Pointers to functions

- C++ doesn't require that pointers point only to data;
  - Pointer may also point to functions.

- Function pointers point to memory addresses where functions are stored, e.g.

```
void (*fp) (void);
```

- A function's name may be viewed as a constant pointer to a function.

Pointers to functions

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

void func_0() {
    cout << "I am func_0():" << endl;
}

int main() {
    void (*fp) () = func_0;

    fp();
    (*fp)();
    (fp)();

    return 0;
}
```

```
I am func_0():
I am func_0():
I am func_0():
```

- Function name may be viewed as a const pointer.
- Parameters and return type must match

- Either form may be used to invoke the function

Arrays of Pointers to functions

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

void func_0() { cout << "I am func_0():" << endl; }
void func_1() { cout << "I am func_1():" << endl; }
void func_2() { cout << "I am func_2():" << endl; }
void func_3() { cout << "I am func_3():" << endl; }

int main() {
    void (*fp[5]) () = {
        func_0,
        func_1,
        func_2,
        func_3
    };
    fp[4] = nullptr;

    int i=0;
    while (fp[i]) {
        fp[i]();
        i++;
    }
    return 0;
}
```

```
I am func_0():
I am func_1():
I am func_2():
I am func_3():
```

- We can also have arrays of function pointers
- In this particular example, we are using a nullptr as a sentinel.

- **Why do we need all of this?**
  - Because we need to learn about **"vtbl"**

The vtbl

- Is an array (i.e. a table) of function pointers.
- Each class that has at least one virtual method, has a vtbl
- Each object has a pointer (vptr) to its class’ vtbl
- **Virtual functions are invoked using the vtbl (which is an array of function pointers).**

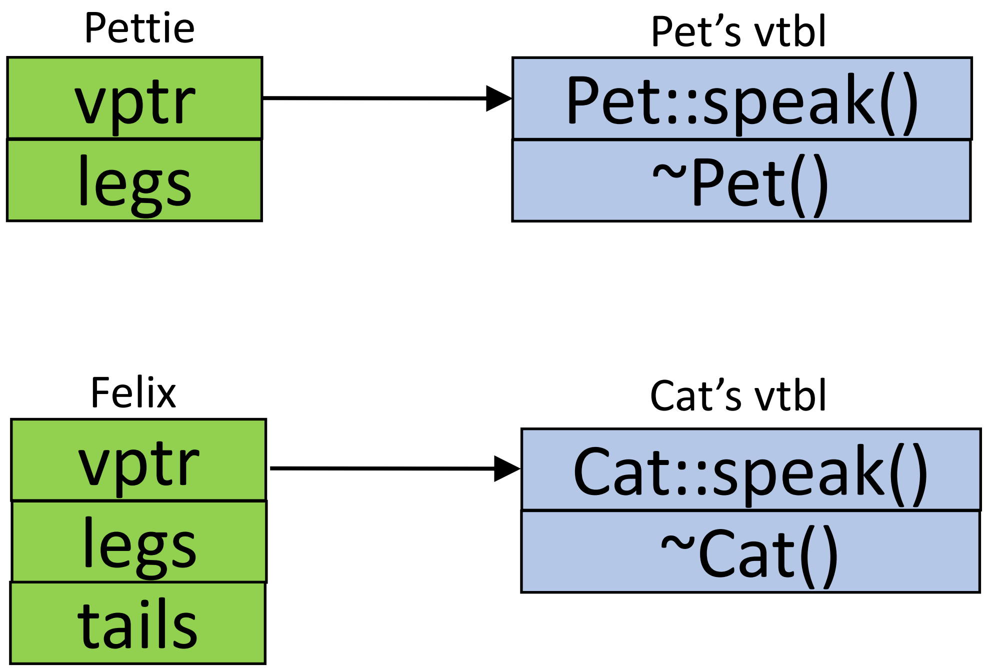
The vtbl

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

class Pet {
public:
    virtual void speak() { cout<< "Animal speaking ..\n"; }
    virtual ~Pet() {}
private:
    int legs;
};
class Cat : public Pet {
public:
    void speak() { cout << "Cat meowing ..\n"; }
private:
    int tails;
};

int main() {
    Pet Pettie;
    Cat Felix;
    Pet& aRef = Felix;

