CSE 1325: Object-Oriented Programming

Lecture 20

C++ Class Members

With Sets

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

A bicycle can't stand alone; it is two tired.

Today’s Topics

- Multiple Inheritance
- Class Members and Friends
  - Constructors, destructors, and static members
  - Friends and other functions
- The Rule of 3*
  - Copy constructors, destructors, and copy assignment operators
  - Initialization lists
- Sets

* Or 4. Or sometimes 5.
  They’re more *guidelines*, really...

Multiple Inheritance

- What if a subclass is derived from more than one superclass?
  - This is called *multiple inheritance*
  - You inherited traits from both your biological mother and father* – *multiple inheritance*
- With multiple inheritance, each superclass's members are laid out in memory after the subclass's members
  - Note that Java does NOT support multiple inheritance for *classes*, although it does for *interfaces*

Multiple Inheritance is a subclass inheriting class members from two or more superclasses.

* Unless you're a clone, in which case – *single inheritance*!

Multiple Inheritance in UML and C++

- In C++, just list multiple comma-separated superclasses

multiple_inheritance.cpp

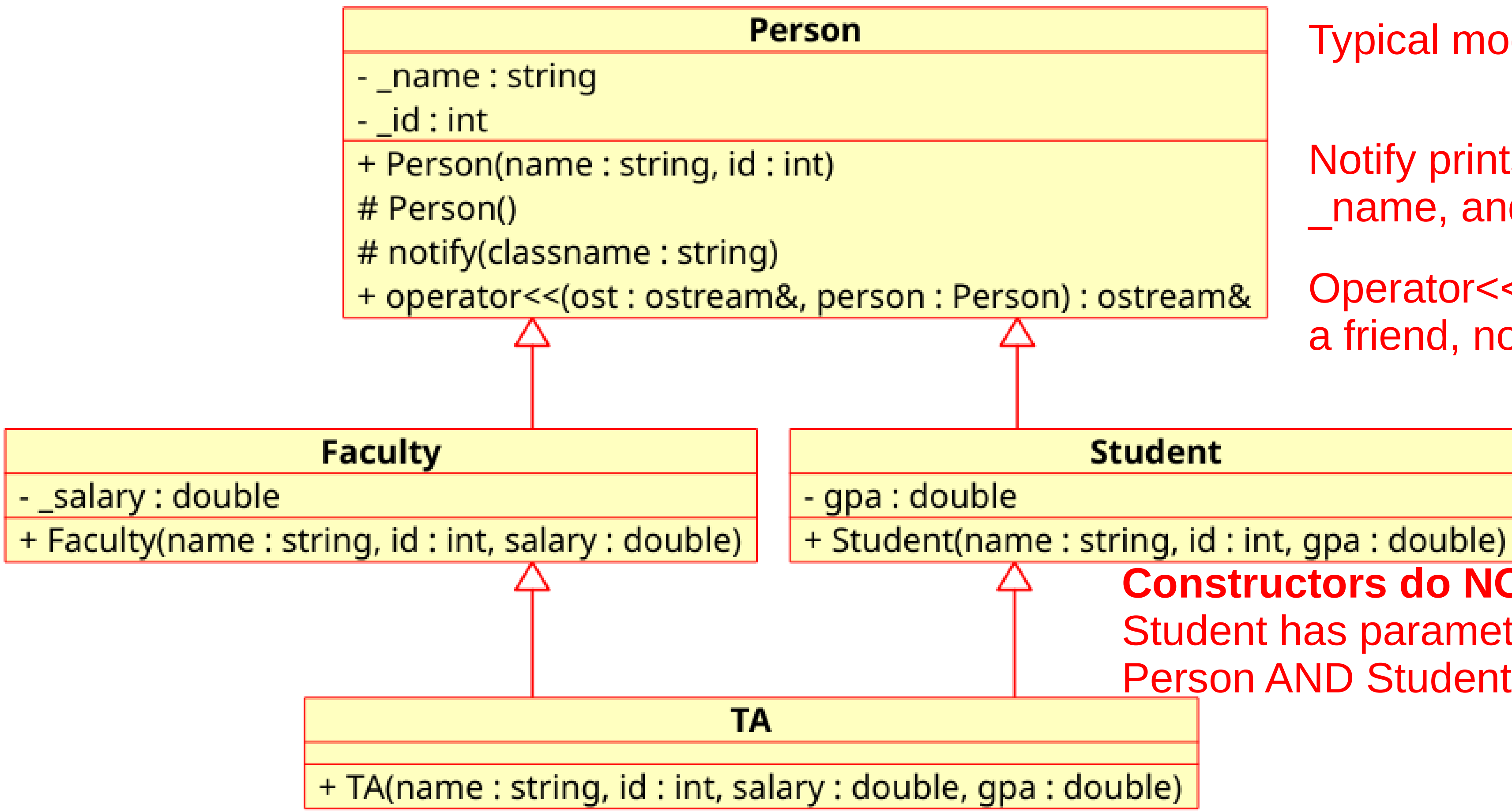
```
#include <iostream>
class A {
  public:
    A()  { std::cout << "A's constructor called" << std::endl;}
    virtual ~A() { std::cout << "A's destructor called" << std::endl;}
};
class B {
  public:
    B()  {std::cout << "B's constructor called" << std::endl;}
    virtual ~B() {std::cout << "B's destructor called" << std::endl;}
};
class C: public A, public B {
  public:
    C()  {std::cout << "C's constructor called" << std::endl;}
    virtual ~C() {std::cout << "C's destructor called" << std::endl;}
};
int main() {
    C c;
}
```

```
ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$ ./multi
A's constructor called
B's constructor called
C's constructor called
C's destructor called
B's destructor called
A's destructor called
ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$
```

Constructors are called in the order listed.
Destructors are called in the reverse order listed.

More Multiple Inheritance

Note: Destructors are omitted to conserve space



Typical modern Person

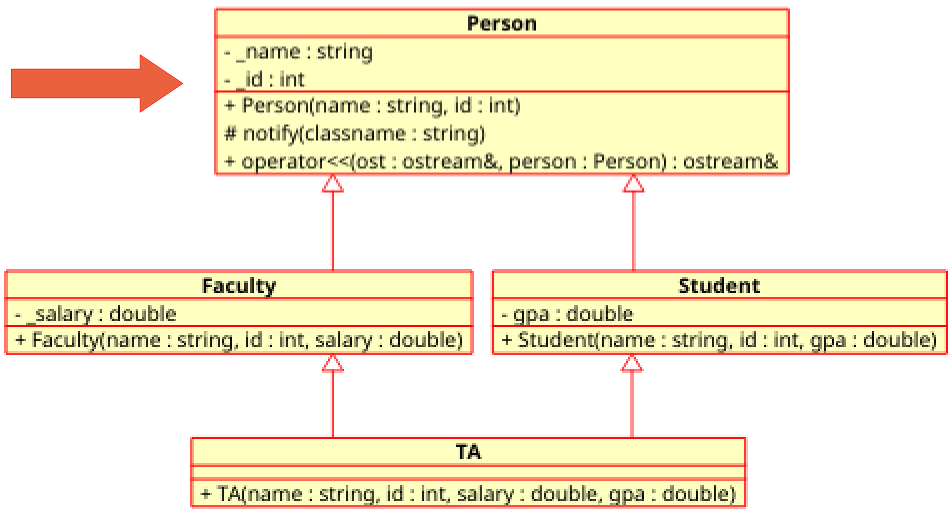
Notify prints classname, _name, and _id

Operator<< is (obviously) a friend, not a method

Constructors do NOT inherit!
Student has parameters for Person AND Student constructor.

