==========

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 equalDates(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 (equalDates(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 equalDates(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 equalDates(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 (equalDates(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
- **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*).

- class **Node**
  {
  ...
  friend class **List**;
  }
- class **List**
  {
  ...
  friend class **Node**;
  }

- General outline of how you set things:

```
class Node; //forward declaration

class List
{
   public:
      ...
   friend class Node;
      ...
};

class Node
{
   ...
}
```

==========

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] = {10,20,30};

  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(3).setMonth(5).setYear(2018);
  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.
      //  or the compare function (see below)
      // It is already defined for 'int'

}


template
< class T,                          // set::key_type/value_type
  class Compare = less<T>,          // set::key_compare/value_compare
                                    // is a function object (see next pages)
  class Alloc = allocator<T>        // set::allocator_type;
                                    // for now you will not need it
> class set;
```

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 inf, int sup); // constructor
      int operator()();        // function call operator
private:
      int limInf, limSup; // interval limits: [limInf..limSup]
};

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

//------------------------------------------------------------------------
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 inf, int sup);
      int operator()();
      int operator()(int sup);
      int operator()(int inf, int sup);
private:
      int limInf, limSup; // interval limits: [limInf..limSup]
};

//------------------------------------------------------------------------
RandomInt::RandomInt(int inf, int sup)
{
      limInf = inf; limSup = sup;
}

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

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

//------------------------------------------------------------------------
int RandomInt::operator()(int inf, int sup)
{
      return inf + rand() % (sup - inf + 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 inf, int sup)
{
      return inf + rand() % (sup - inf  + 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 object 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;
```

==========

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 dinamically 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
```

```
// 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";  // MESSAGE 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;

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

```
//-------------------------------------------------------------------------
// 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 ed, 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;
}

//-------------------------------------------------------------------------
// CONCATENATION OPERATOR
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;`
      - `Myclass x(y);`
    - when a copy of an object is made to be passed to a function
      - `func(x);`
    - 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 that the first part of this task matches the action of the **destructor**.

==========

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;

}
```

==========

NAMESPACES

==========

Namespace concept

- A namespace is a **collection of name definitions**,
  such as class definitions, function definitions and variable declarations
- If a program uses **classes and functions written by different programmers**,
  it may be that the **same name** is **used for different things**
- Namespaces help us deal with this problem

The "using" directive

- #include <iostream> places names such as **cin** and **cout** in the **std** namespace
- The program does not know about names in the **std** namespace until you add
  using namespace std;
- if you do not use the **std** namespace,
  you can define **cin** and **cout** to behave differently !!!

The global namespace

- Code that you write is in a namespace
  - it is in the global namespace unless you specify a namespace
  - the global name space is refered to using just **::**
- The global namespace does not require the using directive

Creating and using a namespace

- To place code in a namespace, use a namespace grouping

```
namespace Name_Space_Name
{
      // Some_Code
}
```

- To use the namespace created, use the appropriate using directive

```
using namespace Name_Space_Name;
```

Declaring and defining functions in a namespace

- To add a function to a namespace,
  declare the function in a namespace grouping

```
namespace ns1
{
      void greeting();
}
```

- To define a function declared in a namespace,
  define the function in a namespace grouping

```
namespace ns1
{
      void greeting()
      {
        Cout << "Hello!\n";
      }
}
```

Using a function

- To use a function defined in a namespace, include the **using directive** in the program where the namespace is to be used
- Call the function as the function would normally be called

```
int main( )
{
      {
            using namespace ns1;
            greeting( );
      }
 //...
}
```

Name conflicts

- If the same name is used in two namespaces
  the namespaces cannot be used at the same time
- Example: If my_function is defined in namespaces **ns1** and **ns2**,
  the two versions of my_function could be used in one program
  by using local using directives this way:

```
{
      using namespace ns1;
      my_function();
}
```

```
{
      using namespace ns2;
      my_function();
}
```

- NOTE:
  - A using directive potentially introduces a name
  - If **ns1** and **ns2** both define **my_function()**

    ```
    using namespace ns1;
    using namespace ns2;
    ```

    is OK, provided **my_function()** is never used!

- Suppose you have the namespaces below:

```
namespace ns1
{
   fun1();
   my_function();
}
```

```
namespace ns2
{
   fun2();
   my_function();
}
```

- Is there an easier way to use both namespaces considering that my_function() is in both?
- **Using declarations** (not directives) allow us to select individual functions to use from namespaces
  - using ns1::fun1; //makes only fun1 in ns1 available
- The scope resolution operator - :: - identifies a namespace here
- Means we are using only namespace ns1's version of fun1
- If you only want to use the function once, call it like this:
  - ns1::fun1( );
- A **using declaration** (ex: **using std::cout;**) makes only one name available from the namespace
- A **using directive** makes all the names in the namespace available
- A using declaration introduces a name into your code: no other use of the name can be made

```
using ns1::my_function;
using ns2::my_function;
```

  is **illegal**, even if my_function is never used.

Naming Namespaces

- To avoid choosing a name for a namespace that has already been used
  - Add your name initials to the name of the namespace
  - Or, use some other unique string (the name of your company, ...)

```
// NAMESPACES - usage example

#include <iostream>

using std::cout;

//-----------------------------------------------------------------------
// Namespace declaration

namespace ns1
{
	void hello();
}

namespace ns2
{
	void hello();
}

//-----------------------------------------------------------------------
int main()
{
	ns1::hello();
	ns2::hello();
}


//-----------------------------------------------------------------------
// Namespace implementation

namespace ns1
{
	void hello()
	{
		cout << "hello 1 !\n";
	}
}

namespace ns2
{
	void hello()
	{
		cout << "hello 2 !\n";
	}
}
```

```
// NAMESPACES - usage example

#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

// Namespace declaration & function implementation

namespace ns1
{
  string creator="Mary";

  void hello()
  {
    cout << "hello from " << creator << "!\n";
  }
}

namespace ns2
{
  string creator="John";

  void hello()
  {
    cout << "hello from " << creator << "!\n";
  }
}

//-------------------------------------------------------------------------
int main()
{
  cout << "ns1 creator: " << ns1::creator << endl;
  cout << "ns2 creator: " << ns2::creator << endl << endl;

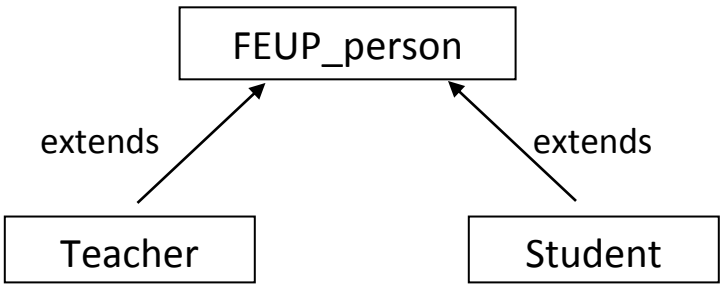
  ns1::hello();
  ns2::hello();
}


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

ALTERNATIVE main():

int main()
{
      {
            using namespace ns1;
            hello();
      }

      {
            using namespace ns2;
            hello();
      }

}
```

==========

OBJECT ORIENTED LANGUAGES

==========

To support the principles of object-oriented programming (OOP),
all OOP languages have three traits in common:

Encapsulation :

- the mechanism that **binds** together **code** and the **data** it manipulates
  and keeps both **safe from outside interference and misuse**.
  - code and data may be combined in such a way that
    a self-contained "**black-box**" is created

Inheritance :

- the process by which **one object can acquire the properties of another object**
- this is important because it supports the **concept of classification**
  - most knowledge is made manageable by hierarchical classifications
    - ex: a Human is a Primate
      a Primate is a Mammal
      a Mammal is an Animal

Polymorphism:

- In programming languages, polymorphism means that
  some code or **operations** or **objects** **behave differently** **in different contexts**.
- Helps reduce the complexity by allowing the **same interface** to be used
  to access a general class of actions

- In C++, both **compile-time** (static) and **run-time** (dynamic) polymorphism
  are supported
  - Forms of **compile-time** polymorphism in C++
    - overloading
    - templates
  - Forms of **run-time** polymorphism in C++
    - inheritance + virtual functions

- Polymorphism (run-time)
  - describes a set of objects of different classes with similar behavior.
  - Inheritance is used to express the commonality between the classes, and virtual functions enable variations in behavior.

