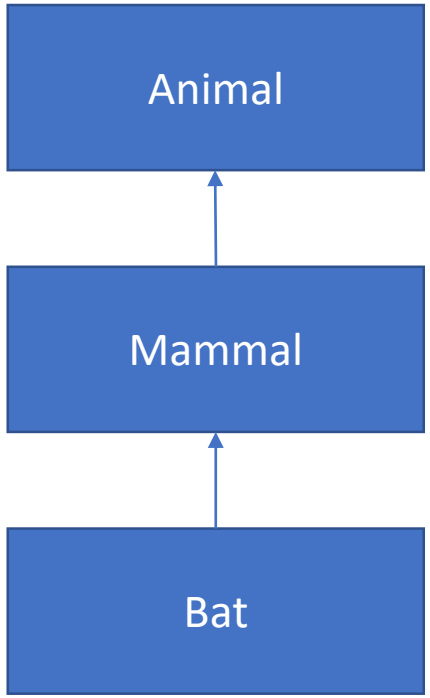
TDDE18 & 726G77

Multilevel and Multiple inheritance

Different kind of inheritance – Multilevel

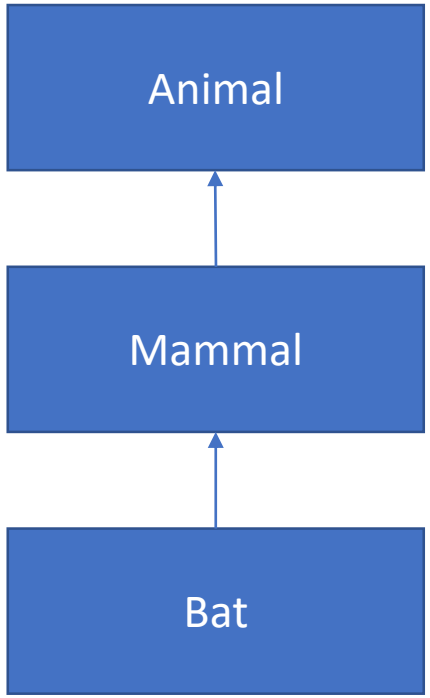
- In C++ you can derive a class from a base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

```
class Animal {
    ...
};
class Mammal : public Animal {
    ...
};
class Bat : public Mammal {
    ...
};
```

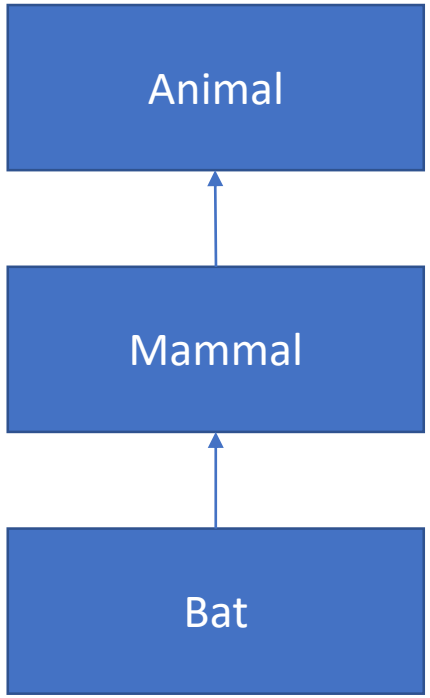


Different kind of inheritance – Multilevel

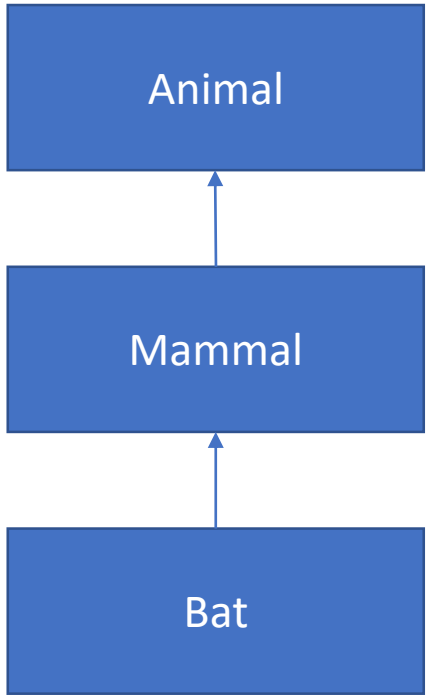
```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {};
class Bat : public Mammal {};

int main() {
    Bat bat{};
    bat.move();     // Animal move.
}
```



```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {
public:
    void move() {
        cout << “Mammal move.” << endl;
    }
};
class Bat : public Mammal {};

int main() {
    Bat bat{};
    bat.move();    // Mammal move.
}
```

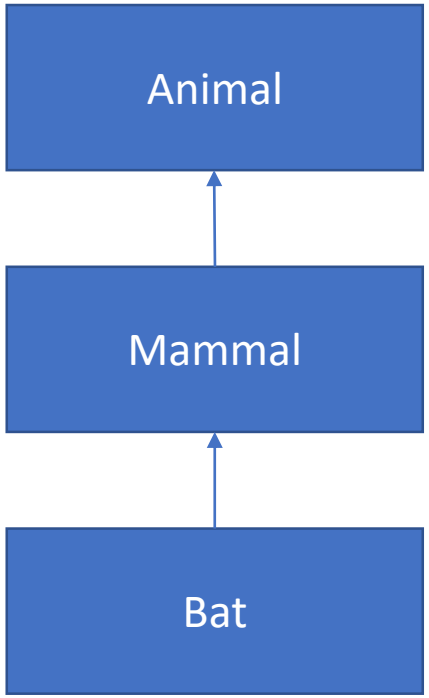


```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {
public:
    void move() {
        cout << “Mammal move.” << endl;
    }
};
class Bat : public Mammal {
public:
    void move() {
        cout << “Animal move” << endl;
    }
};

int main() {
    Bat bat{};
    bat.move();    // Mammal move.
}
```

