COMP 3522

Object Oriented Programming in C++

Week 7

Agenda

1. Static
2. Copy Elision & Return value optimization
3. Right left rule

COMP 3522

STATIC

Static declarator

- In C++, member variables can be static:
  - Only **one copy** per class exists
  - Permit a single resource to be **shared** between instances
  - **Independent static storage** for life of program
  - This can be useful for the Singleton design pattern (we'll look at it later, but the name is a hint)
  - **static keyword used with declaration** not definition

```
class X { static int n; }; // incomplete declaration
int X::n = 1; // definition
```

Accessing statics

- Two forms can be used:
  1. Qualified name

     Class::member
  2. Member access expression

     Class->member or Class.member

- Exist even if no objects have been defined

Initializing static

- **Can't** be initialized inside the class definition

```
class X {
  static int m = 5; //ERROR
  static int n; //OK

};

int X::n = 5; //OK
```

Static constants: const

- **Integer and Char can** be initialized with an initializer in which every expression is a **constant** expression right inside the class definition

```
class X {
  const static int m = 9; //fundamental types can be initialized inside
  const static int k;
  static const double * pointer; //non fundamental cannot be initialized
                                 inside
};
const int X::k = 3; //initialized outside
const double * X::pointer = new double[3]; //initialize outside
```

Static constants: constexpr

- **Must** be initialized with an initializer in which every expression is a **constant expression** right inside the class definition

```
class X {
  constexpr static int arr[] = { 1, 2, 3 }; // OK
  constexpr static std::complex<double> n = {1,2}; // OK
  constexpr static int k; // Error
};
```

Static member functions

- Can only access static data and invoke static member functions
- There is no *this pointer
- Cannot be virtual or const
  - `virtual static void staticFunction() //ERROR`
  - `static void staticFunction() const //ERROR`

static.cpp, static_functions.cpp

Applications?

- Classes that don't need to be instantiated to be used
- Static calculator
  - A class that contains a series of common math operations
  - Add, subtract, multiply, divide
  - Instead of instantiating a Calculator class, just call it statically

```
//available only within scope of c variable
Calculator c;
c.add(num1, num2);

//available globally
Calculator::add(num1, num2)
```

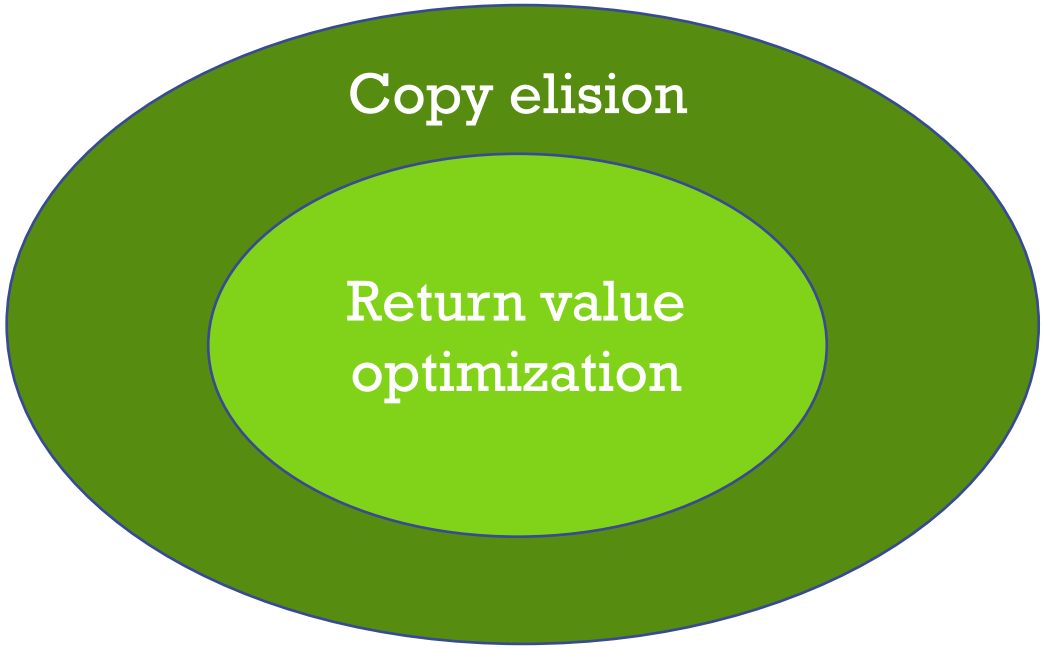
calculator.cpp & staticCalculator.cpp

COPY ELISION & RETURN VALUE OPTIMIZATION

Copy Elision & Return value optimization

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

Return value optimization is a subset of copy elision. **Avoids unnecessary copying of objects <u>returned from a function</u>**



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

Return value optimization (RVO)

```
//Number.hpp
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.hpp
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.hpp
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

Directly constructs the object initialized inside the createNumber function into the memory space of **n**

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 1 parameter 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

ACTIVITY

- The compiler will often elide copies when we are creating an object and returning it by value from a function.
- 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 invoking the copy constructor directly.
   - `MyClass c(otherC); MyClass c1 = c;`
2. When passing an object by value to a **function**
   - `void myFunc(MyClass c) {}`
3. When returning an object by value from a **function**.
   - `MyClass myFunc() { MyClass c; return c }`

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.

RIGHT-LEFT RULE

What's this monstrosity?

```
int a [100];
int (&ref) [100] = a;
```

What's this monstrosity?

```
void swap(int*&p, int *&q)
{
    int * temp = p;
     p = q;
     q = temp;
}
```

Right left rule breakdown

```
int (**var[])();
```

- Start from variable name (var)

“**var** is …”

Right left rule breakdown

```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right parentheses)

"var is an **array**…"

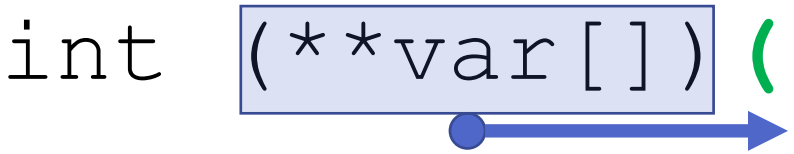
Right left rule breakdown

```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right parentheses)
- Then go left of var. Read what we see until we see left parentheses (

"var is an array of **pointers** to **pointers** …"

Right left rule breakdown



- Start from variable name (var)
- Keep going right. Read what we see until we see right parentheses)
- Then go left of var. Read what we see until we see left parentheses (
- Exit parentheses and go right. Read what we see until right parentheses or semicolon

"var is an array of pointers to pointers to a **function** …"

Right left rule breakdown

- Start from variable name (var)
- Keep going right. Read what we see until we see right parentheses)
- Then go left of var. Read what we see until we see left parentheses (
- Exit bracket and go right. Read what we see until we see right parentheses or semicolon
- Go left of where we exited bracket. Read what we see until right parentheses or finish
- Repeat steps as needed

"var is an array of pointers to pointers to a function that **returns int**"

Right left rule breakdown

```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right parentheses)
- Then go left of ppf. Read what we see until we see left parentheses (
- Exit bracket and go right. Read what we see until we see right parentheses or semicolon
- Go left of where we exited bracket. Read what we see until right parentheses or finish
- Repeat steps as needed

"var is an array of pointers to pointers to a function that returns int"

More RL rule (step by step)

```
int * (* (*fp1) (int) ) [10];
```

Start from the variable name (fp1)

Nothing to right but) so go left to find * (is a pointer)

Jump out of parentheses and encounter (int) (to a function that takes an int parameter)

Go left, find *(and returns a pointer)

Jump put of parentheses, go right and hit [10] (to an array of 10)

Go left find * (pointers to)

Go left again, find int (ints)

More RL rule (step by step)

```
int *( *( *arr[5])())();
```

Start from the variable name (arr)
Go right, find array subscript (is an array of 5)
Go left, find * (pointers)
Jump out of parentheses, go right to find () (to functions)
Go left, encounter * (that return pointers)
Jump out, go right, find () (to functions)
Go left, find * (that return pointers)
Continue left, find int (to ints).

