UNIVERSITY OF
RICHMOND

Class Members

CMSC 240 Software Systems Development

Today

- Constructors

- Enumerations

- Static members

- Operator overloading

- In-class activity

Today

- <mark>Constructors</mark>

- Enumerations

- Static members

- Operator overloading

- In-class activity

How do we design a class?

We must specify the **3 parts**:

1. Member variables: What variables make up this new type?

2. Member functions: What functions can you call on a variable of this type?

3. Constructor: What happens when you make a new instance of this type?

September 21, 2023

1. **Member variables**: What variables make up this new type?

2. **Member functions**: What functions can you call on a variable of this type?

3. **Constructor**: What happens when you make a new instance of this type?

```c
1    #ifndef DATE_H
2    #define DATE_H
3
4    class Date
5    {
6    public:
7        Date(int yyyy, int mm, int dd);  // constructor
8        void add_day(int num);
9        int getYear() { return year; }   // inline method declarations
10       int getMonth() { return month; }
11       int getDay() { return day; }
12   private:
13       int year, month, day;
14       bool is_valid();
15   };
16
17   #endif
```

```c
C Date.h > ...
  1   #ifndef DATE_H
  2   #define DATE_H
  3
  4   class Date
  5   {
  6   public:
  7       Date(int yyyy, int mm, int dd);  // constructor
  8       void add_day(int num);
  9       int getYear() { return year; }   // inline method declarations
 10       int getMonth() { return month; }
 11       int getDay() { return day; }
 12   private:
 13       int year, month, day;
 14       bool is_valid();
 15   };
 16
 17   #endif
```

Member variables

```c
C Date.h > ...
1     #ifndef DATE_H
2     #define DATE_H
3
4     class Date
5     {
6     public:
7         Date(int yyyy, int mm, int dd);  // constructor
8         void add_day(int num);
9         int getYear() { return year; }   // inline method declarations
10        int getMonth() { return month; }
11        int getDay() { return day; }
12    private:
13        int year, month, day;
14        bool is_valid();
15    };
16
17    #endif
```
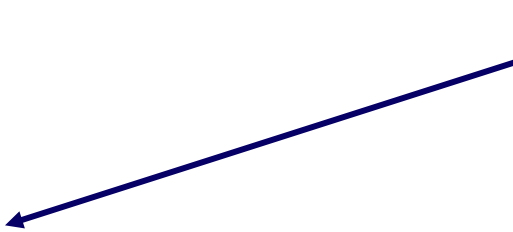
Member functions

```c
C Date.h > ...
1    #ifndef DATE_H
2    #define DATE_H
3
4    class Date
5    {
6    public:
7        Date(int yyyy, int mm, int dd);   // constructor
8        void add_day(int num);
9        int getYear() { return year; }    // inline method declarations
10       int getMonth() { return month; }
11       int getDay() { return day; }
12   private:
13       int year, month, day;
14       bool is_valid();
15   };
16
17   #endif
```

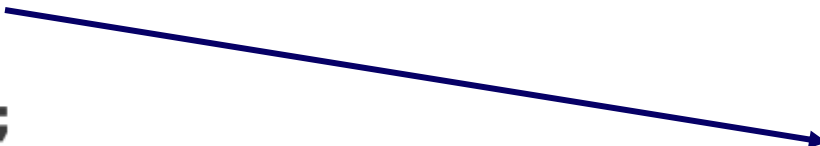Constructor

```cpp
#include "Date.h"

Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}
```

```cpp
#include "Date.h"

Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}

//    Date::Date(int yyyy, int mm, int dd)  // constructor
//    {
//        year = yyyy;                       // initialize members
//        month = mm;
//        day = dd;
//
//        is_valid();
//    }
```
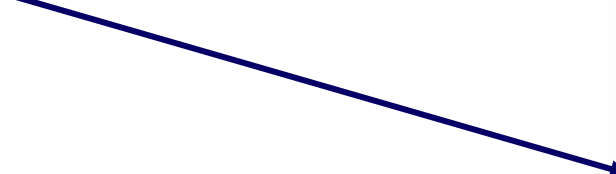
```cpp
#include "Date.h"

Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}

//      Date::Date(int yyyy, int mm, int dd)  // constructor
//      {
//          year = yyyy;                        // initialize members
//          month = mm;
//          day = dd;
//
//          is_valid();
//      }
```