A TA is both Faculty and Student. TA has parameters for each class from which it inherits, back to Person.

Class Person

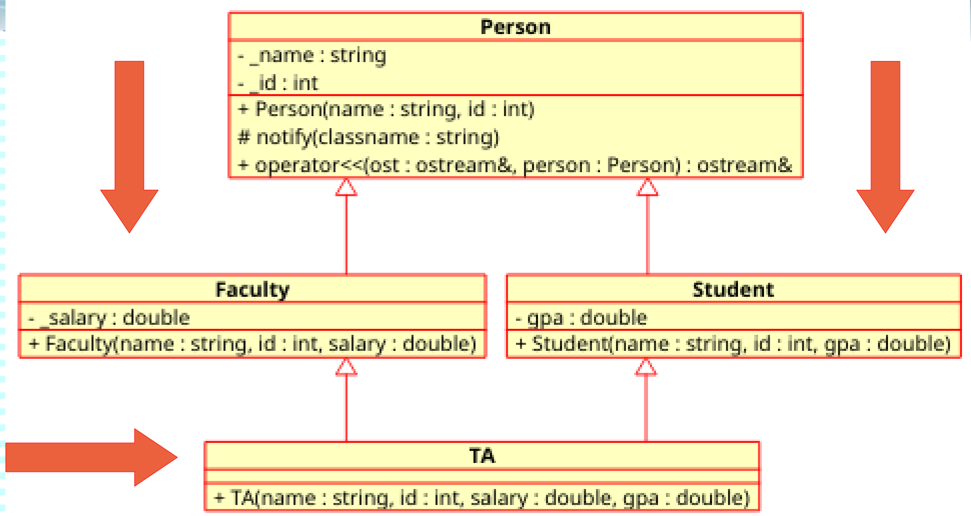



```cpp
#include <iostream>
#include <ostream>


class Person {
  public:
    Person(std::string name, int id)
        : _name{name}, _id{id}  {
        notify("Person");
    }
    friend std::ostream& operator<<(std::ostream& ost, Person& person);
  protected:
    void notify(std::string classname) {
        std::cout << classname << ' ' << _name << " constructed" << std::endl;
    }
  private:
    std::string _name;
    int _id;
};

std::ostream& operator<<(std::ostream& ost, Person& person) {
    ost << person._name << " (" << person._id << ')';
    return ost;
```

Notify is primarily used to announce execution of a constructor.
It's protected, and thus also available to Faculty, Student, and TA.

Faculty, Student, and TA instances are also Person instances.
They can also be streamed out with this overload!
(Preview of next week's lecture – don't miss it!)

Classes Faculty, Student, and TA



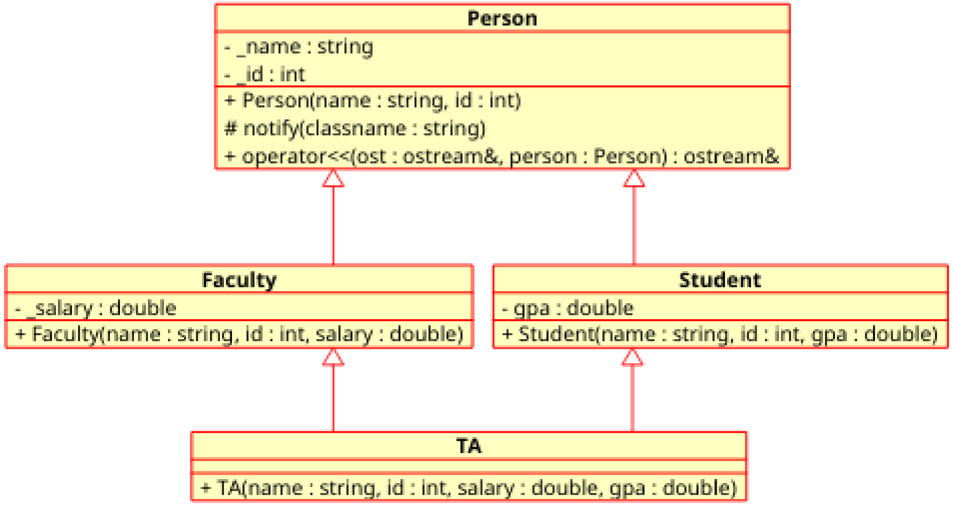

```
class Faculty : virtual public Person {
    double _salary;
  public:
    Faculty(std::string name,
            int id, double salary)
        : Person(name, id), _salary{salary} {
       notify("Faculty");
    }
};

class Student : virtual public Person {
    double _gpa;
  public:
    Student(std::string name, int id, double gpa)
        : Person(name, id), _gpa{gpa} {
        notify("Student");
    }
};

class TA : public Faculty, public Student {
  public:
    TA(std::string name, int id, double salary, double gpa)
        : Person(name, id), Student(name, id, gpa), Faculty(name, id, salary) {
        notify("TA");
    }
};
```

Faculty and Student first delegate to Person, then construct their own fields.

TA first delegates to Person, Student, and Faculty. **The order is irrelevant**: C++ will invoke each ancestor's constructor *exactly once* as specified *here*, in the order declared on the class declaration.

Main

```
student@cse1325:/media/sf_dev/07$ make ta
g++ --std=c++17 -c ta.cpp
g++ --std=c++17 -o ta ta.o
student@cse1325:/media/sf_dev/07$ ./ta
Person Wang Fang constructed
Faculty Wang Fang constructed
Student Wang Fang constructed
TA Wang Fang constructed
Our TA is Wang Fang (100032918)
student@cse1325:/media/sf_dev/07$
```



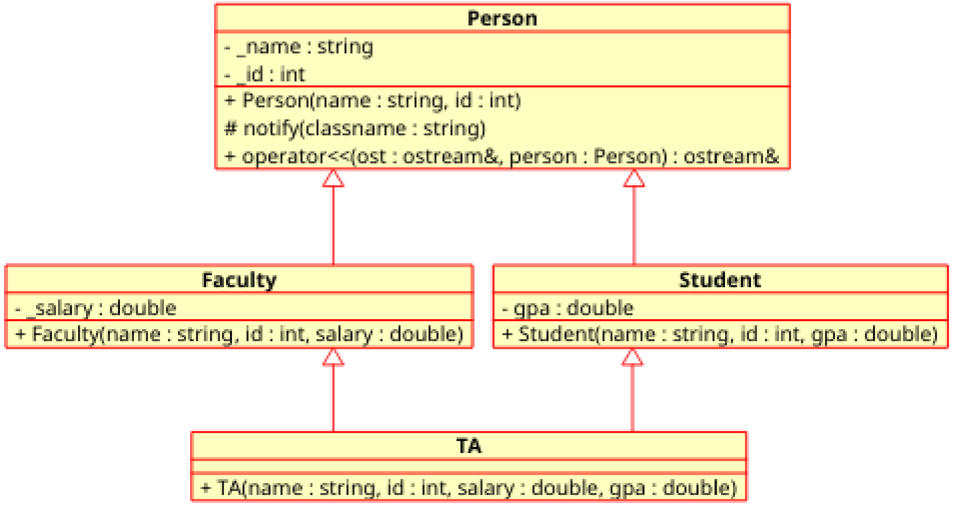

```cpp
int main()  {
    TA ta("Wang Fang", 100032918, 14.50, 3.92);
    std::cout << "Our TA is " << ta << std::endl;
}
```

Note that each class' constructor is called exactly once *as specified by class TA*. Delegation of a constructor is not "calling" that constructor; it merely specifies how that constructor should be invoked. C++ defines the actual order of invocation.

