CLASS DESTRUCTORS:

- The name of a destructor is a ~Name_of_the_Class
- A destructor is a member function of a class that is called automatically when an object of the class goes out of scope
- This means that if an object of the class type is a local variable for a function, then the destructor is automatically called as the last action before the function call ends.
- Destructors are used to eliminate any dynamic variables that have been created by the object, so that the memory occupied by these dynamic variables is returned to the freestore.
- Destructors may perform other cleanup tasks as well.

===

LINKED LISTS
more on .... STRUCTS, POINTERS, CLASSES, DESTRUCTORS, ...

===

```
/*
POINTERS, STRUCTS & DINAMIC MEMORY ALLOCATION
IMPLEMENTATION OF A LINKED LIST CLASS FOR STORING INT VALUES

An example where a DESTRUCTOR is REQUIRED

see, for example:
http://www.codeproject.com/KB/cpp/linked_list.aspx
uses malloc() and free()
*/

// #define NDEBUG  // see comment on assert() in LinkedList::clear()
#include <iostream>
#include <cstddef>
#include <cassert>

using namespace std;

class LinkedList{
public:
	LinkedList();
	size_t size() const;
	void insertEnd(int value);
	void insertBegin(int value);
	bool insertAfter(size_t index, int value);
	bool remove(int value);
	void clear();
	void display() const;
	~LinkedList(); // NOTE: NEEDED IN THIS CASE
private:
	struct node{      // node is a TYPE !!! could also have defined a Node class
		int data;     // the elements of the list are integers (only) ...
		node *next;
	} *p;
	size_t listSize;
};


//-----------------------------------------------------------------------
// constructor

LinkedList::LinkedList()
{
	p = NULL;
	listSize = 0;
}


//-----------------------------------------------------------------------
// return the list size

size_t LinkedList::size() const
{
	return listSize;
}
```

```
//-------------------------------------------------------------------
//insert a new node at the beginning of the linked list

void LinkedList::insertBegin(int value)
{
      node *q;
      q = new node;
      q->data = value; //note: access to a struct field through a struct pointer
      q->next = p;
      p = q;
      listSize++;
}

//-------------------------------------------------------------------
// insert a new node at the end of the linked list

void LinkedList::insertEnd(int value)
{
      node *q,*t;
      //if the list is empty
      if (p == NULL) //alternative: if (listSize==0)
      {
            p = new node;
            p->data = value;
            p->next = NULL;
            // listSize++;
      }
      else
      {
            q = p;
            while(q->next != NULL)
                  q = q->next;
            t = new node;
            t->data = value;
            t->next = NULL;
            q->next = t;
            // listSize++;
      }
      listSize++;
}

//-------------------------------------------------------------------
// insert a node at a specified location
// 'index' - location
// 'value' - contents of the node
// return value - indicates if insertion was successful

bool LinkedList::insertAfter(size_t index, int value)
{
      node *q, *t;
      size_t i;

      if (index > listSize-1) //if (index > size()-1)
            return false;
      else
      {
            q = p;
            for (i = 0; i < index; i++)
                  q = q->next;
            t = new node;
            t->data = value;
            t->next = q->next;
            q->next = t;
            listSize++;
            return true;
      }
}
```

```
//----------------------------------------------------------------------
// deletes the specified value from the linked list
// 'value' - contents of the node to be deleted
// return value - indicates if removal was successful

bool LinkedList::remove(int value)
{
      node *q,*r;
      q = p;
      //if node to be deleted is the first node
      if (q->data == value)
      {
            p = p->next;
            delete q;
            listSize--;
            return true;
      }
      r = q;
      while(q != NULL)
      {
            if(q->data == value)
            {
                  r->next = q->next;
                  delete q;
                  listSize--;
                  return true;
            }
            r = q;
            q = q->next;
      }
      return false;
}

//----------------------------------------------------------------------
// deletes all the list elements

void LinkedList::clear()
{
      node *q;
      if( p == NULL )
            return;

      while( p != NULL )
      {
            q = p->next;
            delete p;
            listSize--;
            p = q;
      }

      //assert(listSize==0); //#define NDEBUG before #include <cassert>
                             //is equivalent to commenting assert's
}

//----------------------------------------------------------------------
// shows all the list elements

void LinkedList::display() const
{
      node *q;

      cout << "(" << listSize << "): ";
      for(q = p; q != NULL; q = q->next)
            cout <<  " " << q->data;
      cout << endl << endl;
}
```

```
//--------------------------------------------------------------------------
// Destructor
// MUST BE IMPLEMENTED WHEN DYNAMIC MEMORY WAS ALLOCATED
// to free all the memory allocated for the list nodes

LinkedList::~LinkedList()
{
	clear(); //see LinkedList::clear()
}

//--------------------------------------------------------------------------
void main()
{

	LinkedList list;
	int index;
	int value;


	cout << "insertBegin() 1, 2, 3, 4, 5\n";
	for (value=1; value<=5; value++)
	{
		list.insertBegin(value);
		list.display();
	}

	//list.clear();
	//list.display();

	value = 6;
	cout << "insertEnd() " << value << "\n";
	list.insertEnd(value);
	list.display();

	index = 0; value = 7;
	cout << "insertAfter(" << index << "," << value << ")\n";
	if (!list.insertAfter(index,7))
		cout << "there is no such node index: " << index << endl;
	list.display();

	index = 10; value = 8;
	cout << "insertAfter(" << index << "," << value << ")\n";
	if (!list.insertAfter(index,value))
		cout << "there is no such node index: " << index << endl;
	list.display();

	value = 2;
	cout << "remove(" << value << ")\n";
	if (!list.remove(value))
		cout << "there is no such node value: " << value << endl;
	list.display();

	value = 9;
	cout << "remove(" << value << ")\n";
	if (!list.remove(value))
		cout << "there is no such node value: " << value << endl;
	list.display();

	cout << endl;
}
```

TO DO BY STUDENTS:
implement method   bool LinkedList::removeNode(size_t index)
to remove the node at a specified location, 'index', if it exists

==

TEMPLATES – GENERIC PROGRAMMING

==

FUNCTION TEMPLATES / GENERIC FUNCTIONS

```
/*
FUNCTION OVERLOADING (remembering ...)
*/

#include <iostream>

using namespace std;

//------------------------------------------------------------------------
void swapValues(int &x, int &y)
{
     int temp = x;
     x = y;
     y = temp;
}

//------------------------------------------------------------------------
void swapValues(double &x, double &y)
{
     double temp = x;
     x = y;
     y = temp;
}
//------------------------------------------------------------------------
void swapValues(char &x, char &y)
{
     char temp = x;
     x = y;
     y = temp;
}

//------------------------------------------------------------------------
void main()
{
     int    i1 = 1,    i2 = 2;
     double d1 = 1.5,  d2 = 2.5;
     char   c1 = 'A',  c2 = 'B';

     swapValues(i1,i2);
     swapValues(d1,d2);
     swapValues(c1,c2);

     cout << "i1 = " << i1 << ", i2 = " << i2 << endl;
     cout << "d1 = " << d1 << ", d2 = " << d2 << endl;
     cout << "c1 = " << c1 << ", c2 = " << c2 << endl;

}
```

```cpp
/*
FUNCTION TEMPLATES- example 1 (swapping values)

Compare with previous example: function overloading

When the operations are the same for each overloaded function,
they can be expressed more compactly and conveniently
using function templates

Generic programming
involves writing code in a way that is independent of any particular type
*/

#include <iostream>
#include <string>

using namespace std;

//---------------------------------------------------------------------
template <class T>   // OR template <typename T>
void swapValues(T &x, T &y)
{
	T temp = x;
	x = y;
	y = temp;
}

//---------------------------------------------------------------------
void main()
{
	int     i1 = 1,    i2 = 2;
	double  d1 = 1.5,  d2 = 2.5;
	char    c1 = 'A',  c2 = 'B';
	 string s1="ABC",   s2="DEF";

	// NOTE:
	// the type attached to the template arguments
	// is inferred from the value argument list
	swapValues(i1,i2);
	swapValues(d1,d2);
	swapValues(c1,c2);
	swapValues(s1,s2);

	cout << "i1 = " << i1 << ", i2 = " << i2 << endl;
	cout << "d1 = " << d1 << ", d2 = " << d2 << endl;
	cout << "c1 = " << c1 << ", c2 = " << c2 << endl;
	cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
}

//=====================================================================
```

NOTE:

