Smart Pointers for Classes

On this page:

Automatic Memory Management with Smart Pointers

A recurring theme for C++ programmers is the need to deal with memory allocations and deallocations in their programs. The difficulty of doing so is well known: in the face of exceptions, multiple return paths from functions, and callee-allocated memory that must be deallocated by the caller, it can be extremely difficult to ensure that a program does not leak resources. This is particularly important in multi-threaded programs: if you do not rigorously track ownership of dynamic memory, a thread may delete memory that is still used by another thread, usually with disastrous consequences.

To alleviate this problem, Ice provides smart pointers for classes. These smart pointers use reference counting to keep track of each class instance and, when the last reference to a class instance disappears, automatically delete the instance.

Smart pointer classes are an example of the *RAII* (*Resource Acquisition Is Initialization*) idiom [1].

Smart pointers are generated by the Slice compiler for each class type. For a Slice class `<class-name>`, the compiler generates a C++ smart pointer called `<class-name>Ptr`. Rather than showing all the details of the generated class, here is the basic usage pattern: whenever you allocate a class instance on the heap, you simply assign the pointer returned from `new` to a smart pointer for the class. Thereafter, memory management is automatic and the class instance is deleted once the last smart pointer for it goes out of scope:

C++

```
{                                       // Open scope
    TimeOfDayPtr tod = new TimeOfDayI;  // Allocate instance
    // Initialize...
    tod->hour = 18;
    tod->minute = 11;
    tod->second = 15;
    // ...
}                                       // No memory leak here!
```

As you can see, you use `operator->` to access the members of the class via its smart pointer. When the `tod` smart pointer goes out of scope, its destructor runs and, in turn, the destructor takes care of calling `delete` on the underlying class instance, so no memory is leaked.

A smart pointer performs reference counting of its underlying class instance:

- The constructor of a class sets its reference count to zero.
- Initializing a smart pointer with a dynamically-allocated class instance causes the smart pointer to increment the reference count of the instance by one.
- Copy-constructing a smart pointer increments the reference count of the instance by one.
- Assigning one smart pointer to another increments the target's reference count and decrements the source's reference count. (Self-assignment is safe.)
- The destructor of a smart pointer decrements the reference count by one and calls `delete` on its class instance if the reference count drops to zero.

Suppose that we default-construct a smart pointer as follows:

C++

```
TimeOfDayPtr tod;
```

This creates a smart pointer with an internal null pointer.

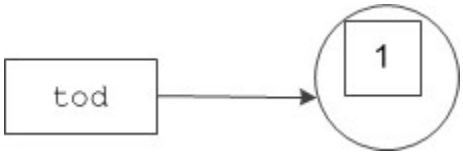
Newly initialized smart pointer.

Constructing a class instance creates that instance with a reference count of zero; the assignment to the smart pointer causes the smart pointer to increment the instance's reference count:

C++

```
tod = new TimeOfDayI;   // Refcount == 1
```

The resulting situation is shown below:



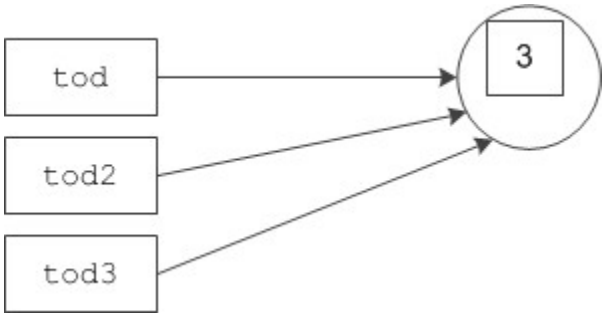
Initialized smart pointer.

