COMP 3522

Object Oriented Programming in C++

Week 4

Agenda

1. Abstract classes & interfaces
2. Multiple Inheritance

COMP 3522

ABSTRACT CLASSES

Java Abstract classes

```java
public abstract class AbstractClass {
}

public class ConcreteClass extends AbstractClass {
}
```

C++ Abstract classes

- Cannot be instantiated (just like Java!)
- Are used to define an implementation or a base class
- Intended to be extended by derived classes
- **Implemented as a class that has one or more pure virtual functions**

What is a purely virtual function?

```
class AbstractClass
{
public:
   virtual void AbstractMemberFunction( ) = 0;
   virtual void NonAbstractMemberFunction1( );
   void NonAbstractMemberFunction2( );
   int x;
};
```

PURE SPECIFIER

What is a purely virtual function?

```
class ConcreteClass : public AbstractClass
{
public:
    void AbstractMemberFunction( ) override { }
  void NonAbstractMemberFunction1( ) override { }
};
```

Pure Specifier

- A pure virtual function **MUST** be overridden by a concrete derived class
- A function declaration cannot have both a **pure specifier** and a **definition**
- For example, the **compiler will not allow the following**:

```
class A
{
    virtual void g() { } = 0; // ERROR!
};
```

Rules for abstract classes

- We **cannot** use an abstract class as a:
  - Function return type
  - Parameter type

```
class A // Abstract class
{
    virtual void g() = 0;
};
A functionA(); // WRONG cannot return an A
void functionB(A aParam); // WRONG cannot accept an A
```

Rules for abstract classes

- We **can** use:
  - Pointers to an abstract class
  - References to an abstract class

```
class A // Abstract class
{
    virtual void g() = 0;
};
A* pa; // OK
A& functionA(A& aParam); // OK
```

Virtual members are inherited

- A class derived from an abstract class will be abstract unless we override each purely virtual function in the derived class (just like Java!)
- We can derive an abstract class from a non-abstract class
- **CAUTION**: calling (directly or indirectly) a purely virtual function from an abstract class constructor is **UNDEFINED**

- see **oop_abstract.cpp** and oop_virtual.cpp*

INTERFACES

Java Interfaces

```
public interface Animal {
}

public class Dog implements Animal {
}
```

C++ Interfaces

- Describe behavior of class **without committing to an implementation**
- **<u>No implementation</u>**
- Specifies a polymorphic interface
- **Virtual destructor** to ensure that when an instance of an implementing class is deleted polymorphically, the correct destructor of the derived class is called
- Pure virtual functions, no other kinds of functions
  - \+ regular virtual destructor

Interfaces

```
class Animal
{
public:
    virtual ~Animal( ) {} //regular virtual destructor
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void eat( ) = 0;
}
```

Abstract class vs interface

- **Abstract class** is used to define an implementation and is intended to be extended by concrete classes
- Enforces a contract between the class designer and the users of that class
  - At least one pure virtual function
  - Can have:
    - data members
    - virtual and non-virtual member functions
- An **interface** is a "pure abstract class" in C++:
  - Purely virtual functions
  - No data

Not implemented ⟷ Fully implemented

- **Interface**
- Only pure virtual functions
- Virtual destructor
- No data members
- Can NOT be instantiated

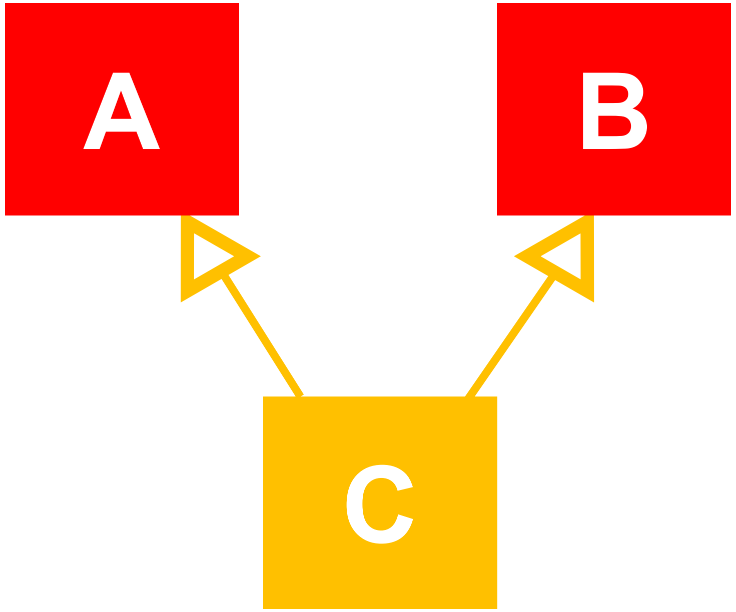
- **Abstract class**
- At least 1 pure virtual function
- Virtual destructor
- Has functions and data members
- Can NOT be instantiated

- **Concrete Class**
- No pure virtual functions
- Virtual destructor if base class that has children
- Has functions and data members
- CAN be instantiated

MULTIPLE INHERITANCE

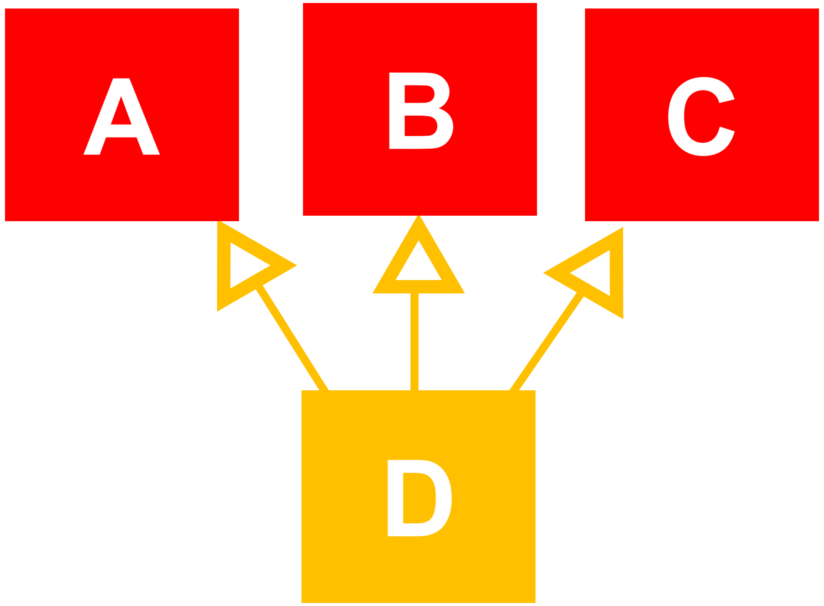
Multiple Inheritance

- Java: each subclass has one superclass
- C++: a **derived class** can have **more than one base class**
- With two parents, the class hierarchy looks like a V



