[CS 225] Advanced C/C++

Lecture 3: Advanced Inheritance

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Rationale behind inheritance
- Multiple inheritance
- Virtual inheritance
- Overriding vs. overloading
- Under-the-hood

Rationale behind Inheritance

Generalization

- Avoiding repetition by putting reusable code in common base class.
- Sharing a base type among multiple classes.

Rationale behind Inheritance

Specialization

- Extending existing implementation or interface.
- Overriding existing implementation to specialize it.

Rationale behind Inheritance

Realization

- Realization of **interface classes** in implementation classes.
- A relationship of public inheritance from an interface class.
- As ***vptr*** of an interface can be reused in a derived class; single inheritance from an interface class has no extra memory cost.

Rationale behind Inheritance

- An **interface class** is a special class:
  - Everything is public; no protected or private members.
  - All its member functions are pure virtual.
  - A destructor is virtual and has a default implementation.
  - No static members.
  - No data members.

Be careful: an interface class is commonly called ***an** interface* and it is easy to confuse it with ***the** interface* part of source code that is usually provided in header files (as opposed to the implementation in translation unit files).

Multiple Inheritance

Rationale

- Gaining benefits of inheritance from each base class individually
  - Using more than one **generalized** base class.
  - **Specializing** more than one base class.
  - **Realizing** more than one interface class.

- Most object-oriented programming languages support only **realization** with multiple inheritance as they clearly distinguish interface classes from implementation classes; C++ does not.

Multiple Inheritance

Construction and destruction

- All base classes are constructed in order of specification (not in order of appearing in the initialization list) before the rest of the initialization list and the body of a constructor of a derived class.
- Destruction is invoked in the reverse order.

Multiple Inheritance

Memory layout

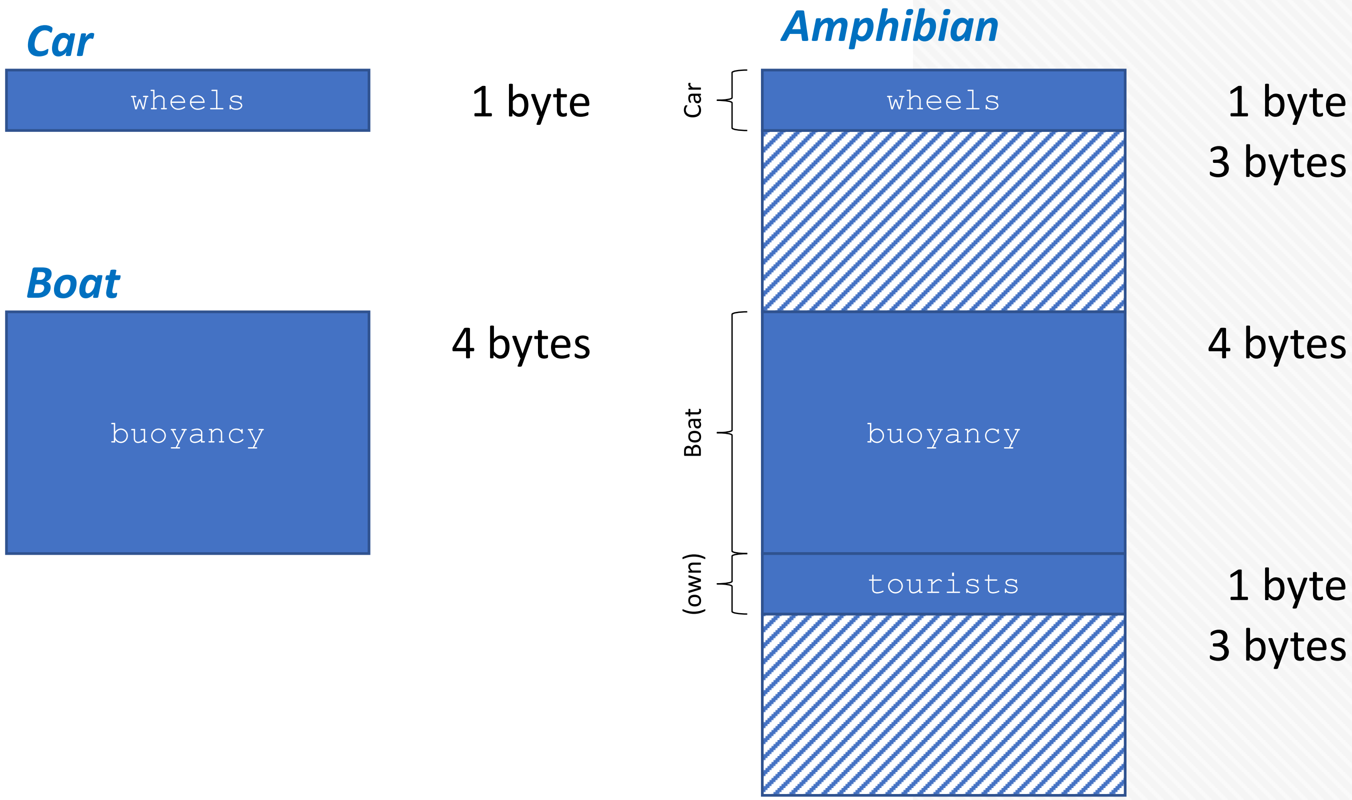
- Data members of the first base class are followed by data members of subsequent bases in order of specification.
- Own data member come last.
- Exception: virtual inheritance (covered later).

