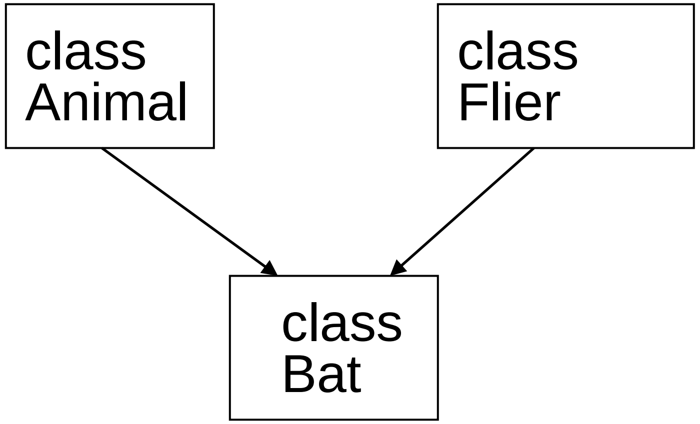
Multiple Inheritance

- A derived class can have more than one base class
- Different usages:
  - Inheriting multiple abstract classes (**interfaces**) → c++ answer to Java's interfaces is multiple inheritance.
  - Needing objects that are of **multiple types**
- Each base class can have its own access specification in derived class's definition:

```
class Bat : public Anmial, public Flier;
```



Multiple Inheritance

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

class Flier {
public:
    virtual void fly() { cout << "I can fly!!!\n"; }
};
class Animal {
public:
    virtual void display() { cout << “I am an Animal\n"; }
};

class Bat : public Animal, public Flier { };

class Insect : public Animal, public Flier {
public:
    void fly() { cout << "Bzzzz. ";  Flier::fly();  }
};

class Plane : public Flier {};

int main() {
    Bat battie;
    battie.display();
    battie.fly();
    Plane aPlane;
    Insect anInsect;

    cout << "==========\n";
    vector<Flier*> vf;
    vf.push_back(&battie);
    vf.push_back(&aPlane);
    vf.push_back(&anInsect);
    for (Flier* flier : vf) {
        flier->fly();
    }

}
```

```
I am an Animal
I can fly!!!
==========
I can fly!!!
I can fly!!!
Bzzzz. I can fly!!!
```

- What if I want to add a display() method to Flier that just says “I am a Flier”?

Multiple Inheritance

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

class Flier {
public:
    virtual void fly() { cout << "I can fly!!!\n"; }
    virtual void display(){ cout << "I am a Flier\n"; }
};
class Animal {
public:
    virtual void display() { cout << "I am an Animal\n"; }
};

class Bat : public Animal, public Flier {
public:

};
class Insect : public Animal, public Flier {
public:
    void fly() { cout << "Bzzzz. ";  Flier::fly();  }
};
class Plane : public Flier {};

int main() {
    Bat battie;
    Plane aPlane;
    Insect anInsect;

    battie.display();
    aPlane.display();
    anInsect.display();

}
```

```
Build started...
1>------ Build started: Project: Lect_01_B, Configuration: Release x64 ------
1>10.mi_om.cpp
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\10.mi_om
.cpp(33,19): error C2385: ambiguous access of 'display'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\10.mi_om
.cpp(33,19): message : could be the 'display' in base 'Animal'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\10.mi_om
.cpp(33,19): message : or could be the 'display' in base 'Flier'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\10.mi_om
.cpp(35,21): error C2385: ambiguous access of 'display'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\10.mi_om
.cpp(35,21): message : could be the 'display' in base 'Animal'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\10.mi_om
.cpp(35,21): message : or could be the 'display' in base 'Flier'
1>Done building project "cs2124.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
ailed, 0 up-to-date, 0 skipped ==========
```

- What if I want to add a display() method to Flier that just says “I am a Flier”?