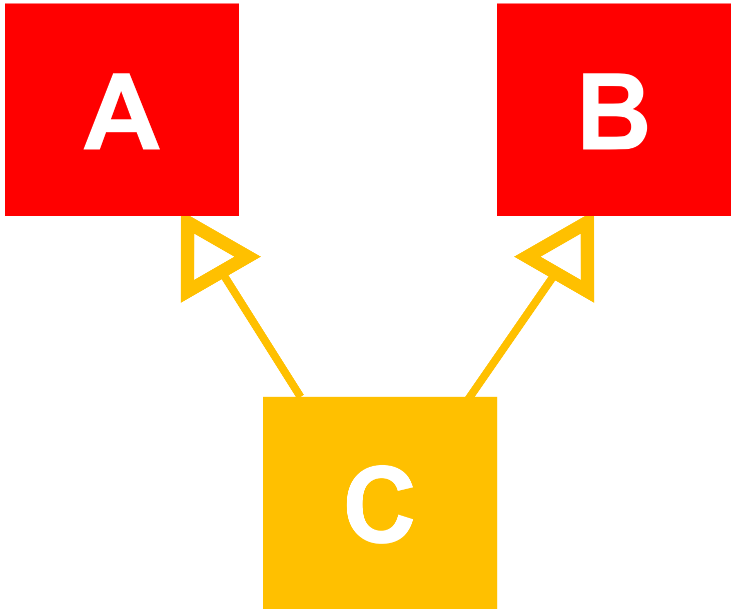
Multiple Inheritance

- With many parents, the class hierarchy looks like a bouquet



- The members of the **derived class** are the **union** of all **base class** members
- **DANGER: there can be ambiguities!**

Multiple Inheritance

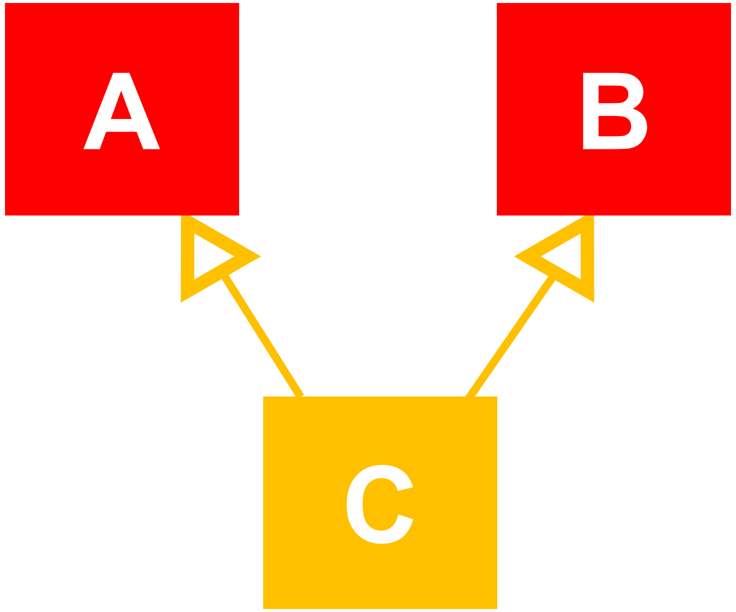


- A has `int x, void function1()`
- B has `int y, void function2()`

- C inherits everything that's public/protected
  - `int x`
  - `int y`
  - `function1();`
  - `function2();`

- There are no conflicts in this case. Data members and functions all have different names in class C

Multiple Inheritance (Ambiguity)

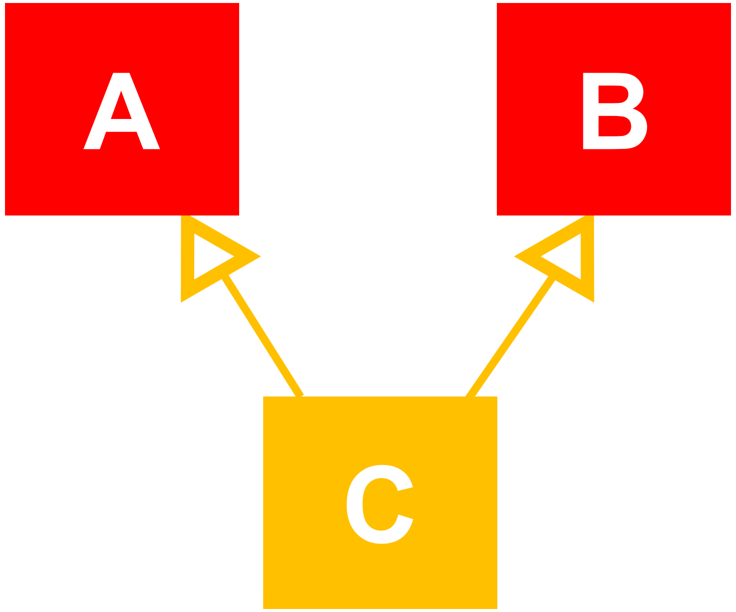


- **A** has `int x, void function1()`
- **B** has `int` **`x`**`, void` **`function1()`**

- C inherits everything that's public/protected
  - `int x`
  - `int` **`x`** `//same name as other x`
  - `function1();`
  - **`function1();`** `//same name as other function`

- C can't access **x** and **`function1`** directly. Ambiguous

Multiple Inheritance (Ambiguity)



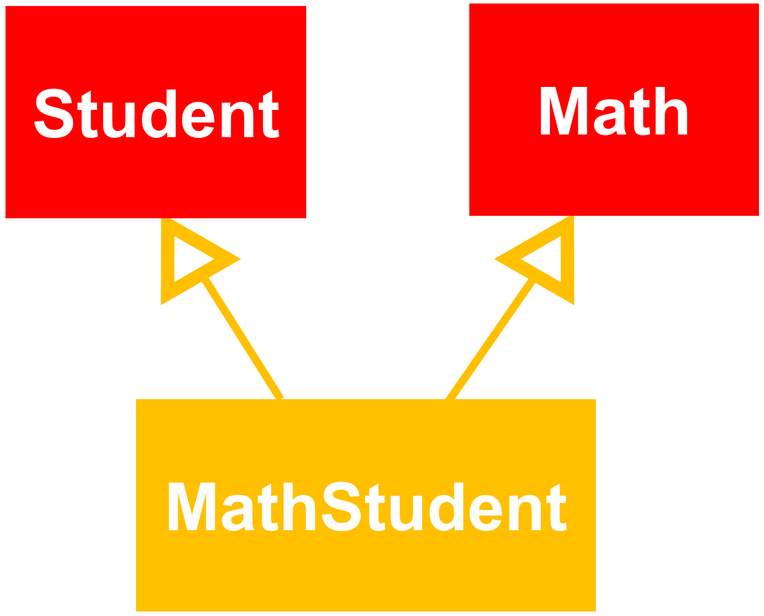
- Can get around ambiguity by scoping

```
C c;
int num = c.B::x;
c.B::function1();
```

