Classes That Need Other Classes

class Data
{
public:
  Data(int x):m_x(x) {}
int getData() const;
private:
  int m_x;
};
int Data::getData() const
{
  return m_x;
}
void main()
{
  Data data(10);
  General object;
  object.printX(data);
}

Friend Class

class Data
{
  friend class General;  // can be put anywhere in class
public:
  Data(int x):m_x(x) {}
private:
  int m_x;
};
int Data::getData() const
{
  return m_x;
}
void main()
{
  Data data(10);
  General object;
  object.printX(data);
}

Note: friendship is granted, not taken
Friend Function

- "Friendship" can be restricted to a specific function
- Suppose we have another function printIntro() in General but we don’t want to grant it the access to Data::m_x
  ```cpp
  class General {
  public:
    void printX(Data inputObject);
    void printIntro();
  
  void General::printIntro() {
    cout << "Welcome to the General class.";
  }
  
  Grant only General::printX() as a friend
  ```

Implication of Friends

- Granting friend classes in C++ essentially breaks the encapsulation of that class. Basically, the class and its friend classes should be considered as a whole body. When you try to modify any implementation of the class, you need to think of all possible usages in its friend classes.
- The class boundary marks a natural module in C++. However, in some design patterns, several separate class interfaces could capture more precisely the physical operating mechanisms. These classes operate on many common data, have strong coupling, and need to be considered as a whole module.
- Without such kind of physical operating models, do not granting a friend class or a friend function just because it is convenient to write codes or to save the time in designing suitable interfaces and abstractions.

Example: Company Database

- Assume we have a company database program in which a "manager" class needs to access an employee class
  ```cpp
  class Manager {
  public:
    void doJob(Employee *worker);
  
  void Manager::doJob(Employee *worker) {
    if (worker->getSalary()<100000 && worker->getSalary()>40000)
      fireEmployee(worker);
  
  getSalary() is a private member function of Employee.
  void Manager::fireEmployee(Employee *worker) {
    cout << "Employee " << worker->getName() << ", has been terminated."
    delete worker;
  
  getName() is a public member function, so fireEmployee() need not be a friend
  }`
  ```

Example (cont’d)

```cpp
void main() {
  Employee *worker;
  Manager *boss;
  worker = new Employee("Wally", 45000);
  boss = new Manager;
  boss->doJob(worker);
}
```

Output

```
Employee Wally has been terminated.
```
Example: Link List

- Suppose we want to implement a linked list class for storing integers. We can do this by means of two classes, one for the data, the other for the linked list itself.

```cpp
class Data {
friend class LinkedList;
private:
    Data(int value);  // constructor
    int m_value;
    Data *m_next;
};
class LinkedList {
public:
    LinkedList();    // constructor
    ~LinkedList();   // destructor
    void append(int value);
    void display();   // display function
private:
    Data *m_tail, *m_head;
};
```

No public interface is defined in this class.

Data::Data(int value): m_value(value), m_next(0) {
}
LinkedList::LinkedList(): m_head(0), m_tail(0) {
}
void LinkedList::append(int value)
{
    Data *temp = new Data(value);
    if (m_head == 0) {
        m_head = temp;
        m_tail = temp;
    }
    else {
        m_tail->m_next = temp;
        m_tail = temp;
    }
}

When do you need a destructor for Data?

Link List (Cont’d)

- Member functions
  - LinkedList::~LinkedList()
  - void LinkedList::display()

```cpp
void LinkedList::display()
{
    Data *temp;
    for (temp=m_head; temp!=0; temp=temp->m_next)
        cout << temp->m_value << " ";
}
```

Containers and Iterators

- A container class is any class designed to hold a collection of objects. Typical containers are arrays, linked lists, binary trees, stacks, and queues.
- Using polymorphic pointers, a container can hold heterogeneous objects.
- An iterator member function allows the client to step through the container.

```cpp
class Array {
public:
    Array(int arraySize);
    Array();    // default constructor
    void insertElement(int slot, int element);
    void reset();    // Iterator function
    int next();    // Iterator function
private:
    int m_arraySize;  // Array size
    int *m_array;
    int m_iterator;    // Iterator
};
```

