

-
-
-
-
-
-
-
-

Friends



C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

✧ Classes that need other classes

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++
- ❖ Do we really need friends in C++?

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++
- ❖ Do we really need friends in C++?
- ❖ Granting friendship to another class

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++
- ❖ Do we really need friends in C++?
- ❖ Granting friendship to another class
- ❖ Granting friendship to another function

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++
- ❖ Do we really need friends in C++?
- ❖ Granting friendship to another class
- ❖ Granting friendship to another function
- ❖ Company database example

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++
- ❖ Do we really need friends in C++?
- ❖ Granting friendship to another class
- ❖ Granting friendship to another function
- ❖ Company database example
- ❖ Linked list example

Contents

- ❖ Classes that need other classes
- ❖ Friend member functions in C++
- ❖ Do we really need friends in C++?
- ❖ Granting friendship to another class
- ❖ Granting friendship to another function
- ❖ Company database example
- ❖ Linked list example
- ❖ Containers and iterators

Classes That Need Other Classes

```
class Data
{
public:
    Data(int x):m_x(x) {}
    int getData() const;
private:
    int m_x;
};
```

Classes That Need Other Classes

```
class Data
{
public:
    Data(int x):m_x(x) {}
    int getData() const;
private:
    int m_x;          accessor
};

int Data::getData() const
{
    return m_x;
}
```

Classes That Need Other Classes

class **Data**

```
{  
public:  
    Data(int x):m_x(x) {}  
    int getData() const;  
private:  
    int m_x;           accessor  
};  
  
int Data::getData() const  
{  
    return m_x;  
}
```

class **General**

```
{  
public:  
    void printX(Data inputObject);  
};
```

Classes That Need Other Classes

class **Data**

```
{  
public:  
    Data(int x):m_x(x) {}  
    int getData() const;  
private:  
    int m_x;           accessor  
};  
int Data::getData() const  
{  
    return m_x;  
}
```

class **General**

```
{  
public:  
    void printX(Data inputObject);  
};  
void General::printX(Data inputObject)  
{  
    cout << inputObject.getData() << " \n";  
}
```

Classes That Need Other Classes

```
class Data
```

```
{  
public:  
    Data(int x):m_x(x) {}  
    int getData() const;  
private:  
    int m_x;           accessor  
};  
  
int Data::getData() const  
{  
    return m_x;  
}
```

```
class General
```

```
{  
public:  
    void printX(Data inputObject);  
};  
  
void General::printX(Data inputObject)  
{  
    cout << inputObject.getData() << " \n";  
}
```

```
void main()
```

```
{  
    Data data(10);  
    General object;  
    object.printX(data);  
}
```

Classes That Need Other Classes

class **Data**

```
{
public:
    Data(int x):m_x(x) {}
    int getData() const;
private:
    int m_x;           accessor
};
int Data::getData() const
{
    return m_x;
}
```

class **General**

```
{
public:
    void printX(Data inputObject);
};
void General::printX(Data inputObject)
{
    cout << inputObject.getData() << " \n";
}
```

void **main()**

```
{
    Data data(10);
    General object;
    object.printX(data);
}
```

```
Output
10
```

Friend Class

```
class Data
```

```
{
```

```
public:
```

```
    Data(int x):m_x(x) {}
```

```
private:
```

```
    int m_x;
```

```
};
```


Friend Class

```
class Data
```

```
{
```

```
public:
```

```
    Data(int x):m_x(x) {}
```

```
private:
```

```
    int m_x;
```

```
};
```

```
class General
```

```
{
```

```
public:
```

```
    void printX(Data inputObject);
```

```
};
```

Friend Class

```
class Data
```

```
{
```

```
public:
```

```
    Data(int x):m_x(x) {}
```

```
private:
```

```
    int m_x;
```

```
};
```

```
class General
```

```
{
```

```
public:
```

```
    void printX(Data inputObject);
```

```
};
```

```
void General::printX(Data inputObject)
```

```
{
```

```
    cout << inputObject.m_x << " \n";
```

```
}
```