Multiple Inheritance

- Problem: **Ambiguity** when member variables/functions with the same name.
- Solutions:
  - Derived class redefines the method
  - Invoke the method in a particular base class using scope resolution operator : :
- Compiler errors occur if derived class uses base class function without one of these solutions

Multiple Inheritance

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

class Flier {
public:
    virtual void fly() { cout << "I can fly!!!\n"; }
    virtual void display(){ cout << "I am a Flier\n"; }
};
class Animal {
public:
    virtual void display() { cout << "I am an Animal\n"; }
};

class Bat : public Animal, public Flier {
public:
    void display() { Animal::display(); }
};
class Insect : public Animal, public Flier {
public:
    using Animal::display;
    void fly() { cout << "Bzzzz. ";  Flier::fly();  }
};
class Plane : public Flier {};

int main() {
    Bat battie;
    Plane aPlane;
    Insect anInsect;

    battie.display();
    aPlane.display();
    anInsect.display();

}
```

I am an Animal
I am a Flier
I am an Animal

- What if I want to add a display() method to Flier that just says “I am a Flier”?

Multiple Inheritance

- Arguments can be passed to both base classes' constructors:

- Base class constructors are called in order given in class declaration, not in order used in class constructor

Function Templates

- <u>Function template</u>: a pattern for a function that can work with many data types
- When written, parameters (and function returns) are generic data types
- When called, compiler generates code for specific data types in function call

Function Template Example

```
template <typename T>

T times10(T num)
{
        return 10 * num;
}
```

template prefix

generic data type

type parameter

What gets generated when `times10` is called with an `int`:	What gets generated when `times10` is called with a `double`:

`int times10(int num)`
`{`
`return 10 * num;`
`}`
	`double times10(double num)`
	`{`
	`return 10 * num;`
	`}`

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

- Call a template function by specifying the type inside the angled brackets:

```
int ival = 3;
double dval = 2.55;
cout << times10<int>(ival); // displays 30
cout << times10<double>(dval); // displays 25.5
```

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

- Call a template function in the usual manner if the input parameter(s) **unambiguously** specifies the type:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```

Function Template Notes

- Can define a template to use multiple data types:

```
template<class T1, class T2>
```

- Example:

```
template<class T1, class T2>      // T1 and T2 will be
double mpg(T1 miles, T2 gallons) // replaced in the
{                                 // called function
 return miles / gallons;         // with the data
}                                 // types of the
                                  // arguments
```

Function Template Notes

- Function templates can be overloaded Each template must have a unique parameter list.
- If a function only has generic parameter types (e.g. T) then overloading the function requires the use of unique number of parameters.

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumAll(T1 num1, T2 num2) ...
```

Function Template Notes

- All data types specified in template prefix must be used in template definition
- Like regular functions, function templates must be defined before being called

Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory (code generated only if a function call exists)
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition

Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

Class Templates

- Classes can also be represented by templates. When a class object is created, type information is supplied to define the type of data members of the class.
- Classes are instantiated by supplying the type name (`int`, `double`, `string`, etc.) at object definition

Class Template Example

```
class grade
{
    private:
          double score;
     public:
          grade(double);
          void setGrade(double);
          double getGrade()
};
```

```
template <class T>
class grade
{
    private:
          T score;
     public:
          grade(T);
          void setGrade(T);
          T getGrade()
};
```

Class Template Example

- Pass type information to class template when defining objects:

```
grade<int> testList[20];
grade<double> quizList[20];
```

- Use as ordinary objects once defined

Class Templates and Inheritance

- Class templates can inherit from other class templates:

```
template <class T>
class Rectangle
  { ... };
template <class T>
class Square : public Rectangle<T>
  { ... };
```

- Must use type parameter `T` everywhere base class name is used in derived class

Generic Vector

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


class Vector {
public:
    class Iterator {

        // Not needed, but we usually implement != in terms of ==
        friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
            return lhs.ptr == rhs.ptr;
        }
    public:
        // Used by begin / end
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        // pre-increment. This is what the ranged for needs
        // Could certainly also implement post-increment
        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        // dereference operator. Allows modification of the Vector
        // but not of the Iterator (that's what the const is there to say).
        int& operator*() const { return *ptr; }
    private:
        int* ptr;
    };

    class Const_Iterator {
        // Not needed, but we usually implement != in terms of ==
        friend bool operator==(const Const_Iterator& lhs, const
Const_Iterator& rhs) {
            return lhs.ptr == rhs.ptr;
        }
    public:
        // Used mostly by begin() const / end() const
        Const_Iterator(int* ptr = nullptr) : ptr(ptr) {}