    Pettie.speak();
    Felix.speak();
    aRef.speak();
}
```

- **How many vtbl's?**
- **How many entries in vtbl's?**

The vtbl

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

class Pet {
public:
    virtual void speak() { cout<< "Animal speaking ..\n"; }
    virtual ~Pet() {}
private:
    int legs;
};
class Cat : public Pet {
public:
    void speak() { cout << "Cat meowing ..\n"; }
private:
    int tails;
};

int main() {
    Pet Pettie;
    Cat Felix;
    Pet& aRef = Felix;

    Pettie.speak();
    Felix.speak();
    aRef.speak();
}
```

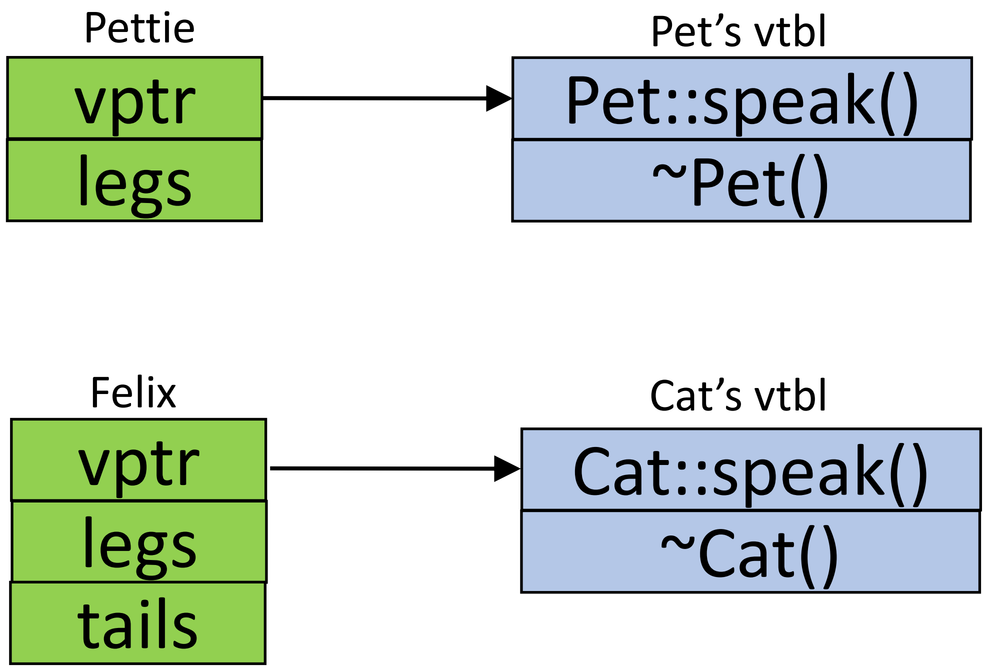


The vtbl

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

class Pet {
public:
    virtual void speak() { cout<< "Animal speaking ..\n"; }
    virtual ~Pet() {}
private:
    int legs;
};
class Cat : public Pet {
public:
    void speak() { cout << "Cat meowing ..\n"; }
private:
    int tails;
};

int main() {
    Pet Pettie;
    Cat Felix;
    Pet& aRef = Felix;

