COMP 3522

Object Oriented Programming in C++

Week 5

Agenda

1. Polymorphism pt.2
2. Abstract classes & interfaces
3. Sizeof and typeid
4. Multiple Inheritence
5. Exceptions

COMP 3522

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, vehicles.cpp

Tricky bug

- Be careful when working with inheritance and private variables

- DerivedX.cpp

ABSTRACT CLASSES

Java Abstract classes

```
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& functionA(A& aParam); // A O K
A* pa; // A O K
```

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
- No implementation
- 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

Interfaces

```
class Animal
{
public:
    virtual ~Animal( ) {}
    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 and regular functions
- An **interface** is a "pure abstract class" in C++:
  - Purely virtual functions
  - No data

Not implemented ⟷ Fully implemented

- **Interface**
- All pure virtual functions
- Virtual destructor
- 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

Inheriting constructors

- C++11 allows us to inherit all constructors from a base class with a `using declaration`
- When constructors with the same signature exist in both classes, the version from the derived class is used

See: inherit_constructor.cpp

oop_constructors.cpp

sizeof AND typeid

Interesting aside: sizeof operator

Returns a size_t representing the number of bytes of the object representation of the type

- **Works for an array** because an array size is known at compilation
- **Does NOT work with pointers** – it gives us the byte size of the pointer itself

See: sizeof.cpp

Another aside: the typeid operator

- We can use the typeid operator for:
    1. Run-time type identification
    2. Identification of a type.
- #include <typeinfo>
- I think of this as C++'s instanceof

See: typeid.cpp Clion and Visual Studio

One final interesting aside: arguments

The order of evaluation of arguments is not defined in C++

