The C++ Handle Template Adaptors

`IceUtil` provides adaptors that support use of smart pointers with STL algorithms. Each template function returns a corresponding function object that is for use by an STL algorithm. The adaptors are defined in the header `IceUtil/Functional.h`.

Here is a list of the adaptors:

```
memFun
memFun1
voidMemFun
voidMemFun1

secondMemFun
secondMemFun1
secondVoidMemFun
secondVoidMemFun1

constMemFun
constMemFun1
constVoidMemFun
constVoidMemFun1

secondConstMemFun
secondConstMemFun1
secondConstVoidMemFun
secondConstVoidMemFun1
```

As you can see, the adaptors are in two groups. The first group operates on non-const smart pointers, whereas the second group operates on `const` smart pointers (for example, on smart pointers declared as `const MyClassPtr`).

Each group is further divided into two sub-groups. The adaptors in the first group operate on the target of a smart pointer, whereas the `second<`*name*`>` adapters operate on the second element of a pair, where that element is a smart pointer.

Each of the four sub-groups contains four adaptors:

`memFun`

This adaptor is used for member functions that return a value and do not accept an argument. For example:

C++

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    int getVal() { return _i; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(42));
mcp.push_back(new MyClass(99));

transform(mcp.begin(), mcp.end(),
          ostream_iterator<int>(cout, " "),
          IceUtil::memFun(&MyClass::getVal));
cout << endl;
```

This code invokes the member function `getVal` on each instance that is pointed at by smart pointers in the vector `mcp` and prints the return value of `getVal` on `cout`, separated by spaces. The output from this code is:

```
42 99
```

memFun1

This adaptor is used for member functions that return a value and accept a single argument. For example:

C++

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    int plus(int v) { return _i + v; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

int A[3] = { 5, 7, 9 };
transform(mcp.begin(), mcp.end(), A,
          ostream_iterator<int>(cout, " "),
          IceUtil::memFun1(&MyClass::plus));
cout << endl;
```

This code invokes the member function `plus` on each instance that is pointed at by smart pointers in the vector `mcp` and prints the return value of a call to `plus` on `cout`, separated by spaces. The calls to `plus` are successively passed the values stored in the array `A`. The output from this code is:

```
7 11 15
```

voidMemFun

This adaptor is used for member functions that do not return a value and do not accept an argument. For example:

C++

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    void print() { cout << _i << endl; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

for_each(mcp.begin(), mcp.end(), IceUtil::voidMemFun(&MyClass::print));
```

This code invokes the member function `print` on each instance that is pointed at by smart pointers in the vector `mcp`. The output from this code is:

```
2
4
6
```

voidMemFun1

This adaptor is used for member functions that do not return a value and accept a single argument. For example:

C++

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    void printPlus(int v) { cout << _i + v << endl; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

for_each(
    mcp.begin(), mcp.end(),
    bind2nd(IceUtil::voidMemFun1(&MyClass::printPlus), 3));
```

This code invokes the member function `printPlus` on each instance that is pointed at by smart pointers in the vector `mcp`. The output from this code is:

```
5
7
9
```

As mentioned earlier, the `second<`*name*`>` versions of the adaptors operate on the second element of a `std::pair<T1, T2>`, where `T2` must be a smart pointer. Most commonly, these adaptors are used to apply an algorithm to each lookup value of a map or multi-map. Here is an example:

C++

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    int plus(int v) { return _i + v; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

map<string, MyClassPtr> m;
m["two"] = new MyClass(2);
m["four"] = new MyClass(4);
m["six"] = new MyClass(6);

int A[3] = { 5, 7, 9 };
transform(
    m.begin(), m.end(), A,
    ostream_iterator<int>(cout, " "),
    IceUtil::secondMemFun1<int, string, MyClass>(&MyClass::plus));
```

This code invokes the `plus` member function on the class instance denoted by the `second` smart pointer member of each pair in the dictionary `m`. The output from this code is:

```
9 13 11
```

Note that `secondMemFun1` is a template that requires three arguments: the return type of the member function to be invoked, the key type of the dictionary, and the type of the class that is pointed at by the smart pointer.

In general, the `second`<*name*> adaptors require the following template arguments:

C++

```
secondMemFun<R, K, T>
secondMemFun1<R, K, T>
secondVoidMemFun<K, T>
secondVoidMemFun<K, T>
```

where `R` is the return type of the member function, `K` is the type of the first member of the pair, and `T` is the class that contains the member function.

See Also

- The C++ Handle Template