COMP 3522

Object Oriented Programming in C++

Week 4 Day 1

Agenda

1. Destructors
2. Friends
3. Function overloading
4. Operator overloading
5. Copy Assignment operator

COMP 3522

DESTRUCTOR

Standard C++ class member functions

So far:

1. Default constructor
2. Copy constructor

Next:

The **destructor**.

Destructor

- Member function (of a class)
- **Purpose: to free resources the object acquired during its lifetime**
- Invoked when the lifetime of an object ends
  - Program termination (for statics)
  - End of scope
  - Explicitly call delete, delete[]

Destructor

- The destructor is the complementary operation of the default constructor
- It uses the notation for the complement: ~

```
class complex
{
     public:
     ~complex() { cout << “Destroyed! << endl; }
     …
}
```

Destructor implementation rules

We will return to these in the next few weeks (remind me to tell you why!):

1. **Never throw exceptions from a destructor** (we will learn about C++ exceptions soon!)
2. **If a class contains a virtual function, the destructor should be virtual too**

Destructor example

```
class vector
{
    public:
        ~vector() { delete[] data; }
    …
    private:
        unsigned vector_size;
        double * data;
};
```

Mini-review: dynamic memory

- **new** calls the constructor
- **delete** calls the destructor
- If we use [] when allocation (with new), use [] when deleting
- Assuming C is a class:

```
C * c = new C; //MEMORY LEAK – NOT DELETED
C * p1 = new C(1);
C * p2 = new C[100];
delete p1;
delete[] p2;
```

Some fun examples

```
C a; // Static allocation (default constructor)
C b(a); // Copy constructor
C c = b; // Copy constructor
a = b; // Assignment operator
C d[10]; // Default constructor x 10
```

Some fun examples

```
C * p = new C; // Default constructor
C * q = new C(a); // Dynamic allocation (copy
                  // constructor)
C * r = new C[5]; // Default constructor x 5
delete p; // Destructor
delete q; // Destructor
delete[] r; // Destructor x 5
```

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

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

```
//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 section or the public section**)

FUNCTION OVERLOADING

Function overloading

- More than one function with the **same name in the same scope**
- Overloading considerations:
  1. Number of arguments
  2. Type of arguments
  3. const (when applied to the entire function)

We do **not** consider the return type

Function overloading

```
//functions
void print(int i) {...};
void print(double f) {...};
void print(char *c) {...};

int main() {
print(5);
print(500.263);
print("Hello C++");
…}
```

So we have a conundrum

- When we invoke a function, we may need to select the best match from multiple candidate functions in a given scope.
- We must look for the “most suitable” function
- How do we pick it?
- We have to rank the candidate functions with respect to each parameter
- We have to rank the “conversions”
- We look for the “most suitable” function

What is the conversion sequence? (page 1)

1. Exact match
    1. Without qualifications
    2. Trivial conversion
2. Match using promotions
    1. char, unsigned char, short to int
    2. Short to int (widening to an **int**)
    3. float to double

What is the conversion sequence? (page 2)

3. Match using standard conversions
    1. Integral (signed to unsigned, unsigned to signed)
    2. Floating point (signed to unsigned, unsigned to signed)
    3. Integral to floating point
    4. Pointer conversion

4. Now choose the best match, or throw a compiler error!

https://en.cppreference.com/w/cpp/language/implicit_conversion#Order_of_the_conversions

What are “trivial” conversions

type_name to type_name&

```
int num = 5;
doSomething(num);
```

```
doSomething(int& param) {…}
```

type_name& to type_name

```
int num = 5;
int& numRef = var;
doSomething(varRef);
```

```
doSomething(int param) {…}
```

What are “trivial” conversions

- type_name[ ] to type_name*
- type_name to const type_name
- type_name* to const type_name*

conversion.cpp*

Function overloading - How does this work

1. The compiler creates a set of "**candidate**" functions

```
//functions
void print(int i) {…}
void print(int i, int j) {…}
void print(int i, string s) {…}

int main() {
print(5,"Dog");
…}
```

Function overloading - How does this work

1. The compiler creates a set of "**candidate**" functions

```
//functions
~~void print(int i) {…};~~
void print(int i, int j) {…}
void print(int i, string s) {…}

int main() {
print(5,"Dog");
…}
```

Function overloading - How does this work

2. Candidate functions are functions in which the **actual** argument in that position **can be converted** to the type of the **formal** argument

```
//functions
void print(int i, int j) {…}
void print(int i, string s) {…}

int main() {
print(5, “Dog”);
…}
```

Function overloading - How does this work

3. A set of "best matching functions" is built for each argument, and the selected function is the intersection of all sets.

