Object-Oriented Programming

Lecture 11: Polymorphism

What is Polymorphism?

Polymorphism, from its Greek roots "poly" (many) and "morph" (form), means the ability to take multiple forms.
In C++, it means that the same function call or operator can behave differently depending on the object it's operating on.

lect11_00.cpp

Example of Polymorphism

lect11_00.cpp

Animal* animal1 = &dog;

- animal1 is a pointer of type Animal* (a pointer to an Animal object).
- The pointer animal1 is assigned the address of the dog object (&dog), which is an instance of the Dog class.
- Since Dog is derived from Animal, a Dog object is-a Animal (due to inheritance). This means you can use a base-class pointer (Animal*) to point to a derived-class object (Dog).

Types of Polymorphism in C++

Compile-time Polymorphism

- Function Overloading:
  Multiple functions with the same name but different parameter lists. The compiler selects the appropriate version at compile time.
  (lec11_01_function_overload.cpp)
- Operator overloading allows you to redefine the way operators work with user-defined types. (lec11_02_operator_overload.cpp)

Runtime Polymorphism (achieved through virtual functions and inheritance)

- Virtual Functions: Functions in a base class marked with the virtual keyword and overridden in derived classes.
- When a virtual function is called using a base class pointer or reference , the program determines which version of the function to call at runtime based on the actual object's type.

Virtual Table

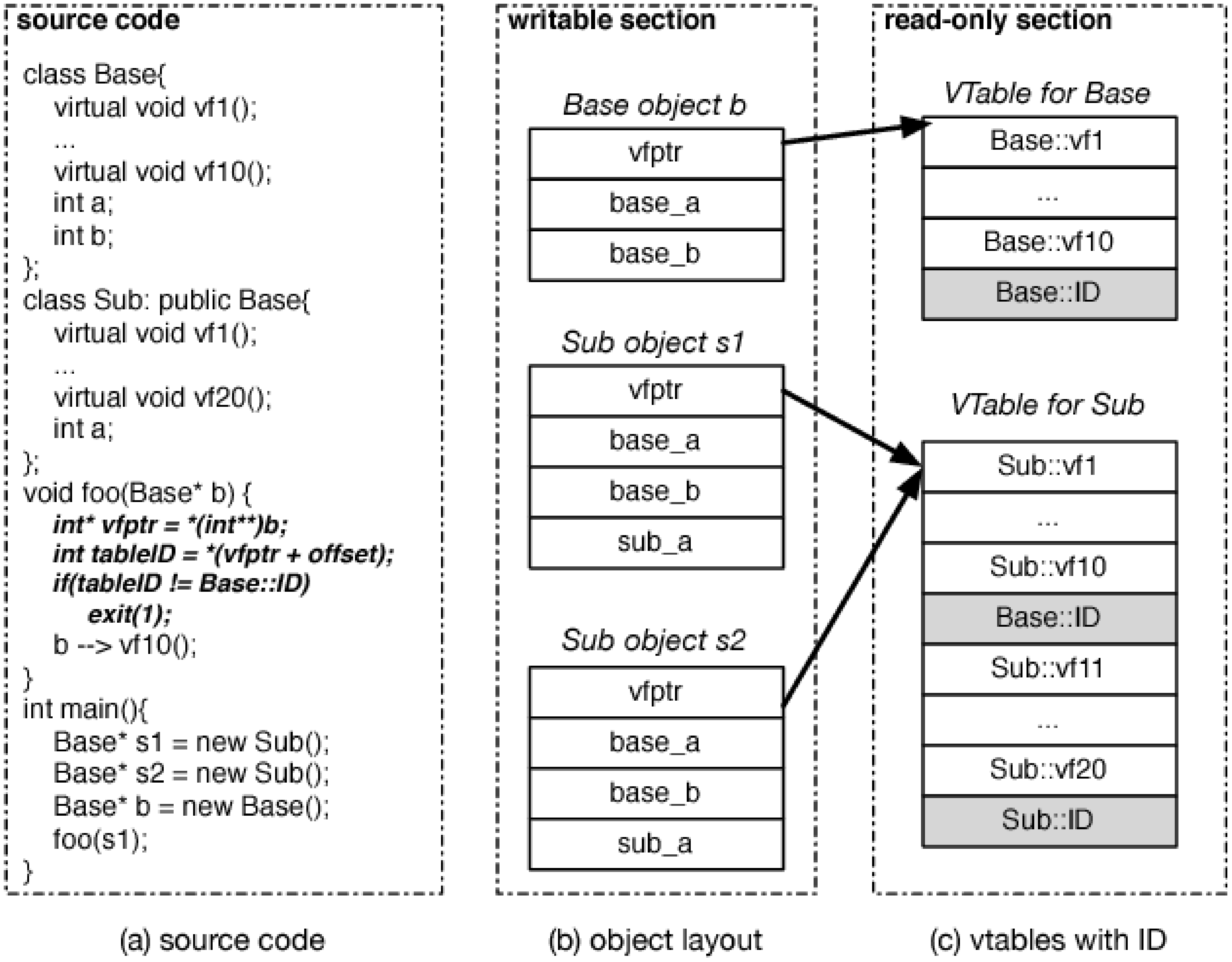
- vtable is a mechanism used in C++ to support dynamic (runtime) polymorphism through virtual functions.
- Each class that has virtual functions (or inherits from a class that has virtual functions) has its own vtable. This table is an array of pointers to the virtual functions defined in the class.
- When a virtual function is called on an object, the compiler uses the vtable of the object's class to look up the address of the correct function to call, based on the actual type of the object.