Friend Class

```
class Data
```

```
{
```

```
    friend class General; // can be put anywhere in the class
```

```
public:
```

```
    Data(int x):m_x(x) {}
```

```
private:
```

```
    int m_x;
```

```
};
```

```
class General
```

```
{
```

```
public:
```

```
    void printX(Data inputObject);
```

```
};
```

```
void General::printX(Data inputObject)
```

```
{
```

```
    cout << inputObject.m_x << " \n";
```

```
}
```

Friend Class

```
class Data
```

```
{
```

```
    friend class General; // can be put anywhere in the class
```

```
public:
```

```
    Data(int x):m_x(x) {}
```

```
private:
```

```
    int m_x;
```

```
};
```

```
void main()
```

```
{
```

```
    Data data(10);
```

```
    General object;
```

```
    object.printX(data);
```

```
}
```

```
class General
```

```
{
```

```
public:
```

```
    void printX(Data inputObject);
```

```
};
```

```
void General::printX(Data inputObject)
```

```
{
```

```
    cout << inputObject.m_x << " \n";
```

```
}
```

Friend Class

```
class Data
```

```
{
```

```
    friend class General; // can be put anywhere in the class
```

```
public:
```

```
    Data(int x):m_x(x) {}
```

```
private:
```

```
    int m_x;
```

```
};
```

```
void main()
```

```
{
```

```
    Data data(10);
```

```
    General object;
```

```
    object.printX(data);
```

```
}
```

```
class General
```

```
{
```

```
public:
```

```
    void printX(Data inputObject);
```

```
};
```

```
void General::printX(Data inputObject)
```

```
{
```

```
    cout << inputObject.m_x << " \n";
```

```
}
```

❖ **Note: friendship is granted, not taken**

Friend Member Function

- ✧ “Friendship” can be restricted to a specific function

Friend Member Function

- ❖ “Friendship” can be restricted to a specific function

```
class Data {  
  
    public:  
        Data(int x):m_x(x) {}  
    private:  
        int m_x;  
};
```

Friend Member Function

❖ “Friendship” can be restricted to a specific function

❖ Suppose we have another function `General::printIntro()` but we don't want to grant it the access to `Data::m_x`

```
class General {  
public:  
    void printX(Data inputObject);  
    void printIntro();  
};  
void General::printIntro() {  
    cout << "Welcome to class General."  
}
```

```
class Data {  
  
public:  
    Data(int x):m_x(x) {}  
private:  
    int m_x;  
};
```


Friend Member Function

❖ “Friendship” can be restricted to a specific function

❖ Suppose we have another function `General::printIntro()` but we don't want to grant it the access to `Data::m_x`

❖ Grant only `General::printX()` as a friend member function

```
class General {  
public:  
    void printX(Data inputObject);  
    void printIntro();  
};  
void General::printIntro() {  
    cout << "Welcome to class General."  
}
```

```
class Data {  
  
public:  
    Data(int x):m_x(x) {}  
private:  
    int m_x;  
};
```

Friend Member Function

❖ “Friendship” can be restricted to a specific function

❖ Suppose we have another function `General::printIntro()` but we don't want to grant it the access to `Data::m_x`

❖ Grant only `General::printX()` as a friend member function

```
class General {  
public:  
    void printX(Data inputObject);  
    void printIntro();  
};  
void General::printIntro() {  
    cout << "Welcome to class General."  
}
```

```
class Data {  
    friend void General::printX(Data inputObject);  
public:  
    Data(int x):m_x(x) {}  
private:  
    int m_x;  
};
```

Implication of Friends

- ❖ Granting friend classes in C++ essentially **breaks the encapsulation** of one 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 and friend functions.