int year = 1999;

```cpp
#include "Date.h"


Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}


//    Date::Date(int yyyy, int mm, int dd)  // constructor
//    {
//        year = yyyy;
//        month = mm;
//        day = dd;
//
//        is_valid();
//    }
```

int year = 1999;

int year;
    // ...
year = 1999;

```cpp
#include "Date.h"

int main()
{
    // Correct
    Date today1 = {2023, 9, 21};

    // Also correct
    Date today2{2023, 9, 21};

    // Also correct
    Date today3(2023, 9, 21);

    // Also correct
    Date today4 = Date{2023, 9, 21};

    // Also correct
    Date today5 = Date(2023, 9, 21);

    // Put the new Date object on the heap
    Date* todayPointer = new Date{2023, 9, 21};

    return 0;
}
```

Don't forget to free your memory

```
20        // Put the new Date object on the heap
21        Date* todayPointer = new Date{2023, 9, 21};
22
23        delete todayPointer;
```

```cpp
#include "Date.h"

int main()
{
    Date today1;                         // Error: no default constructor exists

    Date today2{};                       // Error: empty initializer

    Date today3{2023};                   // Error: too few arguments

    Date today4{1, 2, 3, 4};             // Error: to many arguments

    Date today5{2023, "sep", 21};        // Error: incorrect argument types

    return 0;
}
```

```c
1    #ifndef DATE_H
2    #define DATE_H
3
4    class Date
5    {
6    public:
7        Date();                              // default constructor
8        Date(int yyyy, int mm, int dd);  // constructor
9        void add_day(int num);
10       int getYear() { return year; }   // inline method declarations
11       int getMonth() { return month; }
12       int getDay() { return day; }
13   private:
14       int year, month, day;
15       bool is_valid();
16   };
17
18   #endif
```

```cpp
#include "Date.h"

Date::Date()  // default constructor
    : year{2021}, month{1}, day{1}
{
}


Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}
```

```cpp
#include "Date.h"

int main()
{
    Date today1;                        // It works...

    Date today2{};                      // It works...

    Date today3{2023};                  // Error: too few arguments

    Date today4{1, 2, 3, 4};            // Error: to many arguments

    Date today5{2023, "sep", 21};       // Error: incorrect argument types

    return 0;
}
```

Ask a question

Today

- ~~Constructors~~

- <mark>Enumerations</mark>

- Static members

- Operator overloading

- In-class activity

Enumerations

- An `enum` is a very simple user-defined type
    - Use them when you want a set of values as symbolic constants

```
1  enum class Month
2  {
3      jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
4  };
```

The body of an enumeration is simply a list of enumerators.

Enumerations

- An `enum` is a very simple user-defined type
  - Use them when you want a set of values as symbolic constants

```
1   enum class Month
2   {
3       jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
4   };
```
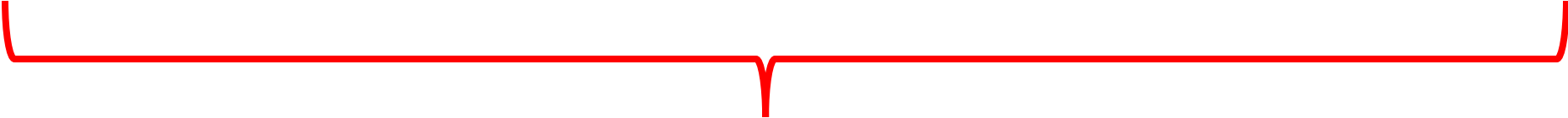
For any other enumerator whose definition does not have an initializer, the associated value is the value of the previous enumerator plus one

Enumerations

- An `enum` is a very simple user-defined type
  - Use them when you want a set of values as symbolic constants

```
1    enum class Month
2    {
3        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
4    };
```

```
1    enum class Month
2    {
3        jan=1, feb=2, mar=3, apr=4, may=5, jun=6, jul=7, aug=8, sep=9, oct=10, nov=11, dec=12
4    };
```

Enumerations

- An **enum** is a very simple user-defined type
  - Use them when you want a set of values as symbolic constants

```
1   enum class Month
2   {
3       jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
4   };
```
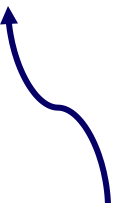
The **class** in **enum class** means that the
enumerators are in the scope of the enumeration.

