Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #03: C++ Standard Template Library & Dynamic Memory

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Understand *Class Templates*

• Understand and how to use *The Standard Library (STL)*

• Familiarize with *C++ STL Vectors*

• Understand *Pointers & Dynamic Memory*

• Familiarize with *The miniVector Class*

• Familiarize with *The matrixr Class*
Class Templates
Class Templates

• A template is a "pattern" for a class or function in which certain type names and constants are left as general template parameters.

• Class templates allow us to write general patterns for classes from which the compiler then generates multiple instances.

• Class templates are especially useful for containers, data structures that serve as collections of other, smaller data types.

• Template parameters are specified in a template header at the start of the class declaration.

• Template header consists of the keyword “template” followed by the list of replaceable names inside <...>.

• Instantiate a class template by supplying the parameter value:
  • Class TemplateName<myFavoriteType> myVariable;
Class Templates

• Format:

    template <class T>
    class ClassName{
        definition
    }

• Need not use "T", any identifier will work

• To create an object of the class, type

    ClassName< type > myObject;

    Example: Stack< double > doubleStack;
Class Templates

• Template class functions
  • Declared normally, but preceded by `template<class T>`
  • Generic data in class listed as type T
  • Binary scope resolution operator used
  • Template class function definition:
    ```cpp
    template<class T>
    MyClass< T >::MyClass(int size)
    {
        myArray = new T[size];
    }
    ```
  • Constructor definition - creates an array of type T
Example: Template Declaration/Impl.

sequence.h

template<class Data>
class Sequence{
    Data* data;
    int numStrings;
    public:
        Sequence(int maxSize);
        void add(Data newValue, int position);
};
template<class Data>
Sequence<Data>::Sequence(int maxSize) : numStrings(0) {
    data = new string[maxSize];
}
template<class Data>
void Sequence<Data>::add(Data newValue, int position) {
    for (int i = numStrings-1; i >= position; --i)
        data[i+1] = data[i];
    data[position] = newValue;
    ++numStrings;
}
Example: Template Instantiation

main.cc

```cpp
#include "sequence.h"

int main()
{
    // ...
    Sequence<std::String> peopleName;
    // ...

    return 0;
}
```
Class Templates and Non-type Parameters

• Can use non-type parameters in templates
  • Default argument
  • Treated as `const`

• Example:

  ```cpp
  template< class T, int elements >
  Stack< double, 100 > mostRecentSalesFigures;
  ```

  • Declares object of type `Stack< double, 100>`

• This may appear in the class definition:

  ```cpp
  T stackHolder[ elements ];  //array to hold stack
  ```

  • Creates array at compile time, rather than dynamic allocation at execution time
Templates and friends

- Friendships allowed between a class template and
  - Global function
  - Member function of another class
  - Entire class

- friend functions
  - Inside definition of class template X:
    - friend void f1();
      - f1() a friend of all template classes
    - friend void f2( X< T > & );
      - f2( X< int > & ) is a friend of X< int > only. The same applies for float, double, etc.
    - friend void A::f3();
      - Member function f3 of class A is a friend of all template classes
Templates and friends (II)

- friend void C< T >::f4( X< T > & );
  - C<float>::f4( X< float > & ) is a friend of class X<float> only

friend classes

- friend class Y;
  - Every member function of Y a friend with every template class made from X

- friend class Z<T>;
  - Class Z<float> a friend of class X<float>, etc.
Templates and static Members

• Non-template class
  • static data members shared between all objects

• Template classes
  • Each class (int, float, etc.) has its own copy of static data members
  • static variables initialized at file scope
  • Each template class gets its own copy of static member functions
The Standard Library (STL)
Overview

- A large project builds upon libraries of ADTs built for that specific project (a.k.a. domain-specific ADTs).

- These in turn are generally built upon the library of ADTs provided by all C++ compilers.

- This standard C++ library is called “std”.
• The key components of std are
  • I/O library: e.g., streams
  • Utility ADTs: e.g., string
  • Containers (ADTs that hold one or more values of some other ADT): vector, set, . . .
  • Iterators: "positions" within a container
  • Algorithms: common programming "patterns"

• Initial draft of std, defined by C++ standards committee, included what we now recognize as the I/O and utility sections.

• A group of researchers from HP then proposed addition of their STL library to std.
  • STL consisted of a set of containers, iterators, and function templates (algorithms).

• “STL” stands for “Standard Template Library”
std Containers

• We call an ADT a “container” if its main purpose is to provide a collection of items of some other, simpler types.

• Types of Containers

  • **Sequence Containers**: data are stored and accessed by position in linear order.

  • **Associative Containers**: elements are stored and accessed by a key such as name or number that has no relation to the location of the element (i.e., random).

  • **Adapter Containers**: contains another container as underlying structure with a restricted set of operations.

<table>
<thead>
<tr>
<th>Sequence Containers</th>
<th>Adapter Containers</th>
<th>Associative Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Stack</td>
<td>Set, Multiset</td>
</tr>
<tr>
<td>Deque</td>
<td>Queue</td>
<td>Map, Multimap</td>
</tr>
<tr>
<td>List</td>
<td>Priority Queue</td>
<td></td>
</tr>
</tbody>
</table>
List Container

Inserting into a List Container

Before

front
3 → 12 → 6 → 4

After

front
12 → # → 6 → 4

front
15

next
next
next
next

front
rear
Stack Container

• A stack allows access at only one end of the sequence, called the top.

