==========

CLASSES

==========

C++ the Object Based Paradigm

Object Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability.
- Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.
- The important features of object–oriented programming are:
    - Bottom–up approach in program design
    - Programs organized around objects, grouped in classes
    - Focus on data with methods to operate upon object's data
    - Interaction between objects through functions
    - Reusability of design through creation of new classes by adding features to existing classes
- Some examples of object-oriented programming languages are:
    - C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

(source: http://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.html)

Object:

- An object has state,
- exhibits some well-defined behaviour,
- and has a unique identity.

Class:

- A class describes a set of objects that share a common structure, and a common behaviour.
- A single object is an instance of a class.

Object-Oriented Analysis

- Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.
- The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions.
  They are modelled after real-world objects that the system interacts with.
  In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

- The **primary tasks in object-oriented analysis** (OOA) are:
  - Identifying objects
  - Organizing the objects by creating object model diagram
  - Defining the internals of the objects, or object attributes
  - Defining the behavior of the objects, i.e., object actions
  - Describing how the objects interact

- The common models used in OOA are use cases and object models.

Object-Oriented Design

- Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis.
  In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.
- The implementation details generally include:
  - Restructuring the class data (if necessary),
  - Implementation of methods, i.e., internal data structures and algorithms,
  - Implementation of control, and
  - Implementation of associations.

(source: http://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.html)

An example: a class Date

```
#include ...

...
class Date
{
public: // access specifier; users can only access the PUBLIC members
      Date();  // constructor; constructors have the name of the class
      Date(unsigned int y, unsigned int m,unsigned int d);
      Date(string yearMonthDay); // constructors can be overloaded
      void setYear(unsigned int y) ; // member function OR method
      void setMonth(unsigned int m) ;
      void setDay(unsigned int d) ;
      void setDate(unsigned int y, unsigned int m, unsigned int d) ;
      unsigned int getYear() ;
      unsigned int getMonth() ;
      unsigned int getDay() ;
      string getStr();  // get (return) date as a string
      void show();
private: // PRIVATE data & function members are hidden from the user
      unsigned int year; // data member
      unsigned int month;
      unsigned int day;
      // the date could have been represented internally as a string
      // the internal representation is hidden from the user
};   // NOTE THE SEMICOLON

Date::Date()  // constructors do not have a return type
{
// ... CONSTRUCTOR DEFINITION
}

Date::Date(unsigned int y, unsigned int m, unsigned int d)
{
   year = y;
   month = m;
   day  = d;
}
//... DEFINITION OF OTHER MEMBER FUNCTIONS

void Date::show()  //scope resolution is needed; other classes could have a show() method
{
// ...
}

int main()
{
      Date d1;
      Date d2(2011,03,18);
      Date d3("2011/03/18");

      d2.setDay(19);

      d2.show();

      string d2_str = d2.getStr();
      cout << d2_str << endl;

}
```

Classes in C++

- A class is a user-defined type.
- A class declaration specifies
  - the representation of objects of the class
  - and the set of operations that can be applied to such objects.

A *class* comprises:

- ***data members*** (or ***fields***) :
  each object of the class has its own copy of the data members (local state)
- ***member functions*** (or ***methods***):
  applicable to objects of the class

Data members

- describe the ***state*** of the objects
- they have a type, and are declared as:
  `type dataMemberId`

Member functions

- denote a **service** that objects offer to their clients
- the ***interface*** to such a service is specified by
  - its return type
  - and formal parameter(s):
    `returnType memberFunctId( formalParams )`
- In particular, a function with **`void`** return type
  usually indicates a function which modifies/shows the state of the object.

Access specifier

- a class may have several private and public sections
- keyword `public` marks the beginning of each public section
- keyword `private` marks the beginning of each private section
- by default, members (data and functions) are private
- normally, the **data members** are placed in **private** section(s)
  and the **function members** in **public** section(s)
- public members can be accessed by both member and nonmember functions

