Vector Implementation Basics

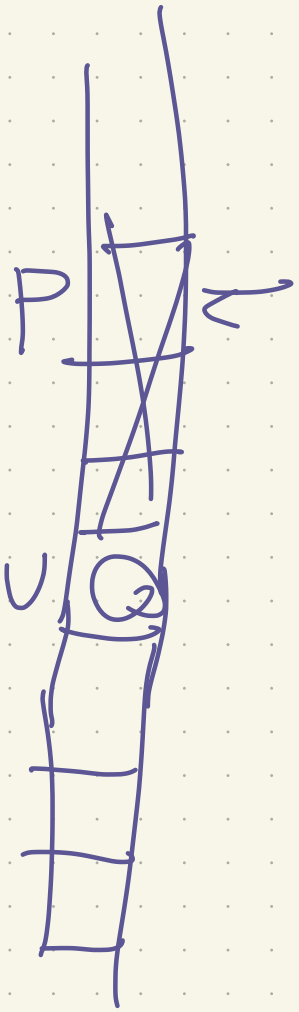
CMPT 225

Fall 2021

Lecture 4

Example Using a Simple "Vector" Class.

```
#include "IVector.h"
#include <iostream>
using namespace std;

int main( )
{
    const int N = 20;

    IVector v ; // Make an int Vector
    v.display(); // print its contents

    // Store N ints in the Vector
    for( int i = 0 ; i < N; ++i )
    {
        v.push_back( i );
    }

    // print the contents
    v.display();

    return 0;
}
```



p, q are int *

float y

x 5

p

q

q = p + 1

? *q

Implementation in IVector.h

```
#include <iostream>

class IVector
{
  public:
    // Constructor
    IVector( int initSize = 0 )
      : theSize{ initSize }, theCapacity{ initSize + 10 }
      {
         objects = new int[ theCapacity ];
      }

    // Destructor
    ~IVector( )
      { delete [ ] objects; }

    // Check for emptyness
    bool empty( ) const  { return size( ) == 0; }

    // Return size of list
    int size( ) const { return theSize; }

    // Access the element at a given index
    // This is the non-const version, so you can change the element.
    int & operator[]( int index )
    {
        return objects[ index ];
    }

    // Access the element at a given index
    // This is the const version, for accessing the value only
    const int & operator[]( int index ) const
    {
        return objects[ index ];
    }

    // Increase the capacity (i.e., array size)
    void reserve( int newCapacity )
    {
        if( newCapacity > theSize ){

           int *newArray = new int[ newCapacity ];
           for( int k = 0; k < theSize; ++k ){
                 newArray[ k ] = std::move( objects[ k ] );
           }

           theCapacity = newCapacity;
           std::swap( objects, newArray );
           delete [ ] newArray;
        }
    }

    // Add new element to end of the list
    void push_back( const int & x )
    {
        if( theSize == theCapacity ) reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    // remove last element from list
    void pop_back( )
    {
        --theSize;
    }

    // Print out the size and contents of the list
    void display() const
    {
        std::cout << "size=" << theSize << std::endl;

        for( int i = 0; i < theSize ; ++i ){
           std::cout << "[" << i << "]=" << objects[i] << std::endl;
        }
    }

  private:
    int theSize;
    int theCapacity;
    int * objects; // The array is of type int.
};

#endif
```

Implentation in IVector.h

```
#include <iostream>

class IVector
{
  public:
    // Constructor
    IVector( int initSize = 0 )
      : theSize{ initSize }, theCapacity{ initSize + 10 }
      {
         objects = new int[ theCapacity ];
      }

    .
    .
    .


  private:
    int theSize;
    int theCapacity;
    int * objects; // The array is of type int.
};
```

In C, C++ a variable of type int array is just a pointer to an int.

IVector.h

```
#include <iostream>

class IVector
{
  public:
    // Constructor
    IVector( int initSize = 0 )
      : theSize{ initSize }, theCapacity{ initSize + 10 }
      {
         objects = new int[ theCapacity ];
      }

    // Destructor
    ~IVector( )
      { delete [ ] objects; }

    // Check for emptyness
    bool empty( ) const  { return size( ) == 0; }

    // Return size of list
    int size( ) const { return theSize; }

    // Access the element at a given index
    // This is the non-const version, so you can change the element.
    int & operator[]( int index )
    {
        return objects[ index ];
    }

    // Access the element at a given index
    // This is the const version, for accessing the value only
    const int & operator[]( int index ) const
    {
        return objects[ index ];
    }
```