To refer to **jan** we have to say **Month::jan**

Enumerations

- An `enum` is a very simple user-defined type
  - Use them when you want a set of values as symbolic constants

```
1    enum class Day
2    {
3        mon, tue, wed, thr, fri, sat, sun
4    };
```

If we don't initialize the first enumerator, the count starts with 0.

Here `mon` is represented as 0 and `sun` is represented as 6.

When to use an Enumeration

```cpp
1    #include "Date.h"
2
3    int main()
4    {
5
6        // Supposed to by Year, Month, Day
7        // But we entered Year, Day, Month
8        Date notValid = {2023, 21, 9};
9
10       // Correct
11       Date today = {2023, 9, 21};
12   }
```

```c
#ifndef DATE_H
#define DATE_H

enum class Month
{
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

class Date
{
public:
    Date(int yyyy, Month mm, int dd);  // constructor using enum
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return int(month); }
    int getDay() { return day; }
private:
    int year;
    Month month;
    int day;
    bool is_valid();
};

#endif
```

```c
#ifndef DATE_H
#define DATE_H

enum class Month
{
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

class Date
{
public:
    Date(int yyyy, Month mm, int dd);  // constructor using enum
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return int(month); }
    int getDay() { return day; }
private:
    int year;
    Month month;
    int day;
    bool is_valid();
};

#endif
```

```c
1   #ifndef DATE_H
2   #define DATE_H
3
4   enum class Month
5   {
6       jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
7   };
8
9   class Date
10  {
11  public:
12      Date(int yyyy, Month mm, int dd);  // constructor using enum
13      void add_day(int num);
14      int getYear() { return year; }
15      int getMonth() { return int(month); }
16      int getDay() { return day; }
17  private:
18      int year;
19      Month month;
20      int day;
21      bool is_valid();
22  };
23
24  #endif
```

```c
#ifndef DATE_H
#define DATE_H

enum class Month
{
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

class Date
{
public:
    Date(int yyyy, Month mm, int dd);  // constructor using enum
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return int(month); }
    int getDay() { return day; }
private:
    int year;
    Month month;
    int day;
    bool is_valid();
};

#endif
```

```c
#ifndef DATE_H
#define DATE_H

enum class Month
{
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

class Date
{
public:
    Date(int yyyy, Month mm, int dd);  // constructor using enum
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return int(month); }
    int getDay() { return day; }
private:
    int year;
    Month month;
    int day;
    bool is_valid();
};

#endif
```

```cpp
#include "Date.h"

Date::Date(int yyyy, Month mm, int dd)      // constructor
        : year{yyyy}, month{mm}, day{dd}    // member initializer list
{
    is_valid();
}
```

```cpp
1    #include "Date.h"
2
3    Date::Date(int yyyy, Month mm, int dd)      // constructor
4              : year{yyyy}, month{mm}, day{dd}      // member initializer list
5    {
6        is_valid();
7    }
```

```cpp
1   #include <iostream>
2   #include "Date.h"
3
4   int main()
5   {
6       Date today = {2023, Month::sep, 21};
7
8       std::cout << "year == " << today.getYear() << std::endl;
9       std::cout << "month == " << today.getMonth() << std::endl;
10      std::cout << "day == " << today.getDay() << std::endl;
11  }
```
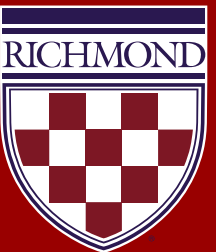
```cpp
1    #include <iostream>
2    #include "Date.h"
3
4    int main()
5    {
6        Date today = {2023, Month::sep, 21};
7
8        std::cout << "year == " << today.getYear() << std::endl;
9        std::cout << "month == " << today.getMonth() << std::endl;
10       std::cout << "day == " << today.getDay() << std::endl;
11   }
```

```c
C Toaster.h > ...
1    #ifndef TOASTER_H
2    #define TOASTER_H
3
4
5    class Toaster
6    {
7
8    public:
9        Toaster(int initialLevel);
10       void toast();
11       void cancel();
12       bool isOn();
13       int getLevel();
14       void setLevel(int newLevel);
15   private:
16       int heatLevel;
17       bool isToasting;
18       bool isValidLevel(int level);
19   };
20
21   #endif
```

Where could you add an enumeration to your design?