Multiple Inheritance

```
struct Car
{
  unsigned char wheels;
};

struct Boat
{
  float buoyancy;
};

struct Amphibian : Car, Boat
{
  unsigned char tourists;
};
```

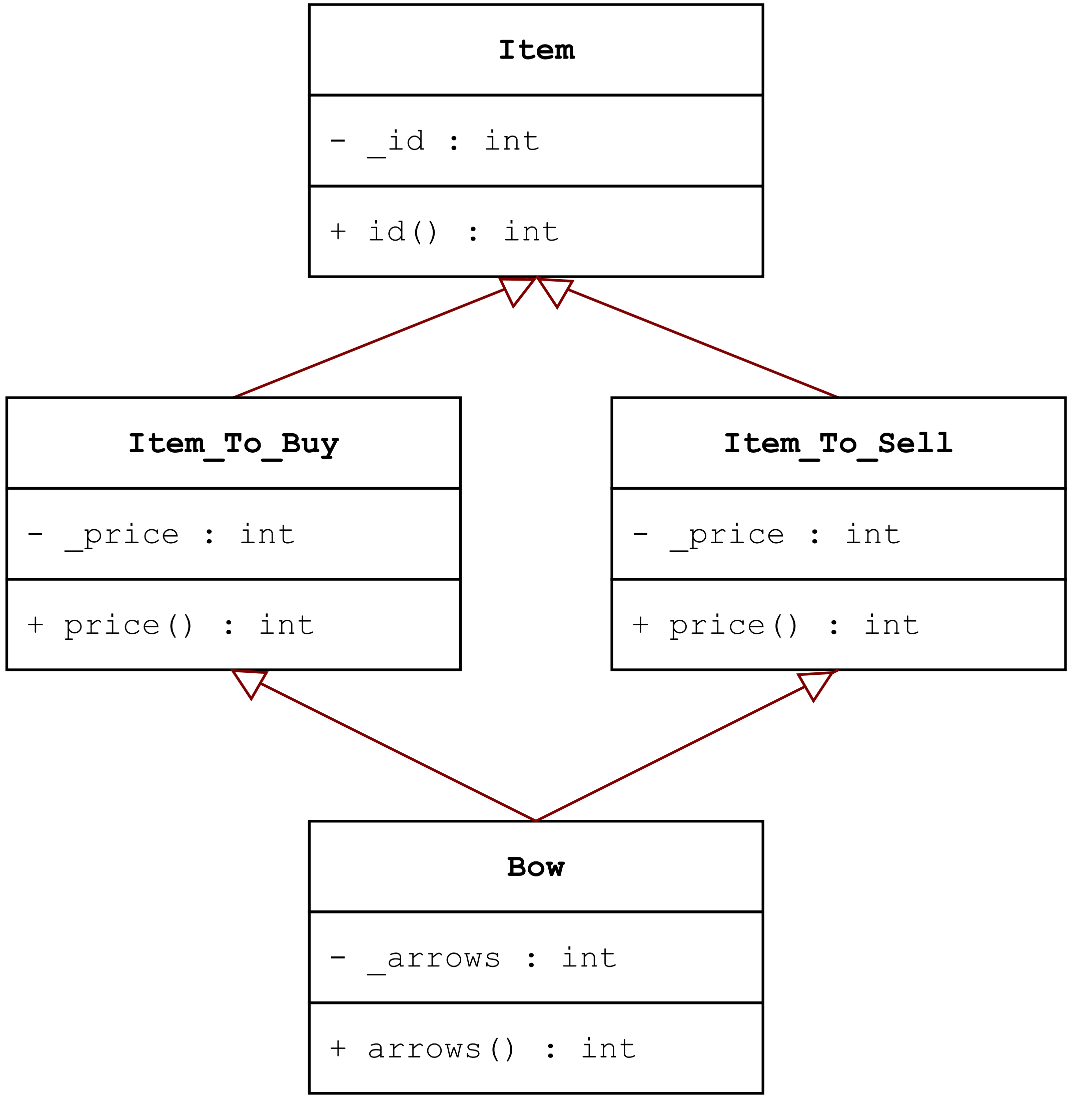


Multiple Inheritance

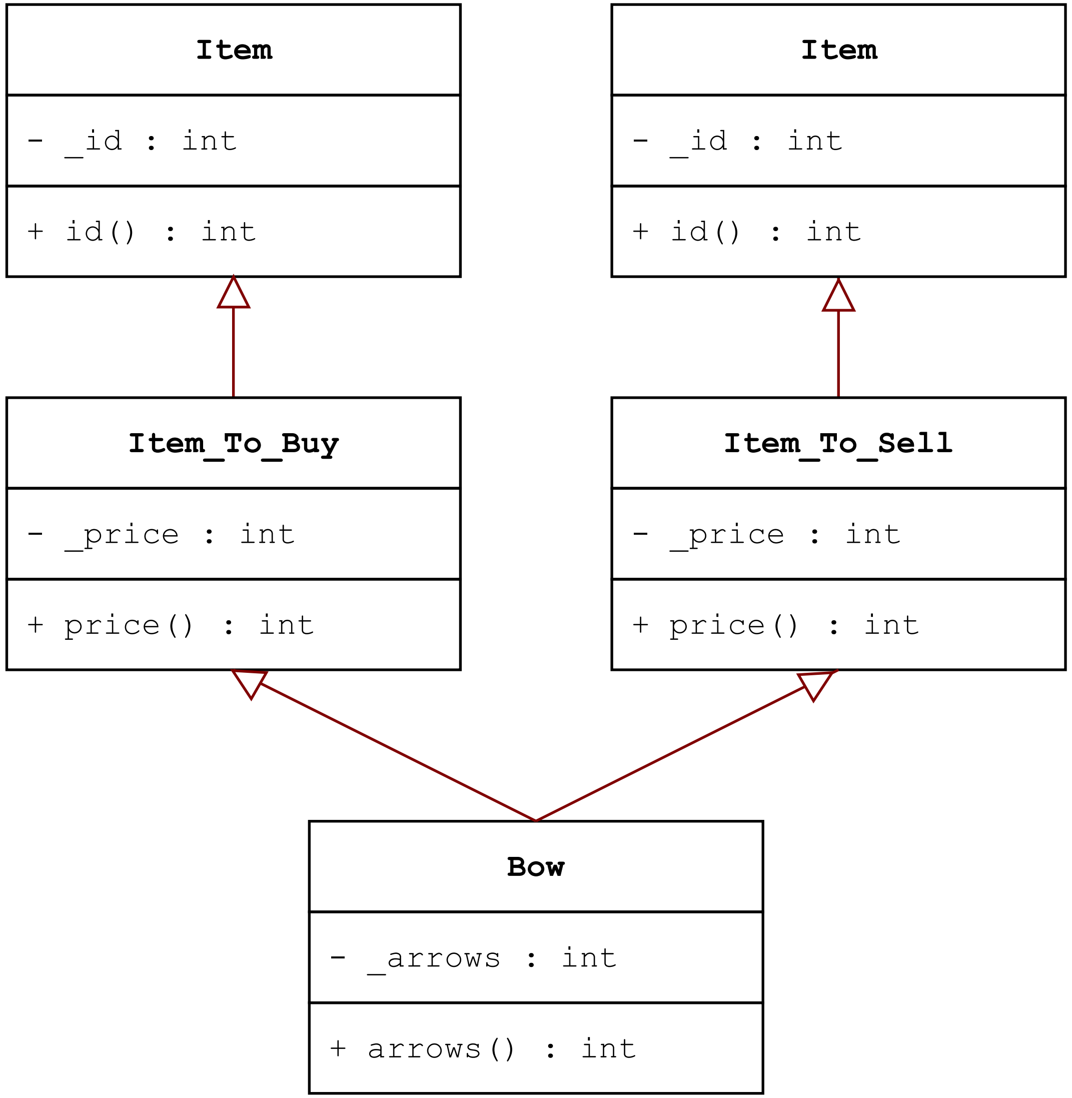
Challenges:

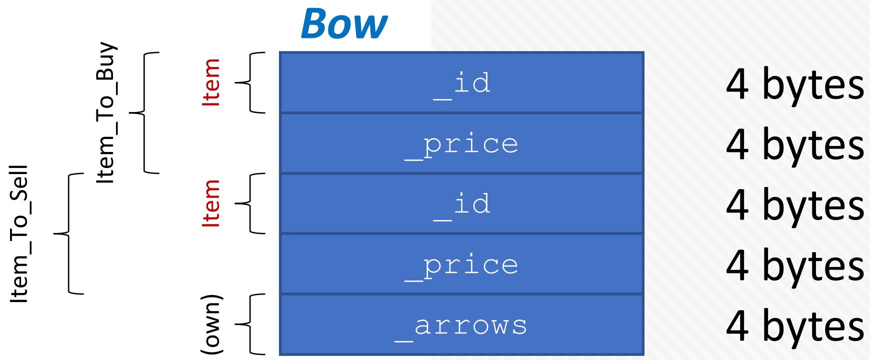
- Members with same names inherited from multiple bases are ambiguous. To access them:
    1. Qualify members with their scope in each access.
    2. Use base class pointers or base class references to access members unambiguously.
- A pointer to a base class differs from the pointer of a derived class for each but the first base class (this results from the memory layout and impacts cast operations).
- If multiple bases are inheriting from the same class, its members will be present in the most derived class multiple times.
- A class cannot inherit from a base class more than once.

Multiple Inheritance



Multiple Inheritance





Virtual Inheritance

Rationale:

- Multiple inheritance without repeating base members.
- Creating derived class that inherit from base classes that share an instance of their own base class.

Virtual Inheritance

Benefits

- Simplifies class hierarchy.
- Removes duplication of data members.
- Removes ambiguity from casts.