Assigning or copy-constructing a smart pointer assigns and copy-constructs the smart pointer (not the underlying instance) and increments the reference count of the instance:

C++

```
TimeOfDayPtr tod2(tod); // Copy-construct tod2
TimeOfDayPtr tod3;
tod3 = tod;             // Assign to tod3
```

Here is the situation after executing these statements:



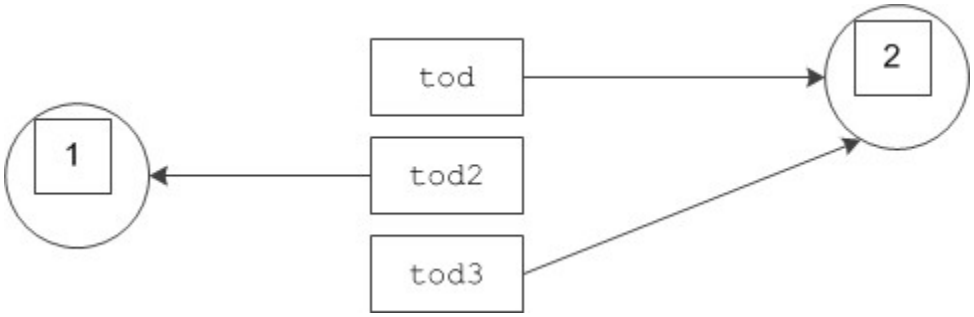
Three smart pointers pointing at the same class instance.

Continuing the example, we can construct a second class instance and assign it to one of the original smart pointers, `tod2`:

C++

```
tod2 = new TimeOfDayI;
```

This decrements the reference count of the instance originally denoted by `tod2` and increments the reference count of the instance that is assigned to `tod2`. The resulting situation becomes the following:



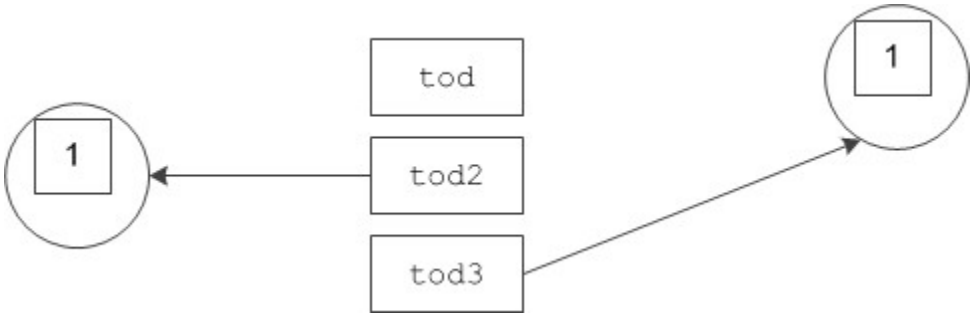
Three smart pointers and two instances.

You can clear a smart pointer by assigning zero to it:

C++

```
tod = 0;        // Clear handle
```

As you would expect, this decrements the reference count of the instance, as shown here:



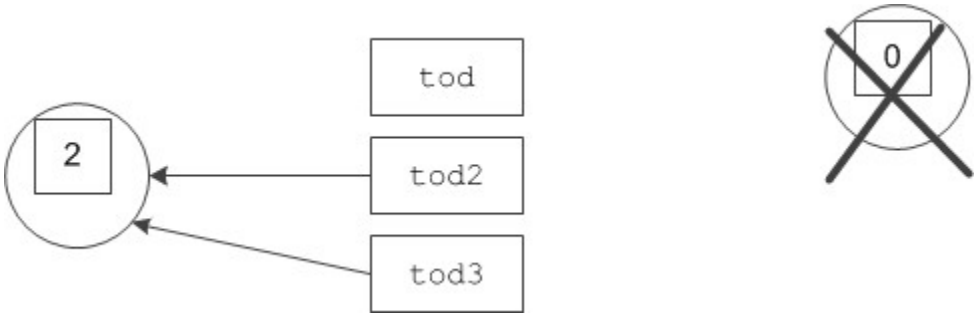
Decremented reference count after clearing a smart pointer.

If a smart pointer goes out of scope, is cleared, or has a new instance assigned to it, the smart pointer decrements the reference count of its instance; if the reference count drops to zero, the smart pointer calls `delete` to deallocate the instance. The following code snippet deallocates the instance on the right by assigning `tod2` to `tod3`:

C++

```
tod3 = tod2;
```

This results in the following situation:



Deallocation of an instance with a reference count of zero.