Constructor

- special function that is a member of the class and has the same name as the class
- does not have a return type
- is automatically called when an object of that class is created

```
/*
CLASSES
Fraction class (partial implementation)
TO DO:
- implement other arithmetic operations
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>

using namespace std;

class Fraction
{
public: //access-specifier
      Fraction(); // default constructor; constructors have the same name of the class
      Fraction(int num, int denom); // constructor overloading; parameterized constructor
     //   ~Fraction(); // destructor (sometimes not necessary, as in this case)
      void read();
      void setNumerator(int num);   // member function OR class method
      void setDenominator(int num); // mutator function
      int getNumerator() const; //const member functions can't modify the object that invokes it
      int getDenominator() const;   // accessor function
      bool isValid() const;
      void setValid(bool v);
      void show() const;
      void showAll() const;
      Fraction multiply(const Fraction &f);
      // Fraction divide(const Fraction &f);
      // Fraction sum(const Fraction &f);
      // Fraction subtract(const Fraction &f);
      void reduce();
private: //access-specifier
      int numerator; // data member OR attribute
      int denominator;
      bool valid; // fractions with denominator = 0    or   that
                  // were not read in the format "n/d" are considered invalid !!!
      int gcd(int x, int y) const; // can only be invoked inside class methods
}; // NOTE THE SEMICOLON

// ------------------------------------------------------------------------
// MEMBER FUNCTIONS DEFINITIONS
// ------------------------------------------------------------------------

// Constructs a fraction with numerator=0 and denominator=1
// Constructors DO NOT HAVE A RETURN TYPE
Fraction::Fraction()  // :: is named the scope resolution operator
{
      numerator = 0;
      denominator = 1;
      valid = true;
}
```

```
// Constructs a fraction with numerator=num and denominator=denom
Fraction::Fraction(int num, int denom)
{
      numerator = num;
      denominator = denom;
      valid = (denominator != 0);
}
//---------------------------------------------------------------------------
/* //UNCOMMENT AND INTERPRET WHAT HAPPENS
Fraction::~Fraction()
{
     cout << "fraction destroyed" << endl;
}
*/
//---------------------------------------------------------------------------
// Reads a fraction; fraction must have format 'numerator' / 'denominator'
void Fraction::read()
{
    string fractionString;
    char fracSymbol;
    int num;
    int denom;

    cout << "n / d ? ";  // should not be done here …?
    getline(cin,fractionString);

    istringstream fractionStrStream(fractionString);
    valid = false;
    if (fractionStrStream >> num >> fracSymbol >> denom)
    {
      numerator = num;
      denominator = denom;
      valid = (fracSymbol == '/' && denom !=0);
    }
}
//---------------------------------------------------------------------------
// Set fraction numerator to 'n'
void Fraction::setNumerator(int n)
{
      numerator = n;
}
//---------------------------------------------------------------------------
// Set fraction denominator to 'n'
void Fraction::setDenominator(int n)
{
      denominator = n;
       valid = (denominator != 0);
}
//---------------------------------------------------------------------------
// Set the valid fraction information
void Fraction::setValid(bool v)
{
      valid = v;
}
//---------------------------------------------------------------------------
// Returns the fraction numerator
int Fraction::getNumerator() const
{
      return numerator;
}
```

```
//---------------------------------------------------------------------------
// Returns the fraction denominator
int Fraction::getDenominator() const
{
      return denominator;
}
//---------------------------------------------------------------------------
// Returns the valid fraction information
bool Fraction::isValid() const
{
      return valid;
}
//---------------------------------------------------------------------------
// Multiply current fraction by fraction 'f'
Fraction Fraction::multiply(const Fraction &f)
{
      Fraction result;

      result.setNumerator(numerator * f.getNumerator());
      result.setDenominator(denominator * f.getDenominator());
      result.setValid(valid && f.isValid());

       result.reduce();

      return result;
}
//---------------------------------------------------------------------------
void Fraction::reduce()
{
      if (valid)
      {
            int n = gcd(numerator,denominator);
            numerator = numerator / n;
            denominator = denominator / n;
      }
}
//---------------------------------------------------------------------------
// Show fraction; format is 'numerator / denominator'
void Fraction::show() const
{
      cout << numerator << "/" << denominator;
}
//---------------------------------------------------------------------------
// Show fraction; format is 'numerator / denominator' followed by
// 'valid/invalid' information
void Fraction::showAll() const
{
      show();
      cout << (valid ? " valid" : " invalid") << endl << endl;
}
//---------------------------------------------------------------------------
```

```
// Compute greatest common divisor between 'x' and 'y'
// using Euclid's algorithm
int Fraction::gcd(int x, int y) const
{
     x = abs(x);  y=abs(y);  //  for dealing with negative numbers
     if (valid)
     {
          while (x != y)
          {
               if (x < y)
                    y = y - x;
               else
                    x = x - y;
          }
          return x;
     }
     else
          return 0; // impossible to calculate gcd
}
//-------------------------------------------------------------------------
// Defines and reads several fractions
// and executes some multiplication operations with them
int main()
{
     Fraction f1, f2(2,5), f3(5,7), f4(0,0), f5(1,0), f6, f7, f8;

     cout << "f1" << endl;
     f1.show(); cout << endl; //later on w'll see how can we do: cout << f1; ☺
     f1.showAll();

     // f1.Fraction(1,2); // can't invoke constructor on existing object
     f1 = Fraction(1,2); // ... but can do this :-) EXPLICIT CONSTRUCTOR CALL
     cout << "f1 new" << endl;
     f1.show(); cout << endl;
     f1.showAll();

     cout << "f2" << endl;
     f2.show(); cout << endl;
     f2.showAll();

     cout << "f3" << endl;
     f3.show(); cout << endl;
     f3.showAll();

     cout << "f4" << endl;
     f4.show(); cout << endl;
     f4.showAll();

     cout << "f5" << endl;
     f5.show(); cout << endl;
     f5.showAll();

     cout << "f6 = f2 * f3" << endl;
     f6 = f2.multiply(f3);  // assignment is defined for objects; comparison (==) is not
     f6.show(); cout << endl;
     f6.showAll();

     f6.reduce();
     cout << "f6 reduced" << endl;
     f6.show(); cout << endl;
     f6.showAll();
```

```
    cout << "f7 = f2 * f4" << endl;
    f7 = f2.multiply(f4);
    f7.show(); cout << endl;
    f7.showAll();

    cout << "f8 - ";
    f8.read();
    cout << "f8" << endl;
    f8.show(); cout << endl;
    f8.showAll();

    cout << "f8 = f6 * f8" << endl;
    f8 = f6.multiply(f8);
    f8.show(); cout << endl;
    f8.showAll();

    return 0;
}
```

NOTES:

- **INCLUDE A DEFAULT CONSTRUCTOR IN YOUR CLASSES SPECIALLY WHEN YOU DO CONSTRUCTOR OVERLOADING**
  - If you define no constructor the compiler will define a default constructor that does nothing
  - But if you only define a constructor with arguments, ex:
    `Fraction(int num, int denom);`
    no default constructor will be defined by the compiler;
    so, the following declaration will be illegal
    `Fraction f1;`

- **... UNLESS YOU DON'T WANT TO HAVE A DEFAULT CONSTRUCTOR**
  - Ex: what should a default constructor for the **Date** class do …?

