#X3J16/96-0030 WG21/N0848

A Standard Adapter to Support Polymorphic Containers

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

There is no direct support for using member functions (object-oriented programming) together with standard algorithms (generic programming). I propose the addition of standard library binders, `mem_fct()` and `mem_fct_ref()` that allows member functions to be applied to sequences of pointers to class objects or sequences of class objects by standard algorithms.

1 Introduction

It is often claimed that STL doesn't support polymorphism. If true, that would be a serious problem. It is not. First of all, containers of pointers work just fine (see also Andrew Koenig's proposal relating to associative containers of pointers #96-0020/N0838), and those are the ones we care about when polymorphism (classical object-oriented programming) is the issue.

However, many standard algorithms apply non-member functions to objects. There are several ways of dealing with that − many of you will have a favorite. I would like to propose the simplest and most general that I have found as part of the standard.

Consider the classical example of applying `Shape::draw()` to every element of an array:

```
void rotate_all(vector<shape*>& v)
{
    for (int i = 0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

We would like to express this using the standard library algorithm `for_each()`. This cannot be done directly. Like all standard algorithms that takes a function as an argument, that function must be a non-member function. Thus, we must either provide a new algorithm that takes a member function or somehow provide a non-member function that does what what `Shape::draw()` does given a `Shape*`. For example:

```
void draw_shape(Shape* p) { p->draw(); }

void rotate_all2(vector<shape*>& v)
{
    for_each(s.begin(),s.end(),&draw_shape);
}
```

This is tedious, but it works. There may be other fundamental ways of tackling the problem, but I know of none that are both general and does no violence to the the concepts of the library.

What I propose is to ''mechanize'' the production of functions such as `draw_shape()` by using templates. For example:

```
void rotate_all3(vector<shape*>& v)
{
    for_each(s.begin(),s.end(),mem_fct(&Shape::draw));
}
```

The adapter `mem_fct()` takes a pointer to member and produces something that can be applied to a `Shape*`.

This solution has the property of being simple, apparently consisting of a single template function, introducing little overhead, and applies uniformly over the STL. The rest of this note explains how `mem_fct()` can be defined.

The reason for including `mem_fct()` into the standard is that otherwise every programmer who wants to use object-oriented techniques and the standard library will face this problem. I hope that will be essentially every C++ programmer will (at some time) use a combination of OOP and the standard library.

I fear that in the absence of a standard solution to this problem, the standard library algorithms (and possibly also the containers) will remain underused in favor for a variety of homebrew and commercial ‘‘true OO’’ solutions. Thus, the intersection of two of the most critical programming paradigms supported by C++ (object-oriented and generic programming) will be left unsupported by the standard.

2 What should `mem_fct()` Look Like?

Here is a first draft of the adaptor that does the trick:

```
template<class T> class Mem_fct {
    void (T::*pmf)();
public:
    explicit Mem_fct(void (T::*p)()) :pmf(p) {}
    void operator()(T* p) { (p->*pmf)(); }
};

template<class T> Mem_fct<T> mem_fct(void (T::*f)())
{
    return Mem_fct<T>(f);
}
```

It works (at least on the compiler I tried it on). It covers the case of a member function taking no arguments and returning no value.

2.1 Objects and Pointers

I expect `mem_fct()` to be used mostly with containers of pointers. However, there is an equivalent need to invoke member functions on sequences of objects.

The helper class `Mem_fct` is easily extended to handle both cases:

```
template<class T> class Mem_fct {
    void (T::*pmf)();
public:
    explicit Mem_fct(void (T::*p)()) :pmf(p) {}
    void operator()(T* p) { (p->*pmf)(); }
    void operator()(T& p) { (p.*pmf)(); }
};
```

However, if we did that, the result would not fit with other function objects and binders (§2.4) so I introduce a separate class to deal with objects (as opposed to pointers to objects):

```
template<class T> class Mem_fct_ref {
    void (T::*pmf)();
public:
    explicit Mem_fct(void (T::*p)()) :pmf(p) {}
    void operator()(T& p) { (p.*pmf)(); }
};

template<class T> Mem_fct_ref<T> mem_fct_ref(void (T::*f)())
{
    return Mem_fct<T>(f);
}
```

We can now write:

```
class X {
public:
    void m();
};

void f(vector<X>& v, list<X*>& p)
{
    for_each(v.begin(), v.end(), mem_fct_ref(&X::m));
    for_each(p.begin(), p.end(), mem_fct(&X::m));
}
```

Because the primary interaction between standard algorithms and member functions is expected to be containers of pointers, I chose the shortest and most obvious name for the version that applies to pointers.

2.2 Arguments

Overloading makes the definition of a version that takes arguments easy:

```
template<class T, class A> class Mem_fct1 {
    void (T::*pmf)(A);
public:
    explicit Mem_fct1(void (T::*p)(A)) :pmf(p) {}
    void operator()(T* p, A x) { (p->*pmf)(x); }
};

template<class T, class A> Mem_fct1<T,A> mem_fct(void (T::*f)(A))
{
    return Mem_fct1<T,A>(f);
}
```

Examples of member functions taking more than one argument are plentiful. However, I see no need for a standard `Mem_fct2`, `Mem_fct3`, etc., The standard library algorithms take functions of no arguments (that is not non-static members), unary functions (non-static member functions with no arguments), or binary functions (non-static member function taking one argument). Where needed, binders for more arguments are easily defined given the pattern of `Mem_fct1`.

2.3 Return Types

I would prefer the definition of `Mem_fct` to be:

```
template<class S, class T> class Mem_fct {
    S (T::*pmf)();
public:
    explicit Mem_fct(S (T::*p)()) :pmf(p) {}
    S operator()(T* p) { return (p->*pmf)(); }
};

template<class S, class T> Mem_fct<S,T> mem_fct(S (T::*f)())
{
    return Mem_fct<T>(f);
}
```

That way, return values are handled in the obvious way. However, this definition does not handle a `void` member function under the current language rules.

If we accept my proposal #X3J16/96-0031,WG21/N0849, this example will handle `void` member functions correctly. If not, we need to add a (partial) specialization:

```
template<class S, class T> class<void,T> Mem_fct {
    void (T::*pmf)();
public:
    explicit Mem_fct(void (T::*p)()) :pmf(p) {}
    void operator()(T* p) { (p->*pmf)(); }
};

template<class S, class T> Mem_fct<void,T> mem_fct(void (T::*f)())
{
    return Mem_fct<void,T>(f);
}
```

I would prefer to rely on #X3J16/96-0031,WG21/N0849, but the user interface of `mem_fct()` is the same in either case.

2.4 Binders

The adaptor `ptr_fun()` (§20.3.7) allows functions to be used as predicates and to have their arguments bound by the binders (§20.3.5). The class generated from `Mem_fct` as defined above can be called like a function, but a pointer to it cannot be passed as an argument to `ptr_fun()`. Deriving the versions of `Mem_fct` from `unary_function` and `binary_function`, respectively, solves this problem.

`Mem_fct_ref` is handled similarly. Note that the need to interact properly with binders is the real reason why we need both `mem_fct()` and `mem_fct_ref()` (§2.1).

3 Working Paper Additions

This proposal is a pure extension. The text is probably best added to §20.3.7 [lib.function.pointer.adaptors]:

```
template<class S, class T> class Mem_fct
    : public unary_function<T*,S> {
public:
    explicit Mem_fct(S (T::*p)());
    S operator()(T* p);
};
```

`Mem_fct` calls the member function it was initialized with given an pointer or a reference argument.

```
template<class S, class T, class A> class Mem_fct1
    : public binary_function<T*,A,S> {
public:
    explicit Mem_fct1(void (T::*p)(A));
    S operator()(T* p, A x);
};
```

`Mem_fct1` calls the member function it was initialized with given an pointer or a reference argument and an additional argument of the appropriate type.

```
template<class S, class T> Mem_fct<S,T>
    mem_fct(S (T::*f)());

template<class S, class T, class A>
    Mem_fct1<S,T,A> mem_fct(S (T::*f)(A));
```

`mem_fct(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any) .

```
template<class S, class T> class Mem_fct_ref
    : public unary_function<T,S> {
public:
    explicit Mem_fct_ref(S (T::*p)());
    S operator()(T* p);
};
```

`Mem_fct_ref` calls the member function it was initialized with given an pointer or a reference argument.

```
template<class S, class T, class A> class Mem_fct1_ref
    : public binary_function<T,A,S> {
public:
    explicit Mem_fct1_ref(void (T::*p)(A));
    S operator()(T* p, A x);
};
```

`Mem_fct1_ref` calls the member function it was initialized with given an pointer or a reference argument and an additional argument of the appropriate type.

```
template<class S, class T> Mem_fct_ref<S,T>
    mem_fct_ref(S (T::*f)());

template<class S, class T, class A>
    Mem_fct1_ref<S,T,A> mem_fct_ref(S (T::*f)(A));
```

`mem_fct_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any) .

4 Acknowledgements

Matt Austern, Andrew Koenig, and Alex Stepanov made constructive comments on this proposal.