- It is not always possible to write a single template function that is suited for every possible template argument.
  - For example, the above template can not be used to swap C-strings.
- The solution is to use "**template specialization**". This is an advanced topic, which is out of the scope of this course.

```cpp
/*
FUNCTION TEMPLATES: example 2 (printing arrays)
*/

#include <iostream>
#include <cstddef>
#include <string>

using namespace std;

//-----------------------------------------------------------------------
template <typename T>  // OR template <class T>  //suggestion: use typename
void printArray(ostream &out, const T data[], size_t count)
{
	out << "[";
	for (size_t i = 0; i < count; i++)
	{
		if (i > 0)
			out << ", ";
		out << data[i];
	}
	out << "]";
}

//-----------------------------------------------------------------------
void main()
{

	int a[] = {10, 20, 30, 40, 50}; // an example of array initialization

	// call integer function-template specialization
	printArray(cout,a,5);
	cout << endl;

	double b[] = {1.1, 1.2, 1.3};
	// call double function-template specialization
	printArray (cout,b,3);
	cout << endl;

	string c[] = {"Mary", "John", "Fred"};
	// call string function-template specialization
	printArray (cout,c,3);
	cout << endl;
}
```

CLASS TEMPLATES / GENERIC CLASSES
(TEMPLATES FOR DATA ABSTRACTION)

```
/*
TEMPLATE CLASSES

IMPLEMENTATION OF A GENERIC "LINKED LIST" CLASS

Compare with previous example: linked list of integer values
Common solution to develop a Template function/class:
- develop a function/class with a 'fixed' type, then create the Template

*/

#include <iostream>
#include <cstddef>
#include <cassert>

using namespace std;

template <class T>  //instead of <class T> could have used <typename T>
class LinkedList {
public:
      LinkedList();
      size_t size() const;
      void insertEnd(T value);
      void insertBegin(T value);
      bool insertAfter(size_t index, T value);
      bool remove(T value);
      void clear();
      void display() const;
      ~LinkedList();
private:
      struct node{
            T data;
            node *next;
      } *p;
      size_t listSize;
};

//---------------------------------------------------------------------
// constructor
template <class T> //instead of <class T> could have used <typename T>
LinkedList<T>::LinkedList()
{
      p = NULL;
      listSize = 0;
}


//---------------------------------------------------------------------
// return the list size
template <class T>
size_t LinkedList<T>::size() const
{
      return listSize;
}
```

```
//---------------------------------------------------------------------
//insert a new node at the beginning of the linked list

template <class T>
void LinkedList<T>::insertBegin(T value)
{
      node *q;
      q = new node;
      q->data = value;
      q->next = p;
      p = q;
      listSize++;
}

//---------------------------------------------------------------------
// insert a new node at the end of the linked list
template <class T>
void LinkedList<T>::insertEnd(T value)
{
      node *q,*t;
      //if the list is empty
      if(p == NULL)
      {
            p = new node;
            p->data = value;
            p->next = NULL;
            listSize++;
      }
      else
      {
            q = p;
            while(q->next != NULL)
                  q = q->next;
            t = new node;
            t->data = value;
            t->next = NULL;
            q->next = t;
            listSize++;
      }
}

//---------------------------------------------------------------------
// insert a node at a specified location
// 'index' - location
// 'value' - contents of the node
template <class T>
bool LinkedList<T>::insertAfter(size_t index, T value)
{
      node *q, *t;
      size_t i;

      if (index > size()-1)
            return false;
      else
      {
            q = p;
            for (i = 0; i < index; i++)
                  q = q->next;
            t = new node;
            t->data = value;
            t->next = q->next;
```

```
            q->next = t;
            listSize++;

            return true;
      }
}

//-----------------------------------------------------------------------
// deletes the specified node from the linked list
// 'value' - contents of the node to be deleted
template <class T>
bool LinkedList<T>::remove(T value)
{
      node *q,*r;
      q = p;
      //if node to be deleted is the first node
      if (q->data == value)
      {
            p = p->next;
            delete q;
            listSize--;
            return true;
      }
      r = q;
      while(q != NULL)
      {
            if(q->data == value)
            {
                  r->next = q->next;
                  delete q;
                  listSize--;
                  return true;
            }
            r = q;
            q = q->next;
      }
      return false;
}

//-----------------------------------------------------------------------
// deletes all the list elements
template <class T>
void LinkedList<T>::clear()
{
      node *q;
      if( p == NULL )
            return;

      while( p != NULL )
      {
            q = p->next;
            delete p;
            p = q;
            listSize--;
      }

      //assert(listSize==0);
}
```

```
//---------------------------------------------------------------------
// shows all the linked list elements
template <class T>
void LinkedList<T>::display() const
{
	node *q;

	cout << "(" << listSize << "): ";
	for(q = p; q != NULL; q = q->next)
		cout <<  " " << q->data;
	cout << endl << endl;
}

//---------------------------------------------------------------------
// destructor
// MUST BE IMPLEMENTED WHEN DYNAMIC MEMORY HAS BEEN ALLOCATED
// to free all the memory allocated for the list nodes
template <class T>
LinkedList<T>::~LinkedList()
{
	clear();
}

//---------------------------------------------------------------------
void main()
{
	LinkedList<char> list;
	int index;
	char value;


	cout << "insertBegin() 'A', 'B', 'C', 'D', 'E'\n";
	for (value='A'; value<='E'; value++)
	{
		list.insertBegin(value);
		list.display();
	}

	//list.clear();
	//list.display();


	value = 'F';
	cout << "insertEnd() '" << value << "'\n";
	list.insertEnd(value);
	list.display();


	index = 0; value = 'G';
	cout << "insertAfter(" << index << ",'" << value << "')\n";
	if (!list.insertAfter(index,value))
		cout << "there is no such node index: " << index << endl;
	list.display();


	index = 10; value = 'H';
	cout << "insertAfter(" << index << ",'" << value << "')\n";
	if (!list.insertAfter(index,value))
		cout << "there is no such node index: " << index << endl;
	list.display();
```

```
    value = 'B';
    cout << "remove(" << value << ")\n";
    if (!list.remove(value))
          cout << "there is no such node value: " << value << endl;
    list.display();

    value = 'J';
    cout << "remove(" << value << ")\n";
    if (!list.remove(value))
          cout << "there is no such node value: " << value << endl;
    list.display();

    cout << endl;
}
```

```
// Class Templates
// Implementing a (circular) queue based on an array
// JAS

#include <iostream>
#include <cstddef>
#include <string>
#include <sstream>

using namespace std;

template <typename T>            // OR template <class T>
// template <typename T = int> // T defaults to 'int' = > "Queue < > q;" is possible
class Queue
{
public:
  static const size_t MAXSIZE = 10; //all queues have a max. size of 10; not flexible ...
  Queue();
  bool insertLast(T value);    // insert elemento into the queue
  bool removeFirst(T &value); // remove element from the queue
  size_t getNumElems() const; // get number of queue elements
private:
  T v[MAXSIZE];    // array elements are of type T
  size_t first;    // index of first queue element
  size_t last;     // index of last queue element
  size_t nElems;   // number of elements in queue
};

// ----------------------------------------------------------------------

template <typename T>
Queue<T>::Queue()
{
  first = 0;
  last = 0;
  nElems = 0;
}

// ----------------------------------------------------------------------

template <typename T>
bool Queue<T>::insertLast(T value)  //returns true if value was inserted
{
  if (nElems == 0)
  {
    v[last] = value;
    nElems = 1;
    return true;
  }
  else if (nElems < MAXSIZE)
  {
    last = (last + 1) % MAXSIZE;
    v[last] = value;
    nElems++;
    return true;
  }
  return false;
}
// ----------------------------------------------------------------------
```

```
template <typename T>
bool Queue<T>::removeFirst(T &value) //returns true if queue is not empty
{
  if (nElems > 0)
  {
    value = v[first];
    nElems--;
    if (nElems !=0)
      first = (first + 1) % MAXSIZE;
    return true;
  }
  return false;
}

// ------------------------------------------------------------------------

template <typename T>
size_t Queue<T>::getNumElems() const
{
  return nElems;
}

// ------------------------------------------------------------------------
// Converts integer to string
// 'n' - an integer value

string int_to_string(int n)
{
  ostringstream outstr;
  outstr << n;
  return outstr.str();
}

// ------------------------------------------------------------------------
int main()
{
  cout << "Max. queue size is " << Queue<string>::MAXSIZE << endl;
  //cout << "Max. queue size is " << Queue< >::MAXSIZE << endl; //possible when
T has a default type, e.g. int; see alternative template, in class Queue definition

  //-------------------------------------
  cout << "INTEGER QUEUE:\n";

  Queue<int> q;

  for (size_t i=1; i<=3; i++)      // try with other numbers of insertions
  {                                // or a different MAXSIZE
