



UNIVERSITY OF
RICHMOND

CMSC 240 Lecture 8

CMSC 240 Software Systems Development
Fall 2023

Today

- Constructors
- Enumerations
- Static members
- Operator overloading
- In-class activity



Today

- Constructors
- In-class activity
- Enumerations
- Static members
- Operator overloading



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 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
```

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 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 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

⌚ Date.cpp > ...

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

⌚ Date.cpp > ...

```
1 #include "Date.h"
2
3 Date::Date(int yyyy, int mm, int dd) // constructor
4     : year{yyyy}, month{mm}, day{dd} // member initializer list
5 {
6     is_valid();
7 }
8
9 // Date::Date(int yyyy, int mm, int dd) // constructor
10 // {
11 //     year = yyyy;                      // initialize members
12 //     month = mm;
13 //     day = dd;
14 //
15 //     is_valid();
16 // }
```

⌚ Date.cpp > ...

```
1 #include "Date.h"
2
3 Date::Date(int yyyy, int mm, int dd) // constructor
4     : year{yyyy}, month{mm}, day{dd} // member initializer list
5 {
6     is_valid();
7 }
8
9 // Date::Date(int yyyy, int mm, int dd) // constructor
10 // {
11 //     year = yyyy;                      // initialize members
12 //     month = mm;
13 //     day = dd;
14 //
15 //     is_valid();
16 // }
```

int year = 1999;

⌚ Date.cpp > ...

```
1 #include "Date.h"
2
3 Date::Date(int yyyy, int mm, int dd) // constructor
4     : year{yyyy}, month{mm}, day{dd} // member initializer list
5 {
6     is_valid();
7 }
8
9 // Date::Date(int yyyy, int mm, int dd) // constructor
10 // {
11 //     year = yyyy;
12 //     month = mm;
13 //     day = dd;
14 //
15 //     is_valid();
16 // }
```

int year = 1999;

// ...

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

```
1 #include "Date.h"
2
3 int main()
4 {
5     // Correct
6     Date today1 = {2023, 9, 21};
7
8     // Also correct
9     Date today2{2023, 9, 21};
10
11    // Also correct
12    Date today3(2023, 9, 21);
13
14    // Also correct
15    Date today4 = Date{2023, 9, 21};
16
17    // Also correct
18    Date today5 = Date(2023, 9, 21);
19
20    // Put the new Date object on the heap
21    Date* todayPointer = new Date{2023, 9, 21};
22
23    return 0;
24 }
```

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;
```

```
1 #include "Date.h"
2
3 int main()
4 {
5     Date today1; // Error: no default constructor exists
6
7     Date today2{}; // Error: empty initializer
8
9     Date today3{2023}; // Error: too few arguments
10
11    Date today4{1, 2, 3, 4}; // Error: to many arguments
12
13    Date today5{2023, "sep", 21}; // Error: incorrect argument types
14
15    return 0;
16 }
```

C Date.h > ...

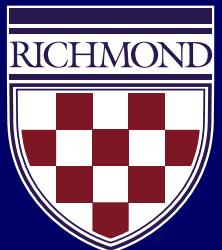
```
1  #ifndef DATE_H
2  #define DATE_H
3
4  class Date
5  {
6  public:
7      Date();                      // default constructor
8      Date(int yyyy, int mm, int dd); // constructor
9      void add_day(int num);
10     int getYear() { return year; }   // inline method declarations
11     int getMonth()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

C++ Date.cpp > ...

```
1 #include "Date.h"
2
3 Date::Date() // default constructor
4     : year{2021}, 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 }
```

```
1 #include "Date.h"
2
3 int main()
4 {
5     Date today1;                      // It works...
6
7     Date today2{};                   // It works...
8
9     Date today3{2023};              // Error: too few arguments
10
11    Date today4{1, 2, 3, 4};        // Error: to many arguments
12
13    Date today5{2023, "sep", 21};   // Error: incorrect argument types
14
15    return 0;
16 }
```

Ask a question



Today

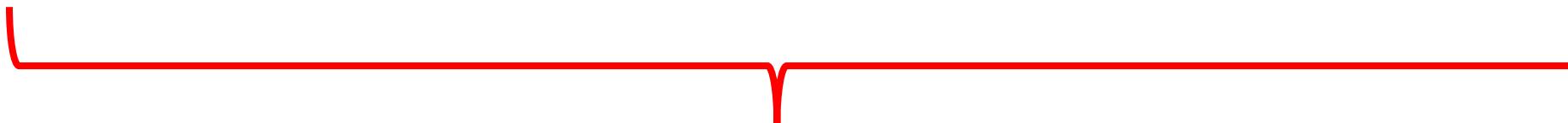
- Constructors
- Enumerations
- Static members
- Operator overloading
- In-class activity



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 }
```

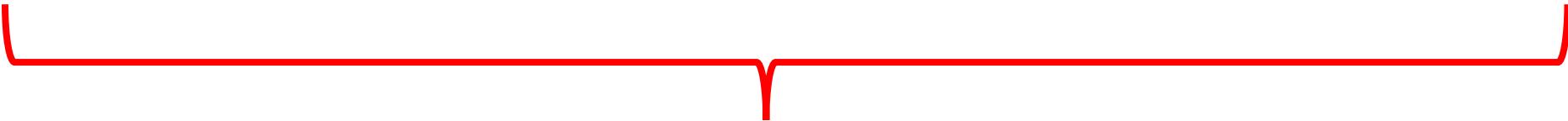


The body of an enumeration is simply a list of **enumerators**.