(a) Push A
(b) Push B
(c) Push C
(d) Pop C
(e) Pop B
(f) Push A
Queue Container

- A queue is a container that allows access only at the front and rear of the sequence.
Priority Queue Container

• A priority queue is a storage structure that has restricted access operations similar to a stack or queue.

• Elements can enter the priority queue in any order. Once in the container, a delete operation removes the largest (or smallest) value.

Value = 8

18  13
  
3  15
  
27
Set Container

• A set is a collection of **unique values**, called keys or set members.

Set A

- 5
- 3
- 27

Set B

- Buick
- Ford
- Honda

- BMW
- Jaguar
- Jeep
Map Container

- A map is a storage structure that implements a **key-value** relationship.

<table>
<thead>
<tr>
<th>Index</th>
<th>Part#</th>
<th>Price</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td>A29-468</td>
<td>D7B-916</td>
<td>4.95</td>
<td>Mirage</td>
</tr>
<tr>
<td>D7B-916</td>
<td>W91-A83</td>
<td>12.50</td>
<td>Calloway</td>
</tr>
<tr>
<td>W91-A83</td>
<td>A29-468</td>
<td>8.75</td>
<td>Martin</td>
</tr>
</tbody>
</table>
Consistent Interface

• One goal of the std library is to provide consistent interfaces to many different containers.
  • Many operations are common to multiple ADTS.
  • These are always given the same name and formal parameters.

• Examples:
  • If a container allows you to add and remove elements from its end, those operations are always called push_back and pop_back.

    myContainer.pop_back(); // discard last element from container
    myContainer.push_back(x); // add x to end of container

  • Similarly, if a container allows you to add and remove elements from its front, those operations are always called.
Consistent Interface

• Examples:
  • Operations to add and remove elements at arbitrary positions are always called insert and erase. These use iterators to specify the position:

    ```
    myContainer.insert (x, position) ;  // insert x at the indicated position,
                                       // shifting other elements out of the way
    myContainer.erase (position) ;     // discard the element at that position
    myContainer.erase (start, stop) ;  // discard all elements at positions
                                       // start up to ( but not including ) stop .
    ```

  • You can test to see if a container is empty with empty(). You can ask how many items are in the container with size().

  • If a container provides iterators, these are called iterator and const_iterator. The beginning position in the container is given by begin(). The position just after the last element is given by end(). When the container is empty, begin()==end().
C++ Vectors
C++ Arrays

• An array is a **fixed-size collection** of values of the **same data type**.

• An array is a container that stores the **n (size)** elements in a **contiguous** block of memory.

```
arr[0]  arr[1]  arr[2]  ...  arr[n-1]
0       1       2       ...      n-1
```

• The size of an array is **fixed** at the time of its declaration and **cannot be changed during the runtime**.
  • An array cannot report its size. A separate integer variable is required in order to keep track of its size.

• C++ arrays **do not allow the assignment** of one array to another.
  • The copying of an array requires the generation of a loop structure with the array size as an upper bound.
Vector Container

• **Generalized array** that stores a collection of elements of the same data type

• Vector – similar to an array
  • Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector

• Vector vs. array
  • Vector has operations that **allow the collection to grow and contract dynamically at the rear** of the sequence

\[
\begin{array}{cccccc}
  0 & 1 & 2 & & n-1 \\
\end{array}
\]
The Vector API

- **vector();**
  - Create an empty vector. This is the default constructor.

- **vector(int n, const T& value = T());**
  - Create a vector with *n* elements, each having a specified value. If the value argument is omitted, the elements are filled with the default value for type *T*. Type *T* must have a default constructor, and the default value of type *T* is specified by the notation *T()*.

- **vector(T *first, T *last);**
  - Initialize the vector using the address range *[first, last)*. The notation *first and *last is an example of pointer notation.

- **T& back();**
  - Return the value of the item at the rear of the vector.
    Precondition: The vector must contain at least one element.

- **const T& back() const;**
  - Constant version of back().
The Vector API

- **bool empty() const;**
  - Return true if the vector is empty and false otherwise.

- **T& operator[](int i);**
  - Allow the vector element at index i to be retrieved or modified.
    - Precondition: The index, i, must be in the range $0 \leq i < n$, where n is the number of elements in the vector.
    - Postcondition: If the operator appears on the left side of an assignment statement, the expression on the right side modifies the element referenced by the index.

- **const T& operator[](int i) const;**
  - Constant version of the index operator.

- **void push_back(const T& value);**
  - Add a value at the rear of the vector.
    - Postcondition: The vector has a new element at the rear and its size increases by 1.
The Vector API

- void **pop_back**();
  - Remove the item at the rear of the vector.
    - Precondition: The vector is not empty.
    - Postcondition: The vector has a new element at the rear or is empty.

- void **resize**(int n, const T& fill = T());
  - Modify the size of the vector. If the size is increased, the value fill is added to the elements on the tail of the vector. If the size is decreased, the original values at the front are retained.
    - Postcondition: The vector has size n.

