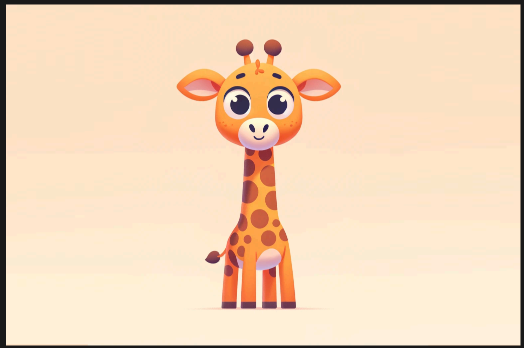
COMP6771

🍰 Dynamic Polymorphism

Lecture 7.1

Author(s): Hayden Smith



(Download as PDF)

🎂 Key Concepts

- **Iheritance**: ability to create new classes based on existing ones
  - Supported by class derivation
- **Polymorphism**: allows objects of a subclass to be used as if they were objects of a base class
  - Supported via virtual functions
- **Dynamic binding**: run-time resolution of the appropriate function to invoke based on the type of the object
  - Closely related to polymorphism
  - Supported via virtual functions

🌋OO Tenets Of C++

- Don't pay for what you don't use
  - C++ Supports OOP
    - No runtime performance penalty
  - C++ supports generic programming with the STL and templates
    - No runtime performance penalty
  - Polymorphism is extremely powerful, and we need it in C++
    - Do we need polymorphism at all when using inheritance?
      - Answer: sometimes
      - But how do we do so, considering that we don't want to make anyone who doesn't use it pay a performance penalty
- One of the guiding principles of C++ is "You don't pay for what you don't use"

Inheritance In C++

```cpp
class BaseClass {
 public:
  int get_int_member() { return int_member_; }
  std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int int_member_;
  std::string string_member_;
}
```

```cpp
class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    return "SubClass";
  }

 private:
  std::vector<int> vector_member_;
  std::unique_ptr<int> ptr_member_;
}
```

👁️‍🗨️ Inheritance In C++

- To inherit off classes in C++, we use "class DerivedClass: public BaseClass"
- Visibility can be one of:
  - **public** (generally use this unless you have good reason not to)
    - If you don't want public, you should (usually) use composition
  - **protected**
  - **private**
- Visibility is the maximum visibility allowed
  - If you specify ": private BaseClass", then the maximum visibility is private
    - Any BaseClass members that were public or protected are now private

👁‍🗨 Inheritance In C++

Memory Layout

BaseClass object		**SubClass object**
int_member_	BaseClass subobject	int_member_
string_member_		string_member_
	SubClass subobject	vector_member_
		ptr_member_

🔢 Code Exploration

```cpp
#include <string>
#include <iostream>

class BaseClass {
 public:
  int get_member() { return member_; }
  std::string get_class_name() {
    return "BaseClass";
  };

 private:
  int member_;
};

class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    (void) subclass_data_;
    return "SubClass";
  }

 private:
  int subclass_data_;
};

void print_class_name(BaseClass base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '\n';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

code-explore.cpp

🔢 Code Exploration

How many bytes is a BaseClass instance?

How many bytes is a SubClass instance?

How can the compiler allocate space for it on the stack, when it doesn't know how big it could be?

🔢 Code Exploration

The answer: since we care about performance, a BaseClass can only store a BaseClass, not a SubClass

If we try to fill that value with a SubClass, then it just fills it with the BaseClass subobject, and drops the SubClass subobject

This is called the **object slicing problem**

🔢 Code Exploration

The **solution** to this is to use references/pointers (preferably references) to the baseclass

🔢 Code Exploration

```cpp
#include <string>
#include <iostream>

class BaseClass {
 public:
  int get_member() { return member_; }
  std::string get_class_name() {
    return "BaseClass";
  };

 private:
  int member_;
};

class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    (void) subclass_data_;
    return "SubClass";
  }

 private:
  int subclass_data_;
};

void print_class_name(BaseClass& base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '\n';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

slicing-reference.cpp

😶‍🌫️ More Problems

```
class BaseClass {
 public:
  int get_member() { return member_; }
  std::string get_class_name() {
    return "BaseClass";
  };

 private:
  int member_;
};
class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

```
void print_class_name(BaseClass& base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '
';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

- How does the compiler decide which version of get_class_name to call?
  - When does the compiler decide this? Compile or runtime?
- How can it ensure that calling get_member doesn't have similar overhead?

💊 Virtual & Override

```
void print_stuff(const BaseClass& base) {
  std::cout << base.get_class_name() << '
';
}
int main() {
  SubClass subclass;
  print_stuff(subclass);
}
```

By default, C++ will call "get_class_name()" of the BaseClass.