Virtual functions

- In contrast to **all other function calls** that are **statically bound**
  (**the compiler determines** which function is called
  by looking only at the type of the implicit parameter variable)
  virtual functions are **dynamically bound**
  (when a virtual function is called,
  the **actual type of the implicit parameter object determines**
  which implementation of the virtual function is invoked)

==========

INHERITANCE

==========

- **Inheritance** is a mechanism for enhancing existing working classes.
- If a new class needs to be implemented and
  a class representing a more general concept is already available,
  then the new class can inherit from the existing class.
- The existing, more general class is called the **base class** or **parent class**.
- The more specialized class that inherits from the base class
  is called the **derived class** or **child class**.

- A **derived class**
  - automatically has
    all the member variables and functions of the base class
  - can have
    additional member variables and/or member functions

- Some examples:
  - example 1:
    - in a store, books, magazines and movies are publications
      having some properties (data members) in common:
      - an ID, a title
    - all these types of publications have some unique properties
    - Book, Magazine and Movie
      can be declared as classes derived from a base class: Publication
  - example 2:
    - a Human **is a** Primate
    - a Primate **is a** Mammal
    - a Mammal **is an** Animal
    - Mammal can be derived from Animal
    - Primate can be derived from Mammal
    - Human can be derived from Primate
  - example 3:
    - FEUP teachers are FEUP people
    - FEUP students are FEUP people

- **Class (/type) hierarchy**:



  - People at FEUP have some characteristics/behaviors in common (?):
    - characteristics (class attributes): ID, name, address, ...
    - behaviours (class methods): show record, change address, ...

  - Students have some things in special:
    - characteristics: course ID, year, classes taken, ...
    - behaviours : change course, add class taken, ...

  - Teachers have some things in special:
    - characteristics: rank (assistant, professor, ...), classes taught, ...
    - behaviours : promote, add a class taught, ...

BASE CLASS: Feup_Person

```
#include <string>

class FeupPerson
{
public
  FeupPerson(int id, std::string name, std::string address);
  void changeAddress(std::string newAddress);
  void showRecord();
protected: //accessible inside the class and by all of its subclasses
  int id;
  std::string name;
  std::string address;
};
```

"Protected" qualifier

- protected members of a class appear to be private outside the class, but are accessible by derived classes
- Using protected members of a class is a convenience to facilitate writing the code of derived classes.
- Protected members are not necessary
  - derived classes can use the public methods of their ancestor classes to access private members
- Many programmers consider it bad style to use protected member variables because
  - the designer of the base class has no control over the authors of derived classes

DERIVED CLASS: Student

```
#include <vector>
#include <string>
#include "FeupPerson.h"
#include "Class.h"

class Student : public FeupPerson {
public:
  Student(int id, std::string name, std::string address, std::string
course, int year);
  void addClassTaken(Class* newClass);
  void changeCourse(std::string newCourse);
  void showRecord();
private:
  std::string course;
  int year;
  std::vector<Class*> classesTaken;  //NOTE the type of array elements
};
```

- **`public`** , the base class access specifier, **is needed**.
  If it were ommited **`Student`** would inherit **privately**.

- If the base class access specifier were **private**,
  public and **protected** members (namely, public methods) of the base class
  would become private members of the derived class.
  This means that:
  - they are still acessible by members of the derived class
  - but cannot be accessed by parts of your program
    that are not members of either the base or derived class.
- If the base class access specifier were **protected**,
  public and **protected** members of the base class
  would become protected members of the derived class.

```
#include <iostream>
#include <string>

using namespace std;

class Person
{
public:
  Person() { name = "UNKNOWN"; };
  Person(string name) : name(name) { };
  string getName() const { return name; }
  void show() { cout << name; };
private:  // try with "protected"
  string name;
};

class Student : public Person // try with "protected"
{
public:
  Student(string name, string course) : Person(name), course(course) {};
  string getCourse() const { return course; }
  void show() { cout << this->getName() << " - " << course; };
  //void show() { cout << name << " - " << course; }; // to use with "protected"
attributes in Person
  //void show() { Person::show();  cout << " - " << course; };
private:
  string course;
};

int main()
{
  Person  p1("Ann");
  p1.show(); cout << endl;
  Student s1("Peter","MIEIC");
  s1.show(); cout << endl;
  cout << s1.getName() << endl; // try with "class Student : protected Person"
}
```

Constructors of the base and the derived classes

```
// in FeupPerson.cpp
FeupPerson::FeupPerson(int id, std::string name, std::string address)
{
  this->id = id;
  this->name = name;
  this->address = address;
}

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

// in Student.cpp
Student::Student(int id, std::string name, std::string address, std::string
course, int year) : FeupPerson(id, name, address)  // call to the base constructor
{
  this->course = course;
  this->year = year;
}
```

Constructing an object of the derived classe

Student bio123 = Student(123, “John Silva”, “St. John Street”, "MIB", 3);

- ID = 123
- name = “John Silva”
- person address = “St. John Street”
- course name = "MIB"
- classes taken = *none yet*
- year = 3

Redefining a method in the child class

```
class FeupPerson {
public:
  FeupPerson(int id, std::string name, std::string address);
  void showRecord();
  void changeAddress(std::string newAddress);
private:
  int id;
  std::string name;
  std::string address;

};
```

```
class Student : public FeupPerson {
public:
  Student(int id, std::string name, std::string address, std::string
course, int year);
  void showRecord(); // redefine the method to display course & classes
  void addClassTaken(Class* newClass);
  void changeCourse(std::string newCourse);
private:
  std::string course;
  int year;
  std::vector<Class*> classesTaken;
};


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


void FeupPerson::showRecord() {  // definition in FeupPerson.cpp
  std::cout << "----------------------------\n";
  std::cout << "Name: " << name << " ID: " << id << " Address: " << address
<< "\n";
  std::cout << "----------------------------\n"; }



void Student::showRecord(){  // definition in Student.cpp
  std::cout << "----------------------------\n";
  std::cout << "Name: " << name << " ID: " << id << " Address: " << address
<< "\n";
  std::cout << "Course: " << course << "\n";
  std::vector<Class*>::iterator it;
  std::cout << "Classes taken:\n";
  for (it = classesTaken.begin(); it != classesTaken.end(); it++)
  {
    Class* c = *it;
    std::cout << c->getName() << "\n";
  }
  std::cout << "----------------------------\n";
}

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

Usage examples