        // pre-increment. This is what the ranged for needs
        // Could implement post-increment also
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int operator*() const { return *ptr; }
    private:
        // Not mandatory that this be a pointer to const but it does
        // help say what we mean.
        const int* ptr;
    };
```

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

template <class T>
class Vector {
public:
    class Iterator {

        // Not needed, but we usually implement != in terms of ==
        friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
            return lhs.ptr == rhs.ptr;
        }
    public:
        // Used by begin / end
        Iterator(T* ptr = nullptr) : ptr(ptr) {}

        // pre-increment. This is what the ranged for needs
        // Could certainly also implement post-increment
        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        // dereference operator. Allows modification of the Vector
        // but not of the Iterator (that's what the const is there to say).
        T& operator*() const { return *ptr; }
    private:
        T* ptr;
    };

    class Const_Iterator {
        // Not needed, but we usually implement != in terms of ==
        friend bool operator==(const Const_Iterator& lhs, const Const_Iterator&
rhs) {
            return lhs.ptr == rhs.ptr;
        }
    public:
        // Used mostly by begin() const / end() const
        Const_Iterator(T* ptr = nullptr) : ptr(ptr) {}

        // pre-increment. This is what the ranged for needs
        // Could implement post-increment also
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }

        T operator*() const { return *ptr; }
    private:
        // Not mandatory that this be a pointer to const but it does
        // help say what we mean.
        const T* ptr;
    };
```

```cpp
// default constructor
   Vector() : data(nullptr), theSize(0), theCapacity(0) {}

   explicit Vector(size_t howMany, int val = 0)
   {
       theSize = howMany;
       theCapacity = howMany;
       data = new int[howMany];
       for (size_t i = 0; i < theSize; ++i) {
           data[i] = val;
       }
   }
   //
   // Vector Copy Control
   //
   // Destructor
   ~Vector() {
       delete[] data;
   }
   // Copy constructor
   Vector(const Vector& rhs) {
       theSize = rhs.theSize;
       theCapacity = rhs.theCapacity;
       data = new int[theCapacity];
       for (size_t i = 0; i < theSize; ++i) {
           data[i] = rhs.data[i];
       }
   }
   // Assignment operator
   Vector& operator=(const Vector& rhs) {
       if (this != &rhs) {
           // Free up the old (destructor)
           delete[] data;
           // Allocate new
           data = new int[rhs.theCapacity];
           // Copy all the stuff
           theSize = rhs.theSize;
           theCapacity = rhs.theCapacity;
           for (size_t i = 0; i < theSize; ++i) {
               data[i] = rhs.data[i];
           }
       }
       // Return ourselves
       return *this;
   }
```

```cpp
// default constructor
   Vector() : data(nullptr), theSize(0), theCapacity(0) {}

   explicit Vector(size_t howMany, T val = 0)
   {
       theSize = howMany;
       theCapacity = howMany;
       data = new T[howMany];
       for (size_t i = 0; i < theSize; ++i) {
           data[i] = val;
       }
   }
   //
   // Vector Copy Control
   //
   // Destructor
   ~Vector() {
       delete[] data;
   }
   // Copy constructor
   Vector(const Vector& rhs) {
       theSize = rhs.theSize;
       theCapacity = rhs.theCapacity;
       data = new T[theCapacity];
       for (size_t i = 0; i < theSize; ++i) {
           data[i] = rhs.data[i];
       }
   }
   // Assignment operator
   Vector& operator=(const Vector& rhs) {
       if (this != &rhs) {
           // Free up the old (destructor)
           delete[] data;
           // Allocate new
           data = new T[rhs.theCapacity];
           // Copy all the stuff
           theSize = rhs.theSize;
           theCapacity = rhs.theCapacity;
           for (size_t i = 0; i < theSize; ++i) {
               data[i] = rhs.data[i];
           }
       }
       // Return ourselves
       return *this;
   }
```

