C++
Programming with Class(es)

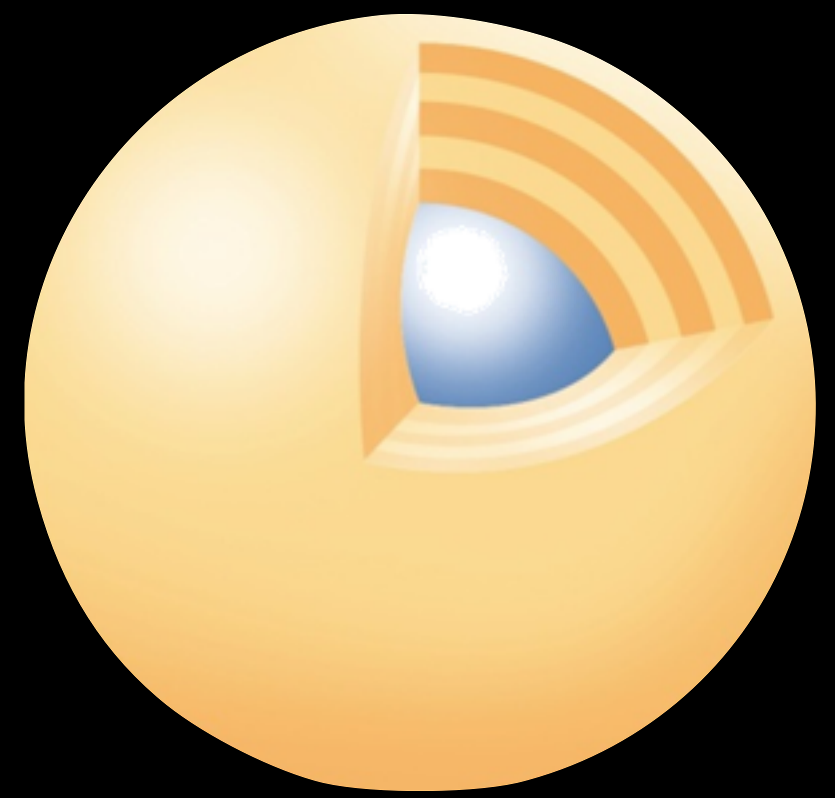
Laurent Michel
Computer Science & Engineering

Outline

- Encapsulation
- Inheritance
- Polymorphism
- Delegation

Encapsulation

- The famous “Has-a” relationship
- Purpose
  - Bundle “things” together as a cohesive package
- Note
  - We already used it!
  - **`struct`** and **`class`**!



Purpose

- Bundle related entities to form a **highly cohesive** new entity
- Attributes can be
  - Scalars
  - Arrays
  - Objects

Encapsulation in C++

- Language abstractions

struct | **class**

- Same mechanics
- Only difference is privacy

Pitfall

- Quite Common in Java
- Easier to avoid in C++
- A result of the policy “every object is on the heap”
- Classic example
  - The rectangle

Points

```
class Point {
  double _x,_y;
public:
  Point() { _x = _y = 0;}
  Point(double x,double y) { _x = x;_y = y;}
  Point(const Point& p) : _x(p._x),_y(p._y) {}
  double getX() const { return _x;}
  double getY() const { return _y;}
  void set(double x,double y) { _x = x;_y = y;}
  Point& operator=(const Point& p) { _x = p._x;_y = p._y;return *this;}
  friend Point operator+(const Point& p1,const Point& p2);
  friend Point operator*(const Point& p1,double s);
  friend std::ostream& operator<<(std::ostream& os,const Point& p);
};
```

Rectangles

```
class Rectangle {
  std::shared_ptr<Point> _corner;
  double _w,_h;
public:
  Rectangle() : _corner(new Point),_w(0),_h(0) {}
  Rectangle(double x,double y,double w,double h)
    : _corner(new Point(x,y)),_w(w),_h(h) {}
  std::shared_ptr<Point> getCorner() const { return _corner;}
  double getWidth()  const { return _w;}
  double getHeight() const { return _h;}
  friend std::ostream& operator<<(std::ostream& os,const Rectangle& r) {
    return os << "rect(" << *r._corner << ","
              << r._w << "," << r._h << ")";
  }
};
```

What’s the issue?

UCONN

Breaking Encapsulation

- You “expose” the corner to the outside
  - It can be modified externally!