Need proof?

Feeding Bad Data to Student and Faculty Constructors as a Test

```
student@cse1325:/media/sf_dev/07$ make ta_test
g++ --std=c++17 -c ta_test.cpp
g++ --std=c++17 -o ta_test ta_test.o
student@cse1325:/media/sf_dev/07$ ./ta_test
Person Wang Fang constructed
Faculty Wang Fang constructed
Student Wang Fang constructed
TA Wang Fang constructed
Our TA is Wang Fang (100032918)
student@cse1325:/media/sf_dev/07$
```

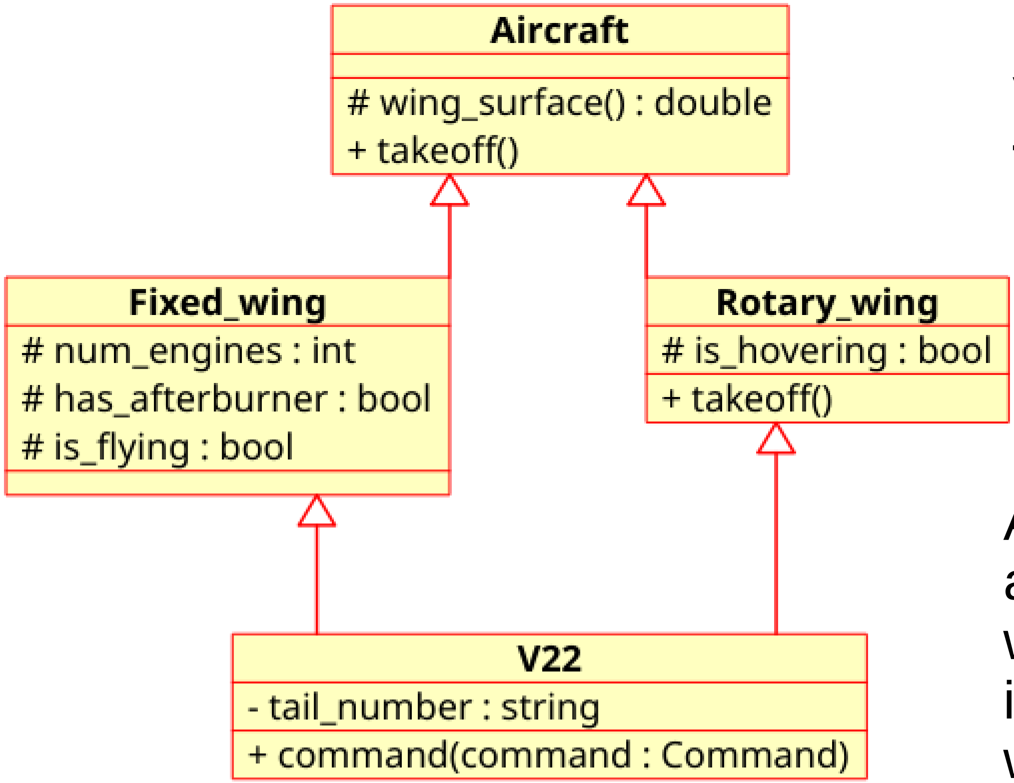



```
class TA : public Faculty, public Student  {
public:
    TA(std::string name, int id, double salary, double gpa)
        : Person(name, id), Student("", 0, gpa), Faculty("", 0, salary) {
        notify("TA");
    }
};

int main()  {
    TA ta("Wang Fang", 100032918, 14.50, 3.92);
    std::cout << "Our TA is " << ta << std::endl;
}
```

No difference! It doesn't *matter* that Student delegates to Person; C++ uses ONLY TA's delegation to construct Person as part of TA. If TA didn't delegate to Person, C++ would attempt to call Person{};

Multiple Inheritance Challenges
(And Solutions!)

Example with Multiple Inheritance



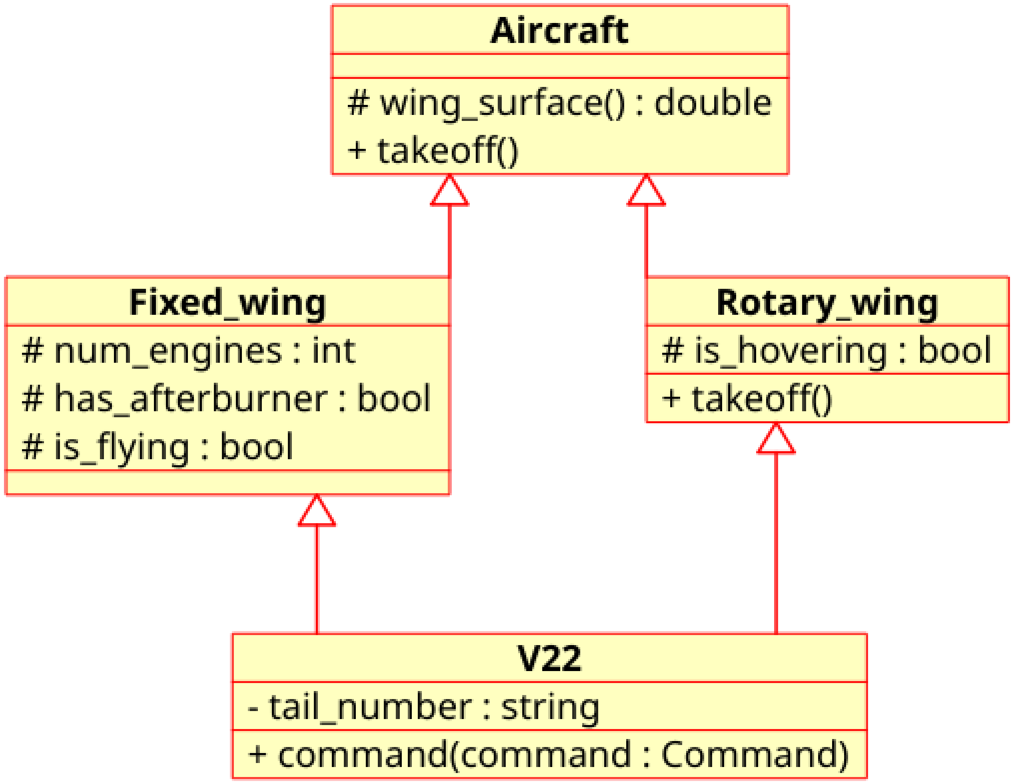
V22 would have wing_surface(), takeoff(), and command() methods

Wait a minute…
***which* takeoff()?**

And, more subtly, both the Fixed_wing and Rotary_wing classes inherit wing_surface(). Which path will inheritance follow, i.e., *which* inherited wing_surface() will V22 invoke?

https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem
http://www.cprogramming.com/tutorial/multiple_inheritance.html
http://www.cprogramming.com/tutorial/virtual_inheritance.html

Example with Multiple Inheritance



This is called the "**diamond problem**" - when inheriting in a "diamond shape", inheritance paths are no longer obvious.