```
int a = 1, b = 2, c;
c = f(++a) + g(++a) + b;
```

Whether f(++a) or g(++a) is evaluated first depends on the compiler implementation.

<u>Don't do this.</u>

See: EvaluationOrder.cpp Clion and Visual Studio

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?

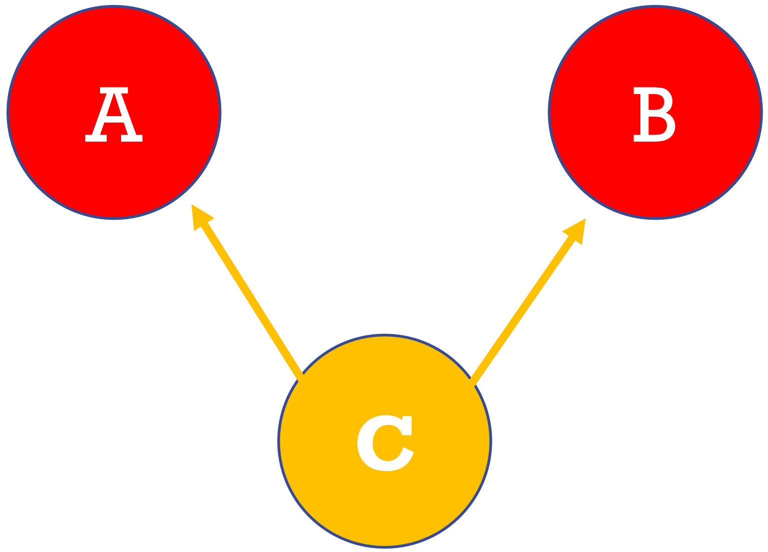
What is the difference between static and dynamic allocation? Provide code examples.

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

MULTIPLE INHERITANCE

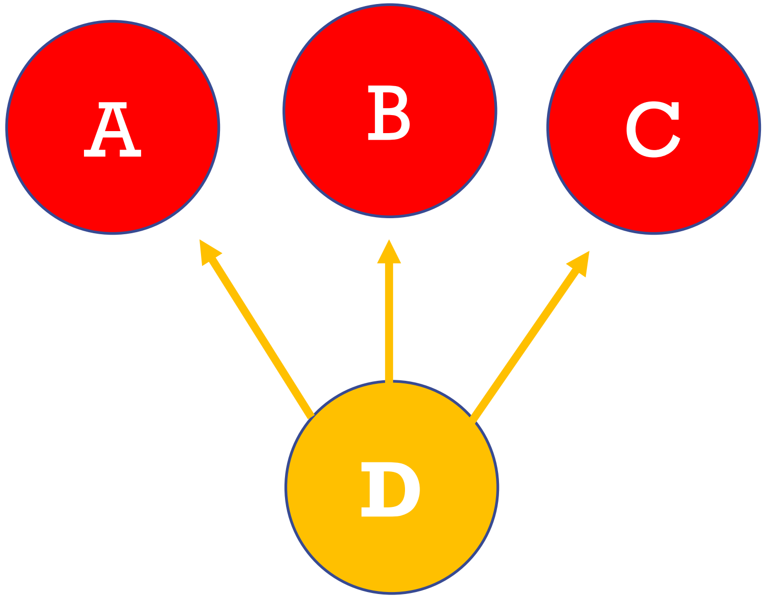
Multiple Inheritance

- 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



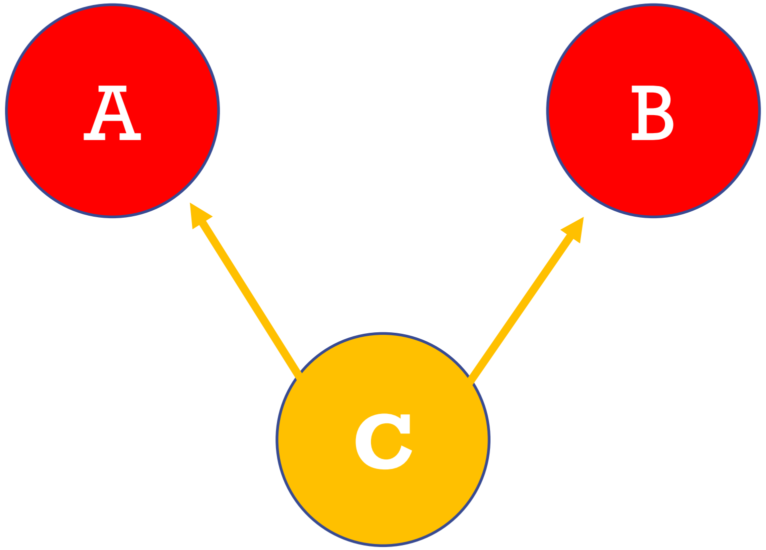
Multiple Inheritance

- 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

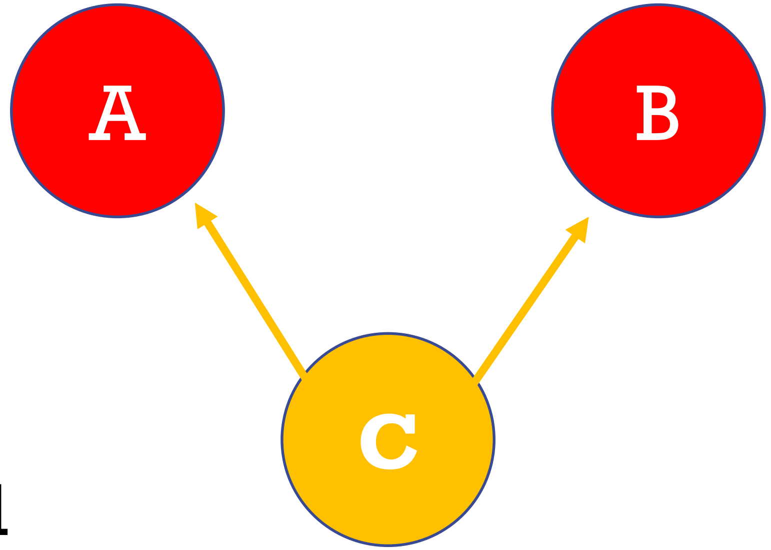


- A has int x, void function1()
- B has int y, void function2()

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

- In this case, things are ok, data members and functions all have different names in class C

Multiple Inheritance (Ambiguity)

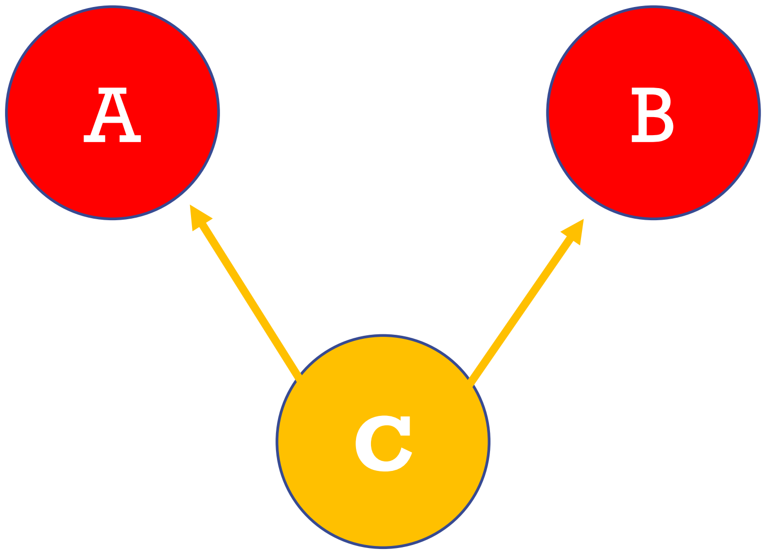


- A has int x, void function1()
- B has int x, void **function1()**

- 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::function();
```

