CMPE 180-92

Data Structures and Algorithms in C++

September 21 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017

Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Assignment #4 Sample Solution

```
#include <iostream>
#include <iomanip>
#include <mpir.h>
#include <stdlib.h>
#include <string.h>

using namespace std;

const int MAX_ITERATIONS = 100;
const int PLACES         = 1000;        // desired decimal places
const int PRECISION      = PLACES + 1;  // +1 for the digit 3 before the decimal

const int BASE       = 10;  // base 10 numbers
const int BIT_COUNT  = 8;   // bits per machine word

const int BLOCK_SIZE = 10;                     // print digits in blocks
const int LINE_SIZE  = 100;                    // digits to print per line
const int LINE_COUNT = PLACES/LINE_SIZE;       // lines to print
const int GROUP_SIZE = 5;                      // line grouping size
```

Assignment #4 Sample Solution

```
void cube_root(mpf_t& x, const mpf_t a)
{
    // Use Halley's method:
    // https://en.wikipedia.org/wiki/Cube_root

    // Multiple-precision variables
    mpf_t x_prev;  mpf_init(x_prev);
    mpf_t temp1;   mpf_init(temp1);
    mpf_t temp2;   mpf_init(temp2);
    mpf_t two_a;   mpf_init(two_a);
    mpf_t x_cubed; mpf_init(x_cubed);

    // Constant 3
    mpf_t three; mpf_init(three); mpf_set_str(three, "3", BASE);

    // Set an initial estimate for x.
    mpf_div(x, a, three);   // x = a/3

    int n = 0;  // iteration counter
```

Assignment #4 Sample Solution, *cont'd*

$$x_{n+1} = x_n \left(\frac{x_n^3 + 2a}{2x_n^3 + a} \right)$$

```
    // Loop until two consecutive values are equal
    // or up to MAX_ITERATIONS times.
    do
    {
        mpf_set(x_prev, x);

        mpf_mul(x_cubed, x, x);
        mpf_mul(x_cubed, x_cubed, x);       // x_cubed = x^3
        mpf_add(two_a, a, a);               // two_a = 2a
        mpf_add(temp1, x_cubed, two_a);     // temp1 = x^3 + 2a
        mpf_add(temp2, x_cubed, x_cubed);   // temp2 = 2x^3
        mpf_add(temp2, temp2, a);           // temp2 = 2x^3 + a
        mpf_div(temp1, temp1, temp2);       // temp1 = (x^3 + 2a)/(2x^3 + a)
        mpf_mul(x, x, temp1);               // x = x((x^3 + 2a)/(2x^3 + a))

        n++;
    } while ((mpf_cmp(x, x_prev) != 0) && (n < MAX_ITERATIONS));
}
```

Assignment #4 Sample Solution, *cont'd*

```
void compute_pi(mpf_t& pi)
{
    // Use a nonic algorithm:
    // https://en.wikipedia.org/wiki/Borwein's_algorithm

    // Multiple-precision constants.
    mpf_t one;          mpf_init_set_str(one,           "1", BASE);
    mpf_t two;          mpf_init_set_str(two,           "2", BASE);
    mpf_t three;        mpf_init_set_str(three,         "3", BASE);
    mpf_t nine;         mpf_init_set_str(nine,          "9", BASE);
    mpf_t twenty_seven; mpf_init_set_str(twenty_seven, "27", BASE);

    mpf_t one_third; mpf_init(one_third);
    mpf_div(one_third, one, three);
```

Assignment #4 Sample Solution, *cont'd*

```
// Multiple-precision variables
mpf_t a;          mpf_init(a);
mpf_t r;          mpf_init(r);
mpf_t s;          mpf_init(s);
mpf_t t;          mpf_init(t);
mpf_t u;          mpf_init(u);
mpf_t v;          mpf_init(v);
mpf_t w;          mpf_init(w);
mpf_t power3;     mpf_init(power3);
mpf_t prev_a;     mpf_init(prev_a);

// Temporaries
mpf_t temp1; mpf_init(temp1);
mpf_t temp2; mpf_init(temp2);
```