Multiple inheritance

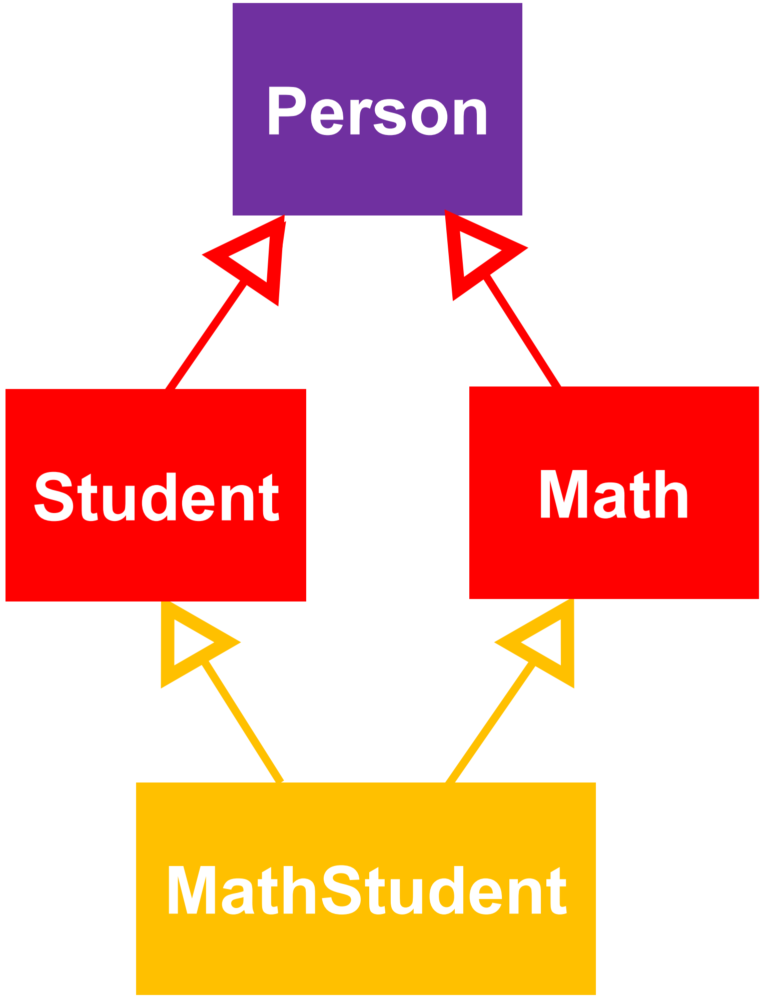
- Consider the example in **oop_multi0.cpp**

- **math_student** inherits a member function from both **student** and **mathematician**
- There is no priority for one or the other
- We say that all_info is not defined in **math_student**, and it is **ambiguously inherited**



Multiple inheritance

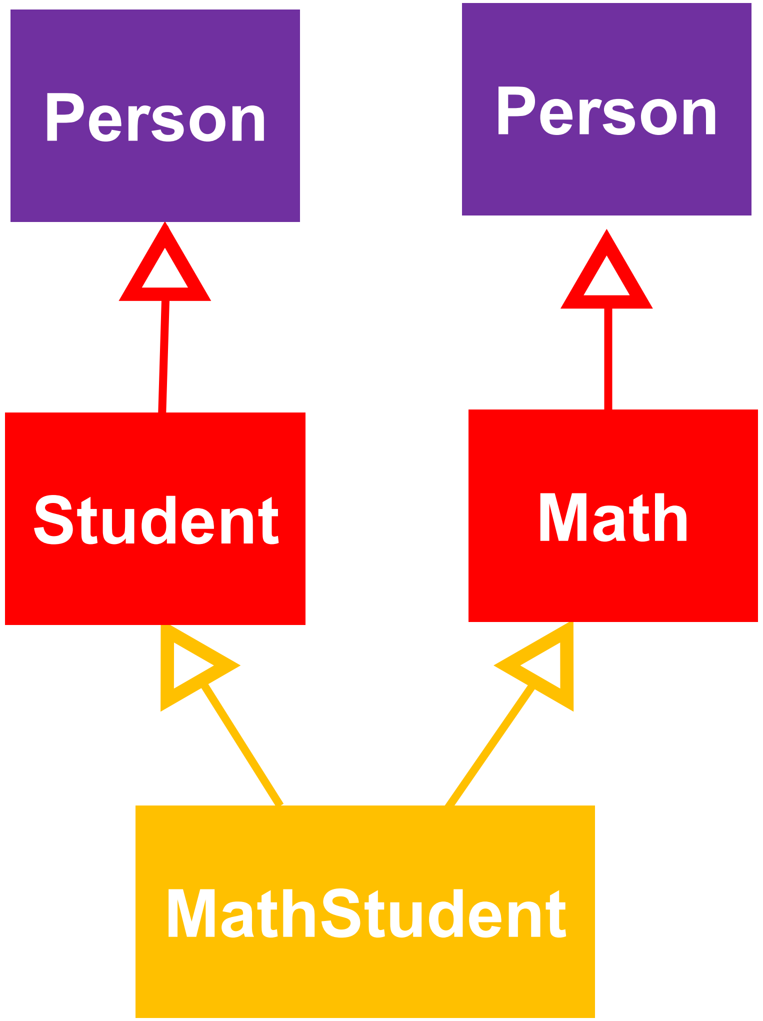
- **Two classes** may be derived from the same **base class**
- These two **derived classes** may be the base class for another **derived class**
- There are common grandparents
- This creates a classic **diamond shape** inheritance configuration
- But how many **grandparents** are created?