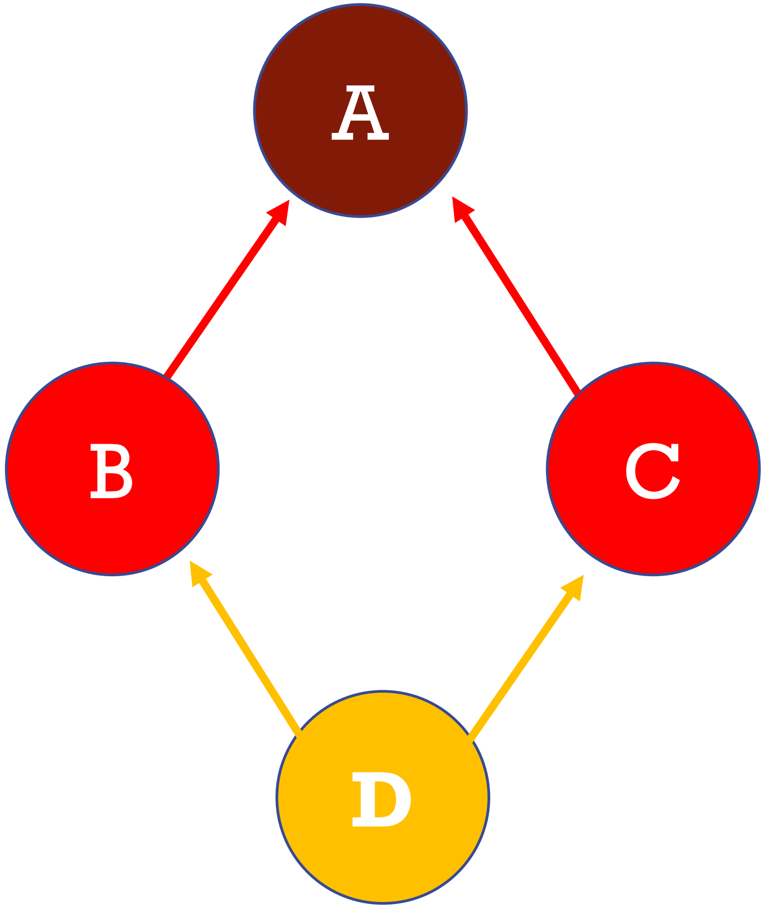
Multiple inheritance

- 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

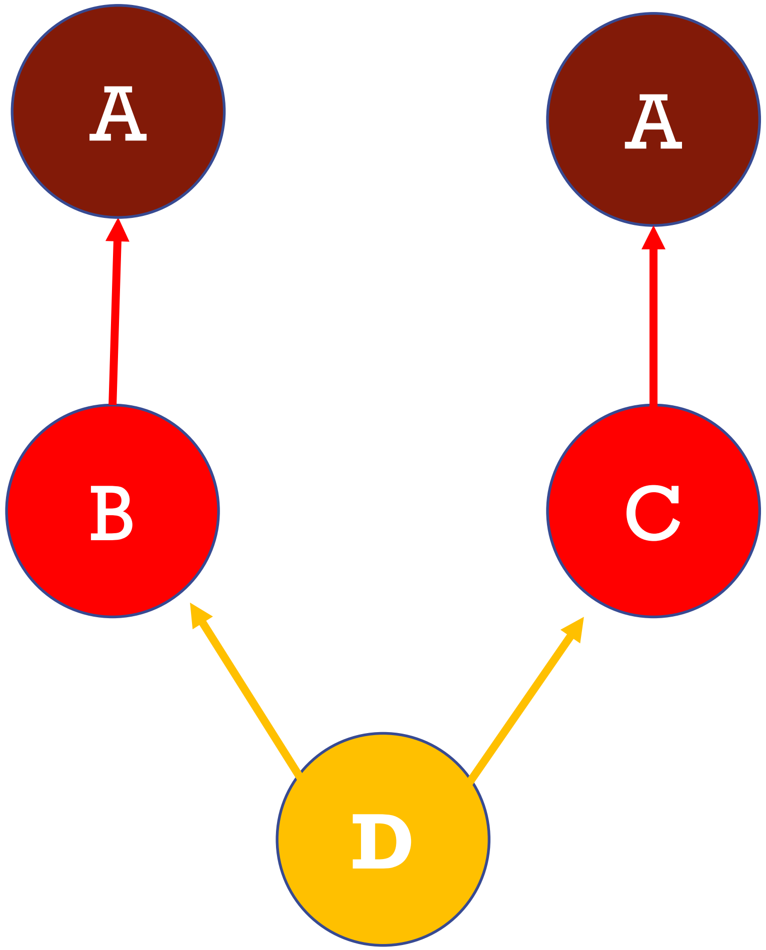
- Two classes may be derived from the same base class
- These two derived classes may be the base class for another derived class
- They are common grandparents
- This creates a classic **diamond shape** inheritance configuration
- But how many grandparents are created?

