COMP 3522

Object Oriented Programming in C++

Week 4, Day 2

Agenda

1. Copy Assignment operator
2. Copy Elision & Return value optimization
3. Inheritance
4. Polymorphism and virtual functions

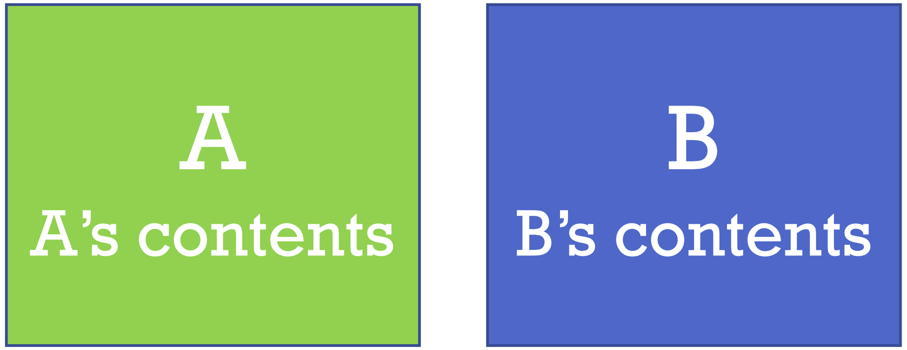
COMP 3522

COPY ASSIGNMENT OPERATOR (=)

Copy-and-swap idiom

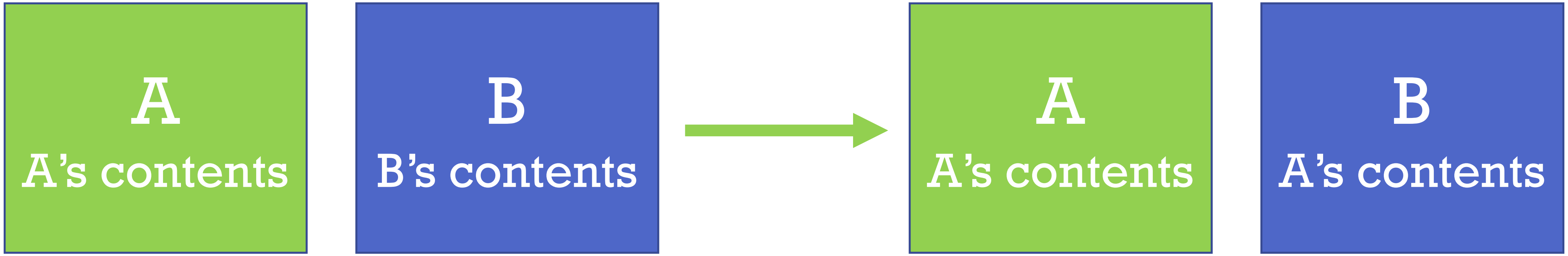
Overload operator =
Copy assignment operator

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



Overload operator =
Copy assignment operator

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



Canonical form: assignment operator

- 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. Mic drop.

Copy and swap

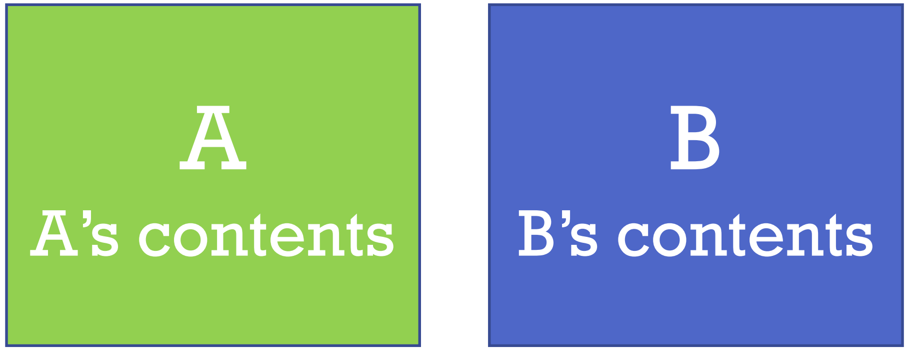
- So what do we need?
  1. Working copy constructor
  2. A swap function.
  3. Working destructor