See: **oop_multil.cpp**

Virtual base classes (motivation)

- When creating a **math_student** object, its constructor must call the **student** constructor and the **mathematician** constructor
- When creating a **student** object its constructor must call the **person** constructor
- When creating a **mathematician** object its constructor must call the **person** constructor
- **We don't want to construct the shared person twice**



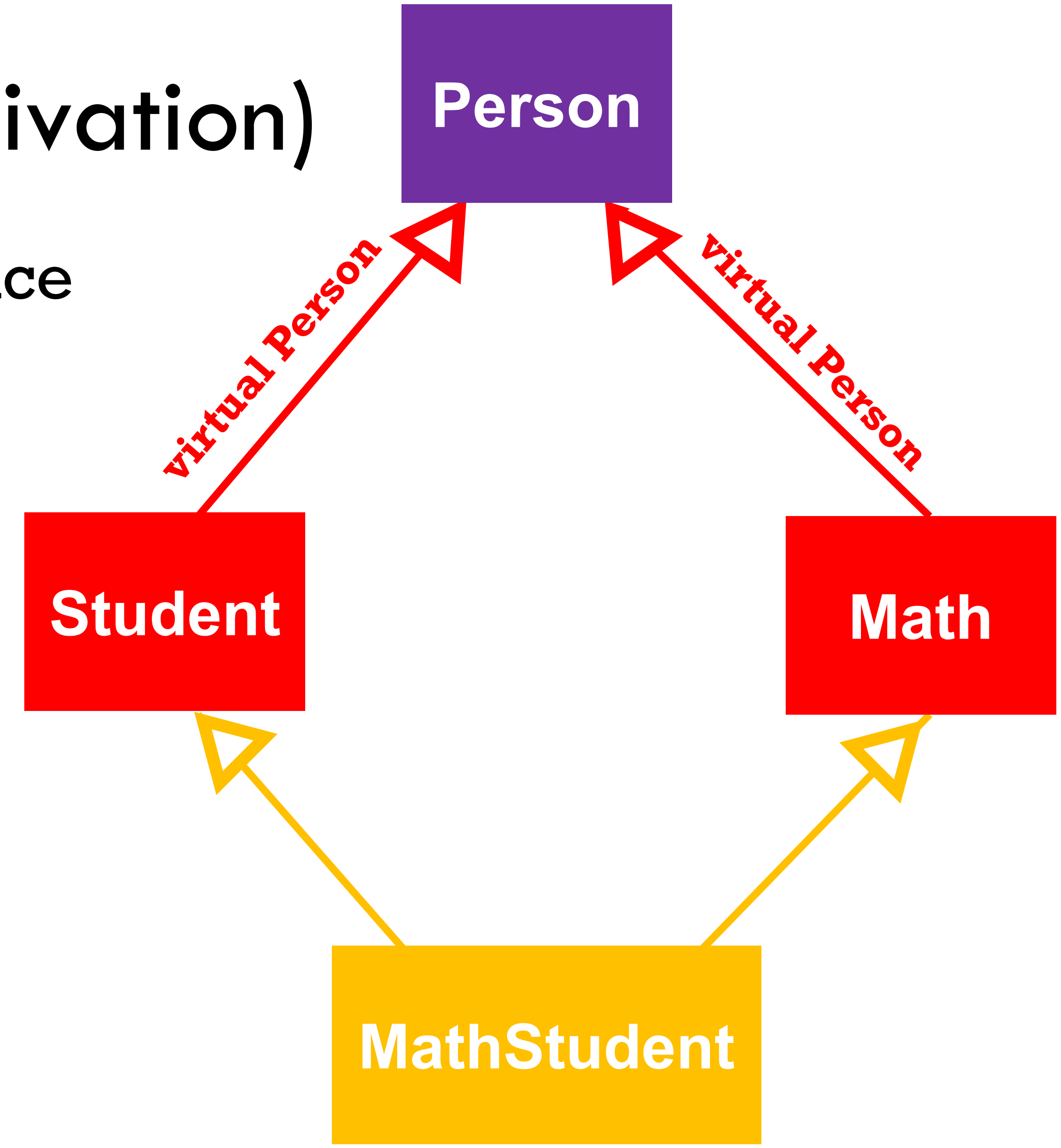
Virtual base classes (motivation)

- By adding **virtual classes**, we get a nice diamond shape as our result

- In code it looks like

```
class Student : virtual public Person
{
… //class code
}

class Math : virtual public Person
{
… //class code
}
```



Virtual base classes

- Permit us to store members in common base super-classes only once
- Consider **oop_multi2.cpp**
- We denote **person** as a virtual base class of both **student** and **mathematician**
- **But our output is not quite what we want!**
- We lost the value of name even though both **student** and **mathematician** called the **person** constructor and passed a name

Virtual base classes

- It is a **derived class**' responsibility to call the **base class** constructor (or the compiler will insert a call to the default constructor)
- We only have 1 version of the **person** base class because both **student** and **mathematician** denote person as a virtual base class
- We can say that **mathematician** and **student** no longer contain the **person** data – they refer to a common object that is part of the **most derived class** **math_student**

Most derived class

- In the case of virtual base classes, it is the responsibility of the most **derived class** (**math_student**) to call the shared **base-class** constructor (**person**)

```
math_student(const string& name,
             const string& passed,
             const string& proved)
: person(name), student(passed), mathematician(proved)
```

- The **person** constructor calls in **mathematician** and **student** are **disabled** when they are indirectly called from a **derived class**

```
mathematician(const string& name, const string&
proved)
  : person(name), proved(proved)
```

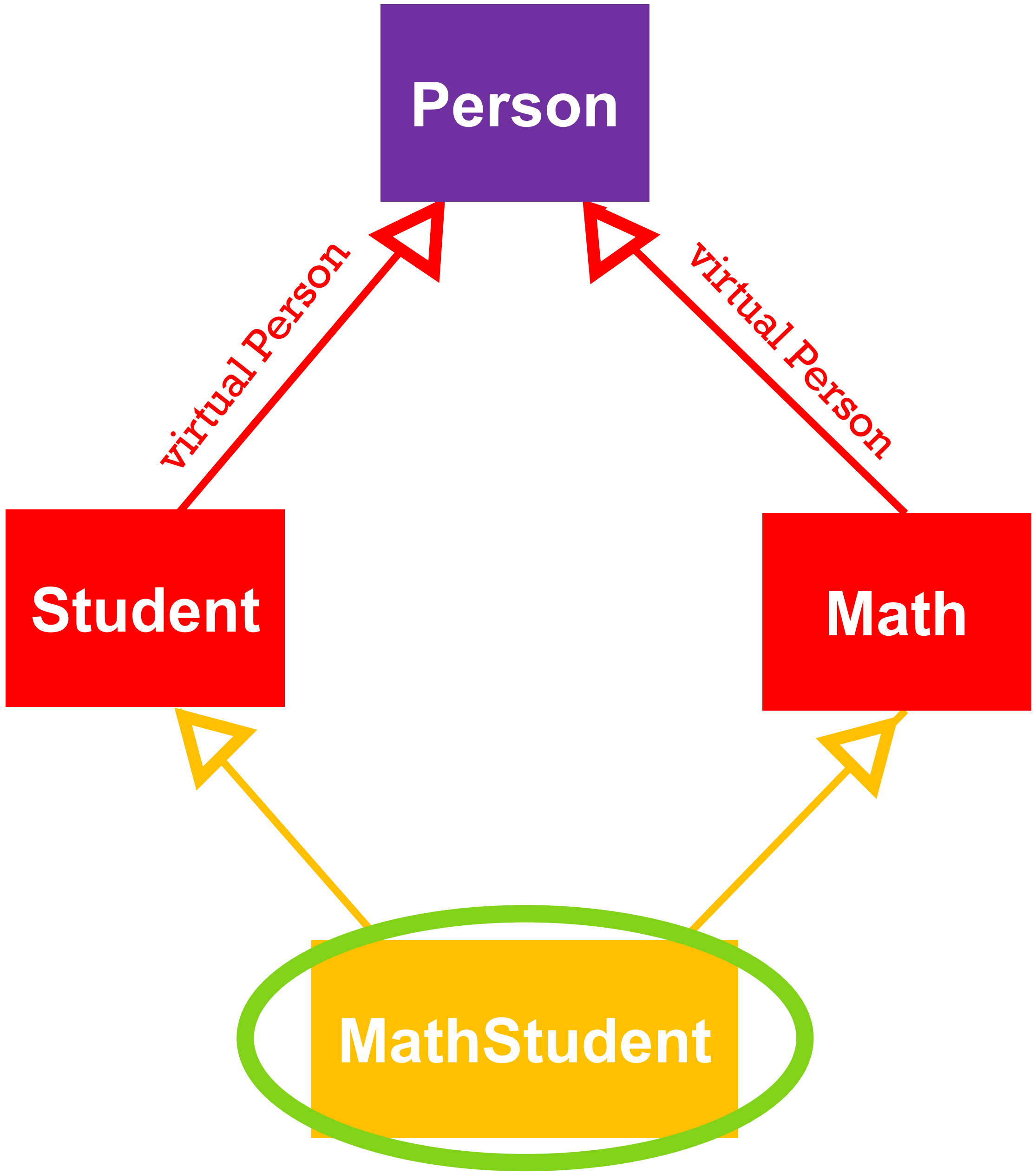
```
student(const string& name, const string&
passed)
  : person(name), passed(passed)
```