Example

```
#include "rect.H"
#include <iostream>

int main()
{
    Rectangle r1(10,20,100,100);
    std::cout << "Before:" << r1 << std::endl;
    Point p1(42,42);
    *r1.getCorner() =  p1;
    std::cout << "after :" << r1 << std::endl;
    return  0;
}
```

Fixing the Problem

```
class Rectangle {
   Point _corner;
   double  _w,_h;
public:
   Rectangle() : _w(0),_h(0) {}
   Rectangle(double x,double y,double w,double h)
      : _corner(x,y),_w(w),_h(h) {}
   Point getCorner() const  { return _corner;}
   double getWidth()  const { return _w;}
   double getHeight() const { return _h;}
   friend std::ostream& operator<<(std::ostream& os,const Rectangle& r) {
      return os << "rect(" << *r._corner << "," << r._w << "," << r._h << ")";
   }
};
```

Inheritance

- What it promotes
  - Code reuse
  - Code specialization
- Related concepts
  - Sub-typing
  - Sub-classing

Purpose

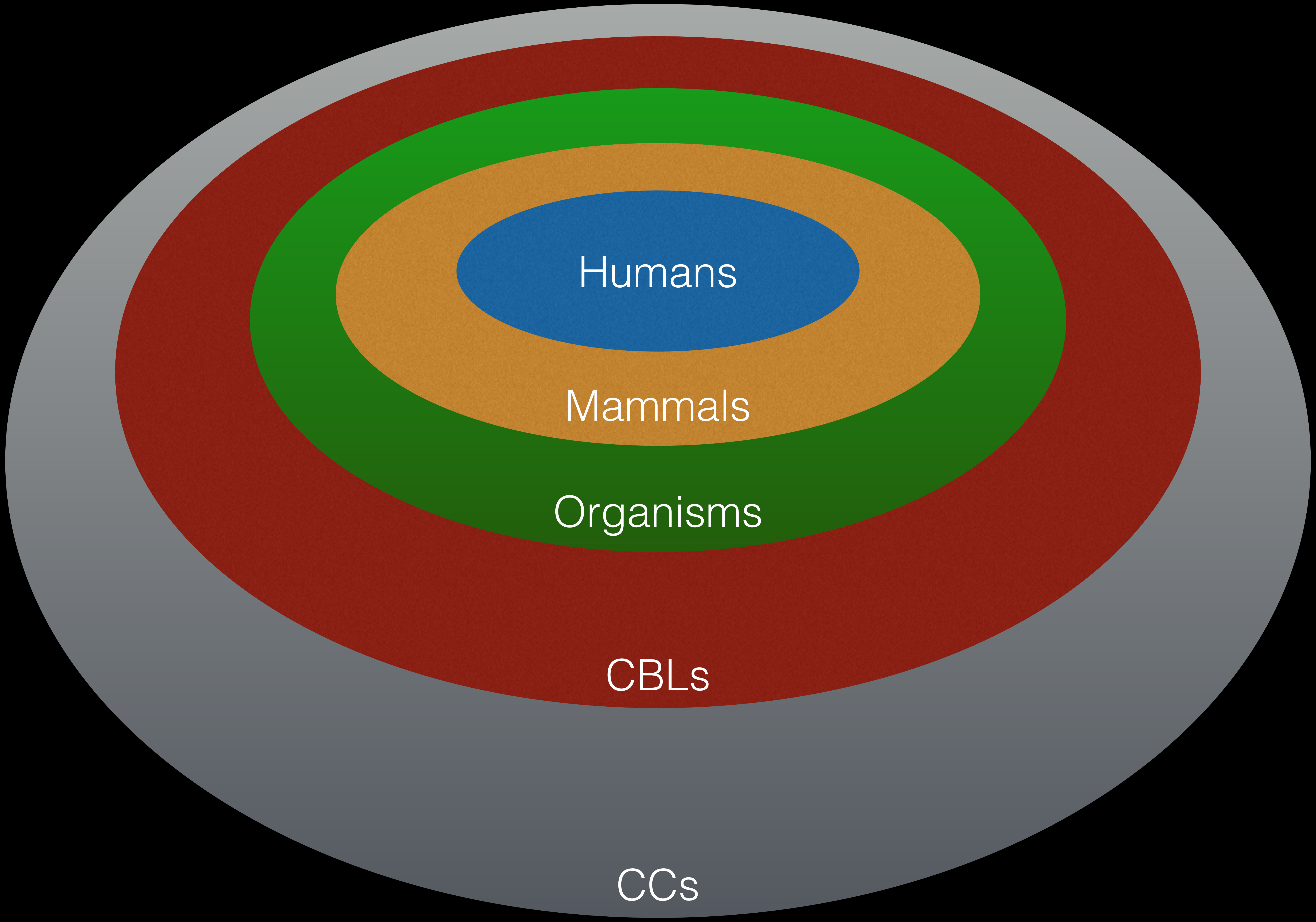
- From a **pragmatic** standpoint
  - Promote code reuse
- From a **theory** standpoint
  - Support type refinements
  - aka sub-typing through sub-classing