What's happening here?

```
int ***ppp;
int (**ppa)[];
int (**ppf)();
int *(*pap)[];
int (*paa)[][];
int (*paf)[]();
```

Let's examine the RL rule

```
int ***ppp;
```

1. Ppp is an int pointer to a pointer to a pointer
2. Ppp is a pointer to a pointer to an int pointer
3. Ppp is a pointer an int pointer to a pointer
4. Ppp is a pointer to a pointer to a pointer returning an int

More RL rule

```
int (**ppa)[];
```

1. Ppa is an array of pointer to a pointer returning ints
2. Ppa is an int pointer to a pointer to an array
3. Ppa is an array of pointers to int pointers
4. Ppa is a pointer to a pointer to an array of ints

More RL rule

```
int (**ppf)();
```

1. Ppf is a pointer to a pointer to a function returning an int
2. Ppf is a pointer to a function returning integer pointers
3. Ppf is a function pointing to an integer pointer
4. Ppf is a int pointer to a pointer to a function

More RL rule

```
int *(*pap)[];
```

1. Pap is a pointer to an integer to a pointer to an array
2. Pap is an integer array of pointers to pointers
3. Pap is an array of pointers to pointers to ints
4. Pap is a pointer to an array of int pointers

More RL rule

```
int (*paa)[][];
```

1. Paa is an array of arrays to pointers to ints
2. Paa is a pointer to an array of arrays of ints
3. Paa is an integer pointer to an array of arrays
4. Paa is an array of pointers to an array of ints

More RL rule

```
int (*paf)[]();
```

1. Paf is a function to an array of int pointers
2. Paf is a int pointer to an array of functions
3. Paf is a pointer to an int array of functions
4. Paf is a pointer to an array of functions returning an int

Even more!

```
int *ptr_to_int;
int *func_returning_ptr_to_int();
int (*ptr_to_func_returning_int)();
int (*array_of_ptr_to_func_returning_int[])();
```

CRAZINESS:

```
int (*(*ptr_to_an_array_of_ptr_to_func_returning_int)[])();
```

Final note: if possible, please don't write messy declarations like this

REVIEW

Review Agenda

1. References
2. Overloaded operators
3. Virtual
4. Functions
5. Copy constructor

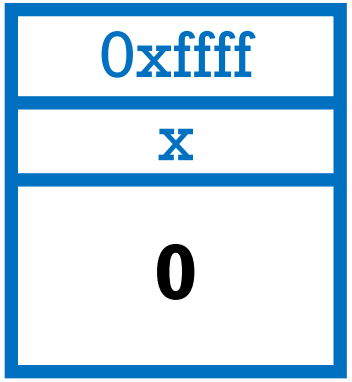
COMP 3522

REFERENCES

When to use References?

- References are an **alias or nickname** for an existing variable
- Can **change the value in variable using the reference**

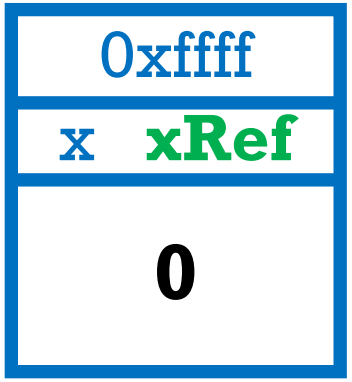
```
int x = 0;
```



When to use References?

- References are an **alias or nickname** for an existing variable
- Can **change the value in variable using the reference**

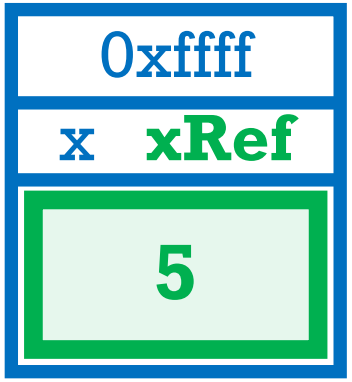
```
int x = 0;
int &xRef = x; //xRef is a nickname for x
```



When to use References?

- References are an **alias or nickname** for an existing variable
- Can **change the value in variable using the reference**

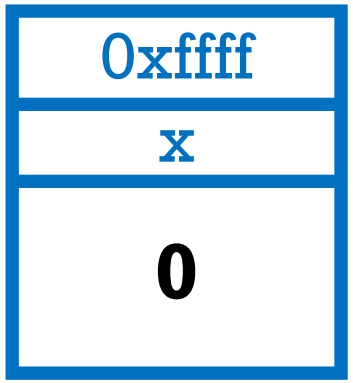
```
int x = 0;
int &xRef = x; //xRef is a nickname for x
xRef += 5; //x changes to 5
cout << x << endl;
```



What is **De**referencing?

- **Dereferencing** is accessing or manipulating the value that a pointer is pointing at. This is a term more related to pointers, than references

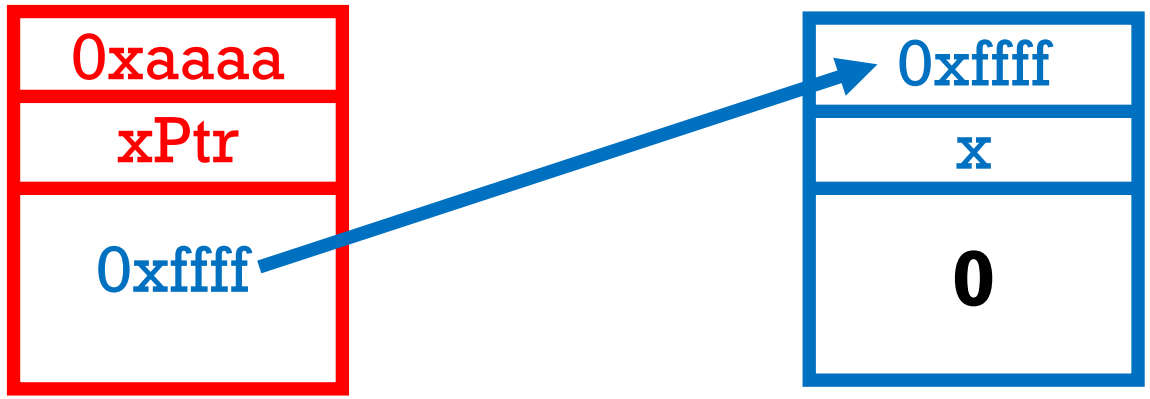
```
int x = 0;
```



What is **De**referencing?