Multiple Inheritance

- Problem
  - I want to instantiate **MathStudent** but **Student** & **Math** can not call **Person**'s constructor.
  - Because **Person** is now virtually inherited by **Student** & **Math**

MathStudent ms; //instantiate **MathStudent**

- Solution
  - The most derived child (**MathStudent**) is now responsible for calling base class' constructor (**Person**)
  - **MathStudent** must call **Person**'s constructor during **MathStudent**'s construction

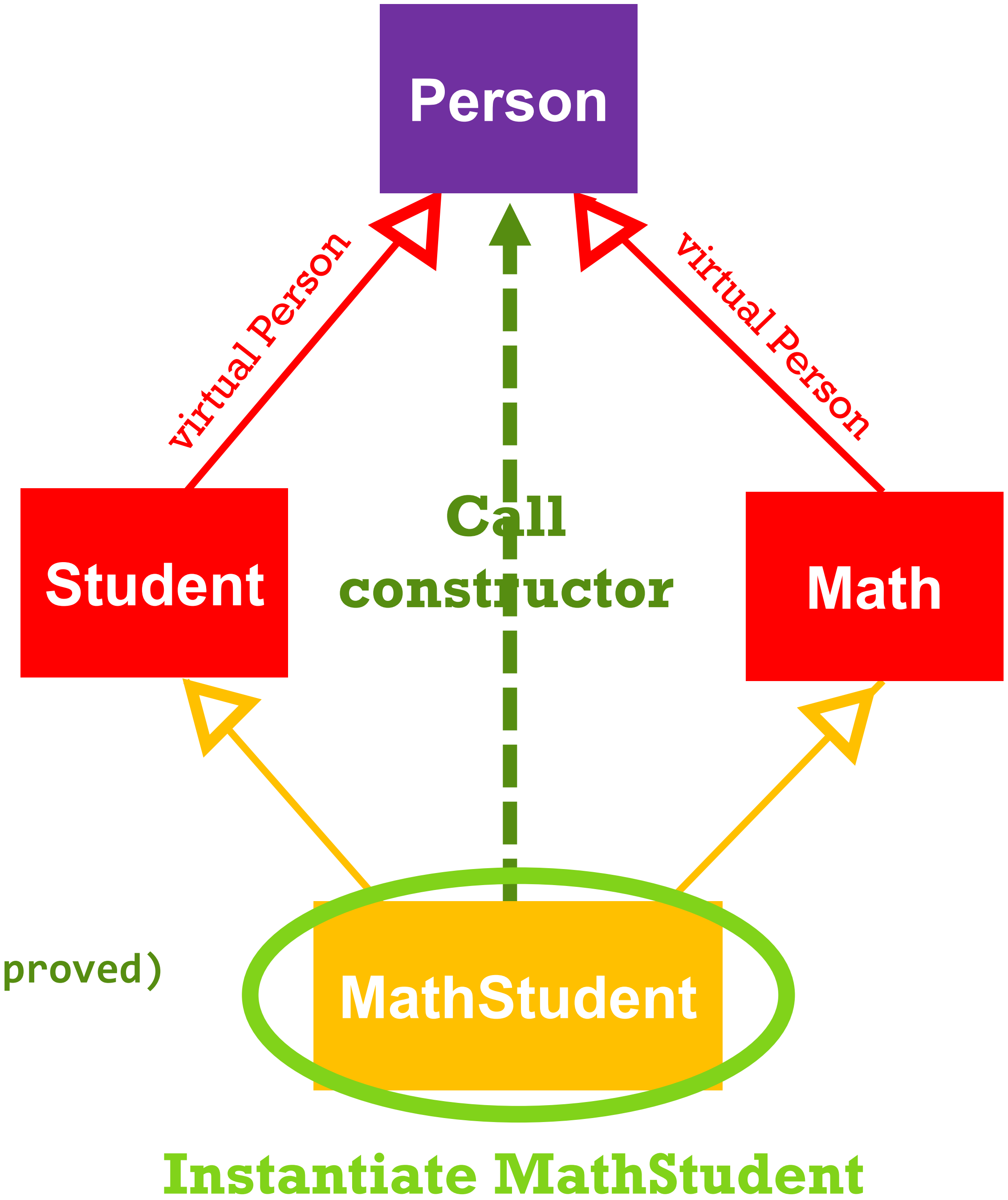


Instantiate MathStudent

Multiple Inheritance

- Solution
  - **MathStudent** will implicitly call **Person**'s default constructor
  - But you can write code in **MathStudent**'s constructor to call any constructor in **Person**

```
math_student(const string& name,
             const string& passed,
             const string& proved)
: person(name), student(passed), mathematician(proved)
```