Copying and Assignment of Classes

Classes have a default memberwise copy constructor and assignment operator, so you can copy and assign class instances:

C++

```
TimeOfDayPtr tod = new TimeOfDayI(2, 3, 4); // Create instance
TimeOfDayPtr tod2 = new TimeOfDayI(*tod);   // Copy instance

TimeOfDayPtr tod3 = new TimeOfDayI;
*tod3 = *tod;                               // Assign instance
```

Copying and assignment in this manner performs a memberwise shallow copy or assignment, that is, the source members are copied into the target members; if a class contains class members (which are mapped as smart pointers), what is copied or assigned is the smart pointer, not the target of the smart pointer.

To illustrate this, consider the following Slice definitions:

Slice

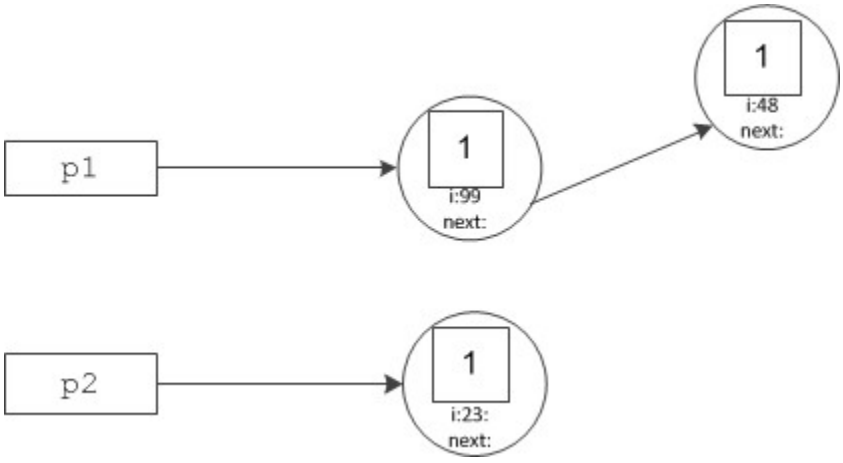
```
class Node {
    int i;
    Node next;
};
```

Assume that we initialize two instances of type `Node` as follows:

C++

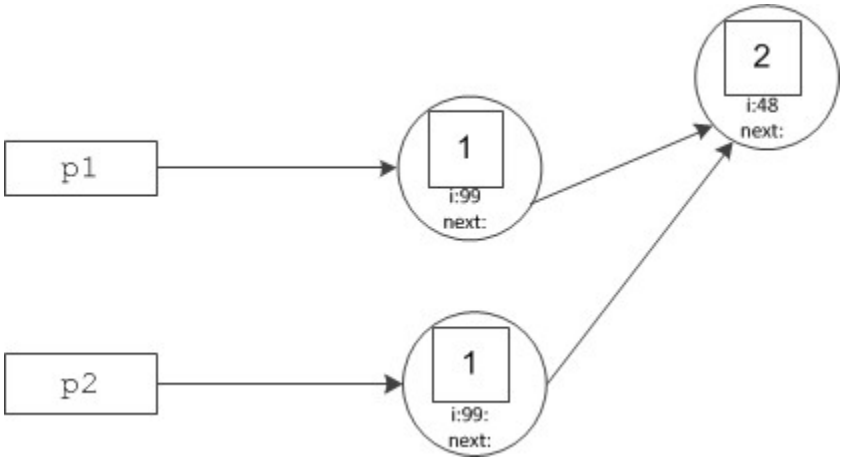
```
NodePtr p1 = new Node(99, new Node(48, 0));
NodePtr p2 = new Node(23, 0);

// ...

*p2 = *p1; // Assignment
```

After executing the first two statements, we have the situation shown below:



Class instances prior to assignment.

After executing the assignment statement, we end up with this result:



Class instances after assignment.

Note that copying and assignment also works for the implementation of abstract classes, such as our `TimeOfDayI` class, for example:

C++

```
class TimeOfDayI;

typedef IceUtil::Handle<TimeOfDayI> TimeOfDayIPtr;

class TimeOfDayI : virtual public TimeOfDay {
    // As before...
};
```

The default copy constructor and assignment operator will perform a memberwise copy or assignment of your implementation class:

C++

```
TimeOfDayIPtr tod1 = new TimeOfDayI;
TimeOfDayIPtr tod2 = new TimeOfDayI(*tod1);     // Make copy
```

Of course, if your implementation class contains raw pointers (for which a memberwise copy would almost certainly be inappropriate), you must implement your own copy constructor and assignment operator that take the appropriate action (and probably call the base copy constructor or assignment operator to take care of the base part).