- int **size**() const;
  - Return the number of elements in the vector.
Vector Container

• Declaring Vector Objects:

```cpp
#include <vector>

... 
vector<int> scores(100);  // 100 integer scores
vector<string> stringList(20);  // list of 20 strings
vector<char> line(80, ' ');  // 80 characters initialized to ' '
vector<time24> openTime(7, time24(9,0));  // list of 7 elements with initial value of 9:00
int intArray[5] = {9,2,7,3,12};  // array with initial values
vector<int> intVector(intArray, intArray+5);  // five-element vector initialized from intArray
vector<double> dblVector;  // empty vector (size=0) of double
```

• Adding/Removing Vector Elements:

```cpp
char vowel[] = {'a', 'e', 'i', 'o', 'u'};
int vowelSize = sizeof(vowel)/sizeof(char);
vector<char> v(vowel, vowel+vowelSize);
cout << v.back();  // output: 'u'
v.push_back('w');  // add 'w' onto end of vector
v.back() = 'y';  // changes back from 'w' to 'y'

// v is double vector with elements {4.6, 6.8}
cout << v.back();  // output: 6.8
v.pop_back();  // remove element 6.8
v.pop_back();  // remove element 4.6
v.empty();  // returns true
```
Vector Container

- Resizing a Vector

```cpp
int arr[] = {7, 4, 9, 3, 1};
int arrSize = sizeof(arr)/sizeof(int);

vector<int> v(arr, arr+arrSize);
v.resize(2*arrSize);  // vector size is doubled; fill value is 0
v.resize(4);         // vector is truncated to 4 elements. Data is lost
v.resize(10, 1);     // vector grows to size 10 with fill 1
```

```cpp
vector<int> v(arr, 5)

| 7 | 4 | 9 | 3 | 1 |

v.resize(10);

| 7 | 4 | 9 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |

v.resize(4);

| 7 | 4 | 9 | 3 |
```
Pointers & Dynamic Memory
C++ Pointers

• Memory consists of a sequence of bytes. Each location in the sequence has an associated address, and the contents at that address define a byte.

• As a vertical display, the memory locations resemble numbered lines on a sheet of paper. The horizontal view displays memory as contagious series of bytes along a row.
C++ Pointers

- The number of bytes to store a data item depends on its type
  - Example: an integer requires 4 bytes while a real number requires 8 bytes
  - Associate with the item the address of its first byte.

- To make addresses more meaningful, C++ provides pointer variables.
- A pointer holds the address of an object of a specified type.
- We can declare integer pointers, character pointers, time24 pointers, etc.
- Pointer variables have value that are memory addresses. The address provides access to the data at the address.
Declaring Pointer Variables

• Declare a pointer by stating the type followed by the variable name, but with a "*" added immediately before the name.

```c
    type *ptr;
```

• The pointer `ptr` is a variable whose value is the address of a data item of the designated type.

```c
    int *intPtr;
    char *charPtr;
```
Assigning Values to Pointers

\[
\text{int } m = 50, *\text{intPtr};
\]

- \&m is the address of the integer in memory. The assignment statement

\[
\text{intPtr = \&m;}
\]

- sets intPtr to point at an actual data item.

- The Figure illustrates the status of the two variables from the declaration statement and the resulting status after the assignment of a value to intPtr.
int x = 50, y = 100,
*px = &x, *py = &y;

*px = *py + 2;  // assign to x the value y + 2 = 102
*py *= 2;       // double the value of y to 200
(*px)++;        // increment x from 102 to 103
Arrays and Pointers

- Array name is a constant pointer pointing to the starting address of the list of elements.
- The compiler associates with the pointer the type, and thus the size, of the data item it references.

```c
int arr[7];
// arr is array of 7 integer elements
```

```c
*arr = arr[0];
*(arr+1) = arr[1];
...
```

![Diagram showing array and pointer operations]
Pointers and Class Types

- Pointers are used to reference objects.
- Operator ‘*’ allows a pointer to access the object it references.
- Referencing data & method members of an object requires the use of combination of ‘*’ operator and ‘.’ operator.
- Parentheses are needed since ‘*’ operator has a lower precedence than the ‘.’ operator.
- To simplify combination referencing, operator ‘->’ is used as a shortcut.

```cpp
rectangle box(2,5), *rectPtr;
//2x5 rectangle and pointer
rectPtr = &box;

cout << (*rectPtr).area();
(*rectPtr).setSides(7,10);

cout << rectPtr->area();
rectPtr->setSides(7,10);
```
Operator ‘new’

- A program allocates (borrows) memory from the heap when additional storage space is required, then it deallocate (return) when it is no longer needed.

- Heap memory is referred to by dynamic memory.

- Use operator ‘new’ to allocate memory. If memory is not available, the system return 0 value.

```
p = new time24;
// *p is 00:00 (midnight)

q = new time24(8, 15);
// *q is 8:15 AM
```
Operator ‘delete’

• To deallocate memory, use operator ‘delete’ with the object pointer.

• deallocating a dynamic array, use a slightly different form of delete. Place square brackets [] between delete and the pointer variable name. The system deallocates all of the memory originally assigned to the dynamic array.

```cpp
arr = new int[arrSize];  // allocated integer array of arrSize
//note the difference of ‘new int (arrSize)’
delete [] arr;
// deallocate dynamic array storage
```
Dynamic Class

• The process of allocating/deallocating memory during program runtime is refereed to as “Memory Management”.

• Static data items, the compiler provides instructions for runtime system for their memory management.