    if (q.insertLast(i))
      cout << i << " inserted\n";
    else
      cout << "full\n";
  }

  for (size_t i=1; i<=5; i++)
  {
    int value;
    if (q.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }
```

```
  //------------------------------------
  cout << endl;

  cout << "STRING QUEUE:\n";

  Queue<string> qs;

  for (size_t i=1; i<=5; i++)
  {
    string s = "value_" + int_to_string(i); // C++11:  to_string(n);
    if (qs.insertLast(s))
      cout << s << " inserted\n";
    else
      cout << "full\n";
  }

  for (size_t i=1; i<=6; i++)
  {
    string value;
    if (qs.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }

  cout << endl;

  return 0;
}
```

```
// Class Templates
// Implementing a (circular) queue based on an array
// Notes:
// - the template class has 2 parameters & accepts the size as a parameter
// - the 2nd parameter is a numeric value (defaults to 10), not a type
// JAS

#include <iostream>
#include <cstddef>
#include <string>
#include <sstream>

using namespace std;
// ----------------------------------------------------------------------
template <typename T,size_t MAXSIZE = 10>
class Queue
{
public:
  Queue();
  bool insertLast(T value);
  bool removeFirst(T &value);
  size_t getNumElems() const;
private:
  T v[MAXSIZE];
  size_t first;
  size_t last;
  size_t nElems;
};

// ----------------------------------------------------------------------

template <typename T, size_t MAXSIZE>
Queue<T,MAXSIZE>::Queue()
{
  first = 0;
  last = 0;
  nElems = 0;
}

// ----------------------------------------------------------------------

template <typename T, size_t MAXSIZE>
bool Queue<T,MAXSIZE>::insertLast(T value)
{
  if (nElems == 0)
  {
    v[last] = value;
    nElems = 1;
    return true;
  }
  else if (nElems < MAXSIZE)
  {
    last = (last + 1) % MAXSIZE;
    v[last] = value;
    nElems++;
    return true;
  }
  return false;
}
// ----------------------------------------------------------------------
```

```
template <typename T, size_t MAXSIZE>
bool Queue<T,MAXSIZE>::removeFirst(T &value)
{
  if (nElems > 0)
  {
    value = v[first];
    nElems--;
    if (nElems !=0)
      first = (first + 1) % MAXSIZE;
    return true;
  }
  return false;
}

// ----------------------------------------------------------------------

template <typename T, size_t MAXSIZE>
size_t Queue<T,MAXSIZE>::getNumElems() const
{
  return nElems;
}

// ----------------------------------------------------------------------
// Converts integer to string
// 'n' - an integer value

string int_to_string(int n)
{
  ostringstream outstr;
  outstr << n;
  return outstr.str();
}

// ----------------------------------------------------------------------

int main()
{
  cout << "INTEGER QUEUE:\n";
  Queue<int,5> q;

  int n=1;
  for (size_t i=1; i<=3; i++)
  {
    if (q.insertLast(n))
    {
      cout << n << " inserted\n";
      n++;
    }
    else
      cout << "full\n";
  }

  for (size_t i=1; i<=2; i++)
  {
    int value;
    if (q.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }
```

```
for (size_t i=1; i<=5; i++)
{
  if (q.insertLast(n))
  {
    cout << n << " inserted\n";
    n++;
  }
  else
    cout << "full\n";
}

for (size_t i=1; i<=10; i++)
{
  int value;
  if (q.removeFirst(value))
    cout << "removed " << value << "\n";
  else
    cout << "empty\n";
}

//-------------------------------------
cout << endl;
cout << "STRING QUEUE:\n";
Queue<string,5> qs;

for (size_t i=1; i<=5; i++)
{
  string s = "value_" + int_to_string(i);
  if (qs.insertLast(s))
    cout << s << " inserted\n";
  else
    cout << "full\n";
}

for (size_t i=1; i<=6; i++)
{
  string value;
  if (qs.removeFirst(value))
    cout << "removed " << value << "\n";
  else
    cout << "empty\n";
}

cout << endl;

//-------------------------------------
cout << "DOUBLE QUEUE:\n";
Queue<double> qd;  // NOTE: MAXSIZE defaults to 10

for (size_t i=1; i<=3; i++)
{
  if (qd.insertLast(i/1.2))
    cout << i << " inserted\n";
  else
    cout << "full\n";
}
```

```
  for (size_t i=1; i<=5; i++)
  {
    double value;
    if (qd.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }

  cout << endl;

  return 0;
}
```

```cpp
// Class Templates
// Implementing a (circular) queue
// using dynamically allocated memory
// JAS

#include <iostream>
#include <cstdlib> // malloc() + free()
#include <cstddef>
#include <string>
#include <sstream>

using namespace std;

template <typename T>          // OR template <typename T>
class Queue
{
public:
  Queue();
  Queue(size_t capac);
  ~Queue();  // destructor; must be implemented in this case
             // because memory is allocated dinamically
  bool insertLast(T value);   // insert elemento into the queue
  bool removeFirst(T &value); // remove element from the queue
  size_t getNumElems() const; // get number of queue elements
  size_t getCapacity() const; // get queue capacity
private:
  static const size_t MAXSIZE = 10; //all queues have a default size of 10;
  T *v;            // pointer to elements of type T
  size_t first;    // index of first queue element
  size_t last;     // index of last queue element
  size_t nElems;   // number of elements in queue
  size_t capacity; // capacity of the queue
};

// ----------------------------------------------------------------------

template <typename T>
Queue<T>::Queue()
{
  first = 0;

  last = 0;
  nElems = 0;
  v = new T [MAXSIZE];    // v = (T *) malloc (MAXSIZE * sizeof(T));
  capacity = MAXSIZE;
}
// ----------------------------------------------------------------------

template <typename T>
Queue<T>::Queue(size_t capac)
{
  first = 0;
  last = 0;
  nElems = 0;
  v = new T [capac];    // v = (T *) malloc (capac * sizeof(T));
  capacity = capac;
}

// ----------------------------------------------------------------------
```

```
template <typename T>
Queue<T>::~Queue()
{
  delete[] v;   // free(v);
}

// ----------------------------------------------------------------------

template <typename T>
bool Queue<T>::insertLast(T value)  //returns true if value was inserted
{
  if (nElems == 0)
  {
    v[last] = value;
    nElems = 1;
    return true;
  }
  else if (nElems < capacity)
  {
    last = (last + 1) % capacity;
    v[last] = value;
    nElems++;
    return true;
  }
  return false;
}
// ----------------------------------------------------------------------

template <typename T>
bool Queue<T>::removeFirst(T &value) //returns true if queue is not empty
{
  if (nElems > 0)
  {
    value = v[first];
    nElems--;
    if (nElems !=0)
      first = (first + 1) % capacity;
    return true;
  }
  return false;}

// ----------------------------------------------------------------------

template <typename T>
size_t Queue<T>::getNumElems() const
{
  return nElems;
}

// ----------------------------------------------------------------------
// Converts integer to string
// 'n' - an integer value
string int_to_string(int n)
{
  ostringstream outstr;
  outstr << n;
  return outstr.str();
}

// ----------------------------------------------------------------------
```

```
int main()
{
  //------------------------------------
  cout << "INTEGER QUEUE:\n";
  Queue<int> q(2);

  for (size_t i=1; i<=3; i++)
  {
    if (q.insertLast(i))
      cout << i << " inserted\n";
    else
      cout << "full\n";
  }

  for (size_t i=1; i<=5; i++)
  {
    int value;
    if (q.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }

 //------------------------------------
  cout << endl;
  cout << "DOUBLE QUEUE:\n";
  Queue<double> qd;

  for (size_t i=1; i<=5; i++)
  {
    double s = i * 1.25;
    if (qd.insertLast(s))
      cout << s << " inserted\n";
    else
      cout << "full\n";
  }

  for (size_t i=1; i<=6; i++)
  {
    double value;
    if (qd.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }

  cout << endl;

//------------------------------------
  cout << endl;
  cout << "STRING QUEUE:\n";
  Queue<string> qs(5);

  for (size_t i=1; i<=5; i++)
  {
    string s = "value_" + int_to_string(i);
    if (qs.insertLast(s))
      cout << s << " inserted\n";
    else
      cout << "full\n";
  }
```

```
  for (size_t i=1; i<=6; i++)
  {
    string value;
    if (qs.removeFirst(value))
      cout << "removed " << value << "\n";
    else
      cout << "empty\n";
  }

  cout << endl;

  return 0;
}
```

```
/*
TEMPLATE CLASSES
Parameterization with more than one type
*/
#include <iostream>
#include <string>

using namespace std;

template <typename T1, typename T2>
class Pair
 {
public:
     Pair(const T1 &f, const T2 &s);
     T1 getFirst() const;
     T2 getSecond() const;
     void show() const;
private:
     T1 first;
     T2 second;
};
//-----------------------------------------------------------------------
// constructor
template <typename T1, typename T2>
Pair<T1,T2>::Pair(const T1 &f, const T2 &s)
{
     first = f;
     second = s;
}
//-----------------------------------------------------------------------

template <typename T1, typename T2>
T1 Pair<T1,T2>::getFirst() const
{
     return first;
}
//-----------------------------------------------------------------------
template <typename T1, typename T2>
T2 Pair<T1,T2>::getSecond() const
{
     return second;
}
//-----------------------------------------------------------------------

template <typename T1, typename T2>
void Pair<T1,T2>::show() const
{
     cout << first << " - " << second << endl;
}
//-----------------------------------------------------------------------
void main()
{
     Pair<string,int> p1("John", 19);      // John' s grade
     Pair<int,string> p2(1,"F.C.Porto");   // 1st in football rank
     Pair<int,int> p3(2018,365);           // Number of days of year

     p1.show();
     p2.show();
     p3.show();

}
```