Virtual Table

Creation : The compiler generates a vtable for each class with virtual functions or inheriting them. The vtable lists pointers to all virtual functions callable on the class’s objects, including inherited ones.

Object Construction : When an object is created, a vptr is added to its memory layout, pointing to the vtable of its class.

Function Call : During a virtual function call, the compiler uses the object's vptr to locate the correct vtable and resolve the function at runtime based on the object's actual type, not the pointer/reference type.



```
class Base{
    virtual void vf1();
    ...
    virtual void vf10();
    int a;
    int b;
};
class Sub: public Base{
    virtual void vf1();
    ...
    virtual void vf20();
    int a;
};
void foo(Base* b) {
    int* vfptr = *(int**)b;
    int tableID = *(vfptr + offset);
    if(tableID != Base::ID)
        exit(1);
    b --> vf10();
}
int main(){
    Base* s1 = new Sub();
    Base* s2 = new Sub();
    Base* b = new Base();
    foo(s1);
}
```

https://memext.wordpress.com/2024/02/21/virtual-tables-visualization-in-c/

Virtual Table

```
class Base
{
public:
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   void function1() override {};
};

class D2: public Base
{
public:
   void function2() override {};
};


int main()
{
   D1 d1 {};
   Base* dPtr = &d1;
    dPtr->function1();

   return 0;
}
```