```
FeupPerson peter = FeupPerson(987, “Peter Lee”, “St. Peter Street”);
//OR FeupPerson peter(987, “Peter Lee”, “St. Peter Street”)

Student bio123 = Student(123, “John Silva”, “St. John Street”, "MIB", 3);

peter.showRecord();

Class* c1 = new Class(“EDA”);
bio123.addClassTaken(c1);

bio123.showRecord();
```

Building a derived class

- A **derived class** (child / descendant class)
  inherits all the members of the **parent class** (ancestor class)
- The parent class contains all the code common to the child classes
- The derived class can **add** member variables and functions
- The derived class can **re-declare** and **re-define** member functions
  of the parent class that will have a different definition in the derived class
- Definitions are not given for inherited functions that are not to be changed

Private members of the parent class

- A member variable (or function) that is private in the parent class
  is not accessible to the child class
- The parent class member functions must be used
  to access the private members of the parent

Derived Class Constructors

- A **base class constructor is not inherited** in a derived class
- The base class constructor can be invoked
  by the constructor of the derived class

- The **constructor of a derived class** begins by
  invoking the constructor of the base class in the initialization section:

```
Student::Student(int id, std::string name, std::string address, std::string
course, int year) : FeupPerson(id, name, address)   // call to the base constructor
{
  this->course = course;
  this->year = year;
}
```

- These are the key points about constructors for derived classes:
  - The base-class object is constructed first.
  - The derived-class constructor should pass base-class information to
    a base-class constructor via a member initializer list.
  - The derived-class constructor should initialize the data members that were
    added to the derived class.
- If a derived class constructor does not invoke a base class constructor explicity,
  the base class default constructor will be used
  (NOTE: if the constructor was overloaded
  don't forget to implement the default constructor)

- If class B is derived from class A and class C is derived from class B,
  when a object of class C is created
  - The base class A's constructor is the first invoked
  - Class B's constructor is invoked next
  - C's constructor completes execution
- Destructors are invoked in reverse order: C –>B -> A

Using objects of the ancestor and the descendent classes

- An object of a class type can be used wherever any of its ancestors can be used
- An ancestor cannot be used wherever one of its descendents can be used

```
FeupPerson p;
Student s;
...
p = s; // possible BUT SOME DATA IS SLICED AWAY - SLICING PROBLEM

s = p; // NOT POSSIBLE -> COMPILER ERROR
```

The slicing problem

- It is possible in C++ to avoid the slicing problem
- Using pointers to dynamic variables
  we can assign objects of a derived class to variables of a base class
  without loosing members of the derived class object

Function redefinition vs. overloading

- A function **redefined** in a derived class
  has the same number and type of parameters
  (the function signature is the same)
  - the derived class has only one function with the same name as the base class
- An **overloaded** function
  has a different number and/or type of parameters than the base class

Function signature

- is the name of the function with the sequence of types in the parameter list
  not including
  the `const` keyword and the `ampersand` (&)
  - Some compilers allow overloading based on including const or not including const

```
//inherit_02.sln
//JAS - 2012/12/07

//-------------------------------------------------------------------------
// FeupPerson.h
#ifndef FEUP_PERSON_H
#define FEUP_PERSON_H

#include <string>

class FeupPerson {
public:
  FeupPerson();
  FeupPerson(int id, std::string name, std::string address);
  void showRecord();
  void changeAddress(std::string newAddress);
protected:
  int id;
  std::string name;
  std::string address;
};

#endif

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//FeupPerson.cpp

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

FeupPerson::FeupPerson()
{
  this->id = 0;
  this->name = "";
  this->address = "";
}

FeupPerson::FeupPerson(int id, std::string name, std::string address)
{
  this->id = id;
  this->name = name;
  this->address = address;
}

void FeupPerson::showRecord() {
  std::cout << "----------------------------\n";
  std::cout << "Name    : " << name << std::endl
            << "ID      : " << id << std::endl
            << "Address : " << address << std::endl;
  std::cout << "----------------------------\n";
}

void FeupPerson::changeAddress(std::string newAddress)
{
  this->address = newAddress;
}

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

```
//---------------------------------------------------------------------
// Student.h
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <vector>
#include <string>
#include "FeupPerson.h"
#include "Class.h"

class Student : public FeupPerson {
public:
  Student() { };
  Student(int id, std::string name, std::string address, std::string
course, int year);
  void showRecord();
  void addClassTaken(Class* newClass);
  void changeCourse(std::string newCourse);
private:
  std::string course;
  int year;     // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;
};
#endif
//---------------------------------------------------------------------
//---------------------------------------------------------------------
//Student.cpp

#include "Student.h"

Student::Student(int id, std::string name, std::string address,
             std::string course, int year) : FeupPerson(id, name, address)
{
  this->course = course;
  this->year = year;
}

void Student::addClassTaken(Class* newClass)
{
  classesTaken.push_back(newClass);
}

void Student::showRecord()
{
  std::cout << "-----------------------------\n";
  std::cout << "Name    : " << name << std::endl
            << "ID      : " << id << std::endl
            << "Address : " << address << std::endl;
  std::cout << "Course  : " << course << std::endl;

  std::vector<Class*>::iterator it;

  std::cout << "Classes taken:\n";
  for (it = classesTaken.begin(); it != classesTaken.end(); it++){
    Class* c = *it;
    std::cout << c->getName() << std::endl;
  }
  std::cout << "-----------------------------\n";
}
//---------------------------------------------------------------------
```

```
//---------------------------------------------------------------------
// Class.h
#ifndef CLASS_H
#define CLASS_H

#include <string>

class Class {  // What a class name!!!
public:
  Class(std::string name);
  std::string getName();
private:
  std::string name;
};

#endif  // CLASS_H

//---------------------------------------------------------------------
//---------------------------------------------------------------------
// Class.cpp
#include "Class.h"

Class::Class(std::string name)
{
  this->name = name;
}

std::string Class::getName()
{
  return name;
}

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

//---------------------------------------------------------------------
// main.cpp
#include <iostream>
#include <string>
#include "Student.h"

int main(){

  FeupPerson dei987 = FeupPerson(987, "Peter Lee", "St. Peter Street");

  Student bio123 = Student(123, "John Silva", "St. John Street", "MIB", 3);

  Class *c1 = new Class("EDA");
  Class *c2 = new Class("EDE");
  Class *c3 = new Class("EDI");

  dei987.showRecord();

  bio123.addClassTaken(c1);
  bio123.showRecord();

  bio123.addClassTaken(c2);
  bio123.addClassTaken(c3);
  bio123.showRecord();
}

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

- **Replace the previous main() function with the following one:**

```
//---------------------------------------------------------------------
// main.cpp
#include <iostream>
#include <string>
#include "Student.h"

int main(){

  Class* c1 = new Class("EDA");
  Class* c2 = new Class("EDE");
  Class* c3 = new Class("EDI");

  Student *mieic234 = new Student(234, "John Souza", "St. John Street",
"MIEIC", 2);
  mieic234->addClassTaken(c1);
  mieic234->addClassTaken(c3);
  mieic234->showRecord();

  FeupPerson *mieic345 = new Student(345, "Liz Tanner", "St. Liz Street",
"MIEIC", 2);
  mieic345->showRecord();  //NOTE THE RESULT !!!