Today

- ~~Constructors~~

- ~~Enumerations~~

- <mark>Static members</mark>

- Operator overloading

- In-class activity

Static Member Variables

- **Static member variables** can be accessed on the class itself, without creating an instance of the class

- Exists only once, regardless of how many instances of the class are created

- Shared among all instances of the class

- Value is set outside the class, typically in a source (.cpp) file, even if it's declared const (this is required to allocate storage for it)

```c
#ifndef DATE_H
#define DATE_H

class Date
{
public:
    Date();
    Date(int yyyy, int mm, int dd);
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return month; }
    int getDay() { return day; }
    static int DEFAULT_YEAR;
private:
    int year, month, day;
    bool is_valid();
};

#endif
```

```c
#ifndef DATE_H
#define DATE_H

class Date
{
public:
    Date();
    Date(int yyyy, int mm, int dd);
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return month; }
    int getDay() { return day; }
    static int DEFAULT_YEAR;
private:
    int year, month, day;
    bool is_valid();
};

#endif
```

```cpp
#include "Date.h"

Date::Date()  // default constructor
    : year{DEFAULT_YEAR}, month{1}, day{1}
{
}

Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}

int Date::DEFAULT_YEAR = 2001;
```

```cpp
Date.cpp > ...
  1    #include "Date.h"
  2
  3    Date::Date()  // default constructor
  4        : year{DEFAULT_YEAR}, month{1}, day{1}
  5    {
  6    }
  7
  8    Date::Date(int yyyy, int mm, int dd)  // constructor
  9        : year{yyyy}, month{mm}, day{dd}  // member initializer list
 10    {
 11        is_valid();
 12    }
 13
 14    int Date::DEFAULT_YEAR = 2001;
```

```cpp
#include "Date.h"
#include <iostream>
using namespace std;

int main()
{
    cout << Date::DEFAULT_YEAR << endl;

    return 0;
}
```

```cpp
1    #include "Date.h"
2    #include <iostream>
3    using namespace std;
4
5    int main()
6    {
7        cout << Date::DEFAULT_YEAR << endl;
8
9        return 0;
10   }
```

```cpp
#include "Date.h"
#include <iostream>
using namespace std;

int main()
{
    // Update the default year in all
    // future instances of the Date class
    Date::DEFAULT_YEAR = 2023;

    cout << Date::DEFAULT_YEAR << endl;

    return 0;
}
```

C++ TestDate.cpp > ...

```cpp
1    #include "Date.h"
2    #include <iostream>
3    using namespace std;
4
5    int main()
6    {
7        // Update the default year in all
8        // future instances of the Date class
9        Date::DEFAULT_YEAR = 2023;
10
11       cout << Date::DEFAULT_YEAR << endl;
12
13       return 0;
14   }
```

```c
#ifndef DATE_H
#define DATE_H

class Date
{
public:
    Date();
    Date(int yyyy, int mm, int dd);
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return month; }
    int getDay() { return day; }
    static const int DEFAULT_YEAR;
private:
    int year, month, day;
    bool is_valid();
};

#endif
```

```c
1    #ifndef DATE_H
2    #define DATE_H
3
4    class Date
5    {
6    public:
7        Date();
8        Date(int yyyy, int mm, int dd);
9        void add_day(int num);
10       int getYear() { return year; }
11       int getMonth() { return month; }
12       int getDay() { return day; }
13       static const int DEFAULT_YEAR;
14   private:
15       int year, month, day;
16       bool is_valid();
17   };
18
19   #endif
```

Date.cpp > ...

```cpp
1    #include "Date.h"
2
3    Date::Date()  // default constructor
4        : year{DEFAULT_YEAR}, month{1}, day{1}
5    {
6    }
7
8    Date::Date(int yyyy, int mm, int dd)  // constructor
9        : year{yyyy}, month{mm}, day{dd}  // member initializer list
10   {
11       is_valid();
12   }
13
14   const int Date::DEFAULT_YEAR = 2001;
```

```cpp
TestDate.cpp > ...
1    #include "Date.h"
2    #include <iostream>
3    using namespace std;
4
5    int main()
6    {
7        // Update the default year in all
8        // future instances of the Date class
9        Date::DEFAULT_YEAR = 2023;
10
11       cout << Date::DEFAULT_YEAR << endl;
12
13       return 0;
14   }
```

Error: Can not modify a **const** value.

