CISS245: Advanced Programming
Assignment 8

Objectives

- Write constructors.
- Use default values for parameters.
- Use initializer list for constructor.
- Write get and set methods.
- Overloaded operators.
- Pass objects to methods by reference.
- Pass objects to methods by constant reference.
- Write constant methods.
- Use `this` pointer to return a copy of an object.
- Write functions with object parameters.

Q1. You are given the following (possibly incomplete files):

- `vec2d.h`
- `vec2d.cpp`
- `main.cpp` (the test code)

Important Warning: Again, the files are meant to be skeleton file and might not be complete and might have deliberate missing details or even errors.

Create directory `ciss245/a/a08/a08q01`. Keep all your files in this directory.

If you're doing a copy-and-paste of the given code, note that some character might be changed by PDF to other characters. In particular the - character might actually not be the dash character. Looking at the compiler error message will help you find these minor annoying issues so that you can correct them.

Study the given test code. Add tests if necessary to test all methods and functions. Such low level function/method tests are called **unit tests**.

Observe the following very carefully:

- All methods must be constant whenever possible.
- All parameters which are objects (or struct variables) must be pass by reference or pass by constant reference as much as possible.
- Reuse code as much as possible. For instance `operator!=()` should use `operator==()`.

Let me know ASAP if you see a typo.

Q1. Vectors

Here's a quick note about vectors. A 2-dimensional vector is sort of like a point. For instance you know (from MATH106 elementary algebra) that the following is a point:

$$(1.1, 3.5)$$

Here's another point:

$$(0, 0)$$

A 2-dimensional vector is similar but they are frequently written like this:

$$\langle 1.1, 3.5 \rangle$$

The 1.1 is called the x-component of the above vector and 3.5 is the y-component of the vector. You can add vectors. Here's an example:

$$\langle 1.1,\ \ 3.5 \rangle + \langle 2.2,\ \ 4.1 \rangle = \langle 1.1 + 2.2,\ \ 3.5 + 4.1 \rangle = \langle 3.3,\ \ 7.6 \rangle$$

In other words you just add their corresponding components to get a new vector. In general

$$\langle a,\ \ b \rangle + \langle c,\ \ d \rangle = \langle a + c,\ \ b + d \rangle$$

You can multiply vector with a `double`:

$$2.0 \cdot \langle 1.1,\ \ 3.5 \rangle = \langle 2 \cdot 1.1,\ \ 2 \cdot 3.5 \rangle = \langle 2.2, 7.0 \rangle$$

In general

$$c \cdot \langle a,\ \ b \rangle = \langle ca,\ \ cb \rangle$$

Of course you can multiply the `double` on the right of the vector:

$$\langle 1.1, 3.5 \rangle \cdot 2.0 = \langle 2.2, 7.0 \rangle$$

This product is called the scalar product. Not too surprising, you can divide vectors with a `double`:

$$\langle 1.1, 3.5 \rangle / 2.0 = \langle 1.1/2.0, 3.5/2.0 \rangle = \langle 0.55, 1.75 \rangle$$

In general

$$\cdot \langle a,\ \ b \rangle / c = \langle a/c,\ \ b/c \rangle$$

There's another product for vectors called the dot product. Mathematically, it is written like this:

$$\langle 1.1,\ \ 3.5 \rangle \cdot \langle 2.2,\ \ 4.1 \rangle$$

This is defined to be the sum of the product of the components, i.e.,

$$\langle 1.1,\ \ 3.5\rangle \cdot \langle 2.2,\ \ 4.1\rangle = (1.1)(2.2) + (3.5)(4.1) = 2.42 + 14.35 = 16.77$$

In general

$$\langle a,\ \ b\rangle \cdot \langle c,\ \ d\rangle = ac + bd$$

Of course the dot product is a real number and not a vector.

