

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/ood_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**, or **attributes**) :
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 memberFuncId(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
};

// -----
// 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 to 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;`
 - Assigning a default value to all the parameters of a constructor is equivalent to define a default constructor:
`Fraction(int num=0, int denom=1); // fraction with value zero`
- ... **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 denominator);  
    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); // ALTERNATIVE: append _ to the name
    // of each attribute; ex: _numerator
}
//-----
// 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 <cstdlib>

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 <cstdlib>
#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;
};

```

```

//-----
// UTILITY FUNCTIONS
// Note: in C++11 there are functions for converting numbers <-> strings
// (see previous notes)
//-----

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, global integers are set to zero

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

Book::Book()
{
    // suggestion: do not increment numBooks in 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
    lib.saveBooks("bookfile.txt");
}

```

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**.

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

```

//-----
// SEPARATE COMPILE EXAMPLE - STEP BY STEP
//-----

//=====
// SOLUTION 1 (sc1.sln) - Date class definition and implementation
//=====

//=====
// FILE: Date.h
//=====
class Date
{
public:
    Date();
    Date(int year, int month, int day);
    void show() const;
private:
    int year, month, day;
};

//=====
//=====
// FILE: Date.cpp
//=====
#include <iostream>
#include "Date.h"

Date::Date()
{
    this->year = this->month = this->day = 0; // an invalid date !
}
//=====
Date::Date(int year, int month, int day)
{
    this->year = year;
    this->month = month;
    this->day = day;
}
//=====
void Date::show() const
{
    std::cout << this->year << '-' << this->month << '-' << this->day;
}
//=====

// TODO:
// Build > Compile
// see the Date.obj file in C:\.....\sc01\sc01\Debug

```

NOTE:

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

```

//=====
// SOLUTION 2 (sc2.sln) - add main() to sc1
//=====
//=====
// FILE: main.cpp
//=====
#include <iostream>
#include "Date.h"

using namespace std;

int main()
{
    Date d1;
    Date d2(2016, 4, 21);

    cout << "d1 = ";
    d1.show();
    cout << endl;

    cout << "d2 = ";
    d2.show();
    cout << endl << endl;
}

//=====
// SOLUTION 3 (sc3.sln) - add Person class definition and implementation to sc2
//                               modify main()
//=====
//=====
// FILE: Person.h
//=====
#include <string>

class Person
{
public:
    Person();
    Person(std::string name, int age);
    void show() const;
private:
    std::string name;
    int age;
};

//=====
//=====
// FILE: Person.cpp
//=====
#include <iostream>
#include "Person.h"

Person::Person()
{
    this->name = "NO_NAME";
    this->age = 0;
}

```

```

//=====
Person::Person(std::string name, int age)
{
    this->name = name;
    this->age = age;
}

//=====
void Person::show() const
{
    std::cout << this->name << " - " << this->age;
}
//=====

//=====
// FILE:main.cpp
//=====
#include <iostream>
#include "Date.h"
#include "Person.h"

using namespace std;

int main()
{
    Date d1;
    Date d2(2016, 4, 21);

    cout << "d1 = ";
    d1.show();
    cout << endl;

    cout << "d2 = ";
    d2.show();
    cout << endl << endl;

    //-----

    Person p1;
    Person p2("Rui Silva",18);

    cout << "p1 = ";
    p1.show();
    cout << endl;

    cout << "p2 = ";
    p2.show();
    cout << endl << endl;
}

```

```

//=====
// SOLUTION 4 (sc4.sln) - add Date attribute to Person class
//=====
//=====
// FILE: Person.h (new version)
//=====
#include <string>
#include "Date.h"    // NOTE THIS

class Person
{
public:
    Person();
    Person(std::string name, Date birthDate);    // NOTE THIS
    void show() const;
private:
    std::string name;
    // int age;        // NOTE THIS
    Date birthDate;    // NOTE THIS
};

//=====
//=====
// FILE: Person.cpp (new version)
//=====
#include <iostream>
#include "Person.h"

Person::Person()
{
    this->name = "NO_NAME";
    //this->age = 0;
    this->birthDate = Date();    // NOTE THIS
}

//=====
Person::Person(std::string name, Date birthDate)    // NOTE THIS
{
    this->name = name;
    //this->age = age;        // NOTE THIS
    this->birthDate = birthDate;    // NOTE THIS
}

//=====
void Person::show() const
{
    std::cout << this->name << " - ";
    (this->birthDate).show();
}

//=====
//=====
// FILE:main.cpp (new version)
//=====
#include <iostream>
#include "Date.h"
#include "Person.h"

using namespace std;

int main()
{
    Date d1;
    Date d2(2016, 4, 21);

```

```

cout << "d1 = ";
d1.show();
cout << endl;

cout << "d2 = ";
d2.show();
cout << endl << endl;

