Chameleon Objects,
or how to write a generic, type safe wrapper class

Volker Simonis[¶]
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, 72076 Tübingen, Germany
E-mail: simonis@informatik.uni-tuebingen.de

29th November 1999

Abstract

Generic programming [Musser-Stepanov] offers the ability to parameterize functions and classes with arbitrary data types. This new technique allows us to focus on the nature of algorithms rather than on their implementations for special types. Unfortunately, in a language like C++ it is not always possible to use parameterization : For example, virtual functions cannot be parameterized ([ANSI-CPP] 14.5.2). Furthermore, the primitive data types often don't integrate very well into a system of derived classes. This article describes one solution to this problem: a simple and elegant wrapper class which can hold arbitrary data types and can be used to pass these objects between different program units while maintaining type safety.

1 The problem

The motivation for this article arose when I tried to implement Java Serialization in C++[Ser-Spec]. Because in C++ there's no runtime system which, like the Java Virtual Machine, can create new objects and set or query their data fields, this functionality has to be provided by the C++ classes themselves.

Therefore, in my library, every C++ class which wants to be serialized has to be derived from the abstract base class `Serializable`. `Serializable` declares a set of pure virtual functions for setting and querying the object's data fields. These functions must be implemented by all derived classes individually.

For a class `List` defined as follows:

```
class List : public Serializable {
  List *next;
  // ...
};
```

the method `setValue()`, for setting a data field, would look like this:

```
void List::setValue(const string& var, Serializable *val) {
  if (var == "next") {
    next = dynamic_cast<List*>(val);
  }
}
```

The `dynamic_cast<>` ensures that conversions to more derived classes are performed at runtime and only if meaningful (i.e., type-safe) in that context.

This worked fine as long as a class contained only members that were also derived from `Serializable`. But

[¶] Part of this work has been gracefully supported by debis Systemhaus Fellbach/St.Petersburg

what if it had primitive data members, or owned members of third party classes for which we had no source code? Since the function `setValue()` is virtual, we can't write something like:

```
template <class T>
virtual void setValue(const string&, T*) = 0;
```

Instead, there are two common solutions. If we know the number of possible data types, and this number is not too big, we can write a specialized `setValue()` function for every type:

```
virtual void setValue(const string&, Serializable*) = 0;
virtual void setValue(const string&, int*) = 0;
virtual void setValue(const string&, double*) = 0;
...
```

The disadvantage of this solution is the high amount of code that must be written for every derived class, even for types that class doesn't actually use. It is possible instead to supply a default implementation for every method which for example prints an error message or throws an exception. This minimizes the number of methods which actually have to be implemented by derived classes, but we still have to change the base class `Serializable` itself if, in the future, we want to handle a new data type in one of the derived classes.

The second solution is to use only a `void*` as data type, and cast the pointer according to the desired result. This would be the typical C approach, but now we can't use `dynamic_cast<>` anymore, and therefore we have no guarantee of the correctness of the cast[1].

2 The solution

To solve the problems mentioned above, we created a new class called `Value`. An arbitrary variable `v` of type `T` can be assigned to an instance of `Value`, and thereafter the `Value` object itself can be assigned to any instance of type T, just as if it were `v` itself. If the caller tries to assign a `Value` object to a variable of a type other than `T`, it will throw an `Incompatible_Type_Exception`.

We first look at an example:

```
Value v1(999.999);
Value v2;
v2 = (string)"hello";
string s = v2;
try {
  int i = v1;
} catch (Incompatible_Type_Exception) {}
```

Here, two instances of `Value` are created. The first, `v1`, is initialized with the value 999.999 (implicitly a `double`). `v2` is assigned a `string` value. After `"s = v2"`, `s` will contain the string `"hallo"`, just as if `v2` had been of type `string`. Moreover, `"int i = v1"` will throw an `Incompatible_Type_Exception` because `v1` is of type `double` at the time of the assignment. Notice, however, that this type checking is done at runtime. If we precede the assignment with `"v1 = 1"`, no exception will be thrown. In other words, an assignment to a `Value` object will always succeed. Depending on the context, it may or may not change the object's "internal" type. An assignment of a `Value` object to an object of any other type, on the other hand, is always dependent on the `Value` object's actual internal type and should always be guarded be a `try/catch` clause.