Java, C#, and Ruby (for example) sidestep this problem and do **NOT** implement multiple inheritance*, using Interfaces (also called Protocols) instead.

Python makes inheritance order significant, and calls the first method found – left to right, then bottom to top.

How does C++ react?

https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem
http://www.cprogramming.com/tutorial/multiple_inheritance.html
http://www.cprogramming.com/tutorial/virtual_inheritance.html

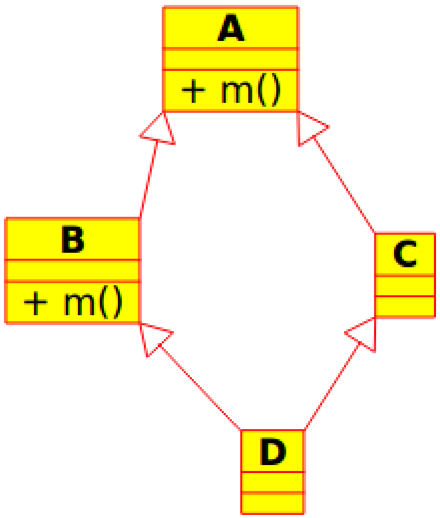
* Java SE 8 allows inheriting interface definitions, which is *sorta* multiple inheritance.

The ABCD Diamond

```
#include <iostream>

class A {
  public:
    virtual void m() {std::cout << "m of A" << std::endl;}
};

class B : public A {
  public:
    virtual void m() override {std::cout << "m of B" <<
std::endl;}
};

class C : public A { };

class D : public B, public C { };

int main() {
  D d;
  d.m();
}
```

"override" is the C++ version of Java's @Override – same effect

Let's simplify and test (a GREAT strategy for learning a new language!).

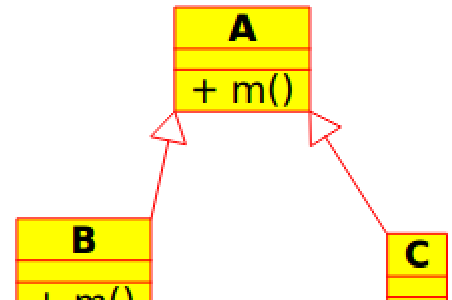


The Diamond Problem

```
#include <iostream>

class A {
  public:
    virtual void m() {std::cout << "m of A" << std::endl;}
};

class B : public A {
  public:
    virtual void m() override {std::cout << "m of B" << std::endl;}
};

class C : public A { };

class D : public B, public C

int main() {
  D d;
  d.m();
}
```

C++ (for once) refuses to make an assumption and instead raises a compiler error.

Let's simplify and test (a GREAT strategy for learning a new language!).




```
student@cse1325:/media/sf_dev/07$ make diamond_ambiguous
g++ --std=c++17    diamond_ambiguous.cpp   -o diamond_ambiguous
diamond_ambiguous.cpp: In function 'int main()':
diamond_ambiguous.cpp:19:5: error: request for member 'm' is ambiguous
   d.m();
     ^
diamond_ambiguous.cpp:5:18: note: candidates are: virtual void A::m()
     virtual void m() {std::cout << "m of A" << std::endl;}
                  ^
diamond_ambiguous.cpp:10:18: note:                 virtual void B::m()
     virtual void m() override {std::cout << "m of B" << std::endl;}
                  ^
<builtin>: recipe for target 'diamond_ambiguous' failed
make: *** [diamond_ambiguous] Error 1
student@cse1325:/media/sf_dev/07$
```

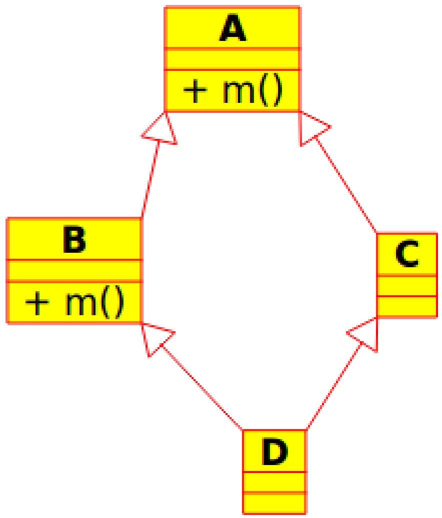
The Diamond Resolved... In Part

```
#include <iostream>

class A {
  public:
    virtual void m() {std::cout << "m of A" << std::endl;}
};

class B : public A {
  public:
    virtual void m() override {std::cout << "m of B" <<
std::endl;}
};

class C : public A { };

class D : public B, public C { };

int main() {
  D d;
  d.B::m();
  d.C::m();
}
```

C++ (for once) follows our explicit selection of the m() we want to call.

Ambiguity resolved!

What if we specify the method using namespace resolution operator?




```
student@cse1325:/media/sf_dev/07$ make diamond_explicit
g++ --std=c++17    diamond_explicit.cpp   -o diamond_explicit
student@cse1325:/media/sf_dev/07$ ./diamond_explicit
m of B
m of A
student@cse1325:/media/sf_dev/07$
```

The Diamond Problem... In Full

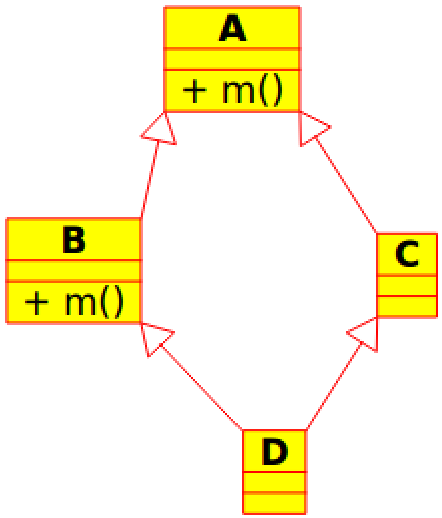
```
#include <iostream>

class A {
  public:
    virtual void m() {std::cout << "m of A" << std::endl;}
};

class B : public A {
  public:
    virtual void m() override {std::cout << "m of B" <<
std::endl;}
};

class C : public A { };

class D : public B, public C { };

int main() {
  D d;
  d.A::m();
  d.B::m();
  d.C::m();
}
```

C++ claims A::m() is ambiguous – that our code has *more than one A*. How is that possible???

Wait... what???

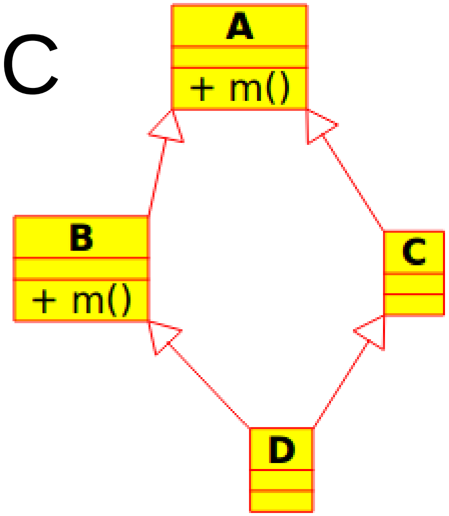
```
student@cse1325:/media/sf_dev/07$ make diamond2_explicit
g++ --std=c++17     diamond2_explicit.cpp   -o diamond2_explicit
diamond2_explicit.cpp: In function ‘int main()’:
diamond2_explicit.cpp:19:8: error: ‘A’ is an ambiguous base of ‘D’
    d.A::m();
         ^
<builtin>: recipe for target 'diamond2_explicit' failed
make: *** [diamond2_explicit] Error 1
student@cse1325:/media/sf_dev/07@
```

What if we want to invoke A’s m method directly?