    Pettie.speak();
    Felix.speak();
    aRef.speak();
}
```

Animal speaking ..
Cat meowing ..
Cat meowing ..

A reference is used:
Declared type: Pet
Actual Type: Cat

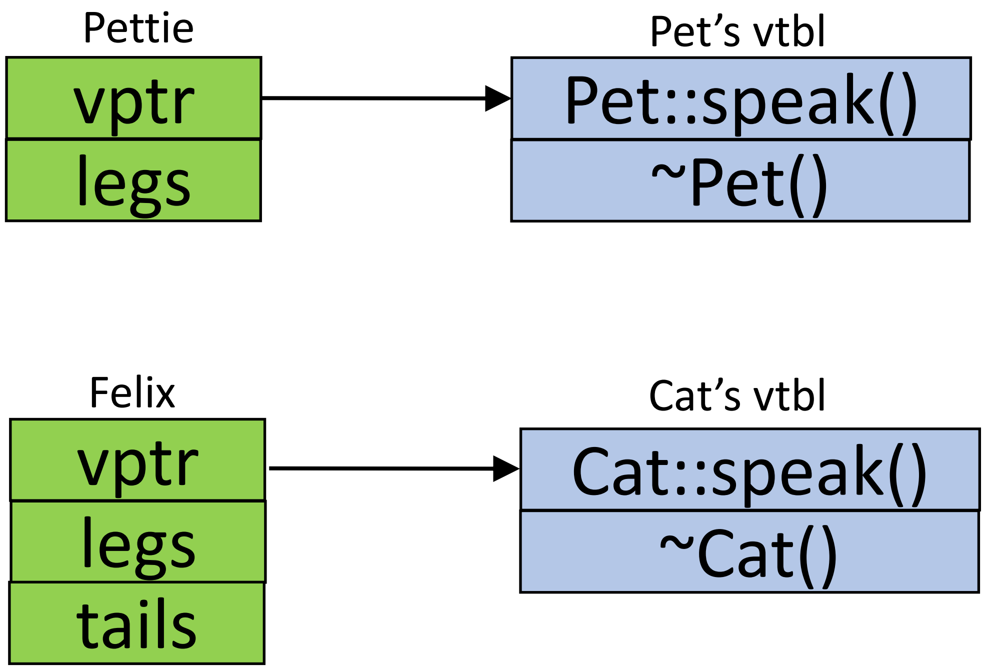


- **Virtual functions are invoked using the vtbl**
  - → "**Dynamic Binding**"
  - Call depends on the "**actual**" type of the object because the object contains a pointer to it's class' vtbl.
- Non virtual functions are resolved at compile time based on the "**declared**" type of the object.

The vtbl

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

class Pet {
public:
    virtual void speak() { cout<< "Animal speaking ..\n"; }
    virtual ~Pet() {}
private:
    int legs;
};
class Cat : public Pet {
public:
    void speak() { cout << "Cat meowing ..\n"; }
private:
    int tails;
};

int main() {
    Pet Pettie;
    Cat Felix;
    Pet& aRef = Felix;

    Pettie.speak();
    Felix.speak();
    aRef.Pet::speak();
}
```

```
Animal speaking ..
Cat meowing ..
Animal speaking ..
```



- Using a fully qualified method name → method is not invoked using the vtbl, rather via a direct function call.

Vtbl – another example

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

class A1 {};
class B1 : public A1 {};
class C1 : public B1 {};

class A2 {
public:
    void foo() {}
};
class B2 : public A2 {};
class C2 : public B2 {};

class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };

int main() {
    // A class with no members
    cout << "sizeof(A1): " << sizeof(A1)
        << ",  sizeof(B1): " << sizeof(B1)
        << ", sizeof(C1) " << sizeof(C1) << endl;

    // A class with just non-virtual methods
    cout << "sizeof(A2): " << sizeof(A2)
        << ",  sizeof(B2): " << sizeof(B2)
        << ", sizeof(C2) " << sizeof(C2) << endl;

    // A class with a virtual method
    cout << "sizeof(A3): " << sizeof(A3)
        << ",  sizeof(B3): " << sizeof(B3)
        << ", sizeof(C3) " << sizeof(C3) << endl;
    cout << "============\n";

    // What about the addresses?
    A3 a3;
    B3 b3;
    C3 c3;
    cout << &a3 << ' '
         << &b3 << ' '
         << &c3 << endl;

}
```

Vtbl – another example, cont.

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

class A1 {};
class B1 : public A1 {};
class C1 : public B1 {};

class A2 {
public:
    void foo() {}
};
class B2 : public A2 {};
class C2 : public B2 {};

class A3 {
public:
    virtual void foo() {}
};
class B3 : public A3 { };
class C3 : public B3 { };

int main() {
    // A class with no members
    cout << "sizeof(A1): " << sizeof(A1)
        << ",  sizeof(B1): " << sizeof(B1)
        << ", sizeof(C1) " << sizeof(C1) << endl;

    // A class with just non-virtual methods
    cout << "sizeof(A2): " << sizeof(A2)
        << ",  sizeof(B2): " << sizeof(B2)
        << ", sizeof(C2) " << sizeof(C2) << endl;

    // A class with a virtual method
    cout << "sizeof(A3): " << sizeof(A3)
        << ",  sizeof(B3): " << sizeof(B3)
        << ", sizeof(C3) " << sizeof(C3) << endl;
    cout << "============\n";

    // What about the vtable?
    A3 a3;
    B3 b3;
    C3 c3;
    cout << &a3 << ' '
         << &b3 << ' '
         << &c3 << endl;

}
```