Implication of Friends

- ❖ Granting friend classes in C++ essentially **breaks the encapsulation** of one 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 and friend functions.
- ❖ A class is 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.

Implication of Friends

- ❖ Granting friend classes in C++ essentially **breaks the encapsulation** of one 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 and friend functions.
- ❖ A class is 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.
- ❖ If you are not considering such kind of physical operating models, **do NOT grant** 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

Example: Company Database

- ❖ Assume we have a company database program in which a “manager” class needs to access an employee class

```
class Manager {  
public:  
    void doJob(Employee *worker);  
private:  
    void fireEmployee(Employee *worker);  
};
```

Example: Company Database

❖ Assume we have a company database program in which a “manager” class needs to access an employee class

```
class Manager {  
public:  
    void doJob(Employee *worker);  
private:  
    void fireEmployee(Employee *worker);  
};
```

❖ The Employee class makes the name of the employee public but not the salary. Since the manager class needs information of both, the Employee class will grant partial friendship to `Manager::doJob()`

Example: Company Database

- ❖ Assume we have a company database program in which a “manager” class needs to access an employee class

```
class Manager {  
public:  
    void doJob(Employee *worker);  
private:  
    void fireEmployee(Employee *worker);  
};
```

- ❖ The Employee class makes the name of the employee public but not the salary. Since the manager class needs information of both, the Employee class will grant partial friendship to `Manager::doJob()`

```
class Employee {  
    friend void Manager::doJob(Employee *worker);  
public:  
    Employee(const char *name, long salary);  
    ~Employee();  
    char *getName() const;  
private:  
    long getSalary() const;  
    char *m_name;  
    long m_salary;  
};
```

Example (cont'd)

```
void Manager::doJob(Employee *worker) {  
    if (worker->getSalary()<100000 && worker->getSalary()>40000)  
        fireEmployee(worker);  
}
```

Example (cont'd)

```
void Manager::doJob(Employee *worker) {  
    if (worker->getSalary()<100000 && worker->getSalary()>40000)  
        fireEmployee(worker);  
}
```

getSalary() is a private member function of Employee.

Example (cont'd)

```
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.\n";  
    delete worker;  
}
```

Example (cont'd)

```
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.\n";  
    delete worker;  
}
```

getName() is a public member function, so **fireEmployee()** need not be a friend

Example (cont'd)

```
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.\n";  
    delete worker;  
}
```

getName() is a public member function, so **fireEmployee()** need not be a friend

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

Example (cont'd)

```
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.\n";  
    delete worker;  
}
```

getName() is a public member function, so **fireEmployee()** need not be a friend

```
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.

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.

```
class Data {  
    friend class LinkedList;  
private:  
    Data(int value);  
    int m_value;  
    Data *m_next;  
};
```

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.

No public interface defined
for this class



```
class Data {  
    friend class LinkedList;  
private:  
    Data(int value);  
    int m_value;  
    Data *m_next;  
};
```

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.

No public interface defined for this class



```
class Data {  
    friend class LinkedList;  
private:  
    Data(int value);  
    int m_value;  
    Data *m_next;  
};
```

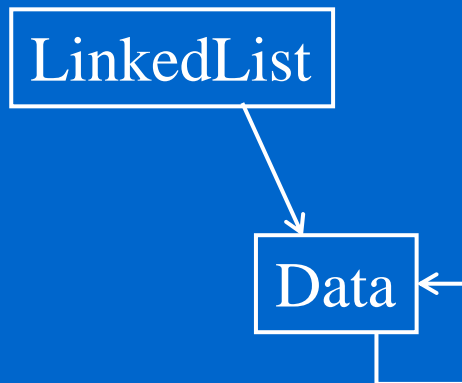
```
class LinkedList {  
public:  
    LinkedList();  
    ~LinkedList();  
    void append(int value);  
    void display();  
private:  
    Data *m_tail, *m_head;  
};
```

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.

No public interface defined for this class