==

STL – STANDARD TEMPLATE LIBRARY

==

- **C++ Standard Library**
- **Standard Template Library (STL)**

- Recognizing that many data structures and algorithms are commonly used, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library.
- The STL defines powerful, template-based, reusable components that implement many **common data structures and algorithms** used to process those data structures.
- In the previous examples, we built linked lists; object space was allocated dinamically and objects were linked together with pointers.
- Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors, with no compiler complaints.
- Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work.
- An advantage of the STL is that you can reuse the STL **containers**, **iterators** and **algorithms** to implement common data representations and manipulations.
- Programming with the STL will enhance the **portability** of your code.

STL Containers

- data structures capable of storing objects of almost any data type
- 3 styles of containers
  - first-class containers
  - adapters
  - near containers

STL Iterators

- used by programs to manipulate the STL container elements
- have properties similar to those of pointers
- 5 categories

STL Algorithms

- functions that perform common data manipulation (ex: search, sort, ...)
- ~ 70 algorithms
- each algorithm has minimum requirements for the type of iterators that can be used with it
- a container's supported iterator type determines whether the container can be used with a specific algorithm.

Generic programming

- STL approach allows general program to be written so that the code does not depend on the underlying container

CONTAINERS

- First class containers
  - Sequence containers -
    represent linear data structures, such as vectors and linked lists
    - **vector**
    - deque
    - **list**
    - *NOTE: string - supports the same functionality as a sequence container, but stores only character data*
  - Associative containers -
    nonlinear containers that typically can locate elements quickly;
    can store sets of values or key - value pairs
    - **set**
    - **multiset**
    - **map**
    - **multimap**
- Container adapters
  - stack
  - queue
  - priority-queue
- Near containers (non-STL, but with STL-like characteristics and behaviors)
  - bitsets
  - valarrays
  - strings
  - C-like pointer-based arrays

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Fig. 22.1

Member function	Description
default constructor	A constructor to create an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.
`empty`	Returns `true` if there are no elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the first container is less than the second container; otherwise, returns `false`.
`operator<=`	Returns `true` if the first container is less than or equal to the second container; otherwise, returns `false`.
`operator>`	Returns `true` if the first container is greater than the second container; otherwise, returns `false`.

Fig. 22.2

Member function	Description
`operator>=`	Returns `true` if the first container is greater than or equal to the second container; otherwise, returns `false`.
`operator==`	Returns `true` if the first container is equal to the second container; otherwise, returns `false`.
`operator!=`	Returns `true` if the first container is not equal to the second container; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.
Functions found only in first-class containers	
`max_size`	Returns the maximum number of elements for a container.
`begin`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the first element of the container.
`end`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the next position after the end of the container.
`rbegin`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the last element of the container.

Fig. 22.2

Member function	Description
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the next position after the last element of the reversed container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

Fig. 22.2

Standard Library container header files	
<vector>	
<list>	
<deque>	
<queue>	Contains both queue and priority_queue.
<stack>	
<map>	Contains both map and multimap.
<set>	Contains both set and multiset.
<valarray>	
<bitset>	

Fig. 22.3

New C++11 containers

- Sequence containers:
  - array
  - forward_list
- Unordered associative containers:
  - unordered_set
  - unordered_multiset
  - unordered_map
  - unordered_multimap

- Unordered associative containers are containers that provide
  fast lookup of objects based on keys.
  Worst case complexity is linear
  but on average much faster for most of the operations.
  This is due to the use of hashing to store objects.
  - Elements in an unordered associative container are organized into buckets, keys with the same hash will end up in the same bucket.
    The number of buckets is increased when the size of the container increases to keep the average number of elements in each bucket under a certain value.
  - *Hashing will be treated in a future course.*
- The containers can still be iterated through like a regular associative container.
- NOTE: **unordered_set /unordered_map** containers are faster than **set/map** containers to access individual elements by their key,
  although they are generally less efficient for range iteration
  through a subset of their elements.

typedef	Description
`allocator_type`	The type of the object used to allocate the container's memory.
`value_type`	The type of element stored in the container.
`reference`	A reference to the type of element stored in the container.
`const_reference`	A constant reference to the type of element stored in the container. Such a reference can be used only for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer to the type of element stored in the container.
`const_pointer`	A pointer to a constant of the container's element type.
`iterator`	An iterator that points to an element of the container's element type.
`const_iterator`	A constant iterator that points to the type of element stored in the container and can be used only to *read* elements.
`reverse_iterator`	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.

Fig. 22.4

typedef	Description
`const_reverse_iterator`	A constant reverse iterator that points to the type of element stored in the container and can be used only to *read* elements. This type of iterator is for iterating through a container in reverse.
`difference_type`	The type of the result of subtracting two iterators that refer to the same container (operator `-` is not defined for iterators of `list`s and associative containers).
`size_type`	The type used to count items in a container and index through a sequence container (cannot index through a `list`).

Fig. 22.4

`*it` - dereferences iterator **`it`**

- **`it++`** - moves **`it`** to the next element of the container
- other available operators: **`--`**, **`==`**, **`!=`**
  `(<, <=, > or >= only for random-access iterators)`

- container member-functions
  - **`begin()`** – returns an iterator located at the 1st element in the container
  - **`end()`** – returns an iterator located one beyond the last element in the container

- Basic outline of how an iterator can cycle through all elements of a container:

```
STL_container<type>::iterator p;
for (p = container.begin(); p != container.end(); p++)
     process_element_at_location p;
```

Ex:

```
vector<int> v1;
// FILL v1 WITH SOME VALUES ...
vector<int>::iterator p;
for (p = v1.begin(); p != v1.end(); p++)
     cout << *p << endl; // ... AND SHOW THE VALUES
```

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Fig. 22.6

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Fig. 22.8

- The **iterator category** that each **container** supports determines whether the container can be used with specific **algorithms** in the STL
    - An algorithm that requires only forward iterators can be used with any container that supports forward, bidirectional or random-access iterators
    - But an algorithm that requires random-access iterators can be used only with containers that support random-access iterators

Predefined typedefs for iterator types	Direction of ++	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
const_reverse_iterator	backward	read

Fig. 22.9

Constant iterators

- a constant iterator is an iterator that
  does not allow you to change the element at its location

Reverse iterators

- a reverse iterator is an iterator that
  can be used to cycle through all elements of a container, in reverse order,
  provided that the container has bidirectional iterators;
- basic outline of how a reverse iterator can cycle through all elements of a container:

```
STL_container<type>::reverse_iterator p;
for (p = container.rbegin(); p != container.rend(); p++)
     process_element_at_location p;
```

Ex:

```
vector<int> v1;
// FILL v1 WITH SOME VALUES ...
vector<int>::reverse_iterator p;
for (p = v1.rbegin(); p != v1.rend(); p++)
     cout << *p << endl; // ... AND SHOW THE VALUES
```

- **`rbegin()`** – member-function that returns an iterator located at the last element
- **`rend()`** – member-function that returns a sentinel that marks the "end" of the elements in reverse order
- **note** that for a reverse_iterator, the increment operator, **++**,
  moves backward through the elements

Iterators and element insertion/removal

- note that when you insert or remove an element into or from a container,
  that can affect the other iterators
- in general, there is no guarantee that the iterators will be located at the same element after an addition or deletion
- some containers do, however, guarantee that the iterators will not be moved by additions or deletions, except if the iterator is located at an element that is removed (ex: list; *vector and deque make no such guarantee*)

Iterator operation	Description
All iterators	
++p	Preincrement an iterator.
p++	Postincrement an iterator.
Input iterators	
*p	Dereference an iterator.
p = p1	Assign one iterator to another.
p == p1	Compare iterators for equality.
p != p1	Compare iterators for inequality.
Output iterators	
*p	Dereference an iterator.
p = p1	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.