```
//functions
void print(int i, int j) {…}
void print(int i, string s) {…}

int main() {
print(5,"Dog");
…}
```

```
//functions
~~void print(int i, int j) {…}~~
void print(int i, string s) {…}

int main() {
print(5, "Dog");
…}
```

Function overloading - How does this work

4. If the intersection size > 1, the overloading is ambiguous and the compiler generates an error.

```
//functions
void print(int i, int j) {…}
void print(int i, string s) {…}

int main() {
print(5,“Dog”);
…}
```


```
//functions
~~void print(int i, int j) {…}~~
void print(int i, string s) {…}

int main() {
print(5, “Dog”);
…}
```

Suppose we are adding some fractions

```
fraction& add( fraction& f, long l );          // Version 1
fraction& add( long l, fraction& f );          // Version 2
fraction& add( fraction& f, fraction& f );  // Version 3

fraction fraction_one;
fraction_two = add( fraction_one, 22 );
```

WHICH VERSION GETS USED?

Here's what the compiler does

The compiler builds 2 sets:

Set 1: Candidate functions whose first argument is type fraction	Set 2: Candidate functions whose second argument can be converted to type int
Version 1	**Version 1** (because int can be converted to long using standard conversion)
Version 3	

INTERSECTION = VERSION 1

What does an ambiguous call look like?

```
fraction& add( fraction& f, long l );          // Version 1
fraction& add( long l, fraction& f );          // Version 2
fraction& add( fraction& f, fraction& f );  // Version 3

fraction_two = add( 2510, 3469 );
```

WHICH VERSION GETS USED?

Here's what the compiler does

The compiler builds 2 sets:

Set 1: Candidate functions whose first argument is type int	Set 2: Candidate functions whose second argument can be converted to type int
Version 2 (because int can be converted to long using standard conversion)	**Version 1**(because int can be converted to long using standard conversion)

NO INTERSECTION = COMPILER ERROR!

What about const

- Function arguments of a const **type** **ARE NOT** treated differently from function arguments of a type

```
void functionA(int a)
{
}

void functionA(const int a)
{
}
```

COMPILER ERROR

What about const

HOWEVER…

- Function arguments of a const **reference** **ARE** treated differently from function arguments of a non-const reference

```
void constRef(const int& a)
{
}

void constRef(int& a)
{
}
```

```
const int varA = 5;
constRef(varA);
```



```
int varA = 5;
constRef(varA);
```

What about const

- And member functions with the same arguments are treated differently if one of the functions is const

```
void Circle::doSomething(int a)
{
}

void Circle::doSomething(int a) const
{
}
```

Circle c;
c.doSomething(5);

const Circle c;
c.doSomething(5);

Const overloading example

```
class container
{
  public:
    container() { cout << "container default constructor\n"; }
    container( container &o ) { cout << "container&\n"; }
    container( const container &co ) { cout << "const container&\n"; }
};

int main()
{
    container o1;           // Calls default constructor
    container o2( o1 );     // Calls container( container& )
    const container o3;     // Calls default constructor
    container o4( o3 );     // Calls container( const container& )
}
```

Function overloading example

1. `int f(int, int);`
2. `double f(double, double);`

```
float x{1.0};
long lint{2l};
f(1, 1);
f(x, lint);
f(x, 1);
```

Function overloading example

1. `int f(int, int);`
2. `double f(double, double);`

```
float x{1.0};
long lint{2l};
f(1, 1);    // Calls 1
f(x, lint); // Calls 2
f(x, 1);    // Ambiguous!
```

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
- Can be implemented as:
  - Member function of the left operand's type
  - Non-member function.

- 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**

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

```
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;
  cout << 5;
}
```

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;
}
```

```
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;
}
```

```
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

Another way to say

lhs <= rhs

lhs is less than or equal to rhs

lhs is ??? ???? ??? rhs

```
5. friend bool operator<=(const X& lhs, const X& rhs)
```

Canonical form: comparison operators next 3

Another way to say

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

Another way to say

lhs >= rhs

lhs is greater than or equal to rhs

lhs is ??? ??? ??? rhs

6. friend bool operator>=(const X& lhs, const X& rhs)

Canonical form: comparison operators next 3

Another way to say

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 and postfix versions
- If you overload **decrement**, ensure you overload prefix and postfix
- 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+ should be a **friendly non-member**
- operator+= changes the left argument, so should be a **member function**

- And note that operator+ is defined in terms of +=

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;
```