Virtual Inheritance

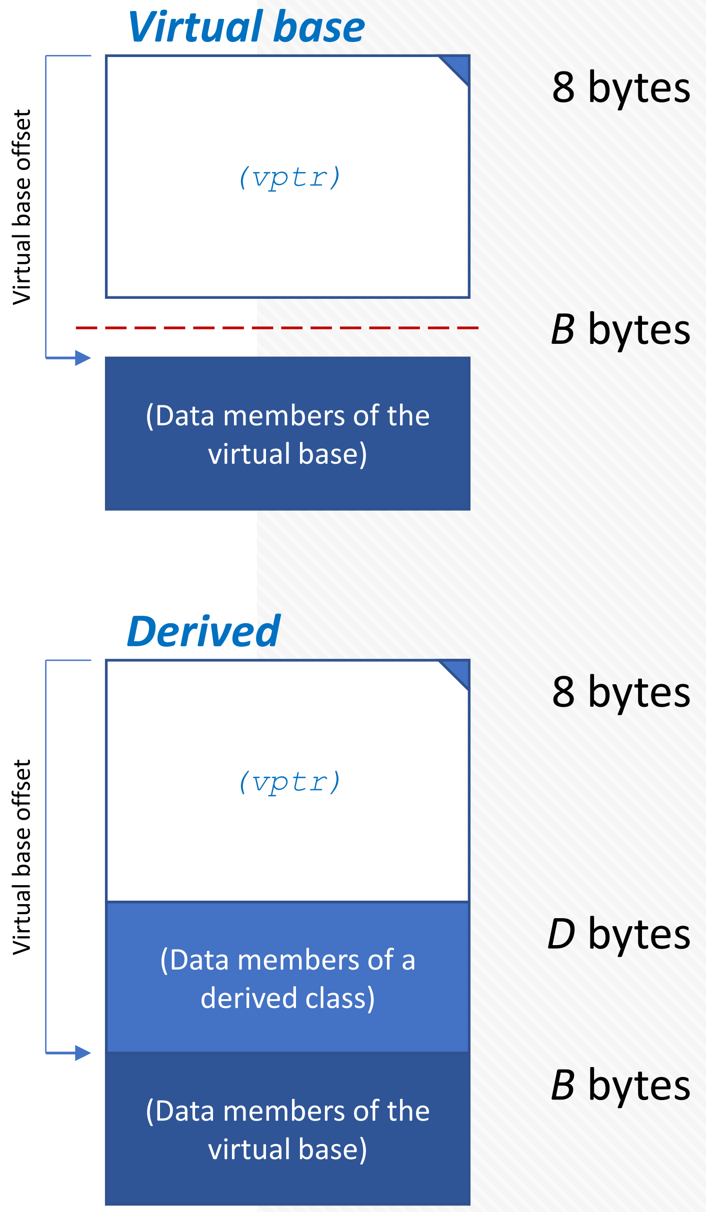
Cost

- Minor run-time cost (uses virtual tables).
- Often increases the size of derived classes as base classes contain additional *vptr* that enlarge objects directly and result in additional padding.
- Requires the programmer to remember about proper construction!

Virtual Inheritance

Layout

- Classes' own data members are located as expected, while data members of their shared virtual base class are located "somewhere else in the object".
- Each class contains *vptr* to its *vtable* that indicates a **virtual base offset** where the first data member of a virtual base class can be found.