- 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”

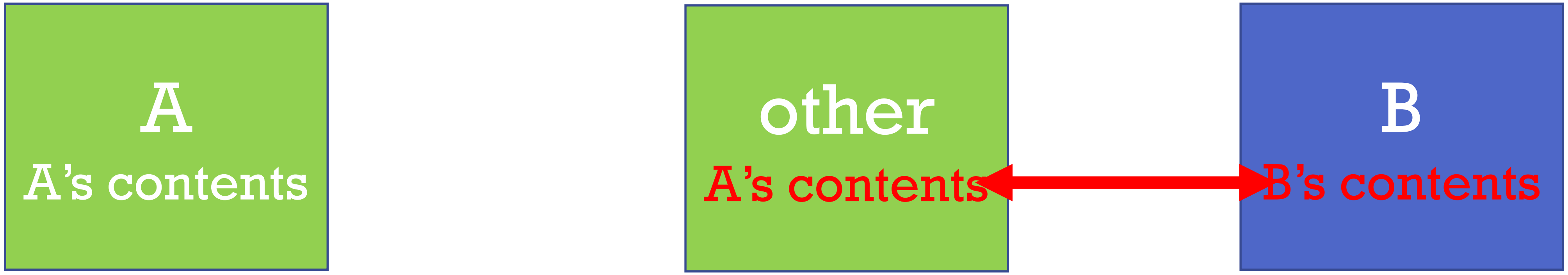
```
Example& operator=(Example other)
{
    mySwap(*this, other);
    return *this;
}
```

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

Finished assignment operator for Example

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

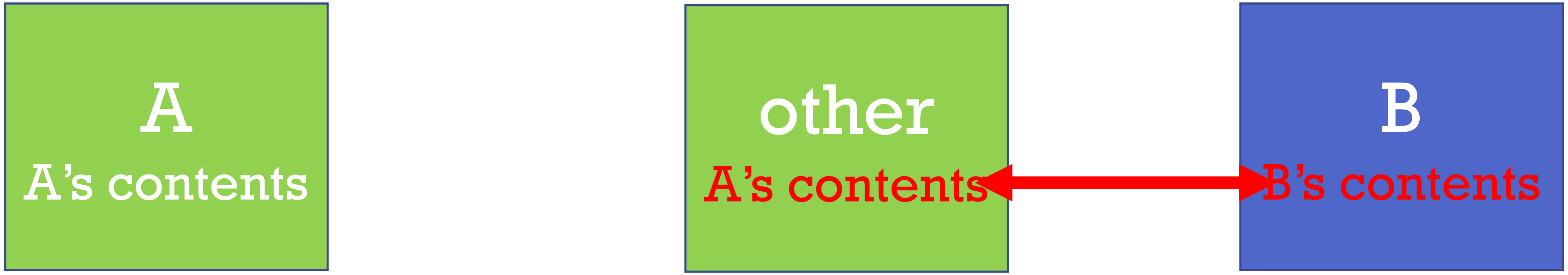
```
//main.cpp
Example A;
Example B;
B = A;
```



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

copySwap.cpp

Copy and swap example page 2 of 4

```
public:
  Example(const Example& other) // copy ctr
    : 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.

COPY ELISION & RETURN VALUE OPTIMIZATION

Copy Elision

Copy Elision (or copy omission) is a compiler optimization technique that avoids unnecessary copying of objects

https://www.geeksforgeeks.org/copy-elision-in-c/

Return value optimization (RVO)

```
//Number.cpp
class Number {
private:
	int num
public:
	Number(int n) : num (n) {}
	Number(const Number &n2)
	{num = n2.num;}
}
```

```
//main.cpp
Number createNumber(int num)
{
	return Number(num);
}

//main function
Number n = createNumber(5);
```

Return value optimization (RVO)

```
//Number.cpp
class Number {
private:
        int num
public:
        Number(int n) : num (n) {}
        Number(const Number &n2)
        {num = n2.num;}
}
```