Assignment #4 Sample Solution, *cont'd*

Start by setting

$$a_0 = \frac{1}{3}$$
$$r_0 = \frac{\sqrt{3}-1}{2}$$
$$s_0 = (1 - r_0^3)^{1/3}$$

```
// Initialize a
mpf_set(a, one_third);        // a = 1/3

// Initialize r
mpf_sqrt(temp1, three);       // temp1 = sqrt(3)
mpf_sub(temp1, temp1, one);   // temp1 = sqrt(3) - 1
mpf_div(r, temp1, two);       // r = (sqrt(3) - 1)/2

// Initialize s
mpf_mul(temp1, r, r);
mpf_mul(temp1, temp1, r);     // temp1 = r^3
mpf_sub(temp1, one, temp1);   // temp1 = 1 - r^3
cube_root(s, temp1);          // s = cbrt(1 - r^3)

// Initialize power3
mpf_set(power3, one_third);
```

Assignment #4 Sample Solution, *cont'd*

```
// Loop until two consecutive values are equal
// or up to MAX_ITERATIONS times. Iterate at least twice.
int n = 0;
do
{
    // Save the previous a for later comparison.
    mpf_set(prev_a, a);                 // prev_a = a

    mpf_div(temp1, one, prev_a);

    // Compute t
    mpf_add(temp1, r, r);               // temp1 = 2r
    mpf_add(t, one, temp1);             // t = 1 + 2r

    // Compute u
    mpf_add(temp1, one, r);             // temp1 = 1 + r
    mpf_mul(temp2, r, r);               // temp2 = r^2
    mpf_add(temp1, temp1, temp2);       // temp1 = 1 + r +r^2
    mpf_mul(temp1, nine, temp1);
    mpf_mul(temp1, r, temp1);           // temp1 =  9r(1 + r + r^2)
    cube_root(u, temp1);                // u = cbrt(9r(1 + r + r^2))
```

Then iterate

$$
\begin{aligned}
t_{n+1} &= 1 + 2r_n \\
u_{n+1} &= (9r_n(1 + r_n + r_n^2))^{1/3} \\
v_{n+1} &= t_{n+1}^2 + t_{n+1}u_{n+1} + u_{n+1}^2 \\
w_{n+1} &= \frac{27(1 + s_n + s_n^2)}{v_{n+1}} \\
a_{n+1} &= w_{n+1}a_n + 3^{2n-1}(1 - w_{n+1}) \\
s_{n+1} &= \frac{(1 - r_n)^3}{(t_{n+1} + 2u_{n+1})v_{n+1}} \\
r_{n+1} &= (1 - s_{n+1}^3)^{1/3}
\end{aligned}
$$

Assignment #4 Sample Solution, *cont'd*

```
// Compute v
mpf_mul(temp1, t, t);             // temp1 = t^2
mpf_mul(temp2, t, u);             // temp2 = tu
mpf_add(temp1, temp1, temp2);     // temp1 = t^2 + tu
mpf_mul(temp2, u, u);             // temp2 = u^2
mpf_add(v, temp1, temp2);         // v = t^2 + tu + u^2

// Compute w
mpf_add(temp1, one, s);           // temp1 = 1 + s
mpf_mul(temp2, s, s);             // temp2 = s^2
mpf_add(temp1, temp1, temp2);     // temp1 = 1 + s + s^2
mpf_mul(temp1, temp1, twenty_seven);    // temp1 = 27(1 + s + s^2)
mpf_div(w, temp1, v);             // w = (27(1 + s + s^2))/v

// Compute next a
mpf_mul(temp1, w, a);             // temp1 = wa
mpf_sub(temp2, one, w);           // temp2 = 1 - w
mpf_mul(temp2, power3, temp2);    // temp2 =  (3^(2n-1))(1 - w)
mpf_add(a, temp1, temp2);         // a = wa + (3^(2n-1))(1 - w)
```

Then iterate

$$
\begin{aligned}
t_{n+1} &= 1 + 2r_n \\
u_{n+1} &= (9r_n(1 + r_n + r_n^2))^{1/3} \\
v_{n+1} &= t_{n+1}^2 + t_{n+1}u_{n+1} + u_{n+1}^2 \\
w_{n+1} &= \frac{27(1 + s_n + s_n^2)}{v_{n+1}} \\
a_{n+1} &= w_{n+1}a_n + 3^{2n-1}(1 - w_{n+1}) \\
s_{n+1} &= \frac{(1 - r_n)^3}{(t_{n+1} + 2u_{n+1})v_{n+1}} \\
r_{n+1} &= (1 - s_{n+1}^3)^{1/3}
\end{aligned}
$$