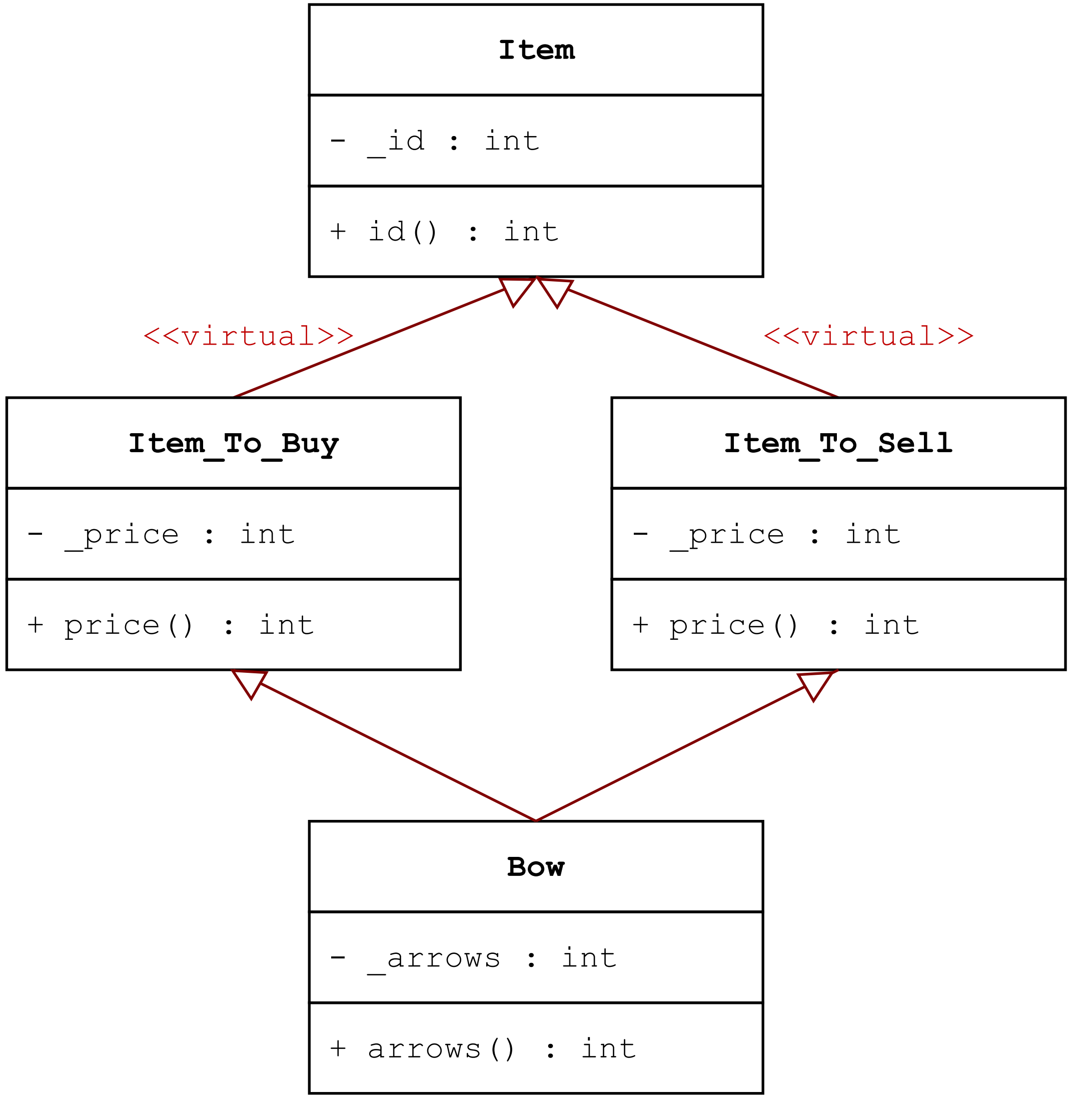
Virtual Inheritance

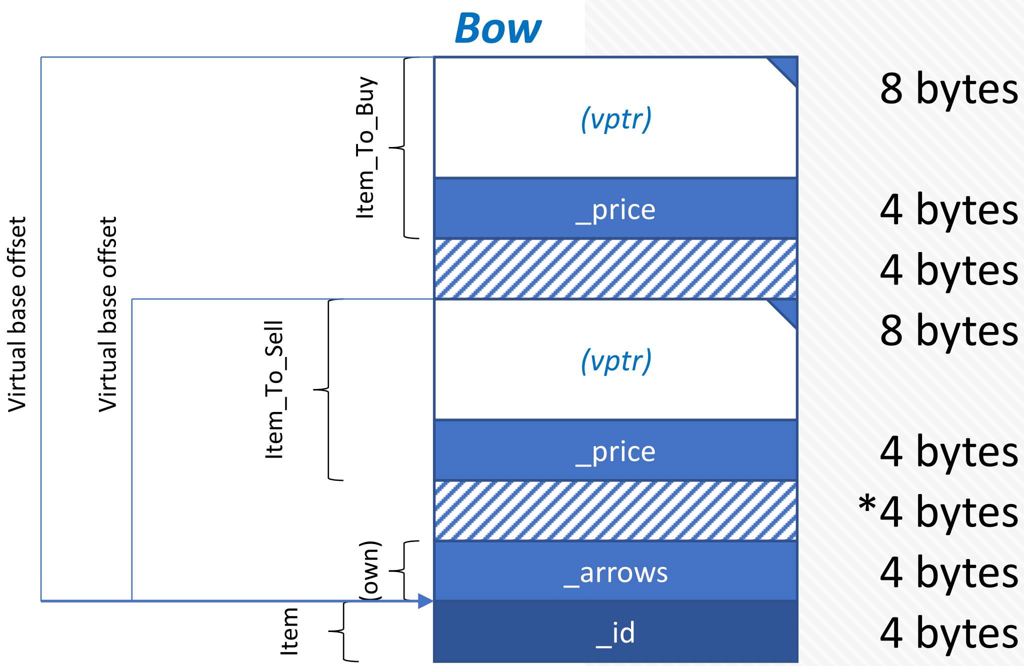
Construction and destruction

- The most-derived class constructs all virtual base classes:
  - Own virtual bases and
  - Virtual bases of all classes in its inheritance hierarchy!
- Then all non-virtual base classes.
- Then the rest of the constructor’s initializer list.
- Lastly, own constructor is executed.