Fig. 22.10

Iterator operation	Description
Bidirectional iterators	
--p	Predecrement an iterator.
p--	Postdecrement an iterator.
Random-access iterators	
p += i	Increment the iterator p by i positions.
p -= i	Decrement the iterator p by i positions.
p + i *or* i + p	Expression value is an iterator positioned at p incremented by i positions.
p - i	Expression value is an iterator positioned at p decremented by i positions.
p - p1	Expression value is an integer representing the distance between two elements in the same container.
p[i]	Return a reference to the element offset from p by i positions
p < p1	Return true if iterator p is less than iterator p1 (i.e., iterator p is before iterator p1 in the container); otherwise, return false.

Fig. 22.10

Iterator operation	Description
p <= p1	Return true if iterator p is less than or equal to iterator p1 (i.e., iterator p is before iterator p1 or at the same location as iterator p1 in the container); otherwise, return false.
p > p1	Return true if iterator p is greater than iterator p1 (i.e., iterator p is after iterator p1 in the container); otherwise, return false.
p >= p1	Return true if iterator p is greater than or equal to iterator p1 (i.e., iterator p is after iterator p1 or at the same location as iterator p1 in the container); otherwise, return false.

Fig. 22.10

ALGORITHMS

Generic algorithms

- "Generic Algorithms" are template functions that use iterators as template parameters.

Classification 1

- Nonmodifying Algorithms
  - ex: find, max_elem, min_elem
- Modifying Algorithms
  - ex: copy, remove
- Sorting and Related Algorithms
  - ex: sort, merge
- Numeric Algorithms
  - ex: accumulate

Classification 2

- Initialization Algorithms
  - ex: fill, copy, generate
- Transformations
  - ex: sort, transform, reverse, random_shuffle
- Searching Algorithms
  - find, max_elem, min_elem, binary_search
- Removal and Replacement Algorithms
  - remove (=> *container*.erase() on the next program instruction), replace
- Other Algorithms
  - ex: count, count_if, accumulate

NOTE

- **Algorithms act on container elements**; they don't act on containers
- **parameters are iterators not containers**;
- the container properties (ex:size) remain the same

Example:

```
remove(numbers.begin(),numbers.end(),0); //remove zeros
```

- does not change the size of `numbers`;
- rather it moves the elements of `numbers` that are not equal to zero
  to the beginning of `numbers`
  and returns an iterator that points to the first element after them
- if one wants to discard the zeros, must do it explicitly, using `erase` method

```
numbers.erase(remove(numbers.begin(),numbers.end(),0),numbers.end());
```

Nonmodifying algorithms

- Algorithms that do not modify the container they operate upon.
- The declaration of the generic function `find` algorithm:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T &value);
```

- The declaration tells us that `find` works with
  any container that provides an iterator at least as strong as an **input iterator**.
- Type T objects must be equality comparable.

`Iter find(Iter first, Iter last, const T &value);`

- The generic algorithm **`find( )`** locates an element within a sequence.
  It takes three arguments.
- The first two specify a range: [first, last[,
  the third specifies a target value for the search.
- If requested value is **found**, `find( )` returns
  an iterator that points to the first element that is identical to the sought-after value.
- If the requested value is **not found**, `find( )` returns
  an iterator pointing one element past the final element in the sequence
  (that is the `last` iterator, *see above parameters*).

More nonmodifying algorithms

- **count**
  - counts occurrences of a value in a sequence
- **equal**
  - asks: are elements in two ranges equal ?
- **search**
  - looks for the first occurrence of a match sequence within another sequence
- **binary_search**
  - searches for a value in a sorted container.
    This is an efficient search for sorted sequences with random access iterators.

Modifying algorithms

- Container modifying algorithms change the content of the elements or their order.
- **copy**
  - copies from a source range to a destination range;
    this can be used to shift elements in a container to the left provided that
    the first element in the source range is not contained in the destination range.
- **remove**
  - removes all elements from a range equal to the given value.
  - must be followed by **erase()**
- **random_shuffle**
  - shuffles the elements of a sequence.

Sorting algorithms

- **sort**
  - sorts elements in a range in nondescending order,
    or in an order determined by a user-specified binary predicate.
- **merge**
  - merges two sorted source ranges into a single destination range.

Numeric algorithms

- **accumulate**
  - sums the elements in a container.

NOTE:
in general, generic algorithms do not alter the size of the containers they operate on
(see example about **`remove()`** algorithm)

==

STL – STANDARD TEMPLATE LIBRARY

ITERATORS

==

```
// STL - ITERATORS
// Pointers and iterators are similar
// An iterator is a generalization of a pointer

#include <iostream>
#include <vector>

using namespace std;

int main()
{
	const int SIZE = 6;

	int a[SIZE] = {1,2,3,4,5,6};

	vector<int> v(a,a+SIZE); // initializating a vector from an array, using one of the vector constructors

	// scanning an array, using a pointer
	cout << "a[] = { ";
	for (int *aPtr = a; aPtr != a + SIZE; aPtr++)
		cout << *aPtr << " ";
	cout << "}" << endl;

	// scanning a vector, using an iterator
	// note the similarity with the previous cycle
	cout << "v[] = { ";
	for (vector<int>::iterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
		cout << *vPtr << " "; // could / should be const_iterator ???
	cout << "}" << endl;

	return 0;
}
```

NOTE:
The use of an initializer list
(a list of initializers inside brackets ({ }))
  vector<int> v = {1,2,3,4,5};
is legal since C++11 but not legal in previous C++ standards

```
// STL - ITERATORS (another version of the previous program, using typedef)
// Pointers and iterators are similar
// An iterator is a generalization of a pointer
#include <iostream>
#include <vector>
using namespace std;

int main()
{
      const int SIZE = 6;

      int a[SIZE] = {1,2,3,4,5,6};

      vector<int> v(a,a+SIZE);

      typedef vector<int>::const_iterator vecIntIterator;

      // scanning an array, using a pointer
      cout << "a[] = { ";
      for (int *aPtr = a; aPtr != a + SIZE; aPtr++)
            cout << *aPtr << " ";
      cout << "}" << endl;

      // scanning a vector, using an iterator
      // note the similarity with the previous cycle
      cout << "v[] = { ";
      for (vecIntIterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
            cout << *vPtr << " ";
      cout << "}" << endl;

      return 0;
}
```

NOTE: Iterators in C++11

In C++11, the **auto** keyword makes this a little easier:

```
      for (auto vPtr = v.begin(); vPtr != v.end(); vPtr++)
            cout << *vPtr << endl;
```

Range-based for() loops

An even simpler syntax to allow us to iterate through sequences, called a range-based **for** statement (or “for each”):

```
      for (auto x: v) // OR    for (const auto &x: v)  => x can't be modified
            cout << x << endl;
```

You can translate this as “for each value of **x** in **v**”.

If you want to modify the value of **x**, you can make **x** a reference

```
      for (auto &x: v) // x can modified
            x = 10 * x ;
```

This syntax **works for C-style arrays** and anything that supports an iterator via **begin()** and **end()** functions. This includes all standard template library container classes (including string).

```
// STL - ITERATORS
// ::iterator and ::const_iterator
// Another example of template functions

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
void showVector(string vName, const vector<T> &v)
{
	cout << vName << "[] = { ";
	for (vector<T>::const_iterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
		cout << *vPtr << " ";
	cout << "}\n";
}


int main()
{
	const int SIZE = 5;

	int a[SIZE] = {1,2,3,4,5};

	vector<int> v1(a,a+SIZE);  // initializing a vector from an array

	vector<double> v2(10,0.1); // 10 elements, all equal to 0.1


	showVector("v1",v1);

	for (vector<int>::iterator vPtr = v1.begin(); vPtr != v1.end(); vPtr++)
		*vPtr = *vPtr * 10;

	showVector("v1",v1);


	showVector("v2",v2);

	// what is the result of this cycle ?
	for (vector<double>::iterator vPtr = v2.begin() + 1; vPtr != v2.end();
vPtr++)
		*vPtr = *vPtr + *(vPtr-1); //vector supports RANDOM ITERATORS

	showVector("v2",v2);

	return 0;
}
```

TO DO BY STUDENTS:

- rewrite the code using new C++11 allowed syntax for the cycle.