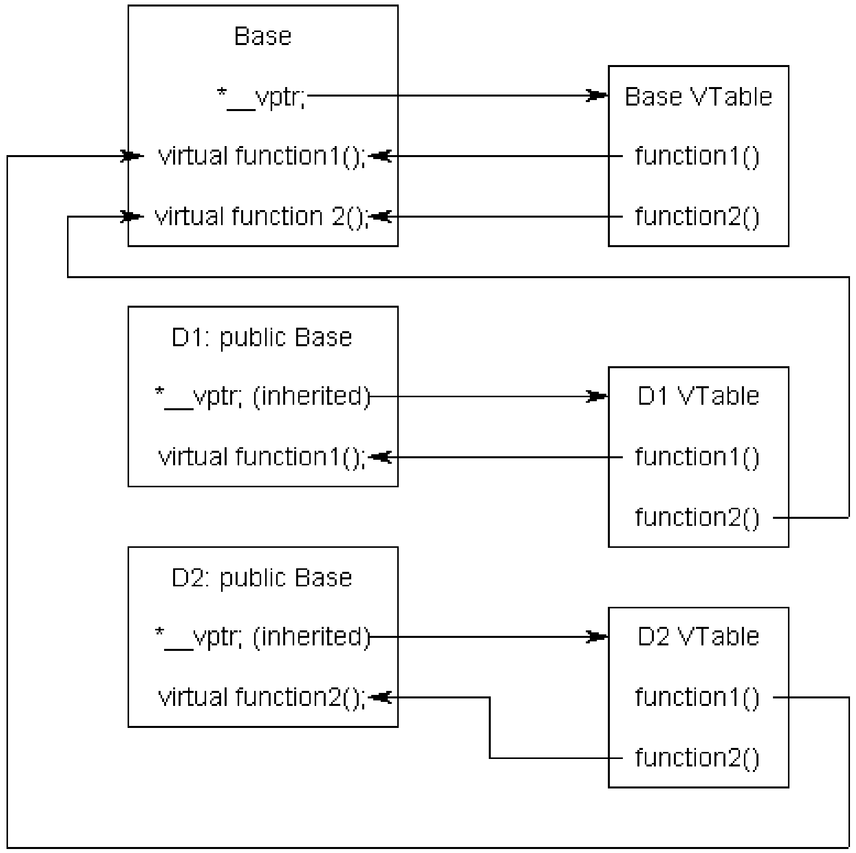
```
class Base
{
public:
   VirtualTable* __vptr;
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   void function1() override {};
};

class D2: public Base
{
public:
   void function2() override {};
};

int main()
{
   Base* dPtr = new d1();
   dPtr>function1();

   return 0;
}
```



Example of Run-time Polymorphism

(lec11_03_rt_polymorphism.cpp)

Conceptual Modeling: Using a base class like Shape with derived classes like Circle and Square reflects the real-world relationship and hierarchy between these entities.

Specialization: Derived classes represent a specialization of the base class.

For example, both Circle and Square are specific types of Shape.

Adding New Functionality : Derived classes can extend the base class by adding new functionality. This could include additional methods and properties that are specific to the derived class

(lec11_04_rt_polymorphism_2.cpp)

Aiming Derived-Class Pointers at Base-Class Objects

- Base class pointer - A base class pointer can point to any object of any class that inherits from it.
- This allows to use objects of different classes (that derive from a common base class) interchangeably, leveraging the concept of polymorphism.

```
void printArea(Shape* shape) {
   cout << "Area: " << shape->getArea() << endl;
}

// Can use with any shape
Circle circle(5);
Square square(4);
printArea(&circle);  // Works!
printArea(&square);  // Works too!
```

- Important considerations and limitations when pointing derived-class pointers at base-class objects, which is generally not recommended or safe.

Aiming Derived-Class Pointers at Base-Class Objects

Base-Class Pointer to Derived-Class Object: This is a common and safe practice in C++. You can assign the address of a derived-class object to a pointer of the base class.
Allows the base class pointer to access the parts of the derived object that are common with the base class.

```
class Base {};
class Derived : public Base {};

Base* basePtr;
Derived derivedObj;
basePtr = &derivedObj; // Safe and common
```

Aiming Derived-Class Pointers at Base-Class Objects

Don’t do:

```
Circle* circlePtr;
Shape baseShape;
circlePtr = &baseShape;  // DANGER!
```

Better

```
// Good practice:
Shape* basePtr = new Circle(5);    // ✅ Safe
basePtr->getArea();               // Works fine

// Bad practice:
Circle* derivedPtr = (Circle*)&baseShape;  // ❌ Dangerous
```

Aiming Derived-Class Pointers at Base-Class Objects

- Derived-Class Pointer to Base-Class Object: This is generally unsafe and not recommended.
- The derived-class pointer expects the object it points to, to have all the features of the derived class (including member variables and functions).
- However, a base-class object lacks these additional features that the derived class might have added.

```
class Base {};
class Derived : public Base {};

Derived* derivedPtr;
Base baseObj;
derivedPtr = &baseObj; // Unsafe, can lead to undefined behavior
```

Derived-Class Member-Function Calls via Base-Class Pointers

- Using a pointer of the base class type to call member functions of a derived class. This is possible and safe under the following conditions:
- Polymorphism: The member functions being called are virtual in the base class and are overridden in the derived class. This allows the base class pointer to dynamically bind to the derived class's implementation of the function at runtime (dynamic polymorphism).
- Correct Object Type: The base class pointer must actually point to an object of the derived class or a class further derived from it.