Assignment #4 Sample Solution, *cont'd*

```
// Compute next s
mpf_sub(temp2, one, r);           // temp2 = 1 - r
mpf_mul(temp1, temp2, temp2);
mpf_mul(temp1, temp1, temp2);     // temp1 = (1 - r)^3
mpf_add(temp2, t, u);
mpf_add(temp2, temp2, u);         // temp2 = t + 2u
mpf_mul(temp2, temp2, v);         // temp2 = (t + 2u)v
mpf_div(s, temp1, temp2);         // s = ((1 - r)^3)/((t + 2u)v)

// Compute next r
mpf_mul(temp1, s, s);
mpf_mul(temp1, temp1, s);         // temp1 = s^3
mpf_sub(temp1, one, temp1);       // temp1 = 1 - s^3
cube_root(r, temp1);              // r = (1 - s^3)^(1/3)
```

Then iterate

$$
\begin{aligned}
t_{n+1} &= 1 + 2r_n \\
u_{n+1} &= (9r_n(1 + r_n + r_n^2))^{1/3} \\
v_{n+1} &= t_{n+1}^2 + t_{n+1}u_{n+1} + u_{n+1}^2 \\
w_{n+1} &= \frac{27(1 + s_n + s_n^2)}{v_{n+1}} \\
a_{n+1} &= w_{n+1}a_n + 3^{2n-1}(1 - w_{n+1}) \\
s_{n+1} &= \frac{(1 - r_n)^3}{(t_{n+1} + 2u_{n+1})v_{n+1}} \\
r_{n+1} &= (1 - s_{n+1}^3)^{1/3}
\end{aligned}
$$

Assignment #4 Sample Solution, *cont'd*

```
        // Compute next power of 3
        mpf_mul(power3, power3, nine);  // power3 = 3^(2n-1)

        n++;
    } while (   ((n < 2) || (mpf_eq(a, prev_a, PRECISION) == 0))
             && (n < MAX_ITERATIONS));

    // Compute pi = 1/a
    mpf_div(pi, one, a);
}
```

Then iterate

$$
\begin{aligned}
t_{n+1} &= 1 + 2r_n \\
u_{n+1} &= (9r_n(1 + r_n + r_n^2))^{1/3} \\
v_{n+1} &= t_{n+1}^2 + t_{n+1}u_{n+1} + u_{n+1}^2 \\
w_{n+1} &= \frac{27(1 + s_n + s_n^2)}{v_{n+1}} \\
a_{n+1} &= w_{n+1}a_n + 3^{2n-1}(1 - w_{n+1}) \\
s_{n+1} &= \frac{(1 - r_n)^3}{(t_{n+1} + 2u_{n+1})v_{n+1}} \\
r_{n+1} &= (1 - s_{n+1}^3)^{1/3}
\end{aligned}
$$

Assignment #4 Sample Solution, *cont'd*

```
/**
 * Print the decimal places of a multiple-precision number x.
 * @param pi the multiple-precision number to print.
 */
void print(const mpf_t& pi)
{
    mp_exp_t exp;  // exponent (not used)

    // Convert the multiple-precision number x to a C string.
    char *str = NULL;
    char *s = mpf_get_str(str, &exp, BASE, PRECISION, pi);
    char *p = s+1;  // skip the 3 before the decimal point

    cout << endl;
    cout << "3.";

    char block[BLOCK_SIZE + 1];  // 1 extra for the ending \0
```

Assignment #4 Sample Solution, *cont'd*

```
    // Loop for each line.
    for (int i = 1; i <= LINE_COUNT; i++)
    {
        // Loop to print blocks of digits in each line.
        for (int j = 0; j < LINE_SIZE; j += BLOCK_SIZE)
        {
            strncpy(block, p+j, BLOCK_SIZE);
            block[BLOCK_SIZE] = '\0';
            cout << block << " ";
        }

        cout << endl << "  ";

        // Print a blank line for grouping.
        if (i%GROUP_SIZE == 0) cout << endl << "  ";

        p += LINE_SIZE;
    }

    free(s);
}
```

Structures

- A structure represents a collection of values that can be of different data types.