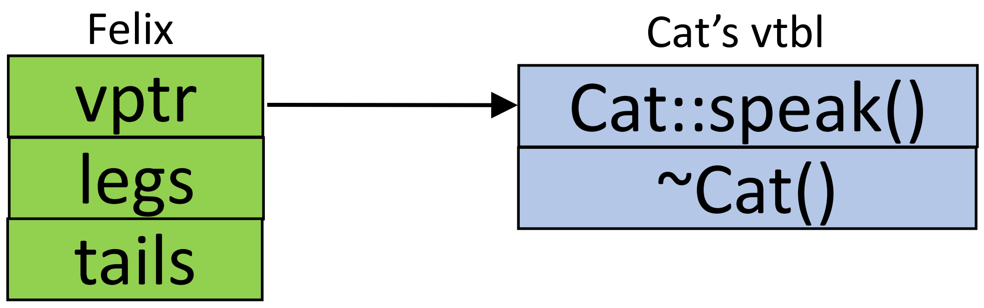
```
sizeof(A1): 1,  sizeof(B1): 1, sizeof(C1) 1
sizeof(A2): 1,  sizeof(B2): 1, sizeof(C2) 1
sizeof(A3): 8,  sizeof(B3): 8, sizeof(C3) 8
============
000000DB0EF1FD40 000000DB0EF1FD48
000000DB0EF1FD50
```

- The C++ standard does not permit objects (or classes) of size 0;
  - This is because that would make it possible for two distinct objects to have the same memory location (please recall pointer arithmetic, e.g. ptr++ increments by the size of the object).

- The size of every class now includes a pointer, 8. (No, not 9.)

The vtbl – slicing revisited

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

class Pet {
public:
    virtual void speak() { cout<< "Animal speaking ..\n"; }
    virtual ~Pet() {}
private:
    int legs;
};
class Cat : public Pet {
public:
    void speak() { cout << "Cat meowing ..\n"; }
private:
    int tails;
};

int main() {
    Cat Felix;

    Pet& aRef = Felix;
    Pet aCopy = Felix;

    aRef.speak();
    aCopy.speak();

}
```



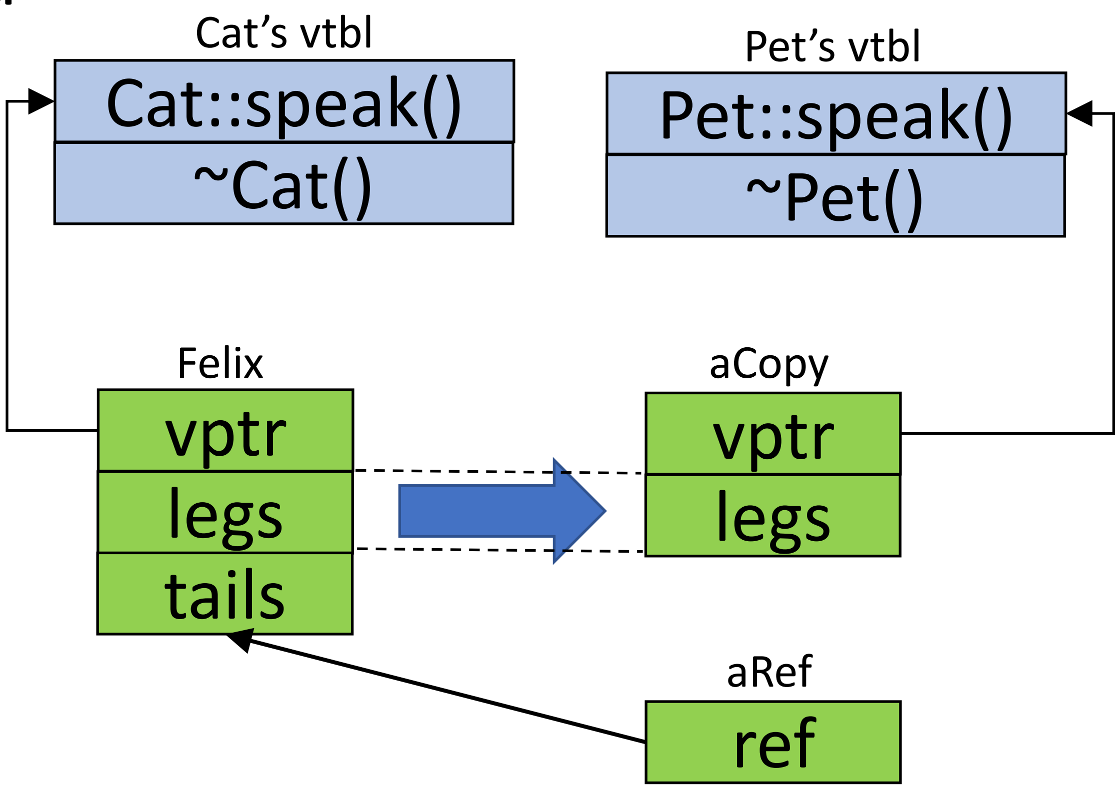
Cat meowing ..
Animal speaking ..

Why are we invoking a method that’s in Pet’s vtbl?

The vtbl – slicing revisited

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

class Pet {
public:
    virtual void speak() { cout<< "Animal speaking ..\n"; }
    virtual ~Pet() {}
private:
    int legs;
};
class Cat : public Pet {
public:
    void speak() { cout << "Cat meowing ..\n"; }
private:
    int tails;
};

int main() {
    Cat Felix;

    Pet& aRef = Felix;
    Pet aCopy = Felix;

    aRef.speak();
    aCopy.speak();

}
```