- Destructors are invoked in the reverse order.

Virtual Inheritance





* *Compiler specific padding – if a compiler pads base classes, padding will be as indicated. Otherwise, data member may be shifted upwards in memory.*

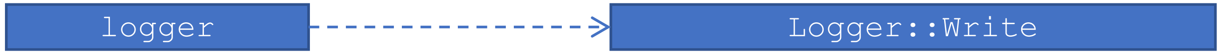
Overriding vs. Overloading

Overriding

- Non-virtual member function call

```
logger.Write("ABC");
```

```
Logger::Write(&logger, "ABC");
```

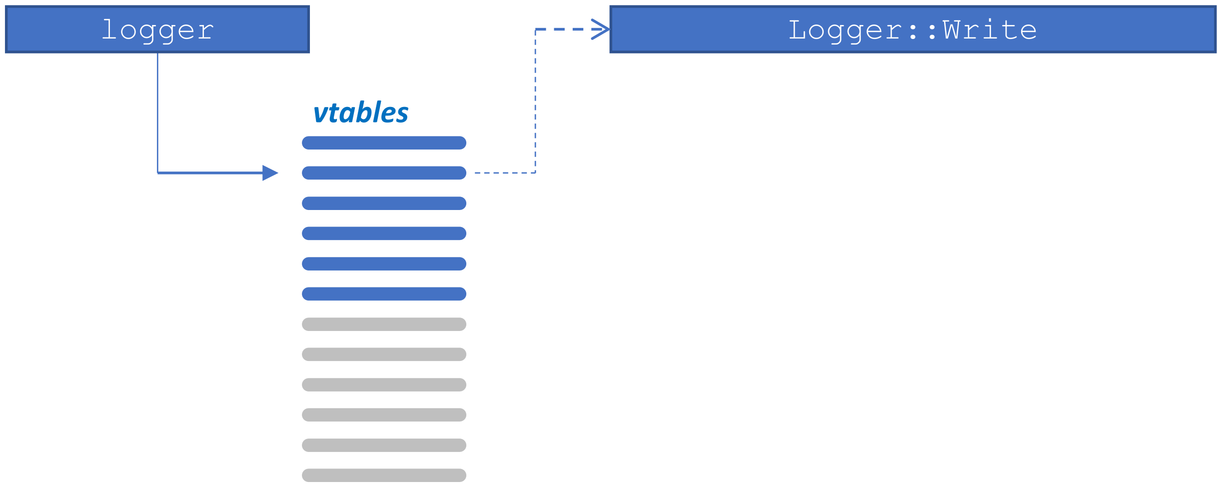


Overriding vs. Overloading

Overriding

- Virtual member function call on a base class pointer or reference

```
logger.Write("ABC");        logger->vptr[index](&logger, "ABC");
```