Note that the preceding code uses `TimeOfDayIPtr` as a typedef for `IceUtil::Handle<TimeOfDayI>`. This class is a template that contains the smart pointer implementation. If you want smart pointers for the implementation of an abstract class, you must define a smart pointer type as illustrated by this type definition.

Copying and assignment of classes also works correctly for derived classes: you can assign a derived class to a base class, but not vice-versa; during such an assignment, the derived part of the source class is sliced, as per the usual C++ assignment semantics.

Polymorphic Copying of Classes

As shown in Inheritance from `Ice::Object`, the C++ mapping generates an `ice_clone` member function for every class:

C++

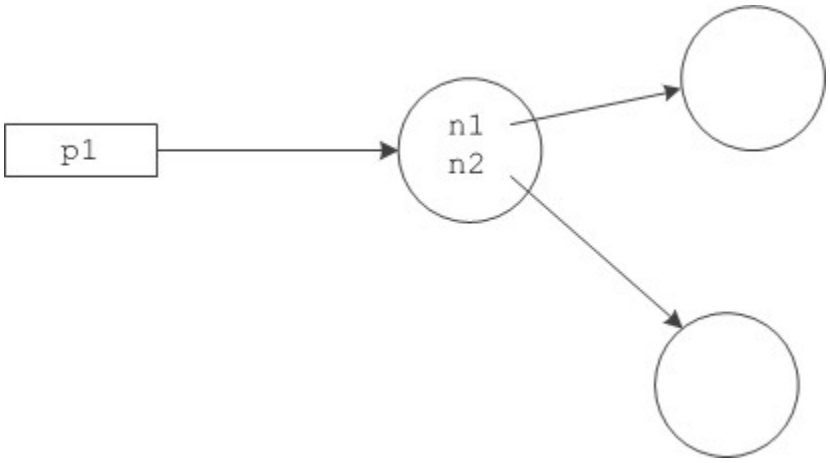
```
class TimeOfDay : virtual public Ice::Object {
public:
    // ...

    virtual Ice::ObjectPtr ice_clone() const;
};
```

This member function makes a polymorphic shallow copy of a class: members that are not class members are deep copied; all members that are class members are shallow copied. To illustrate, consider the following class definition:

Slice

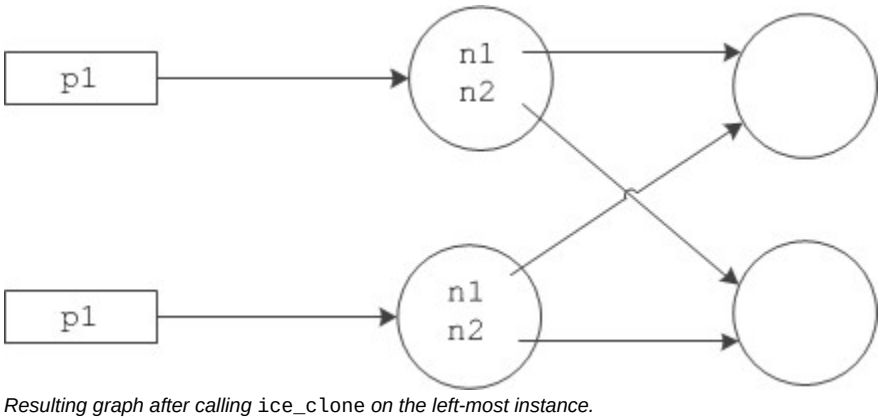
```
class Node {
    Node n1;
    Node n2;
};
```

Assume that we have an instance of this class, with the `n1` and `n2` members initialized to point at separate instances, as shown below:



A class instance pointing at two other instances.

If we call `ice_clone` on the instance on the left, we end up with this situation:



Resulting graph after calling `ice_clone` *on the left-most instance.*

As you can see, class members are shallow copied, that is, `ice_clone` makes a copy of the class instance on which it is invoked, but does not copy any class instances that are pointed at by the copied instance.