```
// STL - ITERATORS
// Using a reverse_iterator, rbegin() and rend()

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
void showVector(string vName, const vector<T> &v)
{
	cout << vName << "[] = { ";
	for (vector<T>::const_iterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
		cout << *vPtr << " ";
	cout << "}\n";
}

int main()
{
	const int SIZE = 5;

	int a[SIZE] = {1,2,3,4,5};
	vector<int> v1(a,a+SIZE);

	showVector("v1",v1);

	// What is the result of this loop ? (see previous program)
	for (vector<int>::iterator vPtr = v1.begin() + 1;
	     vPtr != v1.end(); vPtr++)
		*vPtr = *vPtr + *(vPtr-1);

	showVector("v1",v1);

	// What is the result of this loop ?
	// compare with previous loop

	for (vector<int>::reverse_iterator vPtr = v1.rbegin() + 1;
	     vPtr != v1.rend(); vPtr++)
		*vPtr = *vPtr + *(vPtr-1);

	showVector("v1",v1);

	return 0;
}
```

NOTE:

- there is no support for iterating in reverse order in the range-based for loops ☹

===

STL – STANDARD TEMPLATE LIBRARY

CONTAINERS

===

SEQUENCE CONTAINERS - VECTOR

```
// STL - SEQUENCE CONTAINERS - VECTOR
// insert(), erase() and clear() methods

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
void showVector(string vName, const vector<T> &v)
{
	typedef vector<T>::const_iterator constIterator;
	cout << vName << "[] = { ";
	for (constIterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
		cout << *vPtr << " ";
	cout << "}\n";
}

int main()
{
	const int SIZE1 = 10;
	const int SIZE2 = 2;
	const int SIZE3 = 3;

	vector<int> v1(SIZE1);
	vector<int> v2(SIZE2);
	int a[SIZE3] = {10,20,30};

	showVector("v1",v1);
	showVector("v2",v2);

	for (size_t i=0; i<v1.size(); i++)
		v1[i] = i;
	cout << endl;
	showVector("v1",v1);

	cout << "\nv1 status after ... \n\n";

	// inserting a value
	v1.insert(v1.begin()+2,-1);
	showVector("insert(v1.begin()+2, -1)",v1);
```

```
    // inserting a vector into another one (v2 into v1)
    // NOTE: v2.end() points past the end of the vector
    v1.insert(v1.begin()+5, v2.begin(), v2.end());
    showVector("insert(v1.begin()+5, v2.begin(), v2.end())",v1);

    // inserting an array into a vector (a into v1)
    // NOTE: a+SIZE3 points past the end of the array
    v1.insert(v1.begin(), a, a+SIZE3);
    showVector("insert(v1.begin(), a, a+SIZE3)",v1);

    cout << endl;

    // erasing an element
    v1.erase(v1.begin()+5);
    showVector("erase(v1.begin()+5)",v1);

    // erasing an element sequence
    v1.erase(v1.begin(),v1.begin()+3);
    showVector("erase(v1.begin(), v1.begin()+3)",v1);


    // clearing the vector
    v1.clear();
    showVector("clear()",v1);
    cout << endl;

    return 0;
}
```

SEQUENCE CONTAINERS - LIST

```
// STL - SEQUENCE CONTAINERS - LIST
// Some methods:
// push_front(), push_back(), insert(), remove() and sort()
// Compare with previous program for manipulating "linked lists"

#include <iostream>
#include <list>
#include <string>

using namespace std;

template <typename T>
void showList(string lstName, const list<T> &lst)
{
      typedef list<T>::const_iterator constIterator;
      cout << lstName << " = { ";
      for (constIterator lstPtr = lst.begin(); lstPtr != lst.end();
lstPtr++)
            cout << *lstPtr << " ";
      cout << "}\n\n";
}

void main()
{
      list<int> lst;  // why not list<int> list; ??? the compiler accepts it ...
      size_t index;
      int value;

      showList("lst",lst);

      cout << "push_front() 1, 2, 3, 4, 5\n";
      for (value=1; value<=5; value++)
      {
            lst.push_front(value);
            showList("lst",lst);
      }

      value = 6;
      cout << "push_back(" << value << ")\n";
      lst.push_back(value);
      showList("lst",lst);
```

```
    index = 2; value = 7;
    cout << "insertAt(" << index << ",'" << value << "')\n";
    //lst.insert(lst.begin()+index,value); //NOT ALLOWED lst.begin()+index
    if (index > lst.size())
        cout << "there is no such node index: " << index << endl;
    else
    { // TO DO : implement as a function
        list<int>::iterator lstIterator = lst.begin();
        for (size_t i = 0; i < index; i++)
            lstIterator++;  // alternative: use advance algorithm
        lst.insert(lstIterator,value);
    }
    showList("lst",lst);

    index = 10; value = 8;
    cout << "insertAt(" << index << ",'" << value << "')\n";
    //lst.insert(lst.begin()+index,value); //NOT ALLOWED lst.begin()+index
    if (index > lst.size())
        cout << "there is no such node index: " << index << endl;
    else
    { // TO DO : implement as a function
        list<int>::iterator lstIterator = lst.begin();
        for (size_t i = 0; i < index; i++)
            lstIterator++;  // alternative: use advance() algorithm
        lst.insert(lstIterator,value);
    }
    showList("lst",lst);

    value = 2;
    cout << "remove(" << value << ")\n";
    lst.remove(value); // returns 'void' ! No way to see if 'value' exists ...
                        // Do not confuse with remove() algorithm
    showList("lst",lst);

    cout << "sort()\n";
    lst.sort();  // NOTE: vector class does not have a sort() method
    showList("lst",lst);

    cout << endl;
}
```

NOTE:

- **list** supports bidirectional iterators, not random ones ...
- list::**erase**() - erases elements by their position (iterator)
    - complexity of **erase()** is O(1) for lists and O(n) for vectors
    - http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-summary.html
- list::**remove**() - removes elements by their value;
  do not confuse with **remove() algorithm**

SEQUENCE CONTAINER - DEQUE

- deque = "double ended queue"

- provides many of the benefits of a vector and a list in one container
  - efficient indexed access (using subscripting) for reading and modifying elements
    much like a vector
  - efficient insertion and deletion at its front and back
    much like a list
  - support for random-access iterators
    => can be used with all STL algorithms

- most common use: implementation of First-In-First-Out (FIFO) queues

- performance of operations
  - in general has higher overhead than vector
  - insertions and deletions in the middle is more efficient than in vectors
    but less efficient than in lists

```
// STL - SEQUENCE CONTAINERS - DEQUE

#include <iostream>
#include <cstddef>
#include <deque>

using namespace std;

template <typename T>
void displayDeque(const deque<T> &dq)
{
	for (size_t i=0; i<dq.size(); i++) cout << dq.at(i) << " ";
	cout << endl;
}

int main() {

	//  create an empty deque and fill with ints, using push_back()
	deque<int> dq;

	for (int i=0; i<10; ++i) {
		dq.push_back(i);
	}

	displayDeque(dq);

	//  add 3 numbers to the front of the deque, using push_front()
	for (int i=1; i<=3; ++i) {
		dq.push_front(10*i);
	}

	displayDeque(dq);

	//  remove first and last elements with pop_back() and pop_front()
	dq.pop_front();
	dq.pop_back();

	displayDeque(dq);

	//  insert 3 copies of the number -1
	dq.insert(dq.begin()+2, 3, -1);

	displayDeque(dq);

	//  declare raw array of ints
	int a[]= { 100, 200, 300, 400, 500 };
	//  insert the array using iterators
	dq.insert(dq.begin()+2, a, a+sizeof(a)/sizeof(int));

	displayDeque(dq);

	return 0;
}
```

ASSOCIATIVE CONTAINERS – MAP & MULTIMAP

```
// STL - Vectors and maps

#include <iostream>
#include <vector>
#include <map>

using namespace std;

int main()
{
  int n;


  vector<int> v;

  //v[0] = 10; // UNCOMMENT and interpret what happens
  //v[9] = 90;

  cout << "VECTOR\n";
  n=0;

  for (vector<int>::const_iterator vi=v.begin(); vi!=v.end(); vi++)
  {
    n++;
    cout << n << " - " << *vi << endl;
  }


  map <int,int> m;