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 (A)



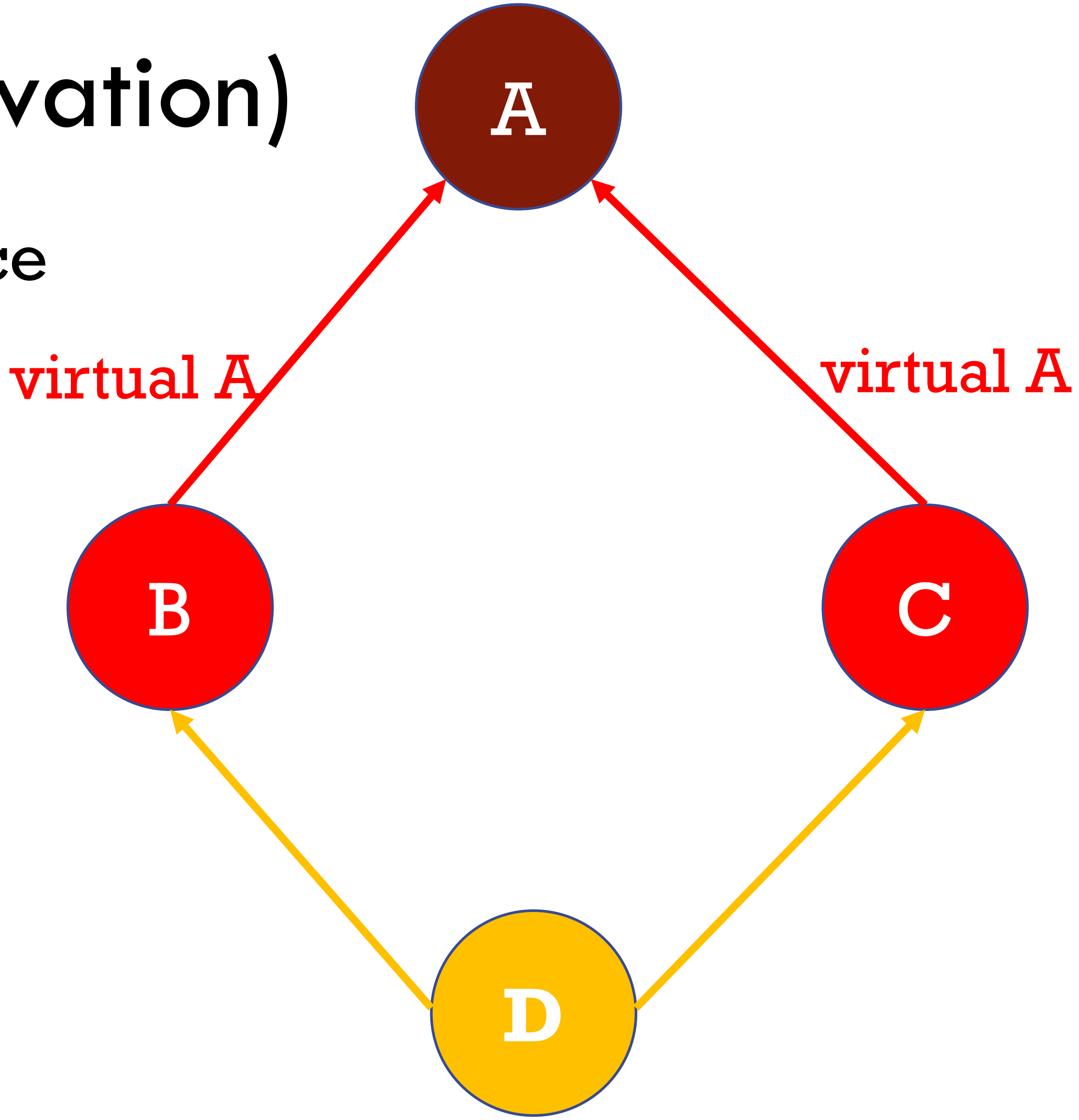
Virtual base classes (motivation)

- 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

- 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'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 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