With this in mind, we can now rewrite our `setValue()` function from section 1 in the following way:

```
virtual void setValue(const string& s, Value val) = 0;
```

In the implementation of the function, we can simply assign `val` to any arbitrary typed variable without a

[1] Java, for example, doesn't offer parameterization at all. Instead, in Java, all objects inherit from the single abstract base class `Object`. This allows us to write some kinds of generic programs (for example the Java Generic Library, JGL, a Java STL-clone [JGL]), although these are not really parameterized. Instead, we have to use manual type casts like in the example above. The difference between casts from `Object` in Java and `void*` in C++ is the type safety in Java, because Java casts are performed at run time and thus are identical to a `dynamic_cast<>` in C++, even though the syntax is more similar to the simple static casts of C.

cast, since type checking is done automatically for us at runtime and a type mismatch is signaled through an exception.

3 The implementation

First of all, we define the exception class. Its only purpose is to be thrown when a `Value` object is assigned to an object of incompatible type. In this case, `errorType` will hold the type name of the object we were trying to assign to.

```
class Incompatible_Type_Exception {
private:
  string errorType;
public:
  Incompatible_Type_Exception(const string& s) { errorType = s; }
  string getError() const { return errorType; }
};
```

The class `Value` itself is defined as follows:

```
class Value {
private:
  enum Action { SET, GET };
  template <class T> T& value(T t = T(), Action action = GET)
    throw (Incompatible_Type_Exception&);
public:
  Value() {}                                          // Default constructor
  template <class T> Value(const T&) { value(t, SET); }  // Generic constructor
  template <class T> operator T() const throw (Incompatible_Type_Exception&) {
    return const_cast<Value*>(this)->template value<T>();// const_cast is safe
  }
  template <class T> T& operator=(const T &t) { return value(t, SET); }
};
```

Notice that Value itself is not a template class. But it heavily uses parameterization, since all of its methods, including the constructor, are template functions.

Furthermore, we can see that all public methods of the class contain only a call to the private method `value()` which is itself a template function. Also notice that `Value` has no data members; i.e. a `Value` object uses no memory. Unfortunately, since objects are known to the compiler only by address, even stateless objects such as these are given an address in memory. That's why even objects like this use at least one byte of memory (try `sizeof()`).

Now let's look at `value()`, since it seems to contain all the magic of our class:

```
template <class T>
T& Value::value(T t, Action action) throw (Incompatible_Type_Exception&) {
  static map<Value*, T > values;
  switch(action) {
    case SET :  {
      values[this] = t;
      return t;
    }
    case GET : {
      if (values.count(this)) return values[this];
```

➥

```
      else throw Incompatible_Type_Exception(typeid(T).name());
    }
  }
}
```

And indeed, we have finally found the trick behind it. The function `value()` contains the static local variable `values`, which is of type `map<Value*, T>` ([Musser-Saini]). Since this function is instantiated by the compiler once for every data type `T` we use in conjunction with `Value` objects, there will be a separate variable called `values` for each type `T`. Each `Value` object consumes one slot in the map corresponding to its actual internal type. Because these maps are static, they will be created before the start of our program, and because read/write operations to maps are guaranteed only to need logarithmic time [Lee-Stepanov], we can at least expect logarithmic time for our `Value` class, too. This is okay for an example like this, but can and should be changed to something more efficient in a real world application, since the user of the class will expect an assignment to use only constant time[2].