  //mieic345->addClassTaken(c1);  //UNCOMMENT AND INTERPRET THE RESULT

}

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

- **NOTE** that:
  - a base class pointer can be used to point to a derived class object
  - ...but ... what happens when **`showRecord()`** method
    is invoked through the base class pointer
    that points to the derived class object ... ?!

==========

POLYMORPHISM (virtual functions)

==========

- **Ability of type A to <u>appear as</u> and <u>be used like</u> another type B**
  - *ex: a <u>Student</u> object can be used <u>in place</u> of a <u>FeupPerson</u> object*

- Suppose one would like to **build** a **vector of Teachers / Students**.

```
int main()
{
  std::vector<FeupPerson> p(3);

  p[0] = Teacher(987, "Pedro Santos", "Rua do Pedro", "Assistente",
"MIB");
  p[1] = Student(123, "Nuno Silva", "Rua do Nuno", "MIB", 3);
  p[2] = Student(234, "Ana Sousa", "Rua da Ana", "MIEIC", 2);

  for (unsigned int i=0; i<p.size(); i++)
        p[i].showRecord();}
  ...
}
```

- Although the assigments in yellow are possible,
  some of the fields of Teacher/Student shall be lost (<u>slicing problem</u>)

- This problem is very typical of code that
  needs to manipulate objects from a mixture of data types.
  - Derived-class objects are usually bigger than base-class objects
    and objects of different derived-classes have different sizes
  - A vector of objects cannot deal with this variation in sizes
  - **But a vector of pointers to objects can ... (see next example)**
    if **`showRecord()`** is declared **`virtual`**, in **`FeupPerson`** class

- In the following, the code for **<u>class Teacher</u>** is shown

```
-------------------------------------------------------------
// Teacher.h
#ifndef TEACHER_H
#define TEACHER_H
#include <iostream>
#include <vector>
#include <string>
#include "FeupPerson.h"
#include "Class.h"

class Teacher : public FeupPerson {
public:
  Teacher() {};
  Teacher(int id, std::string name, std::string address, std::string rank,
std::string course);
  void showRecord();
  void addClassTaught(Class* newClass);
  void changeCourse(std::string newCourse);

  void showCourse() {std::cout << course << std::endl;};

private:
  std::string rank;
  std::string course;
  std::vector<Class*> classesTaught;
};

#endif
//=========================================================================
//Teacher.cpp
#include "Teacher.h"

Teacher::Teacher(int id, std::string name, std::string address,
                 std::string rank, std::string course) : FeupPerson(id,
name, address) {
  this->rank = rank;
  this->course = course;
}

void Teacher::addClassTaught(Class* newClass) {
  classesTaught.push_back(newClass);
}

void Teacher::showRecord() {
  std::cout << "----------------------------\n";
  std::cout << "Name    : " << name << std::endl
            << "ID      : " << id << std::endl
            << "Address : " << address << std::endl
            << "Rank    : " << rank << std::endl
            << "Course  : " << course << std::endl;

  std::vector<Class*>::iterator it;

  std::cout << "Classes taught:\n";
  for (it = classesTaught.begin(); it != classesTaught.end(); it++) {
    Class* c = *it;
    std::cout << c->getName() << std::endl;
  }
  std::cout << "----------------------------\n";
}
//-------------------------------------------------------------------------
```

```
int main()
{
  std::vector<FeupPerson *> p(3);

  p[0] = new Teacher(987, "Pedro Santos", "Rua do Pedro", "Assistente",
"MIB");
  p[1] = new Student(123, "Nuno Silva", "Rua do Nuno", "MIB", 3);
  p[2] = new Student(234, "Ana Sousa", "Rua da Ana", "MIEIC", 2);

  for (unsigned int i=0; i<p.size(); i++)
        p[i]->showRecord();
  ...
}
```

- Note that
  - the assignments of the above code
    assign a derived-class pointer of type Teacher* or Student*
    to a base-class pointer of type FeupPerson*
  - this is **legal**
  - the reverse (from a base-class pointer to a derived-class one) is an **error**
- **But** when one runs the above code, the ouput is something like:

```
----------------------------
Name    : Pedro Santos
ID      : 987
Address : Rua do Pedro
----------------------------
----------------------------
Name    : Nuno Silva
ID      : 123
Address : Rua do Nuno
----------------------------
----------------------------
Name    : Ana Sousa
ID      : 234
Address : Rua da Ana
----------------------------
```

- The compiler generated code only to call the FeupPerson's **`showRecord()`** method ...
  ... because **`p[i]`** is of type **`FeupPerson*`**

- However it is possible to **alert the compiler** that the function call must be preceeded by the appropriate function selection.
- This selection must be done at run-time.

- To tell the compiler that a particular call needs to be bound dynamically the function must be tagged as **virtual**:

```
class FeupPerson {
public:
  FeupPerson(); // NOTE: CONSTRUCTORS CAN'T BE MADE VIRTUAL
  FeupPerson(int id, std::string name, std::string address);
  //virtual ~FeupPerson();  // SEE NOTE ON NEXT PAGES
  virtual void showRecord();
  void changeAddress(std::string newAddress);
protected:
  int id;
  std::string name;
  std::string address;
};
```

- The outupt is:

```
-----------------------------
Name    : Pedro Santos
ID      : 987
Address : Rua do Pedro
Rank    : Assistente
Course  : MIB
Classes taught:
-----------------------------
-----------------------------
Name    : Nuno Silva
ID      : 123
Address : Rua do Nuno
Course  : MIB
Classes taken:
-----------------------------
-----------------------------
Name    : Ana Sousa
ID      : 234
Address : Rua da Ana
Course  : MIEIC
Classes taken:
-----------------------------
```

- Such a selection/call combination is called **dynamic binding (**or **late binding)** in contrast to the traditional call which always invokes the same function being called **static binding**.
- The **virtual** keyword must be used **in the base class**.
- All functions with the same name and parameter types in derived classes are then **automatically virtual**.
    - However it is considered good taste to supply the **virtual** keyword for the derived-classes as well

- Whenever a virtual function is called,
  the compiler determines the type of the implicit parameter
  in the particular call **at run time**.
  - Ex: `p[i]->showRecord()`

    always calls the function belonging to the actual type of the object
    to which **`p[i]`** points,
    either
    `Teacher::showRecord()`
    or
    `Student::showRecord()`

- Only member functions can be virtual.

- You should use virtual functions
  only when you need the flexibility of dynamic binding at run time.

- The
  `vector<FeupPerson *> p(3);`
  collects a mixture of both kinds of FEUP persons.

- Such a **collection** is called **polymorphic**.

