INTRODUCTION

In this week we cover the following topics

5.1 Templates
5.2 Vector Data Structure
5.3 `vector.h`

5.1 TEMPLATES

- The only data structure we have so far for storing multiple sets of data is the array and class. C++ allows us to create generic data structures using the concept of a *template class*.

- The template class has the following structure

```
template<typename dataType> class xyz {
   public:
      dataType data;

      dataType detData() {
         return data;
      }

      void setData(dataType T) {
         data = T;
      }
};
```

- This class has no specific type. It is waiting for an instance of the class to be declared and given a type. For example
  `xyz<int> myxzy;`
  will create an object of type `xyz` called `myxyz` that uses `dataType` of `int`s.

- Note that all the code for a template class must be contained in the one function. Furthermore, the code for each member function must be declared within the class.

5.2 VECTOR DATA STRUCTURE

- As a data structure, arrays are limited because they are fixed in size at compile time and because they are prone to memory errors. That is, trying to access elements outside the array boundaries.

- By using classes and templates we can create a generic data structure called a *vector* that is like an array but solves the limitations of arrays.

- It will have all the operations of an array, including using the `[]` operator, plus be dynamic in size – growing as we need it to – and have routines to catch memory errors that exit the program gracefully rather than causing strange, difficult to find errors. It will also have extra functions that will allow us to manipulate the vector in ways that arrays cannot.