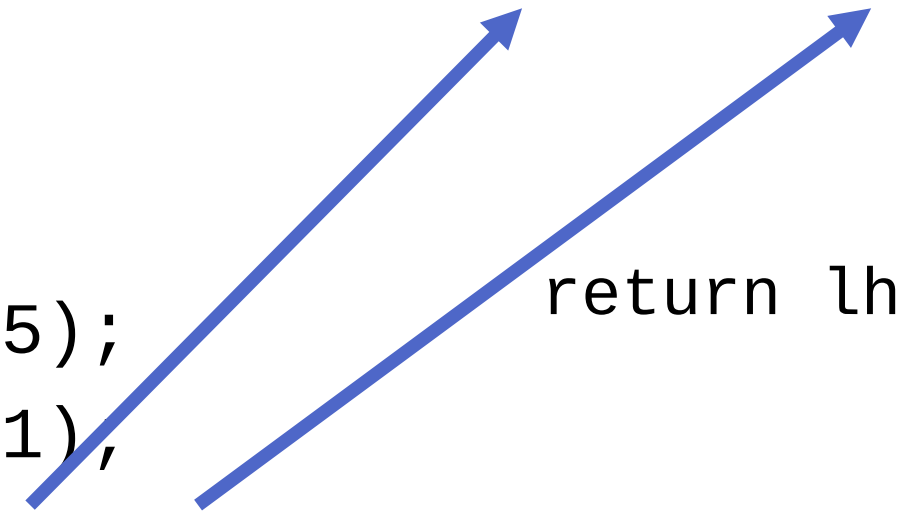
Canonical form: addition operator

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

```
lhs += rhs //lhs = 6.6 but a copy
```

```
return lhs; //6.6 assigned to z
```

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



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

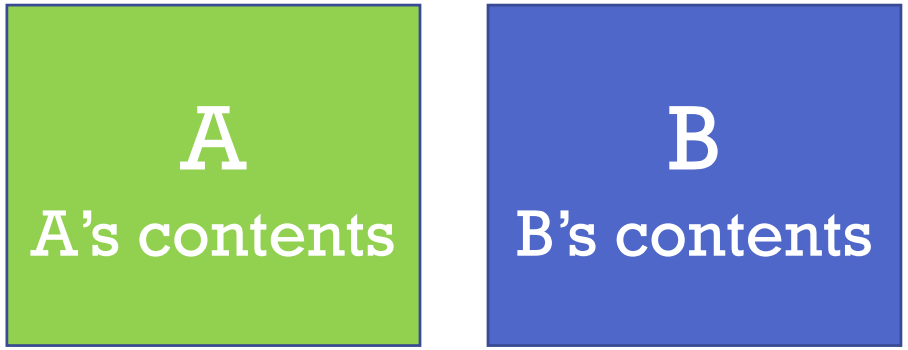
operators.cpp

COPY ASSIGNMENT OPERATOR (=)

Copy-and-swap idiom

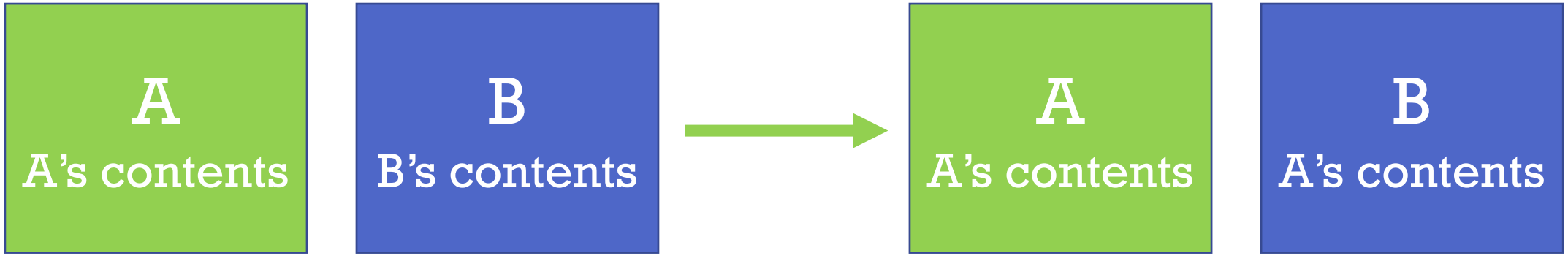
Overload operator =
Copy assignment operator

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



Overload operator =
Copy assignment operator

```
MyClass A;
MyClass B;
B = A; //B copies contents of A
```



Canonical form: assignment operator

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

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

This is a **member function**

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. Mic drop.