```
//main.cpp
Number createNumber(int num)
{
        return Number(num);
}

//main function
Number n = createNumber(5);
```

How many copies of type Number are created?

Return value optimization (RVO)

```
//Number.cpp
class Number {
private:
	int num
public:
	Number(int n) : num (n) {}
	Number(const Number &n2)
	{num = n2.num;}
}
```

```
//main.cpp
Number createNumber(int num)
{
	return Number(num);
}

//main function
Number n = createNumber(5);
```

How many copies of type Number are created?

Return value optimization (RVO)

Return value optimization lets the compiler remove the temporaries by directly initializing n

Use memory space of **n**, outside function createNumber, to directly construct the object initialized inside the function and that is returned from it

Happens automatically, but requires returned object to be constructed on a return statement

```
//main.cpp
Number createNumber(int num)
{
	return Number(num);
}

//main function
Number n = createNumber(5);
```

Named return value optimization (NRVO)

Named return value optimization can remove the temporaries by directly initializing n even if the returned object is named

Note, there's no guarantee that all compilers use RVO and NRVO

```
//main.cpp
Number createNumber(int num)
{
        Number tempN(num);
        return tempN;
}

//main function
Number n = createNumber(5);
```

https://www.fluentcpp.com/2016/11/28/return-value-optimizations/

Copy Elision

```
//Number.cpp
class Number {
private:
        int num
public:
        Number(int n) : num (n) {}
        Number(const Number &n2)
        {num = n2.num;}
}
```

```
//main.cpp
Number createNumber(int num)
{
        return Number(num);
}

//main function
Number n (Number(2));
```

Which constructor is called after Number(2) temporary created?

Copy Elision

```
//Number.cpp
class Number {
private:
	int num
public:
	Number(int n) : num (n) {}
	Number(const Number &n2)
	{num = n2.num;}
}
```

Based on code

```
//main.cpp
Number createNumber(int num)
{
	return Number(num);
}

//main function
Number n (Number(2));
```

Copy Elision

```
//Number.cpp
class Number {
private:
        int num
public:
        Number(int n) : num (n) {}
        Number(const Number &n2)
        {num = n2.num;}
}
```

With copy elision

```
//main.cpp
Number createNumber(int num)
{
        return Number(num);
}

//main function
Number n (Number(2));
```

Copy Elision

Copy Elision (or copy omission) is a compiler optimization technique that avoids unnecessary copying of objects

In our example, the compiler optimized our call by avoiding the call to copy constructor and directly called regular constructor

https://www.geeksforgeeks.org/copy-elision-in-c/

Copy Elision + RVO

```
//Number.cpp
class Number {
private:
        int num
public:
        Number(int n) : num (n) {}
        Number(const Number &n2)
        {num = n2.num;}
}
```