5.3 `vector.h`

```
sally% cat dataobject.h

#ifndef DATAOBJECT_H_
#define DATAOBJECT_H_

#include <algorithm>

/*************************************************\
   class for holding data to be stored in
   container objects
\*************************************************/
```

```
struct dataObject
{
   int keyval;

   /*********************************************\
      place any other data declarations the
      data object may need here
   \*********************************************/

   /*********************************************\
      constructors
   \*********************************************/

   // constructor
   dataObject() : keyval(-1) {}
   dataObject(int val) : keyval(val) {}

   // copy constructor
   dataObject(const dataObject &other) :
      keyval(other.keyval) {
      // copy any data in other to this
   }

   /*********************************************\
      misc functions
   \*********************************************/

   void swapObjects(dataObject &other) {
      std::swap(keyval, other.keyval);
      // swap any other data in dataObject
   }

   bool isKeyval(int val) {
      return (val == keyval);
   }

   /*********************************************\
      overloaded operators
   \*********************************************/
```

```
    // overloaded assignment operator
    dataObject& dataObject::operator =
       (const dataObject &other) {
       dataObject temp(other);
       swapObjects(temp);
       return *this;
    }

    // overloaded relational operators
    bool dataObject::operator ==
       (const dataObject &other) const {
       return (keyval == other.keyval);
    }

    bool dataObject::operator !=
       (const dataObject &other) const {
       return (keyval != other.keyval);
    }

    bool dataObject::operator >
       (const dataObject &other) const {
       return (keyval > other.keyval);
    }

    bool dataObject::operator <
       (const dataObject &other) const {
       return (keyval < other.keyval);
    }

    bool dataObject::operator >=
       (const dataObject &other) const {
       return (keyval >= other.keyval);
    }

    bool dataObject::operator <=
       (const dataObject &other) const {
       return (keyval <= other.keyval);
    }
};

#endif

// ###############################################
```

```
sally% cat vector.h

#ifndef VECTOR_H_
#define VECTOR_H_

#include <stdexcept>

/*************************************************\
   template class for vector
\*************************************************/

template<typename dataType> class vector
{
   private:
      static const int INITIAL_CAPACITY = 100;
      int currentCapacity;
      int numItems;
      dataType* theData;

      /*****************************************\
         misc functions
      \*****************************************/

      void resize(int newCapacity) {
         if (newCapacity > currentCapacity) {
            // set aside space with
            // extra capacity
            currentCapacity = newCapacity;
            dataType *newData =
                        new dataType[newCapacity];

            // copy the items across to
            // the new space
            for (int i=0; i<numItems; i++) {
               newData[i] = theData[i];
            }

            delete[] theData;
            theData = newData;
         }
      }
```

```
    void resize() {
       resize(currentCapacity * 2);
    }

    void swap(vector<dataType> &other) {
       // swap this vector with the other
       std::swap(numItems, other.numItems);
       std::swap(currentCapacity,
                        other.currentCapacity);
       std::swap(thedata, other.thedata);
    }

public:

    /*****************************************\
       constructors and destructors
    \*****************************************/


    // default constructor
    vector() :
       currentCapacity(INITIAL_CAPACITY),
       numItems(0) {
       theData =
                new dataType[INITIAL_CAPACITY];
    }

    // copy constructor
    vector(const vector<dataType> &other) :
       currentCapacity(other.currentCapacity),
       numItems(other.numItems) {
       theData =
           new dataType[other.currentCapacity];

       for (int i=0; i<numItems; i++) {
          theData[i] = other.theData[i];
       }
    }

    // destructor
    ~vector() {
       delete[] theData;
    }
```

```
/*****************************************\
   vector size functions
\*****************************************/

bool empty() const {
   return (numItems == 0);
}

int size() const {
   return numItems;
}

/*****************************************\
   insertion and push functions
\*****************************************/

void insert(const dataType& newData,
            int index) {
   // check index range
   if (index < 0) {
      throw std::out_of_range(
              "vector<dataType>::insert ");
   }

   // ensure there is enough capacity
   // to insert newData
   while (numItems >= currentCapacity ||
          index >= currentCapacity) {
      // current capacity insufficient
      // for appending newData
      resize();
   }

   // make space for inserting newData
   for (int i=numItems; i > index; i--) {
      theData[i] = theData[i-1];
   }

   // insert the newData
   theData[index] = newData;
   numItems++;
}
```

```
void push_back(const dataType& newData) {
   // append newData to end of the vector
   insert(newData, numItems);
}

void push_front(const dataType& newData) {
   // insert newData to front of the vector
   insert(newData, 0);
}

/*****************************************\
   erase, remove and pop functions
\*****************************************/

void erase (int index) {
   // check index range
   if (index < 0 || index >= numItems) {
      throw std::out_of_range(
               "vector<dataType>::erase ");
   }

   // close up data and overwrite
   // data to be erased
   for (int i= index+1; i<numItems; i++) {
      theData[i-1] = theData[i];
   }
   numItems--;
}

void pop_back() {
   erase(numItems-1);
}

void pop_front() {
   erase(0);
}


/*****************************************\
   overloaded operators
\*****************************************/
```

```
      // overloaded [] operator
      dataType& operator [] (int index) {
         if (index < 0 || index >= numItems) {
            throw std::out_of_range("vector []");
         }
         return theData[index];
      }

      // overloaded const [] operator
      const dataType& operator []
                              (int index) const {
         return (*this)[index];
      }

      // overloaded assignment operator
      vector<dataType>& operator =
                (const vector<dataType> &other) {
         vector<dataType> theCopy(other);
         swap(theCopy);
         return *this;
      }
};

#endif

// ##############################################

sally% cat testmain.cpp

/*********************************************\
   Test program for demonstrating container types
\*********************************************/
#include <sys/time.h>
#include <time.h>

#include <iostream>

#include "dataobject.h"
#include "vector.h"

using namespace std;
```

```
double difUtime(struct timeval *first,
                struct timeval *second);
bool findData(vector<dataObject> &testVector,
              int keyval, int &location);

double difUtime(struct timeval *first,
                struct timeval *second)
{
   // return the difference in seconds,
   // including milli seconds

   double difsec =
                second->tv_sec - first->tv_sec;
   double difmil =
                second->tv_usec - first->tv_usec;

   return (difsec + difmil / 1000000.0);
}

bool findData(vector<dataObject> &testVector,
              int keyval, int &location)
{
   // find the location of data in vector with
   // keyval return true or false if location
   // found or not

   int loc;

   for (loc=0; loc<testVector.size(); loc++) {
      if (testVector[loc].keyval == keyval) {
         location = loc;
         return true;
      }
   }
   return false;
}

int main()
{
   const int MAXDATA = 1000000;
   dataObject *doPtr;
   int i, keyvals[MAXDATA];
   vector<dataObject> testVector;
```

```
// data for calculating timing
struct timeval first, second;
double usecs;

try {
   /*****************************************\
      Initialise things to demonstrate
      the container
      - fill keyvals and scramble it
   \*****************************************/

   for (i=0; i<MAXDATA; i++) keyvals[i] = i;
   srand(time(NULL));
   for (i=0; i<MAXDATA; i++)
      swap(keyvals[i],
           keyvals[rand() % MAXDATA]);

   /*****************************************\
      test inserting MAXDATA data pieces into
      the container with keyval 0 to
      MAXDATA-1 in random order
   \*****************************************/

   gettimeofday(&first, NULL);
   for (i=0; i<MAXDATA; i++)
   {
      doPtr = new dataObject(keyvals[i]);
      testVector.push_back(*doPtr);
   }
   gettimeofday(&second, NULL);
   usecs = difUtime(&first, &second);
   cout << MAXDATA <<
      " items in container in random order\n";
   cout << "time taken to push data = "
      << usecs << " seconds\n\n";

   /*****************************************\
      test finding data in the container
   \*****************************************/

   gettimeofday(&first, NULL);
   findData(testVector, keyvals[0], i);
   gettimeofday(&second, NULL);
   usecs = difUtime(&first, &second);
```

```
cout << "time taken to find 1st keyval in
    container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
findData(testVector,
        keyvals[MAXDATA/2], i);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find data with "
     << keyvals[MAXDATA/2];
cout << " keyval = " << usecs
     << " seconds\n";

gettimeofday(&first, NULL);
findData(testVector, MAXDATA, i);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find data with "
      << MAXDATA;
cout << " keyval doesn't exist = "
       << usecs << " seconds\n\n";

/*****************************************\
   test removing data from the container
\*****************************************/

gettimeofday(&first, NULL);
testVector.erase(0);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase first item in
    container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testVector.erase(testVector.size()/2);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase middle item in
    container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testVector.pop_back();
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
```

```
      cout << "time taken to erase last item in
         container = " << usecs << " seconds\n";
   } catch (out_of_range &ex) {
      cout << "\nERROR - Out of Range Exception
                thrown\n" << ex.what() << "\n";
      exit(1);
   } catch(...) {
      cout << "\nERROR - undefined Exception
                thrown\n";
      exit(1);
   }
}
```

Results from running `testmain`

```
sally% ./testmain
1000000 items in container in random order
time taken to push data = 2.97715 seconds

time taken to find 1st keyval in container = 4e-
06 seconds
time taken to find data with 487117 keyval =
0.123493 seconds
time taken to find data with 1000000 keyval
doesn't exist = 0.240686 seconds

time taken to erase first item in container =
0.412186 seconds
time taken to erase middle item in container =
0.2008 seconds
time taken to erase last item in container = 1e-
06 seconds
```