Pointer Dereferencing vs. Conversion Operators

Pointer Dereferencing

The pointer dereferencing operator does exactly what it is expected to do – overloads the dereferencing of that particular class.

```
template <class T>
class Pointer
{
public:
     Pointer(T *rhs = NULL ) : pointee(rhs) {}
     bool operator<( const Pointer & rhs ) const
     {
          return *pointee < *rhs.pointee;
     }

     // Pointer dereferencing operator
     const T operator * () const
     {
          return *pointee;
     }

private:
     T* pointee;
};
```

Conversion Operator

What the CMSC 341 book describes as the pointer dereferencing operator is actually a **Conversion Operator** – which is basically an operator that implicitly (and automatically) converts one type to another. You never actually use the operator directly…it is instead indirectly used in the background. Let's look at some code:

```
template <class T>
class Pointer
{
public:
     Pointer(T *rhs = NULL ) : pointee(rhs) {}
     bool operator<( const Pointer & rhs ) const
     {
          return *pointee < *rhs.pointee;
     }

     // Conversion operator
     operator const T * () const
     {
```

```
            return pointee;
      }

private:
      T* pointee;
};
```

In the above class, the `operator const T * () const` is the conversion operator – which is converting an object of type `Pointer` into an object of type `const T`.

Differences

You should notice the difference is basically the position of the `operator` keyword. In the Conversion Operator – there is no return type, in fact the "name" of the operator is actually the entire signature: `operator const T *`. The return type is actually inferred from the name of the method. In the Pointer Dereferencing Operator – there is an (expected) explicit return value which is why this version actually returns the dereferenced pointee.

Using Both Operators

The interesting part is that both operators can actually be used in the same way, but the code is actually interpreted slightly differently. Let's look at an example:

```
template <class T>
ostream& operator <<(ostream &sout, Pointer<T> p)
{
      sout << *p << endl;
      return sout;
}
```

Here's an example of a chunk of code that could be interpreted differently depending on which operator you were using.

Using the Conversion Operator – it is actually converting p to an object of type T and then dereferencing it. Since T is a pointer to some other object, we return the pointee, it is dereferenced and we get the original object.

Using the Pointer Dereferencing Operator – p is actually dereferenced which means that we have to return an object, not the pointer to that object. This is why we dereference pointee before returning it.

If both operators are overloaded, then the dereferencing operator has higher precedence.

Additional References

You can feel free to search Google (or your favorite search engine) for information on Conversion Operators, but I've gone ahead and put together a few for your perusal…

ANSI/ISO C++ Professional Programmer's Handbook

http://www-f9.ijs.si/~matevz/docs/C++/ansi_cpp_progr_handbook/ch03/ch03.htm#Heading12

C++ Annotations Version 6.1.2
http://www.icce.rug.nl/documents/cplusplus/cplusplus09.html#l144

