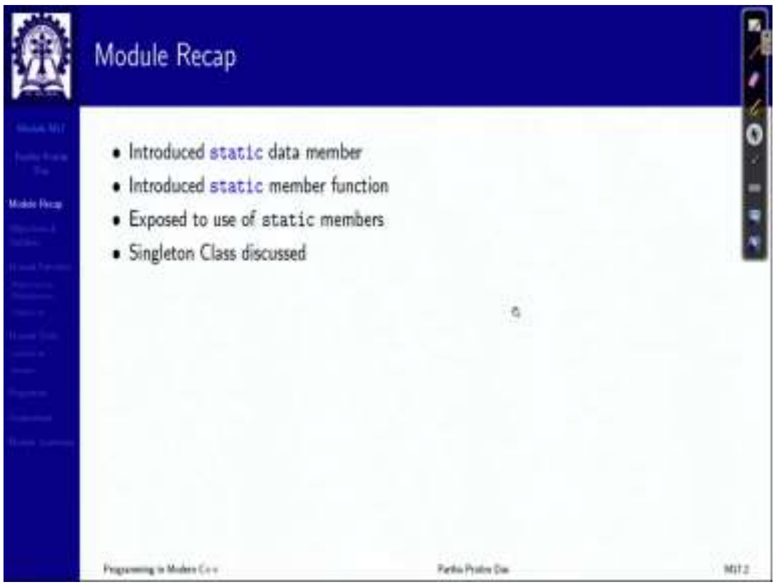
Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Week 04, Lecture 17
Friend Function and Friend Class

Welcome to Programming in Modern C++, we are in week 4. And we are going to discuss module 17.

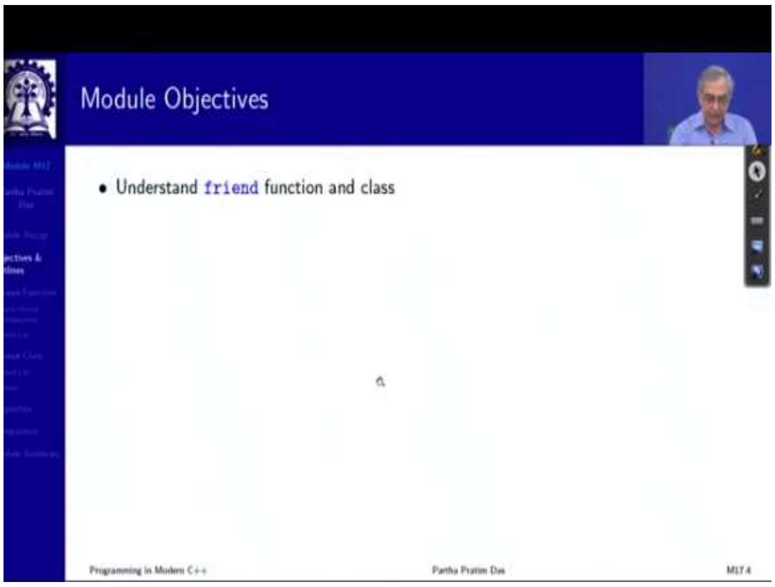
(Refer Slide Time: 00:36)



Module Recap

- Introduced static data member
- Introduced static member function
- Exposed to use of static members
- Singleton Class discussed