Since we declared the conversion operator `operator T()` as a template function, we can assign a Value object to another object of any arbitrary type. The compiler will generate the right conversion function for us under the hood. The same applies to the parameterized assignment operator `operator=()`. It allows assignments of arbitrary data types to our value object. These two operator functions just call the `values()` function of the appropriate type, which queries the map of the right type. Notice that we need `values()` as an auxiliary function, since we must have only one map for each type. Since the `Value` class itself is not parameterized, the only place we can store this map is in a static variable in a parameterized function.

I will try to explain this in more detail now. Recall the code example from the beginning of section 2. After compiling and starting it, we will have a memory layout similar to the one shown in figure 1 (a). Notice that the compiler has created three static maps of the types `map<Value*, double>`, `map<Value*, string>` and `map<Value*, int>` respectively. The first one was created because `v1` is initialized with the `double` value 999.999. This means the parametrized constructor of the `Value` class will be called with a `double` argument and will call the `value()` method for the type `double`. When the new value() function is instantiated, the static variable `values`, which in this case is of type `map<Value*, double>`, will also be instantiated.

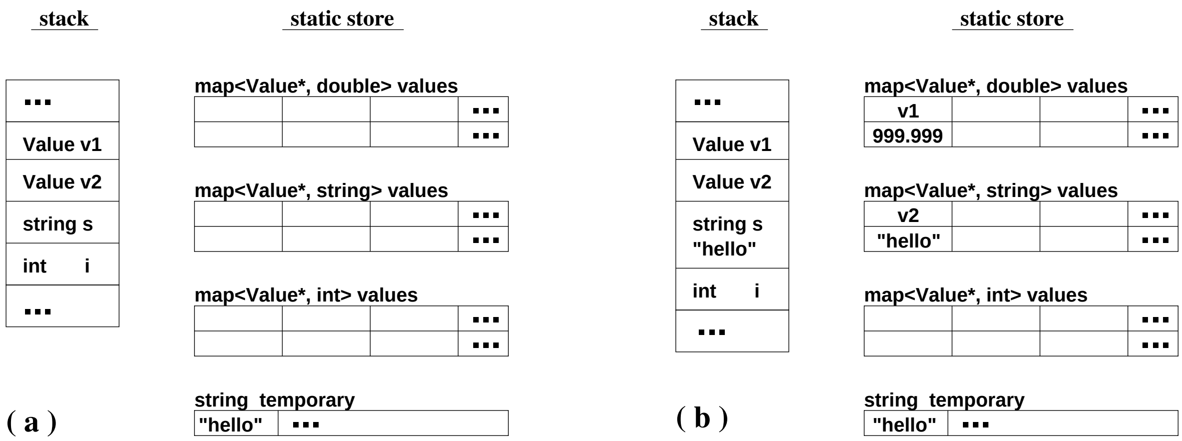

Figure 1: The memory layout of the program produced by the code at the beginning of section 2. (a) at program start, (b) at the time of entering the `try/catch` block

.

The same description applies to the `<Value*, string>` map, since in line 3 of the code example we assign a `string` object to the `Value` variable `v2`. The line "**int** `i = v1`" finally leads to the creation of the third map, since

[2] It would be possible here to replace the `map` container with a hash-table (for example `hash_map` from the SGI-STL implementation [SGI-STL]. If the initial size of the hash-table will be chosen carefully, nearly constant time for assignments can be achieved.

it will call the `int()` operator of the `Value` class, which in turn will call the `value()` method with an `int` as the template argument.

Figure 1(b) shows the situation after entering the `try` block. The `<Value*, double>` map holds an entry which maps the address of `v1` to the `double` value 999.999 and the `<Value*, string>` map holds an entry which associates `v2` with the `string` object "`hello`". As mentioned earlier, the objects `v1` and `v2` haven't changed at all (in fact, they cannot change because they have no data fields).