- We want to treat the collection as a single item.
  - Example:

structure tag

```
struct Employee
{
    int id;
    string first_name;
    string last_name;
    double salary;
};
```

members

Structures are Types

```
struct Employee
{
    int id;
    string first_name;
    string last_name;
    double salary;
};
```

- A structure is a type:

```
Employee mary, john;

mary.id = 12345;
mary.first_name = "Mary";
mary.last_name  = "Poppins";
mary.salary = 150000.25;

mary.salary = 1.10*mary.salary;
```

john

id	98765
first_name	“John”
last_name	“Johnson”
salary	75000.00

mary

id	12345
first_name	“Mary”
last_name	“Poppins”
salary	150000.25

Scope of Structure Member Names

- Two different structure types can contain members with the same name:

```
struct Employee
{
    int id;
    ...
};
```

```
struct Student
{
    int id;
    ...
};
```

- To access the value of one of the structure's members, use a member variable such as `mary.salary`

Structure Variables

- If you have two variables of the same structure type, you can assign one to the other:

```
john = mary;
```

- This is equivalent to:

```
john.id         = mary.id;
john.first_name = mary.first_name;
john.last_name  = mary.last_name;
john.salary     = mary.salary;
```

Structure Variables, *cont'd*

- An array of employees:

```
Employee team[10];

team[4].id = 39710;
team[4].first_name = "Sally";
```

- Pass a structure variable to a function:

```
void foo(Employee emp1, Employee& emp2);
```

- Return a structure value:

```
Employee find_employee(int id);
```

Structure Variables, *cont'd*

- Pointer to a structure:

```
Employee *emp_ptr;

emp_ptr = new Employee;
(*emp_ptr).id = 192837;
emp_ptr->salary = 95000.00;
```

- Nested structures:

```
struct Employee
{
    int id;
    string first_name;
    string last_name;
    double salary;
    Birthday bday;
};
```

```
struct Birthday
{
    int month, day, year;
};
```

```
Employee tom;
tom.bday.year = 1992;
```

Break

Object-Oriented Programming

- Object-oriented programming (OOP) is about
  - encapsulation
  - inheritance
  - polymorphism

Combine variables and functions into a single class.

- Work with values called objects.
  - Objects have member functions that operate on the objects.
  - Example: A string is an object. Strings have a length method, so that if `str` is a string variable, then `str.length()` is the length of its string value.

Classes

- A class is a data type whose values are objects.
  - Like structure types, you can define your own class types.

- A class type definition includes both member variables and declarations of member functions.
  - Example:

```
class Birthday
{
public:
     int month, day, year;
     void print();
};
```

Defining Member Functions

- Define member functions <u>outside</u> of the class definition:

```
class Birthday
{
public:
    int month, day, year;
    void print();
};

void Birthday::print()
{
    cout << month << "/" << day << "/" << year << endl;
}
```

- Scope resolution operator `::`

Public and Private Members

- Members of a class are either public or private.
- Private members of a class can be accessed only by member functions of the same class.
- You can provide public getters and setters for any private member variables.
  - AKA accessors and mutators
- A member function (public or private) can be labelled `const`.
  - It will <u>not modify</u> the value of any member variable.

Public and Private Members, *cont'd*

Birthday1.cpp

```
class Birthday
{
public:
    void set_year(int y);
    void set_month(int m);
    void set_day(int d);

    int get_year()  const;
    int get_month() const;
    int get_day()   const;
    void print()    const;

private:
    int year, month, day;
};
```

Public and Private Members, *cont'd*

Birthday1.cpp

```
int Birthday::get_year()  const { return year; }
int Birthday::get_month() const { return month; }
int Birthday::get_day()   const { return day; }

void Birthday::set_year(int y)  { year = y; }
void Birthday::set_month(int m) { month = m; }
void Birthday::set_day(int d)   { day = d; }

void Birthday::print() const
{
    cout << month << "/" << day << "/" << year << endl;
}
```

Public and Private Members, *cont'd*

Birthday1.cpp

```
int main()
{
    Birthday bd;
    bd.set_year(1990);
    bd.set_month(9);
    bd.set_day(2);
    bd.print();
}
```

```
9/2/1990
```

Constructors

- A class can define special member functions called constructors that initialize the values of member variables.
- A constructor has the same name as the class itself.
  - It has no return type, not even void.
  - The default constructor has no parameters.
- A constructor is called automatically whenever an object of the class is declared.

