1 Classes and Data Abstraction I, Exercise - date Example

We begin looking at object-oriented aspects of C++. There are two parallel versions of an application which uses date computation. There is an object-oriented C version and a C++ version. Both compile to about the same object. At this point, all C++ supplies is a superior organizational notation and tools. At the machine level, the code generated is about the same in either case.

- C version written using best practice to avoid confusion and name clashes with other libraries.
- Consider **date** routines as a separate library. Show how declarations and definitions work in C++ compared with C. What goes in header files? What goes in implementation files?
- C++ member functions get unqualified access to object members (data, types and functions) Don't need to qualify with object name as C routines do. **this** keyword to get at object itself. Can also be used (needlessly) to qualify access to object data members. Unqualified member names can hide global functions/objects. Use :: prefix if you need to get at these.
- using **private:** keyword to protect data members. client code versus implementation code. implementation code for one object might be client code for another object it uses. Protecting data members allows implementation to change without code in client objects having to change. (They will most likely need to be recompiled.)
- using **class** instead of **struct**
- declaring methods **const. const** key word appears directly after parentheses. Must appear in decl and def. Part of "signature".
- static keyword

Associating functions with objects. They get declared as members of the objects they are to manipulate. These members (unless they are declared **virtual**) don't take up space in the object. Declaration and definition. Defining function in class declaration makes it inline. Don't do this just because it's convenient - remember about code duplication.
Constructors - a special member function for initialization. A behind-the-scenes function call. No return value. No address.

- **date** constructors. No-arg constructor: In defining vars, parentheses *must* be left off, otherwise compiler thinks its a function declaration; in creating temporaries, parentheses must be included, e.g. *cout ¡¡ date()* prints today's date. When using **new**, either notation can be used, e.g. **new date** or **new date()**.
- () vs = variation for single arg constructor
- effect of making constructor private
- specifying construction of contained or inherited objects.

In speaking of pieces of C++ code and their relationship to the various classes, it is useful to differentiate between "implementation" code and "client" code. Implementation code for a particular class is the code used to define its methods. Client code is other code which manipulates instances of the class. This terminology is always used with respect to a particular class. A piece of implementation code for one class may be client code for another class.
Write both the following codes, compile and compare the differencies.
–object-oriented C version

```
/* date.c  C code for date example - based on example in
           B. Stroustrup, The C++ Programming Language (2nd ed.), pp. 145.

   The implementation code for the date library is written so it could be
   used with a host of other libraries with minimum confusion, and minimum
   possibility of name clashes with the other libraries.
*/

#include <stdio.h>
#include <time.h>

struct date
{
  enum month_names
  {DATE_JAN, DATE_FEB, DATE_MAR, DATE_APR, DATE_MAY, DATE_JUN,
   DATE_JUL, DATE_AUG, DATE_SEP, DATE_OCT, DATE_NOV, DATE_DEC} month;
  char day;
  short year;
};
```

```
const char date_limits[12] =
{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

void date_set(struct date *dt, enum month_names m, int d, int y)
{
  dt->month = m;
  dt->day = d;
  if ((dt->year = y) < 100) dt->year += 1900 ;
}

void date_set_today(struct date *dt)
{
  time_t t;
  struct tm *localt;

  time(&t);
  localt = localtime(&t);
  dt->month = localt->tm_mon;
  dt->day = localt->tm_mday;
  dt->year = 1900 + localt->tm_year;
}

struct date *date_next(struct date *dt)
{
  if /* is more work than just incrementing the day needed? */
  (
    /* only if the result is > 28 */
    ++dt->day > 28 &&
    /* and leap year does not apply */
    (dt->month != DATE_FEB || dt->day != 29 ||
     dt->year % 4 != 0 || (dt->year % 100 == 0 && dt->year % 400 != 0)) &&
    /* and the fixed monthly constraints are exceeded */
    dt->day > date_limits[dt->month]
  )
  {
    dt->day = 1;
    if (++dt->month > DATE_DEC)
    {
      dt->month = DATE_JAN;
      ++dt->year;
    }
  }
  return dt;
}

FILE *date_print(const struct date *dt, FILE *f)
{

  fprintf(f, "%02u/%02u/%02u", dt->month + 1, dt->day, dt->year % 100);
  return f;
}

int main()
{
  struct date today, bjarnes_bday, day;

  date_set_today(&today);
  printf("Today is ");
  date_print(&today, stdout); puts(".");
  date_set(&bjarnes_bday, DATE_DEC, 30, 1950);
  printf("Bjarne's birthday is ");
  date_print(&bjarnes_bday, stdout); puts(".");
  day = bjarnes_bday;
  printf("On "); date_print(date_next(&day), stdout);
  printf(", he was one day old.\n");
  printf("On "); date_print(date_next(&day), stdout);
  printf(", he was two days old.\n");

  return 0;
}
```

