CS 246 Fall 2018 — Tutorial 8

November 7, 2018

Summary

1 `virtual` and `override`

- When working with inherited classes, we will often want to define the methods so that it will work differently for separate subclasses:

```
// Full example at animals/animals.cc
struct Animal {
    virtual bool fly() const {
        return false;
    }
};

struct Bird : public Animal {
    bool fly() const override {
        return true;
    }
};

struct Goose : public Bird {
    bool fly() const override {
        cout << "THANK MR. GOOSE" << endl;
        return true;
    }
};
```

- Note that we have declared `fly()` as a `virtual` method.
  - Declaring a method virtual means if we override it in a subclass, the subclass version of the method will be called through polymorphic pointers.

- If we do not override the method, the definition in the closest related ancestor will be used. For instance, calling `fly()` on a `Cat` will return `false`.

- **Note:** the virtual method will only be called when dealing with objects *through pointers/references*. This does not work with objects directly.
  - For example:

```
// What is this line of code actually doing?
Animal a = Bird{};
a.fly(); // returns false
```

- Using the keyword `override` tells the compiler to check that the method is actually the override of a `virtual` method in a superclass. This will causes a compiler error if it can't find a `virtual` function of the same name.
  - Although the keyword is not required to override a `virtual` method, *it is highly recommended to prevent hours of debugging for a simple mistake (such as a typo in the function signature).*

2 Pure Virtual Methods

- **Pure Virtual methods** are methods that subclasses will need to provide an implementation for if they want to be instantiable.
- Most of the times, pure virtual methods will not have an implementation.
  - Pure virtual methods can have an implementation.
- We declare a method as pure virtual when we add `virtual` to the front and `= 0` to the end of its declaration in the class definition:

```
class A {
    virtual void someFunction() = 0;
};
```

- Typically, pure virtual methods are used if it does not make sense for a method to have an implementation in the base class, or if we want to make the class abstract.

3 Abstract and Concrete Classes

- A class is **abstract** if it has one or more pure virtual methods, and a class is **concrete** if it has no pure virtual methods.
- This means that all classes must be either abstract or concrete, but not both.

- Abstract classes are not **instantiable**. This means that you cannot create objects of abstract classes, i.e. you can only create objects of concrete classes.

```
class A {
    int a;

public:
    A(int a) : a{a} {}
    virtual void foo() = 0;
};

class B {
    int b;

public:
    B(int b) : b{b} {}
    void bar() {
        cout << "This is not pure virtual" << endl;
    }
};

int main() {
    A a{1}; // This will cause a compiler error
    B b{2}; // This is fine
}
```

- The purpose of an abstract class is to allow subclasses to inherit from a base class containing information that is common to all subclasses, but it doesn't make sense to have an instance of the base class.

- A subclass of an abstract class is also abstract. If the subclass implemented *all* pure virtual methods of the base class, then it becomes concrete.

```
class C : public A {
    int c;

public:
    C(int a, int c) : A{a}, c{c} {}
    void foo() override {
        cout << "Overriding foo" << endl;
    }
};

int main() {
    // A a{1}; // This will still cause a compiler error
    C c{1, 2}; // This is fine
}
```

4 Destructors Revisited

- Now with inheritance and (pure) virtual methods, we need to revisit the destructor.
  1. If your want the class to be inherited, the destructor should always be `virtual`. Why?
     - To ensure the right destructor is called when polymorphism is involved.
  2. They must always have an implementation; even if they are pure virtual. Why?
     - Because the destructor of the base class is always called when a derived class is destroyed.
     - This is needed since every derived class possesses the components of the base class.
     - Thus, the destructor of the base class must have an implementation (even if the implementation is empty).

```
class B {
public:
    // The method hello is pure virtual,
    // which makes B an abstract class.
    virtual ~B() = 0;
    virtual string hello() = 0;
};

class A : public B {
    A *arr;

public:
    A() : arr{new A[5]} {}
    ~A() {
        delete[] arr;
    }
    string hello() override {
        return "Bonjour!";
    }
};

// IN B.cc

// We need this implementation, even if it's empty
B::~B() {}
```

- Class A inherits from the abstract class B which has a pure virtual destructor.
  - Now, every class that inherits from B has to provide its own implementation of the destructor (this includes using the compiler-generated destructor).

  - There is no need to call the superclass destructor explicitly.

5 Vectors

- In the C++ STL (Standard Template Library), there is a `vector` template class that can be used in place of a dynamic array.
- Vectors make resizing arrays easier.
  - Any time you find yourself in need of a heap-allocated array, you can achieve the same result by using a vector on the stack
  - Vectors will manage the heap-allocated array for you internally
- Example:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // creating a vector of integers
    vector<int> arr;
    int x;

    while (cin >> x) {
        // Continuously adding integers to arr
        // and increasing the size of the vector
        // without manually allocating more memory
        arr.emplace_back(x);
    }
    // Iterating over the vector
    for (int i = 0; i < arr.size(); i++) {
        // outputting the values in arr
        cout << arr[i];
    }
}
```

- Vectors come with iterators
  - This means you can use a range-based for loop.
  - You can also use functions inside the `algorithm` library, since they all take iterators as arguments.
- Some helpful vector methods

- void vector<T>::emplace_back(params): takes the parameters to a constructor call of T for the object being stored in the list and builds the object as the last object.
- T& vector::back(): returns a reference to the last element in the array.
- void vector::pop_back(): deletes the last element in the array.
- T& vector::operator[](int n): returns the element in position n. Does not do range checking.
- T& vector::at(int n): returns the element in position n. Does range checking.
- vector<T>::iterator vector::begin(): returns an iterator to the first element in the vector.
- vector<T>::iterator vector::end(): returns an iterator to the position one past the end of the vector.
- int vector::size(): returns the number of elements in the vector.