However, if the base class has the function marked as **virtual**, it will happily look toward the derived class.

Virtual & Override

For example:

```
class BaseClass {
 public:
  int get_member() { return member_; }
  virtual std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int member_;
}
```

```
class SubClass: public BaseClass {
 public:
  std::string GetClassName() override {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

Note the use of the **override** keyword to.

💊 Virtual & Override

- While override isn't required by the compiler, you should always use it
- Override fails to compile if the function doesn't exist in the base class. This helps with:
  - Typos
  - Refactoring
  - Const / non-const methods
  - Slightly different signatures

💊 Virtual & Override

Let's explore some outputs wiht virtual members

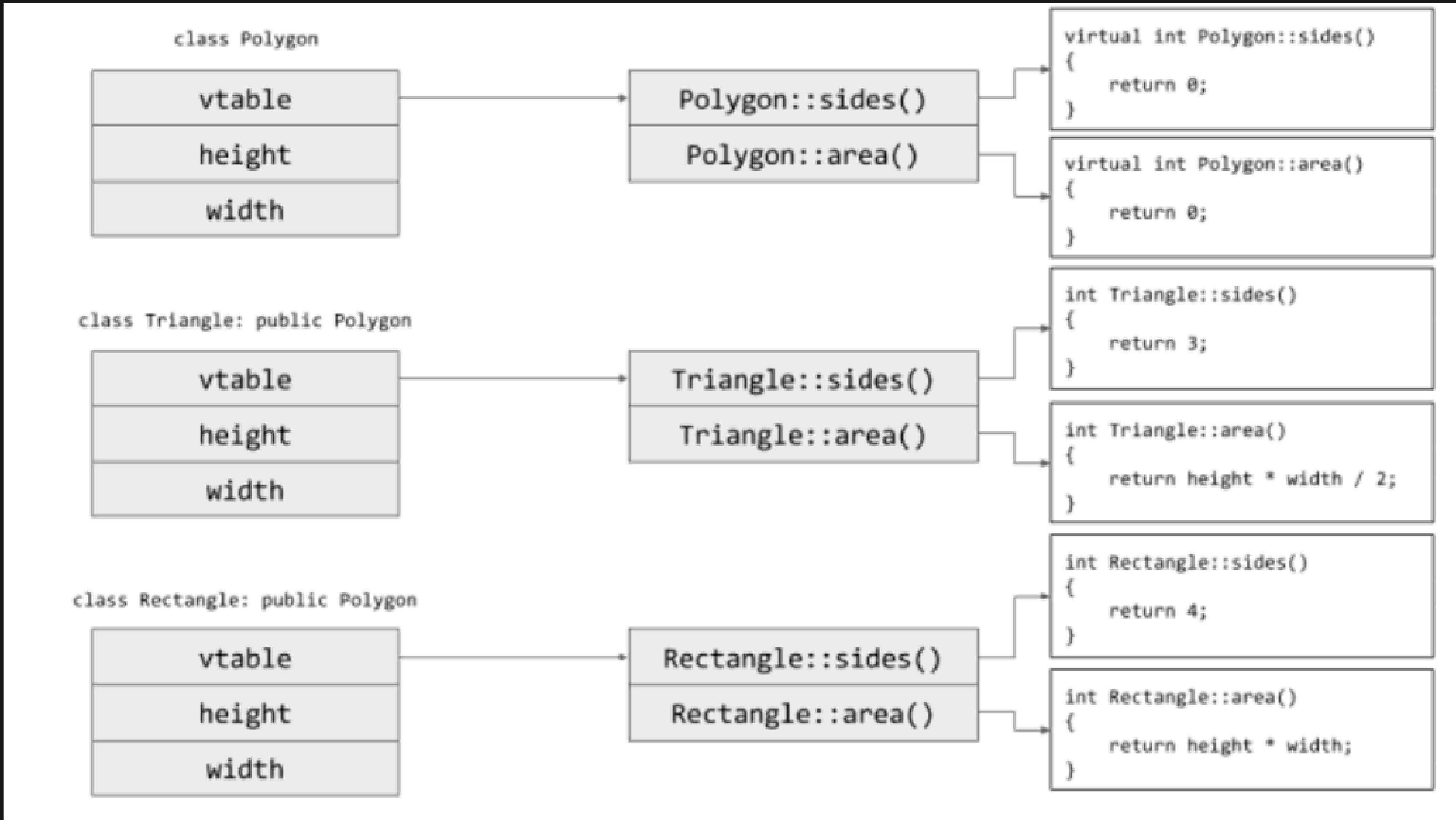
```cpp
#include <iostream>
#include <string>

class BaseClass {
 public:
  virtual std::string get_class_name() const {
    return "BaseClass";
  };

  virtual ~BaseClass() {
   std::cout << "Destructing base class\n";
 }
};

class SubClass: public BaseClass {
 public:
  std::string get_class_name() const override {
    return "SubClass";
  }

