==========

FRIEND FUNCTIONS

==========

-
- Class operations are typically implemented as member functions
- Some operations are better implemented as ordinary (nonmember) functions

```
//Program to demonstrate the use of function equal() that compares 2 dates

#include <iostream>
using namespace std;

class Date
{
public:
      Date( ); //Initializes the date to January 1st.
      Date(int y, int m, int d);
      void input( );
      void output( ) const;
      int get_year( ) const;
      int get_month( ) const;
      int get_day( ) const;
private:
      int year;
      int month;
      int day;
};

//-----------------------------------------------------------------------
Date::Date()
{
      year = 1970;
      month = 1;
      day = 1;
}

//-----------------------------------------------------------------------
Date::Date(int y, int m, int d)
{
      year = y;
      month = m;
      day = d;
}

//-----------------------------------------------------------------------
int Date::get_year( ) const
{
      return year;
}

//-----------------------------------------------------------------------
int Date::get_month( ) const
{
      return month;
}
```

```
//-------------------------------------------------------------------------
int Date::get_day( ) const
{
      return day;
}

//-------------------------------------------------------------------------
void Date::input( )
{
      cout << "year (number) ? ";
      cin >> year;
      cout << "month (number) ? ";
      cin >> month;
      cout << "day (number) ? ";
      cin >> day;
}

//-------------------------------------------------------------------------
void Date::output( )
{
      cout << year << "/" << month << "/" << day << endl;
}

//-------------------------------------------------------------------------
bool equal(const &Date date1, const Date &date2)
{
      return (
            date1.get_year( ) == date2.get_year( ) &&
            date1.get_month( ) == date2.get_month( ) &&
            date1.get_day( ) == date2.get_day( )
            );
}

//-------------------------------------------------------------------------
int main( )
{
      Date date1(2014,5,14), date2;

      cout << "Date1 is ";
      date1.output( );

      cout << endl;
      cout << "Enter Date2:\n";
      date2.input( );
      cout << "Date2 is ";
      date2.output( );

      if (equal(date1, date2))
            cout << "date1 is equal to date2\n";
      else
            cout << "date1 is NOT equal to date2\n";
      return 0;
}
```

```
// Program to demonstrate the use of function equal() that compares 2 dates
// A more efficient version that declares
// equal() as a friend function of class Date


#include <iostream>
using namespace std;

class Date
{
      friend bool equal(const Date &date1, const Date &date2);
public:
      Date(int y, int m, int d);
      Date( ); //Initializes the date to January 1st.
      void input( );
      void output( ) const;
      int get_year( ) const;
      int get_month( ) const;
      int get_day( ) const;
private:
      int year;
      int month;
      int day;
};

//-----------------------------------------------------------------------
Date::Date()
{
      year = 1970;
      month = 1;
      day = 1;
}

//-----------------------------------------------------------------------
Date::Date(int y, int m, int d)
{
      year = y;
      month = m;
      day = d;
}

//-----------------------------------------------------------------------
int Date::get_year( ) const
{
      return year;
}

//-----------------------------------------------------------------------
int Date::get_month( ) const
{
      return month;
}

//-----------------------------------------------------------------------
int Date::get_day( ) const
{
      return day;
}
```

```
//-------------------------------------------------------------------------
void Date::input( )
{
      cout << "year (number) ? ";
      cin >> year;
      cout << "month (number) ? ";
      cin >> month;
      cout << "day (number) ? ";
      cin >> day;
}

//-------------------------------------------------------------------------
void Date::output( ) const
{
      cout << year << "/" << month << "/" << day << endl;
}

//-------------------------------------------------------------------------
bool equal(const Date &date1, const Date & date2)
// NOTE:
// 1) DOES NOT include friend nor Date::
// 2) can access the private members (data and functions) of Date class
{
      return (
            date1.year == date2.year &&
            date1.month == date2.month &&
            date1.day == date2.day
            );
}

//-------------------------------------------------------------------------
int main( )
{
      Date date1(2014,5,14), date2;

      cout << "Date1 is ";
      date1.output( );

      cout << endl;
      cout << "Enter Date2:\n";
      date2.input( );
      cout << "Date2 is ";
      date2.output( );

      if (equal(date1, date2))
            cout << "date1 is equal to date2\n";
      else
            cout << "date1 is NOT equal to date2\n";
      return 0;
}
```

FRIEND FUNCTIONS

- Friend functions are not members of a class, but
  can access private member variables of the class
- A friend function is declared using the keyword **friend** in the class definition
- A friend function is not a member function
- A friend function is an ordinary function

FRIEND FUNCTION DECLARATION, DEFINITION & CALLING

- A friend function is declared as a **friend** in the class definition
- A friend function is defined as a nonmember function
  without using the "::" operator
- A friend function is called without using the '.' operator

ARE FRIEND FUNCTIONS NECESSARY ?

- Friend functions can be written as non-friend functions
  using the normal accessor and mutator functions that should be part of the class
- The code of a friend function is **simpler** and it is **more efficient**

WHEN TO USE FRIEND FUNCTIONS ?

- How do you know when a function should be a friend or a member function?
- In general, use a member function if
  the task performed by the function involves only one object
- In general, **use a nonmember function** if
  the task performed by the function **involves more than one object**
- Choosing to make the nonmember function a friend is a decision of
  efficiency and personal taste

FRIEND CLASSES

- Classes may also be declared friend of other classes.
- Declaring a class as a friend means that
  the friend class and all of its member functions
  have access to the private members of the other class
- class **Node**
  {
  ...
  friend class **List**;
  }

- General outline of how you set things:

```
class F; //forward declaration

class C
{
   public:
      ...
   friend class F;
      ...
};

class F
{
   ...
}
```

- **NOTE**: friend classes may be "dangerous", because the designer of a class usually knows what friends are going to do, but cannot predict what a derived class might do (*derived classes will be studied later*).