- However ...
  if you try to call a method that is not implemented in the base class
  - as `showRecord()` is -
  for example:
  - `p[0]->addClassTaught(c1);`

  you'll get a compiler error:
  - `error C2039: 'addClassTaught' : is not a member of 'FeupPerson'`

- This problem can be solved using a **dynamic_cast** to downcast the pointer

```
int main()
{
  Class* c1 = new Class("EDA");
  Class* c2 = new Class("EDU");

  std::vector<FeupPerson *> p(3);

  p[0] = new Teacher(987, "Pedro Santos", "Rua do Pedro", "Assistente",
"MIB");
  p[1] = new Student(123, "Nuno Silva", "Rua do Nuno", "MIB", 3);
  p[2] = new Student(234, "Ana Sousa", "Rua da Ana", "MIEIC", 2);

  std::cout << "BEFORE DYNAMIC_CAST:\n";
  for (unsigned int i=0; i<p.size(); i++)
    p[i]->showRecord();

  for (unsigned int i=0; i<p.size(); i++)
  {
    Teacher *t = dynamic_cast<Teacher *> (p[i]);
    if (t != NULL)
      t->addClassTaught(c1);
    else
    {
      Student *s = dynamic_cast<Student *> (p[i]);
      if (s != NULL)
        s->addClassTaken(c2);
    }
  }

  std::cout << "AFTER DYNAMIC_CAST:\n";
  for (unsigned int i=0; i<p.size(); i++)
      p[i]->showRecord();
}
```

```
BEFORE DYNAMIC_CAST:
----------------------------
Name    : Pedro Santos
ID      : 987
Address : Rua do Pedro
Rank    : Assistente
Course  : MIB
Classes taught:
----------------------------
----------------------------
Name    : Nuno Silva
ID      : 123
Address : Rua do Nuno
Course  : MIB
Classes taken:
----------------------------
----------------------------
Name    : Ana Sousa
ID      : 234
Address : Rua da Ana
Course  : MIEIC
Classes taken:
----------------------------
AFTER DYNAMIC_CAST: (and call to addClassTaught() and addClassTaken())
```

```
AFTER DYNAMIC_CAST: (and call to addClassTaught() and addClassTaken())
----------------------------
Name    : Pedro Santos
ID      : 987
Address : Rua do Pedro
Rank    : Assistente
Course  : MIB
Classes taught:
EDA
----------------------------
----------------------------
Name    : Nuno Silva
ID      : 123
Address : Rua do Nuno
Course  : MIB
Classes taken:
EDU
----------------------------
----------------------------
Name    : Ana Sousa
ID      : 234
Address : Rua da Ana
Course  : MIEIC
Classes taken:
EDU
----------------------------
```

Dynamic casts

- **dynamic_cast** can be used only with pointers and references to objects.
  Its purpose is to ensure that the result of the type conversion is
  a valid complete object of the requested class.
- **dynamic_cast** is always successful when we cast a class to one of its base classes
- When **dynamic_cast** cannot cast a pointer because it is not a complete object of the required class , it returns a null pointer to indicate the failure.

Named casts

- General form for named cast:
  - *cast-name<type> (expression)*
- where *cast-name* may be
  - `static_cast`
  - `dynamic_cast`
  - `const_cast`
  - `reinterpret_cast`
- *type* is the target type of the conversion
- *expression* is the value to be cast
- Note: use casts with caution.

Static cast

- Any type of conversion that the compiler performs implicitly can be explicitly requested using a static_cast.
  - ```
    double d = 97.0;
    int i = static_cast<int> (d);
    ```
  - Compilers often generate a warning for assignment of a larger arithmetic type to a smaller type. The cast informs both the reader and the compiler that we are aware of and not concerned with the potential loss of precision.
- A static_cast can also be used to perform conversions that the compiler will not generate automatically:
  - ```
    double q = static_cast<double>(3) / 4;
    ```
    OR
  - ```
    double d;
    void *p = &d; // OK: a void pointer is compatible with any other pointer
    // pointer p can be used as argument of some special functions
    // that need to receive a void * as argument
    // ex: pthread_create() to be studied in the Operating Systems course.
    // That function will convert the void * back to a double *, using:
    double *pd = static_cast<double*>(p);
    ```

Const cast

- **const_cast** can be used to remove or add **const** to a variable.
- Only for "advanced" uses… (*see: http://en.cppreference.com/w/cpp/language/const_cast*)
- Note: even though **const_cast** may remove constness (or volatility) from any pointer or reference, using the resulting pointer or reference to write to an object that was declared const (or to access an object that was declared volatile) invokes undefined behavior.

Reinterpret cast

- **reinterpret_cast** generally performs a low-level reinterpretation of the bit pattern of its operands.
- Must be used with caution. The result is machine dependent. Its use requires completely understanding the types involved as well as the details of how the compiler implements the cast.
- Example
  - ```
    int *pi = new(int);
    *pi = 65 + 66 * 256 + 67 * 256 * 256;
    char *pc = reinterpret_cast<char *> (pi);
    cout << pc << endl; // which is the result?
    ```

Dynamic casts

- *See previous section.*

Old-style casts

- Prior to the introduction of named cast operators, an explicit cast was performed using one of two forms:
  - ***(type) expression;*** `// C-language-style cast notation`
  - ***type (expression);*** `// function-style cast notation`
- Depending on the types involved, an old-style cast has the same behavior as a const_cast, a static_cast, or a reinterpret_cast.
  - ```
    int i; double d;
    i = i + (int) d; // same as static_cast
    ```
  - ```
    const char *s = "ABCDE";
    func((char *) s); // same as const_cast; casts away const
                      // BUT func() CANNOT MODIFY s ...!
    ```
  - ```
    int *pi; // SEE PREVIOUS EXAMPLE
    char *pc = (char *) pi; // same as reinterpret_cast
    ```
- Notes:
  - Although the old-style cast notation is supported by the Standard C++, its use is recommended only when writing code to be compiled under the C language.
  - Two main reasons are that C++ casting operators are intended to make the casting operations more explicit (see above) and safer:
    - Example:

```
char ch = 65;         // 1 byte
int *p1 = (int*)&ch; // 4 bytes

*p1 = 112;           // may result in run-time error
                     // or in destruction of other variables

int *p2 = static_cast<int*>(&c); // results in compile-time error
```
```

Virtual details

- To define a function differently in a derived class and to make it virtual
  - Add keyword **virtual** to the function declaration in the base class
  - "virtual" is not needed for the function declaration in the derived class, but is often included
  - **"virtual" is not added to the function definition**
- Virtual functions require considerable overhead so excessive use reduces program efficiency
  (see section 13.7 from Deitel, 7th ed, for an explanation how it works)
- Making a function virtual tells the compiler that you don't know how the function is implemented and to wait until the function is used in a program, then get the implementation from the object.
  - This is called **late binding**

Overriding vs. redefinition

- Virtual functions whose definitions are changed in a derived class are said to be overridden (*in Portuguese, "substituídas"/"ignoradas"/"canceladas"*)
- Non-virtual functions whose definitions are changed in a derived class are redefined

Virtual Destructors

- **Destructors should be made virtual**
  (In C++, construtors cannot be made virtual –
  to create an object, you must know its exact type)
- Consider

```
Base *pBase = new Derived;
…
delete pBase;
```