C/C++ Pointers
http://uvsc.freshsources.com/Operator_Overloading.ppt

Example Code

Below is some example code that I whipped up to help demonstrate the concepts of overloading the dereferencing and conversion operators. By commenting-out each of the overloaded operators (in turn) – you can see them both serve essentially the same purpose, just doing slightly different jobs.

```
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>

using namespace std;

class Student
{
public:
   Student(int i, string first, string last) : id(i), firstName(first),
lastName(last) {}
   int id;
   string firstName;
   string lastName;
   bool operator <(const Student &s)
   {
      return id < s.id;
   }
};

ostream& operator <<(ostream &sout, Student s)
{
   sout << setw(9) << s.firstName << setw(12) << s.lastName
        << setw(3) << s.id;
   return sout;
}

template <class T>
class Pointer
{
public:
   Pointer(T *rhs = NULL ) : pointee(rhs) {}
   bool operator<( const Pointer & rhs ) const
   {
      return *pointee < *rhs.pointee;
   }
```

```
   // Conversion operator
   operator const T * () const
   {
      return pointee;
   }

   // Pointer dereferencing operator
   const T operator * () const
   {
      return *pointee;
   }

private:
   T* pointee;
};

template <class T>
ostream& operator <<(ostream &sout, Pointer<T> p)
{
   sout << *p;
   return sout;
}

template <class T>
class PQueue
{
public:
   PQueue() {}
   void insert(T item)
   {
      // Percolate Up
      heap.push_back(item);
      int current = heap.size()-1;
      int parent;
      if (current % 2 == 0)   // if curr is even, right child
         parent = (current - 2) / 2;
      else                           // if curr is odd, left child
         parent = (current - 1) / 2;

      // while we're not at the root of the heap
      while (current != 0)
      {
         // if the current item is less than its parent, swap them
         if (heap[current] < heap[parent])
         {
            T temp = heap[current];
            heap[current] = heap[parent];
            heap[parent] = temp;
            current = parent;
            if (current % 2 == 0)   // if curr is even, right child
                  parent = (current - 2) / 2;
            else                           // if curr is odd, left child
                  parent = (current - 1) / 2;
         }
         else
            break;
      }
```

```
}

T deleteMin()
{
   // Percolate Down
   T retVal = heap[0];
   heap[0] = heap[size()-1];
   heap.pop_back();

   bool swapped = true;
   int current = 0;
   int left = current * 2 + 1;
   int right = current * 2 + 2;

   while (left < heap.size() && swapped)
   {
      swapped = false;
      // if current has one child
      if (right >= heap.size())
      {
         if (heap[left] < heap[current])
         {
               T temp = heap[current];
               heap[current] = heap[left];
               heap[left] = temp;
               current = left;
               swapped = true;
         }
      }
      // if current has two children, and right is lesser child
      else if (heap[right] < heap[left])
      {
         // if current is greater than right
         if (heap[right] < heap[current])
         {
               T temp = heap[current];
               heap[current] = heap[right];
               heap[right] = temp;
               current = right;
               swapped = true;
         }
      }
      // if current has two children, and left is lesser child
      else if (heap[left] < heap[right])
      {
         // if current is greater than left
         if (heap[left] < heap[current])
         {
               T temp = heap[current];
               heap[current] = heap[left];
               heap[left] = temp;
               current = left;
               swapped = true;
         }
      }

      left = current * 2 + 1;
```

```
         right = current * 2 + 2;
      }
      return retVal;
   }

   void print()
   {
      for (int i = 0; i < heap.size(); ++i)
         cout << heap[i] << endl;
   }

   int size()
   {
      return heap.size();
   }

private:
   vector<T> heap;
};

int main()
{
   PQueue<Pointer<Student> > queue;

   Student* s = new Student(3, "George", "Jetson");
   queue.insert(Pointer<Student>(s));

   s = new Student(4, "G.I.", "Joe");
   queue.insert(Pointer<Student>(s));

   s = new Student(1, "Barney", "Rubble");
   queue.insert(Pointer<Student>(s));

   s = new Student(5, "Pebbles", "Flintstone");
   queue.insert(Pointer<Student>(s));

   s = new Student(2, "Garfield", "The Cat");
   queue.insert(Pointer<Student>(s));

   s = new Student(6, "Odie", "The Dog");
   queue.insert(Pointer<Student>(s));

   s = new Student(0, "Marvin", "The Martian");
   queue.insert(Pointer<Student>(s));

   s = new Student(-1, "Bugs", "Bunny");
   queue.insert(Pointer<Student>(s));

   s = new Student(-2, "Daffy", "Duck");
   queue.insert(Pointer<Student>(s));

   s = new Student(-3, "Minnie", "Mouse");
   queue.insert(Pointer<Student>(s));

   s = new Student(-4, "Donald", "Duck");
   queue.insert(Pointer<Student>(s));
```

```
    cout << "*******  The Current Priority Queue *******" << endl;
    queue.print();

    cout << endl;
    cout << "*******  Printing DeleteMin for all items *******" << endl;
    while (queue.size() > 0)
    {
       Student temp = *(queue.deleteMin());
       cout << temp << endl;
    }

    return 0;
}
```