The length of a vector $\langle x, y\rangle$ is defined by Pythagorus formula:

$$\text{len}(\langle x,\ \ y\rangle) = \sqrt{x^2 + y^2}$$

The normalization of a vector is the vector divided by its length. Therefore the normalization of $\langle 1.1, 3.5\rangle$ is

$$\begin{aligned}
\langle 1.1,\ \ 3.5\rangle/\sqrt{1.1^2 + 3.5^2} &= \langle 1.1,\ \ 3.5\rangle/\sqrt{13.46} \\
&= \langle 1.1,\ \ 3.5\rangle/3.6687\cdots \\
&= \langle 1.1/3.6687\ldots,\ \ 3.5/3.6687\ldots\rangle \\
&= \langle 1.1/3.6687\ldots,\ \ 3.5/3.6687\ldots\rangle \\
&= \langle 0.2998\ldots,\ \ 0.9539\ldots\rangle
\end{aligned}$$

So what are vectors good for? While points are just points in space, vectors represent motion. The vector

$$\langle 1.1, -3.5\rangle$$

represents the fact that something is moving in the x–axis direction by 1.1 and in the y–axis direction by -3.5. Vectors are used extensive in modeling motion in the real world. Therefore it is extremely important in physics and therefore extremely important in engineering and computer science. For instance, in scientific computation/simulation and computer games, frequently, you're modeling some physical entities in the real world. You need vectors to describe their motion. To simulate water waves, you need vectors.

(Yes, the above is really generated by a program – it's not a photo.) Vectors are really everywhere.

The purpose of this assignment is to build a vector library. This will only be for 2-dimensional vectors and therefore will be useful for 2D games. But once you have done 2-dimensional vectors, it's not too difficult to build a library for 3-dimentional vectors or even for n-dimensional vectors. You do not need to know how to use vectors to solve problems and you do not know about fluid dynamics for simulating sea waves. You only need to know how to interpret basic numeric computations and the language of C++ object-oriented programming.

vec2d header file

Note that the header file contains both methods and functions. The prototypes for the functions are given. All others must be methods in the class.

```
/*****************************************************************************

File  : vec2d.h
Author: smaug
Date  : 01/01/2017

Description
This is the header file of the vec2d class.

Each vec2d object models a 2-dimensional vector with double coordinates. The
methods and operators supported are described below. In the following, u, v,
and w are vec2d objects while c, d are doubles.

vec2d u(x,y)    - Constructor. This sets the x and y values of u to values of
                  parameters x and y respectively. The default values are 0,0
                  respectively.

std::cout << u - Prints vector u. The format is "<x, y>" where x and y
                  are the x and y values of u.
std::cin >> u  - Input for u.

u.get_x()       - Returns the x_ value of vec2d object u.
u.get_y()       - Returns the y_ value of vec2d object u.
u.set_x(x0)     - Sets the x_ value of u to the value of x0.
u.set_y(y0)     - Sets the y_ value of u to the value of y0
u[i]            - Returns the x_ value or a reference to the x_ value of u
                  if i is 0; otherwise the y_ value or reference to the y_
                  value is returned.

u == v          - Returns true iff the respective x_,y_ values of u and v are
                  the same.
u != v          - The boolean opposite of (u == v)
u = v           - Copies the x,y values of v to u.

+u              - Returns a copy of u.
u += v          - Increments the x_ and y_ values of u by the x and y values
                  of v respectively. A reference to u is returned so that
                  you can also execute
                  (u += v) += w.
u + v           - Returns the vector sum of u and v.
```

```
-u              - Return the negative of u, i.e., the vec2d object returned
                  has x_,y_ values which are the negative of the x_,y_ values of
                  u.
u -= v          - Decrements the x_ and y_ values of u by the x_ and y_ values
                  of v. A reference to u is returned so that you can also
                  execute (u -= v) -= w).
u - v            - Returns the vector difference of u and v.

u *= c            - Multiplies the x_,y_ values of u by double c and set the
                  x_,y_ values to these new values (respectively). A
                  reference to u is returned so that you can also execute
                  (u *= c) *= d.
u * c            - Returns the vector scalar product of vector u by double
                  c, i.e., a vec2d object is returned where the x_,y_ values
                  of this new vec2d object are the x_,y_ values of u
                  multiplied by c.
c * u           - Similar to u * c.

u /= c          - Divides the x,y values of u by double c and set the x_,y_
                  values to these new values (respectively). A reference
                  to u is returned so that you can also execute
                  (u /= c) /= d.
u / c           - Returns the vector scalar product of vector u by the double
                  (1/c).

len(u)          - Returns the length of vector u as a double. This is the
                  square root of x_*x_ + y_*y_ where x_,y_ are the values
                  in u. Note that this is a non-member function.
u.len()         - Same as len(u) except that this is a method.

dotprod(u, v)   - Returns the dot product of vectors u and v as a double.
norm(u)         - Return the normalized vector of u, i.e., this function
                  returns u / len(u), a vector of length 1 going in the
                  same direction as u.

*********************************************************************************/
#ifndef VEC2D_H
#define VEC2D_H

#include <iostream>

class vec2d
{
public:
```

```
    vec2d(double x = 0, double y = 0)
        : x_(x), y_(y)
    {}

    bool operator==(const vec2d &) const;
    bool operator!=(const vec2d &) const;

    double get_x() const;
    double get_y() const;
    void set_x(double);
    void set_y(double);

    double operator[](int) const;
    double & operator[](int);

    vec2d operator+() const;
    vec2d & operator+=(const vec2d &);
    vec2d operator+(const vec2d &) const;

    vec2d operator-() const;
    vec2d & operator-=(const vec2d &);
    vec2d operator-(const vec2d &) const;

    vec2d & operator*=(double);
    vec2d operator*(double) const;

    vec2d & operator/=(double);
    vec2d operator/(double) const;

    double len() const;

private:
    double x_, y_;
};

vec2d operator*(double, const vec2d &);
double len(const vec2d &);
double dotprod(const vec2d &, const vec2d &);
vec2d norm(const vec2d &);
std::ostream & operator<<(std::ostream &, const vec2d &);
std::istream & operator>>(std::istream &, vec2d &);

#endif
```

