The C++ Handle Template

`IceUtil::Handle` implements a smart reference-counted pointer type. Smart pointers are used to guarantee automatic deletion of heap-allocated class instances.

`Handle` is a template class with the following interface:

C++

```
template<typename T>
class Handle : /* ... */ {
public:

    typedef T element_type;

    T* _ptr;

    T* operator->() const;
    T& operator*() const;
    T* get() const;

    operator bool() const;

    void swap(HandleBase& other);

    Handle(T* p = 0);

    template<typename Y>
    Handle(const Handle<Y>& r);

    Handle(const Handle& r);

    ~Handle();

    Handle& operator=(T* p);

    template<typename Y>
    Handle& operator=(const Handle<Y>& r);

    Handle& operator=(const Handle& r);

    template<class Y>
    static Handle dynamicCast(const HandleBase<Y>& r);

    template<class Y>
    static Handle dynamicCast(Y* p);
};

template<typename T, typename U>
bool operator==(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator!=(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator<(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator<=(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator>(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator>=(const Handle<T>& lhs, const Handle<U>& rhs);
```

Note that the actual implementation is split into a base and a derived class. For simplicity, we show the combined interface here. If you want to see the full implementation detail, it can be found in `IceUtil/Handle.h`.

The template argument must be a class that derives from `Shared` or `SimpleShared` (or that implements reference counting with the same interface as these classes).

This is quite a large interface, but all it really does is to faithfully mimic the behavior of ordinary C++ class instance pointers. Rather than discussing each member function in detail, we provide a simple overview here that outlines the most important points. Please see the discussion of Ice objects for more examples of using smart pointers.

`element_type`

This type definition follows the STL convention of defining the element type with the fixed name `element_type` so you can use it for template programming or the definition of generic containers.

`_ptr`

This data member stores the pointer to the underlying heap-allocated class instance.

Constructors, copy constructor, and assignment operators

These member functions allow you to construct, copy, and assign smart pointers as if they were ordinary pointers. In particular, the constructor and assignment operator are overloaded to work with raw C++ class instance pointers, which results in the "adoption" of the raw pointer by the smart pointer. For example, the following code works correctly and does not cause a memory leak:

C++

```
typedef Handle<MyClass> MyClassPtr;

void foo(const MyClassPtr&);

// ...

foo(new MyClass); // OK, no leak here.
```

`operator->`, `operator*`, and `get`

The arrow and indirection operators allow you to apply the usual pointer syntax to smart pointers to use the target of a smart pointer. The `get` member function returns the class instance pointer to the underlying reference-counted class instance; the return value is the value of `_ptr`.

`dynamicCast`

This member function works exactly like a C++ `dynamic_cast`: it tests whether the argument supports the specified type and, if so, returns a non-null pointer; if the target does not support the specified type, it returns null.

> The reason for not using an actual `dynamic_cast` and using a `dynamicCast` function instead is that `dynamic_cast` only operates on pointer types, but `IceUtil::Handle` is a class.

For example:

C++

```
MyClassPtr p = ...;
MyOtherClassPtr o = ...;

o = MyOtherClassPtr::dynamicCast(p);
if (o)
{
    // o points at an instance of type MyOtherClass.
}
else
{
    // p points at something that is
    // not compatible with MyOtherClass.
}
```

Note that this example also illustrates the use of `operator bool`: when used in a boolean context, a smart pointer returns true if it is non-null and false otherwise.

Comparison operators: `==, !=, <, <=, >, >=`

The comparison operators for smart pointers delegate to the operators of the underlying class, therefore the author of the reference-counted class defines the semantics of smart pointer comparison. For example, in the case of Slice classes, the base class `Ice::Object` implements the comparison operators in terms of pointer addresses. On the other hand, the base class for Slice proxies implements comparison using value semantics.

See Also

- Smart Pointers for Classes
- C++ Mapping for Interfaces
- The C++ Shared and SimpleShared Classes