  ...
};
```

```
typedef double storage_t[2]; // 8 bytes for vtable ptrs, 8 bytes for storage.

template<typename T, int N=sizeof(storage_t)>
struct is_small {
  enum { value = sizeof(T) <= N &&
              std::has_nothrow_constructor<typename T::value_type>::value };
};

template <typename ConcreteType>
struct choose_storage_type :
  boost::mpl::if_<is_small<PlaceableStateLocal<ConcreteType> >,
                  PlaceableStateLocal<ConcreteType>,
                  PlaceableStateRemote<ConcreteType> > {};

struct PlaceableInterface {
  ...
  virtual PlaceableInterface* clone(void* storage) const = 0;
  ...
}

template <typename T>
struct PlaceableImplementation : choose_storage_type<T>::type {
  typedef typename choose_storage_type<T>::type storage_type;

  void measure(extents_t& result) { Placeable<T>::measure(this−>get(), result); }
  void place(const place_data_t& place_data)
    { Placeable<T>::place(this−>get(), place_data); }

  PlaceableImplementation* clone(void* storage) const
    { return new (storage) PlaceableImplementation(this−>get()); }

  PlaceableImplementation(const T& x) : storage_type(x) {}
  PlaceableImplementation() : storage_type() {};
};
```

Fig. 7. The small-object optimization: Interface and Implementation. **PlaceableInterface** is unchanged except for the clone function that now needs a parameter to indicate where to clone the object.

```
struct PlaceableHandle {
  void measure(extents_t& result) { interface_ref().measure(result); }
  void place(const place_data_t& place_data) { interface_ref().place(place_data); }

  PlaceableHandle(const PlaceableHandle& x) { x.interface_ref().clone(storage()); }
  ~PlaceableHandle() { interface_ref().~PlaceableInterface(); }

  template <Placeable T>
  explicit PlaceableHandle(const T& x) {
    new (storage()) PlaceableImplementation<T>(x);
  }

  PlaceableHandle& operator=(const PlaceableHandle& x) {
    if (x.type_info() == type_info())
      interface_ref().assign(x.interface_ref());
    else {
      interface_ref().~PlaceableInterface();
      x.interface_ref().clone(storage());
    }
    return *this;
  }

  PlaceableInterface& interface_ref()
    { return *static_cast<PlaceableInterface*>(storage()); }

  const PlaceableInterface& interface_ref() const
    { return *static_cast<const PlaceableInterface *>(storage()); }

  void* storage() { return &data_m; }
  const void* storage() const { return &data_m; }

  storage_t data_m;
};
```

Fig. 8. The small-object optimization: Handle.

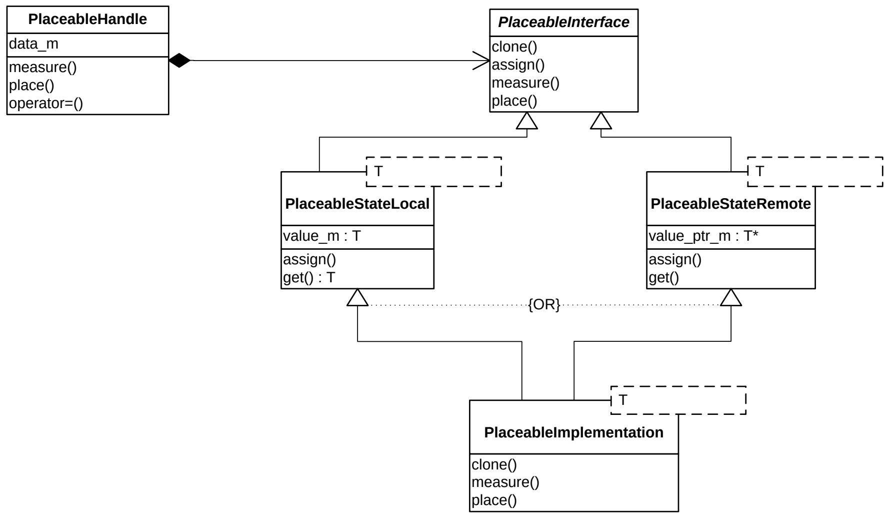

Fig. 9. UML static structure diagram for the **Placeable** wrapper class with the small-object optimization

```
template <typename I> // Here I will be PlaceableInterface
 struct poly_interface : value_interface, I {
  typedef interface_type I;
};
```

The handle class can now refer to an instantiation of the **poly_interface** class template as **poly_interface<PlaceableInterface>**, instead of the **Placeable**-specific **PlaceableInterface**.

The local and remote storage classes in Figure 9 were only **Placeable**-specific in as much as they inherited from **PlaceableInterface**. We can remove this specificity by employing parametrized inheritance—we add a template parameter for the interface class from which they are to inherit:

```
template <typename T, typename Interface>
struct poly_state_local : poly_interface<Interface> {
  typedef T value_type;
  typedef Interface interface_type;
  ...
};
```

```
template <typename T, typename Interface>
struct poly_state_remote : poly_interface<Interface> {
  typedef T value_type;
  typedef Interface interface_type;
  ...
};
```

We omit the unchanged bodies of the storage class templates.

Having removed application-specific code from the interface and storage classes, we next focus on the implementation class. The **poly_implementation** class below implements the **clone** function, instead of requiring that from the **PlaceableImplementation** class.