- To call a constructor without arguments do
  this → **`Fraction f1;`**
  not this → `Fraction f1();`

- A constructor behaves like a function that returns an object of its class type. That is what happens when you do
  `f1 = Fraction(3,5);`

```
/*
Classes

Fraction class (partial implementation)

SOLUTION SIMILAR TO THE PREVIOUS ONE BUT USING the this POINTER

NOTE that by using this->
parameters can have the same name as the data members of the class
(not particularly useful...)

TO DO:
- implement reduceFraction
- implement other arithmetic operations
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>
#include <cstdlib>

using namespace std;

class Fraction
{
public:
      Fraction(); // default constructor
      Fraction(int numerator, int denominator); // alternative constructor
      void read();
      void setNumerator(int numerator);
      void setDenominator(int numerator);
      int getNumerator() const;
      int getDenominator() const;
      bool isValid() const;
      void setValid(bool v);
      void show() const;
      void showAll() const;
      Fraction multiply(const Fraction &f);
      // Fraction divide(const Fraction &f);
      // Fraction sum(const Fraction &f);
      // Fraction subtract(const Fraction &f);
      // Fraction subtract(const Fraction &f);
      void reduce();
private:
      int numerator;
      int denominator;
      bool valid;
      int gcd(int x, int y) const;
};
//-------------------------------------------------------------------------

// Constructs a fraction with numerator=0 and denominator=1
Fraction::Fraction()
{
      numerator = 0;
      denominator = 1;
      valid = true;
}
//-------------------------------------------------------------------------
```

```
// Constructs a fraction with numerator and denominator equal to the
parameter values
Fraction::Fraction(int numerator, int denominator)
{
      this->numerator = numerator;      // when member data & parameters
      this->denominator = denominator; // have the same name(s)
      this->valid = (denominator != 0);
}
//---------------------------------------------------------------------------
// Reads a fraction; fraction must have format 'numerator' / 'denominator'
void Fraction::read()
{
    string fractionString;
    char fracSymbol;
    int numerator;
    int denominator;

    cout << "n / d ? ";
    getline(cin,fractionString);

    istringstream fractionStrStream(fractionString);
    this->valid = false;
    if (fractionStrStream >> num >> fracSymbol >> denom)
    {
      this->numerator = numerator;
      this->denominator = denominator;
      this->valid = (fracSymbol == '/' && denom !=0);
    }
}
//---------------------------------------------------------------------------
// Set fraction numerator to 'numerator' value
void Fraction::setNumerator(int numerator)
{
      this->numerator = numerator;
}
//---------------------------------------------------------------------------
// Set fraction denominator to 'denominator' value
void Fraction::setDenominator(int denominator)
{
      this->denominator = denominator;
       valid = (denominator != 0);
}
//---------------------------------------------------------------------------
// Set the valid fraction information
void Fraction::setValid(bool valid)
{
      this->valid = valid;
}
//---------------------------------------------------------------------------
// Returns the fraction numerator
int Fraction::getNumerator() const
{
      return numerator;
}
//---------------------------------------------------------------------------
// Returns the fraction denominator
int Fraction::getDenominator() const
{
      return denominator;
}
```

```
//---------------------------------------------------------------------------
// Returns the valid fraction information
bool Fraction::isValid() const
{
     return valid;
}
//---------------------------------------------------------------------------
// Multiply current fraction by fraction 'f'
Fraction Fraction::multiply(const Fraction &f)
{
     Fraction result;

     result.setNumerator(this->numerator * f.getNumerator());
     result.setDenominator(this->denominator * f.getDenominator());
     result.setValid(valid && f.isValid());

     return result;
}
//---------------------------------------------------------------------------
void Fraction::reduce()
{
     if (valid)
     {
          int n = gcd(numerator,denominator);
          numerator = numerator / n;
          denominator = denominator / n;
     }
}
//---------------------------------------------------------------------------
// Show fraction; format is 'numerator / denominator'
void Fraction::show() const
{
     cout << numerator << "/" << denominator;
}
//---------------------------------------------------------------------------
// Show fraction; format is 'numerator / denominator' followed by
'valid/invalid' information
void Fraction::showAll() const
{
     show();
     cout << (valid ? " valid" : " invalid") << endl << endl;
}
//---------------------------------------------------------------------------
// Compute greatest common divisor between 'x' and 'y'- Euclid's algorithm
int Fraction::gcd(int x, int y) const
{
     x = abs(x);  y=abs(y);
     if (valid)
     {
          while (x != y)
          {
               if (x < y)
                    y = y - x;
               else
                    x = x - y;
          }
          return x;
     }
     else
          return 0; // impossible to calculate gcd
}
```

```
//------------------------------------------------------------------------
// Defines and reads several fractions
// and executes some multiplication operations with them
int main()
{
      Fraction f1, f2(2,5), f3(5,7), f4(0,0), f5(1,0), f6, f7, f8;

      cout << "f1" << endl;
      f1.show(); cout << endl;
      f1.showAll();

      cout << "f2" << endl;
      f2.show(); cout << endl;
      f2.showAll();

      cout << "f3" << endl;
      f3.show(); cout << endl;
      f3.showAll();

      cout << "f4" << endl;
      f4.show(); cout << endl;
      f4.showAll();

      cout << "f5" << endl;
      f5.show(); cout << endl;
      f5.showAll();

      cout << "f6 = f2 * f3" << endl;
      f6 = f2.multiply(f3);
      f6.show(); cout << endl;
      f6.showAll();

      f6.reduce();
      cout << "f6 reduced" << endl;
      f6.show(); cout << endl;
      f6.showAll();


      cout << "f7 = f2 * f4" << endl;
      f7 = f2.multiply(f4);
      f7.show(); cout << endl;
      f7.showAll();

      cout << "f8 - ";
      f8.read();
      cout << "f8" << endl;
      f8.show(); cout << endl;
      f8.showAll();

      cout << "f8 = f6 * f8" << endl;
      f8 = f6.multiply(f8);
      f8.show(); cout << endl;
      f8.showAll();

      return 0;
}
```

```
/*
Application for library management

class Book and Library - preliminary definition and implementation
class User - not yet defined

Using static class attributes
*/

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

using namespace std;

typedef unsigned long IdentNum;

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

class Book
{
public:
      Book();   //default constructor
      Book(string bookName);   //another constructor
      void setName(string bookName);
      IdentNum getId() const;
      string getName() const;
      void show() const;
private:
      static IdentNum numBooks; //static => only one copy for all objects
                                // no storage is allocated for numBooks
                                // numBooks must be defined outside the class
      IdentNum id; // each object has data members id and name
      string name;
};

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

class Library
{
public:
      Library(); //only the default constructor is declared
      void addBook(Book book);
      void showBooks() const;
private:
      vector<Book> books;
};
```

```
//-----------------------------------------------------------------------
// CLASS Book - MEMBER FUNCTIONS IMPLEMENTATION
//-----------------------------------------------------------------------

IdentNum Book::numBooks = 0; //static variable definition and initialization

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

Book::Book()
{
      numBooks++;
      id = numBooks;
      name = "UNKNOWN BOOK NAME";
}

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

Book::Book(string bookName)
{
      numBooks++;
      id = numBooks;
      name = bookName;
}

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

IdentNum Book::getId() const
{
      return id;
}

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

void Book::setName(string bookName)
{
      name = bookName;
}
//-----------------------------------------------------------------------

string Book::getName() const
{
      return name;
}
```

```
//------------------------------------------------------------------------
// CLASS Library - MEMBER FUNCTIONS IMPLEMENTATION
//------------------------------------------------------------------------

Library::Library()
{
      books.clear();   // clear() is a method from vector class
}

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

void Library::addBook(Book b)
{
      books.push_back(b);
}

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

void Library::showBooks() const
{
      for (size_t i=0; i<books.size(); i++)
            cout << books[i].getId() << " - " << books[i].getName() << endl;
}

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

int main()
{
      Library lib;

      Book b1;   // which constructor is used in each case ?
      Book b2("My First C++ Book");

      lib.addBook(b1);
      lib.addBook(b2);

      Book b3;

      string bookName;
      cout << "Book name ? ";
      getline(cin,bookName);
      b3.setName(bookName);

      lib.addBook(b3);

      lib.showBooks();
}
```