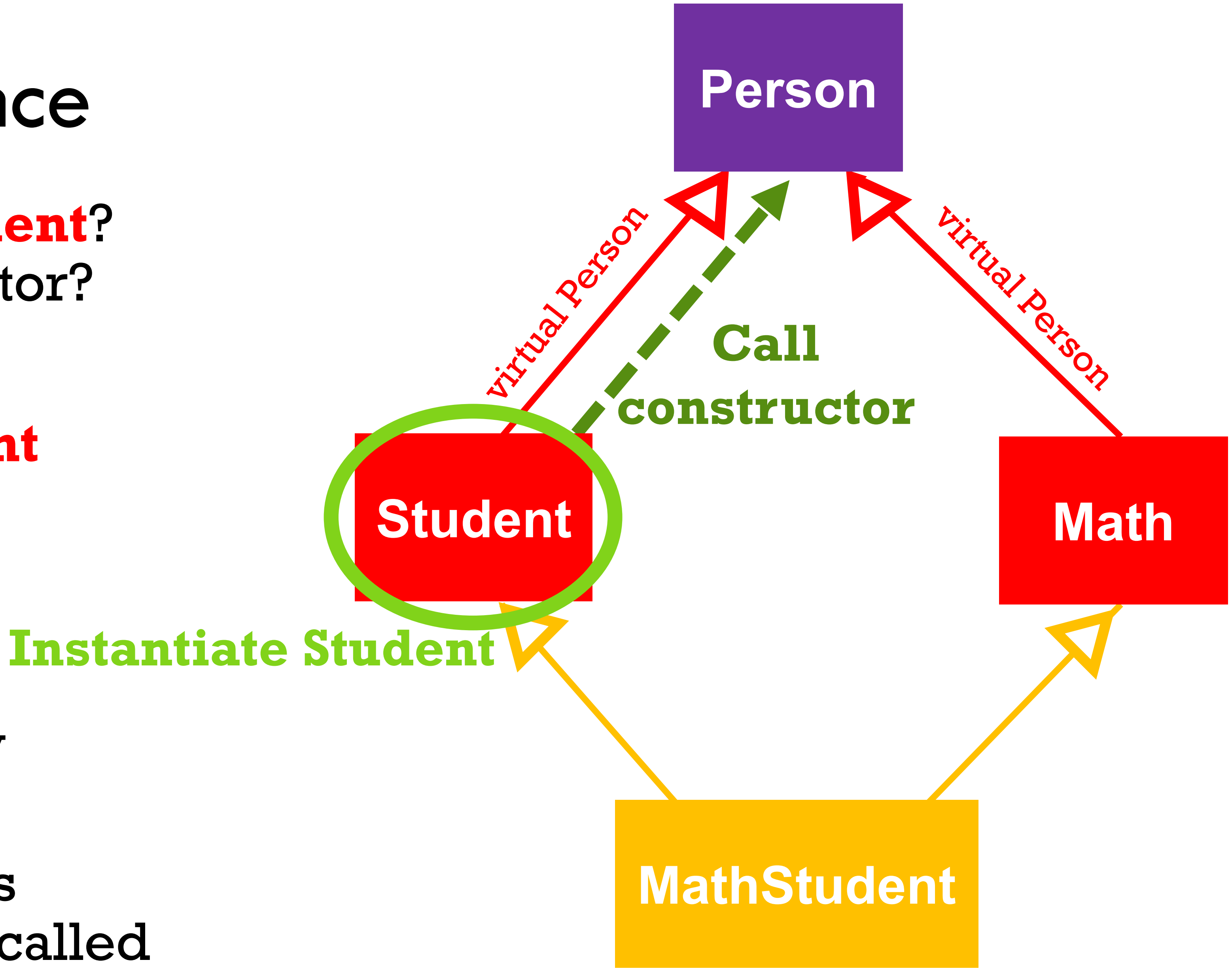


Multiple Inheritance

- How about instantiating **Student**? Can it call **Person**'s constructor?

Student s; //instantiate **Student**

- YES!
- **Student** can call **Person**'s constructor when it's directly instantiated
- **Student** can not call **Person**'s constructor only if it's being called through **MathStudent**



oop_multi3.cpp, diamond.cpp, diamondmethods.cpp

Activity

Midterm Practice questions

What is a copy constructor? When is it used? How can it be invoked in code?

What is a reference? How is it different from a pointer?

In C++, we may pass arguments to functions by value, pointer, or reference. What is the difference? Why would we choose one over the other?

What is the difference between static and dynamic allocation? Provide code examples.

What is a memory leak? How we prevent leaks in C++? Write a short function to demonstrate code that generates leaks. Add code that fixes the leak and underline it

Agenda

1. Friends
2. Operator overloading
3. Copy Assignment operator

COMP 3522

FRIENDS

Friends

- Grants a function (or another class) **access to private and protected members**
- The **friend declaration** appears in a class body
- The **definition** appears outside of the class body

```cpp
class Dollar {
   private:
     int num; //private
     friend Dollar sum(const Dollar &d1, const Dollar &d2);
   public:
     Dollar(int d) : num(d){};
}
```

```cpp
//main.cpp
Dollar sum(const Dollar &d1, const Dollar &d2)
{
   return Dollar(d1.num + d2.num);
}

Dollar d1{5};
Dollar d2{7};
Dollar dollarSum = sum(d1,d2);
```

dollar.cpp

Friends - Functions

Friends - Classes

```
class Spy; //forward declaration
class Boss {
      friend class Spy;
      int pin; //private
   public:
      Boss(int p) : pin(p){}
};
class Spy {
      int pin; //private
   public:
      Spy(int p) : pin(p){}
      void print(Boss b) {
         cout << b.pin;
      }
};
```

```
//main.cpp
Boss boss{1111};
Spy spy{2222};
spy.print(boss);
```

FriendClass.cpp

Friendship

- **Not** transitive
  - Transitive example: A < B and B < C that means A < C //YES
  - Boss friend Spy, Spy friend Minion, Boss friend Minion? //**NO**
- **Not** inherited
- Access specifiers have no effect (**friends can be in the private, protected, or public section**)

OPERATOR OVERLOADING

Operator overloading

- We can customize C++ operator for operands of user-defined types
- We can overload any of the following 38 operators:

+, -, *, /, %, ^, &, |, ~, !, =, <, >, +=, -=, *=, /=, %=, ^=,
&=, |=, <<, >>, >>=, <<=, ==, !=, <=, >=, &&, ||, ++, --,
,(the comma operator), ->*, ->, ( ) , [ ].