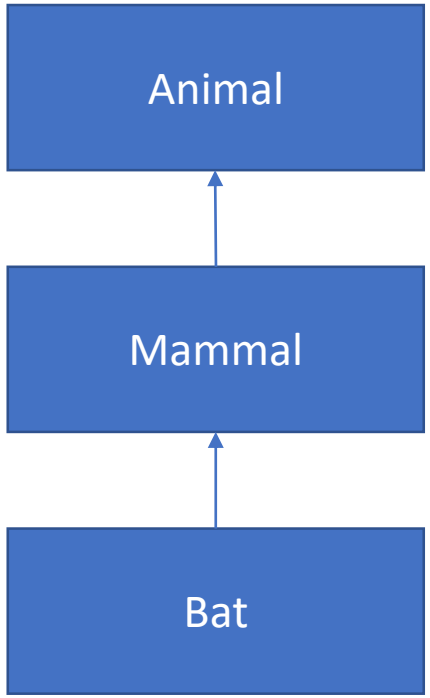


```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {
public:
    void move() {
        cout << “Mammal move.” << endl;
    }
};
class Bat : public Mammal {
public:
    void move() {
        cout << “Animal move” << endl;
    }
};

int main() {
    Bat bat{};
    bat.move();    // Mammal move.
}
```

Missing dot

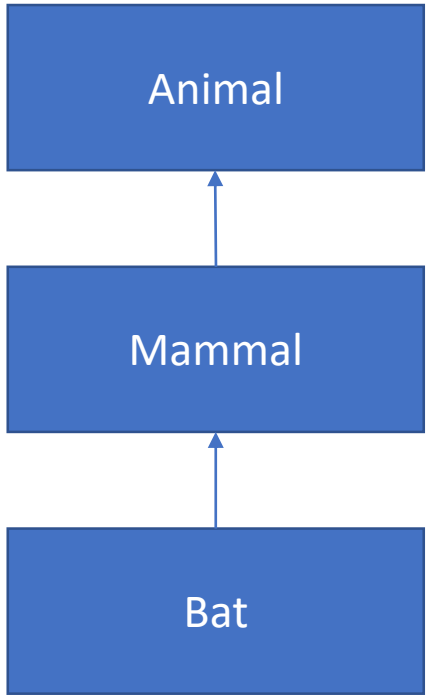


```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {
public:
    void move() {
        cout << “Mammal move.” << endl;
    }
};
class Bat : public Mammal {
public:
    using Animal::move;
};

int main() {
    Bat bat{};
    bat.move();    // Animal move.
}
```



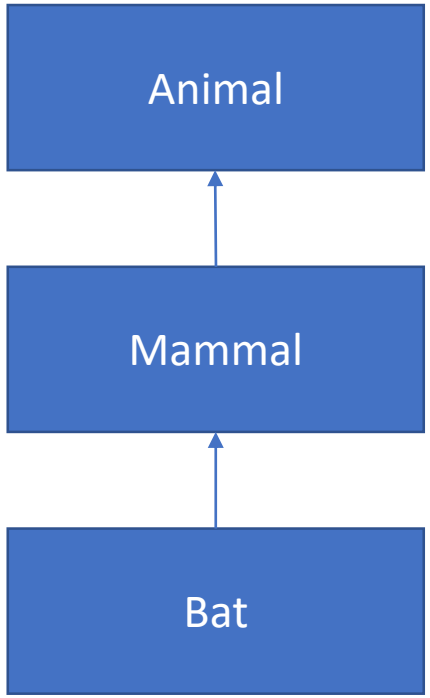
Calling base class function

```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {};
class Bat : public Mammal {
public:
    void move() {
        // Call Animal’s move function
        // Call Mammal’s move function
        // Do own stuff
    }
};
```



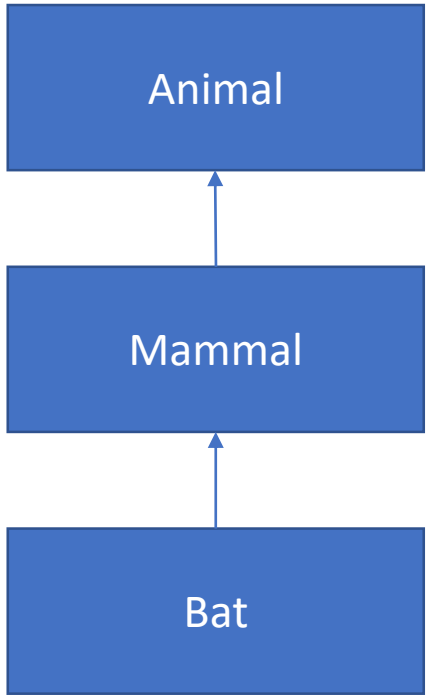
Calling base class function

```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {};
class Bat : public Mammal {
public:
    void move() {
        Animal::move();
        // Call Mammal’s move function
        // Do own stuff
    }
};
```



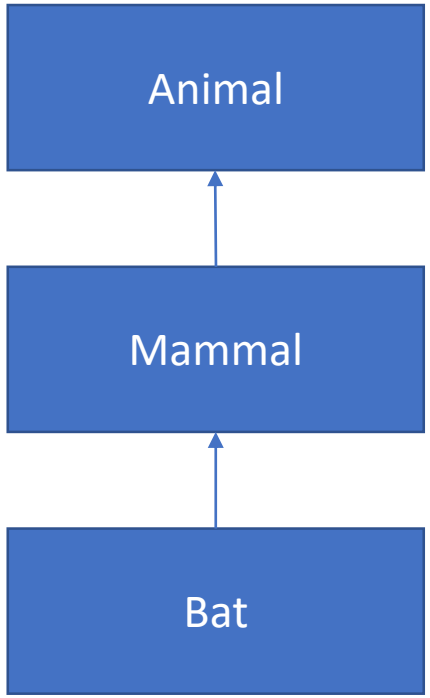
Calling base class function

```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {};
class Bat : public Mammal {
public:
    void move() {
        Animal::move();
        Mammal::move();
        // Do own stuff
    }
};
```