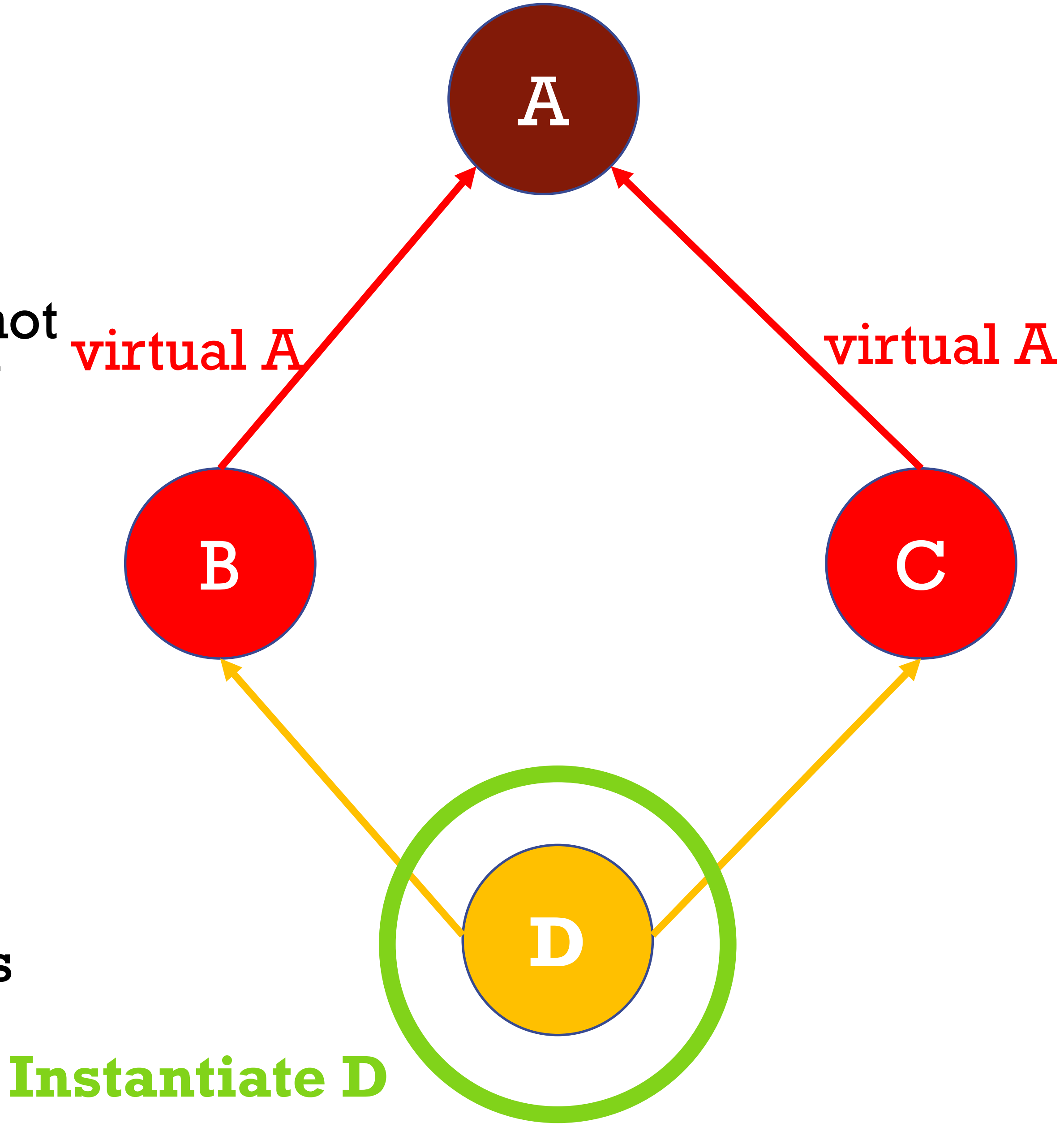
- In the case of virtual base classes, it is the responsibility of the most derived class (math_student) to call the shared base-class constructor (person)
- The person constructor calls in mathematician and student are disabled when they are indirectly called from a derived class