 ~SubClass() {
   std::cout << "Destructing subclass\n";
 }
};

void print_stuff(const BaseClass& base_class) {
  std::cout << base_class.get_class_name() << '\n';
}

int main() {
  auto subclass = static_cast<std::unique_ptr<BaseClass>>(
    std::make_unique<SubClass>());
  std::cout << subclass->get_class_name();
}
```

virtual.cpp

💻 Vtables

- Each class has a VTable stored in the data segment
  - A vtable is an array of function pointers that says which definition each virtual function points to for that class
- If the VTable for a class is non-empty, then every member of that class has an additional data member that is a pointer to the vtable
- When a virtual function is called on a reference or pointer type, then the program actually does the following
  1. Follow the vtable pointer to get to the vtable
  2. Increment by an offset, which is a constant for each function
  3. Follow the function pointer at vtable[offset] and call the function

💻 Vtables


class Polygon
vtable
height
width
Polygon::sides()
Polygon::area()
virtual int Polygon::sides()
{
return 0;
}
virtual int Polygon::area()
{
return 0;
}
class Triangle: public Polygon
vtable
height
width
Triangle::sides()
Triangle::area()
int Triangle::sides()
{
return 3;
}
int Triangle::area()
{
return height * width / 2;
}
class Rectangle: public Polygon
vtable
height
width
Rectangle::sides()
Rectangle::area()
int Rectangle::sides()
{
return 4;
}
int Rectangle::area()
{
return height * width;
}

✔️ Final

- Specifies to the compiler "this is not virtual for any subclasses"
- If the compiler has a variable of type SubClass&, it now no longer needs to look it up in the vtable
- This means static binding if you have a SubClass&, but dynamic binding for BaseClass&

```cpp
class BaseClass {
 public:
  int get_member() { return member_; }
  virtual std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int member_;
}
```

```cpp
class SubClass: public BaseClass {
 public:
  std::string get_class_name() override final {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

✔️ Final

Syntax	Name	Meaning
`virtual void fn() = 0;`	pure virtual	Inherit interface only
`virtual void fn()`	virtual	Inherit interface with optional implementation
`void fn()`	nonvirtual	Inherit interface and mandatory implementation

- **Note: nonvirtuals can be hidden by writing a function with the same name in a subclass**
  - DO NOT DO THIS

💌 Abstract Base Classes (ABCs)

- Might want to deal with a base class, but the base class by itself is nonsense
  - What is the default way to draw a shape? How many sides by default?
  - A function takes in a "Clickable"
- Might want some default behaviour and data, but need others
  - All files have a name, but are reads done over the network or from a disk
- If a class has at least one "abstract" (pure virtual in C++) method, the class is abstract and cannot be constructed
  - It can, however, have constructors and destructors
  - These provide semantics for constructing and destructing the ABC subobject of any derived classes

🪽 Pure Virtual Functions

- Virtual functions are good for when you have a default implementation that subclasses may want to overwrite
- Sometimes there is no default available
- A pure virtual function specifies a function that a class must override in order to not be abstract

```
class Shape {
  // Your derived class "Circle" may forget to write this.
  virtual void draw(Canvas&) {}

  // Fails at link time because there's no definition.
  virtual void draw(Canvas&);

  // Pure virtual function.
  virtual void draw(Canvas&) = 0;
};
```

🔨 Creating Polymorphic Objects

- In a language like Java, everything is a pointer
  - This allows for code like on the left
  - Not possible in C++ due to objects being stored inline
    - This then leads to slicing problem
- If you want to store a polymorphic object, use a pointer

```cpp
// Java-style C++ here
// Don't do this.

auto base = std::vector<BaseClass>();
base.push_back(BaseClass{});
base.push_back(SubClass1{});
base.push_back(SubClass2{});
```

```cpp
// Good C++ code
// But there's a potential problem here.
// (*very* hard to spot)

auto base = std::vector<std::unique_ptr<BaseClass>>();
base.push_back(std::make_unique<BaseClass>());
base.push_back(std::make_unique<Subclass1>());
base.push_back(std::make_unique<Subclass2>());
```