```
template <typename Implementation>
struct poly_implementation : Implementation {
    typedef typename Implementation::value_type value_type;
    typedef typename Implementation::interface_type interface_type;

    poly_implementation(const value_type& x) : Implementation(x){ }
    poly_instance(): Implementation() { }

    value_interface* clone(void* storage) const
      { return new (storage) poly_implementation(this−>get()); }
};
```

Finally, we turn our attention to refactoring the **PlaceableHandle** class, which comprises **Placeable**-specific delegating functions, a storage buffer and protocol for accessing it, and construction, assignment and destruction protocol. As with our other artifacts, we factor out the **Placeable**-specific delegating functions into a new class, **placeable** in Figure 11 (note the lower-case spelling), and a base class template, **poly_base**, encapsulating the boilerplate functionality, in Figure 10.

We now wrap the handle in a final layer to be able to refactor **PlaceableTwopass_cast** as **poly_cast**:

```
template <class UserConceptRep>
class poly : public UserConceptRep {
public:
  template <typename T> explicit poly(const T& s) : UserConceptRep(s) {}
};

template <typename T, typename U>
T poly_cast(poly<U>& x) {
  typedef typename boost::remove_reference<T>::type target_type;
  typedef typename target_type::Interface target_interface_type;
  if(!x.template is_dynamic_convertible_to<target_interface_type>())
    throw bad_cast(typeid(poly<U>), typeid(T));
  return reinterpret_cast<T>(x);
}
```

This completes are description of the structure of the **poly** library. With our library, programmers wishing to create runtime polymorphic value wrappers must complete three tasks. For example, three responsibilities must be carried out in order for the library to generate the **Placeable**-related classes which we wrote by hand above. The wrapper-author must provide the **placeable** class containing **Placeable**-specific delegation routines, as in Figure 11. They must also supply the **PlaceableInterface** class, as in the beginning of this section, and the **PlaceableImplementation** class template. Once these tasks have been carried out, the wrapper class is available as **poly<placeable>**.

```
template <typename I, template <typename> class Instance>
struct poly_base {
    template <typename T, template <typename> class U>
    friend struct poly_base;

    typedef poly_interface<I> interface_type;
    typedef I Interface;

    // Construct from value type
    template <typename T>
    explicit poly_base(const T& x,
                 typename std::disable_if<std::is_base_of<poly_base, T> >::type∗ = 0);

    // Construct from related interface (may throw on downcast)
    template <typename J, template <typename> class K>
    explicit poly_base(const poly_base<J, K>& x,
        typename std::enable_if<is_base_derived_or_same<I, J> >::type∗ dummy = 0);

    poly_base(const poly_base& x);
    friend inline void swap(poly_base& x, poly_base& y);
    poly_base& operator=(const poly_base& x);
    ~poly_base();

    template <typename J, template <typename> class K>
    static bool is_dynamic_convertible_from(const poly_base<J, K>& x);
    template <typename J>
    bool is_dynamic_convertible_to() const;

    const std::type_info& type_info() const;
    template <typename T> const T& cast() const;
    template <typename T> T& cast();
    template <typename T> bool cast(T& x) const;
    template <typename T> poly_base& assign(const T& x);
        // Assign from related (may throw if downcastisng)
    template <typename J, template <typename> class K>
    typename std::enable_if<is_base_derived_or_same<I, J> >::type
    assign(const poly_base<J, K>& x);
    const interface_type∗ operator−>() const;
    interface_type∗ operator−>();
    interface_type& interface_ref();
    const interface_type& interface_ref() const;
    void∗ storage();
    const void∗ storage() const;
    implementation::storage_t data_m;
};
```

Fig. 10. The **poly_base** class template.

```
struct placeable : public poly_base<PlaceableInterface, PlaceableImplementation> {
  template <typename T>
  explicit placeable(const T& s)
    : poly_base<poly_placeable_interface, poly_placeable_instance>(s) {}

