Virtual function

In object-oriented programming, a **virtual function** or **virtual method** is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. This concept is an important part of the polymorphism portion of object-oriented programming (OOP).

**Purpose**

The concept of the virtual function solves the following problem:

In object-oriented programming when a derived class inherits from a base class, an object of the derived class may be referred to via a pointer or reference of either the base class type or the derived class type. If there are base class methods overridden by the derived class, the method actually called by such a reference or pointer can be bound either 'early' (by the compiler), according to the declared type of the pointer or reference, or 'late' (i.e. by the runtime system of the language), according to the actual type of the object referred to.

Virtual functions are resolved 'late'. If the function in question is 'virtual' in the base class, the most-derived class's implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer or reference. If it is not 'virtual', the method is resolved 'early' and the function called is selected according to the declared type of the pointer or reference.

Virtual functions allow a program to call methods that don't necessarily even exist at the moment the code is compiled.

In C++, **virtual methods** are declared by prepending the `virtual` keyword to the function's declaration in the base class. This modifier is inherited by all implementations of that method in derived classes, meaning that they can continue to over-ride each other and be late-bound.
Example

For example, a base class Animal could have a virtual function eat. Subclass Fish would implement eat() differently than subclass Wolf, but one can invoke eat() on any class instance referred to as Animal, and get the eat() behavior of the specific subclass.

```cpp
class Animal
{
    void /*nonvirtual*/ move() {
        cout << "This animal moves in some way" << endl;
    }
    virtual void eat() {}
};

// The class "Animal" may possess a declaration for eat() if desired.
class Llama : public Animal
{
    // The non virtual function move() is inherited but cannot be overridden
    void eat() {
        cout << "Llamas eat grass!" << endl;
    }
};
```

This allows a programmer to process a list of objects of class Animal, telling each in turn to eat (by calling eat()), without needing to know what kind of animal may be in the list, how each animal eats, or what the complete set of possible animal types might be.

Abstract classes and pure virtual functions

A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class, if that class is not abstract. Classes containing pure virtual methods are termed "abstract" and they cannot be instantiated directly. A subclass of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class. Pure virtual methods typically have a declaration (signature) and no definition (implementation).

As an example, an abstract base class MathSymbol may provide a pure virtual function doOperation(), and derived classes Plus and Minus implement doOperation() to provide concrete implementations. Implementing doOperation() would not make sense in the MathSymbol class, as MathSymbol is an
abstract concept whose behaviour is defined solely for each given kind (subclass) of \texttt{MathSymbol}. Similarly, a given subclass of \texttt{MathSymbol} would not be complete without an implementation of \texttt{doOperation()}. Although pure virtual methods typically have no implementation in the class that declares them, pure virtual methods in C++ are permitted to contain an implementation in their declaring class, providing fallback or default behaviour that a derived class can delegate to, if appropriate.

Pure virtual functions can also be used where the method declarations are being used to define an interface - similar to what the interface keyword in Java explicitly specifies. In such a use, derived classes will supply all implementations. In such a design pattern, the abstract class which serves as an interface will contain \textit{only} pure virtual functions, but no data members or ordinary methods. In C++, using such purely abstract classes as interfaces works because C++ supports multiple inheritance. However, because many OOP languages do not support multiple inheritance, they often provide a separate interface mechanism. An example is the Java programming language.

**Behavior during construction and destruction**

Languages differ in their behavior while the constructor or destructor of an object is running. For some languages, notably C++, the virtual dispatching mechanism has different semantics during construction and destruction of an object. While it is recommended that virtual function calls in constructors should be avoided for C++, in some other languages, for example C# and Java, the derived implementation can be called during construction and design patterns such as the Abstract Factory Pattern actively promote this usage in languages supporting the ability.

**Virtual destructors**

Object-oriented languages typically manage memory allocation and deallocation automatically when objects are created and destroyed. However, some object-oriented languages allow a custom destructor method to be implemented, if desired. If the language in question uses automatic memory management, the custom destructor (generally called a finalizer in this context) that is called is certain to be the appropriate one for the object in question. For example, if an object of type \texttt{Wolf} that inherits \texttt{Animal} is created, and both have custom destructors, the one called will be the one declared in \texttt{Wolf}.

In manual memory management contexts, the situation can be more complex, particularly as relates to static dispatch. If an object of type \texttt{Wolf} is created but pointed to by an \texttt{Animal} pointer, and it is this \texttt{Animal} pointer type that is deleted, the destructor called may actually be the one defined for \texttt{Animal} and not the one for \texttt{Wolf}, unless the destructor is virtual. This is particularly the case with C++, where the behavior is a common source of programming errors.

**References**

Article Sources and Contributors


Image Sources, Licenses and Contributors


License

Creative Commons Attribution-Share Alike 3.0
creativecommons.org/licenses/by-sa/3.0