Cat meowing ..
Animal speaking ..



Why are we invoking a method that's in Pet's vtbl?

Because an object of type **Pet is created** and the **copy constructor for Pet is invoked**:

- It copies only the fields that are known to Pet
- It inherently has vtbl for Pet!

Polymorphic constructors?

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

class Base {
public:
    Base() { foo(); }
    virtual void foo() const { cout << "Base\n"; }
    void display() { this->foo(); }
};

class Derived : public Base {
public:
    Derived(int n) : Base(), x(n) {}
    void foo() const { cout << "Derived: x == " << x << endl; }
private:
    int x;
};

int main() {
    Derived der(17);
    der.display();
}
```

What will the constructor display?

Polymorphic constructors?

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

class Base {
public:
    Base() { foo(); }
    virtual void foo() const { cout << "Base\n"; }
    void display() { this->foo(); }
};

class Derived : public Base {
public:
    Derived(int n) : Base(), x(n) {}
    void foo() const { cout << "Derived: x == " << x << endl; }
private:
    int x;
};

int main() {
    Derived der(17);
    der.display();
}
```

```
Base
Derived: x == 17
```

- At this point, the “Derived” object has not been created yet. The constructor is construcing a “Base” object.
- The vptr points towards Base’s vtbl, NOT Derived’s vtbl
- This actually **prevents disasters**, such as having Derived::foo being called from Base::Base (which accesses x, which has not been created yet!)

Polymorphic constructors?

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

class Base {
public:
    Base() { foo(); }
    virtual void foo() const { cout << "Base\n"; }
    void display() { this->foo(); }
};

class Derived : public Base {
public:
    Derived(int n) : Base(), x(n) {}
    void foo() const { cout << "Derived: x == " << x << endl; }
private:
    int x;
};

int main() {
    Derived der(17);
    der.display();
}
```

```
Base
Derived: x == 17
```

- At this point, the "Derived" object has not been created yet. The constructor is construcing a "Base" object.
- The vptr points towards Base's vtbl, NOT Derived's vtbl
- This actually **prevents disasters**, such as having Derived::foo being called from Base::Base (which accesses x, which has not been created yet!)

- **The runtime system goes out of its way to ensure that calls in a constructor to methods of the class will not be polymorphic.**
  - It uses the Base classes vtable pointer, instead of the current one.

Polymorphic constructors – another ex

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

class A {
public:
    A() {
        cout << "In A(), calls ";
        bar();
    }
    virtual void bar() {
        cout << "A::bar()\n";
    }
    virtual void foo() { cout << "A::foo()\n"; }
};
class B : public A {
public:
    B(){
        cout << "In B(), calls ";
        bar();
    }
    void bar() {
        cout << "B::bar()\n";
    }
    void foo() { cout << "B::foo()\n"; }
};
```

```cpp
class C : public B {
public:
    C() : B() {
        cout << "In C(), calls ";
        bar();
    }
    void bar() {
        cout << "C::bar()\n";
    }
    void foo() { cout << "C::foo()\n"; }
};

int main() {

    cout << "A A;\n";
    A A;
    cout << "============\n";
    cout << "B B;\n";
    B B;
    cout << "============\n";
    cout << "C C;\n";
    C C;
    cout << "============\n";
}
```

Redundant, why?