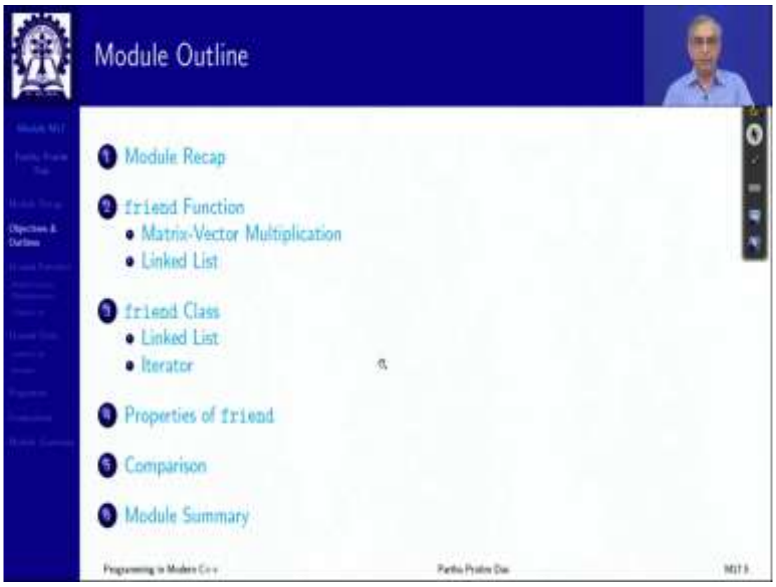
Programming in Modern C++ Partha Pratim Das M17.2



Module Objectives

- Understand friend function and class

Programming in Modern C++ Partha Pratim Das M17.4

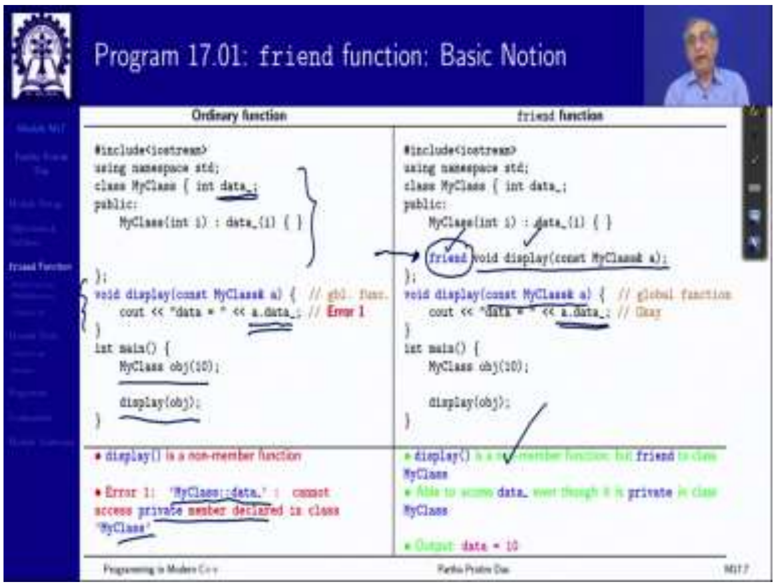

Module Outline

1. Module Recap
2. friend Function
   - Matrix-Vector Multiplication
   - Linked List
3. friend Class
   - Linked List
   - Iterator
4. Properties of friend
5. Comparison
6. Module Summary

Programming in Modern C++ Partha Pratim Das M17.6

We have in the last module discussed about static data members and member functions and singleton pattern, we will discuss a friend on the concept of friend function and class, which is related, but it is a different and this is this is kind of a very, very tricky feature that I am going to discuss in this module today, this outline and let us talk about friend function.

(Refer Slide Time: 01:02)


Program 17.01: friend function: Basic Notion

Ordinary function

```
#include<iostream>
using namespace std;
class MyClass { int data_;
public:
    MyClass(int i) : data_(i) { }

};
void display(const MyClass& a) { // gbl. func.
    cout << "data = " << a.data_; // Error 1
}
int main() {
    MyClass obj(10);

    display(obj);
}
```

- display() is a non-member function
- Error 1: 'MyClass::data_' : cannot access private member declared in class 'MyClass'

friend function

```
#include<iostream>
using namespace std;
class MyClass { int data_;
public:
    MyClass(int i) : data_(i) { }
    friend void display(const MyClass& a);
};
void display(const MyClass& a) { // global function
    cout << "data = " << a.data_; // Okay
}
int main() {
    MyClass obj(10);

    display(obj);
}
```