Calling base class function

```
class Animal {
public:
    void move() {
        cout << “Animal move.” << endl;
    }
};
class Mammal : public Animal {};
class Bat : public Mammal {
public:
    void move() {
        Animal::move();
        Mammal::move();
        cout << “Bat move.” << endl;
    }
};
```



final specifier (1)

- When used in a virtual function declaration or definition, final ensures that the function is virtual and specifies that it may not be overridden by derived classes.
- When used in a class definition, final specifies that this class may not act as a base class to another class (in other words, this class cannot be derived from).
- *final* is an identifier with special meaning when used in a member function declaration or class head.

final specifier (2)

```
class Base {
    virtual void foo();
};

class A : public Base {
    void bar() final; // Error: non-virtual function cannot be final
};
```

final specifier (3)

```
class Base {
    virtual void foo();
};

class A : public Base {
    void foo() final; // A::foo is overridden and it is the final override
};

class B : public A {
    void foo() override; // Error: it's final in A
};
```

final specifier (4)

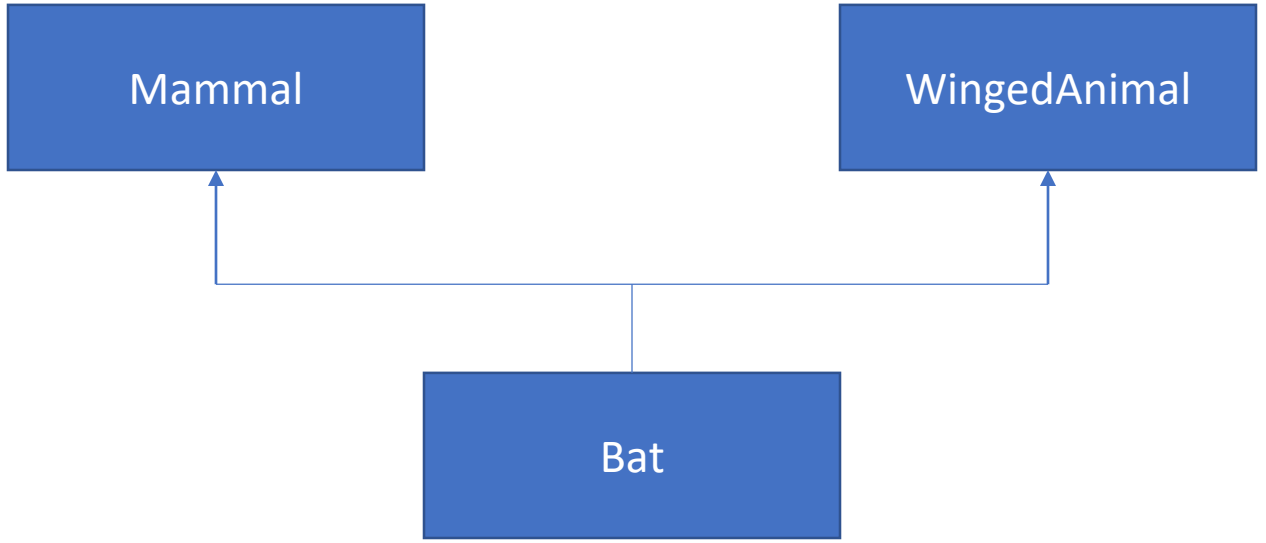
```
class Base {
    virtual void foo();
};

class A final : public Base {
};

class B : public A { // Error: A is final
};
```

final specifier (5) – why?

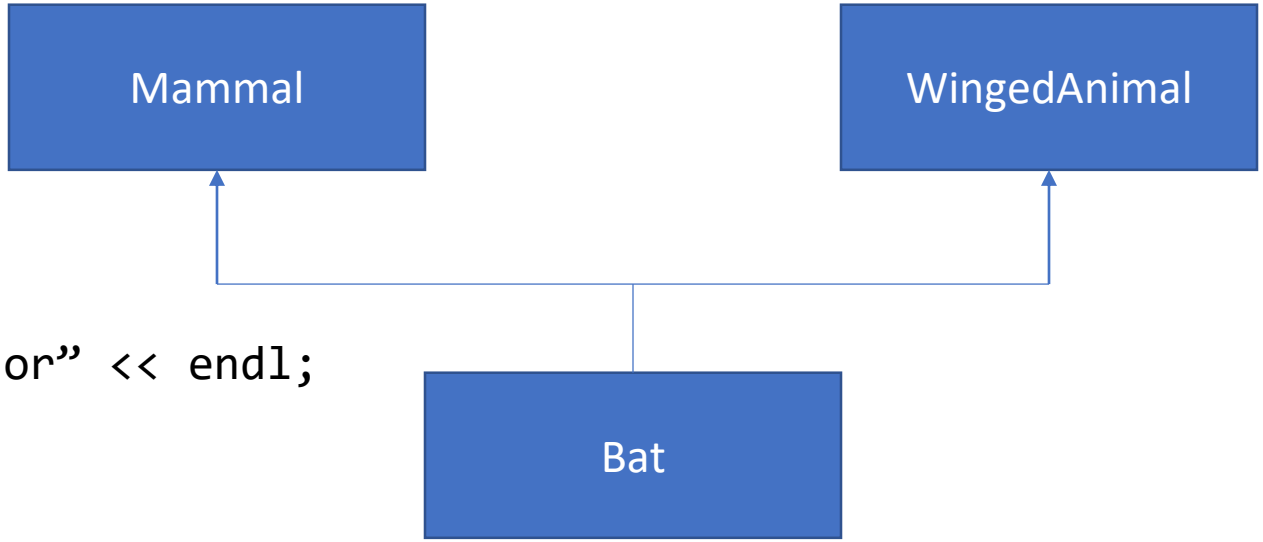
- For efficiency: to avoid your function calls being virtual
- For safety: to ensure that your class is not used as a base class (for example, cryptography)