No, you don’t have to memorize this list

Basic rules of operator overloading

- Adhere to the operator's commonly known semantics
- If you provide one operation from a **set of operations**, you must provide them all:
  - If you overload **+**, you should overload **+=**
  - If you overload the prefix operator, overload the postfix operator too. **(++i**, **i++**)
- When the meaning of an operator is not obviously clear, it should not be overloaded

Operator overloading

- Operators are overloaded in the form of **functions with special names**
  - **operator+=()**, **operator-=()**, **operator<=()**
- Can be implemented as:
  - Member function of the **left operand**'s type. **Unary operators**
    - `objA += 10`, `objA -= objB`, `objA++`
  - Friendly non-member function
    - `objA <= 10`, `objA < objB`

- If a non-member function must access private members of the class, it must be declared a **friend**

Canonical form: **insertion operator**

- Most commonly overloaded operator
- Should be implemented as **friendly non-member function**`cout <<`

What looks easier?

`cout << myObject.getInfo();` OR `cout << myObject;`

```
friend std::ostream& operator<<(std::ostream& os, const T& obj)
{
    os << obj.myString; // write obj to stream
    return os;
}
```

```cpp
class Date
{
    int mo, da, yr;
public:
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }

    friend ostream& operator<<(ostream& os, const Date& dt);
};

ostream& operator<<(ostream& os, const Date& dt)
{
    os << dt.mo << '/' << dt.da << '/' << dt.yr;
    return os;
}

int main()
{
    Date dt(5, 6, 92);
    cout << dt; // 5/6/92
    cout << 77; // 77
}
```

https://msdn.microsoft.com/en-us/library/1z2f6c2k.aspx

Canonical form: **extraction operator**

```
friend std::istream& operator>>(std::istream& is, T& obj)
{
    is >> obj.myVar; // read obj from stream (up to you
                                    how!)
    return is;
}
```

```cpp
class Date
{
    int mo, da, yr;
public:
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }

    friend istream& operator>>(istream &input, Date &dt);
};

istream& operator>>(istream &input, Date &dt) {
        input >> dt.mo >> dt.da >> dt.yr;
        return input;
}

int main()
{
    Date dt(0, 0, 0);
    cin >> dt;
}
```

```cpp
class Date
{
    int mo, da, yr;
public:
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }

    friend istream& operator>>(istream &input, Date &dt)
    {
         input >> dt.mo >> dt.da >> dt.yr;
         return input;
    }
};

int main()
{
    Date dt(0, 0, 0);
    cin >> dt;
}
```

Canonical form: **comparison operators**

- Should be implemented as **friendly non-member** functions
- C++'s standard library contains helpful algorithms and types that will always expect **operator<** to be present
- There are six we should usually define:
    1. **==**
    2. !=
    3. **<**
    4. >
    5. <=
    6. >=

Canonical form: comparison operators first 3

1.
```
friend bool operator==(const X& lhs, const X& rhs)
{ /* do actual comparison */ }
```

2.
```
friend bool operator!=(const X& lhs, const X& rhs)
{
  return !operator==(lhs,rhs); //or return !(lhs == rhs);
}
```

3.
```
friend bool operator< (const X& lhs, const X& rhs)
{ /* do actual comparison */ }
```

Canonical form: comparison operators next 3

```
4. friend bool operator> (const X& lhs, const X& rhs)
   {
      return  operator< (rhs,lhs); //or return rhs < lhs;
   }
```

Canonical form: comparison operators next 3

How can we use the previously implemented operators, **less than**, **not**, **greater than**, to rewrite:

```
lhs <= rhs
```

lhs is less than or equal to rhs

lhs is not greater than rhs

```
5. friend bool operator<=(const X& lhs, const X& rhs)
   {
       return !operator> (lhs,rhs); //or return !(lhs > rhs);
   }
```

Canonical form: comparison operators next 3

How can we use the previously implemented operators, **less than**, **not**, **greater than**, to rewrite:

```
lhs >= rhs
```

lhs is greater than or equal to rhs

lhs is not less than rhs

```
6. friend bool operator>=(const X& lhs, const X& rhs)
   {
       return !operator< (lhs,rhs); //or return !(lhs < rhs);
   }
```

Unary **increment** and **decrement** (--,++)

- Exist in both prefix and postfix forms (like Java, C, etc.)
- Postfix always accepts a **dummy (unused) int argument** so we can tell them apart
- If you overload **increment**, ensure you overload prefix (++i) and postfix (i++) versions
- If you overload **decrement**, ensure you overload prefix (--i) and postfix (i--) versions
- Are **member functions**

Canonical form: increment operator

```
class Counter {
	Counter& operator++() { // Prefix: ++counter
		// do actual increment
		return *this;
	}
	Counter operator++(int) { // Postfix: counter++
		Counter tmp(*this); //copy original value
		operator++(); //internal increment
		return tmp; //return non incremented original
				value
	}
};
```

Canonical form: increment operator

- Note that postfix is defined in terms of prefix
- Postfix performs an extra copy
- So **postfix is slightly slower**
- That's why you might see this in C++:

```
for (int i = 0; i < upperBound; ++i) {
    // Do stuff
}
```

Binary arithmetic operators

- If you overload +, overload +=
- If you overload -, overload -=…

- operator**+** doesn't change left argument, should be a **friendly non-member**
- operator**+=** changes left argument, should be a **member function**
  - int num = 0;
  - int num2 = 0;
  - int sum = **num** **+** num2;
  - **num2** **+=** 5;

- And note that <u>operator+ is defined in terms of +=</u>

Canonical form: addition operator

```
class Fraction {
     Fraction& operator+=(const Fraction& rhs) {
          // actual addition of rhs to *this
          return *this;
     }
};
//main.cpp
Fraction fraction1(5.5);
Fraction fraction2(1.1);
fraction1 += fraction2;
```

Canonical form: addition operator

```
friend Fraction operator+(Fraction lhs, const Fraction& rhs)
{
      lhs += rhs;
      return lhs;
}
```

```
lhs += rhs; //lhs equals Fraction(6.6)
              but a copy
```

```
//main.cpp
Fraction x(5.5);
Fraction y(1.1);
Fraction z = x + y;
```

```
return lhs; //Fraction(6.6) assigned
              to z
```

Addition operator: some notes

```
Fraction& operator+=(const Fraction& rhs)
friend Fraction operator+(Fraction lhs, const Fraction& rhs)
```

Did you notice that:

- operator+= returns its result by reference
- operator+ returns a copy of its result

Why?