- display() is a non-member function; but friend to class MyClass
- Able to access data_ even though it is private in class MyClass
- Output: data = 10

Programming in Modern C++ Partha Pratim Das M17.7

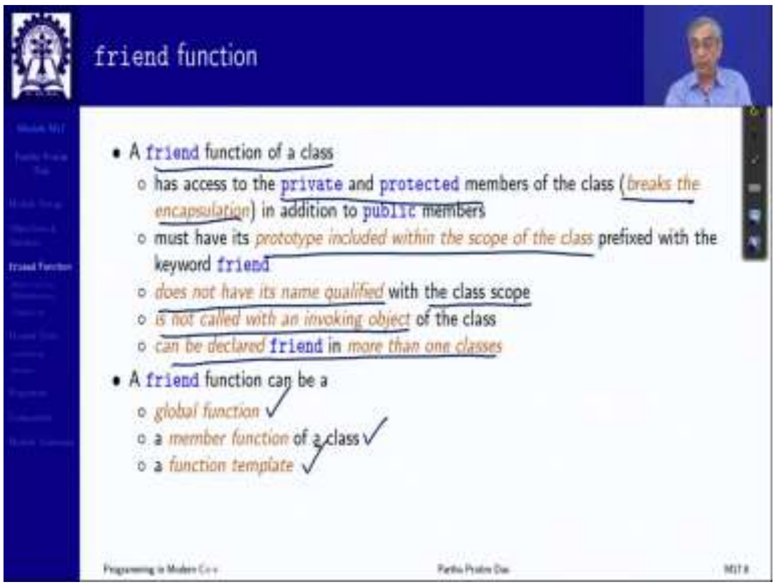
So, let me before getting into the definition, let me just start with an example. Let us see I have a class, MyClass and I am creating an object for that and I have defined a global function display which takes a reference constant reference to an object of MyClass and then it tries to display the data. So, if I invoke display with this object that have created, what do I expect to see, display is a global function it is a non-member. So, any access to private data in displayed by our information hiding principle is prohibited.

So, the compiler says MyClass, colon, colon, data cannot access the private member declared in MyClass. So, this code will not compile understood this is what we have learnt about the private access specifier. Now, I do a small change, I introduce this line inside the class. But I put the signature of the global function and preceded by a key word friend, rest of the code nothing is changed. Now it is perfectly fine. What is happening is display is still a non-member function.

Display gets the reference to an object of this class display is trying to access private data. Now, this is where I had an error in the global function case, and the left in this case, since I have called it a friend, I know that, MyClass is saying that everything is protect, everything is private, and not exposed to anyone else. Except for someone who is my friend. In a friendship, we all know, friendship is, we, the members of your family get into the house, the public on the road is outside cannot get into the house, but if I have a friend, I welcome the friend to my house. It is my prerogative.

So, it is exactly the same thing. So, MyClass has said that this function display which takes constant reference to MyClass and returns nothing is a friend. So, it is allowed to access the private members of the class that is basically the idea of the friend function. And so, you can see that friend is actually a way to selectively break the encapsulation that we have built up with so much of care, and why we do that will become clear soon.

(Refer Slide Time: 04:08)



friend function

- A friend function of a class
  - has access to the private and protected members of the class (*breaks the encapsulation*) in addition to public members
  - must have its *prototype included within the scope of the class* prefixed with the keyword friend
  - *does not have its name qualified* with the class scope
  - *is not called with an invoking object* of the class
  - *can be declared* friend in *more than one classes*
- A friend function can be a
  - *global function* ✓
  - a *member function* of a class ✓
  - a *function template* ✓