Vector is used without the angled brackets

```
 void push_back(int val) {
        // Handle if we have run out of space.
        if (theSize == theCapacity) {
            // Do we have a zero coapacity vector?
            // Special case becasue doubling zero wouldn't help.
            if (theCapacity == 0) {
                theCapacity = 1;
                data = new int[theCapacity];
            }
            else {
                // Remember the old array
                int* oldData = data;
                // Allocate a new array with twice the capacity
                theCapacity *= 2;
                data = new int[theCapacity];
                // Copy over the POINTERS. This is NOT a deep copy.
                for (size_t i = 0; i < theSize; ++i) {
                    data[i] = oldData[i];
                }
                // Free up the old arrayl
                delete[] oldData;
            }
        }
        // Now we know there is enoubh space.
        // theSize is already the index for the new entry
        data[theSize] = val;
        // And bump theSize now that we have a new entry added
        ++theSize;
    }

    size_t size() const { return theSize; }

    void pop_back() { --theSize; }

    void clear() { theSize = 0; };

    // Square bracket operators. Note that overloading is based on
the const
   // op[] that does not allow modification
    int operator[](size_t index) const {
        return data[index];
    }
    // op[] that does allow modification
    int& operator[](size_t index) {
        return data[index];
    }
```

```
 void push_back(T val) {
        // Handle if we have run out of space.
        if (theSize == theCapacity) {
            // Do we have a zero coapacity vector?
            // Special case becasue doubling zero wouldn't help.
            if (theCapacity == 0) {
                theCapacity = 1;
                data = new T[theCapacity];
            }
            else {
                // Remember the old array
                T* oldData = data;
                // Allocate a new array with twice the capacity
                theCapacity *= 2;
                data = new T[theCapacity];
                // Copy over the POINTERS. This is NOT a deep copy.
                for (size_t i = 0; i < theSize; ++i) {
                    data[i] = oldData[i];
                }
                // Free up the old arrayl
                delete[] oldData;
            }
        }
        // Now we know there is enoubh space.
        // theSize is already the index for the new entry
        data[theSize] = val;
        // And bump theSize now that we have a new entry added
        ++theSize;
    }

    size_t size() const { return theSize; }

    void pop_back() { --theSize; }

    void clear() { theSize = 0; };

    // Square bracket operators. Note that overloading is based on the
const
   // op[] that does not allow modification
    T operator[](size_t index) const {
        return data[index];
    }
    // op[] that does allow modification
    T& operator[](size_t index) {
        return data[index];
    }
```

```cpp
 // Iterators allow modification of the Vector
    Iterator begin() { return Iterator(data); }
    Iterator end() { return Iterator(data + theSize); }

    // Const_Iterator is used when the Vector is const
    Const_Iterator begin() const { return Const_Iterator(data); }
    Const_Iterator end() const { return Const_Iterator(data +
theSize); }

private:
    int* data;
    size_t theSize;
    size_t theCapacity;
};

// This is what ranged for needs.
bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator&
rhs) {
    return !(lhs == rhs);
}
bool operator!=(const Vector::Const_Iterator& lhs, const
Vector::Const_Iterator& rhs) {
    return !(lhs == rhs);
}

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) os << val << ' ';
    return os;
}

void printVec(const Vector& vec) {
    // Requires const versions of begin and end.
    cout << "printVec: displaying using ranged for:\n";
    for (int val : vec) cout << val << ' ';
    cout << endl;

//    cout << "printVec: displaying using Const_Iterator:\n";
//    for (Vector::Const_Iterator iter = vec.begin(); iter !=
vec.end(); ++iter) {
//        cout << *iter << ' ';
//    }
//    cout << endl;
}
```

```cpp
 // Iterators allow modification of the Vector
    Iterator begin() { return Iterator(data); }
    Iterator end() { return Iterator(data + theSize); }

    // Const_Iterator is used when the Vector is const
    Const_Iterator begin() const { return Const_Iterator(data); }
    Const_Iterator end() const { return Const_Iterator(data + theSize);
}

private:
    T* data;
    size_t theSize;
    size_t theCapacity;
};

// This is what ranged for needs.
// Implemeting it in terms of op==, as is common.
template <class T>
bool operator!=(const T& lhs, const T& rhs) {
    return !(lhs == rhs);
}


template <class T>
ostream& operator<<(ostream& os, const Vector<T>& rhs) {
    for (T val : rhs) os << val << ' ';
    return os;
}

template <typename T>
void printVec(const Vector<T>& vec) {
    // Requires const versions of begin and end.
    cout << "printVec: displaying using ranged for:\n";
    for (T val : vec) cout << val << ' ';
    cout << endl;

//    cout << "printVec: displaying using Const_Iterator:\n";
//    for (typename Vector<T>::Const_Iterator iter = vec.begin(); iter
!= vec.end(); ++iter) {
//        cout << *iter << ' ';
//    }
//    cout << endl;
}
```