A Exists Twice in the Diamond (!)

- B inherits a complete copy of A
- C inherits a complete copy of A
- D inherits a complete copy of B AND C
  - And thus B's copy of A
  - And thus C's copy of A
- Yep – D has TWO copies of A
  - So A::m is ambiguous – do you want B's A or C's A?

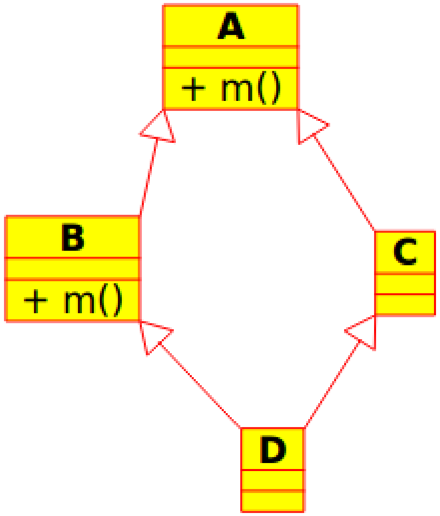


Fully Resolving the Diamond Problem

```
#include <iostream>

class A {
  public:
    virtual void m() {std::cout << "m of A" << std::endl;}
};

class B : virtual public A {
  public:
    virtual void m() override {std::cout << "m of B" <<
std::endl;}
};

class C : virtual public A { };

class D : public B, public C { };

int main() {
  D d;
  d.A::m();
  d.B::m();
  d.C::m();
}
```

MUCH better!

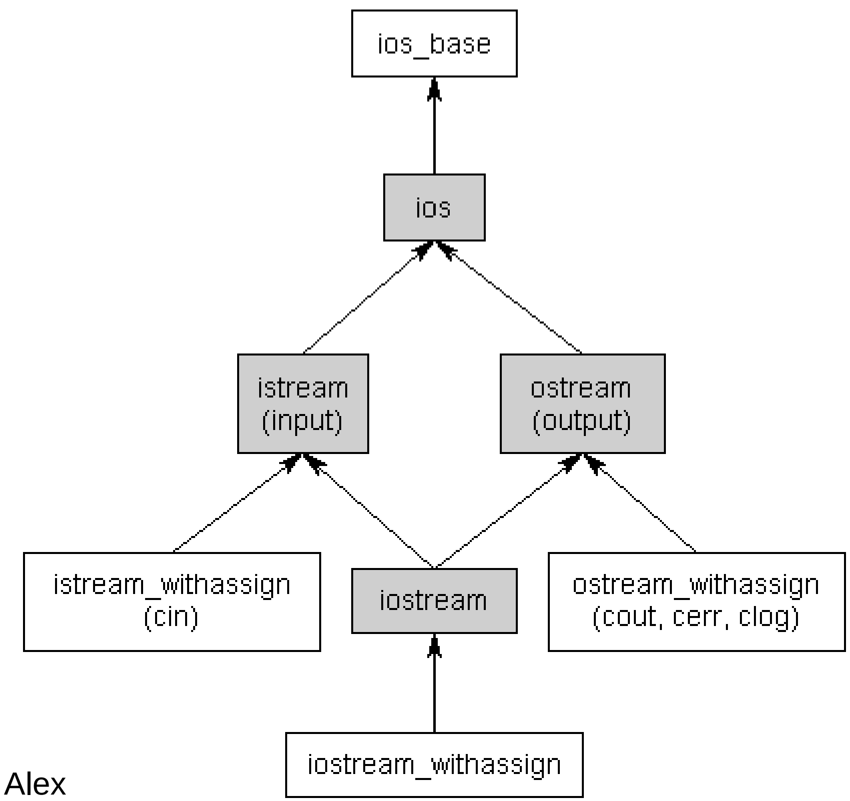
If B and C inherit virtually from A – that is, they *share* a copy of A – the ambiguity is resolved.

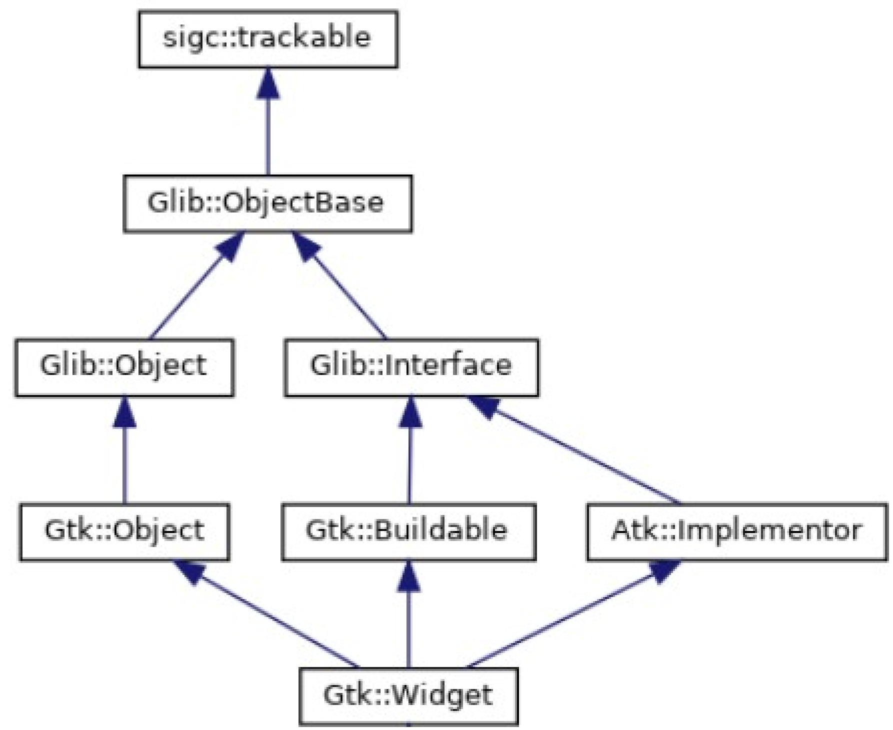



```
student@cse1325:/media/sf_dev/07$ make diamond2
g++ --std=c++17     diamond2.cpp    -o diamond2
student@cse1325:/media/sf_dev/07$ ./diamond2
m of A
m of B
m of A
student@cse1325:/media/sf_dev/07$
```

Wither Multiple Inheritance?

- You will rarely require Multiple inheritance
  - Java and C# do NOT support multiple inheritance of *classes*, but DO support it for *interfaces* (pure abstract classes)
  - But it's often useful, as for the C++ I/O and GUI libraries below





https://www.learncpp.com/cpp-tutorial/181-input-and-output-io-streams/

https://developer.gnome.org/gtkmm/stable/classGtk_1_1Widget.html

Complex Number Class Example

We'll Work With This One Awhile!