–and a C++ version

```
// datecpp.C  C++ code for date example - based on example in B. Stroustrup,
//            The C++ Programming Language (2nd ed.), pp. 145.

#include <iostream.h>
#include <iomanip.h>
#include <time.h>

class date
{
public:
  enum month_names
  {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
  date();
  date(month_names m, int d, int y) { set(m, d, y); }
  date &set(month_names m, int d, int y);
  date &next();
  ostream &print(ostream &) const;
private:
  month_names month;
  char day;
  short year;
  static const char limits[12];
};

const char date::limits[12] =
{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

date::date()
{
  time_t t;
  tm *localt;

  ::time(&t);
  localt = ::localtime(&t);
  month = (month_names)localt->tm_mon;
  day = localt->tm_mday;
  year = 1900 + localt->tm_year;
}
```

```
date &date::set(month_names m, int d, int y)
{
  month = m;
  day = d;
  if ((year = y) < 100) year += 1900 ;
  return *this;
}

date &date::next()
{
  if // is more work than just incrementing the day needed?
  (
    // only if the result is > 28
    ++day > 28 &&
    // and leap year does not apply
    (month != FEB || day != 29 ||
     year % 4 != 0 || (year % 100 == 0 && year % 400 != 0)) &&
    // and the fixed monthly constraints are exceeded
    day > limits[month]
  )
  {
    int m = month;
    day = 1;
    if (++m > DEC)
    {
      m = JAN;
      ++year;
    }
    month = (month_names)m;
  }
  return *this;
}

ostream &date::print(ostream &os) const
{
  char fill = os.fill('0');
  os << setw(2) << month + 1 << '/' << setw(2) << (int)day << '/'
     << setw(2) << year % 100;
  os.fill(fill);
  return os;
}
```

```
int main()
{
  cout << "Today is "; date().print(cout) << ".\n";
  date bjarnes_bday(date::DEC, 30, 1950);
  cout << "Bjarne's birthday was "; bjarnes_bday.print(cout) << ".\n";
  date day = bjarnes_bday;
  cout << "On "; day.next().print(cout) << ", he was one day old.\n";
  cout << "On "; day.next().print(cout) << ", he was two days old.\n";

  return 0;
}
```

2 Classes and Data Abstraction I, Lab Exercise 1 - Complex Numbers

Create a class called **Complex** for performing arithmetic with complex numbers. Write a driver program to test your class. Complex numbers have the form

$$realpart + imaginarypart * i$$

where **i** is

$$\sqrt{-1}$$

Use floating–point variables to represent the **private** data of the class. Provide a constructor function that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided. Provide **public** member functions for each of the following:

- Addition of two **Complex** numbers: The real parts are added together and the imaginary parts are added together

- Substraction of two **Complex** numbers: The real part of the right operand is subtracted from the real part of the left operand and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.

- Printing **Complex** numbers in the form **(a,b)** where **a** is the real part and **b** is the imaginary part.

  The output should appear as follows:

  ```
  (1,7) + (9,2) = (10,9)
  (10,1) - (11,5) = (-1,-4)
  ```

```
#include <iostream>
using std::cout;
using std::endl;
/* Write class definition for Complex */

// member function definitions for class Complex
Complex::Complex( double real, double imaginary )
{
setComplexNumber( real, imaginary );
}
void Complex::addition( const Complex &a )
{
   /* Write statement to add the realPart of a to the class
      realPart */
   /* Write statement to add the imaginaryPart of a to the
      class imaginaryPart */
}
void Complex::subtraction( const Complex &s ) {
   /* Write a statement to subtract the realPart of s from the
      class realPart */
   /* Write a statement to subtract the imaginaryPart of s from
      the class imaginaryPart */
}
void Complex::printComplex( void )
{
cout << '(' << realPart << ", " << imaginaryPart << ')';
}
void Complex::setComplexNumber( double real, double imaginary )
{
   realPart = real;
   imaginaryPart = imaginary;
}
int main() {
   Complex b( 1, 7 ), c( 9, 2 );
   b.printComplex();
   cout << " + ";
   c.printComplex();
   cout << " = ";
   b.addition( c );
   b.printComplex();
   cout << '\n';
   b.setComplexNumber( 10, 1 );
   c.setComplexNumber( 11, 5 );
   b.printComplex();
   cout << " - ";
   c.printComplex();
   cout << " = ";
   b.subtraction( c );
   b.printComplex();
   cout << endl;
   return 0;
}
```

Tips:

- You must write the definition for class **Complex**. Use the details provided in the member function definitions (see above) to assist you.
- Remember to use member–access specifiers **public** and **private** to specify the access level of data members and functions. Carefully consider which access specifier to use for each class member. In general, data members should be **private** and member functions should be **public**.

Questions

1. Why do you think **const** was used in the parameter list of **addition** and **subtraction**?
2. Can **addition** and **subtraction**'s parameters be passed call–by–value instead of call–by–reference? How might this affect the design of class **Complex**? Write a new class definition.