Static Member Functions

- **Static member functions** can be called on the class itself, without creating an instance of the class

- It can only access static member variables or other static member functions directly

- It's often used as a utility function or to interact with static member variables.

```c
#ifndef DATE_H
#define DATE_H

class Date
{
public:
    Date();
    Date(int yyyy, int mm, int dd);
    void add_day(int num);
    int getYear() { return year; }
    int getMonth() { return month; }
    int getDay() { return day; }
    static int DEFAULT_YEAR;
    static void setDefaultYear(int yearDefault);
private:
    int year, month, day;
    bool is_valid();
};

#endif
```

```c
C Date.h > ...
1     #ifndef DATE_H
2     #define DATE_H
3
4     class Date
5     {
6     public:
7         Date();
8         Date(int yyyy, int mm, int dd);
9         void add_day(int num);
10        int getYear() { return year; }
11        int getMonth() { return month; }
12        int getDay() { return day; }
13        static int DEFAULT_YEAR;
14        static void setDefaultYear(int yearDefault);
15    private:
16        int year, month, day;
17        bool is_valid();
18    };
19
20    #endif
```

```cpp
#include "Date.h"

Date::Date()  // default constructor
    : year{DEFAULT_YEAR}, month{1}, day{1}
{
}

Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}

int Date::DEFAULT_YEAR = 2001;

void Date::setDefaultYear(int yearDefault)
{
    DEFAULT_YEAR = yearDefault;
}
```

```cpp
#include "Date.h"

Date::Date()  // default constructor
    : year{DEFAULT_YEAR}, month{1}, day{1}
{
}

Date::Date(int yyyy, int mm, int dd)  // constructor
    : year{yyyy}, month{mm}, day{dd}  // member initializer list
{
    is_valid();
}

int Date::DEFAULT_YEAR = 2001;

void Date::setDefaultYear(int yearDefault)
{
    DEFAULT_YEAR = yearDefault;
}
```

```cpp
TestDate.cpp > ...
1    #include "Date.h"
2    #include <iostream>
3    using namespace std;
4
5    int main()
6    {
7        Date::setDefaultYear(2023);
8
9        cout << Date::DEFAULT_YEAR << endl;
10
11       return 0;
12   }
```

```cpp
TestDate.cpp > ...
1    #include "Date.h"
2    #include <iostream>
3    using namespace std;
4
5    int main()
6    {
7        Date::setDefaultYear(2023);
8
9        cout << Date::DEFAULT_YEAR << endl;
10
11       return 0;
12   }
```
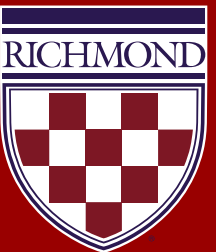
```c
C Toaster.h > ...
 1    #ifndef TOASTER_H
 2    #define TOASTER_H
 3
 4
 5    class Toaster
 6    {
 7
 8    public:
 9        Toaster(int initialLevel);
10        void toast();
11        void cancel();
12        bool isOn();
13        int getLevel();
14        void setLevel(int newLevel);
15    private:
16        int heatLevel;
17        bool isToasting;
18        bool isValidLevel(int level);
19    };
20
21    #endif
```

Today

- ~~Constructors~~

- ~~Enumerations~~

- ~~Static members~~

- Operator overloading

- In-class activity

Method Overloading

You can reuse method names if the **parameters** in the method signature are different.

```c
C  MathOperations.h > ...
1    class MathOperations {
2    public:
3        // Method to add two integers
4        int add(int a, int b) { return a + b; }
5
6        // Works: Method to add three integers
7        int add(int a, int b, int c) { return a + b + c; }
8
9        // Works: Method to add two doubles
10       double add(double a, double b) { return a + b; }
11
12       // Error: note: previous declaration
13       // The return type is NOT considered while differentiating the overloaded methods,
14       // so you cannot create an overloaded method just by changing the return type.
15       float add(double a, double b) { return a + b; }
16   };
```

Operator Overloading

- **Operator overloading** is a feature in C++ that allows you to redefine the behavior of built-in operators (like +, -, *, etc.) for user-defined types like classes