  //m[0] = 10; //UNCOMMENT and interpret what happens
  //m[9] = 90;

  cout << "MAP\n";
  n=0;

  for (map<int,int>::const_iterator mi=m.begin(); mi!=m.end(); mi++)
  {
    n++;
    cout << n << " - " << mi->first << ", " << mi->second << endl;
  }

  /* ALTERNATIVE CODE:
  for (auto p : m)
  {
    n++;
    cout << n << " - " << p.first << ", " << p.second << endl;
    // NOTE: each element 'p' of 'm' is a pair!
  }
  */
      return 0;
}
```

```
// STL - maps

#include <iostream>
#include <map>
#include <utility>

using namespace std;

int main()
{

  map <int,int> m;

  int key = 20, k;
  cout << m[key] << endl; //NOTE: 'key' automatically inserted into map 'm'

  cout << "Key to search? "; cin >> k;
  if (m.find(k) != m.end())
    cout << "key " << k << " found in map 'm'\n";
  else
    cout << "key " << k << " NOT found in map 'm'\n";

  return 0;
}

//==================================================================

// STL - Vectors, maps and pairs

#include <iostream>
#include <vector>
#include <map>
#include <utility>  // needed for 'pair'

using namespace std;

int main()
{

  map<int,int> m;
  map<int,int>::const_iterator mi;

  pair<int,int> p;

  m[20]=10;
  m[5]=500;

  cout << "MAP\n";
  int n=0;
  for (mi=m.begin(); mi!=m.end(); mi++)
  {
    n++;
    p = *mi; // each element of a "map" is a "pair"; a "pair" is a template struct
    cout << n << " - " << p.first << ", " << p.second << endl;
  }

  //NOTE the order by which elements were presented

   return 0;
}
```

```
#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
  map <int, string> phone_user;

  phone_user.insert(pair<int, string> (1234, "Mary"));
  phone_user.insert(pair<int, string> (1234, "John")); //NOTE: key already used !
  phone_user.insert(pair<int, string> (2345, "Ann"));
  for (const auto & x : phone_user)
    cout << x.first << " - " << x.second << endl;

  return 0;
}
//==================================================================================

#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
  map <int, string> phone_user;
  int phoneNumber;
  string phoneUser;

  //create 'phone - user' map
  while (cout << "Phone number & User name (CTRL-Z to end)? ", cin >> phoneNumber
>> phoneUser)
  {
    pair <map <int, string>::iterator, bool> p;
    p = phone_user.insert(pair<int, string>(phoneNumber,phoneUser));
    if (p.second == false) // if insertion failed
      cout << "Phone number already associated to another user !\n";
  }

  // show 'phone - user' map contents
  for (const auto & x : phone_user)
    cout << x.first << " - " << x.second << endl;

  return 0;
}

//==================================================================================
```

```cpp
#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
  multimap <int, string> phone_user;

  phone_user.insert(pair<int, string>(1234, "Mary"));
  phone_user.insert(pair<int, string>(2345, "John"));
  phone_user.insert(pair<int, string>(1234, "Ann"));
 //phone_user.insert(pair<int, string>(1234, "Ann"));  // TRY THIS

  for (const auto & x : phone_user)
    cout << x.first << " - " << x.second << endl;

  return 0;
}
```

TO DO BY STUDENTS
Modify the program above in order to do the following:
- after creating the phone directory, ask the user for a phone number and show the list of all users that use that phone number.

Suggestion: investigate the use of the equal_range() method of multimap

```cpp
// STL - ASSOCIATIVE CONTAINERS - MAPS AND MULTIMAPS
// Telephone directory
// Adapted from Big C++ book

#include <iostream>
#include <string>
#include <map>

using namespace std;

//------------------------------------------------------------------------
// TelephoneDirectory maintains a map of name/number pairs.
class TelephoneDirectory
{
public:
     void add_entry(string name, unsigned int number);
     unsigned int find_number(string name) const;
     void print_all(ostream& out) const;
     void print_by_number(ostream& out) const;
private:
     map<string, unsigned int> database;
     typedef map<string, unsigned int>::const_iterator MapIterator;
};

//------------------------------------------------------------------------
/**
Add a new name/number pair to database.
@param name the new name
@param number the new number
*/
void TelephoneDirectory::add_entry(string name, unsigned int number)
{
     database[name] = number;
}

//------------------------------------------------------------------------
/**
Find the number associated with a name.
@param 'name' the name being searched
@return the associated number, or zero if not found in database
*/
unsigned int TelephoneDirectory::find_number(string name) const
{
     /*
     for (MapIterator p = database.begin(); p != database.end(); p++)
          if (p->first == name)
               return p->second;
     return 0; // not found
     */
     MapIterator p = database.find(name);
     if (p != database.end()) // if name was found
          return p->second;
     else
          return 0;
}
```

```cpp
//----------------------------------------------------------------------
/**
Print all entries on given output stream
in 'name:number' format, ordered by 'name'.
@param 'out' the output stream
*/
void TelephoneDirectory::print_all(ostream& out) const
{
      MapIterator current = database.begin();
      MapIterator stop = database.end();
      while (current != stop)
      {
            out << current->first << " : " << current->second << "\n";
            ++current;
      }
}

//----------------------------------------------------------------------
/**
Print all entries on given output stream
in 'name:number' format, ordered by 'number'.
@param 'out' the output stream
*/
void TelephoneDirectory::print_by_number(ostream& out) const
{
      multimap<unsigned int, string> inverse_database;
      typedef multimap<unsigned int, string>::iterator MMapIterator;

      MapIterator current = database.begin();
      MapIterator stop = database.end();
      while (current != stop)
      {
            inverse_database.insert(multimap<unsigned int, string>
                  ::value_type(current->second, current->first));
            // creates a PAIR object in which
            // 'first', of type 'unsigned int' is the KEY (=current->second) and
            // 'second', of type 'string', is the VALUE (=current->first)
            // ALTERNATIVE SOLUTIONS:
            // 1)
            // inverse_database.insert(
            //      pair<unsigned int, string> (current->second, current->first));
            //      //calls constructor for pair object;
            // 2) using make_pair()
            // pair<unsigned int, string> p = make_pair(current->second, current->first);
            // inverse_database.insert(p);
            ++current;
      }

      MMapIterator icurrent = inverse_database.begin();
      MMapIterator istop = inverse_database.end();
      while (icurrent != istop)
      {
            out << icurrent->first << " : " << icurrent->second << "\n";
            ++icurrent;
      }
}
```

```
//-----------------------------------------------------------------------
int main()
{
      TelephoneDirectory data;

      data.add_entry("Sarah", 227235591);
      data.add_entry("Mary", 223841212);
      // data.add_entry("Mary", 223841213); // UNCOMMENT AND INTERPRET RESULT
      data.add_entry("Fred", 223841212); //NOTE: repeated number


      cout << "Number for Mary " << data.find_number("Mary") << "\n";
      cout << "Number for John " << data.find_number("John") << "\n";

      cout << "\nPRINTING BY NAME \n";
      data.print_all(cout);

      cout << "\nPRINTING BY NUMBER \n";
      data.print_by_number(cout);
      return 0;
}
```

NOTE:

`map::value_type` (*see previous program*) is the type of object stored as an element in a map
`vector::value_type` is a type that represents the data type stored in a vector

For example, it is possible to write this:

```
vector<int>::value_type anIntegerValue;   // equivalent to: int anIntegerValue
anIntegerValue = 13;
cout << anIntegerValue << endl;
```

or even (*like in previous program*) !!!

```
cout << vector<int>::value_type(13); // equivalent to: cout << int(13);
```

```
// STL - ASSOCIATIVE CONTAINERS - MAPS
// What does this program do ...?!