Theory View

- Consider concepts such as
  - Humans, Mammals, Organism, Carbon-based Lifeform (CBL), Chemical compounds (CC)
- They are related

UCONN

Venn Diagram



Hierarchy

- With each specializations
  - The requirements grow
  - The set of entities ***meeting*** those requirements shrinks.
- Specialization is akin to
  - Making subsets
  - Making subtypes

Requirements?

- These can be:
  - Having specific attributes
  - Having specific behaviors

C++ ?

- The language supports sub-classing
  - And sub-classes are sub-types in C++

Sub typing

- Let **A**,**B** be two types
  - We write **A <: B** to state that A is a subtype of B
- Example
  - A = Human
  - B = Mammal
  - A <: B means Human is a subtype of Mammal

UCONN

Subsumption

- Whenever A <: B
  - Anytime a B is expected, one can provide an A
- In other words
  - A is a subtype, aka subset of B, hence every A is also a B
- Example ?

Example I

```cpp
#include <iostream>
class Mammal {
public:
    Mammal() {}
    void print() { std::cout << "I (" << this << ")'m a mammal!" << std::endl;}
};
class Human :public  Mammal {
public:
    Human() {}
    void print() { std::cout << "I (" << this << ")'m a human!" << std::endl;}
};
void foo(Mammal& m) {
    std::cout << "in foo! message:";
    m.print();
}
```

```cpp
int main() {
    Mammal m;
    Human h;
    m.print();
    h.print();
    foo(m);foo(h);
    return 0;
}
```

Noteworthy

- **Both calls to foo are fine!**
  - Even though foo expected a Mammal
  - Since a Human is also a Mammal, the call is legal.
  - The object identity is preserved [see the pointer!]
- There is a **`:public`** annotation in front of the inheritance stanza
  - That’s related to the pragmatic view of inheritance

Pragmatic View

- Given two classes A and B where
  - A and B share lots of common code
  - A *refines / specializes* some of B
    - Adding state
    - Adding / changing behaviors
- Then A should inherit the common code from B

Mantra

Inheritance is *the way* to get code reuse

Book example

- CoreStudent
- GradStudent inherit from CoreStudent

Inheritance & Privacy

- When inheriting you can “alter” privacy of what you inherit
  - Public
  - Protected
  - Private

public

Public Inheritance

Concept

C++

- Semantics
  - What you inherit keeps its status
    - public ➔ public
    - protected ➔ protected
    - private ➔ private
  - What you override
    - Uses the privacy setting in the override

Public Inheritance

- This is the “common” case
  - But it is **NOT** the default when working with a class
  - It is the default when working with a struct
- **When class A publicly inherits from B**
  - **A**
  - **Its sub-classes and**
  - **Every function knows that A inherits from B**

protected

Protected Inheritance

Concept

C++

- Semantics
  - What you inherit changes status
    - public ➔ protected
    - protected ➔ private
  - What you override
    - Uses the privacy setting in the override

Protected Inheritance

- This is not the default
  - You must ask for it with: `class A :protected B { …`
- **When class A inherits in a protected way from B**
  - **A and its sub-classes know that they inherit from B**
  - **Nobody else knows that fact.**

private

Private Inheritance

Concept

C++

- Semantics
  - What you inherit changes status
    - public → private
    - protected → private
  - What you override
    - Uses the privacy setting in the override

Private Inheritance

- This **is the default if you use the "class" keyword**
- **When class A inherits in a private way from B**
  - **A knows that it inherits from B**
  - **A's sub-classes are clueless**
  - **Everything else is clueless**
  - **That inheritance fact is completely hidden.**

Corollary

```
#include <iostream>
class Mammal {
public:
    Mammal() {}
    void print() { std::cout << "I (" << this << ")'m a mammal!" << std::endl;}
};
class Human :  Mammal {
public:
    Human() {}
    void print() { std::cout << "I (" << this << ")'m a human!" << std::endl;}
};

int main() {
    Mammal m;
    Human h;
    Mammal& mr = h;
    return 0;
}
```

That's illegal!