vec2d Implementation File

Warning: Note that the functions in the header file are outside the class, Therefore they do not have direct access to the private instance variables `x_` and `y_` of the object.

```
/*****************************************************************

File  : vec2d.cpp
Author:
Date  :

Description
This is the implementation file the vec2d class. Refer to vec2d.h
for the interface.

*****************************************************************/

#include <iostream> // you need this because of std::ostream, std::istream
#include "vec2d.h"

// Prints the vec2d v in the following format: If v is vec2d(1.1, 2.2),
// then The output is <1.1, 2.2>.
std::ostream & operator<<(std::ostream & cout, const vec2d & v)
{
}

std::istream & operator>>(std::istream & cin, vec2d & v)
{
}
```

Test File

The following is only a skeleton test code. Complete it and test your library thoroughly.

```
/*****************************************************************

File  : main.cpp
Author:
Date  :

Description
This is the test program for the vec2d class.

*****************************************************************/

#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include "vec2d.h"


void test_vec2d_double_double()
{
    double x, y;
    std::cin >> x >> y;
    std::cout << vec2d(x, y) << std::endl;
}


void test_eq()
{
    vec2d u, v;
    std::cin >> u >> v;
    std::cout << (u == v) << std::endl;
}


void test_pluseq()
{
    vec2d u, v;
    std::cin >> u >> v;
    std::cout << (u += v) << ' ';
    std::cout << u << ' ';
    std::cout << ((u += v) += v) << ' '
```

```
    std::cout << u << std::endl;
}


void test_plus()
{
    vec2d u, v;
    std::cin >> u >> v;
    std::cout << (u + v) << std::endl;
}


void test_bracket_const()
{
    vec2d u;
    std::cin >> u;
    std::cout << u[0] << ' ' << u[1] << std::endl;
}


void test_bracket()
{
    vec2d u;
    double x = 0, y = 0;
    std::cin >> x >> y;
    u[0] = x;
    u[1] = y;
    std::cout << u << std::endl;
}


void test_mult()
{
    vec2d u;
    double c = 0;
    std::cin >> u >> c;
    std::cout << u * c << std::endl;
}


void test_double_mult()
{
    vec2d u;
    double c = 0;
    std::cin >> u >> c;
```

```
    std::cout << c * u << std::endl;
}


int main()
{
    int option;
    std::cin >> option;
    switch (option)
    {
        case 1:
            test_vec2d_double_double();
            break;
        case 11:
            test_bracket_const();
            break;
        case 12:
            test_bracket();
            break;
        case 20:
            test_mult();
            break;
        case 24:
            test_double_mult();
            break;
    }

    return 0;
}
```