- Here's an example of a class representing a complex number (contrived a bit to illustrate some important concepts!)

```
#ifndef __COMPLEX_H
#define __COMPLEX_H
#include <string>
typedef double Radians;                              // Type Definition

// Manages complex numbers
class Complex {
  public:
    Complex(double re, double im);                   // Constructor
    virtual ~Complex();                             // Destructor
    friend Complex Polar(double r, Radians theta);  // Friend Function
    std::string to_string();                         // Method
    static void set_polar(bool p);                   // Static method
    static bool get_polar();                         // Static method
  private:
    double _re, _im;                                 // Non-static fields (1 set per object)
    static bool polar;                               // Static fields (1 shared object)
};
Complex Polar(double r, Radians theta);
void print(Complex number);                          // Function
// Friend function Polar is the Factory pattern that "instances" a double
//   using polar coordinates instead of rectangular (as the constructor uses)
// This pattern enables us to define "constructors" for a class with a different name
//   and behavior than a proper constructor.
#endif
```

NOT a class member!

NOT a class member!

typedef

- Typedef allows us to rename an existing type
  - This is “syntactic sugar” - you may freely exchange data between variables declared with either name
  - Use this to make your code more maintainable!

typedef ***existing_type*** ***new_name***

```
typedef double Altitude;         // Altitude now is the “double” type
Altitude altitude;               // More natural than “double altitude”
std::vector<Altitude> skyroads; // MUCH more natural!

// This is PARTICULARLY useful with nested types

typedef std::vector<double> Row;           // As in a spreadsheet
typedef std::vector<Row> Ragged_array;     // Each row can be a different length
// FAR better than
// std::vector<std::vector<double>> Ragged_array;  // Yikes!
```

complex.h

```
typedef double Radians;  // Radians is a clearer name than double for Complex
```

Constructors

- **A constructor is <u>not</u> a method,** but is invoked when a variable of the type is declared to *construct* it

The constructor has the same name as the class and *no return type*. It is otherwise very similar to a method. If you don't specify any constructors, a default will be provided.

We usually start with the *public* declarations first, including the constructors, as this is the *interface* to the class.

Save the private declarations for the end.

```
// Manages complex numbers
class Complex {
   public:
      Complex(double re, double im);
      virtual ~Complex();
      friend Complex Polar(double r, Radians theta);
      std::string to_string();
      static void set_polar(bool p);
```

NOT constructors!

class, its constructor is invoked like calling a function but using curly braces.*

```
int main() {
   Complex c{3.0,4.0};
   std::cout << c.to_string();
}
```

If no curly braces are provided, the default constructor with no parameters is automatically invoked if available, otherwise an error is thrown.

* Parentheses often work, too, for historical reasons. But don't use parentheses.

Destructors

- **A destructor is an "inverse constructor",** to clean up after an object is deleted. It is often omitted when unneeded.

The destructor starts with ~ followed by the class name and *no return type*.

- Stack variables – invoked when exiting scope
- Heap variables – invoked by delete keyword

```
// Manages complex numbers
class Complex {
  public:
    Complex(double re, double im);
    virtual ~Complex();
    friend Complex Polar(double r, Radians theta);
    std::string to_string();
    static void set_polar(bool p);
```

The "new" keyword instances an object on the heap rather than the stack, returning a pointer.

Call methods from a pointer with -> instead of .

```
int main() {
  Complex* c = new Complex{3.0,4.0};
  std::cout << c->to_string();
  delete c;
}
```

The delete keyword invokes the destructor for heap variables ONLY, although in this case it does nothing. (You could add a std::cout statement to the destructor to watch it run. ☺)

Functions

- A function is a named group of statements that perform a task

```
int square(int i) {return i*i;}
```

Function max compares two values and returns the larger

```
int max(int a, int b) {  // this function takes 2 parameters
    if (a<b) return b;
    else     return a;
}
```

A shorter one-line version using the ternary operator

```
int max(int a, int b) {return (a<b) ? b : a;}
```

Avoid writing C++ functions where possible; use *methods*
But in C++, sometimes they are just much more convenient

complex.h

```
Complex Polar(double r, Radians theta);  // Just another C++ function...
```

C++ also supports variadic, or variable length, parameter lists. Google is your friend if you're interested in that.

Compile-time Functions and Vars

- You can define functions and variables that *can be* evaluated *at compile time* using **constexpr**
  - A definition usually results in memory allocation
  - A constexpr does not – the compiler simply substitutes the value of the expression as a constant wherever referenced

```
constexpr double xscale = 10;     // scaling factors
constexpr double yscale = .8;

constexpr Point scale(Point p) { return {xscale*p.x, yscale*p.y}; };

constexpr Point x = scale({123,456});  // evaluated at compile time

void use(Point p) {
    constexpr Point x1 = scale(p);   // error: compile-time evaluation
                                     // requested for a variable argument
    Point x2 = scale(p);             // OK: run-time evaluation
}
```

Friend Functions and Classes

- Classes can have friends
  - **Friends are NOT class members**, but they have access to the class' private class members
  - A class or function may be friends with 0, 1, or more than 1 class
  - Friendship is unidirectional – if class B is a friend of class A, class A is NOT necessarily a friend of class B (although B could befriend A right back)
- Because function Polar is a friend of Complex, it can modify the private field Complex::polar

```
friend Complex Polar(double r, Radians theta);
```

Methods

(or "Class Functions")

- A *method** is a function within a class scope, which (unlike a non-friend function) has access to its private variables

```
class Complex {
  public:
    Complex(double re, double im);
    std::string to_string();
    friend Complex Polar(double r, Radians theta);
    static void set_polar(bool p);
    static bool get_polar();
  private:
    double _re, _im;
    static bool polar;
};
```

NOT methods

Methods

- A method** can only be called on an *instance* of the class

```
int main() {
  Complex c{3.0,4.0};
  std::cout << c.to_string() << std::endl;
}
```

Instance

Method

* Sometimes called a "class function"

** Technically a *non-static* method. We'll get to this distinction next.

Static Class Members

- A static method or variable exists as part of the class, and it's memory location shared among all instances
  - A **static method** may be called without instancing an object, but cannot access any non-static members of the class
  - A **static field** (variable) <u>must</u> be defined outside the class

complex.h

```
    static void set_polar(bool p);
    static bool get_polar();
  private:
    double _re, _im;
    static bool polar;
```

complex.cpp

```
void Complex::set_polar(bool p) {polar = p;}
bool Complex::get_polar() {return polar;}

// The static variable must also be defined in the .cpp to allocate its memory
bool Complex::polar = false;
```

main.cpp

```
    // Instance c2 with polar coords,then reset output form and print rectangular
    Complex c2 = Polar(10.0, 0.6435);
    Complex::set_polar(false);  // Direct call OR c2.set_polar(false);
    std::cout << c2.to_string() << std::endl;
```

Default Class Members

- C++ classes provide 4 essential memory management-related members by default
  - **Default constructor** IF no other constructor is defined (defaults to calling default constructor for each field)
  - **Copy constructor** (defaults to: copy the corresponding member values)
  - **Copy assignment operator** (defaults to: copy the corresponding member values)
  - **Destructor** (defaults to: nothing)
- These 4 members work together to manage class memory
  - If a custom constructor allocates heap memory, it must be handled by custom copy constructor, copy assignment operator, and destructor