All this shows that the assignment of any type `T` to a `Value` object will always succeed, since in fact it only executes :

```
case SET : {
  values[this] = t;
  return t;
}
```

where `values` in this case is of type `map<Value*,T>` for every type `T`. On the other hand, when assigning a `Value` object to a variable of type `T`, we must first check whether our object holds a value of the right type by querying the map corresponding to the appropriate type for the object's address. In fact, "checking if our object holds a value of the right type" in the above sentence really means "checking if the address of the object is stored in the map corresponding to the right type" since a `Value` object itself has no state.

```
case GET : {
  if (values.count(this)) return values[this];
  else throw Incompatible_Type_Exception(typeid(T).name());
}
```

If no such entry is found, an exception is thrown. Notice, however, that the compiler creates the operator functions and the map for that type whether the assignment succeeds or not, since the functions are generated at compile time. At compile time, there's no way to tell whether this map will actually hold any values at runtime. Again, this is the reason why the `<Value*, int>` map in the example above was created even though it never contains anything.

The maps are declared as static local variables in the `value()` method because this way the compiler will generate all the necessary maps for us in the same way that it creates instantiations of all the necessary parameterized methods. Also, we need exactly one map of every type, which is why we cannot use different methods to set and query one `Value` object's value. This is why the `value()` method has to use a dispatching technique to simulate two (later in this paper we will expand it to three) different functions.

Logically, these maps don't inherently belong inside the `Value` class. They could just as appropriately be declared as global variables[3], but then the programmer must make sure that all the necessary global variables have been defined. Even worse, it's not possible to have objects of different type but the same name in one name space. We can only overload function names, not variable names.

4 Some implementation details

Now we have a working system, but there are still two major problems we have to solve. First, we must correctly handle assignment from one `Value` object to another, and second, we must somehow handle object destruction.

4.1 Object cleanup

Let's start with the problem of cleaning up the data stored in `Value` objects. This is not straightforward, and introduces some performance and space overhead. Notice again that Value objects are created with a template constructor, which will insert a reference to the `Value` object and the actual value into an appropriate map. But the `Value` object itself is not parameterized and thus holds no information concerning the type of the value it holds. In other words, since destructors cannot be parameterized (i.e., template functions ([ANSI-CPP] 14.5.2)), there is an asymmetry between object creation and destruction. Of course, templatized destructors would be

[3] In fact, static data members in C++ have external linkage and thus behave like global objects.

nonsense, since in general it would not be feasible to determine which destructor function to call at compile time.

So we must think of something else. The solution is to store in each `Value` object a pointer to a member template function that it calls to delete its value. But once again, since the object itself is not parameterized, the signature of this function must be independent of the template argument. The idea behind this is to enable the `Value` destructor to call a function at destruction time which knows internally what type the calling object actually refers to, but can itself be called through a "generic" pointer to member function. Thus we use a function like the following:

```
template <class T>
inline void Value::deleteValue() {
  static T t;       // Used only as type selector.
  value(t, DELETE);
}
```

and add the following member variable to the definition of the `Value` class.

```
typedef void(Value::*FuncPointer)(void);
FuncPointer fp_DELETE;
```

Notice the type of the pointer. It is not "pointer to a function that takes void and returns void", but "pointer to a member function of `Value` that takes a void argument and returns void" ([ANSI-CPP] 8.3.3). This is a significant difference because one cannot be converted into another. Now, since the name of a template function is considered to name a set of overloaded functions ([ANSI-CPP] 13.4), we can assign the address of an instantiation of `Value::deleteValue()` to our pointer to member function `fp_DELETE`. In fact, all functions generated from the template for `Value::deleteValue()` have the same type. Their template arguments control their implementations, but not their signatures.