- This enables you to use these operators in intuitive ways with objects of your custom types, making your code more readable and expressive

any of the following operators: `+` `-` `*` `/` `%` `^` `&` `|` `~` `!` `=` `<` `>` `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<` `>>` `>>=` `<<=` `==` `!=` `<=` `>=` `<=>` (since C++20) `&&` `||` `++` `--` `,` `->*` `->` `( )` `[ ]`
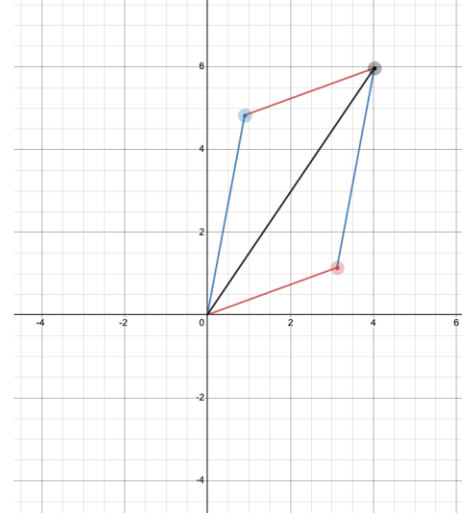
Operator Overloading

- **Syntax**: Operator overloading is achieved by defining special member functions with the keyword `operator` followed by the operator symbol you wish to overload
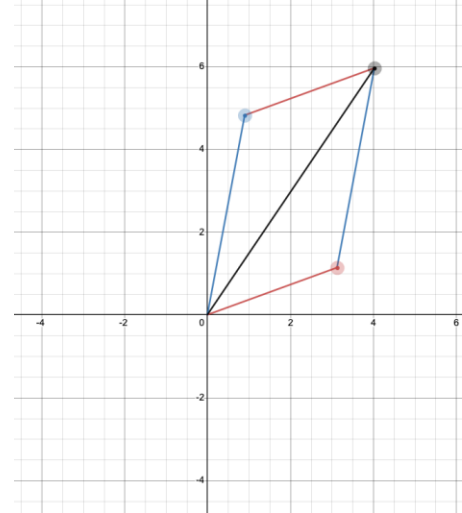
```cpp
1    #ifndef VECTOR_H
2    #define VECTOR_H
3
4    class Vector
5    {
6    public:
7        Vector(float x, float y);  // Constructor
8        Vector operator+(const Vector& other) const;  // Overload +
9        float getX() const { return x; }
10       float getY() const { return y; }
11   private:
12       float x;
13       float y;
14   };
15
16   #endif
```

```cpp
1    #ifndef VECTOR_H
2    #define VECTOR_H
3
4    class Vector
5    {
6    public:
7        Vector(float x, float y);   // Constructor
8        Vector operator+(const Vector& other) const;  // Overload +
9        float getX() const { return x; }
10       float getY() const { return y; }
11   private:
12       float x;
13       float y;
14   };
15
16   #endif
```

```cpp
Vector.cpp > ...
1    #include "Vector.h"
2
3    Vector::Vector(float x, float y) : x(x), y(y) {}
4
5    Vector Vector::operator+(const Vector& other) const
6    {
7        // Return an instance of Vector that is
8        // this vector added to the other vector.
9        return Vector(this->x + other.x, this->y + other.y);
10   }
```

```cpp
Vector.cpp > ...
  1    #include "Vector.h"
  2
  3    Vector::Vector(float x, float y) : x(x), y(y) {}
  4
  5    Vector Vector::operator+(const Vector& other) const
  6    {
  7        // Return an instance of Vector that is
  8        // this vector added to the other vector.
  9        return Vector(this->x + other.x, this->y + other.y);
 10    }
```

```cpp
#include <iostream>
#include "Vector.h"
using namespace std;

int main()
{
    Vector v1(1, 2);
    Vector v2(3, 4);

    Vector v3 = v1 + v2;   // Uses the overloaded + operator

    // Print out the vector.
    cout << "v1 == [" << v1.getX() << ", " << v1.getY() << "]" << endl;
    cout << "v2 == [" << v2.getX() << ", " << v2.getY() << "]" << endl;
    cout << "v3 == [" << v3.getX() << ", " << v3.getY() << "]" << endl;
}
```

```cpp
TestVector.cpp > ...
1    #include <iostream>
2    #include "Vector.h"
3    using namespace std;
4
5    int main()
6    {
7        Vector v1(1, 2);
8        Vector v2(3, 4);
9
10       Vector v3 = v1 + v2;   // Uses the overloaded + operator
11
12       // Print out the vector.
13       cout << "v1 == [" << v1.getX() << ", " << v1.getY() << "]" << endl;
14       cout << "v2 == [" << v2.getX() << ", " << v2.getY() << "]" << endl;
15       cout << "v3 == [" << v3.getX() << ", " << v3.getY() << "]" << endl;
16   }
```

```cpp
#include <iostream>