Corollary 2

```
#include <iostream>
class Mammal {
public:
    Mammal() {}
    void print() { std::cout << "I (" << this << ")'m a mammal!" << std::endl;}
};
class Human :public  Mammal {
public:
    Human() {}
    void print() { std::cout << "I (" << this << ")'m a human!" << std::endl;}
};

int main() {
    Mammal m;
    Human h;
    Mammal& mr = h;
    return 0;
}
```

That's all good!

Inheritance and Overriding

- When class A inherits from class B
  - It has all the attributes of B
  - It has all the methods of B
  - It can add attributes or methods
  - But it can also
    - ***Upgrade [refine]*** some of the methods it inherits
    - That’s called **overriding**

Example: Grading Policy

- Consider two classes
  - For Undergraduates
  - For Graduates
- Graduates are like undergraduates
  - Except for the grading policy

	UG	G
A	$80 < x \le 100$	$90 < x \le 100$
B	$70 < x \le 80$	$80 < x \le 90$
C	$60 < x \le 70$	$70 < x \le 80$
D	$50 < x \le 60$	
F	$x \le 50$	$x \le 70$

UCONN

Demo

UCONN

Caveat Emptor!

- We can override
- But when mixing
  - Subsumption
  - And method call
- You get something unexpected!

What is Missing?

- You **have** inheritance
- You **lack** polymorphism
- The solution is…

Dynamic Binding

UCONN

Polymorphism

- We already discovered **COMPILE-TIME** polymorphism
  - Method overloading
  - Template functions and classes (aka LET-polymorphism)
- We need to look at **RUNTIME** polymorphism
  - Dynamic Binding of overriden methods

Purpose

- Provide the ability for an object to respond to messages based on its *dynamic* type rather than its *compile-time* type.
- Meanwhile, … In Java-land
  - Polymorphic methods are the default!
- Whereas in C++-land
  - The programmer (you) gets to choose!

Syntactically

- Remember the `virtual` keyword in front of methods?
  - That is what it does. It switches from
    - **Static** Binding
    - to **Dynamic** Binding

Revisiting the UG/G Example

```
class UGrad {
protected:
   std::string _name;
   double _grade;
public:
   UGrad(const std::string& n,double g) : _name(n),_grade(g) {}
   virtual const char letterGrade() const;
   friend std::ostream& operator<<(std::ostream& os,const UGrad& ug);
};

class Grad :public UGrad {
public:
   Grad(const std::string& n,double g) : UGrad(n,g) {}
   const char letterGrade() const;
   friend std::ostream& operator<<(std::ostream& os,const Grad& g);
};
```

Revisiting the UG/G Example

```
class UGrad {
protected:
   std::string _name;
   double _grade;
public:
   UGrad(const std::string& n,double g) : _name(n),_grade(g) {}
   virtual const char letterGrade() const;
   friend std::ostream& operator<<(std::ostream& os,const UGrad& ug);
};

class Grad :public UGrad {
public:
   Grad(const std::string& n,double g) : UGrad(n,g) {}
   const char letterGrade() const;
   friend std::ostream& operator<<(std::ostream& os,const Grad& g);
};
```

Wait… It gets Better!

- Why do we need to keep the output operator ?
  - It’s [almost] exactly the same
  - That’s a perfect case for calling a polymorphic method

Revised Example

```
class UGrad {
protected:
   std::string _name;
   double _grade;
public:
   UGrad(const std::string& n,double g) : _name(n),_grade(g) {}
   virtual const char letterGrade() const;
   virtual const std::string kind() const { return "UG";}
   friend std::ostream& operator<<(std::ostream& os,const UGrad& ug) {
      return os << ug._name << "(" << ug.kind() << ") = "
                << ug._grade << '(' << ug.letterGrade() << ')';
   }
};

class Grad :public UGrad {
public:
   Grad(const std::string& n,double g) : UGrad(n,g) {}
   const char letterGrade() const;
   const std::string kind() const { return "G";}
};
```