- What happens to the books when the application ends?

```
/*
Application for library management
class Book and Library - preliminary definition and implementation
class User - not yet defined
Using static attributes and methods in class declaration
Saving library books in a file
*/
#include <iostream>
#include <string>
#include <vector>
#include <cstddef>
#include <fstream>
#include <sstream>
using namespace std;

//-------------------------------------------------------------------------
// AUXILIARY TYPES - DEFINITION
//-------------------------------------------------------------------------

typedef unsigned long IdentNum;

//-------------------------------------------------------------------------
// CLASS Book - DEFINITION
//-------------------------------------------------------------------------
class Book
{
public:
	Book(); // default constructor
	Book(string bookName); //another constructor
	void setId(IdentNum num);
	void setName(string bookName);
	IdentNum getId() const;
	string getName() const;
	void show() const;
	static void setNumBooks(IdentNum n); //static method
	static IdentNum getNumBooks();
	// NOTE: can´t be "static IdentNum getNumBooks() const;"
	// static methods can only refer other static members of the class
private:
	static IdentNum numBooks; //static attribute declaration
                              //static => only one copy for all objects
                                // no storage is allocated for numBooks
                                // numBooks must be defined outside the class
	IdentNum id;
	string name;
};

//-------------------------------------------------------------------------
// CLASS Library - DEFINITION
//-------------------------------------------------------------------------
class Library
{
public:
	Library();
	void addBook(Book book);
	void showBooks() const;
	void saveBooks(string filename);
	void loadBooks(string filename);
private:
	vector<Book> books;

};
```

```
//-------------------------------------------------------------------------
// UTILITARY FUNCTIONS
//-------------------------------------------------------------------------

int string_to_int (string intStr)
{
      int n;
      istringstream intStream(intStr);
      intStream >> n;
      return n;
}

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

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

//-------------------------------------------------------------------------
// CLASS Book - STATIC ATTRIBUTE DEFINITION AND INITIALIZATION
//-------------------------------------------------------------------------

IdentNum Book::numBooks = 0;
// static variables MUST BE DEFINED (space is reserved), outside the class body;
// in this case, initialization is optional; by default, integers are initialized to zero

//-------------------------------------------------------------------------
// CLASS Book - IMPLEMENTATION
//-------------------------------------------------------------------------

Book::Book()
{
     // suggestion: do not increment numBooks is this case
     // useful for instantiating temporary books
      id = 0;
      name = "VOID";  // OR "" OR "UNKNOWN" ...
}

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

Book::Book(string bookName)
{
      numBooks++;
      id = numBooks;
      name = bookName;
}

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

IdentNum Book::getId() const
{
      return id;
}
```

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

string Book::getName() const
{
      return name;
}

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

IdentNum Book::getNumBooks() // NOTE: not "static IdentNum Book::getNumBooks()"
{
      return numBooks;
}
//-------------------------------------------------------------------------

void Book::setNumBooks(IdentNum n) // NOTE: not "static void Book::setNumBooks(IdentNum n)"
{
      numBooks = n;
}