Programming in Modern C++ Partha Pratim Das M17.8

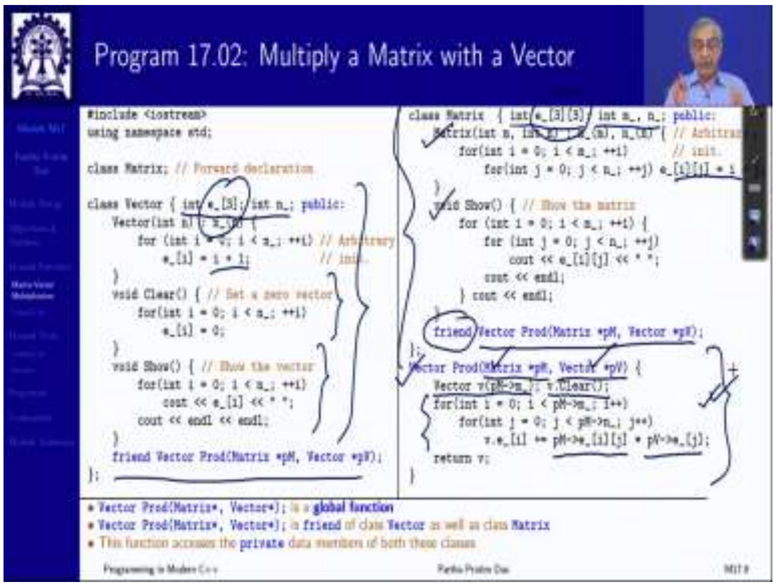
So, by definition, the it is a friend function of a class has access to private in public everybody has an access. So, does this friend function, it has access to private and protected members of the class protected we have still not discussed will come in when we do inheritance, but know that there is another type of access specification. So, clearly breaks encapsulation.

The prototype must be included within the scope of the class, so that the class is saying that I know that this function is my friend, but it is not a part of this class. So, its name is not qualified. It is a function outside of the class. So, it is just saying, I mean, a friend does not become your family. Friend and family are different. Family also has an access because they belong to the same group, but friend is someone who is not a part of that, but has a similar kind of access.

So, the name is not qualified. And it is not called with an invoking object, because it does not have it, this pointer of this class may have at this point or of some other class and a function can be declared a friend in more than one classes, not necessarily in only one class, what can be a friend function, a global function and example that we have seen, even the member function of one class can be a friend of another class. Member function of one class can be said to be friend of another class.

So, in that case, that function will require the, this pointer of the class of which is the member function, but not the, this pointer of the class of which it is friend, or it will not be scoped by that class, it will be scoped by the class in which it is a member function, or it could be a function template, we will talk more about that when you talk about template later, just know that it could be a template.

(Refer Slide Time: 06:13)



Program 17.02: Multiply a Matrix with a Vector

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

class Matrix; // Forward declaration

class Vector { int e_[3]; int n_; public:
    Vector(int n) : n_(n) {
        for (int i = 0; i < n_; ++i) // Arbitrary
            e_[i] = i + 1;           // init.
    }
    void Clear() { // Set a zero vector
        for(int i = 0; i < n_; ++i)
            e_[i] = 0;
    }
    void Show() { // Show the vector
        for(int i = 0; i < n_; ++i)
            cout << e_[i] << " ";
        cout << endl << endl;
    }
    friend Vector Prod(Matrix *pM, Vector *pV);
};
```