Multiple Inheritance

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

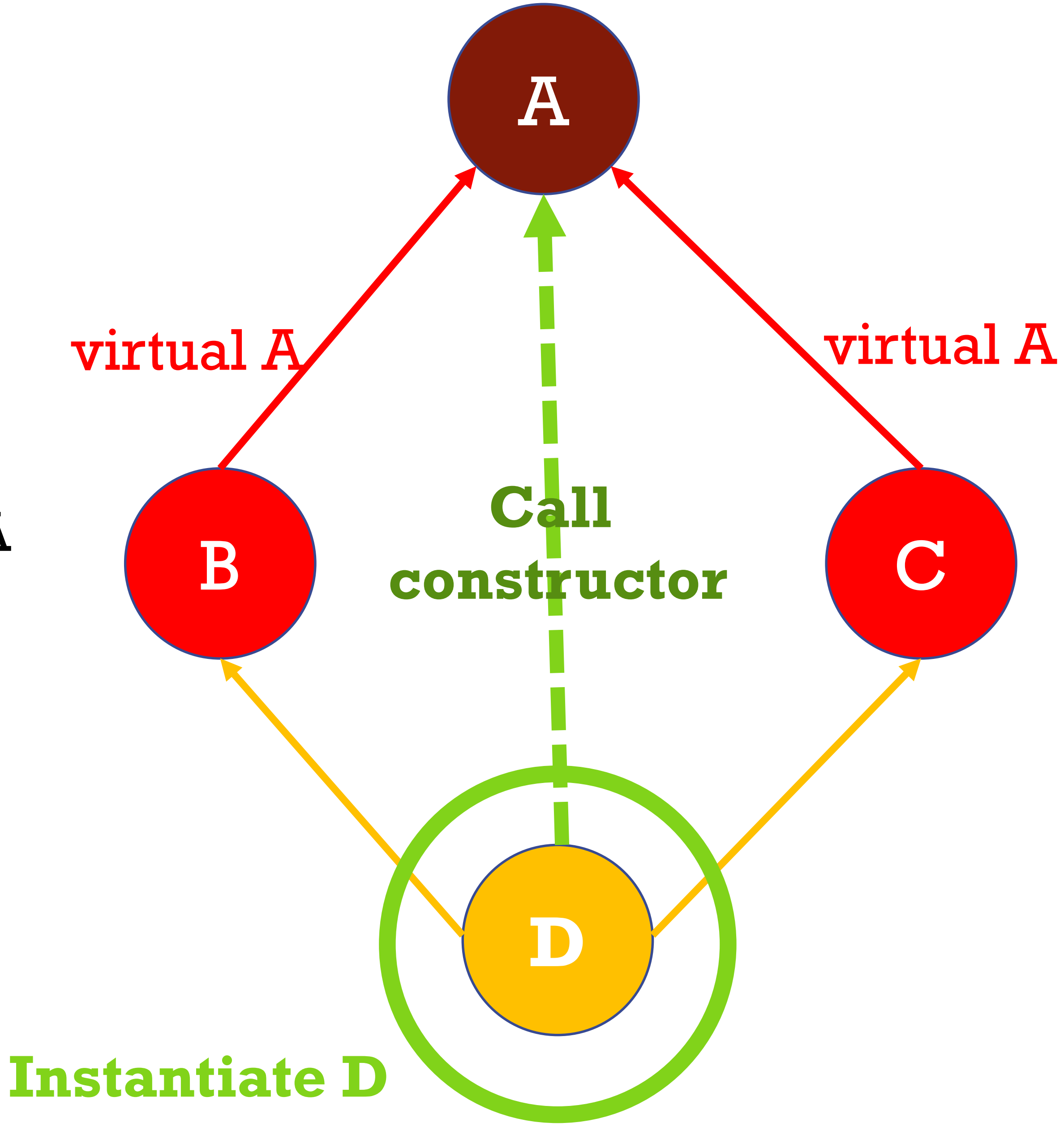
D d; //code to instantiate D

- 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

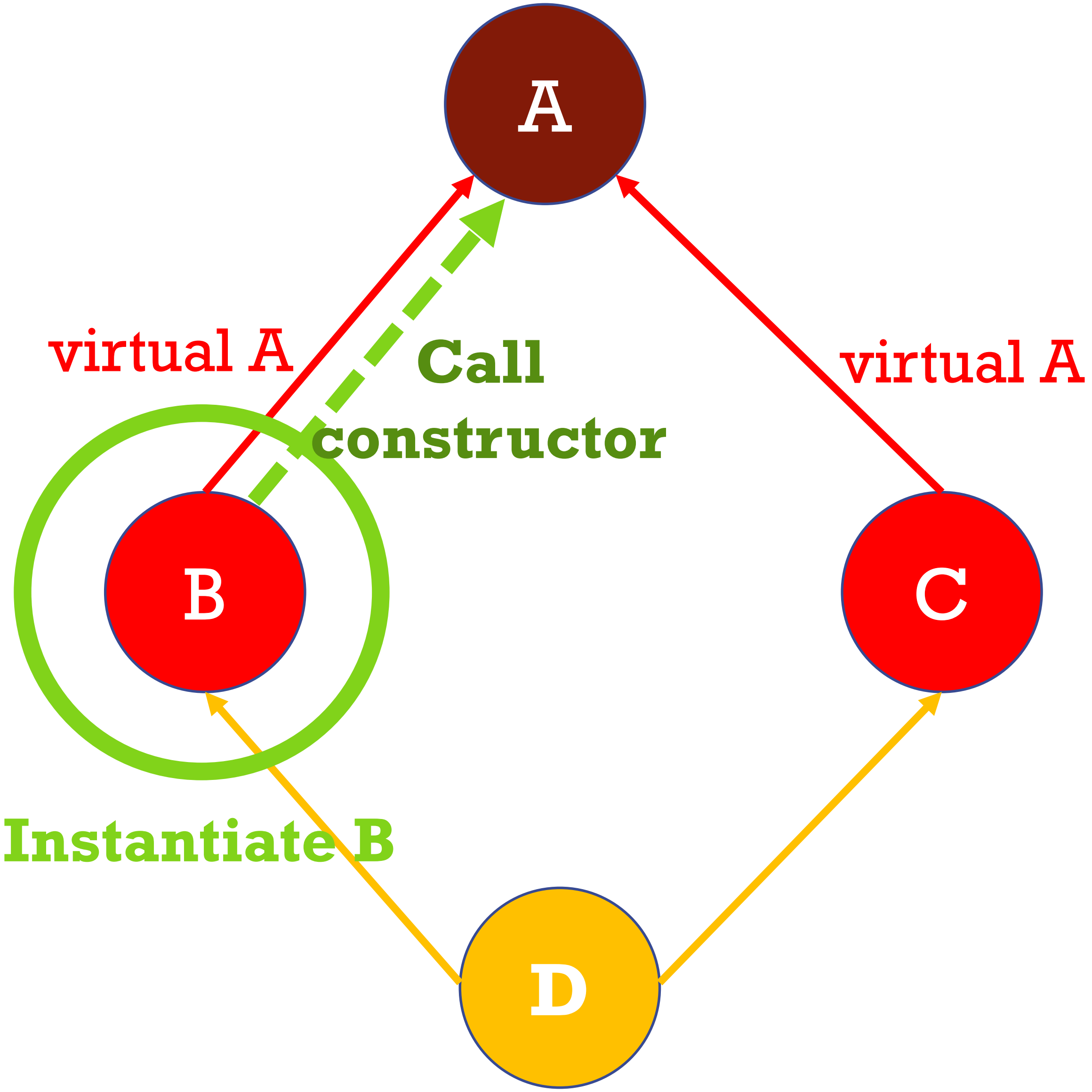


Multiple Inheritance

- How about instantiating B? Can it call A's constructor?

B b; //code to instantiate b

- YES!
- B can call A's constructor when it's directly instantiated
- B can not call A's constructor only if it's being called through D



oop_multi3.cpp, diamond.cpp, diamondmethods.cpp

EXCEPTIONS

Typical Error Situations

- Implementing a class or method incorrectly
- Failing to meet the specification
- Making an inappropriate object request
  - invalid index
- Generating an inconsistent or inappropriate object state
  - arising through class extension

Not ALWAYS Programmer Error*

- Errors often arise from the environment:
  - Incorrect URL entered
  - Network interruption
- File processing is particular error-prone:
  - Missing files
  - Lack of appropriate permissions

* ^^but ^^it ^^usually ^^is

Typical Java exceptions we've seen

Exception	Purpose
NullPointerException	When an application attempts to use an object reference that is set to null
ArrayIndexOutOfBoundsException	Indicate that an index is out of range
ClassCastException	Indicate that the code has attempted to cast an object to a subclass of which it is not an instance
ConcurrentModificationException	Indicate concurrent modification of an object when such modification is not permissible

Dealing with unexpected behaviour in C++

- Two principle approaches:

1. **Assertions** are for detecting programming errors
2. **Exceptions** for situations that prevent proper continuation of the program (errors that cannot be handled locally)

Assertions

- The macro assert from header <cassert> is inherited from C
- Evaluates an expression, immediately terminates the program if false
- Easy to turn off by defining NDEBUG before including <cassert>

```
#define NDEBUG // Turns off assertions
#include <cassert>
```

Assertion example

```
#include <cassert>
// Compute square root of non-negative number
double square_root(double x)
{
  check_somehow(x >= 0);
  … // Perform our calculation
  assert(result >= 0.0); // Should be positive
  return result;
}
```

C error codes

In C, programmers used to return error codes (like main still does)

```
int read_matrix_file(cont char* fname)
{
    fstream f(fname);
    if (!f.is_open()) { return 1; }
    …
    return 0;
}
```

Problem # 1: we can <u>ignore</u> the error code

C error codes

More problems:

1. We can’t return our computational results
2. We have to return a success/error code
3. We are forced to pass references as arguments
4. This can prevent us from building expressions with the results