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

```
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 }
```

lecture8 > enums > **C** Date.h > ...

```
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
```

lecture8 > enums > **C** Date.h > ...

```
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
```

lecture8 > enums > C Date.h > ...

```
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; // This line is highlighted with a red box
20     int day;
21     bool is_valid();
22 };
23
24 #endif
```

lecture8 > enums > C Date.h > ...

```
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
```

lecture8 > enums > C Date.h > ...

```
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); } // Line 15 is highlighted
16     int getDay() { return day; }
17 private:
18     int year;
19     Month month;
20     int day;
21     bool is_valid();
22 };
23
24 #endif
```

lecture8 > enums >  Date.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 }
```

lecture8 > enums >  Date.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 }
```

lecture8 > enums >  TestDate.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 }
```

lecture8 > enums >  TestDate.cpp > ...

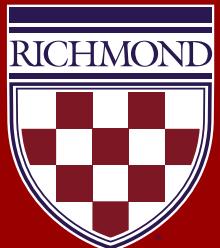
```
1 #include <iostream>
2 #include "Date.h"
3
4 int main()
5 {
6     Date today = {2023, Month::sep, 21};
7
8     std::cout << "year == " << today.getYear() << std::endl;
9     std::cout << "month == " << today.getMonth() << std::endl;
10    std::cout << "day == " << today.getDay() << std::endl;
11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

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

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 private:
15     int year, month, day;
16     bool is_valid();
17 };
18
19 #endif
```

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 private:
15     int year, month, day;
16     bool is_valid();
17 };
18
19 #endif
```

⌚ 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;
```

⌚ 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;
```

⌚ TestDate.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 }
```

⌚ TestDate.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 }
```

C++ 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 }
```

C++ 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 }
```

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 const int DEFAULT_YEAR;
14 private:
15     int year, month, day;
16     bool is_valid();
17 };
18
19 #endif
```

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 const int DEFAULT_YEAR;
14 private:
15     int year, month, day;
16     bool is_valid();
17 };
18
19 #endif
```

C++ 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 const int Date::DEFAULT_YEAR = 2001;
```



C++ 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    cout << Date::DEFAULT_YEAR << endl;
11
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 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
```

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); // Line 14
15 private:
16     int year, month, day;
17     bool is_valid();
18 };
19
20 #endif
```

C++ 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;
15
16 void Date::setDefaultYear(int yearDefault)
17 {
18     DEFAULT_YEAR = yearDefault;
19 }
```

C++ 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;
15
16 void Date::setDefaultYear(int yearDefault)
17 {
18     DEFAULT_YEAR = yearDefault;
19 }
```

⊕ 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 }
```

⊕ TestDate.cpp > ...

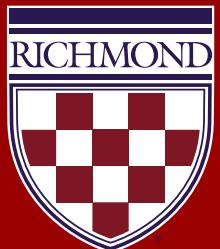
```
1 #include "Date.h"
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     Date::setDefaultYear(2023);
8
9     cout << Date::DEFAULT_YEAR << endl;
10
11    return 0;
12 }
```

C 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 static
variables or methods to your
design?



Today

- Constructors
- Enumerations
- Static members
- Operator overloading
- In-class activity



Method Overloading

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 }
```

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

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

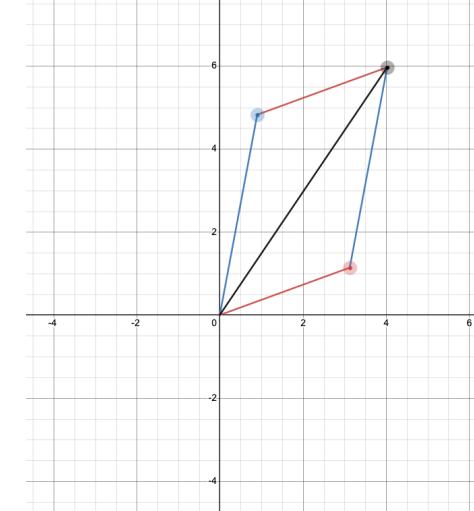
any of the following operators: + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <>= (since C++20) && || ++ -- , ->* -> () []

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

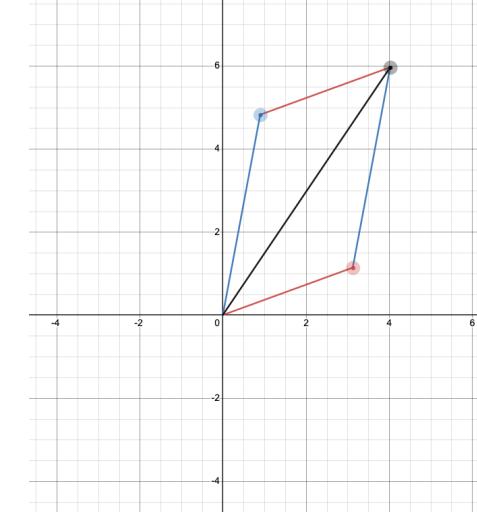
C Vector.h > ...

```
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
```



C Vector.h > ...

```
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
```



C++ 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 }
```

C++ 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 }
```

C++ 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 }
```

C++ 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 }
```

C++ 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 }
```

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

↳ TestVector.cpp > `main()`

```
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 }
```

↳ TestVector.cpp > `main()`

```
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 }
```

↳ TestVector.cpp > `main()`

```
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 }
```

↳ TestVector.cpp > `main()`

```
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 }
```

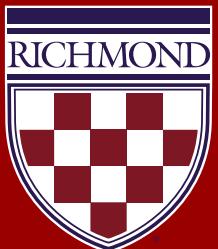
v1 == [1, 2]
v2 == [3, 4]
v3 == [4, 6]

C 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
- In class activity