IVector.h

```
    // Increase the capacity (i.e., array size)
    void reserve( int newCapacity )
    {
        if( newCapacity > theSize ){

           int *newArray = new int[ newCapacity ];
           for( int k = 0; k < theSize; ++k ){
                 newArray[ k ] = std::move( objects[ k ] );
           }

           theCapacity = newCapacity;
           std::swap( objects, newArray );
           delete [ ] newArray;
        }
    }

    // Add new element to end of the list
    void push_back( const int & x )
    {
        if( theSize == theCapacity ) reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    // remove last element from list
    void pop_back( )
    {
        --theSize;
    }

    // Print out the size and contents of the list
    void display() const
    {
        std::cout << "size=" << theSize << std::endl;

        for( int i = 0; i < theSize ; ++i ){
           std::cout << "[" << i << "]=" << objects[i] << std::endl;
        }
    }

  private:
    int theSize;
    int theCapacity;
    int * objects; // The array is of type int.
};
```

objects[theSize] = x;
theSize = theSize + 1;

Templates

- Often, we have algorithms that will work on many data types, with few or no changes.
- In strongly typed languages, we need a way to produce "generic" code - code that can work on different types in different places.
- In C++, templates let us write generic code.
- A template function or class definition has a placeholder for one or more data types that is instantiated at compile time.
- The instantiation may be different at different places in the same code.

```
//
//  Test Program for Basic Stack Class
//
#include <iostream>   // for I/O facilities
using namespace std;
#include "MinimalStack.h" //  basic_stack declaration

int main (int argc, char * const argv[]) {
        cout << "\n\nMinimalStack Template Class Test Procedure - START\n\n";

        // Make some stacks, and verify that empty() says they are empty.

        MinimalStack<int> s1 ;
        MinimalStack<float> s2 ;
        cout << "s1.isEmpty() = " << s1.isEmpty() << "\n";
        cout << "s2.isEmpty() = " << s2.isEmpty() << "\n";

        // Put some things on them.
        cout << "s1.push( " << 1 << " )\n";
        s1.push(1);
        cout << "s1.push( " << 2 << " )\n";
        s1.push(2);
        cout << "s2.push( " << 1.5 << " )\n";
        s2.push(1.5);
        cout << "s2.push( " << 2.5 << " )\n";
        s2.push(2.5);

        // Verify that isEmpty() reports they are not empty,
        // and that the right things are on top.
        cout << "s1.isEmpty() = " << s1.isEmpty() << "\n";
        cout << "s1.top() = " << s1.top() << "\n";
        cout << "s2.isEmpty() = " << s2.isEmpty() << "\n";
        cout << "s2.top() = " << s2.top() << "\n";

        // Empty them, and verify that isEmpty() again reports they are empty.
        while( ! s1.isEmpty() ){
                cout << "s1.pop() = " << s1.pop() << "\n";
        }
        cout << "s1.isEmpty() = " << s1.isEmpty() << "\n";
        while( ! s2.isEmpty() ){
                cout << "s2.pop() = " << s2.pop() << "\n";
        }
```

```
#include "TVector.h"
#include <iostream>
using namespace std;


int main( )
{
    const int N = 20;

    TVector<int> v ; // Make an int Vector
    v.display(); // print its contents

    // Store N ints in the Vector
    for( int i = 0 ; i < N; ++i )
    {
        v.push_back( i );
    }

    // print the contents
    v.display();

    return 0;
}
```

TVector is a templated version of IVector

```
// This is a template class: Objecet is the type of
// the values or objects we are storing in the list.
template <typename Object>
class TVector
{
  public:
    TVector( int initSize = 0 )
      : theSize{ initSize }, theCapacity{ initSize + 10 }
      {
         objects = new Object[ theCapacity ];
      }

    ~TVector( )
      { delete [ ] objects; }

    bool empty( ) const  { return size( ) == 0; }
    int size( ) const { return theSize; }

    Object & operator[]( int index )
    {
        return objects[ index ];
    }

    const Object & operator[]( int index ) const
    {
        return objects[ index ];
    }

    void reserve( int newCapacity )
    {
        if( newCapacity > theSize ){

           Object *newArray = new Object[ newCapacity ];
           for( int k = 0; k < theSize; ++k ){
                 newArray[ k ] = std::move( objects[ k ] );
           }

           theCapacity = newCapacity;
           std::swap( objects, newArray );
           delete [ ] newArray;
        }
    }

    void push_back( const Object & x )
    {
        if( theSize == theCapacity ) reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    void pop_back( )
    {
        --theSize;
    }

    void display() const
    // Assumes that the cout will do something reasonable with
    // whatver type Objects is.
    {
        std::cout << "size=" << theSize << std::endl;

        for( int i = 0; i < theSize ; ++i ){
           std::cout << "[" << i << "]=" << objects[i] << std::endl;
        }
    }

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};

#endif
```

in IVector this is int.