• Dynamic data items, the programmer is responsible to provide the instructions for memory management (i.e., new & delete).

• Dynamic Class is a class type with dynamic data members.

• Memory leak is refereed to the situation of allocating memory but not deallocating it once it is not needed anymore.

• If your class is Dynamic Class, you should provide copy constructor, assignment operator, and destructor.
template <class T>
class dynamicClass
{
private:
    // variable of type T and a pointer to data of type T
    T member1;
    T *member2
public:
    // constructor with parameters to initialize member data
    dynamicClass(const T &m1, const T &m2)
    // copy constructor: create a copy of the input object
    dynamicClass(const dynamicClass<T> &obj)
    // some methods…
    ....
    // assignment operator
    dynamicClass<T> *operator=(const
dynamicClass<T> *rhs)
    // destructor
    dynamicClass(void)
};
// constructor with parameters to initialize member data
DynamicClass<T>::DynamicClass(const T &m1, const T &m2)
{
    // parameter m1 initializes static member
    member1 = m1;
    // allocate dynamic memory and initialize it with value m2
    member2 = new T(m2);
    cout << "Constructor:" << member1 << '/' << *member2 << endl;
}

// pointer variable
dynamicClass<int> *dynamicObj;
// allocate an object
dynamicObj = new dynamicClass<int>(2, 200);

// Dynamic Class object
dynamicClass<int> objA(1, 2);

objA

member1 = 1    member2

Heap memory

*(objA.member2) = 2
//copy constructor: initialize new object to have the same data as obj.
template <class T>
dynamicClass<T>::dynamicClass(const dynamicClass<T> &obj)
{
    // copy static data member from obj to current object
    member1 = obj.member1;
    //allocate dynamic memory and initialize it with value *obj.member2
    member2 = new T(*(obj.member2));
    cout<<"Copy Constructor:"<<member1<<'/'<<member2<<endl;
}
dynamicClass<int> objA(3,5), objB=objA;
// destructor: deallocates memory allocated by the constructor
template <class T>
dynamicClass<T>::~dynamicClass(void)
{
    cout<<"Destructor:"<<member1<<'/'<<member2<<endl;
    delete member2;
}

//If no destructor is defined
void main()
{
    dynamicClass<int> objA(1, 2);
}

Example

void DestroyDemo(int m1, int m2)
{
dynamicClass<int> Obj(m1,m2)
}

void main(void)
{
dynamicClass<int> Obj1(1,100), *Obj2;
    Obj2=new DynamicClass<int>(2,200);
    DestroyDemo(3,300);
    delete Obj2;
}

//Constructor for Obj(3,300)
//Destructor for Obj

//Constructor for Obj1(1,100)
//Constructor for *Obj2(2,200)
//Destructor for Obj2
//Destructor for Obj1
Overloaded Assignment Operator

// initialization
dynamicClass objectA(2,3), objectB=objectA;
// data in objectB is initialized by data in objectA
using copy constructor

// assignment
objectB = objectA; //default assignment operator
//data in Y is overwritten by data in X

// Overloaded assignment operator = returns a reference to the current object
template <class T>
dynamicClass<T>& operator= (const dynamicClass <T>& rhs)
{
    //copy static data member from rhs to the current object
    member1=rhs.member1
    // content of the dynamic memory must be same as that rhs
    *member2 =* (rhs.member2);
    cout <<"Assignment Operator: """<<member1<<'/'<<*member2<<endl
    return *this;
    //reserved word this is used to return a reference to the current object
}
The Pointer ‘this’

\texttt{*this}

\texttt{\textbf{IS the object}}

\texttt{\texttt{objA;}\vspace{10mm}}

\texttt{this->}

\texttt{\textbf{Points to the Object}}

\texttt{member}
The miniVector Class
miniVector Class

- miniVector class is a **simplified** version of STL Vector class.
- miniVector is a **dynamic class**
- The heart of miniVector object is a **dynamically allocated array** that reserves space to store the elements.
- The amount of available space is called the **capacity** of the object.
- The actual number of elements in the object is called **size**.

```cpp
template <class T> class miniVector {
    ...
    private :
        int vCapacity; // amount of available space
        int vSize; // number of elements in the list
        T * vArr; // the dynamic array
    }

T & miniVector<T> :: operator [ ] (unsigned int i)
{
    assert (( i >= 0) && (index < vSize ));
    return vArr [ i ];
}
```
Adding to miniVector Class

// insure that list has sufficient capacity,
// add the new item to the list, and increment vSize
template <typename T>
void miniVector<T>::push_back ( const T& item) {

  // if space is full, allocate more capacity
  if (vSize == vCapacity)
    {
      if (vCapacity == 0)
        // if capacity is 0, set capacity to 1.
        // set copy to false because there are
        // no existing elements
        reserve(1,false);
      else
        // double the capacity
        reserve(2*vCapacity, true);
    }

  // add item to the list, update vSize
  vArr[vSize] = item;
  vSize ++;
}

Array has room
(vSize < vCapacity)

v.push_back(t5);
Adding to miniVector Class

```cpp
// insure that list has sufficient capacity,
// add the new item to the list, and increment vSize

template <typename T>
void miniVector<T>::push_back(const T& item)
{
    // if space is full, allocate more capacity
    if (vSize == vCapacity)
    {
        if (vCapacity == 0)
        {
            // if capacity is 0, set capacity to 1.
            // set copy to false because there are
            // no existing elements
            reserve(1, false);
        }
        else
        {
            // double the capacity
            reserve(2 * vCapacity, true);
        }
    }
    // add item to the list, update vSize
    vArr[vSize] = item;
    vSize++;
}
```