Enter the exception

```
int read_matrix_file(cont char* fname)
{
    fstream f(fname);
    if (!f.is_open()) { throw “Can’t open file”; }
    …
    return 0;
}
```

C++ exceptions

- C++ lets us throw anything as an exception:

  1. Strings
  2. Numbers
  3. User types
  4. Exceptions from the standard library.

- **It is best, however, to define exception types or use exceptions from the standard library.**

Refined exception example

```
struct cannot_open_file { … };

int read_matrix_file(cont char* fname)
{
    fstream f(fname);
    if (!f.is_open()) {throw cannot_open_file }
    …
    return 0;
}
```

Reacting to an exception

- We must catch exceptions (just like Java)
- We use a try-catch block:

```
try
{
    …
}
catch (e1_type1& e1)
{
    //handle the exception
}
```

Some guidelines

1. Catch exceptions by **reference**
   - Captures exceptions that are derived from the reference type
2. When an exception is thrown, the **first catch-block** with a matching type is executed

```
try
{
        …
}
catch (e1_type1& e1) { //handle the exception }
catch (e1_type2& e2) { //handle the exception }
```

Some guidelines

3. Further catch-blocks of the same type or sub-types are ignored

```
try
{
      …
}
catch (e1_type1& e1) {} //catches e1_type1
catch (e1_type1& e2) {} //IGNORED - same type as previous
```

Some guidelines

3. Further catch-blocks of the same type or sub-types are ignored

```
try
{
      …
}
catch (sub_type& e1) {} //order matters. Derived type first
catch (parent_type& e2) {}
```

Some guidelines

4. A catch-block with an **ellipsis**, i.e., three dots, catches all exceptions
   - Obviously this should be the last one

```
try
{
      …
}
catch (e1_type1& e1) {}
catch (e1_type2& e2) {}
catch (…) { // This catches EVERYTHING }
```

Some guidelines

5. If nothing else, consider capturing the exception, providing an informative error message, and terminating the program:

```
try {
    int result = read_matrix_file(“No file”);
} catch (cannot_open_file& e) {
    cerr << “FILE NOT FOUND.  TERMINATING…\n”;
    exit(EXIT_FAILURE); // <cstdlib>
}
```

Some guidelines

6. Alternatively, we can continue after the error message or after implementing some sort of rescue, by **rethrowing** the exception

```
try {
    int result = read_matrix_file(“No file”);
} catch (cannot_open_file& e) {
    cerr << “FILE NOT FOUND.\n”;
    …
    throw; // Rethrows cannot_open_file exception
}
```

Noexcept qualification for functions

- C++03 allowed us to specify which types of exceptions can be thrown from a function (like Java)
- Was very quickly deprecated (don't do this)

- So what should we do?
- **C++11 added a new qualification for specifying that no exceptions must be thrown out of a function**

```
double square_root(double x) noexcept { … }
```

Noexcept qualification for functions

- Benefits:
  1. Calling code never needs to check for thrown exceptions from square_root
  2. If an exception is somehow thrown despite the qualification, the program ends (which is what should happen).

- Destructors are implicitly declared noexcept
  1. **NEVER thrown an exception from a destructor**
  2. If you do, it will be treated as a run-time error and execution will end!

Standard exceptions

- <exception> header
- **std::exception** is a base class designed to be derived
- All objects thrown by members of the standard library are derived from this class
- Contains a virtual member function called **what** that returns a null-terminated char sequence (char *)
- Override this to deliver a meaningful exception message:

```
struct myexception: public exception {
  virtual const char* what() const noexcept
  {    return "My exception happened";  }
}
```

//exception.cpp, exception2.cpp

Derived from std::exception

Exception	Description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_function_call	thrown on a bad call
bad_typeid	thrown by typeid
logic_error	thrown when a logic error occurs
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr * we will see this later!

These exceptions are actually useful

<**stdexcept**> defines two exception types that can be inherited by custom exceptions to report errors:

1. logic_error:
   1. invalid_argument
   2. length_error //exception3.cpp
   3. out_of_range.
2. runtime_error:
   1. range_error
   2. overflow_error.

std::invalid_argument example

```
class Name
{
private:
  std::string first;
public:
  Name(std::string first) : first(first)
  {
    if (first.length() == 0)
    {
      throw std::invalid_argument("No first name!");
    }…
```

//exception4.cpp

What if we don't catch an exception?

- If an exception is not caught by any catch statement because there is no catch statement with a matching type, the special function **terminate** will be called.
- std::terminate is in <exception>
- Calls the termination handler
- The termination handler calls abort
- **CRASH AND BURN**

//exception5.cpp

Next week Review/midterm practice

- Submit request for review topics
- https://forms.gle/arUknhAArUQHq1HF8