==========

OPERATOR OVERLOADING

==========

```
/*
      OPERATOR OVERLOADING example / MAIN.CPP
      An example with fractions
*/

#include "fraction.h"

int main()
{
      // Testing constructors
      Fraction a;  // Value is 0/1
      Fraction b(4);  // Value is 4/1
      Fraction c(6,8);  // Value is 6/8, which is converted to 3/4

      // Overloading output operator
      cout << "overloading output operator\n";
      cout << " a = " << a << endl;
      cout << " b = " << b << endl;
      cout << " c = " << c << endl;
      cout << endl;

      // Using default assignment operator and the default copy constructor
      cout << "using default copy constructor & assignment operator\n";

      Fraction d(c); // d is copy of c
      cout << " Fraction d(c):  d = " << d << endl;
      // the copy constructor for class Fraction is invoked
      // unless the programmer provides one,
      // the compiler will automatically generate a copy constructor

      Fraction e;
      e = c; // the assignment operator is automatically generated
      cout << " e = c:  e = " << e << endl;
      cout << endl;

      // Testing Overloaded arithmetic operators
      cout << "testing arithmetic operators\n";

      e = b + c;
      cout << " e = b + c = " << e << endl;

      Fraction f;
      f = b - c;
      cout << " f = b - c = " << f << endl;

      Fraction g = (b + (-c)); //unary arithmetic operator (minus)
      cout << " g = (b + (-c)) = " << g << endl;
      cout << endl;

      // Testing Overloaded comparison operators
      cout << "testing comparison operators\n";
      if (f == g)
            cout << " f == g; comparison test successful\n";
      else
            cout << " comparison test failed\n";
```

```
        a = Fraction(6,8);   //note 'a' already defined above
        b = Fraction(16,8);  //note 'b' already defined above
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
        if (a < b)
              cout << " a < b ; comparison test successful\n";
        else
              cout << " a < b ; comparison test failed\n";

        // comparing a fraction and an integer
        //  NOTE: the Big C++ book is wrong when saying that one could write:   if (b == 2)
        if (b == Fraction(2))
              cout << " b == Fraction(2) ; comparison test successful\n";
        else
              cout << " b == Fraction(2) ; comparison test failed\n";
        cout << endl;


        // Testing Overloaded input (and output)
        cout << "overloading input (and output) operator\n";
        cout << " fraction c ? ";
        cin >> c;
        cout << " c = " << c << endl;

        cout << " fraction d ? ";
        cin >> d;
        cout << " d = " << d << endl;
        cout << endl;

        // Testing Overloaded increment operators
        cout << "testing increment operators\n";
        e = c++;
        cout << " c = " << c << "; e = c++ = " << e << endl;

        f = ++d;
        cout << " d = " << d << "; f = ++d = " << f << endl;
        cout << endl;

        // Testing Overloaded 'conversion to double' operator
        cout << "testing 'conversion to double' operator\n";
        cout << "double(a) = " << double(a) << endl;


        return 0;
}
```

```
/*
FRACTION.H
OPERATOR OVERLOADING examples
Adapted from Big C++ book
*/
#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>

using namespace std;

class Fraction
{
public:
      Fraction(); // construct fraction 0/1
      Fraction(int t); // construct fraction t/1
      Fraction(int t, int b); // construct fraction t/b
      int numerator() const; //return numerator value
      int denominator() const; //return denominator value
      void display() const; // displays fraction

      // Updates a fraction by adding in another fraction,'right'
      // returns the update fraction
      Fraction& operator+=(const Fraction& right);

      // Increment fraction by 1.
      Fraction& operator++(); // Prefix form      :  ++(++frac) is allowed!
      Fraction operator++(int unused); // Postfix form : but not (frac++)++
      // These operators, in addition to producing a result,
      // alter their argument value; for this reason they are
      // defined as member functions, not as ordinary functions.

      // Converts a fraction into a floating-point value.
      // returns the converted value
      operator double() const; // NOTE: do not specify a return type
                               // return type is implicit in the name

      // Compare one fraction value to another.
      // Result is negative if less than right,
      // zero if equal, and positive if greater than 'right'.
      int compare(const Fraction& right) const;

private:
      // Place the fraction in least common denominator form.
      void normalize();

      // Compute the greatest common denominator of two integers.
      int gcd(int n, int m);

      int top; // fraction numerator
      int bottom; //fraction denominator
};

// Other operators defined as ordinary functions
// ... but they can also be defined as member functions (see later)
Fraction operator+(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& left, const Fraction& right);
Fraction operator*(const Fraction& left, const Fraction& right);
Fraction operator/(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& value); // unary minus
```

```
bool operator==(const Fraction& left, const Fraction& right);
//bool operator==(const Fraction& left, int intValue);
bool operator!=(const Fraction& left, const Fraction& right);
bool operator<(const Fraction& left, const Fraction& right);
bool operator<=(const Fraction& left, const Fraction& right);
bool operator>(const Fraction& left, const Fraction& right);
bool operator>=(const Fraction& left, const Fraction& right);

// These two operators CAN'T BE defined as member functions. WHY?
// Compare the first parameters of the above functions and those of the following ones
ostream& operator<<(ostream& out, const Fraction& value);
istream& operator>>(istream& in, Fraction& r);

#endif
```

QUESTION:
Why is compare() a public method of class Fraction?