Multiple inheritance

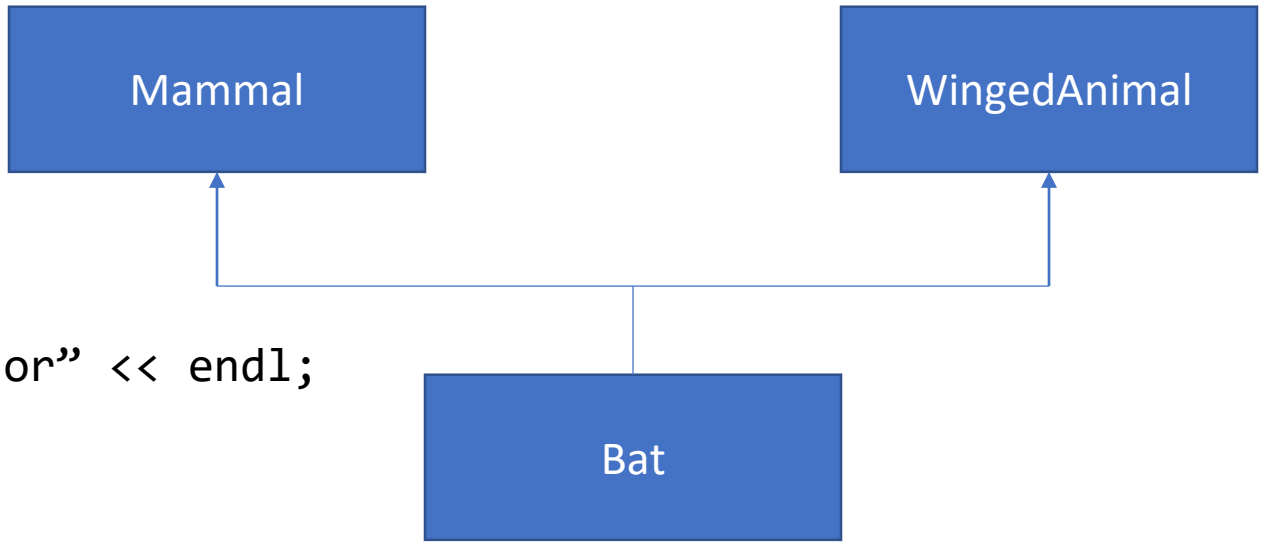
- Deriving direct from more than one class is usually called multiple inheritance.



```
class Mammal {
public:
    Mammal() {
        cout << “Mammal’s constructor” << endl;
    }
};
class WingedAnimal {
public:
    WingedAnimal() {
        cout << “WingedAnimal’s constructor” << endl;
    }
};
class Bat : public Mammal, public WingedAnimal {};

int main() {
     Bat b{};        // Mammal’s constructor
};                   // WingedAnimal’s constructor
```



```
class Mammal {
public:
    Mammal() {
        cout << “Mammal’s constructor” << endl;
    }
};
class WingedAnimal {
public:
    WingedAnimal() {
        cout << “WingedAnimal’s constructor” << endl;
    }
};
class Bat : public WingedAnimal, public Mammal {};

int main() {
     Bat b{};        // WingedAnimal’s constructor
};                   // Mammal’s constructor
```



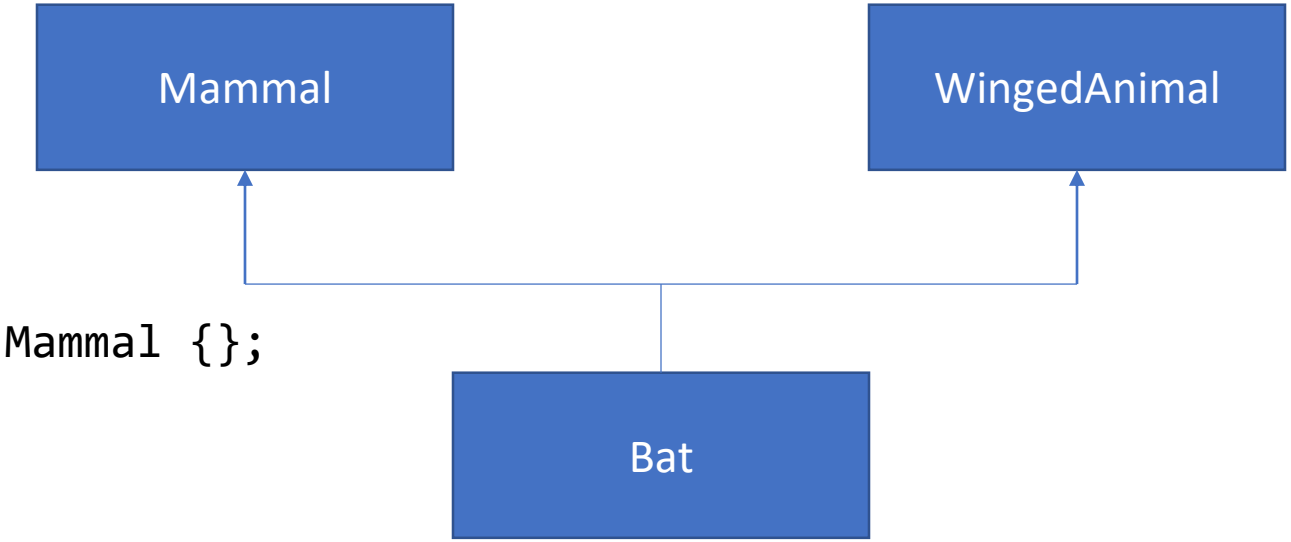
Multiple inheritance - ambiguity

- The most difficult to avoid complication that arises when using multiple inheritance is that sometimes the programmers interested in using this technique to extend the existing code are forced to learn some of the implementation’s details.
- Another problem that might appear when using this technique is the creation of ambiguities:

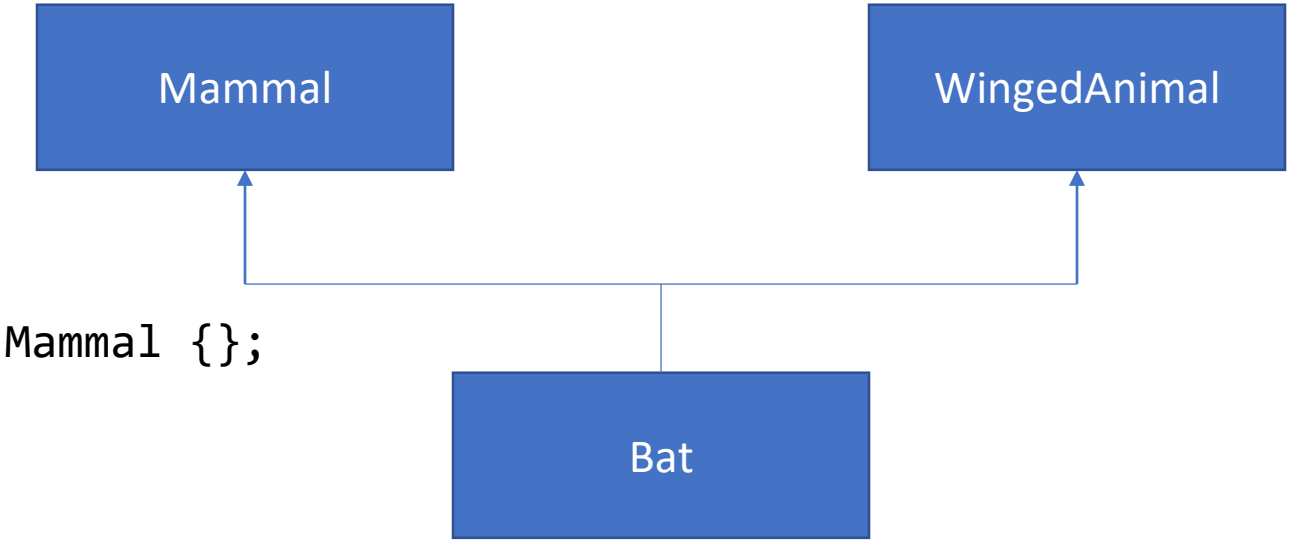
Multiple inheritance - ambiguity

- The most obvious problem with multiple inheritance occurs during function overriding.
- If you try to call the function using the object of the derived class, compiler shows error. It’s because the compiler doesn’t know which function to call.

```
class Mammal {
public:
    void move() {}
};
class WingedAnimal {
public:
    void move() {}
};
class Bat : public WingedAnimal, public Mammal {};

int main() {
     Bat b{};
     b.move();     // Error! Which one?
};
```



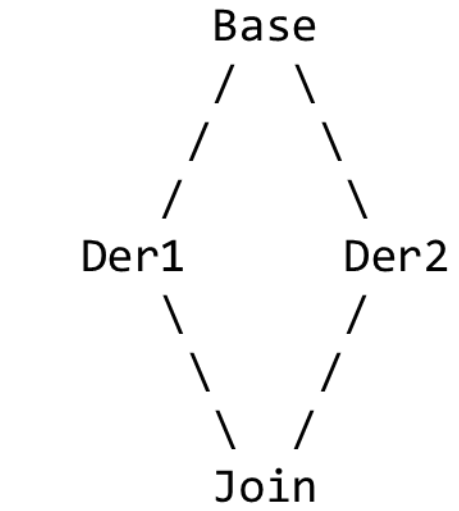
```
class Mammal {
public:
    void move() {}
};
class WingedAnimal {
public:
    void move() {}
};
class Bat : public WingedAnimal, public Mammal {};

int main() {
     Bat b{};
     b.Mammal::move();
};
```



Solved by using scope resolution function

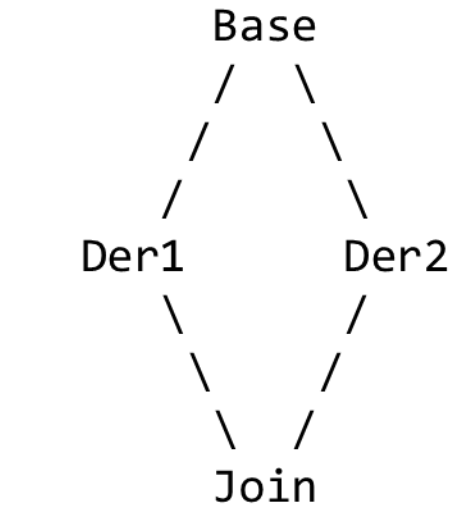
Dreaded diamond

The “dreaded diamond” refers to a class structure in which a particular class appears more than once in a class’s inheritance hierarchy.



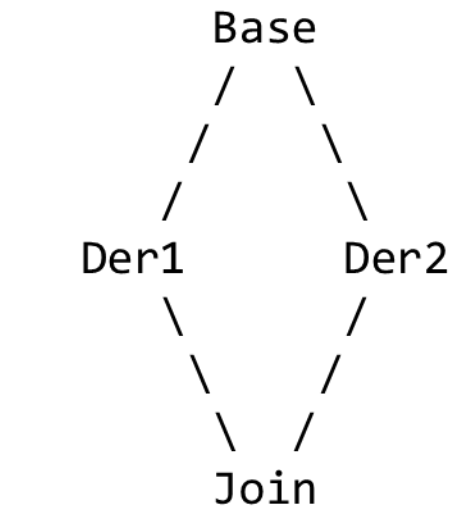
Dreaded diamond

```
class Base {
protected:
    int data;
};
class Der1 : public Base {};
class Der2 : public Base {};
class Join : public Der1, public Der2 {
    void foo() {
        data = 1; // Error: this is ambiguous
    }
};
```