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
      map <string, unsigned int> m;
      typedef map <string, unsigned int>::const_iterator MapIterator;

      string word;

      cout << "Write a text; end with <ENTER> followed by <CTRL-Z>\n";
      while (cin >> word)
            m[word]++;

      for (MapIterator i = m.begin(); i != m.end(); i++)
            cout << i->first << ": " << i->second << endl;
      //cout << (*i).first << " - " << (*i).second << endl; //alternative

      return 0;
}
```

```
// TEXT: A tooter who tooted a flute ...
a tutor who tooted a flute tried to tutor two tooters to toot
said the two to their tutor
is it harder to toot or to tutor two tooters to toot
```

TO DO BY STUDENTS: use a for() range-based loop

```cpp
// STL - ASSOCIATIVE CONTAINERS - MAPS
// A program for searching words in a map

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
      map <string, unsigned int> word_count;
      typedef map <string, unsigned int>::const_iterator MapIterator;

      string word;

      cout << "Write a text; end with <ENTER> followed by <CTRL-Z>\n";
      while (cin >> word)
            word_count[word]++;

      cout << endl;
      cout << "Word list:\n";
      for (MapIterator i = word_count.begin(); i != word_count.end(); i++)
            cout << i->first << ": " << i->second << endl;

     // Searching for a 'word' in 'word_count' map
     // BE CAREFUL !!! What happens when the 'word' does not exist in the map ?!

      cin.clear(); // WHY?  To be able to continue reading after CTRL-Z

      cout << endl;
      cout << "Word to search ? ";
      cin >> word;

      cout << "word_count[" << word << "] = " << word_count[word] << endl;
       // WHAT HAPPENS IF word DOES NOT BELONG THE THE MAP ...?

      cout << endl;
      cout << "Word list:\n"; //IMPLEMENT AS A FUNCTION ...? (see above code)
      for (MapIterator i = word_count.begin(); i != word_count.end(); i++)
            cout << i->first << ": " << i->second << endl;

      return 0;
}
```

```
// STL - ASSOCIATIVE CONTAINERS - MAPS
// Solution for the previous problem (searching for a word that does not exist)

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
	map <string, unsigned int> word_count;
	typedef map <string, unsigned int>::const_iterator MapIterator;

	string word;

	cout << "Write a text; end with <ENTER> followed by <CTRL-Z>\n";
	while (cin >> word)
		word_count[word]++;

	cout << endl;
	cout << "Word list:\n";
	for (MapIterator i = word_count.begin(); i != word_count.end(); i++)
		cout << i->first << ": " << i->second << endl;
	//cout << (*i).first << " - " << (*i).second << endl; //alternative

	cin.clear(); // To be able to continue reading after CTRL-Z
	cout << endl;
	cout << "Word to search ? ";
	cin >> word;

	//SOLUTION FOR THE PREVIOUS PROBLEM
	MapIterator itaux = word_count.find(word);
	if (itaux != word_count.end())
		cout << endl << "word_count[" << word << "] = " << word_count[word] << endl;
		//cout << endl << "word_count[" << word << "] = " << itaux->second << endl;

	else
		cout << "\"" << word << "\" not found !\n";

	cout << endl;
	cout << "Word list:\n";
	for (MapIterator i = word_count.begin(); i != word_count.end(); i++)
		cout << i->first << ": " << i->second << endl;

	return 0;
}
```

==

STL – STANDARD TEMPLATE LIBRARY

ALGORITHMS

http://www.sgi.com/tech/stl/
http://msdn.microsoft.com/en-us/library/c191tb28.aspx
http://msdn.microsoft.com/en-us/library/yah1y2x8.aspx

==

```
// STL - ALGORITHMS (some usage examples)

#include <iostream>
#include <cstddef>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
#include <ctime>

using namespace std;

void displayVec(string title, const vector<int> &v)
{
      cout << title << ": \n";
      for (size_t i=0; i<v.size(); i++)
            cout << setw(3) << v.at(i) << " ";
      cout << endl;

}

int myRand()
{
      return rand() % 10 + 1;
}

int main() {
      srand((unsigned) time(NULL));

      vector<int> v1(10);
      vector<int> v2(10);

      fill(v1.begin(), v1.end(),1);
      displayVec("v1 - fill(v1.begin(), v1.end(),1)",v1);

     // void fill_n (OutputIterator first, Size n, const T& val);
      fill_n(v1.begin()+4, 3, 2);
      displayVec("v1 - fill_n (v1.begin()+4, 3, 2)",v1);

      generate(v2.begin(),v2.end(),myRand);
      displayVec("v2 - generate(...,myRand)",v2);

      return 0;
}
```

```cpp
// STL - ALGORITHMS

#include <iostream>
#include <cstddef>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
#include <ctime>
#include <iterator> //ostream_iterator iterator

using namespace std;

void displayVec(string title, const vector<int> &v)
{
	cout << setw(37) << title << ":";
	for (size_t i=0; i<v.size(); i++)
		cout << setw(3) << v.at(i) << " ";
	cout << endl << endl;

}

int myRand()
{
	return rand() % 10 + 1;
}

int calcSquare(int value)  //calculates de square of 'value'
{
	return value * value;
}

bool equalsTwo(int value)
{
	return value == 2;
}

int main() {
	vector<int> v1(10);
	vector<int> v2(10);

	//fill(v1.begin(), v1.end(),1);
	//displayVec("fill(v1.begin(), v1.end(),1)",v1);

	//fill_n(v1.begin()+4, 3, 2);
	//displayVec("fill_n(v1.begin()+4, 3, 2)",v1);

	srand((unsigned) time(NULL));

	generate(v2.begin(),v2.end(),myRand);
	 displayVec("v2: generate(...,myRand)",v2);

	sort(v2.begin(),v2.end());
	 displayVec("sort(v2.begin(),v2.end())",v2);

	random_shuffle(v2.begin(),v2.end());
	displayVec("random_shuffle(v2.begin(),v2.end())",v2);

	vector<int> v3(v2.size());
	transform(v2.begin(),v2.end(),v3.begin(),calcSquare);
	 displayVec("v2->v3: transform(...,calculateSquare)",v3);

	vector<int> v4(v2.size());
	copy(v2.begin(),v2.begin()+5,v4.begin()+2);
	 displayVec("v4: copy(v2.begin(),v2.begin()+5,v4.begin()+2)",v4);
```

```
    // COMMENT BEFORE NEXT STEP: MERGE
    //reverse(v2.begin(),v2.end());
    //displayVec("reverse(v2.begin(),v2.end())",v2);

    //vector<int> v3(v1.size()+v2.size());
    //sort(v1.begin(),v1.end());
    //merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin()); //vectors must be sorted
    //displayVec("merge(v1...,v2...,v3.begin())",v3);

    cout << "value to remove from v2 ? ";
    int x; cin >> x;
    vector<int>::iterator p1 = remove(v2.begin(),v2.end(),x);
    displayVec("after remove() ",v2);  //NOTE: the elements past the new end ...
    // ... of the vector can still be accessed but their values are unspecified
    v2.erase(p1,v2.end());
    displayVec("after remove() + erase()",v2);

    //vector<int>::iterator p2 = remove_if(v2.begin(),v2.end(),equalsTwo);
    //v2.erase(p2,v2.end());
    //displayVec("after remove_if(....equalsTwo) + erase()",v2);

    //ostream_iterator<int> outputInt(cout); // create ostream_iterator for writing 'int' values to cout
    ostream_iterator<int> outputInt(cout," "); //ALTERNATIVE ostream_iterator (see above one)
    cout << "copy(v2.begin(),v2.end(),outputInt)" << endl;
    copy(v2.begin(),v2.end(),outputInt);
    cout << endl;

    /* // VERY TRICKY !!!
    cout << "Input 2 integers: ";
    istream_iterator<int> inputInt(cin);
    int n1 = *inputInt;
    inputInt++;
    int n2 = *inputInt;

    cout << "The sum is: ";
    *outputInt = n1 + n2; // outputInt declared above
    cout << endl;
    */

    return 0;
}
```

==

STL – STANDARD TEMPLATE LIBRARY

Summary

==

- **STL** - a library of classes that represent **containers** that occur frequently in computer programs in all application areas.
- Each class in the STL supports a relatively small set of operations. Basic functionality is extended through the use of **generic algorithms**.
- The 3 fundamental data structures are the vector, list, and deque.
- Vector and deque are indexed data structures; they support efficient access to each element based on an integer key.
- A list supports efficient insertion into or removal from the middle of a collection. Lists can also be merged with other lists.
- A set maintains elements in order. Permits very efficient insertion, removal, and testing of elements.
- A map is a keyed container.
  Entries in a map are accessed through a key,
  which can be any ordered data type.
  Associated with each key is a value.
  A multimap allows more than one value to be associated with a key.
- Stacks, queues, and priority queues are **adapters** built on top of the fundamental collections.
  A stack enforces the LIFO protocol, while the queue uses FIFO.

Some references

==

STL: http://www.sgi.com/tech/stl/
STL: http://msdn.microsoft.com/en-us/library/c191tb28.aspx
ALGORITHMs: http://msdn.microsoft.com/en-us/library/yah1y2x8.aspx