PME: Properties, Methods and Events

Borland Software Corp.
2003.09.09
N1384=02-0042

Introduction

Modern GUI frameworks have demonstrated the need for a more capable, dynamic object model. Such a model is not incompatible with C++, requiring only a small addition to its basic features. The name given to these additions is called "the PME model", for "Properties, Methods and Events".

A property is a conceptual attribute of an object that can be queried and modified at runtime. It differs from a regular data member in that the underlying value of the property might be computed rather than stored. Standard C++ requires the use of getter and setter methods to emulate this type of functionality, which are only useful at compile-time.

A method, as envisioned in the PME model, is not different from a C++ method. The only extension is the idea of "introspection", which allows an object's methods to be queried and invoked at run-time. Standard C++ requires pre-processing tools to achieve this.

The event part of the PME model refers to event handlers: a property of an object that can hold a pointer to one or more functions, to be called whenever a particular event occurs. The function called might be a member function on the same object, or a different object. And while argument sets may vary, the argument set for a particular event should always remain the same. Standard C++ requires the use of pre-processing tools to express this idea in a scalable fashion[1].

In order to allow Standard C++ to fully support the PME model, and hence provide language facilities for want of which many companies have labored long and hard, we propose a few changes to the standard C++ language. Some of these changes are prerequisite to more complex concepts, while others are only a matter of syntax and can be accepted or rejected without compromising the overall proposal.

Bound pointer to member

For a given argument set, a member function pointer for a particular instance requires that that instance—and its type—be known to all callers of the function. A "bound pointer to member" is a generalized member pointer that stores the type and pointer of that instance, making the member function later callable without knowledge of either.

Here is an example of calling a member function through a pointer, without using bound pointers:

[1] Such as Qt's signal/socket framework. Without such pre-processing tools, C++-only implementations, which rely on a separate helper class for every type/argument-list combination, grow to an unwieldy extent.

```
my_class x;

int foo(int (my_class::*bar)()) {
  int local = (x.*bar)();   // set 'local' to the value
                            // returned by the bar method
}
```

Note two things: '`foo`'s function signature makes reference to '`my_class`', and the call itself uses an object of type '`my_class`' to bind the call.

Here is the same example using a bound pointer:

```
int foo(int (::*bar)()) {
  int local = bar();
}
```

In this case, neither '`my_class`', nor any instances of type '`my_class`', are necessary to make the call. The logic in '`foo`' can be written independently of such knowledge.

Bound pointers allow for abstraction in interfaces that wish to call arbitrary member functions with a known argument set. It is possible, to a certain extent, to use templates to achieve this kind of "type-less" call, but this forces: 1) a new template to be created for every possible combination of function parameters, and 2) that each template be instantiated for every possible type.

Event handlers

Without bound pointers, code wishing to make a member function call through a pointer must know the parent type of the member function, and have a pointer to an instance of that type. This is always required, whether or not the parent type has anything to do with the logic of the calling code. One key idiom this causes problems with is event handlers.

In the case of event handlers, it is desirable for code to be able to call a set of functions whenever a particular event occurs. Often, the user would like member functions from several different objects to handle the event. However, because knowledge of the parent type is required, event handlers must be bound to a pointer to an object of base type, which all of the handling objects must derive from.

In the case of types which do not share a common base, this requirement can be alleviated somewhat by using smart pointer classes implemented as templates. However, this method has one major drawback: Function parameter lists cannot be specified via a template argument. Thus, not only is a template class necessary—to differentiate the parent type of the object when the member function is used—but in addition a *unique* template class must be generated for every possible combination of parameters:

```
template<typename Parent>
class event_i {
  typedef void (Parent::*func_ptr)(int);
  Parent * object;
  func_ptr handler;
public:
  explicit event_i(Parent *obj, func_ptr f) :
    object(obj), handler(f) { }
```

```
  void operator()(int i) {
    (obj->*handler)(i);
  }
};

event_i<SomeClass> event_hndlr((SomeClass *) SomeObject,
                               SomeClass::MyHandler);
event_hndlr(10);
```

As you can imagine, repeating this class declaration for every variation of '`event_i`' gets tiresome quickly, slows down compile time, and increases executable and debug information sizes unnecessarily.

Worse, the type of the event handler continues to be part of the signature, requring all event sending types to be recompiled. That problem can be addressed with another level of indirection:

```
class event_base_i {
public:
  virtual ~event_base_i() { }
  virtual void operator()(int i) = 0;
};

// modify event_i<> to derive from event_base_i

class event_sender {
private:
  event_base_i * event_handler;
  void send_event() {
    (*event_handler)(10);
  }
public:
  void set_handler(event_base_i *);
};

event_sender *s = init_from_somewhere();
s->set_handler(new event_i<SomeObject>(SomeObject,
                                       &SomeClass::MyHandler);
```

We've now doubled the number of classes that need to be defined, and introduced additional costs in terms of run-time execution and memory management.