```
class Data {  
    friend class LinkedList;  
private:  
    Data(int value);  
    int m_value;  
    Data *m_next;  
};
```



```
class LinkedList {  
public:  
    LinkedList();  
    ~LinkedList();  
    void append(int value);  
    void display();  
private:  
    Data *m_tail, *m_head;  
};
```

Link List (Cont'd)

✧ Member functions

Link List (Cont'd)

✧ Member functions

```
Data::Data(int value): m_value(value), m_next(0) {  
}
```

Link List (Cont'd)

✧ Member functions

```
Data::Data(int value): m_value(value), m_next(0) {  
}
```

```
LinkedList::LinkedList(): m_head(0), m_tail(0) {  
}
```

Link List (Cont'd)

❖ Member functions

```
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;  
    }  
}
```


Link List (Cont'd)

❖ Member functions

```
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 dtor
for Data?

Link List (Cont'd)

✧ Member functions

Link List (Cont'd)

✧ Member functions

```
LinkedList::~LinkedList() {  
    Data *current, *next;  
    current = m_head;  
    while (current != 0) {  
        next = current->m_next;  
        delete current;  
        current = next;  
    }  
}
```

Link List (Cont'd)

❖ Member functions

```
LinkedList::~LinkedList() {  
    Data *current, *next;  
    current = m_head;  
    while (current != 0) {  
        next = current->m_next;  
        delete current;  
        current = next;  
    }  
}
```

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

Link List (Cont'd)

❖ Member functions

```
LinkedList::~LinkedList() {  
    Data *current, *next;  
    current = m_head;  
    while (current != 0) {  
        next = current->m_next;  
        delete current;  
        current = next;  
    }  
}
```

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

❖ Main

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

Link List (Cont'd)

❖ Member functions

```
LinkedList::~LinkedList() {  
    Data *current, *next;  
    current = m_head;  
    while (current != 0) {  
        next = current->m_next;  
        delete current;  
        current = next;  
    }  
}
```

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

❖ Main

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

Output
1 2

Containers and Iterators

- ✧ A container class is a class designed to hold a collection of objects. Typical containers are arrays, linked lists, trees, stacks, and queues.

Containers and Iterators

- ❖ A container class is a class designed to hold a collection of objects. Typical containers are arrays, linked lists, trees, stacks, and queues.
- ❖ Using polymorphic pointers, a container can hold heterogeneous objects.

Containers and Iterators

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

Containers and Iterators

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

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

Containers and Iterators

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


```
class Array {  
public:  
    Array(int arraySize);  
    ~Array();  
    void insertElement(int slot, int element);  
    void reset(); // Iterator function  
    int next(); // Iterator function  
private:  
    int m_arraySize;  
    int *m_array;  
    int m_iterator;  
};  
  
Array::Array(int arraySize)  
    : m_arraySize(arraySize), m_iterator(-1) {  
    m_array = new int[arraySize];  
}
```

Containers and Iterators (cont'd)

```
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;  
}
```

Containers and Iterators (cont'd)


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



Containers and Iterators (cont'd)

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



```
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"; cout << array.next() << "\n";  
}
```

Containers and Iterators (cont'd)

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

```
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"; cout << array.next() << "\n";  
}
```

Output

6

10

There are no additional elements in the array

6

10

Containers and Friend Iterators

- ✧ Better implementation with a separate Iterator class declared using a friend function. Why? will show in the next slide.

Containers and Friend Iterators

- ❖ Better implementation with a separate Iterator class declared using a friend function. Why? will show in the next slide.