  - If the destructor in **Base** is virtual,
    the destructor for **Derived** is invoked
    as **pBase** points to a **Derived** object,
    returning **Derived** members to the freestore
- The **Derived** destructor in turn calls the **Base** destructor
- If the **Base** destructor is not virtual, only the **Base** destructor is invoked
  This leaves **Derived** members, not part of **Base**, in memory.
- *NOTE:*
  *when you have a definition "***Derived d;***" the destructor of* **Base** *will always be called.*

```
// WHEN ARE THE CONSTRUCTORS & DESTRUCTORS OF BASE & DERIVED CLASSES CALLED?
//
// ILLUSTRATING THE NEED FOR A VIRTUAL DESTRUCTOR IN A BASE CLASS
// JAS
#include <iostream>
#include <cstring>
using namespace std;

//---------------------------------------------------------------------
class Base
{
public:
  Base();
  virtual ~Base();
  //~Base(); // TRY THIS ALTERNATIVE & ANALYSE THE RESULTS
private:
  int dummyB; // just to have an attribute
  //other attributes
};
//---------------------------------------------------------------------
class Derived: public Base
{
public:
  Derived();
  ~Derived();
private:
  int *dummyD; // will point to dinamically allocated memory
  // other attributes
};
```

```
//-----------------------------------------------------------------------
Base::Base()
{
  cout << "Base constructor called\n";
  // TO DO: initialize attributes
}
//-----------------------------------------------------------------------
Base::~Base()
{
  cout << "Base destructor called\n";
  // TO DO
}
//-----------------------------------------------------------------------
Derived::Derived()
{
  cout << "Derived constructor called\n";
  // TO DO: initialize attributes, dynamically allocating memory
}
//-----------------------------------------------------------------------
Derived::~Derived()
{
  cout << "Derived destructor called\n";
  // TO DO: free the dinamically allocated memory
}
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
int main()
{
  cout << "\n Base b1; --------------------------\n";
  Base b1;

  cout << "\n Derived d1; ------------------------\n";
  Derived d1;

  cout << "\n Base *b2 = new Base(); -------------\n";
  Base *b2 = new Base();

  cout << "\n Derived *d2 = new Derived(); -------\n";
  Derived *d2 = new Derived();

  cout << "\n Base *d3 = new Derived(); ----------\n";
  Base *d3 = new Derived();

  cout << "\n delete b2;   ----------------------\n";
  delete b2;

  cout << "\n delete d2;   ----------------------\n";
  delete d2;

  cout << "\n delete d3;   ----------------------\n";
  delete d3;

  cout << "\n END OF main() ---------------------\n";

  return 0;
}
```

```
// ANOTHER EXAMPLE OF POLYMORPHISM
(C:\Users\jsilva\Documents\AULAS 2010-2011\__PROG\8-Programas\clocks2)

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <ctime>
#include <cassert>

using namespace std;

//=========================================================
/*
   A class that describes a time of day
   (between 00:00:00 and 23:59:59)
*/
class Time
{
public:

   Time(int hour, int min, int sec); //Constructs a time of day
   Time(); //Constructs a Time object that is set to the time at which the
constructor executes
   int get_hours() const; //Gets the hours of this time
   int get_minutes() const; //Gets the minutes of this time
   int get_seconds() const; //Gets the seconds of this time

private:
   int time_in_secs;
};

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

class Clock
{
public:
   Clock(bool use_military);
   virtual string get_location() const; // TRY TO REMOVE virtual qualifier
   virtual int get_hours() const;       // TRY TO REMOVE virtual qualifier
   int get_minutes() const;
   bool is_military() const;
private: // COULD BE DECLARED protected
         // (see comment in TravelClock::get_hours() )
   bool military; // means 24 hour notation -> 00:00 ... 23:59
};

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

class TravelClock : public Clock
{
public:
   TravelClock(bool mil, string loc, int diff);
   string get_location() const;
   int get_hours() const;
private:
   string location;
   int time_difference;
};

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

```
//=========================================================
Time::Time(int hour, int min, int sec)
{
   assert(0 <= hour);
   assert(hour < 24);
   assert(0 <= min);
   assert(min < 60);
   assert(0 <= sec);
   assert(sec < 60);

   time_in_secs = 60 * 60 * hour + 60 * min + sec;
}


Time::Time()
{
  time_t now = time(0);
  struct tm t;
  localtime_s(&t, &now);
  time_in_secs = 60 * 60 * t.tm_hour + 60 * t.tm_min + t.tm_sec;
}

int Time::get_hours() const
{
   return time_in_secs / (60 * 60);
}

int Time::get_minutes() const
{
   return (time_in_secs / 60) % 60;
}

int Time::get_seconds() const
{
   return time_in_secs % 60;
}

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

Clock::Clock(bool use_military)
{
   military = use_military;
}

string Clock::get_location() const
{
   return "Local";
}

int Clock::get_hours() const
{
   Time now;
   int hours = now.get_hours();
   if (military) return hours;
   if (hours == 0)
      return 12;
   else if (hours > 12)
      return hours - 12;
   else
      return hours;
}
```

```
int Clock::get_minutes() const
{
   Time now;
   return now.get_minutes();
}

bool Clock::is_military() const
{
   return military;
}

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

TravelClock::TravelClock(bool mil, string loc, int diff)
   : Clock(mil)
{
   location = loc;
   time_difference = diff;
   while (time_difference < 0)
      time_difference = time_difference + 24;
}

string TravelClock::get_location() const
{
   return location;
}

int TravelClock::get_hours() const
{
   int h = Clock::get_hours();   // NOTE THIS
   if (is_military())
      return (h + time_difference) % 24;
   else
   {
      h = (h + time_difference) % 12;
      if (h == 0) return 12;
      else return h;
   }
}

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

int main()
{
   vector<Clock *> clocks(3);
   clocks[0] = new Clock(true);
   clocks[1] = new TravelClock(true, "Madrid", 1);
   clocks[2] = new TravelClock(false, "Azores", -1);

   for (int i = 0; i < clocks.size(); i++)
   {
      cout << clocks[i]->get_location() << " time is "
         << clocks[i]->get_hours() << ":"
         << setw(2) << setfill('0')
         << clocks[i]->get_minutes()
         << setfill(' ') << "\n";
   }
   return 0;
}
```

Pure virtual functions or abstract functions

```
// Example adapted from
// http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming#cite_note-tcpl-1

#include <iostream>
#include <string>

using namespace std;

class Animal
{
    public:
        Animal(const string& name) : name(name) {}
        virtual string talk() = 0; //pure virtual function (or abstract function)
        const string name; // public attribute !!! TO DO: MAKE IT PRIVATE
};

class Cat : public Animal
{
    public:
        Cat(const string& name) : Animal(name) {}
        virtual string talk() { return "Meow!"; }
};

class Dog : public Animal
{
    public:
        Dog(const string& name) : Animal(name) {}
        virtual string talk() { return "Arf! Arf!"; }
};


int main()
{
        Animal* animals[] =      //NOTE the initialization
        {
                new Cat("Mr. Jinks"),
                new Cat("Garfield"),
                new Dog("Milou")
        };

        for(int i = 0; i < 3; i++)
        {
                cout << animals[i]->name << ": " << animals[i]->talk() << endl;
                delete animals[i];
        }
        return 0;
}
```

TO DO:

- in class Cat
  comment `virtual string talk() { return "Meow!"; }`
  and interpret result
- then replace in class Animal
  `virtual string talk() = 0;`
  with
  `virtual string talk() {return "...!";}`
  and interpret result
- Remove all `virtual` keywords and interpret results

Pure virtual functions (or abstract functions)

- Pure virtual function is a "function hat" has no body at all !
  - indicated by a prototype that has no implementation
  - this is indicated by the **pure specifier**, **= 0**, following the function prototype;
- A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
- When we add a pure virtual function to our class, we are effectively saying, "it is up to the derived classes to implement this function".
- Using a pure virtual function has two main consequences:
  - First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it **can not be instantiated**!
  - Second, any derived class must define a body for this function.

Interface classes

- An **interface class** is a class that has no members variables, and where all of the functions are pure virtual!
- In other words, the class is purely a definition, and has no actual implementation.
- Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