- **Dereferencing** is accessing or manipulating the value that a pointer is pointing at. This is a term more related to pointers, than references

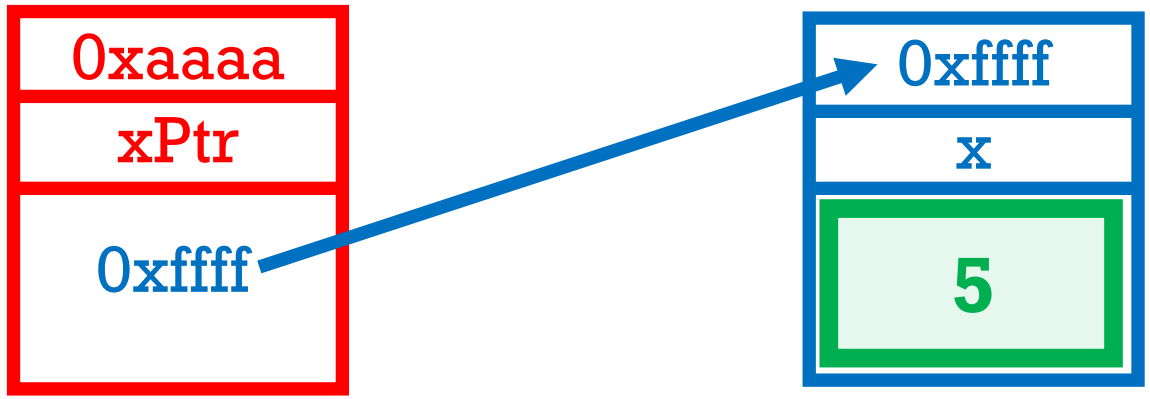
```
int x = 0;
int *xPtr = &x; //xPtr points at x memory address
```



What is **De**referencing?

- **Dereferencing** is accessing or manipulating the value that a pointer is pointing at. This is a term more related to pointers, than references

```
int x = 0;
int *xPtr = &x; //xPtr points at x memory address
*xPtr += 5; //Dereference xPtr to modify x's value
cout << x << endl;
```



When to use References?

- When we need to **change the original value** of a variable when passed into a function

Pass by value

```
void add(int xCopy)
{
    xCopy += 5; //local xCopy changes to 5
}

int main() {
    int x = 0;
    add(x);
    cout << x << endl; //prints 0
    return 0;
}
```

Pass by reference

```
void add(int &x)
{
    x += 5; //original x changes to 5
}

int main() {
    int x = 0;
    add(x);
    cout << x << endl; //prints 5
    return 0;
}
```

When to use References?

- When we don't want to make **unnecessary copies** when passing variable into a function

Don't need to make a copy of mc to print its information, so pass by reference

```
class MyClass {
private:
    int x = 0;

public:
    int getX() const {return x;}
};
```

```
void print(const MyClass& mc)
{
    cout << mc.getX();
}

int main() {
    MyClass mc;
    print(mc);
    return 0;
}
```

When to use References?

- To **allow polymorphic behavior** (also works with pointers)

```
class Animal {
public:
    virtual void speak() = 0;
};

class Dog : public Animal {
public:
    void speak() { cout << "woof" <<
endl;}
};

class Bird : public Animal {
public:
    void speak() { cout << "chirp" <<
endl;}
};
```

```
void animalSpeak(Animal &a)
{
    a.speak(); //calls polymorphic speak
                function
}

int main() {
    Dog d;
    animalSpeak(d); //woof

    Bird b;
    animalSpeak(b); //chirp
    return 0;
}
```

Pointers vs references

Pointers	References
Holds memory addresses `int *numPtr = &num; //numPtr points to memory address of num`	A nickname for an existing variable `int &numRef = num; //numRef and num the same variable`
Can be declared without initialization `int *numPtr; //OK`	Can NOT be declared without initialization outside of class data members. `int &numRef; //ERROR`
Can refer to a nullptr `int *numPtr = nullptr; //OK`	Can NOT refer to NULL or nullptr or nothing `int &numRef = NULL //ERROR`
Access value pointed at by dereferencing `*numPtr = 5;`	Access value referenced directly; `numRef = 5;`

Pointers vs references

Pointers	References

Can move pointer to point to next element
 	
`int arr[3] = {0,1,2};`
`int* numPtr = arr;`
`numPtr++; //moves pointer to point to value 1`
	Can NOT move reference to point to next element
	
	`int arr[3] = {0,1,2};`
	`int (&numRef)[3] = arr;`
	`numRef[0]++; //increments value 0 to 1`

Can point to another memory address after initialization
 	
`int num = 5;`
`int num2 = 10;`
`int* numPtr = &num;`
`numPtr = &num2; //pointer changes to point to num2's address`
	Can NOT refer to a different variable after initialization
	
	`int num = 5;`
	`int num2 = 10;`
	`int& numRef = num;`
	`numRef = num2; //can NOT change numRef to refer to num2. Sets num's value to 10`

OVERLOADED OPERATORS

Overloaded operators

Why use overloaded operators?

They allow us to use shorthand notation to perform operations that are somewhat expected.

Let's say we create a custom MyString class which wraps a regular string. We would like to **print out the string**. Which looks easier to you?

```
MyString ms;
cout << ms.getString(); //cout prints string retrieved
                        from getString()
cout << ms; //cout prints string retrieved from
              overloaded operator<<
```

Overloaded operators

Let's say we create a custom MyString class which wraps a regular string. We would like to **append two strings**. Which looks easier to you?

```
MyString ms;
string s;
ms.append(s); //calls append function to append
              s string to ms object
ms += s; //calls operator+= to append s string
          to ms object
```

Overloaded operators

Assignment 1 could have been written to use no overloaded operators.

However, implementing them allows the code to appear more concise by using the operator shorthand notation

```
//no overloaded operators
Matrix m;
Matrix m2
m.add(2);
m.add(m2);
cout << m.getValues();
```

```
//overloaded operators
Matrix m;
Matrix m2
m += 2;
m += m2;
cout << m;
```

HOW TO IMPLEMENT OVERLOADED OPERATORS

Overloaded operators

Should I write the operator as a member function, or a friendly non-member function?

Generally ask yourself:

"Does the result of this operator modify an operand?"

If **YES**, implement as **member function**

If **NO**, implement as **friendly non-member function**

Overloaded operators

Should I write the operator as a member function, or a friendly non-member function?

Some other factors to consider:

1. Unary or Binary operator?
2. Does the result of the binary operator modify an operand?

Overloaded operators

Unary operator – operator operates on **one operand**

```
++x
x++
--x
x--
!x
...
```

Binary operator – operator operates on **two operands**

```
x += y
x -= y
x + y
x - y
x == y
x != y
...
```

Overloaded operators

Unary operator – operator operates on **one** operand

```
++x
x++
--x
x--
!x
...
```

All of these appear to modify the **x** operand

They should be implemented as **member functions**

Overloaded operators

Binary operator – operator operates on **two** operands

```
x += y
x -= y
x + y
x - y
x == y
x != y
...
```

Not all of these modify the left operand

Only **+=** and **-=** change the value of x in this list

Binary operands that modify the left operand should be implemented as **member functions**

Overloaded operators

Binary operator – operator operates on **two** operands

```
x += y
x -= y
x + y
x - y
x == y
x != y
...
```

Not all of these modify the left operand

+, -, ==, !=

These operands produce a new value and don't modify either operand

Implement as **friendly non-member functions**

operatorsBasic.cpp

operators.cpp

VIRTUAL

Overloaded operators

We've seen the virtual keyword used in several places throughout the term.