🗽 Inheritance And Constructors

- Every subclass constructor must call a base class constructor
  - If none is manually called, the default constructor is used
  - A subclass cannot initialise fields defined in the base class
  - Abstract classes must have constructors

```
class BaseClass {
 public:
  BaseClass(int member): int_member_{member} {}

 private:
  int int_member_;
  std::string string_member_;
}

class SubClass: public BaseClass {
 public:
  SubClass(int member, std::unique_ptr<int>&& ptr): BaseClass(member), ptr_member_(std::move(ptr)) {}
  // Won't compile.
  SubClass(int member, std::unique_ptr<int>&& ptr): int_member_(member), ptr_member_(std::move(ptr)) {}

 private:
  std::vector<int> vector_member_;
  std::unique_ptr<int> ptr_member_;
}
```

🥟 Destructing Polymorphic Objects

- Which constructor is called?
- Which destructor is called?
- What could the problem be?
    - What would the consequences be?
- How might we fix it, using the techniques we've already learnt?

```
// Simplification of previous slides code.

auto base = std::make_unique<BaseClass>();
auto subclass = std::make_unique<Subclass>();
```

Destructing Polymorphic Objects

- Whenever you write a class intended to be inherited from, **always make your destructor virtual**
- Remember: When you declare a destructor, the move constructor and assignment are not synthesized

```
class BaseClass {
  BaseClass(BaseClass&&) = default;
  BaseClass& operator=(BaseClass&&) = default;
  virtual ~BaseClass() = default;
}
```

Forgetting this can be a hard bug to spot

💡 Static & Dynamic Types

- Static type is the type it is declared as
- Dynamic type is the type of the object itself
- Static means compile-time, and dynamic means runtime
  - Due to object slicing, an object that is neither reference or pointer always has the same static and dynamic type

```cpp
int main() {
  auto base_class = BaseClass();
  auto subclass = SubClass();
  auto sub_copy = subclass;
  // The following could all be replaced with pointers
  // and have the same effect.
  const BaseClass& base_to_base{base_class};
  // Another reason to use auto - you can't accidentally do this.
  const BaseClass& base_to_sub{subclass};
  // Fails to compile
  const SubClass& sub_to_base{base_class};
  const SubClass& sub_to_sub{subclass};
  // Fails to compile (even though it refers to at a sub);
  const SubClass& sub_to_base_to_sub{base_to_sub};
}
```

💡 Static & Dynamic Types

- **Static binding:** Decide which function to call at compile time (based on static type)
- **Dynamic binding:** Decide which function to call at runtime (based on dynamic type)
- C++
  - Statically typed (types are calculated at compile time)
  - Static binding for non-virtual functions
  - Dynamic binding for virtual functions
- Java
  - Statically typed
  - Dynamic binding

💡 Static & Dynamic Types

Up-casting

- Casting from a derived class to a base class is called up-casting
- This cast is always safe
  - All dogs are animals
- Because the cast is always safe, C++ allows this as an implicit cast
- One of the reasons to use auto is that it avoids implicit casts

```cpp
auto dog = Dog();
Animal& animal = dog;
Animal* animal = &dog;
```

💡 Static & Dynamic Types

Down-casting

- Casting from a base class to a derived class is called down-casting
- This cast is not safe
  - Not all animals are dogs

```cpp
auto dog = Dog();
auto cat = Cat();
Animal& animal_dog{dog};
Animal& animal_cat{cat};

// Attempt to down-cast with references.
// Neither of these compile.
// Why not?
Dog& dog_ref{animal_dog};
Dog& dog_ref{animal_cat};
```

💡 Static & Dynamic Types

How to down cast

- The compiler doesn't know if an Animal happens to be a Dog
  - If you know it is, you can use static_cast
  - Otherwise, you can use dynamic_cast
    - Returns null pointer for pointer types if it doesn't match
    - Throws exceptions for reference types if it doesn't match

```
auto dog = Dog();
auto cat = Cat();
Animal& animal_dog{dog};
Animal& animal_cat{cat};

// Attempt to down-cast with references.
Dog& dog_ref{static_cast<Dog&>(animal_dog)};
Dog& dog_ref{dynamic_cast<Dog&>(animal_dog)};
// Undefined behaviour (incorrect static cast).
Dog& dog_ref{static_cast<Dog&>(animal_cat)};
// Throws exception
Dog& dog_ref{dynamic_cast<Dog&>(animal_cat)};
```

```
auto dog = Dog();
auto cat = Cat();
Animal& animal_dog{dog};
Animal& animal_cat{cat};

// Attempt to down-cast with pointers.
Dog* dog_ref{static_cast<Dog*>(&animal_dog)};
Dog* dog_ref{dynamic_cast<Dog*>(&animal_dog)};
// Undefined behaviour (incorrect static cast).
Dog* dog_ref{static_cast<Dog*>(&animal_cat)};
// returns null pointer
Dog* dog_ref{dynamic_cast<Dog*>(&animal_cat)};
```