Constructors, *cont'd*

Birthday1.cpp

```
class Birthday
{
public:
    // Constructors
    Birthday();
    Birthday(int y, int m, int d);

    ...
}

Birthday::Birthday() : year(0), month(0), day(0)
{
    // Default constructor with an empty body
}

Birthday::Birthday(int y, int m, int d) : year(y), month(m), day(d)
{
    // Empty body
}
```

Constructors, *cont'd*

Birthday1.cpp

```
int main()
{
    Birthday bd1;                // call default constructor
    Birthday bd2(2000, 9, 2);   // call constructor

    bd1.print();
    bd2.print();
}
```

```
0/0/0
9/2/2000
```

- Do not write: `Birthday bd1();`
  - That is a declaration of a function named **`bd1`** that returns a value of type **`Birthday`**.

Constructors, *cont'd*

- If you provided <u>no</u> constructors for a class, the C++ compiler will generate a <u>default constructor</u> that does nothing.

- However, if you provided <u>at least one</u> constructor for the class, the compiler will <u>not</u> generate a default constructor.

Constructors, *cont'd*

- Suppose you are provided this constructor only:

```
Birthday(int y, int m, int d);
```

- Then the following object declaration is illegal:

```
Birthday bd1;
```

Friend Functions

```
class Birthday
{
public:
    // Constructors
    Birthday();
    Birthday(int y, int m, int d);

    int get_year()  const;
    int get_month() const;
    int get_day()   const;

    void set_year(int y);
    void set_month(int m);
    void set_day(int d);

    void print() const;

private:
    int year, month, day;
};
```

- Write a function that is external to the class (i.e., not a member function) that compares two birthdays for equality.

Friend Functions, *cont'd*

Birthday1.cpp

```
bool equal(const Birthday& bd1, const Birthday& bd2)
{
    return     (bd1.get_year()  == bd2.get_year())
            && (bd1.get_month() == bd2.get_month())
            && (bd1.get_day()   == bd2.get_day());
}
```

- Function **`equal`** must call the accessor (getter) methods because **`year`**, **`month`**, and **`day`** are private member variables.

- Make function **`equal`** a friend of class **`Birthday`** to allow the function to access the private member variables directly.

Friend Functions, *cont'd*

```
class Birthday
{
public:
    // Constructors
    Birthday();
    Birthday(int y, int m, int d);

    int get_year()  const;
    int get_month() const;
    int get_day()   const;

    void set_year(int y);
    void set_month(int m);
    void set_day(int d);

    void print() const;

    friend bool equal(const Birthday& bd1, const Birthday& bd2);

private:
    int year, month, day;
};
```

Birthday2.cpp

Because it is a friend of the class, function **equal** can now access private members.

```
bool equal(const Birthday& bd1,
           const Birthday& bd2)
{
    return    (bd1.year  == bd2.year)
           && (bd1.month == bd2.month)
           && (bd1.day   == bd2.day);
}
```

Have both friend functions and accessor functions.

Operator Overloading

- How many years apart are two birthdays?
- We can write a function **`years_apart`** that takes two birthdays and subtracts their years:

Birthday2.cpp

```
class Birthday
{
public:
    ...
    friend bool equal(const Birthday& bd1, const Birthday& bd2);
    friend int years_apart(const Birthday& bd1, const Birthday& bd2);
    ...
};
```

```
int years_apart(const Birthday& bd1, const Birthday& bd2)
{
    return abs(bd1.year - bd2.year);
}
```

Operator Overloading, *cont'd*

- Overload the subtraction operator and make `operator -` a friend function.