ANSWER: because it is used in the operator overloading functions that are not members os class Fraction

```
/*
FRACTION.CPP
OPERATOR OVERLOADING examples
Adapted from Big C++ book
*/
#include "fraction.h"
#include <string>
#include <sstream>
#include <cassert>
//#include <stdexcept>

//-----------------------------------------------------------------------
// example of constructor with Field Initializer List
Fraction::Fraction() : top(0), bottom(1) { }

//-----------------------------------------------------------------------
Fraction::Fraction(int t) : top(t), bottom(1) { }

//-----------------------------------------------------------------------
Fraction::Fraction(int t, int b) : top(t), bottom(b)
{
      normalize();
}

//-----------------------------------------------------------------------
// When function bodies are very short, the function may be declared 'inline'
// Alternatively the body of the function
// may be inserted directly into the class declaration (without 'inline')
// Although usually running more efficiently, they consume more storage
// NOTE: THE COMPILER MAY IGNORE THE "inline" HINT ...
inline int Fraction::numerator() const
{
      return top;
}

//-----------------------------------------------------------------------
inline int Fraction::denominator() const
{
      return bottom;
}

//-----------------------------------------------------------------------
inline void Fraction::display() const
{
      cout << top << "/" << bottom;
}

//-----------------------------------------------------------------------
void Fraction::normalize()
{
      // Normalize fraction by
      // (a) moving sign to numerator
      // (b) ensuring numerator and denominator have no common divisors

     int sign = 1;

      if (top < 0)
      {
            sign = -1;
            top = - top;
      }
```

```
      if (bottom < 0)
      {
            sign = - sign;
            bottom = - bottom;
      }

      assert(bottom != 0);

      int d = 1;
      if (top > 0) d = gcd(top, bottom);
      top = sign * (top / d);
      bottom = bottom / d;
}

//-----------------------------------------------------------------------
int Fraction::gcd(int n, int m)
{
      // Euclid's Greatest Common Divisor algorithm

      assert((n > 0) && (m > 0));

      while (n != m)
      {
            if (n < m)
                  m = m - n;
            else
                  n = n - m;
      }
      return n;
}

//-----------------------------------------------------------------------
Fraction operator+(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator() +
            right.numerator() * left.denominator(),
            left.denominator() * right.denominator());
      return result;
}
// ALTERNATIVE: no local variable is created;
// the result is constructed as an unnamed temporary
/*
Fraction operator+(const Fraction& left, const Fraction& right)
{
return Fraction ( left.numerator() * right.denominator() +
                  right.numerator() * left.denominator(),
                  left.denominator() * right.denominator());
}
*/

//-----------------------------------------------------------------------
Fraction operator-(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator() -
            right.numerator() * left.denominator(),
            left.denominator() * right.denominator());
      return result;
}
```

```
//-----------------------------------------------------------------------
Fraction operator*(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.numerator(),
            left.denominator() * right.denominator());
      return result;
}

//-----------------------------------------------------------------------
Fraction operator/(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator(),
            left.denominator() * right.numerator());
      return result;
}

//-----------------------------------------------------------------------
Fraction operator-(const Fraction& value) // Unary minus
{
      Fraction result(-value.numerator(), value.denominator());
      return result;
}

//-----------------------------------------------------------------------
// NOTE: the comparison operators, below, are written using 'compare'
int Fraction::compare(const Fraction& right) const
{
      return
            numerator() * right.denominator() -
            denominator() * right.numerator();
      // Return the numerator of the difference
}

//-----------------------------------------------------------------------
bool operator==(const Fraction& left, const Fraction& right)
{
      return left.compare(right) == 0;
}

/*
// To allow comparison of a Fraction and an integer; see comment in main()
bool operator==(const Fraction& left, int intValue)
{
return ((static_cast<double> (left.numerator()) / left.denominator()) ==
                              (static_cast<double> (intValue)));
}
*/

//-----------------------------------------------------------------------
bool operator!=(const Fraction& left, const Fraction& right)
{
      return left.compare(right) != 0;
}
```

```
//-----------------------------------------------------------------------
bool operator<(const Fraction& left, const Fraction& right)
{
     return left.compare(right) < 0;
}

//-----------------------------------------------------------------------
bool operator<=(const Fraction& left, const Fraction& right)
{
     return left.compare(right) <= 0;
}

//-----------------------------------------------------------------------
bool operator>(const Fraction& left, const Fraction& right)
{
     return left.compare(right) > 0;
}

//-----------------------------------------------------------------------
bool operator>=(const Fraction& left, const Fraction& right)
{
     return left.compare(right) >= 0;
}

//-----------------------------------------------------------------------
// NOTE:
// The operators << and >> return the stream value as the result
// This allows "complex" stream expressions like "cout << frac1 << endl;
// (see examples in main() )
//-----------------------------------------------------------------------
ostream& operator<<(ostream& out, const Fraction& value)
{
     out << value.numerator() << "/" << value.denominator();
     return out;    //NOTE THE RETURN VALUE. Why is this done ?
}
// This function could have been declared 'friend' of class Fraction
// Would it have any advantage ?
// class Fraction {
//     friend ostream& operator<<(ostream& out, const Fraction& value);

/*
ostream& operator<<(ostream& out, const Fraction& value)
{
  out << value.top << "/" << value.bottom;
  return out;
}
*/

//-----------------------------------------------------------------------
istream& operator>>(istream& in, Fraction& r) // NOTE: 'r' is non-const
{
     string fractionString;
     char fracSymbol;
     int num;
     int denom;

     getline(in,fractionString); // must be in format 'numerator/denominator'

     istringstream fractionStrStream(fractionString);
     fractionStrStream >> num >> fracSymbol >> denom;
```

```
        assert(fracSymbol == '/'); // input must be inserted correctly !!!
        assert(denom != 0);        // otherwise KABOOM !!!   ...\|/...

        r = Fraction(num, denom);
        return in;
}

//-----------------------------------------------------------------------
//NOTE: do not specify a return type; it is implicit in the name
Fraction::operator double() const
{
        // Convert numerator to double, then divide
        return static_cast<double>(top) / bottom;
}

//-----------------------------------------------------------------------
Fraction& Fraction::operator++() // Prefix form
{
        top += bottom;
        normalize();
        return *this;
        //NOTE: returns the fraction after modification
        //      as a reference to the current fraction
        //      This enables a preincremented Fraction object
        //      to be used as an 'lvalue';
        // ex:   ++++fraction;  // equivalent to ++(++fraction);
        //       OR
        //       ++fraction *= 2; !!! ⇔ fraction = 2 * (fraction + 1)
        //       to be consistent with C++ syntax
        //
        // SEE EXAMPLE OF FUNCTIONS THAT RETURN REFERENCES IN THE NEXT PAGES
}

//-----------------------------------------------------------------------
// NOTE: the additional dummy parameter
Fraction Fraction::operator++(int unused) // Postfix form
{
        Fraction clone(top, bottom);
        top += bottom;
        normalize();
        return clone; //NOTE: returns the fraction before modification
}

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

//NOTE: the assignment operator will be automatically generated
//      but +=, -=, *= and /= will not


Fraction& Fraction::operator+=(const Fraction& right)
{
        top = top * right.denominator() + bottom * right.numerator();
        bottom *= right.denominator();
        normalize();
        return *this;
}
```

Some binary operators (ex: operator+) could have been declared inside class Fraction

Instead of (declaration outside class Fraction) ...

```
//-------------------------------------------------------------------------
// PREVIOUS IMPLEMENTATION (outside class Fraction)

class Fraction
{
public:
...

private:
...

      int top; // fraction numerator
      int bottom; //fraction denominator
};

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

Fraction operator+(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& value); // unary minus
...

//-------------------------------------------------------------------------
Fraction operator+(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator() +
            right.numerator() * left.denominator(),
            left.denominator() * right.denominator());
      return result;
}

Fraction operator-(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator() -
            right.numerator() * left.denominator(),
            left.denominator() * right.denominator());
      return result;
}

...

Fraction operator-(const Fraction& value) // Unary minus
{
      Fraction result(-value.numerator(), value.denominator());
      return result;
}


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

... one could have (declaration **inside** class Fraction)

```
class Fraction
{
public:
...
      Fraction operator+(const Fraction& right);
      Fraction operator-(const Fraction& right);
      Fraction operator-(); // unary minus
...

private:
...
      int top; // fraction numerator
      int bottom; //fraction denominator
};
...

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

Fraction Fraction::operator+(const Fraction& right)
{
      Fraction result(
            top * right.denominator() +
            right.numerator() * bottom,
            bottom * right.denominator());
      return result;
}

Fraction Fraction::operator-(const Fraction& right)
{
      Fraction result(
            top * right.denominator() -
            right.numerator() * bottom,
            bottom * right.denominator());
      return result;
}

...