```
class Matrix { int e_[3][3]; int m_, n_; public:
    Matrix(int m, int n) : m_(m), n_(n) { // Arbitrary
        for(int i = 0; i < m_; ++i)         // init.
            for(int j = 0; j < n_; ++j) e_[i][j] = i
    }
    void Show() { // Show the matrix
        for (int i = 0; i < m_; ++i) {
            for (int j = 0; j < n_; ++j)
                cout << e_[i][j] << " ";
            cout << endl;
        } cout << endl;
    }
    friend Vector Prod(Matrix *pM, Vector *pV);
};
Vector Prod(Matrix *pM, Vector *pV) {
    Vector v(pM->m_); v.Clear();
    for(int i = 0; i < pM->m_; i++)
        for(int j = 0; j < pM->n_; j++)
            v.e_[i] += pM->e_[i][j] * pV->e_[j];
    return v;
}
```

- `Vector Prod(Matrix*, Vector*);` is a **global function**
- `Vector Prod(Matrix*, Vector*);` is **friend** of class `Vector` as well as class `Matrix`
- This function accesses the **private** data members of both these classes

Programming in Modern C++ Partha Pratim Das M17.9

So, let us take an example. The example here is to be able to multiply a matrix with a vector. So, I have a vector class, vector class, it has a constructor which does some arbitrary initialization, this initialization knows no meaning I just initialized with something, it has a way to clear a vector, set it all to 0. And it has a way to display the vector, these are the member functions of vector. So, which give you, tell you the status of the vector.

And I have a Matrix class which has, obviously, the vector has private members, which is the vector array and the size, I am not using the vector component of STL. Just giving you a simple example with array. Similarly, in matrix I have a two dimensional array, and the two dimensions m and n are constructed, which does arbitrary initialization here, I have a function to show that is display what is there in the matrix.

Now, I want to multiply these two So, I write a function here, which takes a matrix and a vector so I multiply a matrix by a vector what we will get I will get a vector. So, I create a vector v and that will get initialized with these things. So, I clear it I make it all 0, that is my result vector, that I want and this is the code, which is easy for you to see that you go over and you keep on making the component twice multiplication going along the rows columns and you can get the ...

Now, the issue here is this function needs to know the access the element here of matrix which is private. It also needs this more needs to access the element e of the vector which is private. So, the question is now, it needs to access that private members of vector as well as private members of matrix to be able to do the multiplication.

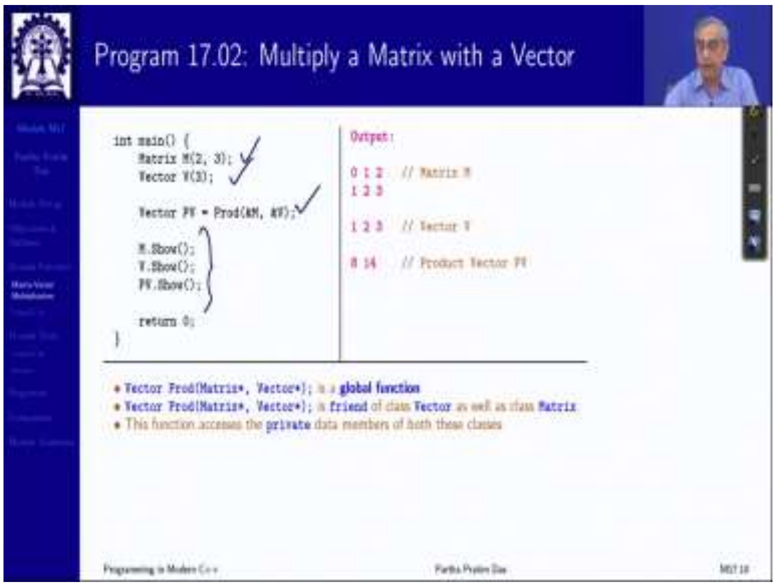
Now, which means that this will have to be a member function. Now, member function of which one for matrix when it does not work for vector for vector, then it does not work for matrix. So, that is where the concept of friend comes very handy. So, I, what I say that this is a global function let it be a global function. It is not a part of anyone, but I include the prototype of that with a friend keyword in matrix and I do the same thing in vector.

So, vector says that prod function is my friend, matrix says prod function is my friend. Since, vector says this any access from pV to the private members will be permitted. Since matrix is a friend any access from pM to the private members would be permitted, done and this code will perfectly run.

So, it is not that he will port make a function friend at any point of time and kind of break the encapsulation because you are actually breaking it, you are letting global functions external functions to take access to the private data which you have very carefully crafted in terms of your information hiding, but in such situations you can judiciously use it to have a very nice solution.

If you did not have this then what you will need to do, you will need to do a lot of things you will need to put a get on every member function you will need to put a get on every member function of vector for matrix. Lot of function calls will get involved then they will have to be optimized by inlining and so on so forth. But here you write a very clean code for doing all.

(Refer Slide Time: 10:34)



Program 17.02: Multiply a Matrix with a Vector