Note that `ice_clone` returns a value of type `Ice::ObjectPtr`, to avoid problems with compilers that do not support covariant return types. The generated `Ptr` classes contain a `dynamicCast` member that allows you to safely down-cast the return value of `ice_clone`. For example, the code to achieve the situation shown in the illustration above, looks as follows:

C++

```
NodePtr p1 = new Node(new Node, new Node);
NodePtr p2 = NodePtr::dynamicCast(p1->ice_clone());
```

`ice_clone` is generated by the Slice compiler for concrete classes (that is, classes that do not have operations). However, because classes with operations are abstract, the generated `ice_clone` for abstract classes cannot know how to instantiate an instance of the derived concrete class (because the name of the derived class is not known). This means that, for abstract classes, the generated `ice_clone` throws a `CloneNotImplementedException`.

If you want to clone the implementation of an abstract class, you must override the virtual `ice_clone` member in your concrete implementation class. For example:

C++

```
class TimeOfDayI : public TimeOfDay {
public:
    virtual Ice::ObjectPtr ice_clone() const
    {
        return new TimeOfDayI(*this);
    }
};
```

Null Smart Pointers

A null smart pointer contains a null C++ pointer to its underlying instance. This means that if you attempt to dereference a null smart pointer, you get an `IceUtil::NullHandleException`:

C++

```
TimeOfDayPtr tod;               // Construct null handle

try {
    tod->minute = 0;            // Dereference null handle
    assert(false);              // Cannot get here
} catch (const IceUtil::NullHandleException&) {
    ; // OK, expected
} catch (...) {
    assert(false);              // Must get NullHandleException
}
```

Preventing Stack-Allocation of Class Instances

The Ice C++ mapping expects class instances to be allocated on the heap. Allocating class instances on the stack or in static variables is pragmatically useless because all the Ice APIs expect parameters that are smart pointers, not class instances. This means that, to do anything with a stack-allocated class instance, you must initialize a smart pointer for the instance. However, doing so does not work because it inevitably leads to a crash:

C++

```
{                               // Enter scope
    TimeOfDayI t;               // Stack-allocated class instance
    TimeOfDayPtr todp;          // Handle for a TimeOfDay instance

    todp = &t;                  // Legal, but dangerous
    // ...
}                               // Leave scope, looming crash!
```

This goes badly wrong because, when `todp` goes out of scope, it decrements the reference count of the class to zero, which then calls `delete` on itself. However, the instance is stack-allocated and cannot be deleted, and we end up with undefined behavior (typically, a core dump).

The following attempt to fix this is also doomed to failure:

C++

```
{                               // Enter scope
    TimeOfDayI t;               // Stack-allocated class instance
    TimeOfDayPtr todp;          // Handle for a TimeOfDay instance

    todp = &t;                  // Legal, but dangerous
    // ...
    todp = 0;                   // Crash imminent!
}
```

This code attempts to circumvent the problem by clearing the smart pointer explicitly. However, doing so also causes the smart pointer to drop the reference count on the class to zero, so this code ends up with the same call to `delete` on the stack-allocated instance as the previous example.

The upshot of all this is: **never allocate a class instance on the stack or in a static variable**. The C++ mapping assumes that all class instances are allocated on the heap and no amount of coding trickery will change this.

You could abuse the `__setNoDelete` member to disable deallocation, but we strongly discourage you from doing this.

You can prevent allocation of class instances on the stack or in static variables by adding a protected destructor to your implementation of the class: if a class has a protected destructor, allocation must be made with `new`, and static or stack allocation causes a compile-time error. In addition, explicit calls to `delete` on a heap-allocated instance also are flagged at compile time.

Tip

You may want to habitually add a protected destructor to your implementation of abstract Slice classes to protect yourself from accidental heap allocation, as shown in Class Operations. (For Slice classes that do not have operations, the Slice compiler automatically adds a protected destructor.)

Smart Pointers and Constructors

Slice classes inherit their reference-counting behavior from the `IceUtil::Shared` class, which ensures that reference counts are managed in a thread-safe manner. When a stack-allocated smart pointer goes out of scope, the smart pointer invokes the `__decRef` function on the reference-counted object. Ignoring thread-safety issues, `__decRef` is implemented like this:

C++

```
void
IceUtil::Shared::__decRef()
{
    if (--_ref == 0 && !_noDelete)
        delete this;
}
```

In other words, when the smart pointer calls `__decRef` on the pointed-at instance and the reference count reaches zero (which happens when the last smart pointer for a class instance goes out of scope), the instance self-destructs by calling `delete this`.