Array is full
(vSize == vCapacity)
Adding to miniVector Class

template <typename T>
void miniVector<T>::reserve(int n, bool copy)
{
T * newArr;
int i;
// allocate a new dynamic array with n elements
newArr = new T[n];
if (newArr == NULL)
throw memoryAllocationError("miniVector reserve(): memory allocation failure");
// if copy is true, copy elements from the old list to the new list
if (copy)
for (i = 0; i < vSize; i++)
newArr[i] = vArr[i];
// delete original dynamic array. if vArr is NULL, the vector was
// originally empty and there is no memory to delete
if (vArr != NULL)
delete [] vArr;
// set vArr to the value newArr. update vCapacity
vArr = newArr;
vCapacity = n;
}

Array is full
(vSize == vCapacity)
Adding to miniVector Class

// insure that list has sufficient capacity,
// add the new item to the list , and increment vSize
template <typename T>
void miniVector<T>::push_back ( const T& item )
{
    // if space is full, allocate more capacity
    if ( vSize == vCapacity )
    {
        if ( vCapacity == 0 )
            // if capacity is 0, set capacity to 1.
            // set copy to false because there are
            // no existing elements
            reserve(1,false);
        else
            // double the capacity
            reserve(2 * vCapacity, true);
    }
    // add item to the list , update vSize
    vArr [ vSize ] = item;
    vSize ++;
}
push_back() Performance

- When we add to the end of a vector that is already filled to capacity(), we
  - allocate a new array with twice the capacity() ~ O(1)
  - copy the old elements into the new array ~ O(size())
  - discard the old array ~ O(1)
  - add the new element to the end ~ O(1)

- Total is O(size())

- How long does it take to do a total of \( n \) push_back() operations?
- Let \( m \) be the current size() of the vector.
  - If \( m \) is a power of 2, say, \( 2^i \) then the array is full and we do \( O(m) \) work on the next call to push_back().
  - If \( m \neq 2^i \), then we do \( O(1) \) work on next call.
push_back() Performance

- Let k be the smallest integer such that \( n \leq 2^k \). For the sake of simplicity, we’ll do the analysis as if we do \( 2^k \) pushes.

- Let m be the current size() of the vector.
  - If m is a power of 2, say, \( 2^i \) then the array is full and we do \( O(m) \) work on the next call to push_back().
  - If \( m \neq 2^i \), then we do \( O(1) \) work on next call.

- We can then add up the total effort as:
  \[
  \sum_{i=1}^{k} \left( O(2^i) + \sum_{j=1}^{2^i-1} O(1) \right)
  = O \left( \sum_{i=1}^{k} 2^i \right)
  = O \left( 1 + 2 + 4 + \ldots + 2^k \right)
  = O \left( 2^{k+1} - 1 \right)
  = O \left( 2^{k+1} \right)
  \]

- The total effort is \( O(2^{k+1}) \)

- Since \( n \leq 2^k \) \( \rightarrow \) total effort is \( O(2n) = O(n) \).
push_back() Performance

• So:
  • an individual call to push_back() may be \( O(n) \),
  • the \textbf{total} effort for all \( n \) push_back 's used to build a vector "from scratch" is also \( O(n) \)

• We say that the \textcolor{red}{amortized} worst-case time of push_back() is therefore \( O(1) \).

\textit{amortize: to decrease (on average) over an extended period of time.}
## miniVector Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector()</td>
<td>O(1)</td>
</tr>
<tr>
<td>vector(n, x)</td>
<td>O(n)</td>
</tr>
<tr>
<td>size()</td>
<td>O(1)</td>
</tr>
<tr>
<td>v[i]</td>
<td>O(1)</td>
</tr>
<tr>
<td>push_back(x)</td>
<td>O(1)</td>
</tr>
<tr>
<td>pop_back</td>
<td>O(1)</td>
</tr>
<tr>
<td>insert</td>
<td>O(size())</td>
</tr>
<tr>
<td>erase</td>
<td>O(size())</td>
</tr>
<tr>
<td>front, back</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
#ifndef MINI_VECTOR
#define MINI_VECTOR

#include "d_except.h" // include exception classes

using namespace std;

template<typename T>
class miniVector
{
    public:
        miniVector(int size = 0);
            // constructor.
            // Postconditions: allocates array with size number of elements
            // and capacity. elements are initialized to T(), the default
            // value for type T
        miniVector(const miniVector<T>& obj);
            // copy constructor
            // Postcondition: creates current vector as a copy of obj
        ~miniVector();
            // destructor
            // Postcondition: the dynamic array is destroyed

    private:
        T* data;
        int capacity;
        int size;

    friend class MiniVectorException;

};
miniVector Class Implementation

miniVector& operator= (const miniVector<T>& rhs);
   // assignment operator.
   // Postcondition: current vector holds the same data
   // as rhs

T& back();
   // return the element at the rear of the vector.
   // Precondition: the vector is not empty. if vector
   // is empty, throws the underflowError exception

const T& back() const;
   // const version used when miniVector object is a constant