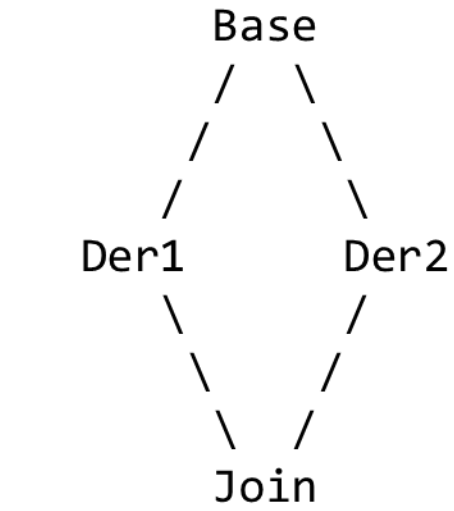
Dreaded diamond

```
class Base {};
class Der1 : public Base {};
class Der2 : public Base {};
class Join : public Der1, public Der2 {};

int main() {
    Base * b{new Join{}};
}
```



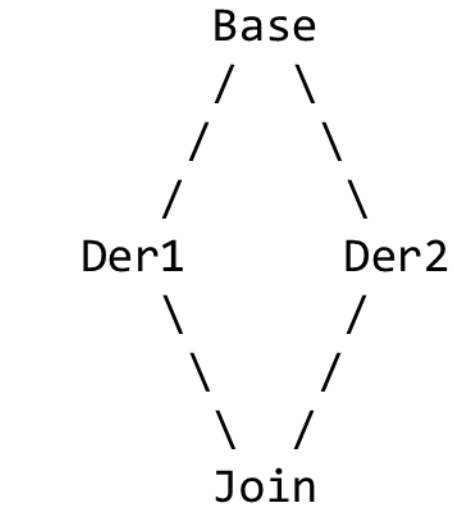
Dreaded diamond – bad solution

```
class Base {
protected:
    int data;
};
class Der1 : public Base {};
class Der2 : public Base {};
class Join : public Der1, public Der2 {
    void foo() {
        Der1::data = 1;
    }
};
```



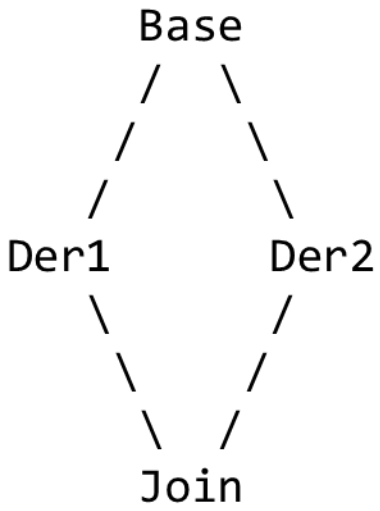
Dreaded diamond – bad solution

```
class Base {};
class Der1 : public Base {};
class Der2 : public Base {};
class Join : public Der1, public Der2 {};

int main() {
    Der1 * d{new Join{}};
    Base * b{d};
}
```



Dreaded diamond – virtual keyword

```
class Base {
    int data;
};
class Der1 : public virtual Base {};
class Der2 : public virtual Base {};
class Join : public Der1, public Der2 {
    void foo() {
        data = 1;
    }
};

int main() {
    Base * b{new Join{}};
}
```



interface

- An interface is an abstract type that is used to specify behavior that concrete classes must implement.
- Interfaces are used to encode similarities which the classes of various types share, but do not necessarily constitute a class relationship.
- Give the ability to use an object without knowing its type of class, but rather only that it implements a certain interface.
- Used a lot in programming language like Java and C#

interface

Below are the nature of *interface* and its C++ equivalents:

- interface can contain only body-less abstract methods; C++ equivalent is pure virtual functions.
- interface can contain only static final data members; C++ equivalent is static const data members which are compile time constants.
- Multiple interface can be implemented by a Java class, this facility is needed because a Java class can inherit only 1 class; C++ supports multiple inheritance straight away with help of virtual keyword when needed.

interface

```
class IList {
    void insert(int number) = 0;
    void remove(int index) = 0;
    static const string name{“List interface”};
};
```

Dynamic type control using typeid

- One way to find out the type of an object is to use *typeid*

```
if (typeid(*p) == typeid(Bat)) ...
```

- A *typeid* expression returns a *type_info* object (a class type)
- type checking is done by comparing two type_info objects

typeid expressions

```
typeid(*p) // p is a pointer to an object of some type

typeid(r) // r is a reference to an object of some type

typeid(T) // T is a type

typeid(p) // is usually a mistake if p is a pointer
```

typeinfo operations

== check if two type_info objects are equal

typeid(*p) == typeinfo(T)

!= check if two type_info objects are not equal

typeid(*p) != typeinfo(T)

name() returns the type name as a string – may be an internal name used by the compiler, a “mangled name”

TDDE18 & 726G77

Vector

Vector

- Vector are sequence containers.
- Vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets.
- Vector can change size and capacity, in contrast to array which size is fixed.
- Very efficient in accessing its elements and relatively efficient adding or removing elements from its end.