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

void Book::setId(IdentNum num)
{
      id = num;
}

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

void Book::setName(string bookName)
{
      name = bookName;
}

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

void Book::show() const
{
            cout << id << " - " << name << endl;
}

//-------------------------------------------------------------------------
// CLASS Library - IMPLEMENTATION
//-------------------------------------------------------------------------

Library::Library()
{
      books.clear();
}
//-------------------------------------------------------------------------

void Library::addBook(Book b)
{
      books.push_back(b);
}

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

```
void Library::showBooks() const
{
	cout << "\n-----------BOOKS-----------\n";
	for (size_t i=0; i<books.size(); i++)
		cout << books[i].getId() << " - " << books[i].getName() << endl;
	cout << "---------------------------\n\n";

}

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

void Library::saveBooks(string filename)
{
	ofstream fout;

	fout.open(filename);
	if (fout.fail( ))
	{
		cout << "Output file opening failed.\n";
		exit(1);
	}

	fout << Book::getNumBooks() << " (last book ID)" << endl << endl;

	for (size_t i=0; i<books.size(); i++)
	{
		fout << books[i].getId() << endl;
		fout << books[i].getName() << endl << endl;
	}

	cout << books.size()  << " books saved in file " << filename << endl;

	fout.close();
}

void Library::loadBooks(string filename)
{
	ifstream fin;
	IdentNum numBooks;
	string bookIdStr;
	string emptyLine;
	//IdentNum bookId;
	string bookName;

	// a static method may be called independent of any object,
	// by using the class name and the scope resolution operator
	// but may also be called in connection with an object (see end of main() function)
	Book::setNumBooks(0);

	books.clear();

	fin.open(filename);
	if (fin.fail( ))
	{
		cout << "Input file opening failed.\n";
		exit(1);
	}
```

```
	fin >> numBooks; fin.ignore(100,'\n');
	getline(fin,emptyLine);
	cout << "'numBooks' obtained from file " << filename << ": " <<
numBooks  << endl;

	for (size_t i=0; i<numBooks; i++)
	{
		getline(fin,bookIdStr); //NOTE: compare with Library::saveBooks()
		//bookId = string_to_int(bookIdStr);
		getline(fin,bookName);
		getline(fin,emptyLine);

		Book b(bookName);
		books.push_back(b);
	}

	cout << books.size() << " books loaded from file " << filename <<
endl;

	fin.close();
}

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

int main()
{
	Library lib;

	Book b1("My First C++ Book");
	Book b2("My Second C++ Book");
	lib.addBook(b1);
	lib.addBook(b2);
	cout << "2 books added to the library\n";

	lib.showBooks();

	lib.saveBooks("bookfile.txt");

	lib.loadBooks("bookfile.txt");

	Book b3("Big C++");
	lib.addBook(b3);
	cout << "1 book added to the library\n";

	lib.showBooks();
	//cout << "numBooks = " << b1.getNumBooks() << endl; // b1.getNumBooks() is a valid call
}
```

```
/*
Application for library management
class Book and Library - preliminary definition and implementation
class User - not yet defined

NOTE:
THESE ARE JUST EXAMPLES OF CLASS USE,
NOT THE WAY YOU SHOULD IMPLEMENT A LIBRARY PROJECT

Using two different constructors for Library class
Using a destructor - NOT USUALLY USED FOR THE ILLUSTRATED PURPOSE
*/

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

using namespace std;


//-------------------------------------------------------------------------
// AUXILIARY TYPES - DEFINITION
//-------------------------------------------------------------------------

typedef unsigned long IdentNum;


//-------------------------------------------------------------------------
// CLASS Book - DEFINITION
//-------------------------------------------------------------------------

class Book
{
public:
      Book();
      Book(string bookName);
      IdentNum getId() const;
      string getName() const;
      void setId(IdentNum num);
      void setName(string bookName);
      void show() const;
      static IdentNum getNumBooks();
      static void setNumBooks(IdentNum n);
private:
      static IdentNum numBooks;
      IdentNum id;
      string name;
};
```

```
//------------------------------------------------------------------------
// CLASS Library - DEFINITION
//------------------------------------------------------------------------

class Library
{
public:
      Library();                        //default constructor
      Library(string filename);         //another constructor
      ~Library();                       //destructor
      void addBook(Book book);
      void showBooks() const;
      void saveBooks(string filename);
      void loadBooks(string filename);
private:
      string libraryFilename;
      vector<Book> books;
};

//------------------------------------------------------------------------
// UTILITARY FUNCTIONS
//------------------------------------------------------------------------

int string_to_int (string intStr)
{
      int n;
      istringstream intStream(intStr);
      intStream >> n;
      return n;
}
//------------------------------------------------------------------------

bool fileExists(string filename)
{
      ifstream fin(filename);
      if (fin.is_open())
      {
            fin.close();
            return true;
      }
      else
            return false;
}



//------------------------------------------------------------------------
// CLASS Book - STATIC ATTRIBUTE INITIALIZATION
//------------------------------------------------------------------------

IdentNum Book::numBooks = 0;
```

```
//------------------------------------------------------------------------
// CLASS Book - IMPLEMENTATION
//------------------------------------------------------------------------

Book::Book()
{
      id = 0;
      name = "VOID";
}
//------------------------------------------------------------------------

Book::Book(string bookName)
{
      numBooks++;
      id = numBooks;
      name = bookName;
}
//------------------------------------------------------------------------

IdentNum Book::getId() const
{
      return id;
}

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

string Book::getName() const
{
      return name;
}
//------------------------------------------------------------------------

IdentNum Book::getNumBooks() // NOTE: not "static IdentNum Book::getNumBooks()"
{
      return numBooks;
}
//------------------------------------------------------------------------

void Book::setId(IdentNum num)
{
      id = num;
}
//------------------------------------------------------------------------

void Book::setName(string bookName)
{
      name = bookName;
}
//------------------------------------------------------------------------

void Book::show() const
{
            cout << id << " - " << name << endl;
}
//------------------------------------------------------------------------

void Book::setNumBooks(IdentNum n) // NOTE: not "static void Book::setNumBooks(IdentNum n)"
{
      numBooks = n;
}
```

```
//-------------------------------------------------------------------------
// CLASS Library - IMPLEMENTATION
//-------------------------------------------------------------------------

Library::Library()
{
      books.clear();
}

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

Library::Library(string filename)
{
      libraryFilename = filename;

      if (fileExists(libraryFilename))
            loadBooks(libraryFilename);
      else
            books.clear();

}

//-------------------------------------------------------------------------
Library::~Library()     //
DESTRUCTOR
{
     // cout << "Library destructor was called.\n";
      saveBooks(libraryFilename);
}

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

void Library::addBook(Book b)
{
      books.push_back(b);
}

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

void Library::showBooks() const
{
      cout << "\n-----------BOOKS-----------\n";
      for (size_t i=0; i<books.size(); i++)
            books[i].show();
      cout << "---------------------------\n\n";

}

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


void Library::saveBooks(string filename)
{
      ofstream fout;

      fout.open(filename);
      if (fout.fail( ))
      {
            cout << "Output file opening failed.\n";
            exit(1);
      }
```

```
	fout << Book::getNumBooks() << " (last book ID)" << endl << endl;

	for (size_t i=0; i<books.size(); i++)
	{
		fout << books[i].getId() << endl;
		fout << books[i].getName() << endl << endl;
	}

	cout << books.size()  << " books saved in file " << filename << endl;

	fout.close();
}

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

void Library::loadBooks(string filename)
{
	ifstream fin;
	IdentNum nBooks;
	string bookIdStr;
	string emptyLine;
	IdentNum bookId;
	string bookName;
	books.clear();

	fin.open(filename);
	if (fin.fail( ))
	{
		cout << "Input file opening failed.\n";
		exit(1);
	}

	fin >> nBooks; fin.ignore(100,'\n');
	getline(fin,emptyLine);
	cout << "'numBooks' obtained from file " << filename << ": "
		<< nBooks  << endl;

	Book::setNumBooks(nBooks);
	// a static method may be called independently of any object,
	// by using the class name and the scope resolution operator

	for (size_t i=0; i<nBooks; i++)
	{
		getline(fin,bookIdStr);
		bookId = string_to_int(bookIdStr);
		getline(fin,bookName);
		getline(fin,emptyLine);

		Book b;
		b.setId(bookId);
		b.setName(bookName);
		books.push_back(b);
	}

	cout << books.size() << " books loaded from file " << filename <<
endl;

	fin.close();
}
```

```
//---------------------------------------------------------------------
// main()
//---------------------------------------------------------------------

int main()
{
      //books are loaded by Library constructor
     Library lib("bookfile.txt");

      Book b1("My First C++ Book");
      Book b2("My Second C++ Book");
      lib.addBook(b1);
      lib.addBook(b2);
      cout << "2 books added to the library\n";

      lib.showBooks();

      Book b3("Accelerated C++");
      lib.addBook(b3);
      cout << "1 book added to the library\n";

      lib.showBooks();

      //books are saved by Library destructor
}
```