T& operator[](int i);
   // provides general access to elements using an index.
   // Precondition: vector is not empty. if vector is empty
   // throws the indexRangeError exception

const T& operator[](int i) const;
   // const version used when miniVector object is a constant

void push_back(const T& item);
   // insert item at the rear of the vector.
   // Postcondition: the vector size is increased by 1
miniVector Class Implementation

void pop_back();
    // remove element at the rear of the vector.
    // Precondition: vector is not empty. if the vector is
    // empty, throws the underflowError exception

void insert(int i, const T& item);
    // insert item at index i in the vector.
    // Precondition: vector is not empty and 0 <= i <= vSize.
    // Postcondition: the vector size increases by 1

void erase(int i);
    // erase the item at index i in the vector.
    // Preconditions: vector is not empty and 0 <= i < vSize.
    // Postcondition: the vector size decreases by 1

int size() const;
    // return current list size

bool empty() const;
    // return true if vector is empty and false otherwise

int capacity() const;
    // return the current capacity of the vector
miniVector Class Implementation

private:

int vCapacity; // amount of available space
int vSize; // number of elements in the list
T *vArr; // the dynamic array

void reserve(int n, bool copy);
    // called by public functions only if n > vCapacity. expands
    // the vector capacity to n elements, copies the existing
    // elements to the new space if copy == true, and deletes
    // the old dynamic array. throws the memoryAllocationError
    // exception if memory allocation fails

};
miniVector Class Implementation

// set the capacity to n elements
template <typename T>
void miniVector<T>::reserve(int n, bool copy)
{
    T *newArr;
    int i;

    // allocate a new dynamic array with n elements
    newArr = new T[n];
    if (newArr == NULL)
        throw memoryAllocationError("miniVector reserve(): memory allocation failure");

    // if copy is true, copy elements from the old list to the new list
    if (copy)
        for(i = 0; i < vSize; i++)
            newArr[i] = vArr[i];

    // delete original list, set arr to point to newVarr.
    // if vArr is NULL, the vector was originally empty
    // and there is no memory to delete
    if (vArr != NULL)
        delete [] vArr;

    // update vCapacity and vArr
    vCapacity = n;
    vArr = newArr;
}
miniVector Class Implementation

// constructor. initialize vSize and vCapacity.
// allocate a dynamic array of vSize integers
// and initialize the array with T()
template <typename T>
miniVector<T>::miniVector(int size):
    vSize(0), vCapacity(0), vArr(NULL)
{
    int i;

    // if size is 0, vSize/vCapacity are 0 and vArr is NULL.
    // just return
    if (size == 0)
    {
        return;
    }

    // set capacity to size. since we are building the vector,
    // copy is false
    reserve(size, false);
    // assign size to vSize
    vSize = size;

    // copy T() into each vector element
    for (i=0; i < vSize; i++)
    {
        vArr[i] = T;
    }
}
// copy constructor. make the current object a copy of obj.
// for starters, use initialization list to create an empty
// vector

template <typename T>
miniVector<T>::miniVector (const miniVector<T>& obj):
    vSize(0), vCapacity(0), vArr(NULL)
{
    int i;

    // if size is 0, vSize/vCapacity are 0 and vArr is NULL.
    // just return
    if (obj.vSize == 0)
        return;

    // set capacity to obj.vSize. since we are building the vector,
    // copy is false
    reserve(obj.vSize, false);
    // assign size to obj.vSize
    vSize = obj.vSize;

    // copy items from the obj.arr to the newly allocated array
    for (i = 0; i < vSize; i++)
        vArr[i] = obj.vArr[i];
}
miniVector Class Implementation

// destructor. deallocate the dynamic array
template <typename T>
miniVector<T>::~miniVector()
{
    if (vArr != NULL)
        // de-allocate memory for the array
        delete[] vArr;
}

// replace existing object (left-hand operand) by rhs (right-hand operand)
template <typename T>
miniVector<T>& miniVector<T>::operator= (const miniVector<T>& rhs)
{
    int i;

    // check vCapacity to see if a new array must be allocated
    if (vCapacity < rhs.vSize)
        // make capacity of current object the size of rhs. don't
        // do a copy, since we will replace the old values
        reserve(rhs.vSize, false);

        // assign current object to have same size as rhs
        vSize = rhs.vSize;

    // copy items from the rhs.arr to current array
    for (i = 0; i < vSize; i++)
        vArr[i] = rhs.vArr[i];

    return *this;
}
miniVector Class Implementation

template <typename T>
T& miniVector<T>::back()
{
    if (vSize == 0)
        throw underflowError("miniVector back(): vector empty");
    return vArr[vSize-1];
}

template <typename T>
const T& miniVector<T>::back() const
{
    if (vSize == 0)
        throw underflowError("miniVector back(): vector empty");
    return vArr[vSize-1];
}

// provides general access to array elements
template <typename T>
T& miniVector<T>::operator[] (int i)
{
    if (i < 0 || i >= vSize)
        throw indexRangeError("miniVector: index range error", i, vSize);
    return vArr[i];
}
// provides general access to array elements. constant version
template<typename T>
const T& miniVector<T>::operator[](int i) const
{
    if (i < 0 || i >= vSize)
        throw indexRangeError("miniVector: index range error", i, vSize);
    return vArr[i];
}