```
int main() {
    Matrix M(2, 3);
    Vector V(3);

    Vector PV = Prod(&M, &V);

    M.Show();
    V.Show();
    PV.Show();

    return 0;
}
```

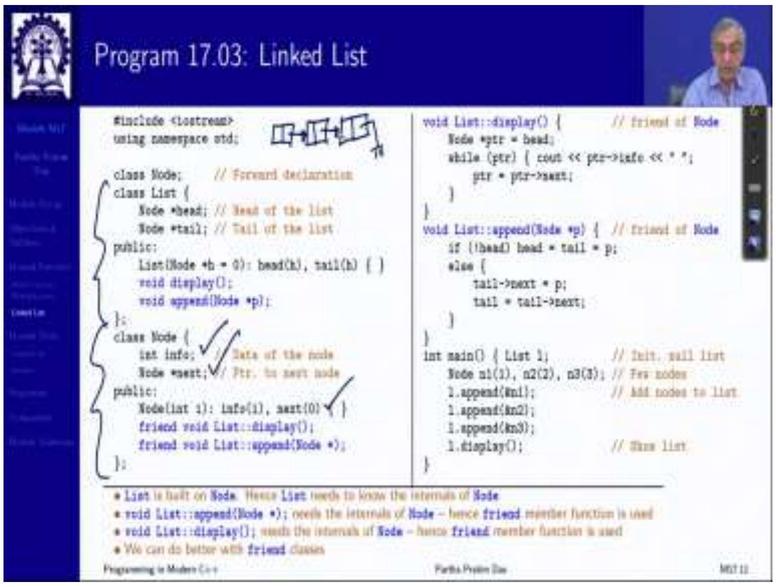
```
Output:

0 1 2   // Matrix M
1 2 3

1 2 3   // Vector V

8 14    // Product Vector PV
```

- Vector Prod(Matrix*, Vector*); is a global function
- Vector Prod(Matrix*, Vector*); is friend of class Vector as well as class Matrix
- This function accesses the private data members of both these classes

Programming in Modern C++ Partha Pratim Das M17.18

So, if you have done that, then this is you can this is for you to basically try out you can put all this code execute and see that you are able to see the product done in a very nice way, very clean way. So, to say that the whole thing will get done.

(Refer Slide Time: 10:52)



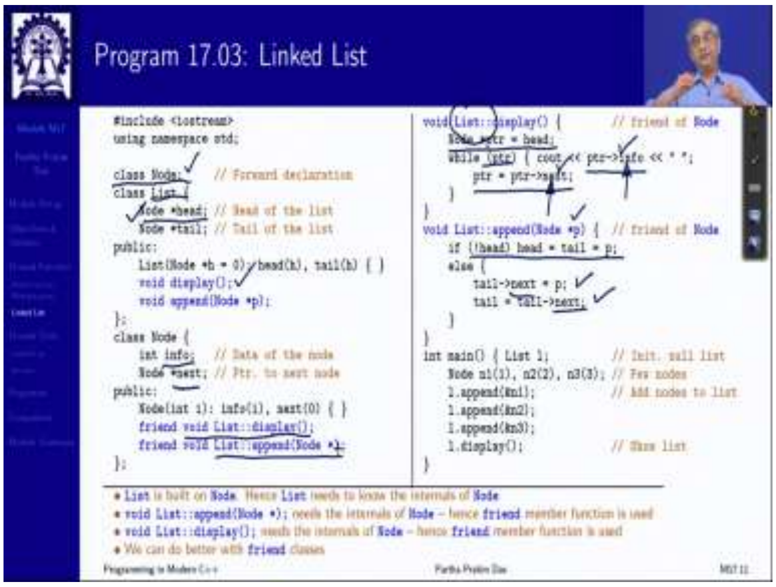
Program 17.03: Linked List

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

class Node;     // Forward declaration
class List {
    Node *head; // Head of the list
    Node *tail; // Tail of the list
public:
    List(Node *h = 0): head(h), tail(h) { }
    void display();
    void append(Node *p);
};
class Node {
    int info;   // Data of the node
    Node *next; // Ptr. to next node
public:
    Node(int i): info(i), next(0) { }
    friend void List::display();
    friend void List::append(Node *);
};
```

```cpp
void List::display() {          // friend of Node
    Node *ptr = head;
    while (ptr) { cout << ptr->info << " ";
        ptr = ptr->next;
    }
}
void List::append(Node *p) {  // friend of Node
    if (!head) head = tail = p;
    else {
        tail->next = p;
        tail = tail->next;
    }
}
int main() { List l;          // Init. null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1);            // Add nodes to list
    l.append(&n2);
    l.append(&n3);
    l.display();              // Show list
}
```