Using bound pointers-to-member, however, the parent type is abstracted, removing the need for a template class. And since the event handler is now merely a call through a pointer, the argument types are only specified once at the time of the pointer's declaration.

```
void (::*handler)(int) = &SomeObject->MyHandler;
(*handler)(10);

class event_sender {
private:
  void (::*event_handler)(int);
```

```
  void send_event() {
    (*event_handler)(10);
  }
public:
  void set_handler(event_base_i *);
};

event_sender *s = init_from_somewhere();
s->set_handler(&SomeObject->MyHandler);
```

Rationale

While member pointers are very useful, they impose a strict penalty on code wishing to use them that does not require—or even want— knowledge of the parent type. Templates provide a way to accommodate this requirement in some cases, but with an associated cost that is unavoidable. It also results in algorithms that are cluttered with information that has only to do with the vagaries of C++, and not the logic of the code itself.

If bound pointers were added to the language, it would create a general, abstract mechanism for referencing member functions, permitting user code to be written independently of where the member function happens to be defined.

There is a cost for such convenience, of course, to the tune of double-width pointers. But the flexibility gained is a necessity in certain circumstances, most particularly event handlers. Additionally, this cost is going to be less than an implementation that is equally flexible and type-safe.

Properties

Properties are another feature whose usefulness has been demonstrated by nearly every GUI framework vendor. A property allows one to interact with an object in a natural, simple fashion, hiding the underlying complexity of implementation from the user. Properties can typically be queried at run-time, streamed into resource files, and manipulated using design tools that support dynamic loading of objects.

Currently, Every vendor who has embraced the facility of properties does so using declarative add-ons to the C++ language. These properties are very restrictive, syntactically speaking. They do not participate as first-class language entities with respect to templates, access adjustment, reference via a pointer-to-member, etc.

This section of our proposal offers a syntactic and semantic definition of properties that fits the basic design goals of C++:

- Only those who use properties should pay for them.
- They should not interfere with current syntax and semantics.
- Declaration and usage should be intuitive to C++ users.

With respect to these goals, we see the following aspects as either necessary or useful for defining true properties in the standard C++ language.

Basic syntax

The basic syntax of a property declaration is to declare a member with the special type '`property`', whose template-style argument is the underlying type of the property.

```
class foo {
public:
  property<int> bar;
};
```

Every property may also have attributes, which if present are declared along with the property, and use an aggregate-style syntax:

```
class foo {
  int get_value ();
  int _bar;

public:
  property<int> bar = {
    read  = get_value,
    write = _bar
  };
};
```

The two attributes currently defined are:

read The method called when a property is accessed, or the data member whose contents will be accessed. If a reader is specified but no writer, attempts to write the property at compile-time should result in a diagnostic, while attempts to write to it at run-time should throw the exception '`std::read_only_property`'.

write The method called when a property is modified, or the data member whose contents will be modified. If a writer is specified but no reader, attempts to read the property at compile-time should result in a diagnostic, while attempts to read from it at run-time should throw the exception '`std::write_only_property`'.

If neither attribute is specified, the default behavior is to define an unnamed member of the same type as the property's underlying type within the class, and both the "read" and "write" attributes of the property will reference this unnamed member.

Getter/setter signatures

There are two types of getters/setters possible, depending on whether the property is const or not. They are:

```
// the non-const case: getter returns by value since all modifications
// must be made through the setter

T (U::*getter)()
void (U::*setter)(const T&)

// the const case: there is no setter!

T (U::*getter)() const
```

Specialized diagnostics for properties

Along with each of these features comes the need for specialized diagnostics to help identify errors in property usage. The following additional diagnostics would make property declaration misuse much easier to debug:

- Attempts to use a reference type as the property's underlying type.
- Declaring a property outside of a class.
- Using a function with an improper signature as getter/setter.
- Using a default value of the wrong type.
- Masking like-named properties in a base class.
- The underlying property type may not be void or a reference.

A benefit of bound pointers: Property collections

Bound pointers-to-member can allow users to manage sets of property references, using collections of pointers to properties:

```
void foo(vector<property<int> (::*)>& bag) {
  // 'set_to_int' is a hypothetical functor that sets each property
  // to the integer value specified by the template argument list

  std::transform(bag.begin(), bag.end(), set_to_int<10>())
}
```

Enriched RTTI

In order to properly support "introspection"[1], it is necessary for users of a C++ object to have run-time access to information about its properties and methods. This information can be made accessible via specific enhancements to that object's RTTI.

"Enriched RTTI" will only be generated for class members within a new '`published`' access section. '`published`' is semantically identically to '`public`' in all respects, with the additional meaning that a class member occuring in a '`published`' section will have enriched RTTI information generated for that member.[2]

The extensions will consist of expanding the definition of '`type_info`' to include read-only access to these published details of the type, as well as extending the underlying binary information.

More details on this aspect of our proposal, as well as a sample implementation, are under development and should be ready by the time of the committee's meeting.

[1] The ability to determine information about an object's properties and methods at run-time.

[2] This scheme allows the "using" keyword to be used for publishing base-class members within a derived class.