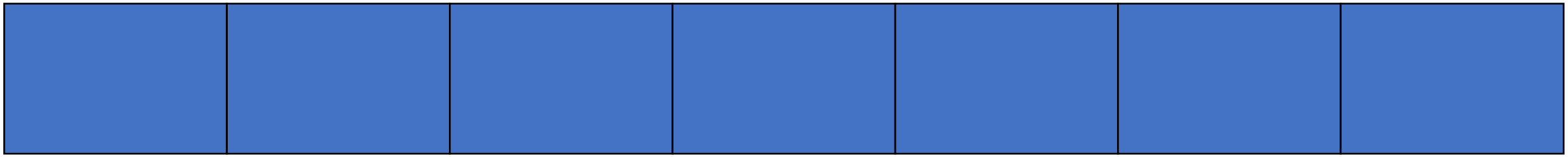
Visualizing Vectors

vector<T> v{7};

Visualizing Vectors

```
vector<T> v{7};
```

Name

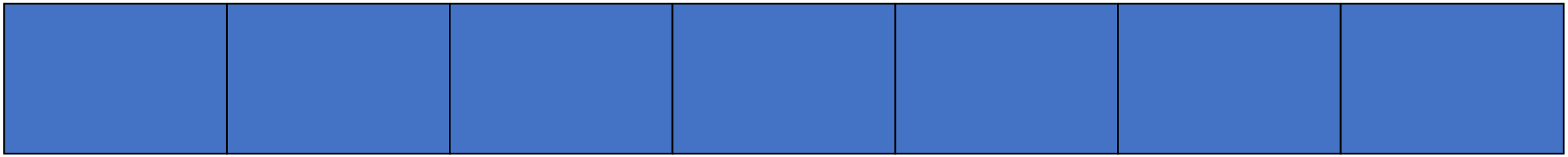


Visualizing Vectors

vector<T> v{7};

Visualizing Vectors

vector<T> v{7};

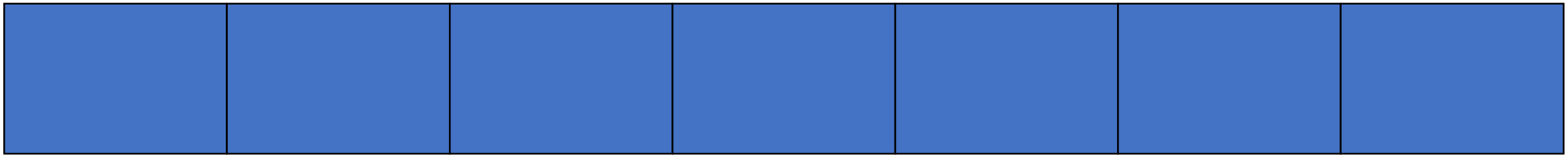


Visualizing Vectors

vector<T> v{7};

Visualizing Vectors

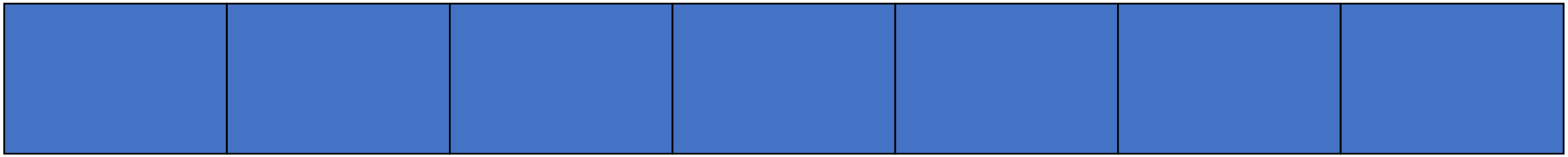
vector<T> v{7};



- Vectors are 0 indexed

Visualizing Vectors

```
vector<double> v{7};
```



- Every element in this vector is of type double
- The size of this vector are 7
- Constructing vectors with a given size will default initialize the elements

Vector member functions

```
vector<double> v{7};
v[0] = 1;
v.at(1) = 2;
v.front();                  // 1
v.back();                   // 0
v.push_back(5);
v.back();                   // 5
v.size();                   // 8
v.pop_back();               // remove the 5
```

auto

- When declaring variables in block scope, in initialization statements of for loops, etc., the keyword auto may be used as the type specifier.
- The compiler determines the type that will replace the keyword auto.
- auto may be accompanied by modifiers, such as const or &, which will participate in the type deduction.

```
auto i{5};                  // i will be of type int
auto i{5.0};                // i will be of type double
auto b_ptr{new Bat{}};      // b_ptr will be of type pointer to bat
```

Using the vector

```
vector<int> v;
...
for (int i{0}; i < v.size(); i++) {
    // do something with v.at(i)
}
```

Using the vector

```
vector<int> v;
...
for (auto i{0}; i < v.size(); i++) {
    // do something with v.at(i)
}
```

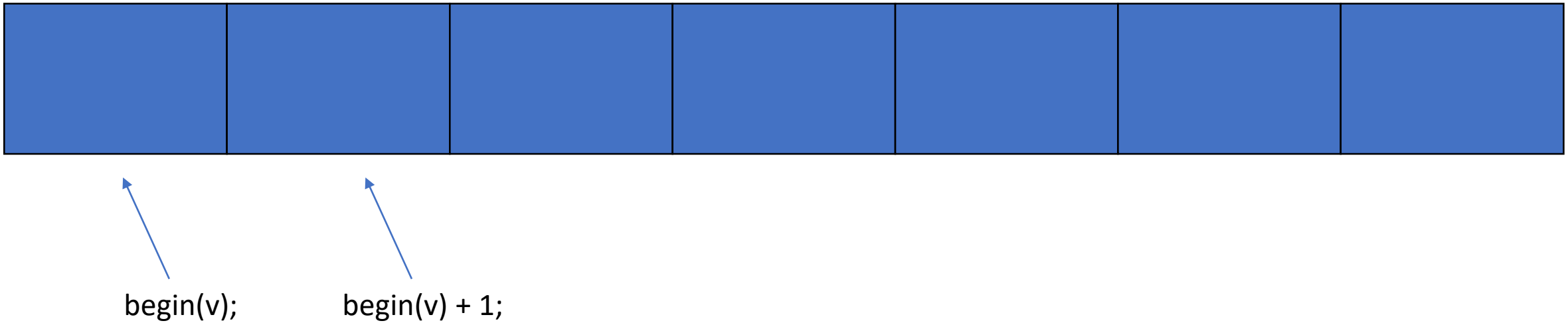
Using the vector

```
vector<int> v;
...
for (auto it{begin(v)}; it != end(v); it++) {
    // do something with *it
    // (it is almost the same thing as pointer)
}
```