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

Containers and Friend Iterators

- ❖ Better implementation with a separate Iterator class declared using a friend function. Why? will show in the next slide.

```
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;  
};
```

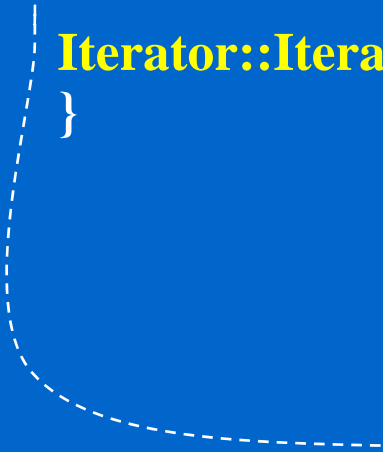
Friend Iterators (Cont'd)

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

Friend Iterators (Cont'd)

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

```
Iterator::Iterator() : m_iterator(-1) {  
}
```



Friend Iterators (Cont'd)

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

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

Friend Iterators (Cont'd)

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

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

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

$$2^2+3^2+2^3+3^3$$

Friend Iterators (Cont'd)

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

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

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

$$2^2+3^2+2^3+3^3$$

Output
Result = 48

Friend Iterators (cont'd)

- ❖ Better implementation with a separate Iterator class declared using a friend function. Each iterator instance is associated with a specific container.

```
class Array;
```


Friend Iterators (cont'd)

- ❖ Better implementation with a separate Iterator class declared using a friend function. Each iterator instance is associated with a specific container.

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

Friend Iterators (cont'd)

- ❖ Better implementation with a separate Iterator class declared using a friend function. Each iterator instance is associated with a specific container.

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

Friend Iterators (cont'd)

- ❖ Better implementation with a separate Iterator class declared using a friend function. Each iterator instance is associated with a specific container.

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

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

Friend Iterators (Cont'd)

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

Friend Iterators (Cont'd)

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

```
Iterator::Iterator(Array &array)  
: m_iterator(-1),  
  m_array(array) {  
}
```

Friend Iterators (Cont'd)

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

```
Iterator::Iterator(Array &array)  
    : m_iterator(-1),  
      m_array(array) {  
}  
  
void Iterator::reset() {  
    m_iterator = -1;  
}
```

Friend Iterators (Cont'd)

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

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

```
Iterator::Iterator(Array &array)  
    : m_iterator(-1),  
      m_array(array) {  
}  
void Iterator::reset() {  
    m_iterator = -1;  
}
```

$$2^2+3^2+2^3+3^3$$

Friend Iterators (Cont'd)

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

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

```
Iterator::Iterator(Array &array)  
    : m_iterator(-1),  
      m_array(array) {  
}  
  
void Iterator::reset() {  
    m_iterator = -1;  
}
```

$$2^2+3^2+2^3+3^3$$

Output
Result = 48

Inner Class Implementation

```
class Array {
```

```
private:
```

```
    int m_arraySize;
```

```
    int *m_array;
```

```
};
```

Inner Class Implementation

```
class Array {
```

```
public:
```

```
    Array(int arraySize);
```

```
    ~Array();
```

```
    void insertElement(int slot, int element);
```

```
private:
```

```
    int m_arraySize;
```

```
    int *m_array;
```

```
};
```

Inner Class Implementation

```
class Array {
```

```
public:
```

```
    class Iterator {
```

```
        private:
```

```
            int m_iterator;
```

```
            Array &m_array;
```

```
        };
```

```
    Array(int arraySize);
```

```
    ~Array();
```

```
    void insertElement(int slot, int element);
```

```
private:
```

```
    int m_arraySize;
```

```
    int *m_array;
```

```
};
```

Inner Class Implementation

```
class Array {
```

```
public:
```

```
class Iterator {
```

Array::Iterator

```
private:
```

```
int m_iterator;
```

```
Array &m_array;
```

```
};
```

```
Array(int arraySize);
```

```
~Array();
```

```
void insertElement(int slot, int element);
```

```
private:
```

```
int m_arraySize;
```

```
int *m_array;
```

```
};
```

Inner Class Implementation

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

Inner Class Implementation

```
class Array {
```

```
public: ←
```

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

public/private specifies whether codes outside Array can use Iterator inner class definition or not.

```
    Array(int arraySize);
```

```
    ~Array();
```

```
    void insertElement(int slot, int element);
```

```
private:
```

```
    int m_arraySize;
```

```
    int *m_array;
```

```
};
```

Inner Class Implementation

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

Array::Iterator

public/private specifies whether codes outside Array can use Iterator inner class definition or not.