#include "Vector.h"
using namespace std;

int main()
{
    Vector v1(1, 2);
    Vector v2(3, 4);

    Vector v3 = v1 + v2;   // Uses the overloaded + operator

    // Print out the vector.
    cout << "v1 == [" << v1.getX() << ", " << v1.getY() << "]" << endl;
    cout << "v2 == [" << v2.getX() << ", " << v2.getY() << "]" << endl;
    cout << "v3 == [" << v3.getX() << ", " << v3.getY() << "]" << endl;
}
```

```
v1 == [1, 2]
v2 == [3, 4]
v3 == [4, 6]
```

```cpp
#include <iostream>
#include "Vector.h"
using namespace std;

ostream& operator<<(ostream& out, const Vector& v)
{
    out << "[" << v.getX() << ", " << v.getY() << "]";
    return out;
}

int main()
{
    Vector v1(1, 2);
    Vector v2(3, 4);

    Vector v3 = v1 + v2;   // Uses the overloaded + operator

    // Print out the vector.
    cout << "v1 == " << v1 << endl;
    cout << "v2 == " << v2 << endl;
    cout << "v3 == " << v3 << endl;
}
```

```cpp
#include <iostream>
#include "Vector.h"
using namespace std;

ostream& operator<<(ostream& out, const Vector& v)
{
    out << "[" << v.getX() << ", " << v.getY() << "]";
    return out;
}

int main()
{
    Vector v1(1, 2);
    Vector v2(3, 4);

    Vector v3 = v1 + v2;   // Uses the overloaded + operator

    // Print out the vector.
    cout << "v1 == " << v1 << endl;
    cout << "v2 == " << v2 << endl;
    cout << "v3 == " << v3 << endl;
}
```

```cpp
#include <iostream>
#include "Vector.h"
using namespace std;

ostream& operator<<(ostream& out, const Vector& v)
{
    out << "[" << v.getX() << ", " << v.getY() << "]";
    return out;
}

int main()
{
    Vector v1(1, 2);
    Vector v2(3, 4);

    Vector v3 = v1 + v2;  // Uses the overloaded + operator

    // Print out the vector.
    cout << "v1 == " << v1 << endl;
    cout << "v2 == " << v2 << endl;
    cout << "v3 == " << v3 << endl;
}
```

```cpp
1   #include <iostream>
2   #include "Vector.h"
3   using namespace std;
4
5   ostream& operator<<(ostream& out, const Vector& v)
6   {
7       out << "[" << v.getX() << ", " << v.getY() << "]";
8       return out;
9   }
10
11  int main()
12  {
13      Vector v1(1, 2);
14      Vector v2(3, 4);
15
16      Vector v3 = v1 + v2;   // Uses the overloaded + operator
17
18      // Print out the vector.
19      cout << "v1 == " << v1 << endl;
20      cout << "v2 == " << v2 << endl;
21      cout << "v3 == " << v3 << endl;
22  }
```
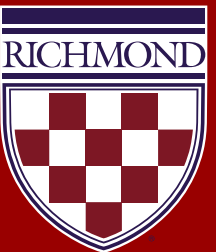
```
v1 == [1, 2]
v2 == [3, 4]
v3 == [4, 6]
```

```c
C Toaster.h > ...
1    #ifndef TOASTER_H
2    #define TOASTER_H
3
4
5    class Toaster
6    {
7
8    public:
9        Toaster(int initialLevel);
10       void toast();
11       void cancel();
12       bool isOn();
13       int getLevel();
14       void setLevel(int newLevel);
15   private:
16       int heatLevel;
17       bool isToasting;
18       bool isValidLevel(int level);
19   };
20
21   #endif
```

Where could you use operator overloading in your design?

Today

- ~~Constructors~~

- ~~Enumerations~~

- ~~Static members~~

- ~~Operator overloading~~

- <mark>In class activity</mark>

Visibility

```
-   private
+   public
#   protected
```