TVector.h

```
// This is a template class: Objecet is the type of
// the values or objects we are storing in the list.
template <typename Object>
class TVector
{
  public:
    TVector( int initSize = 0 )
      : theSize{ initSize }, theCapacity{ initSize + 10 }
      {
        objects = new Object[ theCapacity ];
      }

    ~TVector( )
      { delete [ ] objects; }

    bool empty( ) const  { return size( ) == 0; }
    int size( ) const { return theSize; }

    Object & operator[]( int index )
```

```
    void display() const
    // Assumes that the cout will do something reasonable with
    // whatver type Objects is.
    {
        std::cout << "size=" << theSize << std::endl;

        for( int i = 0; i < theSize ; ++i ){
           std::cout << "[" << i << "]=" << objects[i] << std::endl;
        }
    }

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

```
    void reserve( int newCapacity )
    {
        if( newCapacity > theSize ){

           Object *newArray = new Object[ newCapacity ];
           for( int k = 0; k < theSize; ++k ){
                 newArray[ k ] = std::move( objects[ k ] );
           }

           theCapacity = newCapacity;
           std::swap( objects, newArray );
           delete [ ] newArray;
        }
    }

    void push_back( const Object & x )
    {
        if( theSize == theCapacity ) reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    void pop_back( )
    {
        --theSize;
    }

    void display() const
    // Assumes that the cout will do something reasonable with
    // whatver type Objects is.
    {
        std::cout << "size=" << theSize << std::endl;

        for( int i = 0; i < theSize ; ++i ){
           std::cout << "[" << i << "]=" << objects[i] << std::endl;
        }
    }

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

Vector.h

```
template <typename Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 )
      : theSize{ initSize }, theCapacity{ initSize + SPARE_CAPACITY }
      { objects = new Object[ theCapacity ]; }

    Vector( const Vector & rhs )
      : theSize{ rhs.theSize }, theCapacity{ rhs.theCapacity }, objects{ nullptr }
    {
        objects = new Object[ theCapacity ];
        for( int k = 0; k < theSize; ++k )
            objects[ k ] = rhs.objects[ k ];
    }

    Vector & operator= ( const Vector & rhs )
    {
        Vector copy = rhs;
        std::swap( *this, copy );
        return *this;
    }

    ~Vector( )
      { delete [ ] objects; }

    Vector( Vector && rhs )
      : theSize{ rhs.theSize }, theCapacity{ rhs.theCapacity }, objects{ rhs.objects }
    {
        rhs.objects = nullptr;
        rhs.theSize = 0;
        rhs.theCapacity = 0;
    }

    Vector & operator= ( Vector && rhs )
    {
        std::swap( theSize, rhs.theSize );
        std::swap( theCapacity, rhs.theCapacity );
        std::swap( objects, rhs.objects );
```

```
    // Stacky stuff
    void push_back( Object && x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = std::move( x );
    }

    void pop_back( )
    {
        if( empty( ) )
            throw UnderflowException{ };
        --theSize;
    }

    const Object & back ( ) const
    {
        if( empty( ) )
            throw UnderflowException{ };
        return objects[ theSize - 1 ];
    }

    // Iterator stuff: not bounds checked
    typedef Object * iterator;
    typedef const Object * const_iterator;

    iterator begin( )
      { return &objects[ 0 ]; }
    const_iterator begin( ) const
      { return &objects[ 0 ]; }
    iterator end( )
      { return &objects[ size( ) ]; }
    const_iterator end( ) const
      { return &objects[ size( ) ]; }

    static const int SPARE_CAPACITY = 2;

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

TestVector.cpp

```
#include "Vector.h"
#include <iostream>
#include <algorithm>
using namespace std;

void print( const Vector<Vector<int>> arr )
{
    int N = arr.size( );

    for( int i = 0; i < N; ++i )
    {
        cout << "arr[" << i << "]:";

        for( int j = 0; j < arr[ i ].size( ); ++j )
            cout << " " << arr[ i ][ j ];

        cout << endl;
    }
}

class CompareVector
{
public:
    bool operator() ( const Vector<int> & lhs, const Vector<int> & rhs ) const
    { return lhs.size( ) < rhs.size( ); }
};

int main( )
{
    const int N = 20;
    Vector<Vector<int>> arr( N );
    Vector<int> v;

    for( int i = N - 1; i > 0; --i )
    {
        v.push_back( i );
        arr[ i ] = v;
    }

    print( arr );

    clock_t start = clock( );
    std::sort( begin( arr ), end( arr ), CompareVector{ } );
    clock_t end = clock( );
    cout << "Sorting time: " << ( end - start ) << endl;

    print( arr );

    return 0;
}
```

a[i] = a[i]

j = a[i]

End