1. Virtual functions
2. Virtual destructors
3. Virtual inheritance

The effects of adding virtual differ even though the same keyword is used.

VIRTUAL FUNCTION

Virtual functions

Without virtual functions, classes are not polymorphic. Pointers and references **can NOT** dynamically determine the correct function to call at runtime

`bird.speak();` properly prints out "**Chirp**"

`animalPtr->speak();` incorrectly prints out "**???**"

```
struct Animal {
    void speak(){cout << "???" << endl;}
};

struct Bird : public Animal {
    void speak() { cout << "Chirp" << endl;}
};
```

```
int main() {
    Bird bird;
    Animal *animalPtr = new Bird;
    bird.speak(); //Chirp
    animalPtr->speak(); //???
}
```

Virtual functions

With **virtual** functions, classes become polymorphic. Pointers and references **CAN** dynamically determine the correct function to call at runtime

`bird.speak();` properly prints out "**Chirp**"

`animalPtr->speak();` properly prints out "**Chirp**"

```
struct Animal {
   virtual void speak(){cout << "???" << endl;}
};

struct Bird : public Animal {
   void speak() { cout << "Chirp" << endl;}
};
```

```
int main() {
   Bird bird;
   Animal *animalPtr = new Bird;
   bird.speak(); //Chirp
   animalPtr->speak(); //Chirp
}
```

Virtual functions

Where do I write the virtual keyword?

The first time the function appears in a base class

All future child classes will inherit the virtual function. No need to write virtual keyword again for the same function

```
struct Animal {
    virtual void speak(){cout << "???" << endl;}
};

struct Bird : public Animal {
    void speak() { cout << "Chirp" << endl;}
};
```

```
int main() {
    Bird bird;
    Animal *animalPtr = new Bird;
    bird.speak(); //Chirp
    animalPtr->speak(); //Chirp
}
```

VIRTUAL DESTRUCTOR

Virtual destructor

- When a destructor of a child type is destroyed, it will execute its own destruction code, then call its parent's destructor
- The output shows expected behavior

```
class Animal {
public:
    ~Animal(){ cout << "Animal destroyed" << endl;}
};

class Dog : public Animal {
public:
    ~Dog(){ cout << "Dog destroyed" << endl;}
};
```

```
int main() {
    Dog d;
    return 0;
}
```

Prints:

Dog destroyed
Animal destroyed

Virtual destructor

- Problem happens when we destroy a pointer/reference of a base type, assigned to a child type object
- We're expecting the same output as the previous example, "animal destroyed, dog destroyed"
- But the output is different. BAD

```
class Animal {
public:
    ~Animal(){ cout << "Animal destroyed" << endl;}
};

class Dog : public Animal {
public:
    ~Dog(){ cout << "Dog destroyed" << endl;}
};
```

```
int main() {
    Animal *a = new Dog;
    delete a;
    return 0;
}
```

Prints:

Animal destroyed

Virtual destructor

- When "delete a;" is executed, the pointer is of Animal type, so the Animal class' destructor is called
- Pointer 'a' is pointing to a Dog object, so we actually wanted to call the dog's destructor first
- The reason dog's destructor is not called first is Animal's destructor is not polymorphic

```
class Animal {
public:
    ~Animal(){ cout << "Animal destroyed" << endl;}
};

class Dog : public Animal {
public:
    ~Dog(){ cout << "Dog destroyed" << endl;}
};
```

```
int main() {
    Animal *a = new Dog;
    delete a;
    return 0;
}
```

Prints:
Animal destroyed

Inheritance.cpp
Inheritance2.cpp

Virtual destructor

- Therefore we need to add **virtual** in front of the base class' destructor to polymorphically call the proper child type's destructor first
- **Base classes** should always have a **virtual** destructor
- This yields the appropriate output and behavior

```
class Animal {
public:
    virtual ~Animal(){ cout << "Animal destroyed"
<< endl;}
};

class Dog : public Animal {
public:
    ~Dog(){ cout << "Dog destroyed" << endl;}
};
```

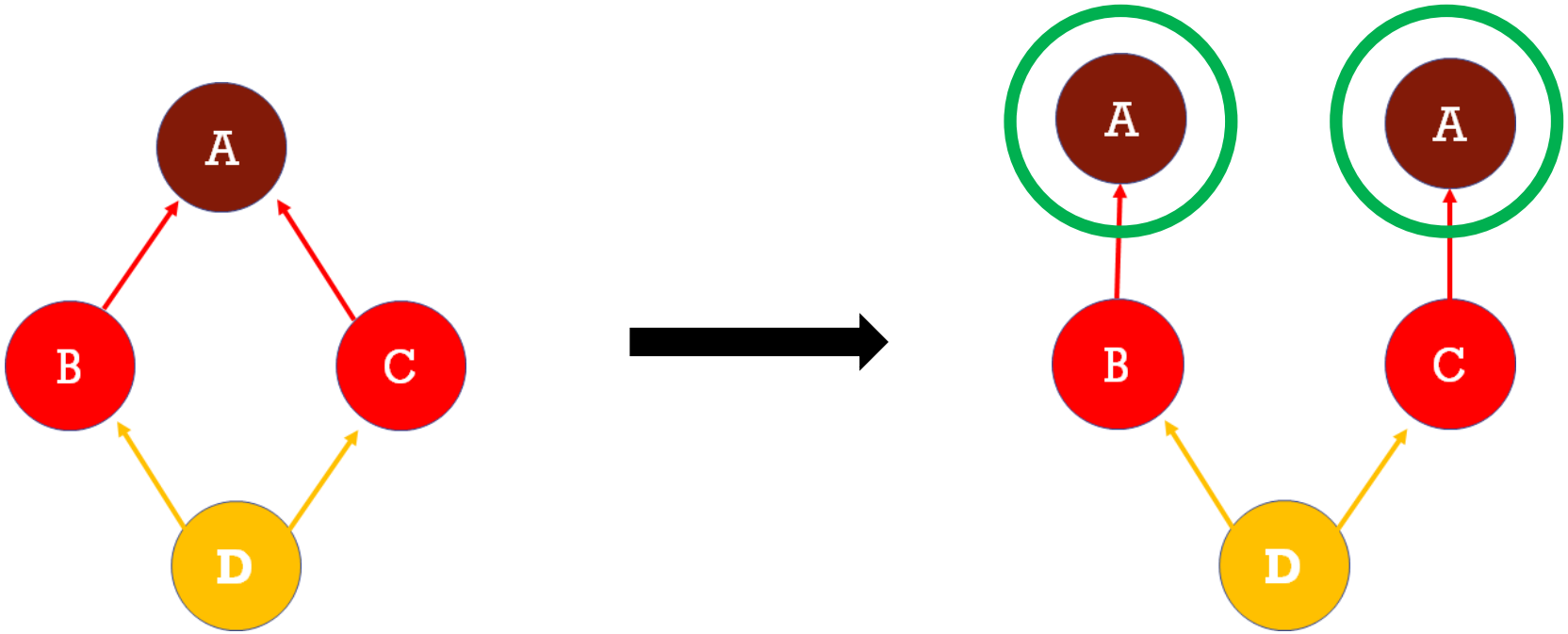
```
int main() {
    Animal *a = new Dog;
    delete a;
    return 0;
}
```