Destructors are described in detail in Learn C++ chapter 15.4 Destructors

C++-Provided Default Constructor

```cpp
#include <iostream>

class Book {
  Public:    // NO explicit constructor
    void set_book(std::string title, int pages) {_title=title; _pages=pages;}

    // preview of overloading the << operator (coming soon!) :)
    friend std::ostream& operator<<(std::ostream& ost, const Book& book) {
        ost << book._title << " (" << book._pages << " pages)";
        return ost;
    }
  private:
    std::string _title;
    int _pages;
};

int main() {
    Book book;
    std::cout << book << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/cse1325-prof/06/code_from_slides@ make book
g++ --std=c++17 -g    book.cpp    -o book
ricegf@pluto:~/dev/cpp/cse1325-prof/06/code_from_slides$ ./book
 (4197696 pages)
ricegf@pluto:~/dev/cpp/cse1325-prof/06/code_from_slides$
```

Explicit Constructors

```cpp
#include <iostream>

class Book {
  public:
    Book(std::string title, int pages) : _title{title}, _pages{pages} { }
    Book() : Book("Unknown", 0) { }

    // preview of overloading the << operator (coming soon!) :)
    friend std::ostream& operator<<(std::ostream& ost, const Book& book) {
        ost << book._title << " (" << book._pages << " pages)";
        return ost;
    }
  private:
    std::string _title;
    int _pages;
};

int main() {
    Book book1;
    Book book2{"War and Peace", 1225};
    std::cout << book1 << '\n' << book2 << std::endl;
}
```

Constructor delegation (or chaining)

```
ricegf@pluto:~/dev/cpp/cse1325-prof/06/code_from_slides$ make book
g++ --std=c++17 -g    book.cpp   -o book
ricegf@pluto:~/dev/cpp/cse1325-prof/06/code_from_slides$ ./book
Unknown (0 pages)
War and Peace (1225 pages)
ricegf@pluto:~/dev/cpp/cse1325-prof/06/code_from_slides$
```

Default Parameters

- In C++, we can replace this...

```
class Book {
  public:
    Book(std::string title, int pages) : _title{title}, _pages{pages} { }
    Book() : Book("Unknown", 0) { }
```

- … with this

```
class Book {
  public:
    Book(std::string title = "Unknown", int pages = 0) : _title{title}, _pages{pages} { }
```

(Virtual) Destructors

- We can also explicitly declare the default destructor

```
class Book {
  public:
    Book(std::string title = "Unknown", int pages = 0) : _title{title}, _pages{pages} { }
    ~Book() { }
```

- If any methods are virtual, meaning we expect to have subclasses with polymorphic method calls, the destructor should be declared virtual
  - For this course, we'll simply always declare virtual destructors*

```
class Book {
  public:
    Book(std::string title = "Unknown", int pages = 0) : _title{title}, _pages{pages} { }
    virtual ~Book() { }
```

* This isn't the default in C++ because of the excessive memory use it would impose on small, non-polymorphic classes

Initialization vs Assignment

This is important for understanding copy constructors vs copy assignment

- Initialization causes *new object r1* to have the same value as existing object r2

Create

  - `Robot r1 = r2; // initialization`
  - `Robot r1{r2};  // initialization – exactly the same`

- Assignment causes *existing object r1* to have the same value as another existing object r2

  - `Robot r1;`
  - `r1 = r2; // assignment`

Replace

- These are two distinct operations in C++

Default Copy Constructors and Copy Assignment

```
student@cse1325:/media/sf_dev/21$ make cc_and_cao2
g++ -std=c++17     cc_and_cao2.cpp    -o cc_and_cao2
student@cse1325:/media/sf_dev/21$ ./cc_and_cao2
bar0 = 0
bar1 = 1
bar2 = 1
bar0 now = 1
student@cse1325:/media/sf_dev/21$
```

```cpp
#include <iostream>

class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {} // Non-default constructor
    Foo() : Foo(0) {}           // Default constructor
    int val() {return _val;}
};

int main() {
  Foo bar0;             // Default constructor
  std::cout << "bar0 = " << bar0.val() << endl;
  Foo bar1{1};          // Non-default constructor
  std::cout << "bar1 = " << bar1.val() << endl;
  Foo bar2{bar1};       // Default copy constructor for initialization
  // Foo bar2 = bar1; // Exactly the same thing: bar2 has same values as bar1
  std::cout << "bar2 = " << bar2.val() << endl;
  bar0 = bar1;          // Default copy assignment for assignment
  std::cout << "bar0 now = " << bar0.val() << endl;
}
```

Copy Constructor

- For initialization, C++ actually invokes a special constructor – the copy constructor
  - If you have not specified one, you'll get the default – all of your variables will be copied directly across
- The copy constructor is thus invoked when:
  - You initialize a *new* object to an existing object
    - `Foo bar2 = bar1;  // Identical to Foo bar2{bar1};`
  - You pass an object as a non-reference parameter
    - `analyze(bar1); // A copy of bar1 is created for analyze`
  - You return an object from a function
    - `return bar1;  // A copy of bar1 is created and returned`

Declaring a Copy Constructor

- A copy constructor just accepts a const reference to the object to be copied
  - I *know* this isn't useful here – we'll come back to when writing a copy constructor is useful shortly

```
#include <iostream>

class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                   // Non-default constructor
    Foo() : Foo(0) {}                             // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}  // Copy constructor
    int val() {return _val;}
};
```

Copy Assignment

- For assignment, C++ invokes the assignment operator to overwrite the left-hand object's values
  - If you have not specified one, you'll get the default – all of your variables will be copied directly across
- The copy assignment is thus invoked when:
  - You assign to an existing object the value of another (*or the same*) existing object
    - `bar2 = bar1;  // bar2 was existing, so we overwrite it,`
      `              //   invoking the copy assignment operator`

Declaring a Copy Assignment

- A copy assignment is a definition of the = operator to handle copying members as needed
  - Not useful here, either – almost there!

```
class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                  // Non-default constructor
    Foo() : Foo(0) {}                            // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}  // Copy constructor
    Foo& operator=(const Foo &rhs) {             // Copy assignment
      if (this != &rhs) _val = rhs.val();
      return *this;
    }
    int val() const {return _val;}
};
```

Destructors

- The destructor is invoked when the object itself is being deleted
  - *Really* useless here! Next slide, promise!

```
class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                   // Non-default constructor
    Foo() : Foo(0) {}                             // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}  // Copy constructor
    Foo& operator=(const Foo &rhs) {              // Copy assignment
      if (this != &rhs) _val = rhs.val();
      return *this;
    }
    virtual ~Foo() {}                            // Destructor
    int val() const {return _val;}
};
```

Practical Use and the Rule of Three

- So when would copy constructors, copy assignment, and destructors actually be *useful*?
  - When your class allocates memory from the heap, keeping a pointer to its address
    - To copy the object, you also need to "deep copy" the allocated heap memory to another heap memory area
    - To assign the object, you need to deallocate the existing heap memory before doing the "deep copy" to allocate the new heap memory
    - To delete the object, you also need to delete the heap memory

Rule of Three: If you define <u>one</u> of the above,
you probably need to define <u>all three</u> of the above!