In Library class, an alternative implementation could define:

```
vector<*Book> books;
```

Do you see any advantage / disadvantage ?

Think what happens when you add a **User 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.

Separate compilation & Abstract Data Types (ADTs)

Until now ... small programs

- code placed into a single file
- typical layout
  - initial comments – what is the program purpose
  - included header files
  - constants
  - typedef's and classes
  - function prototypes (if any)
  - global variables (if any)
  - function / class implementation (+ comments)
  -

When programs get larger or you work in a team ...

- need to separate code into separate source files
- reasons for separating code
  - only those files that you changed need to be recompiled
  - each programmer is solely responsible for a separate set of files (editing of common files is avoided)

C++ allows you to divide a program into parts

- each part can be stored into a separate file
- each part can be compiled separately
- a class definition can be stored separately from a program
- this allows you to use the class in multiple programs

Header files (interface)

- files that define types or functions that are needed in other files
- are a path of communication between the code
- contain
  - definitions of constants
  - definitions of types / classes
  - declarations of non-member functions
  - declarations of global variables

Implementation files

- contain
  - definitions of member functions
  - definitions of nonmember functions
  - definitions of global variables

Abstract Data Types (ADTs)

- An ADT is a class defined to separate the interface and the implementation
- All member variables are private
- The class definition along with the function and operator declarations are grouped together as the interface of the ADT
- Group the implementation of the operations together and make them available to the programmer using the ADT
- The public part of the class definition is part of the ADT interface
- The private part of the class definition is part of the ADT implementation
  - This hides it from those using the ADT
- C++ does not allow splitting the public and private parts of the class definition across files
- The entire class definition is usually in the interface file

Example: a Book ADT interface

- The Book ADT interface is stored in a file named book.h
- The .h suffix means this is a header file
- Interface files are always header files
- A program using book.h must include it using an include directive
  - #include "book.h"

#include < > OR #include " " ?