Prints:
Dog destroyed
Animal destroyed

VIRTUAL INHERITANCE

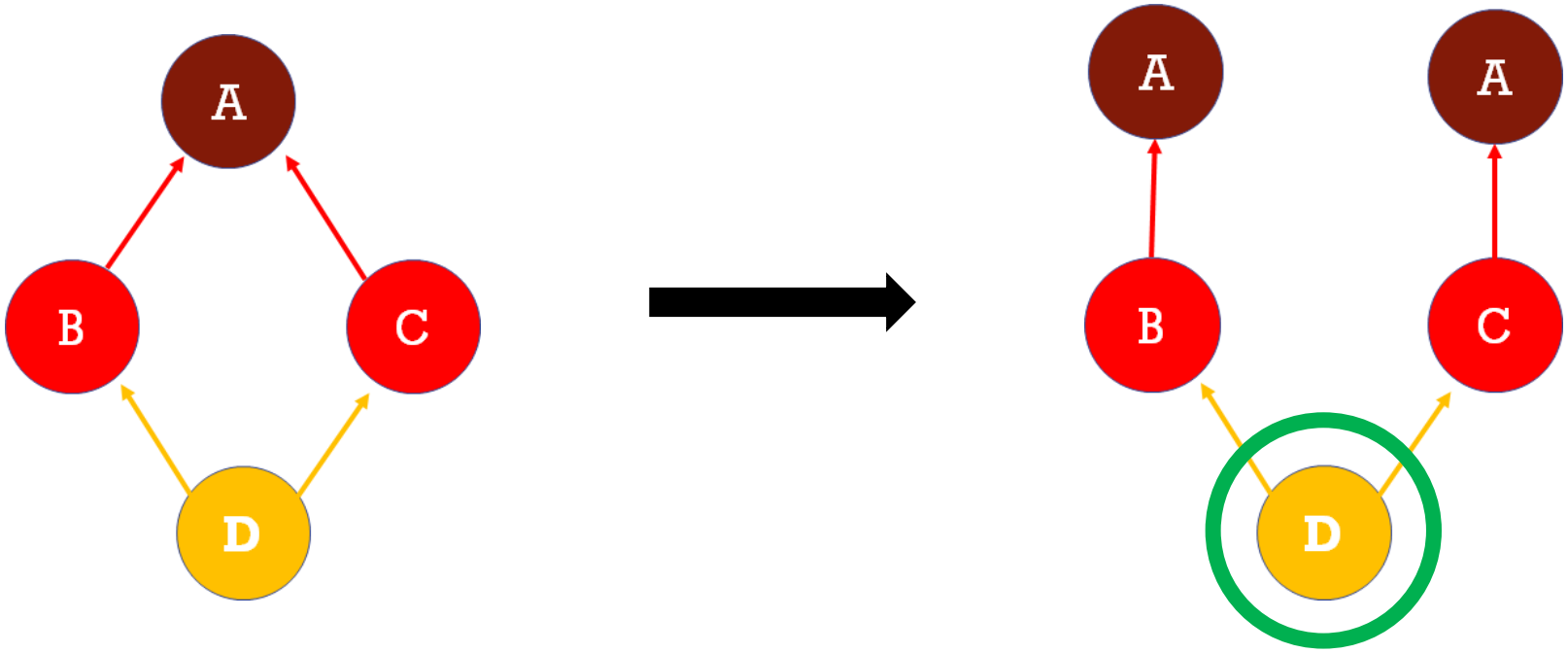
Virtual destructor

- We saw a specific issue that can happen when multiple inheritance leads to a diamond inheritance structure
- **Multiple instances** of the base class are created even though the inheritance structure looks like a diamond



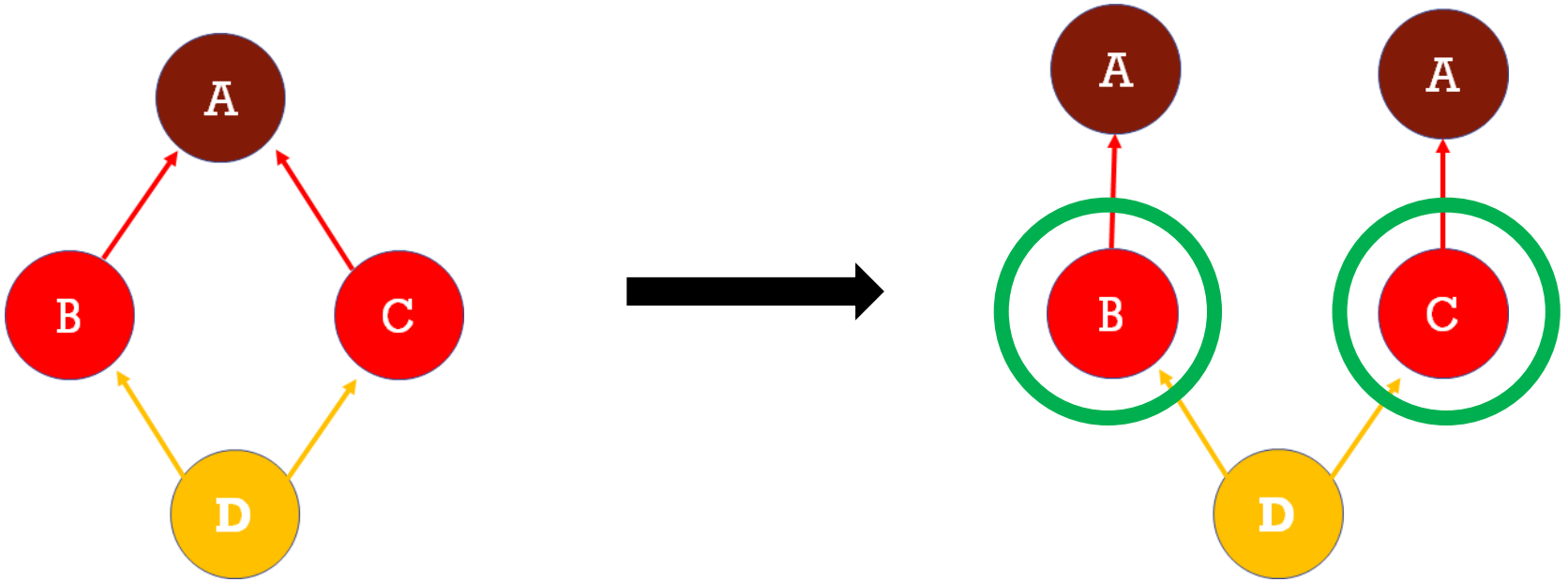
Virtual destructor

- This occurred because **instantiating class D**, calls the constructors of B and C
- Therefore both B and C, call the constructors of each of their A parents