Vector – indexing with begin(v)

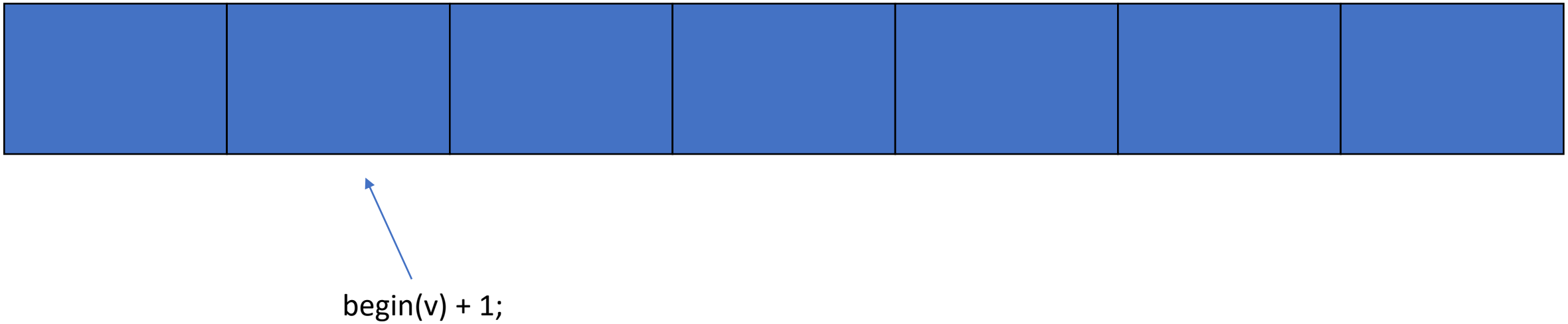
`begin(v)` returns a pointer to the element at index 0

`begin(v) + 1` returns a pointer to the element at index 1



Vector – insert

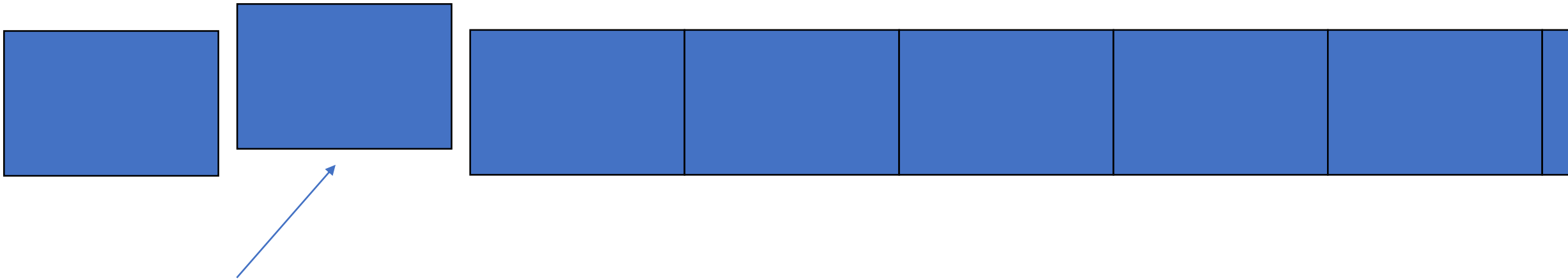
Vector's insert takes an iterator (think pointer for now) as a first argument. This argument tells the function where to insert the new element.




```
v.insert(begin(v) + 1, 3);
```

Vector – insert

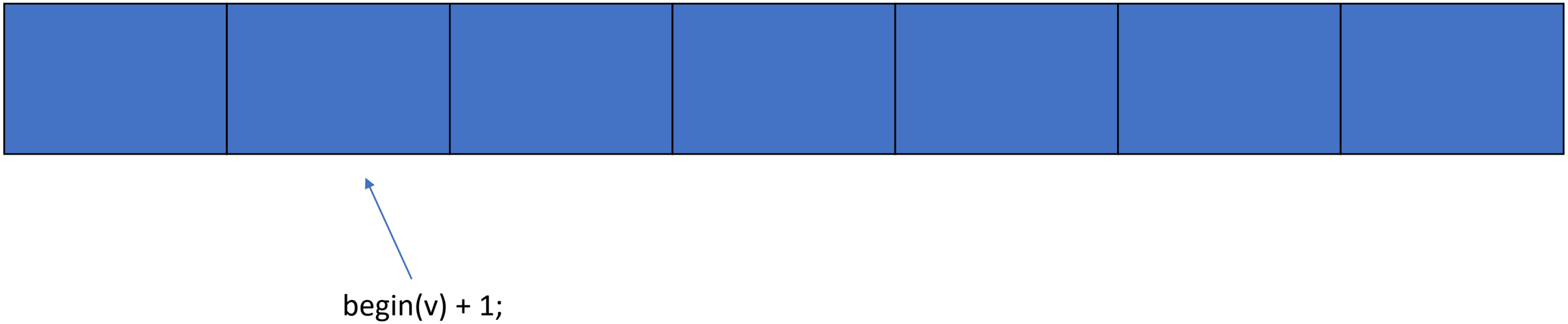
When using insert, everything will be moved one index up




```
v.insert(begin(v) + 1, 3);
```

Vector – erase

Vector's erase takes an iterator as a argument. This argument tells the function where to erase in the vector.




```
v.erase(begin(v) + 1);
```

Vector – erase

When using insert, everything will be moved one index down


```
v.erase(begin(v) + 1);
```

auto

- auto can also be used in a function declaration to indicate that the return type will be deduced from the operand of its return statement.

```
auto foo() {     // auto will be deduced to int
    return 1;
}

auto foo() {     // auto will be deduced to double
    return 1.5;
}

auto foo() {      // auto will be deduced to vector<int>
    return vector<int>{5};
}
```