Here are the test option numbering. First here are the input and output operators (i.e., `operator<<` and `operator>>`) and class methods (i.e. member functions and operators) in the class:

1. Constructor `vec2d::vec2d(double, double)` and printing
2. There's no case 2.
3. Constructor `vec2d::vec2d()` (the default) and print
4. Input (i.e. `operator>>`)
5. `bool vec2d::operator==(const vec2d &) const`
6. `bool vec2d::operator!=(const vec2d &) const`
7. `double vec2d::get_x() const`
8. `double vec2d::get_y() const`
9. `void vec2d::set_x(double)`
10. `void vec2d::set_y(double)`

11. `double vec2d::operator[](int) const`
12. `double & vec2d::operator[](int)`
13. `vec2d vec2d::operator+() const`
14. `vec2d & vec2d::operator+=(const vec2d &)`
15. `vec2d vec2d::operator+(const vec2d &) const`
16. `vec2d vec2d::operator-() const`
17. `vec2d & vec2d::operator-=(const vec2d &)`
18. `vec2d vec2d::operator-(const vec2d &) const`
19. `vec2d & vec2d::operator*=(double)`
20. `vec2d vec2d::operator*(double) const`
21. `vec2d & vec2d::operator/=(double)`
22. `vec2d vec2d::operator/(double) const`
23. `double len() const`

And here are the test options for non-member functions:

24. `vec2d operator*(double, const vec2d &)`
25. `double len(const vec2d &)`
26. `double dotprod(const vec2d &, const vec2d &)`
27. `vec2d norm(const vec2d &)`

Notes

Be efficient when you code. For instance `operator=!` is the opposite of `operator==`. So get `operator==` to work correctly and then code `operator!=` using `operator==`.

Likewise the binary `operator+` should be coded in terms of `operator+=`, etc. It's really a very simple idea. Here's a hint for you. Suppose you look at integers. How do you define `+` in terms of `+=`? Suppose `x`, `y`, `z` are integers and you want to do the following:

```
int x, y = 2, z = 3;
x = y + z;
```

Well, `y + z` is really the value of `y` incremented by `z`. Right? So the above statement can be written as:

```
int x, y = 2, z = 3;
x = y;
x += z;
```

Notice now that `+` does not appear in the two statements, only `=` and `+=`. On the other hand, it's easy to see that you can also write `+=` using `+`. However it's generally expected that `+=` should be faster than `+`. That means that `+` should be implemented in terms of `+=` and not the other way round.

You know if an argument of a function is not changed, you should declare it constant. For instance void `f(const vec2d & a)` would imply that the body/implementation of the function `f` cannot change the value(s) of `a`. However for a class, the object invoking the method does not appear in the parameter list:

```
  class C
{
public:
    void f();
    ...
};
```

So what if you have

```
int main()
{
    C obj;
    obj.f();
}
```

and what if you want to insist that `f()` does not change `obj`? You can make the object invoking the call constant within `f()`. Here's how you do it:

```
class C
{
public:
    void f() const;
    ...
};
```

You've just made method `f()` **constant**. We say that f() is a **constant method**.

Note that so far the object invoking a method does not have a "name" since it does not appear as a parameter in the method's prototype. So what if we need to return the object itself? For instance what if I want to write a `clone()` method that returns a copy of the object:

```
// Robot.h
#ifndef ROBOT_H
#define

class Robot
{
public:
    ...
    Robot clone() const;
    ...
};

#endif
```

Actually there *is* a parameter for the object invoking the call to `clone()`. It's called `this` in the implementation of the `clone()` method. However it's not a `Robot` object. It's a pointer to the `Robot` object invoking the call to `clone()`.

```
// Robot.cpp
...
Robot Robot::clone() const
{
    return *this; // need to dereference the "this" pointer
}
...
```

If `u` and `v` are `vec2d` objects and you have a method `operator+=`:

```
class vec2d
{
public:
    ...
    vec2d & operator+=(const vec2d &);
};
```

Then the statement

```
u += v;
```

is "translated" to

```
u.operator+=(v);
```

Suppose `u`, `v`, `w` are `vec2d` objects. The statement:

```
w = u + v;
```

can be "translated" in two different ways:

```
w = operator+(u, v);
```

or

```
w = u.operator+(v);
```

which will call either the *nonmember function*

```
vec2d operator+(const vec2d &, const vec2d &);
```

or the *method* (in the `vec2d` class)

```
vec2d vec2d::operator+(const vec2d &) const
```

This means that you cannot have both the method `operator+(const vec2d &) const` and the function `operator+(const vec2d &, const vec2d &)`:

```
class vec2d
{
public:
    ...
    vec2d operator+(const vec2d &) const;
    ...
};

...
vec2d operator+(const vec2d &, const vec2d &); // WRONG
...
```

This will cause an error during compiling.