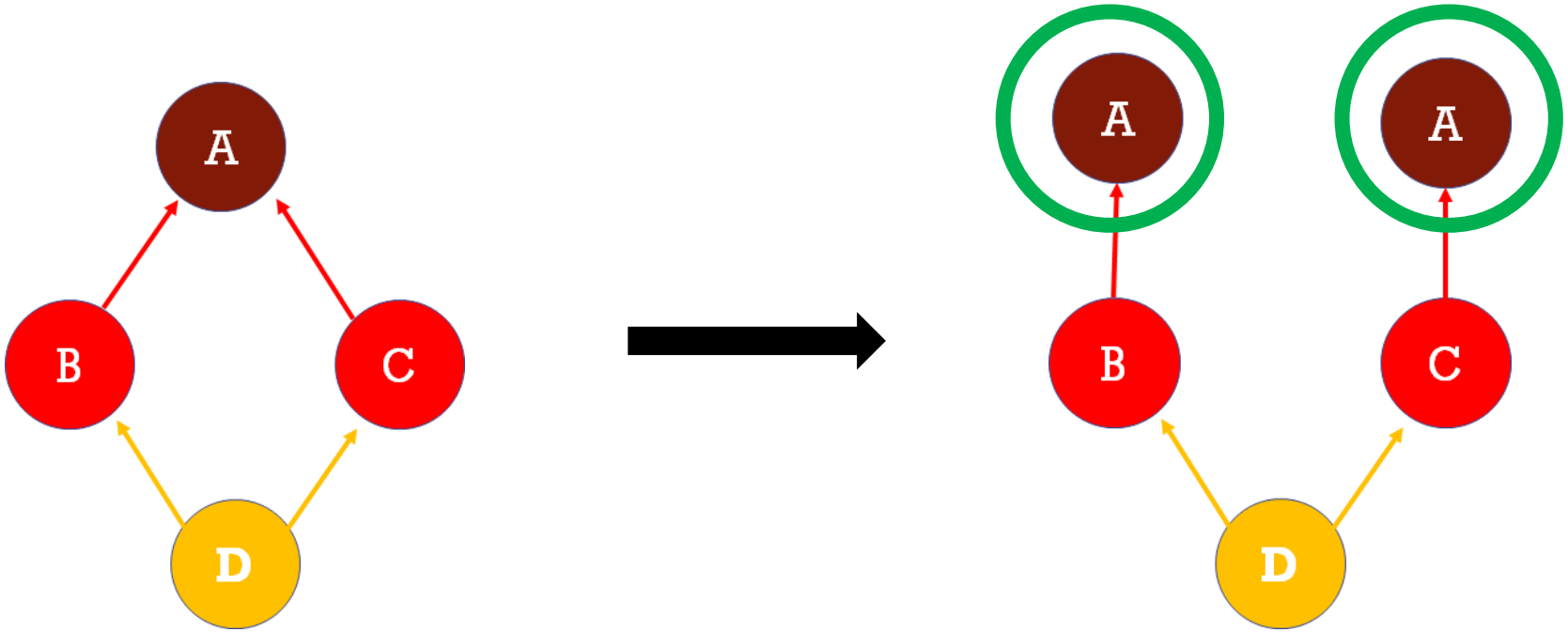
Virtual destructor

- This occurred because instantiating class D, **calls the constructors of B and C**
- Therefore both B and C, call the constructors of each of their A parents



Virtual destructor

- This occurred because instantiating class D, calls the constructors of B and C
- Therefore both B and C, call the **constructors of each of their A parents**



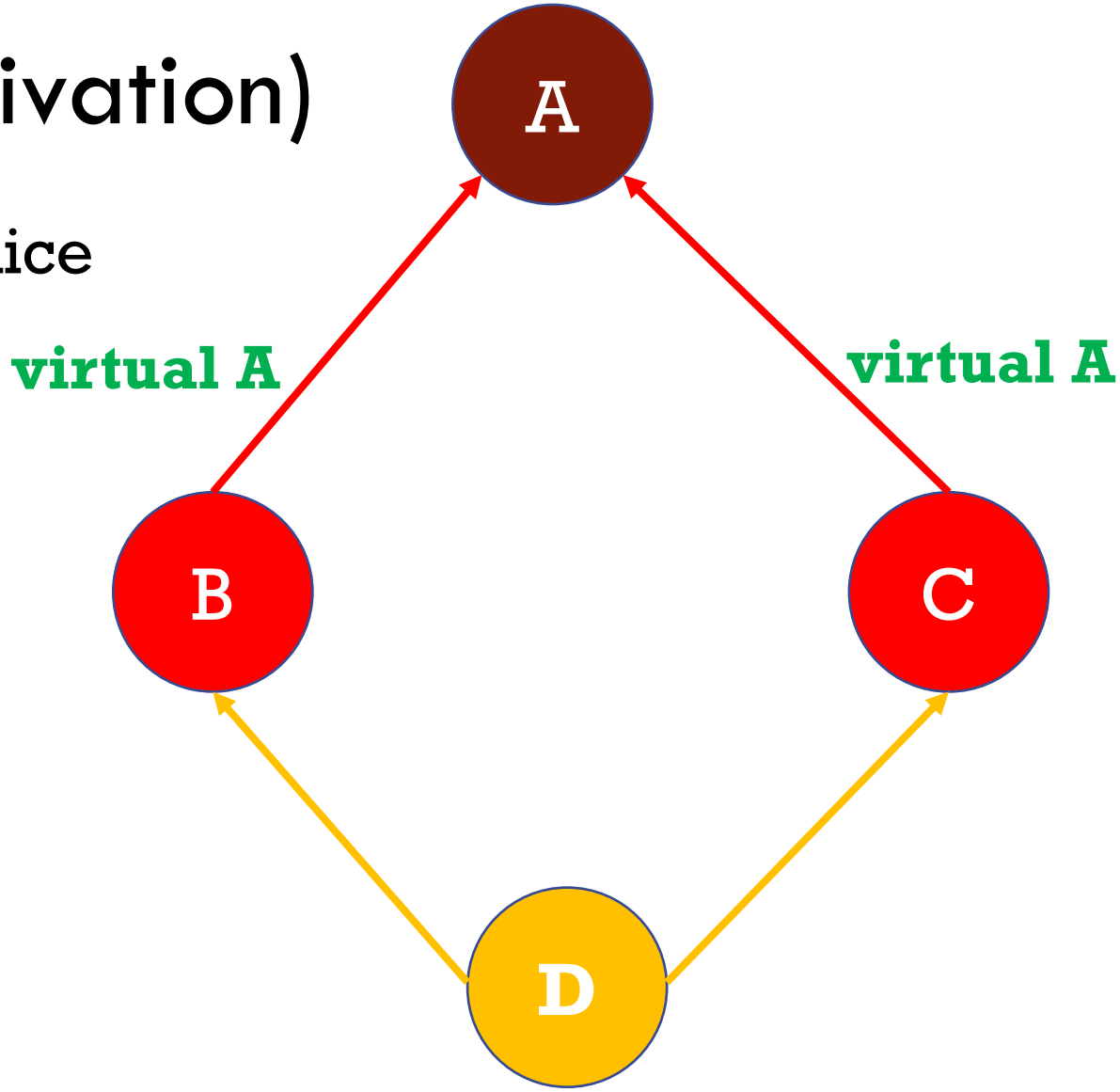
Virtual base classes (motivation)