// insure that list has sufficient capacity, add the new item to the list, and increment vSize
template<typename T>
void miniVector<T>::push_back(const T& item)
{
    // if space is full, allocate more capacity
    if (vSize == vCapacity)
    {
        if (vCapacity == 0)
            // if capacity is 0, set capacity to 1.
            // set copy to false because there are no existing elements
            reserve(1, false);
        else
            // double the capacity
            reserve(2 * vCapacity, true);
    }
    // add item to the list, update vSize
    vArr[vSize] = item;
    vSize++;
}
miniVector Class Implementation

// if not empty, just decrement the size
template <typename T>
void miniVector<T>::pop_back()
{
    if (vSize == 0)
        throw underflowError("miniVector pop_back(): vector is empty");

    vSize--;
}

template <typename T>
void miniVector<T>::insert(int i, const T& item)
{
    int k;

    // make sure the vector is not empty
    if (vSize == 0)
        throw underflowError("miniVector insert(): vector empty");

    // verify that i is in the range [0, vSize]
    if (i < 0 || i > vSize)
        throw indexRangeError("miniVector insert(): index range error", i, vSize);
miniVector Class Implementation

// if space is full, allocate more capacity
if (vSize == vCapacity)
    // double the capacity and copy original vector values to the new memory
    reserve(2 * vCapacity, true);

// slide the elements at indices i+1, i+2, ..., vecSize-1
// to the right one position
// start at the last vector element and copy up to the next
// element until copying vArr[i]. note that if i = vSize, the
// loop does no copying. this corresponds to inserting at
// the back
k = vSize-1;
while (k >= i)
{
    vArr[k+1] = vArr[k];
    k--;
}

// copy item into location i and increment the vector size
vArr[i] = item;
    vSize++;
```cpp
template <typename T>
void miniVector<T>::erase(int i)
{
    int k;

    // make sure the vector is not empty
    if (vSize == 0)
        throw underflowError("miniVector erase(): vector empty");

    // verify that i is in the range [0, vSize)
    if (i < 0 || i >= vSize)
        throw indexRangeError("miniVector erase(): index range error", i, vSize);

    // slide the elements at indices i+1, i+2, ..., vSize-1
    // left one position
    for (k=i+1;k < vSize;k++)
        vArr[k-1] = vArr[k];

    // decrement the vector size
    vSize--;}
```
template <typename T>
int miniVector<T>::size() const
{
    return vSize;
}

template <typename T>
bool miniVector<T>::empty() const
{
    return vSize == 0;
}

template <typename T>
int miniVector<T>::capacity() const
{
    return vCapacity;
}

#endif  // MINI_VECTOR
The matrix Class
matrix Class

- A Matrix is a **two-dimensional array** that corresponds to a row-column table of entries of a specified data type.
- Matrices are referenced using a **pair of indices** that specify the row and column location in the table.

Example:
The element mat[0][3] is 2
The element mat[1][2] is 4.
#ifndef EXTENDED_MATRIX_CLASS
#define EXTENDED_MATRIX_CLASS

#include <iostream>
#include <vector>
#include "d_except.h"

using namespace std;

template <typename T>
class matrix
{
    public:
        matrix(int numRows = 1, int numCols = 1, const T& initVal = T());
        // constructor.
        // Postcondition: create array having numRows x numCols elements
        // all of whose elements have value initVal
        vector<T>& operator[](int i);
        // index operator.
        // Precondition: 0 <= i < nRows. a violation of this
        // precondition throws the indexRangeError exception.
        // Postcondition: if the operator is used on the left-hand
        // side of an assignment statement, an element of row i is changed
matrix Class Implementation

const vector<T>& operator[](int i) const;
    // version for constant objects

int rows() const;
    // return number of rows

int cols() const;
    // return number of columns

void resize(int numRows, int numCols);
    // modify the matrix size.
    // Postcondition: the matrix has size numRows x numCols.
    // any new elements are filled with the default value of type T;

friend matrix<T> operator+ (const matrix<T>& left, const matrix<T>& right);

friend matrix<T> operator- (const matrix<T>& left, const matrix<T>& right);

private:
    int nRows, nCols;
        // number of rows and columns

    vector<vector<T> > mat;
        // matrix is implemented as nRows vectors (rows),
        // each having nCols elements (columns)
matrix Class Implementation

template <typename T>
matrix<T>::matrix(int numRows, int numCols, const T& initVal):
    nRows(numRows), nCols(numCols),
    mat(numRows, vector<T>(numCols, initVal))
{}

// non-constant version. provides general access to matrix elements
template <typename T>
vector<T>& matrix<T>::operator[](int i)
{
    if (i < 0 || i >= nRows)
        throw IndexRangeError("matrix: invalid row index", i, nRows);
    return mat[i];
}

// constant version. can be used with a constant object.
// does not allow modification of a matrix element
template <typename T>
const vector<T>& matrix<T>::operator[](int i) const
{
    if (i < 0 || i >= nRows)
        throw IndexRangeError("matrix: invalid row index", i, nRows);
    return mat[i];
}
template <typename T>
int matrix<T>::rows() const
{
    return nRows;
}

template <typename T>
int matrix<T>::cols() const
{
    return nCols;
}
matrix Class Implementation

template <typename T>
void matrix<T>::resize(int numRows, int numCols)
{
    int i;

    // handle case of no size change with a return
    if (numRows == nRows && numCols == nCols)
        return;