Birthday2.cpp

```
class Birthday
{
public:
    ...
    friend bool equal(const Birthday& bd1, const Birthday& bd2);
    friend int years_apart(const Birthday& bd1, const Birthday& bd2);
    friend int operator -(const Birthday& bd1, const Birthday& bd2);
    ...
};
```

```
int operator -(const Birthday& bd1, const Birthday& bd2)
{
    return abs(bd1.year - bd2.year);
}
```

Operator Overloading, *cont'd*

Birthday2.cpp

```
int main()
{
    Birthday bd1;                  // call default constructor
    Birthday bd2(1990, 9, 2);  // call constructor
    Birthday bd3(2001, 5, 8);  // call constructor

    bd1.print();
    bd2.print();

    cout << years_apart(bd2, bd3) << endl;
    cout << bd2 - bd3 << endl;
}
```

```
0/0/0
9/2/1990
11
11
```

Overload <<

- You can overload the stream insertion operator.
- Suppose you want a `Birthday` object to be output in the form month/day/year.

Birthday2.cpp

```
class Birthday
{
public:
    ...
    friend ostream& operator <<(ostream& outs, const Birthday& bd);
    ...
};
```

```
friend ostream& operator <<(ostream& outs, const Birthday& bd)
{
    outs << bd.month << "/" << bd.day << "/" << bd.year << endl;
    return outs;
}
```

Overload **<<**, *cont'd*

Birthday2.cpp

```
int main()
{
    Birthday bd1;                 // call default constructor
    Birthday bd2(1990, 9, 2);    // call constructor
    Birthday bd3(2001, 5, 8);    // call constructor

    cout << bd1 << ", " << bd2 << ", " << bd3 << endl;
}
```

```
0/0/0, 9/2/1990, 5/8/2001
```

Overload >>

- You want to input birthdays in the format

  {*year*, *month*, *day*}

  - Example: `{1993, 9, 2}`

- Overload the stream extraction operator.

Birthday2.cpp

```
class Birthday
{
public:
    ...
    friend istream& operator >>(istream& ins, Birthday& bd);
    ...
};
```

Overload >>, *cont'd*

```
istream& operator >>(istream& ins, Birthday& bd)
{
    int y, m, d;
    char ch;

    ins >> ch;
    if (ch == '{')
    {
        ins >> y;

        ins >> ch;
        if (ch == ',')
        {
            ins >> m;

            ins >> ch;
            if (ch == ',')
            {
                ins >> d;

                ins >> ch;
                if (ch == '}')
                {
                    bd.year  = y;
                    bd.month = m;
                    bd.day   = d;
                }
            }
        }
    }

    return ins;
}
```

Error checking needed!

Birthday2.cpp

```
int main()
{
    Birthday bd1;
    Birthday bd2;

    cout << "Enter two birthdays: ";
    cin >> bd1 >> bd2;
    cout << bd1 << ", " << bd2 << endl;
}
```

```
Enter two birthdays: {1953, 9, 2}   {1957, 4, 3}
9/2/1953, 4/3/1957
```

Abstract Data Types

- A data type specifies:
  - what values are allowed
  - what operations are allowed
- An abstract data type (ADT):
  - allows its values and operations to be used
  - <u>hides the implementation</u> of values and operations
- Example: The predefined type `int` is an ADT.
  - You can use integers and the operators `+ - * / %`
  - But you don't know how they're implemented.

Abstract Data Types, *cont'd*

- To make your class an ADT, you must separate:
  - The specification of how a type is <u>used</u>.
  - The details of how the type is <u>implemented</u>.
- To ensure this separation:
  - Make all member variables private.
  - Make public all the member functions that a programmer needs to use, and fully specify how to use each one.
  - Make private all helper member functions.

Is the **`Birthday`** class an ADT?

Separate Compilation

- Put each class declaration in a separate .h header file.
  - By convention, name the file after the class name.
  - Any other source file that uses the class would **`#include`** the class header file.
- Put the implementations of the member functions into a .cpp file.
  - By convention, name the file after the class name.
- A class header file is the interface that the class presents to users of the class.