For more detail, I recommend the following paper. This isn't required for the exam.
http://www.keithschwarz.com/cs106l/winter20072008/handouts/170_Copy_Constructor_Assignment_Operator.pdf

Rule of 3 Implementation of Foo

- Allocating and deallocating private variables
- Copy and assignment allocate new heap
  - Rather than pointing to the *same* heap!

```
class Foo {
    int* _val;
  public:
    Foo(int val) : _val{new int{val}} {}                  // Non-default constructor
    Foo() : Foo(0) {}                                     // Chained constructor
    Foo(const Foo &rhs) : _val{new int{rhs.get()}} {} // Copy constructor
    Foo& operator=(const Foo &rhs) {                      // Copy assignment operator
      if (this != &rhs) _val = new int{rhs.get()};
      return *this;
    }
    virtual ~Foo() {delete _val;}                         // Destructor
    int get() const {return *_val;}                       // Getter
    void set(int v) {*_val = v;}                          // Setter
};
```

Resource Acquisition Is Initialization (RAII), also known as
Constructor Acquires, Destructor Releases (CADRe)

Instrumented
Rule of 3 Implementation

```
class Foo {
    int* _val;
  public:
    Foo(int val) : _val{new int{val}}
        {log("constructor");}                       // Non-default constructor
    Foo() : Foo(0)
        {log("default constructor");}               // Default constructor
    Foo(const Foo &rhs) : _val{new int{rhs.get()}}
        {log("copy constructor");}                  // Copy constructor
    Foo& operator=(const Foo &rhs) {                // Copy assignment operator
      if (this != &rhs) _val = new int{rhs.get()};
      log("copy assignment operator");
      return *this;
    }
    virtual ~Foo()
        {log("destructor"); delete _val;}           // Destructor
    int get() const
        {log("getter"); return *_val;}              // Getter
    void set(int v)
        {log("setter"); *_val = v;}                 // Setter
  private:
    void log(std::string s) const
        {std::cerr << "[" << s << "] ";}
};
```

Testing the Rule of 3 Implementation

```
void print(Foo foo) { // Pass by value!
    std::cout << "In method print, foo = " << foo.get();
}

int main() {
    {
        std::cout << "\n1: "; Foo foo1;
        {
            std::cout << "\n2: "; Foo foo2 = foo1;
            std::cout << "\n3: "; foo1.set(42);
            std::cout << "\n4: "; print(foo2);
            std::cout << "\n5: ";
        }
        std::cout << "\n6: "; print(foo1);
        std::cout << "\n7: ";
    }
    std::cout << std::endl;
}
```

Line numbering the output

Comparing this code to the previous page,
what text will be printed for each line number?

Rule of 3 Implementation Activity Explained

```
void print(Foo foo) { // Pass by value!
    std::cout << "In method print, foo = " << foo.get();
}

int main() {
    {
        std::cout << "\n1: "; Foo foo1;
        {
            std::cout << "\n2: "; Foo foo2 = foo1;
            std::cout << "\n3: "; foo1.set(42);
            std::cout << "\n4: "; print(foo2);
            std::cout << "\n5: ";
        }
        std::cout << "\n6: "; print(foo1);
        std::cout << "\n7: ";
    }
    std::cout << std::endl;
}
```

```
1: [constructor] [default constructor]   Constructor chaining
2: [getter] [copy constructor] Copy constructor (NOT assignment), using the getter
3: [setter]                    vv Pass by value (copy) and then return (destroy)
4: [getter] [copy constructor] In method print, foo = [getter] 0[destructor]
5: [destructor]   End scope so foo2 is destroyed
6: [getter] [copy constructor] In method print, foo = [getter] 42[destructor]
7: [destructor]   End scope so foo1 is destroyed
```

Associative Container
set

- **set** is a sorted collection of values (like Java's TreeSet)
  - Essentially a vector of objects with duplicates automatically removed, and always sorted
- As with Java's TreeSet, you may need to define a comparator method such as **`operator<`**
  - In C++, you can define these even for library classes using functions!

http://www.cplusplus.com/reference/set/set/

Common Set Operations

- **s.empty()** is true if the set contains no values
  - **s.clear()** removes all values from the set
- **s.size()** returns the number of values in the set
  - Random access (indexing) isn't supported
- **s.insert(value)** adds value to the set
  - **s.count(value)** returns 1 if value exists, 0 otherwise
  - **s.erase(value)** removes value from the set
- **for(auto& value : s)** iterates over the set values
- **s.begin()** and **s.end()** return "iterators" to the first and one past the last value, which behave like pointers (soon!)

Simple Set Example

List Unique Arguments in Sort Order

```cpp
#include <set>
#include <iostream>

int main(int argc, char* argv[]) {
    std::set<std::string> words;
    for(int i=1; i<argc; ++i)
        words.insert(std::string{argv[i]});
    std::cout << "Here are the unique arguments in alphabetical order:" << std::endl;
    for(auto word : words)
        std::cout << "  " << word << std::endl;
}
```

```
ricegf@antares:~/dev$ ./a.out This is a good time for a great time just in time
Here are the unique arguments in alphabetical order:
  This
  a
  for
  good
  great
  in
  is
  just
  time
ricegf@antares:~/dev$
```

Simple Set Example
Search for Words in a Set

```cpp
#include <set>
#include <iostream>

int main(int argc, char* argv[]) {
    std::set<std::string> words;
    for(int i=1; i<argc; ++i)
        words.insert(std::string{argv[i]});

    std::string word;
    while(true) {
        std::cout << "Search for which word in arguments? ";
        std::cin >> word;
        if(word.empty()) break;
        std::cout << word << ((words.count(word) == 0) ? " is not " : " is ")
                  << "in the argument list" << std::endl;
    }
}
```

```
ricegf@antares:~/dev$ ./a.out when in the course of human events
Search for which word in arguments? course
course is in the argument list
Search for which word in arguments? despot
despot is not in the argument list
Search for which word in arguments? despot
despot is not in the argument list
Search for which word in arguments?
```

Another Set Example

Random Search for Primes

```cpp
#include <set>
#include <iostream>
#include <cmath>

// Returns true if "number" is a prime number
bool is_prime (int number) {
    if (number < 2) return false;
    for (int i=2; i <= std::sqrt(number); ++i) {
        if ((number % i) == 0) return false;
    }
    return true;
}
int main() {
    std::set<int> s;
    for(int i=1; i<=100; ++i) {
        int x = rand()%100;
        if(is_prime(x))
            s.insert(x);
    }
    for(auto i : s) std::cout << i << '\n';
}
```

```
ricegf@antares:~/dev$ ./a.out
2
3
5
11
13
19
23
29
37
43
59
67
73
83
ricegf@antares:~/dev$
```

Variations

- C++ also has an unsorted set similar to Java’s HashSet
  - Called unordered_set
  - Not as commonly used in C++ as HashSet in Java
- C++ also has versions of set and unsorted_set that accept duplicates
  - Called multiset and unordered_multiset
  - In these cases, the count() method may be >1

What We Learned Today

- Multiple Inheritance allows multiple superclasses
  - Specific ALL non-default superclass constructors in each init list!
  - Solve the “diamond problem” with *virtual* inheritance
- Messing with types and values
  - Typedef renames first type to second for clarity
  - Constexpr evaluates an expression at compile time
- *Define* static fields from the .h file in the .cpp file to allocate memory
- Carefully manage C++ memory (especially for “deep” copies)
  - Use copy constructors, copy assignment operators, and destructors
  - Rule of 3: If you need one, you probably need all 3!
- C++ set is like Java’s TreeSet
  - unordered_set is like Java’s HashSet