Polymorphic constructors – another ex

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

class A {
public:
    A() {
        cout << "In A(), calls ";
        bar();
    }
    virtual void bar() {
        cout << "A::bar()\n";
    }
    virtual void foo() { cout << "A::foo()\n"; }
};
class B : public A {
public:
    B() : A() {
        cout << "In B(), calls ";
        bar();
    }
    void bar() {
        cout << "B::bar()\n";
    }
    void foo() { cout << "B::foo()\n"; }
};
```

```
class C : public B {
public:
    C() : B() {
        cout << "In C(), calls ";
        bar();
    }
    void bar() {
        cout << "C::bar()\n";
    }
    void foo() { cout << "C::foo()\n"; }
};

int main() {

    cout << "A A;\n";
    A A;
    cout << "=============\n";
    cout << "B B;\n";
    B B;
    cout << "=============\n";
    cout << "C C;\n";
    C C;
    cout << "=============\n";
}
```

```
A A;
In A(), calls A::bar()
=============
B B;
In A(), calls A::bar()
In B(), calls B::bar()
=============
C C;
In A(), calls A::bar()
In B(), calls B::bar()
In C(), calls C::bar()
=============
```

Polymorphism Review

- When invoking a method, it first has to exist!
    - The compiler finds it based on the **declared type** of the class or any of its ancestors.
        - Remember, every derived class extends its parent, i.e. has everything in parent and a few more.

- It then decides if it's virtual or not:
    - If virtual → invoke using the vptr and vtbl of the actual object
    - If not virtual → invoke using method in declared type.

Polymorphism Review – cont.

How do we determine the correct method to be invoked?

- **Is the method defined** based on the object/pointer's **declared type** or any of its parents?
  - NO → compilation error
  - YES → proceed
- **Is there name hiding?** based on the object/pointer's **declared type**:
  (declaring a method in a derived class with same name (but different parameters) as in base class does not overload but rather **hides** the base class method, be it virtual or not)
  - Yes → Derived methods obscure their Base methods of same name .. may cause compile errors
  - NO → proceed
- **Are we using an object/value or a pointer/ref?**
  - Object → declared type is actual type → use that method
  - Pointer/ref → declared type may not be actual type → proceed
- **Is the method virtual in declared type?**
  (Remember, **once a virtual, always a virtual**)
  - NO → use method of the **declared type**
  - YES → polymorphic call → use virtual table of **actual type**

Resolved at compile-time

Resolved at run-time

Polymorphism Review - ex

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

class SoftDrink {
public:
    virtual void display() const = 0;               // Line A
};

class Soda : public SoftDrink {
public:
    void display() const override { cout << "Soda "; }   // Line B
};

class GingerAle : public Soda {
public:
    void display() { cout << "GingerAle "; }  // Line C
};

int main() {
    GingerAle ga;    // Line D
    Soda* sd = &ga;  // Line E
    sd->display();   // Line F
    ga.display();    // Line G
}
```

O The program will output: GingerAle GingerAle	O Compilation error at Line B
O The program will output: Soda GingerAle	O Compilation error at Line C
O The program will output: Soda Soda	O Compilation error at Line D
O The program will output: GingerAle Soda	O Compilation error at Line E
O The program will compile and not output anything	O Compilation error at Line F
O The program will compile, but will crash when run	O Compilation error at Line G
O Compilation error at Line A	O None of the above

How do we determine the correct method invoked?

Polymorphism Review – ex

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

class SoftDrink {
public:
    virtual void display() const = 0;                // Line A
};

class Soda : public SoftDrink {
public:
    void display() const override { cout << "Soda "; }   // Line B
};

class GingerAle : public Soda {
public:
    void display() { cout << "GingerAle "; }  // Line C
};

int main() {
    GingerAle ga;    // Line D
    Soda* sd = &ga;  // Line E
    sd->display();   // Line F
    ga.display();    // Line G
}
```

O The program will output: GingerAle GingerAle	O Compilation error at Line B
O The program will output: Soda GingerAle	O Compilation error at Line C
O The program will output: Soda Soda	O Compilation error at Line D
O The program will output: GingerAle Soda	O Compilation error at Line E
O The program will compile and not output anything	O Compilation error at Line F
O The program will compile, but will crash when run	O Compilation error at Line G
O Compilation error at Line A	O None of the above

Soda GingerAle

How do we determine the correct method invoked?