Polymorphism in C++

Overview

- **virtual:** The function may be overridden
- **=0:** The function must be virtual and must be overridden
- **override:** The function is meant to override a virtual function in a base class
- **final:** The function is not meant to be overridden

In the absence of any of these controls, a non-static member function is virtual if and only if it overrides a virtual function in a base class.

To get runtime polymorphic behavior in C++, the member functions called must be virtual and objects must be manipulated through pointers or references. When manipulating an object directly (rather than through a pointer or reference), its exact type is known by the compiler so that run-time polymorphism is not needed.

Virtual

By default, a function that overrides a virtual function itself becomes virtual. We can, but do not have to, repeat virtual in a derived class. I is not recommended to use repeating virtual. If you want to be explicit, use override.

Calling a function using the scope resolution operator `::` as is done in Base_class::f() ensures that the virtual mechanism is not used.

Vtables

Vtables (or virtual tables) are how most C++ implementation do polymorphism. For each concrete implementation of a class, there is a table of function pointers to all the virtual methods. A pointer to this table (called the virtual table) exists as a data member in all the objects.

Clearly, to implement polymorphism, the compiler must store some kind of type information in each object of class Employee and use it to call the right version of the virtual function print(). In a typical implementation, the space taken is just enough to hold a pointer: the usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called the `virtual function table` or simply the `vtbl`. Each class with virtual functions has its own vtbl identifying its virtual functions.

Sidenote: The functions in the vtbl allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. The implementation of a caller need only know the location of the vtbl in an object and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the "normal function call" mechanism (within 25%), so efficiency concerns should not deter anyone from using a virtual function where an ordinary function call would be acceptably efficient. Its space overhead is one pointer in each object of a class with virtual functions plus one vtbl for each such class. You pay this overhead only for objects of a class with a virtual function. You choose to pay this overhead only if you need the added functionality virtual functions provide. Had you chosen to use the alternative type-field solution, a comparable amount of space would have been needed for the type field.

Downcasting and Upcasting in C++

Upcasting

It can be achieved by using Polymorphism. C++ allows that a derived class pointer (or reference) to be treated as base class pointer. This is upcasting.

Downcasting

Downcasting is an opposite process to upcasting, which consists in converting base class pointer (or reference) to derived class pointer.

dynamic_cast

`dynamic_cast` in C++ can be used to perform type safe down casting.
`dynamic_cast` is run time polymorphism. The dynamic_cast operator, which safely converts from a pointer (or reference) to a base type to a pointer (or reference) to a derived type.

Example with an Abstract Base Classes

We want to have a function that can run a simulation and print some information:

```C++
void perform(Simulation& s) {
    std::cout << “Running the simulation “ << s.name() << “\n”; s.run(); //run it
}
```

This class must have an name() and a run() member function

```C++
class Simulation{
    public:
        Simulation () {};
        virtual std::string name() const = 0;
        virtual void run() = 0;
};
```

`virtual` means that this function depends on concrete simulation, derived classes can change it. If a derived class is not providing nam() and run() functions, we get a complile time error, as the functions must be provided.