However, as you can see, the instance self-destructs only if `_noDelete` is false (which it is by default, because the constructor initializes it to false). You can call `__setNoDelete(true)` to prevent this self-destruction and, later, call `__setNoDelete(false)` to enable it again. This is necessary if, for example, a class in its constructor needs to pass `this` to another function:

C++

```
void someFunction(const TimeOfDayPtr& t)
{
    // ...
}

TimeOfDayI::TimeOfDayI()
{
    someFunction(this); // Trouble looming here!
}
```

At first glance, this code looks innocuous enough. While `TimeOfDayI` is being constructed, it passes `this` to `someFunction`, which expects a smart pointer. The compiler constructs a temporary smart pointer at the point of call (because the smart pointer template has a single-argument constructor that accepts a pointer to a heap-allocated instance, so the constructor acts as a conversion function). However, this code fails badly. The `TimeOfDayI` instance is constructed with a statement such as:

C++

```
TimeOfDayPtr tod = new TimeOfDayI;
```

The constructor of `TimeOfDayI` is called by `operator new` and, when the constructor starts executing, the reference count of the instance is zero (because that is what the reference count is initialized to by the `Shared` base class of `TimeOfDayI`). When the constructor calls `someFunction`, the compiler creates a temporary smart pointer, which increments the reference count of the instance and, once `someFunction` completes, the compiler dutifully destroys that temporary smart pointer again. But, of course, that drops the reference count back to zero and causes the `TimeOfDayI` instance to self-destruct by calling `delete this`. The net effect is that the call to `new TimeOfDayI` returns a pointer to an already deleted object, which is likely to cause the program to crash.

To get around the problem, you can call `__setNoDelete`:

C++

```
TimeOfDayI::TimeOfDayI()
{
    __setNoDelete(true);
    someFunction(this);
    __setNoDelete(false);
}
```

The code disables self-destruction while `someFunction` uses its temporary smart pointer by calling `__setNoDelete(true)`. By doing this, the reference count of the instance is incremented before `someFunction` is called and decremented back to zero when `someFunction` completes without causing the object to self-destruct. The constructor then re-enables self-destruction by calling `__setNoDelete(false)` before returning, so the statement

C++

```
TimeOfDayPtr tod = new TimeOfDayI;
```

does the usual thing, namely to increment the reference count of the object to 1, despite the fact that a temporary smart pointer existed while the constructor ran.

In general, whenever a class constructor passes `this` to a function or another class that accepts a smart pointer, you must temporarily disable self-destruction.

Smart Pointers and Exception Safety

Smart pointers are exception safe: if an exception causes the thread of execution to leave a scope containing a stack-allocated smart pointer, the C++ run time ensures that the smart pointer's destructor is called, so no resource leaks can occur:

C++

```
{ // Enter scope...

    TimeOfDayPtr tod = new TimeOfDayI; // Allocate instance

    someFuncThatMightThrow();          // Might throw...

    // ...

} // No leak here, even if an exception is thrown
```

If an exception is thrown, the destructor of `tod` runs and ensures that it deallocates the underlying class instance.

There is one potential pitfall you must be aware of though: if a constructor of a base class throws an exception, and another class instance holds a smart pointer to the instance being constructed, you can end up with a double deallocation. You can use the `__setNoDelete` mechanism to temporarily disable self-destruction in this case, as described above.