Now we can add a destructor to the class definition for Value:

```
~Value() { if (fp_DELETE != NULL) (this->*fp_DELETE)(); }
```

and change the function `Value::value()` to handle deletion:

```
template <class T>
T Value::value(T t, Action action) throw (Incompatible_Type_Exception&) {
  static map<Value*, T > values;
  switch(action) {
    case SET : {
      values[this] = t;
      if (fp_DELETE == NULL) fp_DELETE = &deleteValue<T>;
      else if (fp_DELETE != (FuncPointer)&deleteValue<T>) {
        (this->*fp_DELETE)();    // Delete old value of type != T for this obj
        fp_DELETE = &deleteValue<T>; // Remember delete function of right type
      }
      return t;
    }
    case DELETE : { // only called by destructor
      if (values.count(this)) values.erase(this);
      return t;
    }
    case GET : {
      if (values.count(this)) return values[this];
      else throw Incompatible_Type_Exception(typeid(T).name());
    }
  }
}
```

If `value()` is called with the new action `DELETE`, it simply removes the entry for the calling object from the internal map. Furthermore, the `SET` action is changed to assign the right function to the class variable `fp_DELETE`. If `fp_DELETE` points to a function of another type, that function is called first to remove the old value of the object.

With this solution, a `Value` object can hold no value at all, or exactly one value of arbitrary type. Before these changes, it could actually hold one value for every distinct data type. If we want to preserve this behavior now, we must keep not just one pointer to a cleanup function, but a list of pointers, one for every type actually held by the object. This is possible, but would result in an assignment time linear to the number of values the object holds, since this list would have to be adjusted every time.

4.2 The copy assignment operator

Now we can implement the assignment of one `Value` object to another. As with the destructor in the previous section, a `Value` object doesn't know which of the existing maps contains a reference to it, if indeed any of them do. But again, this is the information we need if we want to achieve a behavior like the following:

```
Value v1(1234567);
Value v2(999.999);
v2 = v1;
int i = v2;
```

After assigning `v1` to `v2` we want `v2` to be of internal type integer, too, and to hold the same value as `v1`, namely `1234567`. Remember that with our current implementation, this is not the case. Instead, we will end up with an instance of the assignment operator for the template parameter `Value`, which will contain a map of type `map<Value*, Value>`, making `v2` hold a value of type `Value` – not quite what we intended. Then an `Incompatible_Type_Exception` will be thrown, since `"int i = v2"` is trying to assign a `Value` object to an integer.

The solution is to explicitly define the copy assignment operator. Because it must not be a template function ([ANSI-CPP] 12.8), a specialization of our template assignment operator is not enough. So we add the following declaration to `Value`:

```
Value& operator=(const Value&);
```

and supply the appropriate implementation :

```
Value& Value::operator=(const Value &val) {
  if (this != &val && val.fp_CLONE != NULL) {
    (val.*val.fp_CLONE)(this);
    return *this;
  }
}
```

All this function does is call the member function of its only argument `val` pointed to by `fp_CLONE`, passing its own address as the parameter. Notice the somewhat strange syntax of the function call `"(val.*val.fp_CLONE)(this)"`. It means "Call the member function of `Value` pointed to by the `fp_CLONE` member of `val` for the object `val`". If we only write `"(val.*fp_CLONE)(this)"` the member function pointed to by `this->fp_CLONE` would have been called for `val`, but since `fp_CLONE` is a pointer to a function out of a set of overloaded template functions, this may be a different function.

The pointer to member operators `.*` and `->*` are binary operators, which take two arguments. The left argument is a pointer or a reference to an object (since every member function needs an implicit `this` pointer), while the right argument is a pointer to member ([ANSI-CPP] 5.5).
Finally, the definition of `fp_CLONE`:

```
typedef void(Value::*ClonePointer)(Value*) const;
ClonePointer fp_CLONE;
```

and the appropriate template function `cloneValue(Value*)`:

```
template <class T>
```

➥

```
void Value::cloneValue(Value *val) const {
  val->template value<T>(const_cast<Value*>(this)->template value<T>(), SET);
}
```

The technique is the same one we used in the function `deleteValue()` called by the destructor, only now we must also pass a pointer to the `Value` object we are assigning to, since we want to change its type and value. Notice that we have to use a `const_cast` in this case, because the copy assignment operator defines its argument as `const`. But since, as stated in section 3, we must use the same function for setting and retrieving an object's value, we cannot declare this function to be constant. But since we just retrieve the object's value in this case, we don't actually violate the const constraint declared by the assignment operator. This is a good example of a situation where `const_cast` is necessary.