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

- In code it looks like

```
class B : virtual public A {
… //class code
}

class C : virtual public A {
… //class code
}
```



Virtual base classes

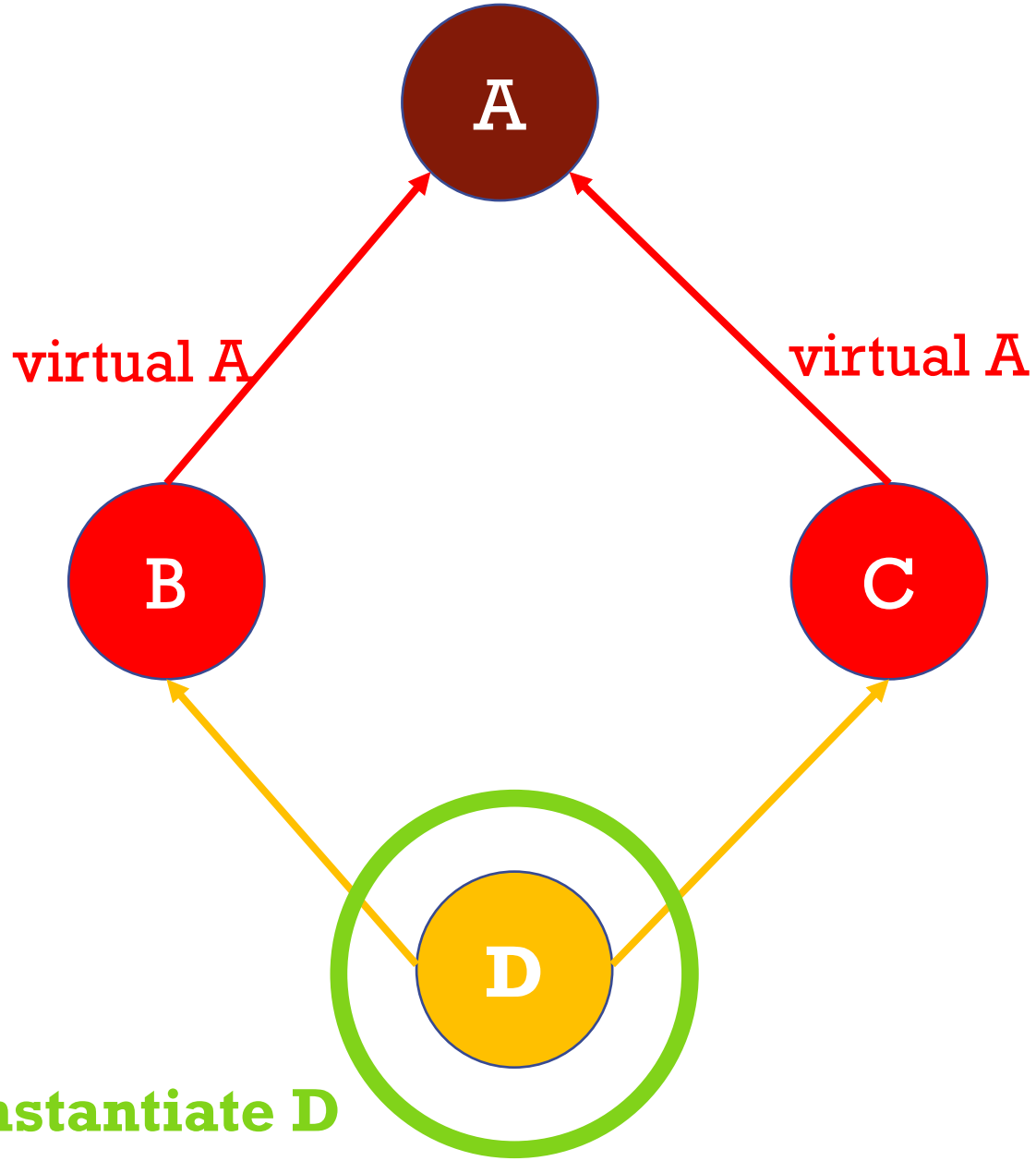
- It is a **derived class's responsibility to call the base class constructor** (or the compiler will insert a call to the default constructor)
- We only have 1 version of the person base class because both `B` and `C` denote `A` as a virtual base class
- `B` and `C` no longer contain the base class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

Multiple Inheritance

- Problem
  - I want to instantiate D but B & C can not call A's constructor. A is now virtually inherited by B & C