Smart Pointers and Cycles

One thing you need to be aware of is the inability of reference counting to deal with cyclic dependencies. For example, consider the following simple self-referential class:

Slice

```
class Node {
    int val;
    Node next;
};
```

Intuitively, this class implements a linked list of nodes. As long as there are no cycles in the list of nodes, everything is fine, and our smart pointers will correctly deallocate the class instances. However, if we introduce a cycle, we have a problem:

C++

```
{                         // Open scope...

    NodePtr n1 = new Node; // N1 refcount == 1
    NodePtr n2 = new Node; // N2 refcount == 1
    n1->next = n2;         // N2 refcount == 2
    n2->next = n1;         // N1 refcount == 2

} // Destructors run:     // N2 refcount == 1,
                          // N1 refcount == 1, memory leak!
```

The nodes pointed to by `n1` and `n2` do not have names but, for the sake of illustration, let us assume that `n1`'s node is called N1, and `n2`'s node is called N2. When we allocate the N1 instance and assign it to `n1`, the smart pointer `n1` increments N1's reference count to 1. Similarly, N2's reference count is 1 after allocating the node and assigning it to `n2`. The next two statements set up a cyclic dependency between `n1` and `n2` by making their `next` pointers point at each other. This sets the reference count of both N1 and N2 to 2. When the enclosing scope closes, the destructor of `n2` is called first and decrements N2's reference count to 1, followed by the destructor of `n1`, which decrements N1's reference count to 1. The net effect is that neither reference count ever drops to zero, so both N1 and N2 are leaked.

Garbage Collection of Class Instances

The previous example illustrates a problem that is generic to using reference counts for deallocation: if a cyclic dependency exists anywhere in a graph (possibly via many intermediate nodes), all nodes in the cycle are leaked.

To avoid memory leaks due to such cycles, Ice for C++ contains a garbage collector. The collector identifies class instances that are part of one or more cycles but are no longer reachable from the program and deletes such instances:

- By default, garbage is collected whenever you destroy a communicator. This means that no memory is leaked when your program exits. (Of course, this assumes that you correctly destroy your communicators.)
- You can also explicitly run the garbage collector by calling `Ice::collectGarbage`. For example, the leak caused by the preceding example can be avoided as follows:

  C++

  ```
  {                         // Open scope...

      NodePtr n1 = new Node; // N1 refcount == 1
      NodePtr n2 = new Node; // N2 refcount == 1
      n1->next = n2;         // N1 refcount == 2
      n2->next = n1;         // N2 refcount == 2

  } // Destructors run:     // N2 refcount == 1,
                            // N1 refcount == 1

  Ice::collectGarbage();    // Deletes N1 and N2
  ```

  The call to `Ice::collectGarbage` deletes the no longer reachable instances N1 and N2 (as well as any other non-reachable instances that may have accumulated earlier).

- Deleting leaked memory with explicit calls to the garbage collector can be inconvenient because it requires polluting the code with calls to the collector. You can ask the Ice run time to run a garbage collection thread that periodically cleans up leaked memory by setting the property `Ice.GC.Interval` to a non-zero value. For example, setting `Ice.GC.Interval` to 5 causes the collector thread to run the garbage collector once every five seconds. You can trace the execution of the collector by setting `Ice.Trace.GC` to a non-zero value.

Note that the garbage collector is useful only if your program actually creates cyclic class graphs. There is no point in running the garbage collector in programs that do not create such cycles. (For this reason, the collector thread is disabled by default and runs only if you explicitly set `Ice.GC.Interval` to a non-zero value.)

Smart Pointer Comparison

As for proxy handles, class handles support the comparison operators ==, !=, and <. This allows you to use class handles in STL sorted containers. Be aware that, for smart pointers, object identity is not used for the comparison, because class instances do not have identity. Instead, these operators simply compare the memory address of the classes they point to. This means that `operator==` returns true only if two smart pointers point at the same physical class instance:

C++

```
// Create a class instance and initialize
//
TimeOfDayIPtr p1 = new TimeOfDayI;
p1->hour = 23;
p1->minute = 10;
p1->second = 18;

// Create another class instance with
// the same member values
//
TimeOfDayIPtr p2 = new TimeOfDayI;
p2->hour = 23;
p2->minute = 10;
p2->second = 18;

assert(p1 != p2);       // The two do not compare equal

TimeOfDayIPtr p3 = p1;  // Point at first class again

assert(p1 == p3);       // Now they compare equal
```

See Also

- Classes
- C++ Mapping for Classes
- Asynchronous Method Invocation (AMI) in C++
- The Server-Side main Function in C++
- Properties and Configuration
- The C++ Shared and SimpleShared Classes

References

1. Stroustrup, B. 1997. *The C++ Programming Language*. Reading, MA: Addison-Wesley.