Separate Compilation, *cont'd*

Birthday.h

```
#ifndef BIRTHDAY_H_
#define BIRTHDAY_H_

using namespace std;

class Birthday
{
public:
    // Constructors
    Birthday();
    Birthday(int y, int m, int d);

    // Destructor
    ~Birthday();

    int get_year() const;
    int get_month() const;
    int get_day() const;

    void set_year(int y);
    void set_month(int m);
    void set_day(int d);

    void print();

    friend bool equal(const Birthday& bd1, const Birthday& bd2);
    friend int years_apart(const Birthday& bd1, const Birthday& bd2);
    friend int operator -(const Birthday& bd1, const Birthday& bd2);
    friend ostream& operator <<(ostream& outs, const Birthday& bd);
    friend istream& operator >>(istream& ins, Birthday& bd);

private:
    int year, month, day;
};

#endif
```

Separate Compilation, *cont'd*

Birthday3.cpp

```
#include <iostream>
#include <cstdlib>
#include "Birthday.h"

using namespace std;

Birthday::Birthday() : year(0), month(0), day(0)
{
    // Default constructor with an empty body
}

Birthday::Birthday(int y, int m, int d) : year(y), month(m), day(d)
{
    // Empty body
}

Birthday::~Birthday()
{
    // Empty body
}

int Birthday::get_year()  const { return year; }
int Birthday::get_month() const { return month; }
int Birthday::get_day()   const { return day; }

void Birthday::set_year(int y)  { year = y; }
void Birthday::set_month(int m) { month = m; }
void Birthday::set_day(int d)   { day = d; }
```

Separate Compilation, *cont'd*

Birthday.cpp

```
void Birthday::print()
{
    cout << month << "/" << day << "/" << year << endl;
}

int operator -(const Birthday& bd1, const Birthday& bd2)
{
    return abs(bd1.year - bd2.year);
}

ostream& operator <<(ostream& outs, const Birthday& bd)
{
    outs << bd.month << "/" << bd.day << "/" << bd.year;
    return outs;
}

istream& operator >>(istream& ins, Birthday& bd)
{
    ...
}
```

Separate Compilation, *cont'd*

BirthdayTester.cpp

```
#include <iostream>
#include "Birthday.h"

int main()
{
    Birthday bd1;                  // call default constructor
    Birthday bd2(1990, 9, 2);    // call constructor
    Birthday bd3(2001, 5, 8);    // call constructor

    bd1.print();
    bd2.print();

    cout << bd2 - bd3 << endl;
    cout << bd1 << ", " << bd2 << ", " << bd3 << endl;

    cout << endl;
    cout << "Enter two birthdays: ";
    cin >> bd1 >> bd2;
    cout << bd1 << ", " << bd2 << endl;
}
```

Assignment #5. Roman Numerals

- Define a C++ class **`RomanNumeral`** that implements arithmetic operations with Roman numerals, and reading and writing Roman numerals.
  - See https://en.wikipedia.org/wiki/Roman_numerals

- Private member variables **`string roman`** and **`int decimal`** store the Roman numeral string (such as **`"MCMLXVIII"`**) and its integer value (1968).

Assignment #5. Roman Numerals, *cont'd*

- Private member functions `to_roman` and `to_decimal` convert between the string and integer values of a `RomanNumeral` object.

- One constructor has an integer parameter, and another constructor has a string parameter.
  - Construct a Roman numeral object by giving either its integer or string value.

- Public getter functions return the object's string and integer values.

Assignment #5. Roman Numerals, *cont'd*

- Override the arithmetic operators `+ - * /`
  - Roman numerals perform integer division.

- Override the equality operators `== !=`

- Override the stream operators `>>` and `<<`
  - Input a Roman numeral value as a string, such as **`MCMLXVIII`**
  - Output a Roman numeral value in the form

    `[`*integer value* **`:`** *roman string***`]`**

    such as **`[1968:MCMLXVIII]`**

Assignment #5. Roman Numerals, *cont'd*

- A test program inputs and parses a text file containing simple two-operand arithmetic expressions with Roman numerals:

```
MCMLXIII + LIII
MMI - XXXIII
LIII * XXXIII
MMI / XXXIII
```

- It performs the arithmetic and output the results:

```
[1963:MCMLXIII] + [53:LIII] = [2016:MMXVI]
[2001:MMI] - [33:XXXIII] = [1968:MCMLXVIII]
[53:LIII] * [33:XXXIII] = [1749:MDCCXLIX]
[2001:MMI] / [33:XXXIII] = [60:LX]
```

Assignment #5. Roman Numerals, *cont'd*

- File RomanNumeral.h contains the class declaration.
- File RomanNumeral.cpp contains the class implementation.
- File RomanNumeralTester.cpp contains two functions to test the class.