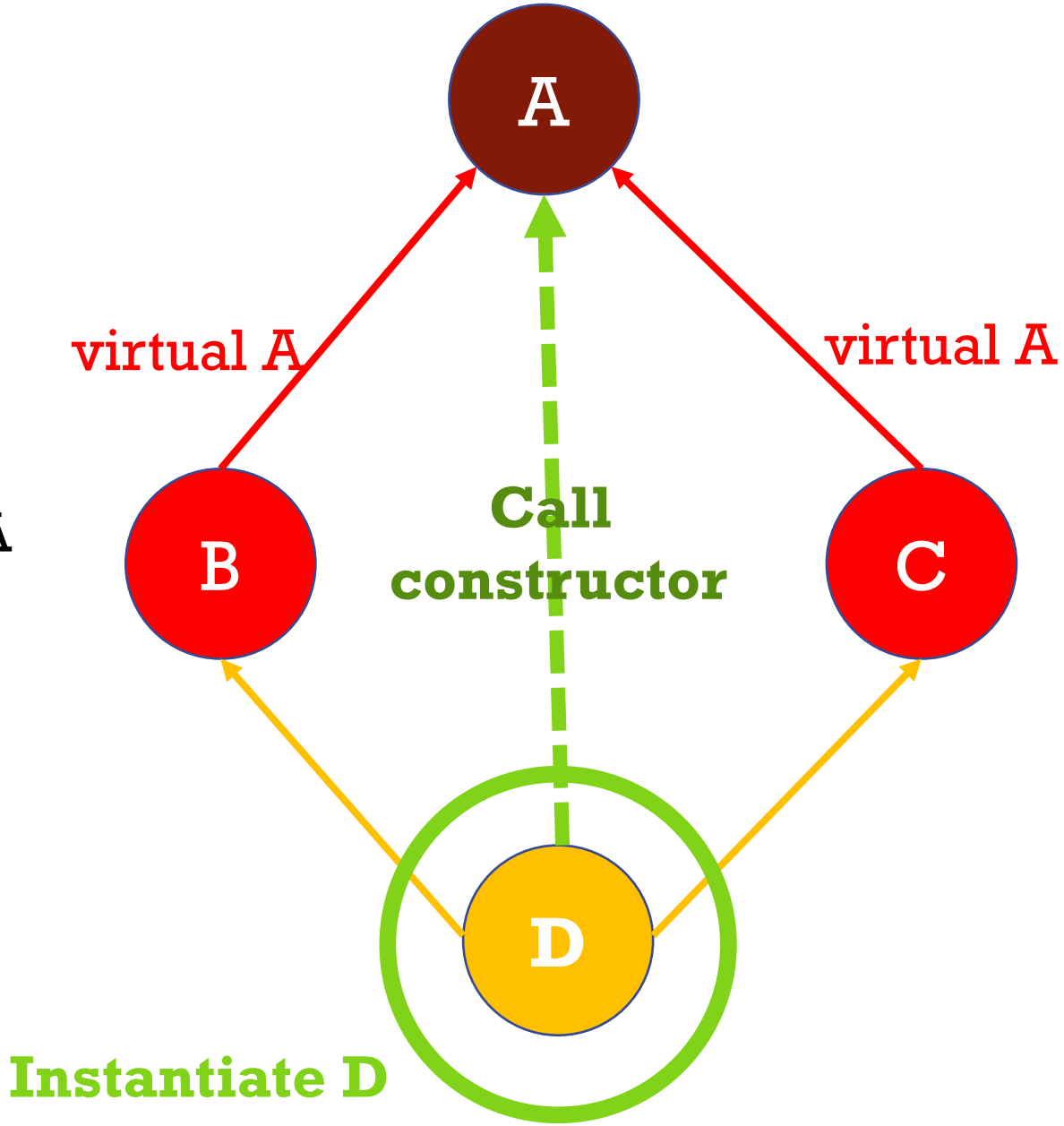
D d; //code to instantiate D

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



Multiple Inheritance

- Solution
  - D will implicitly call A's default constructor
  - But you can write code in D's constructor to call any constructor in A



oop_multi3.cpp

FUNCTIONS

“Regular” Functions

- Purpose: groups code to perform a specific task
- Difference: Functions that don’t belong to a class
- Think of a function in main.cpp

```
void myFunction() {}
```

Member functions

- Purpose: groups code to perform a specific task related to the class it belongs to
  - Get/set data of class
  - Perform operation on class' data members to produce a result
- Difference: Functions that belong to a class
- Example: declare and define function at same time

```
class MyClass {
   void myFunction() { //code; } //declare and define in
                                 class
}
```

Member functions

- Example: Declare function in class, define outside class

```
class MyClass {
public:
   void myFunction(); //declare function prototype
}

void MyClass::myFunction() {} //define function outside of
                              class.
```

Note: must scope the function with “**MyClass::**” if defining outside the class

Friend functions

- Purpose: Same as "regular" functions
- Difference: A **non member function** that has **access to private/protected data members** of the class that declared it
- Friend functions are NOT member functions of a class even though they're declared in a class
  - Visibility specifiers (private/protected/public) don't affect friend functions
- Friend functions can be implemented in the class or outside the class

Friends in C++

```
//MyClass.hpp
class MyClass {
private:
    int x = 0;
    friend void friendFunc(MyClass mc);
};


//main.cpp
void friendFunc(MyClass mc) {
    mc.x++;
    std::cout << mc.x;
}

int main() {
    MyClass mc;
    friendFunc(mc);

    return 0;
}
```

mc.x is **private**, which is normally not accessible

But because friendFunc is a friend function declared within **MyClass**, it has access to all data members of **MyClass**

Friends in C++

- When do we use friend functions?
  - **Overloading operators**
  - Get access to private data of a class
- Shouldn't abuse ability to access private data by making all classes friends
  - Create accessors (getters) and mutators (setters) instead
- Don't scope friend function definitions because friend functions don't belong to a class,
  - ```
    void MyClass::friendFunc(MyClass mc) {
        mc.x++;
        std::cout << mc.x;
    }
    ```
    ```

COPY CONSTRUCTOR

Copy constructor

- When do we use it?
  - When we want to copy an existing object into a new object

```
MyClass mc;
MyClass mc2(mc);
```

  - Implicitly calling copy constructor whenever we pass by value, return by value

```
//main function
MyClass mc;
MyClass mc2 = myFunction(mc);
```

```
MyClass myFunction(MyClass mcCopy)
{
  //local mcCopy is a copy of the mc that was passed
in. This copy is created using your copy constructor
   return mcCopy; //return value is also copied
}
```

Copy constructor

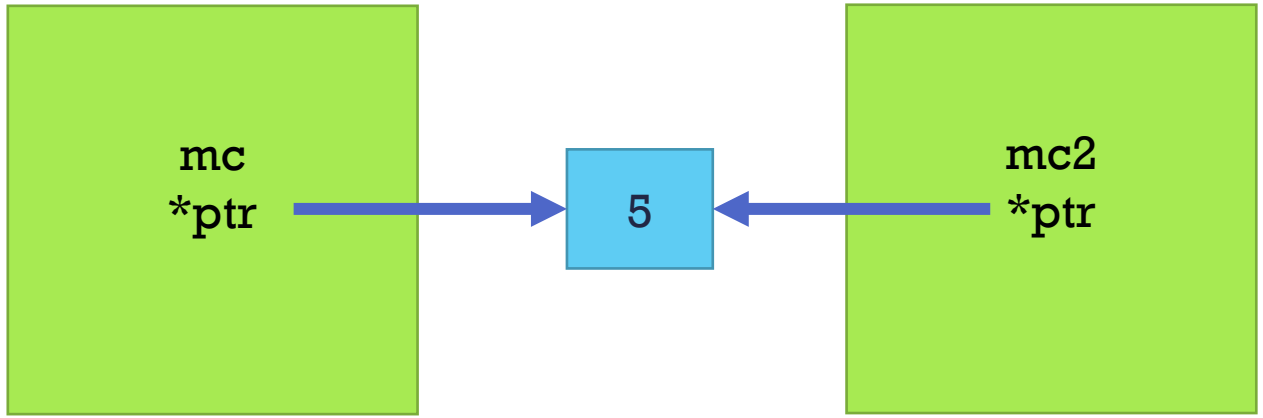
- Why use it?
  - Because If we don't write a working copy constructor, C++ uses a built-in copy constructor that performs shallow copies
  - Shallow copies do NOT properly create new copies of dynamic memory
- When NOT to use it?
  - If your class does NOT have **data members** created using **dynamic memory**
  - If your class does NOT have **data member** pointers to memory that needs to be **deep copied**

Copy constructor

- Did **NOT** write a copy constructor, so default copy constructor copied the address mc is pointing to, to mc2
- This means both mc and mc2 are pointing to the same object

```
struct MyClass {
    int *ptr; //pointer to dynamic memory int
public:
    MyClass() { ptr = new int(5);}
};

int main() {
    MyClass mc;
    MyClass mc2(mc);
    return 0;
}
```



Copy constructor

- Wrote our own copy constructor
- Allocate new dynamic memory for int, copy value from the other ptr, make ptr point to it
- mc & mc2 now point to their own copies of int

```
struct MyClass {
    int *ptr; //pointer to dynamic memory int
public:
    MyClass() { ptr = new int(5);}
    MyClass(const MyClass& other) : ptr(new int(*other.ptr)) {} //deep copy of other int
};


int main() {
    MyClass mc;
    MyClass mc2(mc);
    return 0;
}
```