👰 Covariants

- Read more about covariance and contravariance
- **If a function overrides a base, which type can it return?**
  - If a base specifies that it returns a LandAnimal, a derived also needs to return a LandAnimal
- Every possible return type for the derived must be a valid return type for the base

```
class Base {
  virtual LandAnimal& get_favorite_animal();
};

class Derived: public Base {
  // Fails to compile: Not all animals are land animals.
  Animal& get_favorite_animal() override;
  // Compiles: All land animals are land animals.
  LandAnimal& get_favorite_animal() override;
  // Compiles: All dogs are land animals.
  Dog& get_favorite_animal() override;
};
```

👰 Contravariants

- **If a function overrides a base, which types can it take in?**
  - If a base specifies that it takes in a LandAnimal, a LandAnimal must always be valid input in the derived
- Every possible parameter to the base must be a possible parameter for the derived

```
class Base {
  virtual void use_animal(LandAnimal&);
};

class Derived: public Base {
  // Compiles: All land animals are valid input (animals).
  void use_animal(Animal&) override;
  // Compiles: All land animals are valid input (land animals).
  void use_animal(LandAnimal&) override;
  // Fails to compile: Not All land animals are valid input (dogs).
  void use_animal(Dog&) override;
};
```

🐕‍🦺 Default Arguments And Virtuals

- Default arguments are determined at compile time for efficiency's sake
- Hence, default arguments need to use the static type of the function
- **Avoid default arguments when overriding virtual functions**

```cpp
#include <iostream>

class Base {
public:
  virtual ~Base() = default;
  virtual void print_num(int i = 1) {
    std::cout << "Base " << i << '\n';
  }
};

class Derived: public Base {
public:
  void print_num(int i = 2) override {
    std::cout << "Derived " << i << '\n';
  }
};

int main() {
  Derived derived;
  Base* base = &derived;
  derived.print_num(); // Prints "Derived 2"
  base->print_num(); // Prints "Derived 1"
}
```

default.cpp

Construction Of Derived Classes

- Base classes are always constructed before the derived class is constructed
  - The base class ctor never depends on the members of the derived class
  - The derived class ctor may be dependent on the members of the base class

```
class Animal {...}
class LandAnimal: public Animal {...}
class Dog: public LandAnimals {...}

Dog d;

// Dog() calls LandAnimal()
  // LandAnimal() calls Animal()
    // Animal members constructed using initialiser list
    // Animal constructor body runs
  // LandAnimal members constructed using initialiser list
  // LandAnimal constructor body runs
// Dog members constructed using initialiser list
// Dog constructor body runs
```

Virtuals In Constructors

If a class is not fully constructed, cannot perform dynamic binding

```
class Animal {...};
class LandAnimal: public Animal {
  LandAnimal() {
    Run();
  }

  virtual void Run() {
    std::cout << "Land animal running
";
  }
};
class Dog: public LandAnimals {
  void Run() override {
    std::cout << "Dog running
";
  }
};

// When the LandAnimal constructor is being called,
// the Dog part of the object has not been constructed yet.
// C++ chooses to not allow dynamic binding in constructors
// because Dog::Run() might depend upon Dog's members.
Dog d;
```

🐈 Destruction Of Derived Classes

Easy to remember order: Always opposite to construction order

```
class Animal {...}
class LandAnimal: public Animal {...}
class Dog: public LandAnimals {...}

auto d =  Dog();

// ~Dog() destructor body runs
  // Dog members destructed in reverse order of declaration
    // ~LandAnimal() destructor body runs
    // LandAnimal members destructed in reverse order of declaration
  // ~Animal() destructor body runs
// Animal members destructed in reverse order of declaration.
```

Virtuals In Destructors

- If a class is partially destructed, cannot perform dynamic binding
- Unrelated to the destructor itself being virtual

```
class Animal {...};
class LandAnimal: public Animal {
  virtual ~LandAnimal() {
    Run();
  }

  virtual void Run() {
    std::cout << "Land animal running
";
  }
};
class Dog: public LandAnimals {
  void Run() override {
    std::cout << "Dog running
";
  }
};

// When the LandAnimal constructor is being called,
// the Dog part of the object has already been destroyed.
// C++ chooses to not allow dynamic binding in destructors
// because Dog::Run() might depend upon Dog's members.
auto d = Dog();
```

👂 Feedback

Or go to the form here.