Fraction Fraction::operator-() // Unary minus
{
      Fraction result(-top, bottom);
      return result;
}
```

NOTE:

```
Fraction f1, f2, f3;
...

f3 = add(f1,f2);
```

where

```
Fraction add(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator() +
            right.numerator() * left.denominator(),
            left.denominator() * right.denominator());
      return result;
}
```

is equivalent to:

```
f3 = f1 + f2;
```

where

```
Fraction operator+(const Fraction& left, const Fraction& right)
{
      Fraction result(
            left.numerator() * right.denominator() +
            right.numerator() * left.denominator(),
            left.denominator() * right.denominator());
      return result;
}
```

It is only a question of syntax ...

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

It is easier to read

```
f3 = f1 + f2*f2;
```

than

```
f3 = add(f1,multiply(f2,f2));
```

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

NOTES:

- `f3 = f1 + f2;`

will be interpreted by the **compiler** as (... one could have written the code like this !)

`f3 = operator+(f1,f2);` if operator+ is **not** a member function of class Fraction

or as

`f3 = f1.operator+(f2);` if operator+ is a member function of class Fraction

- **Overloaded (), [], -> and assigment operators must be declared as class members.**

THE "THIS" POINTER

- When defining member functions for a class,
  you sometimes want to refer to the calling object.
- The *this* pointer is a predefined pointer that points to the calling object
- Example:

```
class Fraction
{
public:
...
      Fraction operator+(const Fraction& right);
      Fraction operator-(const Fraction& right);
      Fraction operator-(); // unary minus
...

private:
...

      int top; // fraction numerator
      int bottom; //fraction denominator
};
...
//-------------------------------------------------------------------------

Fraction Fraction::operator+(const Fraction& right)
{
      Fraction result(
            top * right.denominator() +
            right.numerator() * bottom,
            bottom * right.denominator());
      return result;
}
```

Could be written:

```
Fraction Fraction::operator+(const Fraction& right)
{
      Fraction result(
            (*this).numerator() * right.denominator() +
            right.numerator() * (*this).denominator(),
            (*this).denominator() * right.denominator());
      return result;
}
```

or as ...

```
Fraction Fraction::operator+(const Fraction& right)
{
      Fraction result(
            this->numerator() * right.denominator() +
            right.numerator() * this->denominator(),
            this->denominator() * right.denominator());
      return result;
}
```

- Another use:
  when a parameter of a function member
  has the same name as an atribute of the class
- (can be easily avoided)

```
class Fraction
{
public:
      Fraction(); // construct fraction 0/1
      Fraction(int t); // construct fraction t/1
      Fraction(int top, int bottom); // construct fraction t/b
...

private:
...

      int top; // fraction numerator
      int bottom; //fraction denominator
};
```

```
//---------------------------------------------------------------------
Fraction::Fraction(int top, int bottom)
{
      this->top = top;
      this->bottom = bottom;
      normalize();
}
```

The following code avoids the use of this->top and this->bottom .
It is sintactically correct but ...

```
//---------------------------------------------------------------------
Fraction::Fraction(int top, int bottom) : top(top), bottom(bottom)
{
      normalize();
}
```

- Yet another use:
  as we saw, it was not necessary to overload the assignment operator, **operator=**, for class **Fraction**
- But, when **operator=** is overloaded, it must return the ***this** object

```
class Fraction
{
public:
      Fraction(); // construct fraction 0/1
      Fraction(int t); // construct fraction t/1
      Fraction(int t, int b); // construct fraction t/b
...
     // operator= has to be a member of the class
     // it can't be a friend of the class
      Fraction& operator=(const Fraction& right);

...

private:
...

      int top; // fraction numerator
      int bottom; //fraction denominator
};

Fraction & Fraction::operator=(const Fraction &right)
{
      top = right.numerator();
      bottom = right.denominator();
      return *this;
}
```

The primary use of **this** pointer is

- to return the current object,
- or to pass the object to a function.

Returning the left hand object is necessary if one wants to do **multiple assignment operations** (returned as a reference for better efficiency)

```
      f1 = f2 = f3;
```

// RETURNING POINTERS AND REFERENCES - TWO SIMPLE EXAMPLES

```
// WHAT DOES THIS PROGRAM DO ?

#include <iostream>
#include <cstddef>

using namespace std;

int * f(int vec[], size_t vec_size, int value)
{
      for (size_t i =0; i<vec_size; i++)
            if (vec[i]==value)
                  return &vec[i];
      return NULL;
}

int main()
{
      int a[3] = {1,2,3};

      int * px = f(a,sizeof(a)/sizeof(int), 2);
      // int * px = f(a,sizeof(a)/sizeof(int),5);   // TRY THIS

      if (px != NULL)
      {
            cout << *px << endl << endl;
            *px = 10;
      }

      for (int i=0; i<3; i++)
            cout << a[i] << endl;

}
//=========================================================================

// AND THIS ONE ?
// (see implementation of subscript operator in next example: String class implementation)

#include <iostream>
#include <cstddef>

using namespace std;

int & f(int a[], size_t i)
{
      return a[i];  // NOTE: null references are prohibited;
                    // compare w/previous example
                    // were NULL pointer is returned in some cases
}

int main()
{
  int a[3] = {1,2,3};

  int & x = f(a,1);

  cout << "x = " << x << endl << endl;

  x = 10;           // NOTE: equivalent to f(a,1) = 10;
  // f(a,1) = 10;  // a function used on left side of an assigment…?!!!

  for (int i=0; i<3; i++)
    cout << "a[" << i << "] = " << a[i] << endl;
}
```

- Yet another example:

```
#include <iostream>

using namespace std;

class Date {
public:
  Date();
  Date & setDay(int d);
  Date & setMonth(int m);
  Date & setYear(int y);
  void show() const;
private:
  int day, month, year;
};

Date::Date()
{
  day = month = year = 1;

}

// updates 'day' and returns a reference to 'day' ...
Date & Date::setDay(int d)
{
  day = d;
  return *this;
}

Date & Date::setMonth(int m)
{
  month = m;
  return *this;
}

Date & Date::setYear(int y)
{
  year = y;
  return *this;
}
void Date::show() const
{
  cout << day << "/" << month << "/" << year << endl;
}

void main()
{
  Date d;

  // ... thus enabling the use of cascaded 'set_operations':
  d.setDay(10).setMonth(5).setYear(2016);
  d.show();
}
```

TO DO:
- REPLACE `Date &` BY `Date` AND INTERPRET RESULT
- THEN, TRY `d.setDay(10).setMonth(5).setYear(2016).show();`

```
// MORE EXAMPLES OF FUNCTIONS THAT RETURN REFERENCES OR POINTERS TO OBJECTS

#include <iostream>

using namespace std;

class Position
{
public:
  Position(int x, int y);
  int getX() const;
  int getY() const;
  void setX(int x);
  void setY(int y);
private:
  int x, y;
};