Derived-Class Member-Function Calls via Base-Class Pointers

```
auto derivedPtr = dynamic_cast<Derived*>(basePtr);
  if (derivedPtr) {
    derivedPtr->specificFunction(); // Safe, as we've ensured the pointer actually points to a Derived object
  }
```

- dynamic_cast is used for safe type conversion, especially in a hierarchy of classes involving polymorphism. It is used to convert a pointer (or reference) of a base class to a pointer (or reference) of a derived class.
- This type of casting is checked at runtime to ensure the safety of the operation, making dynamic_cast safer than other casts like static_cast or reinterpret_cast.

Derived-Class Member-Function Calls via Base-Class Pointers

- "Derived-Class Member-Function Calls via Base-Class Pointers”
- Referring to the overall process - start with a base-class pointer (basePtr) and use it in a context to call a derived-class-specific member function.
- This process involves temporarily converting (downcasting) the base-class pointer to a derived-class pointer to access those derived-specific features safely.
- The actual call to the derived-class member function (specificFunction()) is indeed made using the derived-class pointer (derivedPtr
- The important part is that this derived-class pointer was obtained by safely downcasting a base-class pointer that was originally pointing to a derived-class object.

Virtual Functions and Virtual Destructors

- A virtual function is a member function in a base class that expect to redefine in derived classes.
- When refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Polymorphism: This is the primary mechanism in C++ to achieve runtime polymorphism.
- Dynamic Binding: When a virtual function is called through a pointer or a reference, the actual function that gets called is resolved at runtime based on the type of the object pointed to, not the type of the pointer or reference.

Virtual Functions and Virtual Destructors

Virtual destructor ensures that when delete a derived class object through a base class pointer, the derived class's destructor is called before the base class's destructor. This is crucial for proper resource cleanup in classes that inherit from base classes with dynamic memory allocation.
Resource Management: Ensures that destructors of derived classes are called, allowing for proper cleanup of resources allocated by derived classes.
Avoid Memory Leaks: Without a virtual destructor in the base class, only the base class's destructor would be called, potentially leading to resource leaks.

Polymorphism, Virtual Functions and Dynamic Binding

Polymorphism

- Allows objects of different classes to be treated as objects of a common superclass. It's enabling the same interface to be used for different underlying forms (data types).
- Compile-time Polymorphism (Static Binding): Achieved using function overloading and operator overloading. The function to be called is determined at compile time.
- Runtime Polymorphism (Dynamic Binding): Achieved using inheritance, virtual functions, and pointers/references. The function to be called is determined at runtime.

Polymorphism, Virtual Functions and Dynamic Binding

- Virtual function is a function in a base class that is declared with the virtual keyword. When a base class reference or pointer points to a derived class object and a virtual function is called, C++ determines at runtime which function to invoke.
- VTable (Virtual Table): Under the hood, C++ uses a virtual table to implement dynamic binding. A virtual table is an array of pointers to virtual functions, maintained per class.
- VPointer (Virtual Pointer): Each object of a class containing virtual functions has a hidden member, a pointer to the virtual table for that class. This pointer, often called a vptr, is set up automatically by the constructor of the class.

Polymorphism, Virtual Functions and Dynamic Binding

- Dynamic binding is the process by which a call to an overridden function is resolved at runtime. This is in contrast to static binding, which resolves the function call at compile time.
- At Object Creation: When an object of a class containing virtual functions is created, the compiler sets the vptr of the object to point to the virtual table of that class.
- At Function Call: When a virtual function call is made through a base class pointer or reference, the compiler generates code to first look at the vptr of the actual object (which points to the virtual table of the actual class of the object). It then uses the vptr to get to the virtual table of the actual object and invokes the function pointed to by the appropriate entry in the table.automatically by the constructor of the class.