you must have the &

you must have "const"

```cpp
int main() {
    Vector v;
    v.push_back(17);
    v.push_back(42);
    v.push_back(6);
    v.push_back(28);

    cout << "Using int* to access Vector v contents:\n";
    for (Vector::Iterator iter = v.begin(); iter != v.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;

    cout << "Using ranged for to access Vector v2 contents:\n";
    for (int val : v) cout << val << ' ';
    cout << endl;

    cout << "Using printVec (uses Const_Iterator) :\n";
    printVec(v);
    cout << endl;

    cout << "Printing the entire vector using operator<< :\n";
    cout << v << endl;
}
```

```cpp
int main() {
    Vector<double> v;
    v.push_back(17.4);
    v.push_back(42);
    v.push_back(6.6);
    v.push_back(28);

    cout << "Using Iterator to access Vector v contents:\n";
    for (Vector<double>::Iterator iter = v.begin(); iter != v.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;

    cout << "Using ranged for to access Vector v2 contents:\n";
    for (double val : v) cout << val << ' ';
    cout << endl;

    cout << "Using printVec (uses Const_Iterator) :\n";
    printVec(v);
    cout << endl;

    cout << "Printing the entire vector using operator<< :\n";
    cout << v << endl;
}
```

```
Using int* to access Vector v contents:
17 42 6 28
Using ranged for to access Vector v2 contents:
17 42 6 28
Using printVec (uses Const_Iterator) :
printVec: displaying using ranged for:
17 42 6 28

Printing the entire vector using operator<< :
17 42 6 28
```

```
Using Iterator to access Vector v contents:
17.4 42 6.6 28
Using ranged for to access Vector v2 contents:
17.4 42 6.6 28
Using printVec (uses Const_Iterator) :
printVec: displaying using ranged for:
17.4 42 6.6 28

Printing the entire vector using operator<< :
17.4 42 6.6 28
```

The C++ Standard Template Library (STL)

A C++ standard library that **uses generic types** (i.e. templated). It provides:

- **Utility library**
  - **Types (e.g. pair)**
  - **Functions (e.g. make_pair(), swap(), etc.)**
- Container classes (e.g. vector, lists, queue, map, set, etc.)
  - All support the begin() and end() methods for a half-open range
- Functional library (e.g. functor, aka function object)
- Algorithms (e.g. find, sort, etc.)

The pair type

Natural to pair data

- (x,y) coordinates
- product description/price
- student/grade
- name/ID

```
std::pair<string, double>
product;
std::pair<int, int>
coord;
std::pair<string, char>
mark;
std::pair<string, string>
employee;
```

```
std::pair<type1, type2>  my_pair;
```

The pair type

```
#include <utility>  // pair, make_pair
#include <iostream>
#include <string>
using namespace std;

pair<int, string> foo() {
    pair<int, string> result(42, "the answer");
    return result;
}

// c++14
auto bar() {
    return make_pair(42, "the answer");
}

int main() {
    pair<int, string> result = foo();
    cout << result.first << ": " << result.second << endl;

    // c++11
    auto result2 = foo();
    cout << result2.first << ": " << result2.second << endl;



}
```

```
42: the answer
42: the answer
```

The use of auto for function return type is a **c++14** feature.

C++ is a statically typed language:

- **auto** May be used to declare an initialized variable, OR
- as a return type

The pair type – structured bindings

```
#include <utility>  // pair, make_pair
#include <iostream>
#include <string>
using namespace std;

pair<int, string> foo() {
    pair<int, string> result(42, "the answer");
    return result;
}

// c++14
auto bar() {
    return make_pair(42, "the answer");
}

int main() {
    pair<int, string> result = foo();
    cout << result.first << ": " << result.second << endl;

    // c++11
    auto result2 = foo();
    cout << result2.first << ": " << result2.second << endl;


    // c++17: structured binding
    auto [a, b] = foo();
    cout << a << ": " << b << endl;

    auto [x, y] = bar();
    cout << x << ": " << y << endl;

}
```

```
42: the answer
42: the answer
42: the answer
42: the answer
```

The use of auto for function return type is a **c++14** feature

Requires C++17

The pair type – cont.

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

//pair<int, int> returnsTwo() {
auto returnsTwo() {
    //     return 6, 28;
    return make_pair(6, 28);
}


int main() {

    pair<int, int> result = returnsTwo();
    cout << result.first << ' ' << result.second << endl;

    auto result2 = returnsTwo();
    cout << result2.first << ' ' << result2.second << endl;

    // structured [un]binding
    auto [x, y] = returnsTwo();
    cout << x << ' ' << y << endl;
}
```

```
6 28
6 28
6 28
```