```
class Document {
public:
    //  Requirements for derived classes;
    //  they must implement these functions.
    virtual string identify() = 0;
    virtual string whereIs() = 0;
};
```

Interface classes

- Interfaces, like classes
  - define a set of properties, methods, and events.
- But unlike classes, interfaces
  - do not provide implementation;
  - they are implemented by classes, and
    defined as separate entities from classes.

- An interface represents a contract, in that
  a class that implements an interface must implement
  every aspect of that interface exactly as it is defined.

- With interfaces, you can
  - define features as
    small groups of closely related members.
  - develop enhanced implementations for your interfaces
    without jeopardizing existing code,
    thus minimizing compatibility problems.
  - add new features at any time
    by developing additional interfaces and implementations.

- Although interface implementations can evolve,
  interfaces themselves cannot be changed once published.
  - Changes to a published interface may break existing code.
  - If you think of an interface as a **contract**,
    it is clear that both sides of the contract have a role to play.
    - The **publisher** of an interface agrees never to change that interface,
    - and the **implementer** agrees to implement the interface
      exactly as it was designed.

==========

OBJECT ORIENTED LANGUAGES (remembering …)

==========

To support the principles of object-oriented programming (OOP),
all OOP languages have three traits in common:

Encapsulation :

- the mechanism that **binds** together **code** and the **data** it manipulates and keeps both **safe from outside interference and misuse**.
    - code and data may be combined in such a way that a self-contained "**black-box**" is created

Inheritance :

- the process by which **one object can acquire the properties of another object**
- this is important because it supports the **concept of classification**
    - most knowledge is made manageable by hierarchical classifications
        - ex: a Human is a Primate
              a Primate is a Mammal
              a Mammal is an Animal

Polymorphism:

- In programming languages, polymorphism means that **some code or operations or objects behave differently in different contexts**.
- Helps reduce the complexity by allowing the **same interface** to be used **to access a general class of actions**

- In C++, both **compile-time** (static) and **run-time** (dynamic) polymorphism are supported
    - Forms of **compile-time** polymorphism in C++
        - overloading
        - templates
    - Forms of **run-time** polymorphism in C++
        - inheritance + virtual functions

- Polymorphism (run-time)
  describes a set of objects of different classes with similar behavior.
- Inheritance is used to express the commonality between the classes, and ...
  ... virtual functions enable variations in behavior.

Virtual functions
in contrast to all other function call that are **statically bound**
(**the compiler determines** which function is called
by looking only at the type of the implicit parameter variable)
virtual functions are **bound dynamically**
(when a virtual function is called,
the **actual type of the implicit parameter object determines**
which implementation of the virtual function is invoked)

==========

EXCEPTION HANDLING

==========

Things sometimes go wrong … ☹

- User input errors
- Device errors
  - disk I/O
- Physical limitations
  - memory is exausted
- Software component failures
  - function performs incorrectly

Some approaches already seen for handling exceptional conditions

- The fail predicate used by the stream I/O library
- The assert macro
- exit the program

Alternative (historical) ways of handling exceptional conditions

- Assume errors will not occur ☹
- Print an error message
- Special return values
  - ```
    Stack s;
    bool ok = s.push(10);
    ```
  - but, in the following case ...how to return the special value ?
    ```
    int i = s.pop();
    ```
  - ... in this way ...?
    ```
    bool ok = s.pop(i); //bool Stack::pop (int & value) ???
    int i = s.pop(ok); //int Stack::pop (bool & status) ???
    ```
  - alternative (the user must take care …)
    ```
    if (s.size() > 0)
        {i = s.pop()};
    ```
- External flags
  - ```
    int n = atoi(numString);
    if (errno == ERANGE) // => #include <cerrno>
      ..... ;
    ```
- Use `assert` to halt execution
- Error Handlers

- **Some errors can be detected and resolved at the point they occur**
- **Others are non-local; i.e., must be resolved at a "higher-level "**

Exception Handling Mechanism

- Later addition to C++
- In C++, exception handling proceeds by:
  - Some library software or your code signals that something unusual has happened
  - This is called **throwing an exception**
  - At some other place in your program you place the code that deals with the exceptional case
  - This is called **handling the exception**
- Exception handling is meant to be used **sparingly**

try-throw-catch mechanism

- An error is signaled by throwing an exception
- Any type can be thrown
- If not handled (caught) locally, function exits
- Does not return to calling point
- Unwinds call stack, looking for an appropriate handler.

Throwing an Exception

- Example:

```
double futureValue(double initialAmount, double tax, int numYears)
{
   if (tax < 0 || numYears < 0)
   {
      logic_error description("illegal future_value parameter");
      throw description;
      // ALTERNATIVE:
      // throw logic_error("illegal future_value parameter");
   }
  return initialAmount * pow(1 + tax / 100, numYears);

 }
```

- Purpose:
  - Abandon this function and **throw** a value to an **exception handler**.

Catching Exceptions

- Supply a handler with the try statement:

```
try
{
   // code that could cause a problem
}
catch (type_name & e) //NOTE: can be received by reference (pref.) or by value
{
   // handler (executed if a problem occurs)
}
```

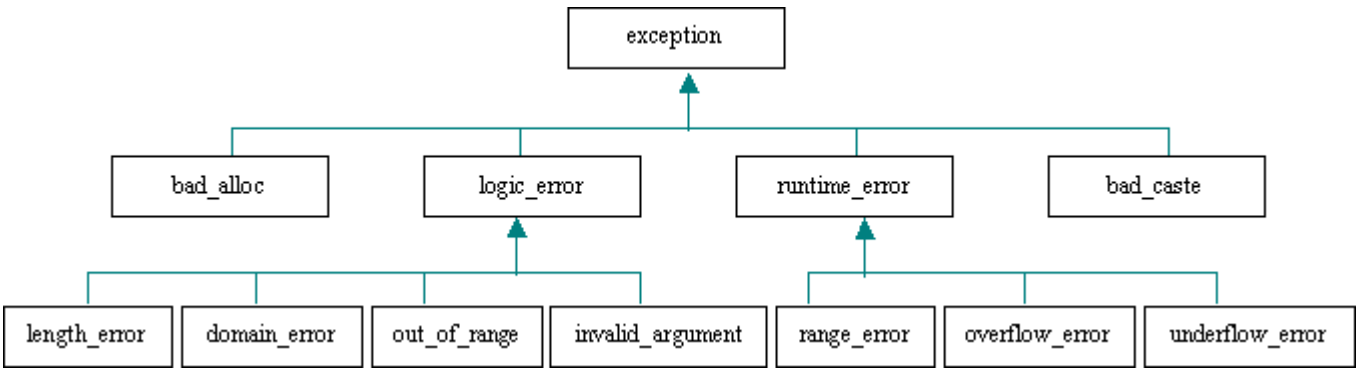
- The try block encloses code that you want to "try" but that could cause a problem
- If an error is thrown in the try clause, execution goes to the catch clause
- The type of the catch-block parameter identifies the kind of value the catch-block can catch
- If no appropriate handler is found, the next outer try block is examined

