The IntArray class

Before continuing on to learn how to use templates for container classes we first examine a new class called **IntArray.** This class will perform like an array of integers but will have some extra attributes. From this class we will learn the following concepts that will be needed in container class templates.

- How to construct a container class so that it can be *called by value* (recall that arrays are only passed by reference/ponters).
- How to overload the subscript operator *operator[]*.
- How to use the *this* pointer to properly overload the assignment operator.


```
class IntArray
{private:
    int * _value;
    int _size;

  public:
  //Default  constructor
     IntArray(int n = 10):
        _value(new int[n]), _size(n){};
  //Copy constructor
     IntArray(const IntArray & A);
    ~IntArray() {delete [] _value;};

    int size() const { return _size; }
    IntArray & operator=(const IntArray &A);
    void display();

    // Access an element
    // check for index overflow
    int & operator[ ](int i)
      { if  (i < _size)
      return _value[i];
      else return _value[_size-1];};
};
```



```
//IntArray.cpp
#include "IntArray.h"
#include <iostream.h>

// Copy constructor
IntArray::IntArray(const IntArray & A):
      _value(new int[A._size]), _size(A._size)
{for (int i =0; i< _size; i++)
       _value[i] = A._value[i];
}

// Assignment operator
IntArray& IntArray::operator=(const IntArray &A)
{ if (this != &A)
   { delete [] _value; //erase current array
     _size = A._size;
     _value= new int[_size];

     for (int i =0; i< _size; i++)
       _value[i] = A._value[i];
   };
  return *this;
}
```



```
//Print out all items
void IntArray::display()
{for (int i =0; i< _size; i++)
  cout << i << " : " << _value[i] << endl;
}
```

IntArray has two private members:

```
int * _value;  //pointer to dynamically
               //  allocated integer array
int _size;     //size of array
```

The default constructor and constructor implementations are straightforward:

```
//Default  constructor
// allocate array (default = 10)
// and set _size
IntArray(int n = 10):
      _value(new int[n]), _size(n){};


//Destructor
//Deallocates space
~IntArray() {delete [] _value;};
```


Copy Constructor

The copy constructor implementation is also straightforward:

```
IntArray::IntArray(const IntArray & A):
      _value(new int[A._size]),
      _size(A._size)
{for (int i =0; i< _size; i++)
       _value[i] = A._value[i];
}
```


operator[]

Suppose that we write

```
IntArray A(5);
A[1] =3;  int i = A[1];
```

This does create **A** to *contain* a dynamically allocated array of 5 integers. But, **A** itself is *not* an array so **A[1]** is not well-defined.

To allow us to sidestep this problem, C++ permits *overloading the suscript operator:*

```
int & operator[ ](int i) {return _value[i];}
```

Writing **A[1]** is equivalent to writing **A.operator[ ](1)** which returns a *reference* to the integer, **_value[1]**.

i = A[1] is equivalent to **i = _value[1]**
A[1] =3 is equivalent to **_value[1] =3.**

This is *exactly* the behavior we want the subscript operator to display.

Note: C++ requires that **operator**[ ] be a member function of the class

To illustrate the versatility of *operator[]* we added some basic index checking to our actual code.

If the index is larger than the size of the array then the last element of the array is accessed.

```
int & operator[ ](int i)
    { if  (i < _size)
          return _value[i];
      else return _value[_size-1];
    }
```

As an example:

IntArray A(3); A[5] =1;

actually sets

A[2]=1.

The Assignment Operator

To assign **A=B** we first need to throw away the old array stored by **a** and then make a new copy of **B**'s array for it.

```
IntArray& IntArray::operator=(const IntArray &A)
{  delete [] _value; //erase current array
     _size = A._size;
     _value= new int[_size];

     for (int i =0; i< _size; i++)
       _value[i] = A._value[i];
  return *this;
}
```

In order to allow chaining of assignments, e.g., **A=B=C** to work properly the assignment operator should return an IntArray.

For both space and scope reasons the IntArray returned should be a *reference* variable. By returning ***this** we are returning a reference to the calling object (after the assignment has completed).

The assignment operator on the previous page has a bug. The command **A=A** will first implement

```
delete [] _value;
```

which will *erase* **A**'s array. This destroys all of **A**'s data so we can't copy the data back in the second step.

The way around this is to first check if the left and right hand side of the equality are the same, e.g., **A=A**. If they are, then do nothing. Only if they are different objects should it do the assignment.

```
IntArray& IntArray::operator=(const IntArray &A)
{ if (this != &A)
   { delete [] _value; //erase current array
     _size = A._size;
     _value= new int[_size];

     for (int i =0; i< _size; i++)
       _value[i] = A._value[i];
   };
  return *this;}
```



```
//CheckIntArray.cpp
#include <iostream.h>
#include "IntArray.h"
main()
{ IntArray A(3), B(4); int i;
  for(i=0; i<3; i++)   A[i] = 2*i;
  for(i=0; i<4; i++)   B[i] = 3*i;

  cout << "A\n-----\n";
  A.display(); cout << endl;

  cout << "B\n-----\n";
  B.display(); cout << endl;

  B=A;
  cout << "B: B=A\n--------\n";
  B.display(); cout << endl;

  B[4] = 20;
  cout << "B: B[4]=20\n-----------\n";
  B.display(); cout << endl;

  cout << "A\n-----\n";
  A.display(); cout << endl;}
```

```
A
-----
0 : 0
1 : 2
2 : 4

B
-----
0 : 0
1 : 3
2 : 6
3 : 9

B: B=A
--------
0 : 0
1 : 2
2 : 4

B: B[4]=20
-----------
0 : 0
1 : 2
2 : 20

A
-----
0 : 0
1 : 2
2 : 4
```



```
//CheckIntArray2.cpp
#include <iostream.h>
#include "IntArray.h"
void check(IntArray X, IntArray & Y)
{ X[0] =-1; Y[0] = -1;}

main()
{ IntArray A(3), B(4); int i;
  for(i=0; i<3; i++)   A[i] = 2*i;
  for(i=0; i<4; i++)   B[i] = 3*i;
  cout << "A\n-----\n";
  A.display(); cout << endl;

  cout << "B\n-----\n";
  B.display(); cout << endl;

  check(A,B);
  cout << "check(A,B);\n\n";

  cout << "A\n-----\n";
  A.display(); cout << endl;

  cout << "B\n-----\n";
  B.display(); cout << endl;}
```



```
A
-----
0 : 0
1 : 2
2 : 4

B
-----
0 : 0
1 : 3
2 : 6
3 : 9

check(A,B);

A
-----
0 : 0
1 : 2
2 : 4

B
-----
0 : -1
1 : 3
2 : 6
3 : 9
```