Position::Position(int x, int y)
{
  this->x = x;
  this->y = y;
}

int Position::getX() const
{
  return x;
}

int Position::getY() const
{
  return y;
}

void Position::setX(int x)
{
  this->x = x;
}

void Position::setY(int y)
{
  this->y = y;
}

ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}

//------------------------------------------------------------------------
int main()
{
  Position p1(1,1);
  cout <<"p1 = " << p1 << endl;
  p1.setX(10);
  p1.setY(20);
  cout << "p1 = " << p1 << endl;
}
```

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

#include <iostream>

using namespace std;

class Position
{
public:
  Position(int x, int y);
  int getX() const;
  int getY() const;
  Position get() const; // IN FACT, NOT NEEDED; TO GET A COPY, JUST DO p2=p1 ...
  void setX(int x);
  void setY(int y);
private:
  int x, y;
};

Position::Position(int x, int y)
{
  this->x = x;
  this->y = y;
}

int Position::getX() const
{
  return x;
}

int Position::getY() const
{
  return y;
}

Position Position::get() const
{
  return *this;
}

void Position::setX(int x)
{
  this->x = x;
}

void Position::setY(int y)
{
  this->y = y;
}

ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}
```

```
//------------------------------------------------------------------------
int main()
{
  Position p1(1, 1);
  cout << "p1 = " << p1 << endl;
  p1.setX(10);
  p1.setY(20);

  Position p2 = p1.get(); // ⇔ Position p2 = p1;
  cout << "p2 = " << p2 << endl;
  p2.setX(30);
  p2.setY(40);

  cout << endl;
  cout << "p1 = " << p1 << endl;
  cout << "p2 = " << p2 << endl;
}
```

```cpp
//========================================================================
#include <iostream>
using namespace std;

class Position
{
public:
  Position(int x, int y);
  int getX() const;
  int getY() const;
  Position get() const;
  void setX(int x);
  void setY(int y);
private:
  int x, y;
};

Position::Position(int x, int y)
{
  this->x = x;
  this->y = y;
}
int Position::getX() const
{
  return x;
}
int Position::getY() const
{
  return y;
}
Position Position::get() const
{
  return *this;
}
void Position::setX(int x)
{
  this->x = x;
}
void Position::setY(int y)
{
  this->y = y;
}
ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}

//------------------------------------------------------------------------
int main()
{
  Position p1(1, 1);
  cout << "p1 = " << p1 << endl;

  p1.get().setX(100);
  p1.get().setY(200);

  cout << endl;
  cout << "p1 = " << p1 << endl;
}
```

```
//========================================================================
#include <iostream>

using namespace std;

class Position
{
public:
  Position(int x, int y);
  int getX() const;
  int getY() const;
  Position& get();  // NOTE: not const
  void setX(int x);
  void setY(int y);
private:
  int x, y;
};

Position::Position(int x, int y)
{
  this->x = x;
  this->y = y;
}
int Position::getX() const
{
  return x;
}
int Position::getY() const
{
  return y;
}
Position& Position::get()
{
  return *this;
}
void Position::setX(int x)
{
  this->x = x;
}
void Position::setY(int y)
{
  this->y = y;
}
ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}

//------------------------------------------------------------------------
int main()
{
  Position p1(1, 1);
  cout << "p1 = " << p1 << endl;

  p1.get().setX(100);
  p1.get().setY(200);

  cout << endl;
  cout << "p1 = " << p1 << endl;
}
```

```cpp
//========================================================================
#include <iostream>

using namespace std;

class Position
{
public:
  Position(int x, int y);
  int getX() const;
  int getY() const;
  Position& setX(int x);
  Position& setY(int y);
private:
  int x, y;
};

Position::Position(int x, int y)
{
  this->x = x;
  this->y = y;
}

int Position::getX() const
{
  return x;
}

int Position::getY() const
{
  return y;
}

Position Position::setX(int x)
{
  this->x = x;
  return *this;
}

Position& Position::setY(int y)
{
  this->y = y;
  return *this;
}

ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}

//------------------------------------------------------------------------
int main()
{
  Position p1(1, 1);
  cout << "p1 = " << p1 << endl;
  p1.setX(30).setY(40);
  cout << "p1 = " << p1 << endl;
}
```

```
//========================================================================
#include <iostream>

using namespace std;

class Position
{
public:
  Position(int x, int y);
  int getX() const;
  int getY() const;
  Position* setX(int x);
  Position* setY(int y);
private:
  int x, y;
};

Position::Position(int x, int y)
{
  this->x = x;
  this->y = y;
}

int Position::getX() const
{
  return x;
}

int Position::getY() const
{
  return y;
}

Position* Position::setX(int x)
{
  this->x = x;
  return this;
}

Position* Position::setY(int y)
{
  this->y = y;
  return this;
}

ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}

//------------------------------------------------------------------------
int main()
{
  Position p1(1, 1);
  cout << "p1 = " << p1 << endl;
  Position *p1Ptr = &p1;
  p1Ptr->setX(30)->setY(40);
  //(*(*p1Ptr).setX(30)).setY(40);
  cout << "p1 = " << p1 << endl;
}
```

CONTAINERS & OPERATOR OVERLOADING

```
// CONTAINERS: a set of random 'int's

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <set>


using namespace std;

//------------------------------------------------------------------------
int main()
{
      set<int> s;

      srand((unsigned) time(NULL));

      for (int i=1; i<=10; i++)
            s.insert(rand()%10);

      for (set<int>::const_iterator i=s.begin(); i!=s.end(); i++)
            cout << *i << endl;

      // NOTES:
      // 1- the number os elements in the set may be less than 10
      // 2- the elements of the set are ordered
      //
      // For that the operator < must be defined.
      // It is already defined for 'int'

}
```

TO DO BY STUDENTS:

- Generate a single bet in EuroMillions (5 + 2 numbers) using sets.

```
// CONTAINERS & THE NEED FOR OPERATOR OVERLOADING
// CONTAINERS: a set of 'Person'
// TRY TO COMPILE THIS PROGRAM AND SEE WHAT HAPPENS

#include <iostream>
#include <iomanip>
#include <string>
#include <set>
using namespace std;

class Person
{
public:
	Person();
	Person(string pName, unsigned pAge);
	string getName() const { return name; };
	unsigned getAge() const { return age; };
	void setName(string pName) { name=pName; };
	void setAge(unsigned pAge) { age=pAge; };
private:
	string name;
	unsigned age;
};

//-------------------------------------------------------------------------
Person::Person()
{
	name = "";
	age = 0;
}

//-------------------------------------------------------------------------
Person::Person(string pName, unsigned pAge)
{
	name = pName;
	age = pAge;
}

//-------------------------------------------------------------------------
int main()
{
	set<Person> s;
	Person p;
	string name;
	unsigned age;

	for (int i=1; i<=3; i++)
	{
		cout << "name age " << i << " ? ";
		cin >> name >> age;

		p.setName(name);
		p.setAge(age);
		s.insert(p);
	}

	cout << endl;
	for (set<Person>::const_iterator i=s.begin(); i!=s.end(); i++)
		cout << setw(10) << left << i->getName() << " " << i->getAge()
<< endl;
}
```

```
// THE PREVIOUS PROGRAM GENERATES A COMPILER ERROR
// BECAUSE OPERATOR < IS NOT DEFINED FOR CLASS Person
#include <iostream>
#include <iomanip>
#include <string>
#include <set>

using namespace std;

class Person
{
      friend bool operator<(const Person& left, const Person& right);
public:
      Person();
      Person(string pName, unsigned pAge);
      string getName() const { return name; }; //const because of const_iterator in main
      unsigned getAge() const { return age; }; //const because of const_iterator in main
      void setName(string pName) { name=pName; };
      void setAge(unsigned pAge) { age=pAge; };
private:
      string name;
      unsigned age;
};

//-------------------------------------------------------------------------
Person::Person()
{
      name = "";
      age = 0;
}

//-------------------------------------------------------------------------
Person::Person(string pName, unsigned pAge)
{
      name = pName;
      age = pAge;
}

//-------------------------------------------------------------------------
bool operator<(const Person& left, const Person& right)
{
      return left.name < right.name; // OR left.age < right.age; you decide
}

//-------------------------------------------------------------------------
int main()
{
      set<Person> s;
      Person p;
      string name;
      unsigned age;

      for (int i=1; i<=3; i++)
      {
            cout << "name age " << i << " ? ";
            cin >> name >> age;

            p.setName(name);
            p.setAge(age);
            s.insert(p);
      }
```

```
      cout << endl;
      for (set<Person>::const_iterator i=s.begin(); i!=s.end(); i++)
            cout << setw(10) << left << i->getName() << " " << i->getAge()
<< endl;
}
```

NOTES:

- the comparison function, that implements **operator<**,
  must yield false when we compare a key with itself.
  Moreover,
  - if we compare two keys, they cannot both be "less than" each other,
  - and if k1 is "less than" k2, which in turn is "less than k3,
    then k1 must be "less than" k3
- it is **not** necessary to define **operator**== and **operator!=**

NOTES:

- an identical compiling error would occur if, for example, you wanted to declare a map whose key is a Person.

OVERLOADING THE () FUNCTION CALL OPERATOR / FUNCTION OBJECTS

- A **function object**
  is an instance of a class (an object)
  that defines the **function call operator**: **operator( )**
- Once the object is created,
  it can be invoked as you would invoke a function
  that's why it is termed a function object.

- Function objects are **used** extensively **by various generic STL algorithms**.

- The function call operator can only be defined as a member function.

- The same happens with the assignment operator, **operator=**
  (later, we shall see an example of **operator=** implementattion,
  for our own String class)

- **NOTE:** function objects can be created on the fly with **lambda's**, introduced in C++11. A **lambda** is an expression that generates a function object on the fly.
  (http://arne-mertz.de/2015/10/new-c-features-lambdas/)

```
// Overloading the function call operator

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

class RandomInt
{
public:
      RandomInt(int a, int b); // constructor
      int operator()();        // function call operator
private:
      int limInf, limSup; // interval limits: [limInf..limSup]
};

//------------------------------------------------------------------------
RandomInt::RandomInt(int a, int b)
{
      limInf = a; limSup = b;
}

//------------------------------------------------------------------------
int RandomInt::operator()()
{
      return limInf + rand() % (limSup - limInf + 1);
}

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

      RandomInt r(1,10); // create an object of type RandomInt (a FUNCTION OBJECT),
                         // initializing the limits of the interval to 1 and 10

      // once the object is created,
      // it can be invoked as you would invoke a function
      // that's why it is termed a FUNCTION OBJECT

      for (int i=1; i<=10; i++)
            cout << r() << endl;

      return 0;
}
```

A **FUNCTION OBJECT**
is an instance of a class that defines the function call operator

```
// Overloading the function call operator
// Generalizing the random number generator from the previsous example

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

class RandomInt
{
public:
      RandomInt(int a, int b);
      int operator()();
      int operator()(int b);
      int operator()(int a, int b);
private:
      int limInf, limSup; // interval limits: [limInf..limSup]
};

//------------------------------------------------------------------------
RandomInt::RandomInt(int a, int b)
{
      limInf = a; limSup = b;
}

//------------------------------------------------------------------------
int RandomInt::operator()()
{
      return limInf + rand() % (limSup - limInf + 1);
}

//------------------------------------------------------------------------
int RandomInt::operator()(int b)
{
      return limInf + rand() % (b - limInf + 1);
}

//------------------------------------------------------------------------
int RandomInt::operator()(int a, int b)
{
      return a + rand() % (b - a + 1);
}

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

      RandomInt r(1,10);

      cout << r() << endl;

      cout << r(100) << endl;

      cout << r(20,25) << endl;
     cout << r() << endl;

      return 0;
}
```

- Function objects are used extensively by various generic STL algorithms.

- In a previous example, we saw how to generate a sequence of random numbers in the interval [1..10].

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

...

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

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

- Suppose that one would like to generate a sequence in which the limits of the interval are set at run time.

- One could be tempted to do

```
int myRand(int a, int b)
{
      return a + rand() % (b - a  + 1);
}


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

      int limInf, limSup;
      cout << "limInf ? "; cin >> limInf;
      cout << "limSup ? "; cin >> limSup;

      generate(v2.begin(),v2.end(),myRand(limInf,limSup));
   ...
```

- This is syntactically incorrect; it will generate a compile error ...

- One could define 'limInf' and 'limSup' as global variables
  but this is not a recommended solution

```
// STL - ALGORITHMS

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;

int limInf, limSup; // GLOBAL VARIABLES  :-(

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

}

int myRand()
{
      return limInf + rand() % (limSup - limInf  + 1); //  :-(
}

int main() {

      srand((unsigned) time(NULL));

      vector<int> v2(10);

      cout << "limInf ? "; cin >> limInf;
      cout << "limSup ? "; cin >> limSup;

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

      return 0;
}
```

- The most commonly used solution is to use a **function object** as 3rd parameter to the **generate()** algorithm:

```
// FUNCTION OBJECTS & STL ALGORITHMS

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;

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

class RandomInt
{
public:
      RandomInt(int a, int b);
      int operator()();
private:
      int limInf, limSup; // interval limits: [limInf..limSup]
};

//-------------------------------------------------------------------------
RandomInt::RandomInt(int a, int b)
{
      limInf = a; limSup = b;
}

//-------------------------------------------------------------------------
int RandomInt::operator()()
{
      return limInf + rand() % (limSup - limInf + 1);
}

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

}

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
int main() {

      srand((unsigned) time(NULL));

      vector<int> v2(10);

      int limInf, limSup;
```

```
    cout << "limInf ? "; cin >> limInf;
    cout << "limSup ? "; cin >> limSup;

    RandomInt r(limInf,limSup); //instantiates object and sets limits

    generate(v2.begin(),v2.end(),r);

    // ALTERNATIVE:
    // using an unnamed temporary that will be destroyed at the end of the call
    //generate(v2.begin(),v2.end(),RandomInt(limInf,limSup));

    displayVec("random numbers",v2);

    return 0;
}
```