```
//main.cpp
Number createNumber(int num)
{
        return Number(num);
}

//main function
Number n = createNumber(5);
```

https://youtu.be/fSB57PiXpRw

copy_elision.cpp and basic.cpp

ACTIVITY

1. Let’s study the copy constructor a little bit more.
2. Let’s learn about **copy elision** and **return value optimization**.
3. Follow the instructions on the next page.

Return value optimization

- The compiler will often elide copies when we are creating an object and returning it by value from a function.
- Let’s study the copy constructor a little bit more.
- Let’s learn about **copy elision** and **return value optimization**.
- Follow the instructions on the next page and be prepared to discuss your observations!

ACTIVITY

- Take a peek at copy_elision.cpp and basic.cpp
- Note:
  1. use of a **static** member variable inside the basic class
  2. initialization of the static member outside the class (it's a global!)
  3. function use_basic accepts an object by value, copies it, then returns a copy of the copy
- **Examine the output. Does it make sense? When is a copy constructor invoked?**

When is the copy constructor called?

1. When we invoke the copy constructor directly
2. When we pass an object by value to a function
3. When we return an object by value from a function.

HOWEVER!

Copy Elision and **Return Value Optimization**
may be used by some compilers to eliminate a temporary object created to hold a function's return value.

INHERITANCE

Inheritance

C++ implements everything we’ve seen in Java:

1. Inheritance
2. Polymorphism
3. Abstract classes and interfaces.

In C++ we talk about:

1. Base class
2. Derived class

Inheritance relationship

Java example:

class Shape { … }

class Circle **extends** Shape { … }

C++ example:

class Shape { … }

class Circle **: public** Shape { … }

Concept is the same

- Push common attributes as high into the inheritance hierarchy as possible
- Derived classes inherit all the accessible members of the base class
- Public access specifier may be replaced by private or protected in the derived class header
- This limits the most accessible level for the members inherited from the base class, i.e., public members may be protected in the base class

Access modifiers

Access	Public	Protected	Private
Members of the same class	yes	yes	yes
Members of a derived class	yes	yes	no
Not members	yes	no	no

What is inherited in C++?

- A publically derived class inherits access to everything **except**:
    1. Constructors *
    2. Destructor *
    3. Friends
    4. Private members.

* Not inherited per such, but they are automatically called by constructors and destructor of derived class

Which base class constructor gets called?

- We can **specify** which one to call in the derived class constructor (just like Java!).
- **In C++ the call to super looks like a member initialization list.**
- Pass the parameters to the base class constructor
- If we don't, the default constructor is called (just like Java!).

- Code Examples: whichconstructor.cpp and private.cpp

POLYMORPHISM AND VIRTUAL FUNCTIONS

What about pol·y·mor·phism

/ˌpäləˈmôrfizəm/

Noun

- from the Greek roots "poly" (many) and "morphe" (form, shape, structure)
- the condition of occurring in several different forms
- a feature of a programming language that allows routines to use variables of different types at different times.

What about polymorphism?

It's EASY! (I promise!)

A pointer to a derived class is type-compatible with a pointer to its base class.

This is just like Java (remember everything is a pointer in Java).

Code Example: polymorphism.cpp

But that area member function…

- There was no area member function in Shape
- Could not use a Shape pointer to ask a Rectangle or Triangle to generate the area

Q: How can we overcome this in C++?

A: Virtual members!

Virtual member

- A base class member function that can be redefined (Java: overridden) in the derived class

- Add the **virtual** keyword to the function declaration

- **Remember: non-virtual members of the derived class <u>cannot</u> be accessed through a reference of the base class**

Virtual member

- Permits a member of the derived class with the same name as the member in the base class to be appropriately called from a pointer
- A class that declares or inherits a virtual function is called a polymorphic class
- Permits dynamic binding aka late binding aka polymorphic method dispatch

Code Example: virtual.cpp

More about virtual functions

- Virtual specifies that a non-static member function supports dynamic binding
- Used with pointers and references
- A call to an overridden virtual function invokes the behaviour in the derived class
- We can invoke the original function by using the base class name and the scope operator (qualified name lookup)

Code Example: virtual2.cpp

Overriding functions

```
class Base
{
  virtual void f() { cout << "base\n"; }
};
class Derived : Base
{
  void f() override { cout << "derived\n";}
};
```

Overriding functions

```
class Base
{
  virtual void f() { cout << "base\n"; }
};
class Derived : Base
{
  void f() override { cout << "derived\n";}
};
```

Still works as virtual function, but risky

Overriding functions

```
class Base
{
  virtual void f() { cout << "base\n"; }
};
class Derived : Base
{
  void f(int a) override { cout << "derived\n";}
};
```

Compiler won't catch function parameter changed without override

Overriding functions

```
class Base
{
  virtual void f() { cout << "base\n"; }
};
class Derived : Base
{
  void f(int a) override { cout << "derived\n";}
};
```

//Compiler error warns function doesn't match virtual function in base

Notes

- A function with the same name but different parameter list does not override the base function of the same name, but ***hides*** it. This is BAD, polymorphism is broken

- We can prevent a **function** from being overridden by using the **final** keyword (just like Java!)
- We can prevent a **class** from being overridden by using the **final** keyword in the class definition.

Code Example: final.cpp