Finally, we have to change the `SET` action so that it sets `fp_CLONE` to point to a `cloneValue()` function of the right "internal" type. We can accomplish this as follows:

```
fp_CLONE = &cloneValue<T>;
```

4.3 The copy constructor

After the implementation of the copy assignment operator with the help of the `cloneValue()` method, the realization of the copy constructor imposes no new problems. Again, because a template constructor can never be a copy constructor ([ANSI-CPP] 12.8), a specialization of our generic constructor is not enough. Instead we have to define it in the usual way:

```
inline Value::Value(const Value &val) : fp_DELETE(NULL) {
  if (val.fp_CLONE != NULL) {
    (val.*val.fp_CLONE)(this);
  }
  else fp_CLONE = NULL;
}
```

If the object used as argument for the constructor currently stores a value, we call its clone method with the address of the object to initialize. Else we set the method pointer `fp_CLONE` of the new `Value` object to NULL. It should be noticed here, that the default constructor defined in section 3 also has to be adopted. The two pointers to members introduced in the last sections have to be initialized to NULL if we want to achieve the desired copy-, assignment- and delete-semantics.

```
inline Value::Value() : fp_DELETE(NULL), fp_CLONE(NULL) {}
```

4.4 Runtime type information and printing

For polymorphic objects, C++ offers the possibility to query the exact dynamic type of an object at runtime. This can be done by using the builtin `typeid()` operator.

It would be convenient if this also would be possible for chameleon objects, since they also have the ability to dynamically change their internal type. Unfortunately it is not possible to overload the `typeid()` operator so we have to come up with an own solution.

According to the `typeid()` operator we define the following method `typeId()`:

```
const type_info& Value::typeId() const throw (bad_typeid&) {
  if (myType) return *myType;
  else throw bad_typeid();
}
```

In fact this method does nothing more than returning the type information of the value the object actually stores. Because the method is not parameterized the only possibility to find out this value is to store it as a private member of the `Value` object itself, which has to be initialized by the constructors or the assignment operators.

```
const type_info* myType;
```

Because type_info has only a private copy constructor and assignment operator ([ANSI-CPP] 18.5.1), it is not possible to copy type_info objects. Therefor, only pointers to such objects can be stored.

Another missing feature for chameleon objects is the inability to print them. As stated already, this is not straight forward because the Value class is not aware of the type of the object it stores. Again, the only solution is to store a pointer to a parameterized method in the object, which implicitly knows how to print the wrapped data.

```
typedef ostream&(Value::*PrintPointer)(ostream&) const;
PrintPointer fp_PRINT;
```

The constructors or assignment operators will set fp_PRINT accordingly to point to the right-typed print method:

```
ostream& Value::printValue(ostream& os) const {
  return os << const_cast<Value*>(this)->template value<T>();
}
```

Additionally, a global output operator can be defined, to simplify output. Note that the output operator has to be a friend of the Value class in order to access the private data member fp_PRINT.

```
ostream& operator<<(ostream& os, const Value& e) {
  if (e.fp_PRINT) return (e.*e.fp_PRINT)(os);
  else return (os << "nil");
}
```

4.5 The need for speed

At this point we have a fully working type safe wrapper class. Of course this functionality was bought with an increasing amount of memory and runtime overhead. The time overhead is caused mainly by the additional cost which result from querying the static data containers of the appropriate type. On the other hand we have a space overhead resulting from the introduced pointers to members. These overheads are especially severe, if the Value class will be used to wrap small data types like for example the builtin types int, double or char. On the other hand, wrapping an object of type string which itself already has a size of some hundred bytes, results only in a modest space overhead of some percent.

Nevertheless, there exists a possibility to improve both, space requirements and performance of the Value class. It was implemented in [Nseq], where also some performance tests can be found.