//-----

Person p1;
Person p2("Rui Silva",d2);

cout << "p1 = ";
p1.show();
cout << endl;

cout << "p2 = ";
p2.show();
cout << endl << endl;
}

```

NOTE THE COMPILATION ERROR: 'Date' : 'class' type redefinition

```

//=====
// SOLUTION 5 (sc5.sln) - solving the 'Date' : 'class' type redefinition problem
//=====
//-----
// FILE: Date.h (new version)
//-----
#ifndef DATE_H // NOTE THIS ALTERNATIVE: #pragma once
#define DATE_H // NOTE THIS

class Date
{
public:
    Date();
    Date(int year, int month, int day);
    void show() const;
private:
    int year, month, day;
};

#endif // NOTE THIS

//=====
// FILE: Person.h (new version)
//=====
#ifndef PERSON_H // NOTE THIS
#define PERSON_H // NOTE THIS

#include <string>
#include "Date.h"

class Person
{
public:
    Person();
    Person(std::string name, Date birthDate);
    void show() const;

```

```

private:
    std::string name;
    // int age;
    Date birthDate;
};

#endif // NOTE THIS

//=====
// SOLUTION 6 (sc6.sln) - adding an auxiliar function
//=====
//=====
// FILE: Auxiliar.h
//=====
#ifndef AUXILAR_H
#define AUXILAR_H

#include <string>

void readString(std::string message, std::string & name);

#endif

//=====
// FILE: Auxiliar.cpp
//=====
#include <iostream>
#include <string>
#include "Auxiliar.h"

void readString(std::string message, std::string & s)
{
    do
    {
        std::cout << message;
        std::getline(std::cin, s);
    } while (s == "");
}

//=====
// FILE: main.cpp (new version)
//=====
#include <iostream>
#include "Date.h"
#include "Person.h"
#include "Auxiliar.h" // NOTE THIS

using namespace std;

int main()
{
    Date d1;
    Date d2(1998, 4, 21);

    cout << "d1 = ";
    d1.show();
    cout << endl;

    cout << "d2 = ";
    d2.show();
    cout << endl << endl;
}

```

```

//-----

Person p1;
Person p2("Rui Silva",d2);

cout << "p1 = ";
p1.show();
cout << endl;

cout << "p2 = ";
p2.show();
cout << endl << endl;

//-----
// NOTE THIS
std::string name;
readString("What is the newborn name ? ", name);
//Date d3 = Date(2016, 11, 23);
Person p3(name, Date(2016,11,23));
cout << "p3= ";
p3.show();
cout << endl << endl;
}

```


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

//=====

```

```

//=====
// LIBRARY.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 (...);
}

//=====

```

Separate compilation – yet another example

Declaring and defining global variables in multiple source file programs

- First of all, it is important to understand the **difference** between defining a variable and declaring a variable:
 - A variable is **defined** when the compiler allocates the storage for the variable.
 - A variable is **declared** when the compiler is informed that a variable exists (and which is its type);
it does not allocate the storage for the variable at that point.
- You may declare a variable multiple times (though once is sufficient); you may define it only once within a given scope.

Using the extern keyword

- Using extern is useful when the program you're building consists of multiple source files linked together, where some of the variables defined, for example, in source file **file1.c** need to be referenced in other source files, such as **file2.c**.
- The best way to declare and define global variables is
 - to use a header file **file3.h** to contain an **extern** declaration of the variable.
 - The header is included by the one source file that defines the variable (ex: **file3.cpp**) and by all the source files that reference the variable.
 - For each program,
one source file (and only one source file) defines the variable.
Similarly, **one header file (and only one header file) should declare the variable.**

```
//=====
// aux1.h

#ifndef AUX1_H
#define AUX1_H

int globalVar = 1000;

void f1();

#endif

//=====
// aux1.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

void f1()
{
    cout << "globalVar = " << globalVar << endl;
}

//=====
// main.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

int main()
{
    cout << "Global Var = " << globalVar << endl;
    f1();
}

//=====
```

Error List					
2 Errors 0 Warnings 0 Messages					
	Description	File	Line	Column	Project
1	error LNK2005: "int globalVar" (?globalVar@@3HA) already defined in main.obj	aux1.obj			Extern_01
2	error LNK1169: one or more multiply defined symbols found	Extern_01.exe	1	1	Extern_01

```

//=====
// aux1.h

#ifndef AUX1_H
#define AUX1_H

extern int globalVar;

void f1();

#endif

//=====
// aux1.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

int globalVar = 1000;

void f1()
{
    cout << "globalVar = " << globalVar << endl;
}

//=====
// main.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

int main()
{
    cout << "Global Var = " << globalVar << endl;
    f1();
}

//=====

```