```
#include <iostream>
#include <cmath>
#include <stdexcept>

using namespace std;

double futureValue(double initialAmount, double tax, int numYears)
{
	if (initialAmount < 0 || tax < 0 || numYears < 0)
	{
		//logic_error description("illegal futureValue parameter");
		//throw description;
		throw logic_error("illegal futureValue parameter");
	}
	return initialAmount * pow(1 + tax / 100, numYears);
}


int main()
{
	double value, amount, tax;
	int years;
	cout << "amount ? "; cin >> amount;
	cout << "tax ? "; cin >> tax;
	cout << "numYears ? "; cin >> years;
	try
	{
		value = futureValue(amount, tax, years);
		cout << "future value = " << value << endl;

	}
	catch (logic_error& e)
	{
		cerr << "Processing error: " << e.what() << "\n";
	}

}
```

Catching Exceptions – Syntax

```
try
{
   statements
}
catch (type_name_1 variable_name_1)
{
   statements
}
catch (type_name_2 variable_name_2)
{
   statements
}
...
catch (type_name_N variable_name_N)
{
 statements
}
catch (...) // DEFAULT CATCH BLOCK - handle exceptions not covered
{
 statements
}
```

Standard exception hierarchy (*partial*), in <stdexcept>



- All of the exceptions thrown by the C++ Standard Library are objects of classes in this hierarchy.
- Each class in the hierarchy supports a **what()** method that returns a char* string describing the exception. You can use this string in an error message.

Values Thrown and Caught

- Can throw (and catch) any type, including integer
- Suggestion: throw a type that derives directly or indirectly from **std::exception**
- Implicit conversion (e.g., int to double) not performed on thrown values

Users can define their own exceptions

```
// objects of this class can carry the kind of information
// you want thrown to the catch-block
class MyApplicationError
{
public:
 MyApplicationError(const string& r);
 string& what() const; // returns C++-string, instead of C-string (see previous page)
private:
 string reason;
};

MyApplicationError::MyApplicationError(const string& r) : reason(r) {}

string& MyApplicationError::what() const
{
 return reason;
}

...

try
{
 . . .
      throw MyApplicationError("illegal value");
 . . .
}
catch (MyApplicationError& e)
{
 cerr << "Caught exception " << e.what() << "\n";
}
```

Inheriting from standard exceptions

```
class FutureValueError : public logic_error
{
public:
 FutureValueError(string reason);
};

FutureValueError::FutureValueError(string reason): logic_error(reason) {}

...
...
...

try
{
 code
}

catch (FutureValueError& e) // only catches FutureValueError
{  handler1  }

catch (logic_error& e) // catches all other logic_error
{  handler2  }

catch (bad_alloc& e)
{  handler3  }
```

Nested try-catch blocks

- Although a try-block followed by its catch-block can be nested inside another try-block
  - It is almost always better to place the nested try-block and its catch-block inside a function definition, then invoke the function in the outer try-block
- An error thrown but not caught in the inner try-catch-blocks is thrown to the outer try-block where it might be caught
- If no appropriate handler is found, the next outer try block is examined

```
#include <iostream>
#include <cmath>
#include <stdexcept>

using namespace std;

double f2(double x)
{
      if (x<0) throw invalid_argument("invalid argument in f2() call");
      else return sqrt(x);
}

double f1(double x)
{
      return 1/f2(x); // the error in f2() is not caught here
}

int main()
{
      try
      {
            cout << f1(-2) << endl;

      }
      catch (invalid_argument& e)
      {
            cerr << "error in f1() call: " << e.what() << "\n";
      }

}
```

When to Throw An Exception

- Throwing exceptions is generally reserved for those cases when handling the exceptional case depends on how and where the function was invoked
- In these cases it is usually better to let the programmer calling the function handle the exception

Rethrowing Exceptions

- The code within a catch-block can throw an exception
- This feature can be used to pass the same or a different exception up the chain of exception handling blocks
- use **throw with no arguments** to rethrow error

Exceptions and Constructors / Destructors

- Constructors and destructors do not return a value
- **Throwing an exception is a clean way to indicate failure in a constructor**
- If a constructor fails, the object is not created
  - The destructor isn't called
  - Subtle source of leaks (can be avoided - see next example)

```
DataArray::DataArray(int size)
{
   data = new int[size];
   try
   {
      init(); // call to auxiliary initialization function that can throw an error …
   }
   catch (...) // Catch any exception that init() throws
   {
      delete[] data;
      data = NULL;
      throw; // Rethrow exception
   }
}
```

- **Don't throw exceptions from within a destructor**
  - destructors are invoked as part of the process of stack unwinding during the recovery from an exception,
  - if a destructor throws an exception this would yield 2 exceptions the one currently being handled by stack unwinding and the one thrown by the destructor ... (it is unclear which one would take priority; the program is halted)
- Caveat: static values are initialized before main is entered. There is no try block to catch exceptions.
-

Exceptions – additional notes Overuse of exceptions

- Throwing an exception allows you to transfer flow of control to almost any place in your program
- Such un-restricted flow of control is sometimes considered poor programming style as it makes programs difficult to understand

Exceptions and performance (*source*: Errors and Exception Handling (Modern C++) - 2016 https://msdn.microsoft.com/en-us/library/hh279678.aspx)

- The exception mechanism has a very minimal performance cost if no exception is thrown. If an exception is thrown, the cost of the stack traversal and unwinding is roughly comparable to the cost of a function call.
- Additional data structures are required to track the call stack after a try block is entered, and additional instructions are required to unwind the stack if an exception is thrown. However, in most scenarios, the cost in performance and memory footprint is not significant.

try-throw-catch review

- The **try-block** includes a **throw-statement**
- If an exception is thrown, the try-block ends and the **catch-block** is executed
- If no exception is thrown, then after the try-block is completed, execution continues with the code following the catch-block(s)
- Catch blocks are examined top to bottom
- `catch(...)` is used to catch any exception; should be last in list
- Executes the first handler that matches, then stops processing that exception
- C++ unwinds call stack in search of a try block to handle the exception; the exception handler could lie one or more function calls up the stack of execution.
- As the control jumps up in the stack, in a process called stack unwinding, all code remaining in each function past the current point of execution is skipped.
- Local objects and variables in each function that is unwound are destroyed as if the code finished the function normally.
- However, in stack unwinding, pointer variables are not freed, and other cleanup is not performed.
- Before each function is terminated, destructors are called on all stack variables
- An uncaught exception ends your program

```
// EXCEPTION HANDLING
// A LAST EXAMPLE

#include <iostream>
#include <stdexcept>

using namespace std;

int main()
{
  int numSlices, numGlasses;
  double ratio;

  try
  {
    cout << "Enter number of \"pizza\" slices:\n";
    cin >> numSlices;
    cout << "Enter number of glasses of \"orange juice\":\n";
    cin >> numGlasses;

    if (numGlasses <= 0)
      throw numSlices;

    ratio = static_cast<double>(numGlasses) / numSlices;
    cout << numSlices << " slices.\n"
      << numGlasses << " glasses of juice.\n"
      << "You have " << ratio
      << " glasses of juice for each slice.\n";
  }
  catch (int n)
  {
    cerr << n << " slices, and NO JUICE!!!\n"
      << "Go buy some JUICE.\n\n";
  }

  cout << endl;
  cout << "End of program.\n";
  cout << "END OF CLASSES :-))\n";
  return 0;
}
```

THE END