- To include a predefined header file use < ..... >
  - #include <iostream>
- < ..... > tells the compiler to look where the system stores predefined header files
- To include a header file you wrote use "....."
  - #include "book.h"
- " ..... " usually causes the compiler to look in the current directory for the header file

The Implementation File

- Contains the definitions of the ADT functions
- Usually has the same name as the header file but a different suffix
- Since our header file is named book.h, the implementation file is named book.cpp
- The implementation file requires an include directive to include the interface file:
  - #include "book.h"

The Application File

- The application file is the file that contains the program that uses the ADT
  - It is also called a driver file
  - Must use an include directive to include the interface file:
    - #include "book.h"

Running The Program

- Basic steps required to run a program:
  (details vary from system to system)
  - Compile the implementation file
  - Compile the application file
  - Link the files to create an executable program using a utility called a linker
    - Linking is often done automatically

Compile book.h ?

- The interface file is not compiled separately
  - The preprocessor replaces any occurrence of #include "book.h"
    with the text of book.h before compiling
  - Both the implementation file and the application file contain #include "book.h"
    - The text of book.h is seen by the compiler in each of these files
    - There is no need to compile book.h separately

Why Three Files?

- Using separate files permits
  - The ADT to be used in other programs
    without rewriting the definition of the class for each
  - Implementation file to be compiled once
    even if multiple programs use the ADT
  - Changing the implementation file
    does not require changing the program using the ADT

Reusable Components

- An ADT coded in separate files can be used over and over
- The reusability of such an ADT class
  - Saves effort since it does not need to be
    - Redesigned
    - Recoded
    - Retested
  - Is likely to result in more reliable components

Multiple Classes

- A program may use several classes
  - Each could be stored in its own interface and implementation files
  - Some files can "include" other files, that include still others
  - It is possible that the same interface file could be included in multiple files
  - C++ does not allow multiple declarations of a class
  - The #ifndef directive can be used to prevent multiple declarations of a class

Using #ifndef directive

- Consider this code in the interface file

```
#ifndef BOOK_H
#define BOOK_H
// the Book class definition goes here
#endif
```

- To prevent multiple declarations of a class, we can use these directives:
  - **#define BOOK_H**
    - adds BOOK_H to a list indicating BOOK_H has been seen
  - **#ifndef BOOK_H**
    - checks to see if BOOK_H has been defined
  - **#endif**
    - if BOOK_H has been defined, skip to #endif

- The first time a #include "book.h" is found,
  BOOK_H and the class are defined
- The next time a #include "book.h" is found,
  all lines between #ifndef and #endif are skipped

- NOTE: **#pragma once** is a non-standard but widely supported preprocessor directive designed to cause the current source file to be included only once in a single compilation; as it is non-standard (yet) its use is not recommended.

Why BOOK_H ?

- BOOK_H is the normal convention for creating an identifier to use with #ifndef
  - it is the file name in all caps
  - use ' _ ' instead of ' . '
- You may use any other identifier, but will make your code more difficult to read

Defining Libraries

- You can create your own libraries of functions
  - You do not have to define a class to use separate files
  - If you have a collection of functions…
    - Declare them in a header file with their comments
    - Define them in an implementation file
    - Use the library files just as you use your class interface and implementation files

```
//---------------------------------------------------------------------------
// SOME UTILITARY FUNCTIONS
//---------------------------------------------------------------------------

int string_to_int (string intStr)
{
      int n;
      istringstream intStream(intStr);
      intStream >> n;
      return n;
}

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

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

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

bool fileExists(string filename)
{
      ifstream fin(filename);
      if (fin.is_open())
      {
            fin.close();
            return true;
      }
      else
            return false;
}

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

// =>
// #include <cstdio>
// #include <cstdlib>

string getTmpFilename() //Microsoft compiler specific
{
      char fnameC[L_tmpnam_s];
      errno_t err;
      err = tmpnam_s( fnameC, L_tmpnam_s ); //safe version of tmpnam()
      if (err)
            return "";
      else
      {
            string fname(fnameC);
            // tmpnam_s returns names like "\s52k.", "\s1sc.", ...
            // in the following instruction the '\' and the '.' are removed
            fname = fname.substr(1,fname.length()-2);
            return fname;
      }
}
```

Separate compilation - an example : Proj_sep_comp

```
//=========================================================================
// FUNCTIONS.H
//-------------------------------------------------------------------------
// SOME UTILITARY FUNCTIONS
//-------------------------------------------------------------------------

#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <string>

using namespace std;

//-------------------------------------------------------------------------
// Converts string to integer
// 'intStr' - string representing a valid integer

int string_to_int (string intStr);

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

string int_to_string(int n);

//-------------------------------------------------------------------------
// Tests if a file exists
// 'filename' - file whose existence is being tested

bool fileExists(string filename);

//-------------------------------------------------------------------------
// Returns a temporary filename

string getTmpFilename();

#endif
```

```
//=========================================================================
// FUNCTIONS.CPP
//-------------------------------------------------------------------------
// SOME UTILITARY FUNCTIONS
//-------------------------------------------------------------------------

#include <string>
#include <fstream>
#include <sstream>

#include "functions.h"
```

```
//---------------------------------------------------------------------------
// Converts string to integer
// 'intStr' - string representing a valid integer

int string_to_int (string intStr)
{
      int n;
      istringstream intStream(intStr);
      intStream >> n;
      return n;
}

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

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

//---------------------------------------------------------------------------
// Tests if a file exists
// 'filename' - file whose existence is being tested

bool fileExists(string filename)
{
      ifstream fin(filename);
      if (fin.is_open())
      {
            fin.close();
            return true;
      }
      else
            return false;
}

//---------------------------------------------------------------------------
// Returns a temporary filename

string getTmpFilename() //Microsoft compiler specific
{
      char fnameC[L_tmpnam_s];
      errno_t err;
      err = tmpnam_s( fnameC, L_tmpnam_s ); //safe version of tmpnam()
      if (err)
            return "";
      else
      {
            string fname(fnameC);
            // tmpnam_s returns names like "\s52k.", "\s1sc.", ...
            // in the following instruction the '\' and the '.' are removed
            fname = fname.substr(1,fname.length()-2);
            return fname;
      }
}
```

```
//=============================================================================
// MAIN.CPP (only used for testing the developed functions)
//-----------------------------------------------------------------------------

#include <iostream>

#include "functions.h"

using namespace std;

int main(void)
{
	cout << string_to_int ("2011") << endl;

	cout << int_to_string(10) << endl;

	cout << "fileExists(\"book.txt\") = " << fileExists("book.txt") <<
endl;

	cout << "temporary file name = " << getTmpFilename() << endl;

	return 0;
}
```