The idea behind this is to replace the static containers which hold the objects of the appropriate types. Instead, in every Value object a private void pointer will point to the corresponding data object.

Because all the conversions are still done in the parameterized operators of the Value class, type safety is still guaranteed. The solution is of course not as general as the one presented before, since we loose now the possibility of “storing” objects of different types in one Value object. On the other hand, we can omit a pointer to member function for deleting the object, since the builtin delete operator will free the space occupied by the object. The objects destructor however will not be called in this case, so this will work for classes with trivial constructors ([ANSI-CPP] 12.4:3) only. In general it is not wise to delete class objects through void pointers ([ANSI-CPP] 5.3.5:3) and it may still be useful to retain the fp_DELETE pointer.

The implementation of this version of the Value class can be found in the file Value_void.hpp at [SOURCE].

Another problem of the implementation presented so far is the fact that a Value object grows with every new pointer to member added to it. But from a logical point of view, these pointers don’t belong to every class object, but should be shared by all objects which hold a value of the same type. Exactly this behavior can be modeled by introducing a structure which holds all the desired pointers and members necessary for a Value object. In turn, the only member required by a Value object is a pointer to the corresponding structure.

This technique is equivalent to the one used for virtual function dispatch. In that case, every class object which contains at least one virtual function, will contain a pointer to a virtual function table. At runtime, all polymorphic objects of the same dynamic type will point at the same unique virtual function table. In our case, all Value objects which store a value of the same type will have a pointer to a unique structure which contains the appropriately typed methods for manipulating this value.

Because the Value class is not parameterized, this pointer must have a fixed type. So we declare a struct VTable as follows:

```
struct VTable {
  const type_info* myType;
  typedef void(Value::*FuncPointer)(void);
  FuncPointer fp_DELETE;
  typedef void(Value::*ClonePointer)(Value*) const;
  ClonePointer fp_CLONE;
  typedef ostream&(Value::*PrintPointer)(ostream&) const;
  PrintPointer fp_PRINT;
  VTable(const type_info* ti, FuncPointer fP, ClonePointer cP, PrintPointer pP)
    : myType(ti), fp_DELETE(fP), fp_CLONE(cP), fp_PRINT(pP) {}
};
```

From this struct we derive a parameterized one, which holds the members corresponding to the parameterizing type:

```
template<class T>
struct Spec_VTable : public VTable {
  Spec_VTable() : VTable(&typeid(T),
                  &(Value::template deleteValue<T>),
                  &(Value::template cloneValue<T>),
                  &(Value::template printValue<T>)) {}
};
```

Now we can remove all data members of Value and replace them by a single pointer to an VTable object. Further on, the constructors and assignment operators have to be updated to initialize this pointer properly and all calls of the methods defined in the VTable struct have to done through the new pointer. Again, a complete implementation can be found in the file Value_VTable.hpp at [SOURCE].

4.6 Compiler internals

At the time of the writing in June 1998, the code in this article could be compiled and tested only with version 2.38 of the EDG compiler front end for Linux [EDG], using egcs-1.0.3 and a patched version of libstdc++.2.8.1 compiled with the EDG frontend and egcs-1.0.3. The following compiler switches were used : "-x" to enable exception handling, "-B" to enable implicit inclusion of template definition files (not needed by the examples, but by the STL classes used) and "-tlocal" for local template instantiation.

In September 1998, version 1.1 of the egcs [EGCS] compiler was released, and it also was able to compile this code. The only change necessary was to replace the line

```
val->value<T>(const_cast<Value*>(this)->value<T>(), SET);
```

in the method cloneValue() of Value with the semantically identical but syntactically slightly different lines

```
T t = T();
val->value(const_cast<Value*>(this)->value(t, GET), SET);
```

since egcs still has some problems with explicit template argument specification ([ANSI-CPP] 14.8.1). Another possibility for this line accepted by egcs and EDG would be to write:

```
val->template value<T>(const_cast<Value*>(this)->template value<T>(), SET);
```

In the spring of 1999, IBM released version 4.0 of its VisualAge compiler, which also managed to translate the code.