- Now, each time generate() calls its function parameter, it uses the call operator from object 'r'.

```
// CONTAINERS & FUNCTION OBJECTS
// An alternative way for sorting the set<Person> by name (or by age)
// (see previous example, about sets and operator< overloading for Person)
// is to create a function object, SortPersonByName,
// that defines the ordering, instead of overloading operator< for Person

#include <iostream>
#include <iomanip>
#include <string>
#include <set>

using namespace std;

class Person
{
public:
      Person();
      Person(string pName, unsigned pAge);
      string getName() const { return name; };
      unsigned getAge() const { return age; };
      void setName(string pName) { name=pName; };
      void setAge(unsigned pAge) { age=pAge; };
private:
      string name;
      unsigned age;
};

//-------------------------------------------------------------------------
class SortPersonByName
{
public:
      bool operator()(const Person &left, const Person &right) const;
};


bool SortPersonByName::operator()(const Person &left, const Person &right) const
{
      return left.getName() < right.getName();
}

//-------------------------------------------------------------------------
Person::Person()
{
      name = "";
      age = 0;
}

//-------------------------------------------------------------------------
Person::Person(string pName, unsigned pAge)
{
      name = pName;
      age = pAge;
}

//-------------------------------------------------------------------------
int main()
{
      set<Person,SortPersonByName> s;
      Person p;
      string name;
      unsigned age;
```

```
	for (int i=1; i<=3; i++)
	{
		cout << "name age " << i << " ? ";
		cin >> name >> age;

		p.setName(name);
		p.setAge(age);

		s.insert(p);
	}

	cout << endl;
	for (set<Person,SortPersonByName>::const_iterator i=s.begin();
i!=s.end(); i++)
		cout << setw(10) << left << i->getName() << " " << i->getAge()
<< endl;
}
```

LINK:
http://www.cplusplus.com/reference/stl/set/set/

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
         > class set;
```

NOTE:
- in a **map** declaration it is also possible to indicate a function object that is used for specifying the ordering of the elements of the map

```
template <
        class Key,                                     // map::key_type
        class T,                                       // map::mapped_type
        class Compare = less<Key>,                     // map::key_compare
        class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
         > class map;
```

// ACESSING COMMAND LINE ARGUMENTS

```
// Program (test.c) that shows its command line arguments.
// Command line arguments are passed to the program as an array of C-strings

// NOTE: run this program from the command prompt
// EX: C:\Users\username> test abc 123

#include <iostream>

using namespace std;

void main(int argc, char **argv) // OR void main(int argc, char *argv[])
{
	for (int i=0; i<argc; i++)
		cout << "argv[" << i << "] = " << argv[i] << endl;
}


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


// Program (sum.c) that shows the command line arguments
// Command line arguments are passed to the program as an array of C-strings

// NOTE: run this program from the command prompt
// EX: C:\Users\username> sum 123 456

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

using namespace std;

int c_string_to_int (char *intStr)
{
	int n;
	istringstream intStream(intStr);
	intStream >> n;
	return n;
}

void main(int argc, char *argv[]) // OR void main(int argc, char **argv)
{
	int n1, n2, n3;

	if (argc != 3)
	{
		cout << "USAGE: " << argv[0] << " integer1 integer2\n";
		exit(1);
	}
	n1 = c_string_to_int(argv[1]); // OR atoi(argv[1]);
	n2 = c_string_to_int(argv[2]);
	n3 = n1 + n2;
	cout << n1 << " + " << n2 << " = " << n3;

}
```

==========

MORE ON ...

... OVERLOADING: copy constructor / operator = / operator [ ]

... DESTRUCTORS: necessary when dynamic memory is allocated

==========

- **COPY CONSTRUCTORS**
  - By default, when an object is used to initialize another, C++ performs a bitwise copy, that is an identical copy of the initializing object is created in the target object
    ex:
    - `MyClass obj1 = obj2;`
    - `Myclass obj1(obj2);`
  - Although this is perfectly adequate for many cases
    – and generally exactly what you want to happen –
    there are situations in which a bitwise copy should **not** be used.
  - One of the most common is
    when an object allocates memory when it is created.
  - A **copy constructor** is a constructor that
    takes as parameter a constant reference to an object of the same class

```
// mystring.h
// a class from emulating C++ strings
// JAS

#ifndef _MYSTRING
#define _MYSTRING

// using namespace std; // should be avoided in header files because it implies
// that the namespace will be included in every file that includes this header file

class String
{
      friend std::ostream& operator<<( std::ostream& out, const String& right);
      friend bool operator==(const String& left, const String& right);
      friend String operator+(const String& left, const String& right);
public:
      String(); // Default constructor
      String(const char s[]); // Simple constructor
      String(const String& right); // Copy constructor
      ~String(); // Destructor
      String& operator=(const String& right); // Assignment operator
      char& operator[](int index); // WHEN IS EACH VERSION OF operator[] USED ?
      char  operator[](int index) const;
      int length() const;
private:
      char* buffer;  //space to be allocated must include '\0' string terminator
      int len; // perhaps, could be avoided ...?
};

#endif
```

```cpp
// mystring.cpp

// a class from emulating C++ strings (implementation)
// JAS

#include <iostream>
#include <cassert>
#include "mystring.h"

using namespace std;

//-----------------------------------------------------------------------
// DEFAULT CONSTRUCTOR (constructs an empty string)
String::String()
{
      cout << "DEFAULT CONSTRUCTOR\n"; //JUST FOR EXECUTION TRACKING

      len = 0;
      buffer = NULL; // No need to allocate space for empty strings
}

//-----------------------------------------------------------------------
// SIMPLE CONSTRUCTOR (constructs string from array of chars)
String::String(const char s[])
{
      cout << "SIMPLE CONSTRUCTOR from array of chars |" << s << "|\n";

      // Determine number of characters in string (alternative:strlen(s))
      len = 0;
      while (s[len] != '\0')
            len++;

      // Allocate buffer array, remember to make space for the '\0' character
      buffer = new char[len + 1];

      // Copy new characters (ALTERNATIVE: strcpy( buffer, s ))
      for (int i = 0; i < len; i++)
            buffer[i] = s[i];
      buffer[len] ='\0'; //terminator could be avoided ... why ? ... but ...
}

//-----------------------------------------------------------------------
// COPY CONSTRUCTOR
String::String(const String& right)
{
      cout << "COPY CONSTRUCTION from |" << right << "|" << endl;

      int n = right.length();
      buffer = new char[n + 1];
      for (int i = 0; i < n; i++)
            buffer[i] = right[i];
      buffer[n] = '\0';
      len = n;
}
```