Array::Array(int arraySize)
{
    m_arraySize = arraySize;
    m_array = new int[arraySize];
}

void Array::reset()
{
    m_iterator = -1;
}

Output

```
1 2
```

Main

```cpp
void main() {
    LinkedList myLinkedList;
    myLinkedList.append(1);
    myLinkedList.append(2);
    myLinkedList.display();
}
```

```cpp
int main() {
    Array myArray;
    myArray.insertElement(1, 1);
    myArray.insertElement(2, 2);
    myArray.insertElement(3, 3);
    myArray.reset(); // Iterator function
    int m_iterator = 0;
    while (m_iterator < myArray.m_arraySize)
    {
        cout << myArray.m_array[m_iterator] << " ";
        m_iterator++;
    }
    return 0;
}
```
Containers and Iterators (cont’d)

```cpp
int Array::next() {
    m_iterator++;
    if (m_iterator < m_arraySize)
        return m_array[m_iterator];
    cout << "There are no additional elements in the array.\n";
    return 0;
}

void main() {
    Array array(2);
    array.insertElement(0, 6);
    array.insertElement(1, 10);
    cout << array.next() << "\n";
    cout << array.next() << "\n";
    cout << array.next() << "\n";
    array.reset();
    cout << array.next() << "\n";
}
```

Containers and Iterators with Friends

▷ Better implementation with a separate Iterator class declared using a friend function

class Array;

class Iterator {
public:
    Iterator();
    void reset();
    int *next(Array &array);
private:
    int m_iterator;
};

class Array {
friend int *Iterator::next(Array &array);
public:
    Array(int arraySize);
    ~Array();
    void insertElement(int slot, int element);
private:
    int m_arraySize;
    int *m_array;
};

Iterators (Cont’d)

```cpp
int *Iterator::next(Array &array) {
    m_iterator++;
    if (m_iterator < array.m_arraySize)
        return &array.m_array[0];
    return 0;
}

void main() {
    int result = 0;
    Array array(2);
    Iterator iter1, iter2;
    int *i, *j;

    array.insertElement(0, 2);
    array.insertElement(1, 10);
    iter1.reset();
    iter2.reset();
    cout << "2^2 + 3^2 + 2^3 + 3^3 = " << result << endl;
}
```

Inner Class Implementation

class Array {
friend int *Iterator::next(Array &array);
public:
    Array(int arraySize);
    ~Array();
    void insertElement(int slot, int element);
private:
    int m_arraySize;
    int *m_array;
};

do not need to specify Array::

```
class Iterator {
public:
    Iterator();
    void reset();
    int *next(Array &array);
private:
    int m_iterator;
};

Array(int arraySize);
~Array();
void insertElement(int slot, int element);
private:
    int m_arraySize;
    int *m_array;
};
```

public/private specifies whether outside Array can use Iterator class definition or not, but it does not control data access.
Inner Class Implementation

```c++
int *Array::Iterator::next(Array &array) {
    m_iterator++;
    if (m_iterator < array.m_arraySize)
        return &array.m_array[m_iterator];
    return 0;
}

void main() {
    int result = 0;
    Array array(2);
    Array::Iterator iter1, iter2;
    int *i, *j;
    array.insertElement(0, 2);
    array.insertElement(1, 3);
    for(i = iter1.next(array); i != 0; i = iter1.next(array), iter2.reset())
        for(j = iter2.next(array); j != 0; j = iter2.next(array))
            result += pow(j, i);
    cout << "Result = " << result << endl;
}
```

Friends and Inheritance

- Friends of Base class is not automatically friends of derived class (i.e. friendship relationship is not inheritable)

```c++
class Base {
    friend class Friend;
    private:
        int m_baseData;
};

class Derived: public Base {
    // friend class Friend;
    private:
        int m_derivedData;
};

class Friend {
    public:
        void func1(Base &base);
        void func2(Derived &derived);
};

class Derived: public Base {
    // friend class Friend;
    private:
        int m_derivedData;
};

class Friend {
    public:
        void func1(Base &base) {
            cout << base.m_baseData << endl;
        }
        void func2(Derived &derived) {
            cout << derived.m_derivedData << endl;
        }
};

error C2248: 'm_derivedData': cannot access private member declared in class 'Derived'
```