  void measure(extents_t& result) { interface_ref().measure(result); }
  void place(const place_data_t& place_data) { interface_ref().place(place_data); }
};
```

Fig. 11. The **placeable** class.

4 Discussion and Conclusions

Modern software applications development demands the ability to combine application-specific functionality with independently developed library modules from a variety of sources. Purely object-oriented libraries can be written to support multiple types that are unknown at compile-time (dynamic polymorphism). However, such libraries suffer from composability limitations, since they require that the types that are to extend the library (intrusively) inherit from particular abstract base classes. Generic libraries, on the other hand, are highly composable but offer no support for dynamic polymorphism. This paper described **poly<>**, a multi-paradigm library that gives programmers a means to support non-intrusive runtime polymorphism, parametrized by concept. We discussed the implementation of the library in stages, contrasting our techniques to other popular idioms in the generic programming, object-oriented programming, and the design patterns communities. We used these ideas to demonstrate how to create non-intrusive heterogeneous containers, and how to retroactively add runtime polymorphism to a templated interface of a generic library. The full version of a **poly<>** library, similar to the version discussed in this paper, can be found at `http://opensource.adobe.com`. That version is C++ 2003 compatible, and thus uses various library techniques to emulate concepts.

The notion of heterogeneous containers of polymorphic value types has been approached by others. The **any** library [12, 13] enables a similar encapsulation of value types: when wrapped inside the **any** class template, values of varying types can be handled uniformly. The **any** library, however, does not support directly operating with values wrapped in **any**—to apply an operation to such a value, it is necessary to first unwrap the value and explicitly recover its type. Our **poly<>** library is parametrized over the concept/interface that the wrapped values must support. In other words, if **empty** is an empty interface, **poly<empty>** corresponds to **any**. Nasonov's *dynamic any* library [20] resembles our **poly**, but takes each allowed operation signature as a distinct template parameter. Neither any nor dynamic any handle the analogue of concept refinement. Our **poly<>** is essentially a type constructor for defining existential types [17], where the hidden type is constrained to be a model of a particular concept. Haskell's "**forall**" construct serves a similar purpose: it allows the definition of types with a hidden part constrained to be a type belonging to a given type class, or classes.

Future work remains, to improve the integration of dynamic polymorphism to generic programming. ConceptC++ supports multi-parameter concepts. In this paper, however, we only discussed the dynamic representation of single parameter concepts. Allowing more than one parameter vary independently at run-time leads to ambiguity issues well-known in the context of multi-methods. Furthermore, generic algorithms often require certain relations between their argument types or between associated types—for example that the value types of two different iterator types are the same. The current framework does not support expressing such requirements at run-time.

Generic programming is a powerful paradigm supporting the creation of efficient, easily composed extensible libraries. Our goal has been to expand the applicability of generic programming beyond the domain of static polymorphism. We illustrated state of the art techniques for extending generic programming into the realm of dynamic polymorphism, borrowing from ConceptC++ concepts and concept maps, traditional generic programming interfaces, template metaprogramming, design patterns, and object-oriented design techniques. While the **poly<>** implementation is complex, its use can be made transparent to the end-user. The benefits, such as the ability to create non-intrusive heterogeneous containers, the improved control over lifetime and aliasing issues, and the transparent adaptation of statically polymorphic interfaces are substantial.

References

1. Adobe Systems, Inc. *Adobe Source Library*, 2005. `opensource.adobe.com`.
2. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, Aug. 2001.
3. Apple Inc. *Carbon API for Mac OS X*, 2007. `developer.apple.com/carbon/`.
4. M. H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
5. E. P. Becker. Working draft, standard for programming language C++. Technical Report N2009=06-0079, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Apr. 2006.
6. L. Bourdev and H. Jin. *Generic Image Library*, 2006. `opensource.adobe.com/gil`.
7. J. C. Dehnert and A. Stepanov. Fundamentals of Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11, Schloss Dagstuhl, Germany, April 1998.
8. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., New York, NY, USA, 1995.
10. D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06)*, pages 291–310. ACM Press, October 2006.

11. D. Gregor and B. Stroustrup. Proposed wording for concepts. Technical Report N2193=07-0053, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2007.
12. K. Henney. From mechanism to method: Valued conversions. *C++ Report*, 12(7), July–August 2000.
13. K. Henney. The Boost.Any library. `http://www.boost.org/libs/any`, 2001.
14. International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages: C++*. Geneva, Switzerland, 2 edition, 10 2003.
15. J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in c++. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 272–282, New York, NY, USA, 2006. ACM Press.
16. M. Jazayeri, R. Loos, D. Musser, and A. Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, Apr. 1998.
17. K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.
18. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1997.
19. D. A. Musser and A. A. Stepanov. Generic Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.
20. A. Nasonov. I/O System: dynamic_any Campaign. *C/C++ Users Journal*, Sept. 2003.
21. S. Parent. Beyond objects: Understanding the software we write. Presentation at C++ Connections, `opensource.adobe.com/wiki/index.php/Image:Regular_object_presentation.pdf`, Nov. 2005.
22. W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The Bioinformatics Template Library–generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.
23. J. Reeves. String in the Real World—part 2. *C++ Report*, Jan. 1999. `www.bleading-edge.com/Publications/C++Report/v9901/Column14.htm`.
24. J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
25. J. Siek and A. Lumsdaine. The Matrix Template Library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.
26. Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. `http://www.sgi.com/tech/stl/`.
27. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, Apr. 1994. `http://www.hpl.hp.com/techreports`.
28. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.