```
//-------------------------------------------------------------------------
// ASSIGNMENT OPERATOR
String& String::operator=(const String& right)
{
      cout << "OPERATOR= |" << right << "|" << endl;

      if (this != &right)     // NOTE THIS TEST   (not "this" pointer...)
      {
            delete[] buffer; // Get rid of old buffer of 'this' object

            len = right.length();
            buffer = new char[len + 1];
            for (int i = 0; i < len; i++)
                  buffer[i] = right[i];
            buffer[len] = '\0';
      }
      return *this;   // WHY IS THIS DONE ?
                      //   NOTE: COULD RETURN 'String' INSTEAD OF 'String&'... but...
}                     // ...MODIFY AND ANALYSE THE "cout" MESSAGES
// RETURN TYPE FROM OPERATOR= SHOULD BE THE SAME AS FOR THE BUILT-IN TYPES (C++Primer, 4th 3d, p.493)
//-------------------------------------------------------------------------
// SUBSCRIPT OPERATOR FOR const  OBJECTS (returns rvalue)
char String::operator[](int index) const
{
      assert((index >=0 ) && (index < len));
      return buffer[index];
}

//-------------------------------------------------------------------------
// SUBSCRIPT OPERATOR FOR non-const OBJECTS (returns lvalue)
char& String::operator[](int index)
{
      assert((index >=0 ) && (index < len));
      return buffer[index];
} //NOTE: be careful when returning references to class data members !!!

//-------------------------------------------------------------------------
// STRING LENGTH member function
int String::length() const
{
      return len;
}

//-------------------------------------------------------------------------
// DESTRUCTOR - in this case, it is fundamental to a have a destructor
String::~String()
{
      if (buffer != NULL)
            cout << "DESTRUCTION OF |" << buffer << "|" << endl;
      else
            cout << "NOTHING TO DESTRUCT\n";
      if (buffer != NULL)
            delete[] buffer;
}
```

```
//-------------------------------------------------------------------------
// EQUALITY OPERATOR
bool operator==(const String& left, const String& right)
{
      if (left.length() != right.length())
            return false;
      for (int i=0; i<left.length(); i++)
            if (left.buffer[i] != right.buffer[i])
                  return false;
      return true;
}

//-------------------------------------------------------------------------
String operator+(const String& left, const String& right)
{
      //if (right.length() == 0)
      //    return left;

      cout << "OPERATOR+ (" << left << "," << right << ")\n";

      int newlen = left.length() + right.length();

      // allocate space for temporary resulting string
      char *tmpCStr = new char[newlen + 1]; // C-string

      // concatenate the 2 strings
      int pos = 0;
      for (int i=0; i<left.length(); i++)
            tmpCStr[pos++] = left.buffer[i];
      for (int i=0; i<right.length(); i++)
            tmpCStr[pos++] = right.buffer[i];
      tmpCStr[pos] = '\0';

      // create String object from temporary string
      String tmpStr(tmpCStr); // invoque String constructor

      // destroy temporary string
      delete[] tmpCStr; // C-string

      return tmpStr;
}
//-------------------------------------------------------------------------

// STRING OUTPUT OPERATOR
std::ostream& operator<<(std::ostream& out, const String& right)
{
      int n = right.length();
      for (int i=0; i<n; i++)
            cout << right[i];
      return out;
}
```

```
// My STRING CLASS
// a class from emulating C++ strings (implementation)
// JAS

// A program for testing my "String class"
// main.cpp


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

using namespace std;

//-----------------------------------------------------------------------
int main(void)
{
      cout << "String s0;    - ";
      String s0;

      cout << "String s1 = \"ABC\";    - ";
      String s1 = "ABC";

      cout << "String s2(\"DEF\");    - ";
      String s2("DEF");

      char s[] = "GHI";
      cout << "String s3(s);    - ";
      String s3(s);

      cout << "String s4 = s1;    - ";
      String s4 = s1;
      // UNCOMMENT AND INTERPRET WHAT HAPPENS   (JAS: see results after END page)
      /*
      cout << "s0 = s1;    - ";
      s0 = s1;
      */

      // UNCOMMENT AND INTERPRET WHAT HAPPENS   (JAS: see results after END page)
      //cout << "---------\n";
      //cout << "s0 = s1 + s2;    - ";
      //s0 = s1 + s2;

      cout << "---------\n";
      cout << "s0 = " << s0 << endl;
      cout << "s1 = " << s1 << endl;
      cout << "s2 = " << s2 << endl;
      cout << "s3 = " << s3 << endl;
      cout << "s4 = " << s4 << endl;
      cout << "---------\n";

      cout << "s4[0] = " << s4[0] << endl;
      cout << "modifying s4[0] = a\n";
      s4[0] = 'a';
      cout << "s4 = " << s4 << endl;
      if (s1 == s4)
            cout << "s1 EQUAL TO s4\n";
      else
            cout << "s1 NOT EQUAL TO s4\n";
      cout << "---------\n";
}
```

- **WHEN IS A DESTRUCTOR NEEDED ?**
  - If no destructor is provided,
    a default destructor will be automatically generated.
    The default destructor has an empty body, that is, it performs no actions.
  - A destructor is only necessary
    if an object requires some kind of resource management.
  - The most common housekeeping task is
    to avoid a memory leak by releasing any dinamically allocated memory.

- **WHEN IS A COPY CONSTRUCTOR EXECUTED ?**
  - C++ defines 2 distinct types of situations in which
    the **value of one object is given to another**:
    - initialization
    - assignment
  - Initialization (=> copy constructor is invoked) can occur any of 3 ways
    - when an object explicitly initializes another, such in a declaration
      - `Myclass x = y;`
    - when a copy of an object is made to be passed to a function
      - `func(y);`
    - when a temporary object is generated
      (most commonly, as a return value)
      - `y = func(); // y receiving a temporary returned object`
      - note: in this case assignment operator is also invoked

  - assignment (=> operator= is invoked)
    - ```
      Myclass x;
      Myclass y;
      ...
      x = y;
      ```
      ```

- **THE "BIG THREE "**
  - The **assignment operator**, **copy constructor** and **destructor** are collectively called "the "big three".
  - A simple **rule of thumb** is that **if you define a destructor then you should always provide a copy constructor and an assignment operator,** and make all three perform in a similar fashion.
    - Analyse what would happen if in the just implemented String class we had defined a destructor but had forgotten to define the copy constructor (a copy constructor would be automatically generated for us):

```
String a = "Peter";
...
{
   String b = a; // memberwise copy;
                 // buffer[] for a and b is the same
   ...
} //destructor b.~String is invoked, a.buffer[] is deleted
```

  - You must implement them for any class that manages heap memory.
  - The equivalence of a **copy constructor** and the **assigment operator** is clear:
    - both are initializing a new value using an existing value.
  - But the **assignment operator** is both deleting an old value and creating a new one.
    You must make sure the first part of this task matches the action of the **destructor**.