Also notice that all section numbers in references to [ANSI-CPP] referred to in this article may be slightly different in other editions of the standard. The source code presented in this paper is available online from [SOURCE].

5 Conclusion

In this article we proposed a technique for building generic, but unparameterized, classes, which are able to store arbitrary typed objects and still maintain type safety. In addition to being useful in daily programming tasks, this example shows how an ordinary unparameterized class can change its "internal type" (i.e., the type of member data it holds) at runtime. We achieve this goal first by using generic pointers to template member functions, which only internally use their type information but externally preserve a constant signature, and second by using only one parametrized method which supplies the functionality of three basic functions through a dispatching mechanism and a static local variable. Furthermore, using operator overloading, these so-called Chameleon objects can be made transparent to the programmer.

The `Value` object is not aware of its internal type at any time, but it always has enough information to call the right functions at run time to properly access its data. During compilation, the compiler will detect and instantiate these functions for all types that the program stores in a Value object.

Thus we have something similar to the virtual function table, which is built up at compile time, but consulted for every function call at runtime. The difference in our case is that the data type that controls the dispatching can change during program execution.

This gives us a kind of "dynamic runtime polymorphism" not for function calls, as provided by inheritance and the virtual function mechanism, but for the data type of an object. This is achieved by combining the template mechanism, which is widely known as "static polymorhism", with a self managed, dynamic data structure.

6 Acknowledgment

First of all I want to thank the people at Edison Design Group for their compiler front end [EDG] , since it was the first– and for a long time the only– compiler that translated my code. Neither Microsoft VC++ nor g++ work at present time (October 1998). Furthermore I want to thank Prof. Küchlin (University of Tübingen) for supervising my master's thesis, Prof. Loos (University of Tübingen) and Prof. Musser (Rensselaer Polytechnic Institute) for supplying me with a copy of the new C++ standard, and Dr. Suworow, V. Kalinenko and J. Höfler from debis Systemhaus for their support. And last but not least, many thanks to my friend Roland Weiss for many discussions (not only about computer science) and for reviewing these pages.

References

[ANSI-CPP] ANSI/ISO Standard *"Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++"*
American National Standards Institute (ANSI), Nov. 1997

[EDG] EDG - The Edison Design Group at http://www.edg.com

[EGCS] EGCS : a new development branch of the gcc compiler at http://egcs.cygnus.com

[JGL] JGL - The Java Generic Library at http://www.objectspace.com
or http://www.javaworld.com/javaworld/jw-06-1997/jw-06-jgl.html

[Lee-Stepanov] Lee, Meng and Stepanov, Alexander A. *"The Standard Template Library"*
http://www.cs.rpi.edu/˜musser/doc.ps

[Musser-Saini] Musser, David R. and Saini, Atul *"STL Tutorial and Reference Guide"*
Adison-Wesley, 1996

[Musser-Stepanov] Musser, David R. and Stepanov, Alexander A. *"Generic Programming"*
Lecture Notes in Computer Science 358, Springer-Verlag 1989, pp. 13-25

[Nseq] Simonis, V. and Weiss, R. *"Heterogeneous, Nested STL Containers in C++"* Lecture Notes in Computer Science 1755, Springer-Verlag 1999, pp. 263-267, http://www-ca.informatik.uni-tuebingen.de/people/simonis/papers.htm

[Ser-Spec] Sun Microsystems (1998). *"Java Object Serialisation Specification"* Sun Microsystems, Revision 1.4.1, February 12, 1998 JDK 1.2

[SGI-STL] The SGI implementation of the STL at http://www.sgi.com/Technology/STL/

[SOURCE] The source code presented in this article can be found at: http://www-ca.informatik.uni-tuebingen.de/people/simonis/papers.htm