Inner Class Implementation

```
class Array {  
    friend int *Iterator::next();
```

do not need to specify Array::

```
public:
```

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

public/private specifies whether codes outside Array can use Iterator inner class definition or not.

```
        Array(int arraySize);  
        ~Array();  
        void insertElement(int slot, int element);
```

```
private:
```

```
    int m_arraySize;  
    int *m_array;
```

```
};
```


Inner Class Implementation

```
class Array {  
    friend int *Iterator::next();
```

do not need to specify Array::

```
public: <
```

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

public/private specifies whether codes outside Array can use Iterator inner class definition or not.

```
Array(int arraySize);  
~Array();  
void insertElement(int slot, int element);
```

```
private:  
    int m_arraySize;  
    int *m_array;
```

```
};
```

Note: An inner class is basically an independent class from its host class, it is not allowed to access the private parts of the host class and vice versa.

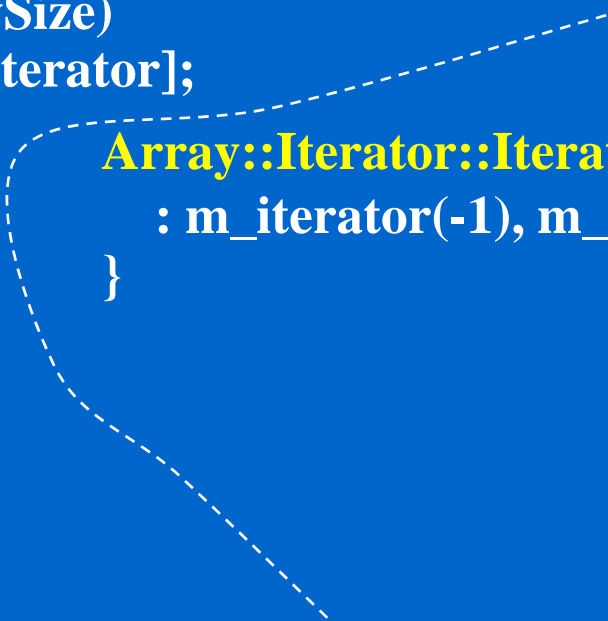
Inner Class Implementation (cont'd)

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

Inner Class Implementation (cont'd)

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

```
Array::Iterator::Iterator(Array &array)  
    : m_iterator(-1), m_array(array) {  
}
```

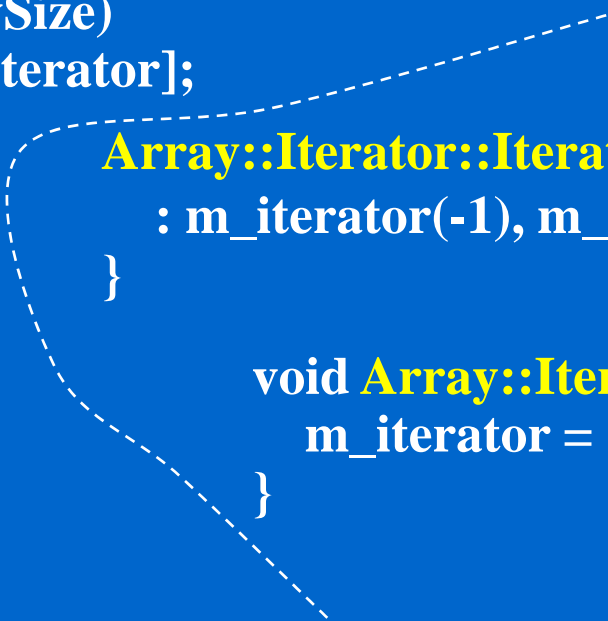


Inner Class Implementation (cont'd)