Overriding vs. Overloading

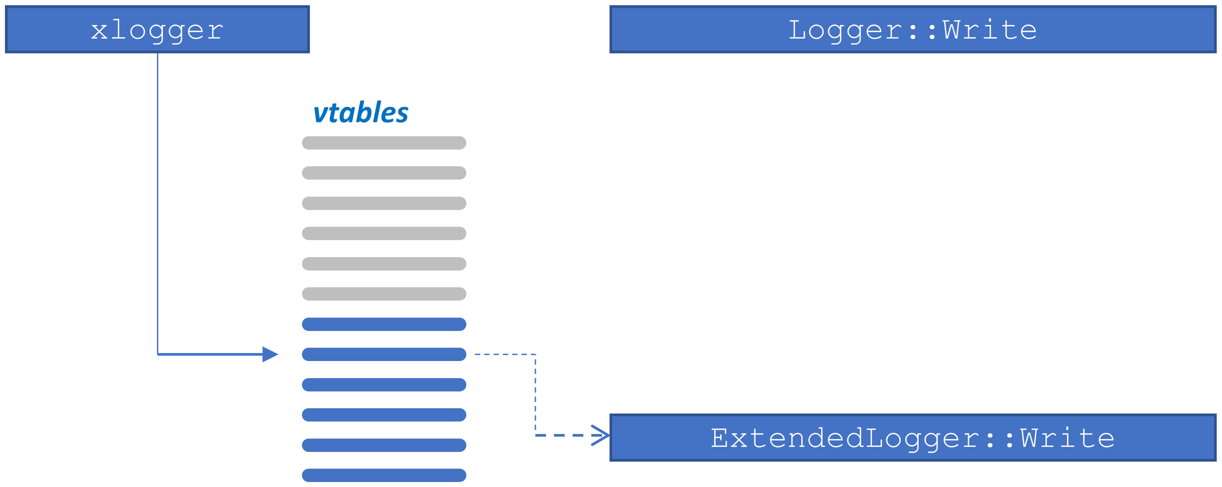
Overriding

- Virtual member function call on a derived pointer or reference

```
xlogger.Write("ABC");
```

→

```
xlogger->vptr[index](&xlogger, "ABC");
```



Overriding vs. Overloading

Overriding

- Definition of a function with the same name and parameters (prototype) in the derived class.
- When a function overrides a **virtual** function, it facilitates dynamic (run-time) **polymorphic behavior**.
- When a function overrides a **non-virtual** function, it **hides** the overridden function. To avoid hiding you can:
    1. Implement a pass-through function in a derived class.
    2. Qualify member functions with their scope in each call.

Overriding vs. Overloading

```cpp
struct Logger
{
  void Write(const std::string& x)
    { std::cout << x; }
};
struct ExtendedLogger : Logger
{
  void Write(const std::string& x)
    { std::cout << "[" << x << "]"; }
};
```

```cpp
struct Logger
{
  virtual void Write(const std::string& x)
    { std::cout << x; }
};
struct ExtendedLogger : Logger
{
  void Write(const std::string& x) override
    { std::cout << "[" << x << "]"; }
};
```

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

// Classes go here

int main()
{
        ExtendedLogger xlogger;
        xlogger.Write("ABC");

        Logger& logger = xlogger;
        logger.Write("ABC");
}
```

The classes above demonstrate a *hiding* behaviour because of the non-virtual override. We expect the following output:

[ABC]ABC

A base class below marks a function as virtual, and so this code results in the virtual override (virtual-ness of member functions is hereditary and it cannot be removed). The output will be:

[ABC][ABC]

Overriding vs. Overloading

Overloading

- Definition of multiple functions with the same name and different parameters in the same scope.
- Feature allowing for static (compile-time) polymorphism.
- Works for global functions as well as member functions, but...

```
struct Logger
{
  void Write(const std::string& x);
  void Write(int x);
};
struct ExtendedLogger : Logger
{
  void Write(double x);
};
```

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

// Classes go here

int main()
{
  ExtendedLogger logger;
  logger.Write("ABC");
}
```

Overriding vs. Overloading

Overloading

- Definition of multiple functions with the same name and different parameters in the same scope.
- Feature allowing for static (compile-time) polymorphism.
- Works for global functions as well as member functions, but a member function overloaded in a derived class ***hides*** all functions with that identifier from the base class.

```
struct Logger
{
  void Write(const std::string& x);
  void Write(int x);
};
struct ExtendedLogger : Logger
{
  void Write(double x);
};
```

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

// Classes go here

int main()
{
  ExtendedLogger logger;
  logger.Write("ABC");    // NC
}
```

Overriding vs. Overloading

Overloading

- To avoid hiding and offer proper overloads, you can:
  1. Implement pass-through functions in a derived class.
  2. Qualify member functions with their scope in each call.
  3. Import identifiers to a derived class to properly overload them.

```
struct Logger
{
  void Write(const std::string& x);
  void Write(int x);
};
struct ExtendedLogger : Logger
{
  using Logger::Write;  // 3
  void Write(double x);
};
```

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

// Classes go here

int main()
{
  ExtendedLogger logger;
  logger.Logger::Write("ABC"); // 2
}
```

Overriding vs. Overloading

Overloading

- Importing base identifiers with `using Base::Identifier;` also allows for changing access modifiers!
- Example: you can inherit members `protected` in a base class, and by importing them under `public` access modifier make them public in a derived class.