Copy and swap

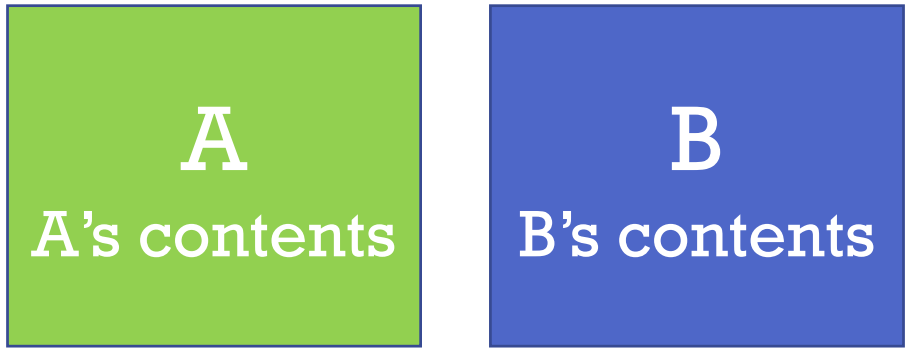
- So what do we need?
  1. Working copy constructor
  2. Working destructor
  3. A swap function.

- The swap function must be a function that does not throw any exceptions and does swap all data members
- Don’t use std::swap – 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;
//B = A;
```



Finished assignment operator for Example

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

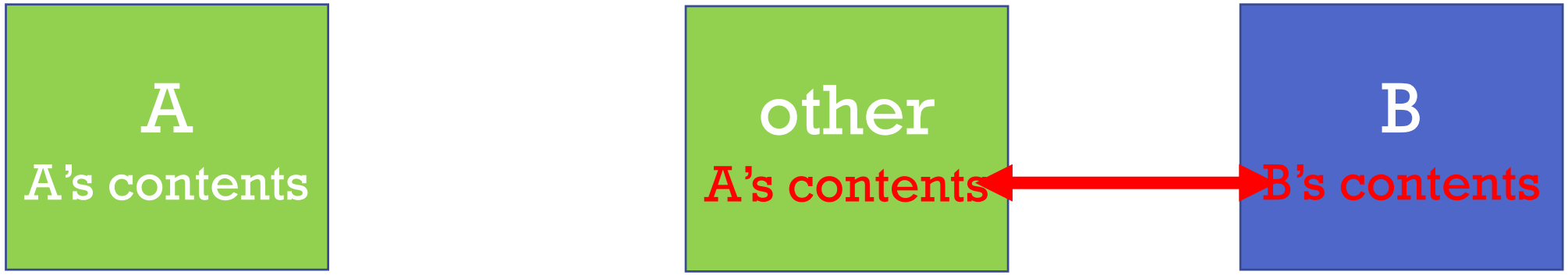
```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

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

Finished assignment operator for Example

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

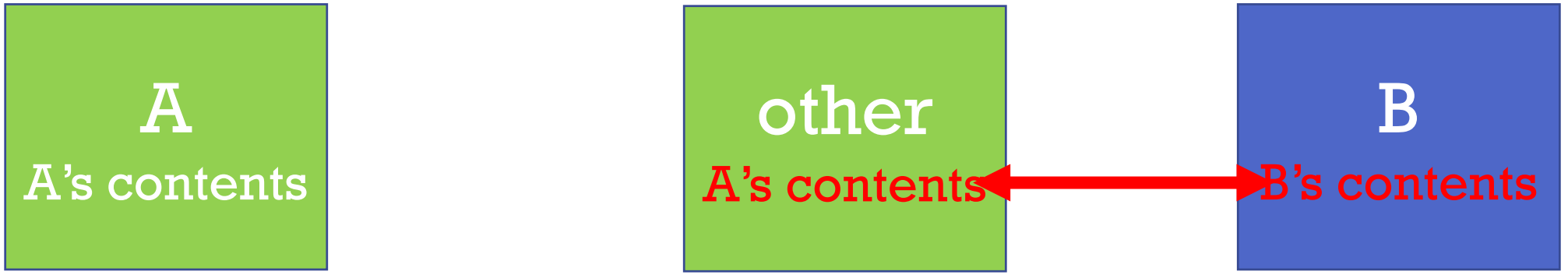
```
//main.cpp
Example A;
Example B;
B = A;
```



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); //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;
}
```

```
//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;
```

other destructor invoked
when leaving function scope

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; }
}; // End of class (or so we think)
```

Copy and swap example page 4 of 4

```
public:
  friend void mySwap(Example& first, Example& second);
}; // Now we are at the end of Example class

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;
}
```

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

Member or non-member function?

1. If it is a **unary** operator, it should be implemented as a member function (++,--,())
2. If it is a **binary** operator that treats both operands **equally** (it leaves them unchanged) it should be a non-member function (+, -,<,>)
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 (+=, -=)

FINAL NOTES

1. You now have all the information you need to finish the first assignment.
2. Remember it is due Sunday Oct 13th
3. NO LATE SUBMISSIONS.