NOTE:

- To compile the functions without having a main() function, in Visual Studio, use Build > Compile (Ctrl+F7)

Separate compilation - another example

```
//=============================================================================
// DEFS.H  (no DEFS.CPP)
//-----------------------------------------------------------------------------

#ifndef DEFS_H
#define DEFS_H

typedef unsigned int IdentNum;

#endif
//=============================================================================

//=============================================================================
// USER.H
//-----------------------------------------------------------------------------

#ifndef USER_H
#define USER_H

#include <string>
#include <vector>
#include "defs.h"

using namespace std;

class User {

private:
      static IdentNum numUsers; //total number of users - used to obtain ID of each new user
      IdentNum ID; // unique user identifier (unsigned integer)
      string name; // user name
      bool active; // only active users can request books
      vector<IdentNum> requestedBooks; // books presently loaned to the user

public:
      //constructors
      User();
      User(string name);

      //get methods
      IdentNum getID() const;
      string getName() const;
      bool isActive() const;
      vector<IdentNum> getRequestedBooks() const;

      bool hasBooksRequested() const;

      //set methods
      void setID(IdentNum userID);
      void setName (string userName);
      void setActive(bool status);
      void setRequestedBooks(const vector<IdentNum> &booksRequestedByUser);
      static void setNumUsers(IdentNum num);

      void borrowBook(IdentNum bookID);
      void returnBook(IdentNum bookID);
};
```

```
#endif

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


//============================================================================
// BOOK.H
//----------------------------------------------------------------------------

#ifndef BOOK_H
#define BOOK_H

#include <string>

#include "defs.h"

using namespace std;

class Book {

private:
	static IdentNum numBooks; //total number of books - used to obtain ID of each new book
	IdentNum ID; // unique book identifier (unsigned integer)
	string title; // book title
	string author; // book author OR authors
	unsigned int numAvailable; // number of available items with this title

public:
	//constructors
	Book();
	Book(string bookTitle, string bookAuthor, unsigned int bookQuantity);

	//get methods
	IdentNum getID() const;
	string getTitle() const;
	string getAuthor() const;
	unsigned int getNumAvailable() const;

	//set methods
	void setID(IdentNum bookID);
	void setTitle(string bookTitle);
	void setAuthor(string bookAuthor);
	void setNumAvailable(unsigned int numBooks);
	static void setNumBooks(IdentNum num);

	void addBooks(int bookQuantity);
	void loanBook();
	void returnBook();
};

#endif


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

```
//============================================================================
// LIBRAY.H
//----------------------------------------------------------------------------

#ifndef LIBRARY_H
#define LIBRARY_H

#include <string>
#include <vector>

#include "defs.h"
#include "book.h"
#include "user.h"

using namespace std;

class Library {

private:
	vector<User> users; // all users that are registered or were registered in the library
	vector<Book> books; // all books that are registered or were registered in the library
	string filenameUsers; // name of the file where users are saved at the end of each program run
	string filenameBooks; // name of the file where books are saved at the end of each program run

public:
	// constructors
	Library();
	Library(string fileUsers, string fileBooks);

	// get functions
	User getUserByID(IdentNum userID) const;
	Book getBookByID(IdentNum bookID) const;

	// user management
	void addUser(User);

	// book management
	void addBook(Book);

	// loaning management
	void loanBook(IdentNum, IdentNum);
	void returnBook(IdentNum, IdentNum);

	// file management methods
	void loadUsers();
	void loadBooks();
	void saveUsers();
	void saveBooks();

	// information display
	void showUsers() const;
	void showUsers(string str) const;
	void showBooks() const;
	void showBooks(string str) const;
	void showAvailableBooks() const;
};

#endif

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

```
//============================================================================
// USER.CPP
//----------------------------------------------------------------------------
#include "user.h"

      // to do ...

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

//============================================================================
// BOOK.CPP
//----------------------------------------------------------------------------
#include "library.h"

      // to do ...

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

//============================================================================
// LIBRARY.CPP
//----------------------------------------------------------------------------
#include "library.h"

Library::Library(string fileUsers, string fileBooks)
{
      // to do ...
}

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


//============================================================================
// MAIN.CPP
//----------------------------------------------------------------------------
#include <iostream>

#include "library.h"

using namespace std;

int main ()
{
      Library library("users.txt","books.txt");

      do
      {
            //show menu
            cout << "#### Menu ####\n\n";
            cout << "1 - New user\n";
            cout << "2 - New book\n";
            // ...
            cout << "0 - Exit\n";
            // TO DO
            // read user option
            // execute user option

      } while (...);
}

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

==========

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{
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;
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();
};


//-----------------------------------------------------------------------
// 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;

}
```

```
/*
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;
}

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

```
/*
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 {
private:
      struct node{
            T data;
            node *next;
      } *p;
      size_t listSize;
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();
};

//-----------------------------------------------------------------------
// 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;
}
```

```
// 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(2,"F.C.Porto");   // 2nd in football rank
      Pair<int,int> p3(2016,366);           // Number of days of year

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

}
```