- When we write a + b, we expect the result to be a new value, which is why operator+ returns a new value
- Note that operator+ accepts the left parameter as a copy

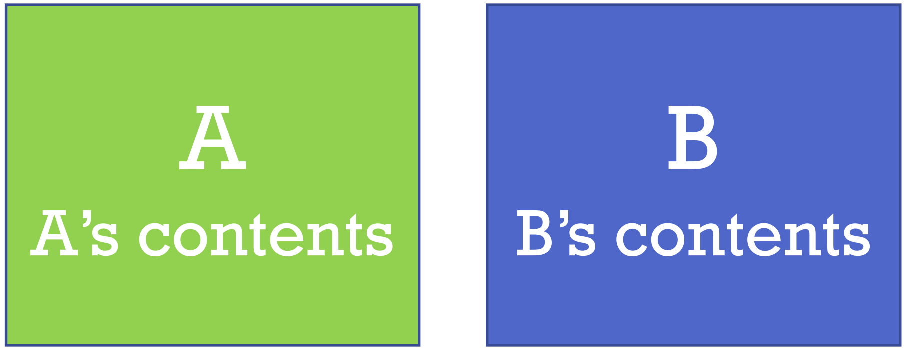
operators.cpp

COPY ASSIGNMENT OPERATOR (=)

Copy-and-swap idiom

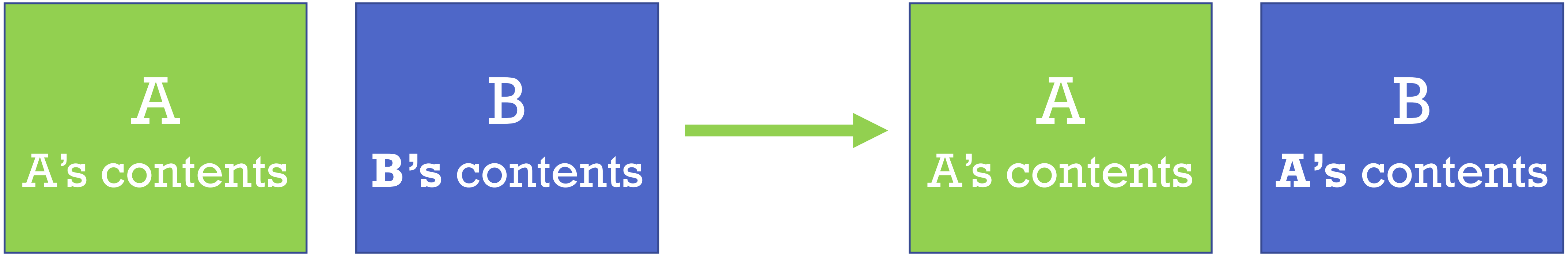
Overload operator=
Copy assignment operator

```
MyClass A;
MyClass B;
```



Overload operator=
Copy assignment operator

```
MyClass A;
MyClass B;
B = A; //A contents copied to B
```



Canonical form: assignment operator

- The assignment operator uses the **copy-and-swap** idiom

This is a **member function**

```
MyClass& MyClass::operator=(MyClass rhs)
{
    mySwap(*this, rhs);
    return *this;
}
```

What is copy and swap?

- **Avoids code duplication**

1. Use the **copy constructor** to create a local copy of the original object
2. Acquire the copied data with a **swap function**, swapping old data with new data
3. Temporary **local copy is destroyed**, taking the old data and leaving us with the new data in destination

Copy and swap

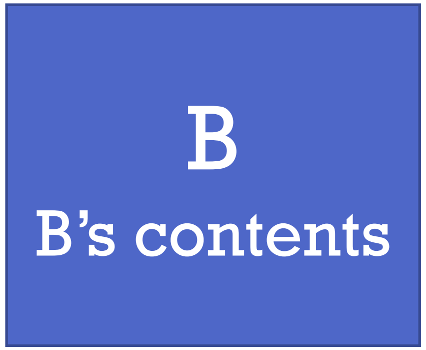
- So what do we need?
  1. Working copy constructor
  2. Working destructor
  3. A swap function.

- The swap function must be a function that does not throw any exceptions and does swap all data members
- Don't use **std::swap** to copy the ENTIRE object– it uses the **copy constructor** and the **copy assignment operator** so we'd have another recursive compiler spiral.

Finished assignment operator for Example

```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

```
//main.cpp
Example A;
Example B;
```



Finished assignment operator for Example

Pass by value – uses your copy constructor to create a copy named “other”

```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

```
//main.cpp
Example A;
Example B;
B = A;
```

Finished assignment operator for Example

```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

```
//main.cpp
Example A;
Example B;
B = A;
```

Finished assignment operator for Example

```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

```
void mySwap(Example& first, Example& second)
{
    using std::swap;
    swap(first.size, second.size); //use std::swap
    swap(first.my_list, second.my_list); //use std::swap
}
```

Finished assignment operator for Example

```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

```
//main.cpp
Example A;
Example B;
B = A;
```

Finished assignment operator for Example

```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

```
//main.cpp
Example A;
Example B;
B = A;
```

A
A's contents

~~other~~
~~B's contents~~

other destructor invoked
when leaving function scope

B
A's contents

Copy and swap example page 1 of 4

```
class Example {
  private:
    size_t list_size;
    int * my_list;
  public:
    Example(size_t size = 0) // default ctr
      : list_size{size},
        my_list{size ? new int[size] : nullptr}
       {}
```

Copy and swap example page 2 of 4

```
public:
  Example(const Example& other) // copy ctr
    : list_size{other.size},
      my_list{size ? new int[size] : nullptr}
  {
    // A loop here to copy the data…
  }
```

Copy and swap example page 3 of 4

```
public:
  ~Example() // destructor
  {
     delete[] my_list;
  }
```

Copy and swap example page 4 of 4

```
public:
    void mySwap(Example& first, Example& second)
    {
       using std::swap;
       swap(first.size, second.size); //using std::swap
       swap(first.my_list, second.my_list); //using std::swap
    }
```

Finished assignment operator for Example

```
    Example& operator=(Example other)
    {
        mySwap(*this, other);
        return *this;
    }
}; // Now we are at the end of Example class
```

Think of assignment as replacing the object's old state with a copy of some other object's state

operators.cpp

Member or non-member function?

1. If it is a **unary** operator, it should be implemented as a *member function* (++,--,()) //x++;
2. If it is a **binary** operator that treats both operands **equally** (it leaves them unchanged) it should be a *non-member function* (+, -,<,>) //x + y
3. If it is a **binary** operator that does NOT treat both operands equally, it should be implemented as a *member function* of the left operand's type (+=, -=) //x += y

Member or non-member function?

Operator	Typically Overloaded As	Why?
Unary operators (**!, ~, ++, --**)	Member function	Operates on a single object (self).
Assignment operators (**=, +=, -=**)	Member function	Modifies the left-hand object.
Comparison operators (**==, !=, <, >**)	Non-member function (friend)	Symmetric comparison between two objects.
Arithmetic operators (**+, -, *, /**)	Non-member function (friend)	Allows flexibility for non-class types (e.g. int + MyClass).
Stream operators (**<<, >>**)	Non-member function (friend)	Needs to work with std::ostream or std::istream.

chatgpt.com

FINAL NOTES

1. You now have most of the information you need to finish the first assignment.
2. NO LATE SUBMISSIONS.