    // assign the new matrix size
    nRows = numRows;
    nCols = numCols;

    // resize to nRows rows
    mat.resize(nRows);

    // resize each row to have nCols columns
    for (i=0; i < nRows; i++)
        mat[i].resize(nCols);
}
template <typename T>
matrix<T> operator+ (const matrix<T>& left,
const matrix<T>& right)
{
    // throw a rangeError exception if the matrices do not have the
    // same dimension
    if (left.nRows != right.nRows || left.nCols != right.nCols)
        throw rangeError(
            "matrix addition: matrices do not have the same dimension");
    // sum is the return matrix
    matrix<T> sum(left.nRows, left.nCols);
    int i,j;

    // form the sum row by row
    for (i=0;i < left.nRows;i++)
        for (j=0;j < left.nCols;j++)
            sum[i][j] = left.mat[i][j] + right.mat[i][j];

    return sum;
}
matrix Class Implementation

template <typename T>
matrix<T> operator- (const matrix<T>& left,

        const matrix<T>& right)
{
    // throw a rangeError exception if the matrices do not have the
    // same dimension
    if (left.nRows != right.nRows || left.nCols != right.nCols)
        throw rangeError(  
            "matrix subtraction: matrices do not have the same dimension");
    // diff is the return matrix
    matrix<T> diff(left.nRows, left.nCols);
    int i,j;

    // form the difference row by row
    for (i=0;i < left.nRows;i++)
        for (j=0;j < left.nCols;j++)
            diff[i][j] = left.mat[i][j] - right.mat[i][j];

    return diff;
}
#endif  // EXTENDED_MATRIX_CLASS
#include <iostream>
#include "mat.h"
using namespace std;

// input mat by rows from the keyboard
template <typename T>
void inputMatrix(matrix<T>& mat);

// output mat by rows
template <typename T>
void outputMatrix(const matrix<T>& mat);

int main()
{
    // declare the matrices
    matrix<int> matrixA(3,3), matrixB(3,3), sum(3,3), diff(3,3);
    cout << "Enter the " << matrixA.rows() << " x " 
    << matrixA.cols() << " matrixA by rows:" << endl;
    inputMatrix(matrixA);
    cout << endl;
    cout << "Enter the " << matrixB.rows() << " x " 
    << matrixB.cols() << " matrixB by rows:" << endl;
    inputMatrix(matrixB);
    cout << endl;
}
matrix Class Example

```cpp
// compute the sum and difference of matrixA and matrixB
sum = matrixA + matrixB;
diff = matrixA - matrixB;

// display the results
cout << "matrixA + matrixB = " << endl;
outputMatrix(sum);
cout << endl;

cout << "matrixA - matrixB = " << endl;
outputMatrix(diff);
return 0;
}

template <typename T>
void inputMatrix(matrix<T>& mat)
{
    int i,j;

    // input row 0, row 1, ..., row mat.rows()-1
    for (i=0;i < mat.rows();i++)
        // input an element from each column of row i
        for (j=0;j < mat.cols();j++)
            cin >> mat[i][j];
}
```
template <typename T>
void outputMatrix(const matrix<T>& mat)
{
    int i,j;

    // output row 0, row 1, ..., row mat.rows()-1
    for (i=0;i < mat.rows();i++)
    {
        // output an element from each column of row i
        for (j=0;j < mat.cols();j++)
        {
            // output newline between rows
            cout << endl;
        }
        // output an element from each column of row i
    }
}
Matrix Class Example

Output Run:

Enter the 3 x 3 matrixA by rows:
1 2 3
4 5 6
7 8 9

Enter the 3 x 3 matrixB by rows:
0 0 0
1 1 1
2 2 2

matrixA + matrixB =
1 2 3
5 6 7
9 10 11

matrixA - matrixB =
1 2 3
3 4 5
5 6 7
Questions?
Assignment #2

• Due Sat Sep 21st, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #4 & #5:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.20</td>
<td>214</td>
</tr>
<tr>
<td>4.21</td>
<td>215</td>
</tr>
<tr>
<td>5.22</td>
<td>273</td>
</tr>
<tr>
<td>5.29</td>
<td>274</td>
</tr>
<tr>
<td>5.31</td>
<td>275</td>
</tr>
</tbody>
</table>

• Programming Assignment
  • Ford & Topp, Chapter #4 & #5:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.31</td>
<td>217</td>
</tr>
<tr>
<td>5.42</td>
<td>279</td>
</tr>
</tbody>
</table>
Assignment #2

• Due Sat Sep 21st, 11:59pm

• Submission Format:
  • **Written Assignment**
    • Create single PDF file with name: `cs361_assignment_2_<firstName>_<lastName>`
    • Have a cover page with your name and your email
    • Submit through Blackboard.
  • **Programming Assignment (for each programming question)**
    • Make sure your program compiles and executes using `g++` on Dept’s Linux machines.
    • Create a “Readme.txt” file that list how to compile and execute your program. Include your name and your email.
    • Your main file should be named “main.cpp”.
    • You should have a Makefile.
    • Zip all files (.h, .c, Makefile, etc.) and name it: `program1`, and `program2` (for the two submitted programs)
    • Submit through Blackboard.
  • **Final Submission Materials (3 Files):**
    • One PDF for written assignment.
    • Two zip files (one for each program)