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

```
Array::Iterator::Iterator(Array &array)  
    : m_iterator(-1), m_array(array) {  
}
```

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



Inner Class Implementation (cont'd)

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

```
Array::Iterator::Iterator(Array &array)  
    : m_iterator(-1), m_array(array) {  
}
```

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

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

$$2^2 + 3^2 + 2^3 + 3^3$$

Inner Class Implementation (cont'd)

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

```
Array::Iterator::Iterator(Array &array)  
    : m_iterator(-1), m_array(array) {  
}
```

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

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

$$2^2 + 3^2 + 2^3 + 3^3$$

Output
Result = 48

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

```
class Base {  
  
private:  
    int m_baseData;  
};
```


Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

```
class Base {  
  
private:  
    int m_baseData;  
};
```

```
class Friend {  
public:  
  
};
```

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

```
class Base {  
  
private:  
    int m_baseData;  
};
```

```
class Friend {  
public:  
    void func1(Base &base);  
};
```

```
void Friend::func1(Base &base) {  
    cout << base.m_baseData << endl;  
}
```

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

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

```
class Friend {  
public:  
    void func1(Base &base);  
};
```

```
void Friend::func1(Base &base) {  
    cout << base.m_baseData << endl;  
}
```

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

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

```
class Friend {  
public:  
    void func1(Base &base);  
};
```

```
class Derived: public Base {  
private:  
    int m_derivedData;  
};
```

```
void Friend::func1(Base &base) {  
    cout << base.m_baseData << endl;  
}
```

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

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

```
class Derived: public Base {  
private:  
    int m_derivedData;  
};
```

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

```
void Friend::func1(Base &base) {  
    cout << base.m_baseData << endl;  
}
```

```
void Friend::func2(Derived &derived) {  
    cout << derived.m_derivedData << endl;  
}
```

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

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

```
class Derived: public Base {  
private:  
    int m_derivedData;  
};
```

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

```
void Friend::func1(Base &base) {  
    cout << base.m_baseData << endl;  
}
```

```
void Friend::func2(Derived &derived) {  
    cout << derived.m_derivedData << endl;  
}
```

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

Friends and Inheritance

- ❖ Friends of **Base** class are not automatically friends of **Derived** class (i.e. **friendship relationship is not inheritable**)

```
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);  
};
```

```
void Friend::func1(Base &base) {  
    cout << base.m_baseData << endl;  
}
```

```
void Friend::func2(Derived &derived) {  
    cout << derived.m_derivedData << endl;  
}
```

Friend and Factory

- ✧ A class can be designed to have a creation rule, such as:

**All new Product must be created indirectly
through a ProductFactory.**

Friend and Factory

❖ A class can be designed to have a creation rule, such as:

**All new Product must be created indirectly
through a ProductFactory.**

★ So that all clients can be unaware of the actual class of the object used, only concentrated on the required abstract interface.

Friend and Factory

❖ A class can be designed to have a creation rule, such as:

**All new Product must be created indirectly
through a ProductFactory.**

- ★ So that all clients can be unaware of the actual class of the object used, only concentrated on the required abstract interface.
- ★ This rule can be enforced by non-public (protected or private) constructors, copy constructors, and assignment operators.

Friend and Factory

❖ A class can be designed to have a creation rule, such as:

All new Product must be created indirectly through a ProductFactory.

- ★ So that all clients can be unaware of the actual class of the object used, only concentrated on the required abstract interface.
- ★ This rule can be enforced by non-public (protected or private) constructors, copy constructors, and assignment operators.
- ★ With friend class or friend function, only the **ProductFactory** class or the specific factory method can access the constructor of the class **Product**.