CS 162
Intro to Computer Science II

Lecture 19
Virtual vs. Pure virtual, Abstract class
Vector
3/1/24
Odds and Ends

- Due Sunday 3/3 midnight
  - Quiz 4
Virtual functions

• Use virtual functions and pointers together to bypass static binding

• A virtual function is one that is declared in the base class with the virtual keyword

    virtual void some_function();

    • This indicates to the compiler that dynamic binding should be used at runtime, to determine which version of the function to call based on what kind of object is being pointed to.
    • i.e.

        Animal* a_ptr = &m1;
        a_ptr->make_noise(4); // make_noise() is a virtual function

• Demo...
More details on virtual

• The determination about which function to call at runtime instead of compile time:
  • When each function is called, C++ will figure out what specific class of object is being pointed to by the base class pointer (i.e. a_ptr)
  • Once it figures out what class of object is pointed to, it will traverse up the inheritance chain (first checking Monkey, then Animal) until it finds an implementation of the called function.
  • The first class to implement the called function in the chain will have that function called.

• This is true polymorphism: a pointer to an Animal object is being treated differently depending on what kind of object it actually points to.
Memory layout of virtual functions

• Let’s start with a simple class:

```cpp
class Base {
    private:
        int var;
    public:
        void fun1();
        void fun2();
};

Base b_obj;
```

• Memory Layout

```cpp
b_obj

<table>
<thead>
<tr>
<th>Addr</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>void Base::fun1()</td>
<td>(ptr to Base::fun1())</td>
</tr>
<tr>
<td>0x2000</td>
<td>void Base::fun2()</td>
<td>(ptr to Base::fun2())</td>
</tr>
</tbody>
</table>

var

//Addr: 0x1000
void Base::fun1() {
}

//Addr: 0x2000
void Base::fun2() {
}```
Memory layout of virtual functions

• Now, let’s add two virtual func:

```cpp
class Base {
private:
    int var;
public:
    void fun1();
    void fun2();
    virtual void fun3();
    virtual void fun4();
};
```

Base b_obj;

• Memory Layout

```
Function def:
//Addr: 0x1000
void Base::fun1() {
}
//Addr: 0x2000
void Base::fun2() {
}
//Addr: 0x3000
void Base::fun3() {
}
//Addr: 0x4000
void Base::fun4() {
}
```
Memory layout of virtual functions

- **vptr (Virtual Pointer)**
  - The pointer which contains address of the Virtual Table
  - vptr is associated with **object**, meaning that each object of that class is having a different vptr pointing to the same Virtual Table

- **Virtual Table (VTable)**
  - A memory space reserved by compiler to place address of virtual functions
  - VTable is associated with **class**, meaning that there will be at most 1 for each class, no matter how many objects of that class have been created. All objects of that class will share the same VTable

```
Function def:

//Addr: 0x3000
void Base::fun3() {
}

//Addr: 0x4000
void Base::fun4() {
}
```
Memory layout of virtual functions

• Memory layout for obj1, obj2, and obj3:

```
//Addr: 0x1000
void Base::fun1() {
}

//Addr: 0x2000
void Base::fun2() {
}

//Addr: 0x3000
void Base::fun3() {
}

//Addr: 0x4000
void Base::fun4() {
}
```

Virtual Table of Base:

0x3000 (ptr to Base::fun3())

0x4000 (ptr to Base::fun4())
Memory layout of virtual functions

Let’s add a class derived from Base

class Base {
    private:
        int var;
    public:
        void fun1();
        void fun2();
        virtual void fun3();
        virtual void fun4();
};
class Derived : public Base {
    public:
        void fun3();
};
Base b_obj; Derived d_obj;

Memory Layout

Virtual Table of Base

<table>
<thead>
<tr>
<th>Address</th>
<th>Function Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>Base::fun1()</td>
</tr>
<tr>
<td>0x1000</td>
<td>Base::fun2()</td>
</tr>
<tr>
<td>0x3000</td>
<td>Base::fun3()</td>
</tr>
<tr>
<td>0x4000</td>
<td>Base::fun4()</td>
</tr>
</tbody>
</table>

Virtual Table of Derived

<table>
<thead>
<tr>
<th>Address</th>
<th>Function Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5000</td>
<td>Derived::fun3()</td>
</tr>
<tr>
<td>0x4000</td>
<td>Base::fun4()</td>
</tr>
</tbody>
</table>

Function def:

```cpp
//Addr: 0x1000
void Base::fun1() {
}

//Addr: 0x2000
void Base::fun2() {
}

//Addr: 0x3000
void Base::fun3() {
}

//Addr: 0x4000
void Base::fun4() {
}

//Addr: 0x5000
void Derived::fun3() {
}
```
Memory layout of virtual functions

• Let’s invoke these methods

Base *b_ptr;
Base b_obj;
Derived d_obj;

b_obj.fun3();
b_ptr = &b_obj;
b_ptr->fun3();

b_ptr = &d_obj;
b_ptr->fun3();

• Memory Layout

Function def:

```
//Addr: 0x1000
void Base::fun1() {
}

//Addr: 0x2000
void Base::fun2() {
}

//Addr: 0x3000
void Base::fun3() {
}

//Addr: 0x4000
void Base::fun4() {
}
```

```
//Addr: 0x3000
(void) Base::fun3();
```
Virtual Destructors

• Let’s define destructors for our Animal, Monkey, Sea_Otter, and Sloth class

• Demo...

• We’ve created a Monkey object, but only the Animal destructor is being called.
  • If we’d allocated memory in the Monkey class that were relying on the destructor to clean, that memory would never be freed, resulting in a memory leak.
Virtual Destructors

• Thus, when using polymorphism, it’s very important to make your base class’s destructor virtual

• Demo...
Additional notes:

• When you declare a function as virtual in a base class, it automatically becomes virtual in all classes derived from that base class, whether you declare it as virtual there or not.

• This form of polymorphism works with references as well as pointers.

```cpp
Animal &a = m1;

a.make_noise(); // will call the make_noise() in Monkey class if it is declared to be virtual in the animal class
```

• This allows us to pass an Monkey object into a function that takes a reference to an Animal object as an argument.

```cpp
void some_func(Animal& a);
Call: some_func(m1);
```
Abstract classes

• Abstract class
  • can only be used as a base class
  • you cannot instantiate objects of an abstract class

• A class becomes abstract when it has at least one virtual function without a definition
  • Such a function is known as a pure virtual function

• To declare a pure virtual function, simply set it equal to zero:

```cpp
class Animal{
    public:

    ...

    virtual void make_noise() = 0;  // =0 means no definition

};
```
Abstract classes

class Animal{
    public:
    ...
    virtual void make_noise() = 0; // = 0 means no definition
};

• Because the make_noise() is purely virtual, the Animal class becomes an abstract class. That means we cannot create an Animal object, i.e. both of these becomes errors:
    Animal a;
    Animal *a = new Animal;

• But you can still create pointers of abstract class, and let them point to classes derived from the abstract class, i.e.
    Animal *a1 = new Monkey;
    Animal *a2 = new Sloth;
    a1->make_noise() // call make_noise() of Monkey
    a2->make_noise() // call make_noise() of Sloth
Use of Abstract Classes

• Note: each pure virtual function needs a definition in all its derived class(es)

• All the common code in derived classes is written in abstract class
  • Same as normal inheritance, why we need abstract class?
Use of Abstract Classes

• Let’s consider our demo...

• Make `make_noise()` pure virtual in Animal class
  • Why? Because every animal can make different noises
  • We wanted all derived class to define this function in their class to make noises

• Demo ...

• Animal now has become abstract class

• Is there any use of Animal class objects?
  • No, they represent nothing.
  • So we need abstract class to prevent making objects of that class

• If you let any 3rd party to implement a Tiger class, making Animal abstract will enforce them to implement the `make_noise()` in the Tiger class
Vector: Example of a template class

• Arrays that can grow and shrink in length while the program is running
• Formed from template class in the Standard Template Library (STL)
• Has a base type and stores a collections of this base type: vector <int> v;
• Still starts indexing at 0, can still use [] to access things
• Use push_back() to add one element to the end
• Number of elements == size
• How much memory currently allocated == capacity
More vectors

- We need to `#include <vector>` to use `std::vector`
- We use `push_back()` to add elements
- Use `pop_back()` to get rid of the last element
- `size()` – how many elements inside the vector
- `capacity()` – how many elements it can hold (allocated memory)
- We can use `operator[]` or `at()` to access specific elements
  - i.e.
    - `vec[1]` or `vec.at(1)`
  - Note: `[]` does not throw an exception for an out-of-range that `at()` does
More vectors

• To make 2D vectors:

```cpp
vector<vector<int>> vec_2d;
for (int i = 0; i < row; i++){
    vector<int> row_vec;
    for (int j = 0; j < col; j++)
        row_vec.push_back(i * j);
    vec_2d.push_back(row_vec);
}
```

• Note:
  • We need the extra space between angle brackets in the declaration of `vec_2d`, to tell it from the `>>` operator
• `std::vector` has a lot more functionalities:
  • It has constructors that allow us to initialize the vector with a specified size and even a specified initial value:
    ```cpp
    vector <int> vec1(20); // Allocate vector of size 20
    vector <int> vec2(10,7); // Fill vector with 10 7s
    ```
More vectors

• **std::vector** has a lot more functionalities:
  • It has constructors that allow us to initialize the vector with a specified size and even a specified initial value:
    ```cpp
    vector <int> vec1(20); // Allocate vector of size 20
    vector <int> vec2(10,7); // Fill vector with 10 7s
    ```
  • `.size()` - returns the size of the vector
  • `.resize()` - changes size
  • `.empty()` - test whether the vector is empty
  • `.front()` - access the first element
  • `.back()` - access the last element
  • `.clear()` - clear content
  • `.swap()` - swap content