Polymorphic Code

- Very convenient
- A few caveats to be aware of:
  - Space usage and memory layout
  - Overload and overrides
  - Deallocation
  - Multiple inheritance caveats
  - Diamond inheritance

Space & Memory

- An object is….
  - A VPTR, a.k.a. a pointer to a table containing pointers to dynamically bound methods
  - Collection of fields for attributes

Overload vs. Overrides

```
#include <iostream>

class B {
public:
   virtual void f(short) {std::cout << "B::f" << std::endl;}
};
class D : public B {
public:
   virtual void f(int) {std::cout << "D::f" << std::endl;}
};

int main() {
   B* aPtr = new D;
   aPtr->f(1);
   return 0;
}
```

Overload vs. Overrides

```
#include <iostream>

class B  {
public:
   virtual void f(int) const {std::cout << "B::f " << std::endl;}
};
class D : public B {
public:
   virtual void f(int) {std::cout << "D::f" << std::endl;}
};

int main() {
   B* aPtr = new D;
   aPtr->f(1);
   return 0;
}
```

Overload vs. Overrides

```cpp
#include <iostream>

class B {
public:
   virtual void f(short) {std::cout << "B::f" << std::endl;}
};
class D : public B {
public:
   virtual void f(int) override {std::cout << "D::f" << std::endl;}
};

int main() {
   B* aPtr = new D;
   aPtr->f(1);
   return 0;
}
```

Finality

- There is also a way to state
  - That a method can no longer be overridden in sub-classes!
  - Add the final qualifier!

Finality

```cpp
#include <iostream>
class B {
public:
   virtual void f(int) {std::cout << "B::f" << std::endl;}
};
class D : public B {
public:
   virtual void f(int) override final {std::cout << "D::f" << std::endl;}
};
class F : public D {
public:
   virtual void f(int) override {std::cout << "F::f" << std::endl;}
};
int main() {
   B* aPtr = new F;
   aPtr->f(1);
   return 0;
}
```

Deallocation

- You might not now the true type of an object when you delete it….

```
int main()
{
    UGrad s1("Bernard",78);
    UGrad*  s2 = new Grad("Billy",67);
    std::cout << s1 << std::endl;
    std::cout << *s2 << std::endl;
    delete s2;
    return 0;
}
```

Deallocation

- Solution…. Make the destructor polymorphic!

```
class UGrad {
protected:
   std::string _name;
   double _grade;
public:
   UGrad(const std::string& n,double g) : _name(n),_grade(g) {}
   virtual ~UGrad();
   virtual const char letterGrade() const;
   virtual const std::string kind() const { return "UG";}
   friend std::ostream& operator<<(std::ostream& os,const UGrad& ug);
};
```

The delete operator

- It triggers a call to the destructor
- Since the destructor is polymorphic
  - It will do the right thing, even when invoked from a pointer to a super-class type.

Multiple Inheritance

- Since inheritance captures code re-use….
- You can inherit from 2 classes or more.
- The question becomes
  - What is the memory layout and what happens to polymorphism?

Default & Delete

- It is possible to control the generation of
  - Default constructors
  - Default destructors
  - Default assignment operators
- Purposes
  - Avoid to write boilerplate
  - Avoid to include some unwanted defaults

Example

```
#include <iostream>
struct NoCopy {
   int _a;
   int _b;
   NoCopy() { _a = _b = 0;}
   NoCopy(const NoCopy& nc) = delete;
   NoCopy& operator=(const NoCopy& nc) = delete;
};

int main() {
   NoCopy nc1;
   NoCopy nc2(nc1);
   NoCopy nc3;
   nc3 = nc1;
   return 0;
}
```

Delegation

- Sometimes overlooked
- Always extremely useful!

Key Idea

- Use a chain of objects
  - Pass along requests along the delegation chain.

Advantages

- You can
  - Have a fixed “front-end” object and vary the backend (at runtime)
  - Retain a fixed “address” despite behavior changes (morphing)
  - Upgrade behaviors over time by changing the delegate
  - Check the GoF book! (Delegation pattern,Facade pattern)

Usage

- Useful for
  - Aspect-oriented programming
  - Dynamic evolution
  - Tombstone
  - Smart pointers (They are using delegation!)

Requirements

- You need
  - Dynamic binding
- The front