This will result in an *ambiguous declaration*. For the `vec2d` class, we want `operator+` to be a method, not a function.

As for the bracket operator, when you do

```
std::cout << v[0];
```

you're really do this:

```
std::cout << v.operator[](0);
```

This requires you to retrieve the value of `v.x_`. However if you do this:

```
v[0] = 4.2;
```

then you're retrieve the value of `v.x_`. Instead you want to put `4.2` at `v.x_`. That's why there are two `operator[]` in your `vec2d` class. (Check your notes on returning references to instance variables.)

Reminder: Giving access to instance variable by returning a reference to an instance variable

If you want to set the value of an instance variable, you can do something like this:

```
#include <iostream>

class X
{
public:
    void set_i(int newi)
    {
        i = newi;
    }

    int i; // make this public for this experiment
};

int main()
{
    X x;
    x.set_i(5);
    std::cout << x.i << '\n'; // you should get 5
    return 0;
}
```

Another way to achieve the same thing is to give the client using your class (in this case client = main) access to the instance variable directly. You do that by returning a reference to the instance variable:

```
#include <iostream>

class X
{
public:
    void set_i(int newi)
    {
        i = newi;
    }
    int & iref() // obviously this method cannot be const
    {
        return i;
    }

    int i; // make this public for this experiment
};

int main()
{
    X x;
    x.set_i(5);
    std::cout << x.i << '\n'; // you should get 5
    x.iref() = 6;             // lefthand side of = is a reference to x.i
    std::cout << x.i << '\n'; // you should get 6
    return 0;
}
```

Make sure you see the difference between `X::set_i(int)` and `X::iref()`. The former sets the value of `i` for you. The client has an indirect write access to `i`. For `X::iref()`, the client has direct access to `i`.

If an instance variable is an array say `a` and you want to give a client access `a[0]`, then you give the client a public method that returns a reference to `a[0]`, not the value of `a[0]`.

Make sure you review notes on references and pointers.

The above is enough information for this assignment. Read on if you want to know a bit more ...

If you have the following in C++ (actually for most languages):

```
x = a + b + c;
```

Notice that there's a difference between the way you use the names on the right (i.e., `a`, `b`, and `c`) and the name on the left (i.e., `x`). Of course `a`, `b`, `c`, `x` all refer to a "boxes" containing

some value (more accurately, they refer to some part of your computer's memorry). However note that for `a`, `b`, `c`, you RETRIEVE (read access) the values at `a`, `b`, and `c`. For `x`, you PUT A VALUE INTO (write access) the box corresponding to `x`. What actually happens is this: you read the values for `a`, `b`, `c` and say `a`,`b`,`c` have value `1`,`2`,`3`, then the expression `a+b+c` generates `6`. Now, `a`, `b`, `c` themselves have values which are of course at some fixed memory location. However the value `6` does not have a fixed location - it's temporary. So we say that `6` is an **rvalue**. On the other hand the `x` on the left of the assignment, since it refers to the box of `x`, or more precisely to a chunk of memory, is an **lvalue**.

Informally, you can think of lvalue as "something that can be assigned a value and appears on the left of an assignment operator" while an rvalue is "something that cannot be assigned, is a value, and appears on the right of the assignment operator". That's enough for right now. The real picture is slightly more complicated.

Knowing the above is important because some compilers will use the above terminology (lvalue and rvalue) for error messages. For instance try to compile this program and then read the error message:

```
int main()
{
    2 = 5;
    return 0;
}
```

If you're using g++, it will give you an error message that basically says that the lefthand side of the assignment generates the value 2 which is not an lvalue: it cannot be assigned a value.