- List is built on Node. Hence List needs to know the internals of Node
- void List::append(Node *); needs the internals of Node – hence friend member function is used
- void List::display(); needs the internals of Node – hence friend member function is used
- We can do better with friend classes

Programming in Modern C++ Partha Pratim Das M17.11



Program 17.03: Linked List

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

class Node;     // Forward declaration
class List {
    Node *head; // Head of the list
    Node *tail; // Tail of the list
public:
    List(Node *h = 0): head(h), tail(h) { }
    void display();
    void append(Node *p);
};
class Node {
    int info;   // Data of the node
    Node *next; // Ptr. to next node
public:
    Node(int i): info(i), next(0) { }
    friend void List::display();
    friend void List::append(Node *);
};
```

```cpp
void List::display() {          // friend of Node
    Node *ptr = head;
    while (ptr) { cout << ptr->info << " ";
        ptr = ptr->next;
    }
}
void List::append(Node *p) {  // friend of Node
    if (!head) head = tail = p;
    else {
        tail->next = p;
        tail = tail->next;
    }
}
int main() { List l;          // Init. null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1);            // Add nodes to list
    l.append(&n2);
    l.append(&n3);
    l.display();              // Show list
}
```

- List is built on Node. Hence List needs to know the internals of Node
- void List::append(Node *); needs the internals of Node – hence friend member function is used
- void List::display(); needs the internals of Node – hence friend member function is used
- We can do better with friend classes

Programming in Modern C++ Partha Pratim Das M17.11

Another example, let us say we are building a data structure link list. So, what does link list involve, link list involves two kinds of classes objects, one is the node every node of the link list, which must have some data, information and the link at least one link if it is doubly linked lists and it adds two links so on. So, I have a node class, which has information, the data of that node and this is a pointer to the next node. So, this is a minimum that I need I have a constructor based on that.

Now, I need another class, which is the list itself, which is a list of nodes. So, I have a class list class, which actually has two pointers, one is so it is like this I have so this is my list. So, this is my nodes. And my list has two pointers, one is a head pointer, one is a tail pointer. Having the head pointer is important. Tail pointer is just a matter of convenience. Because if we want to append, it gets easier if you have the tail pointer. Otherwise, you will have to go all through the list we have to prepend you need the head pointer. So, I have a pair of node pointers.

Now, so, what I have is a pair of classes which are interdependent. So, he can easily see that list is making reference to node. So, they are interdependent. Here, I could have declared node before and then list. I have just chosen to do it this way just to show you that when such things happen, this will give you an error because it does not compile it does not know what is node*. So, I have given a forward declaration for the node without specifying the details.

Now, the question is suppose I have to now display, write the display function on the list that is go over this list of nodes and display the values. So, display function will start with the node header as long as it will keep on traversing by doing pointers next pointer next pointer next and print the pointer data till you come to the last grounded pointer you have to do this.

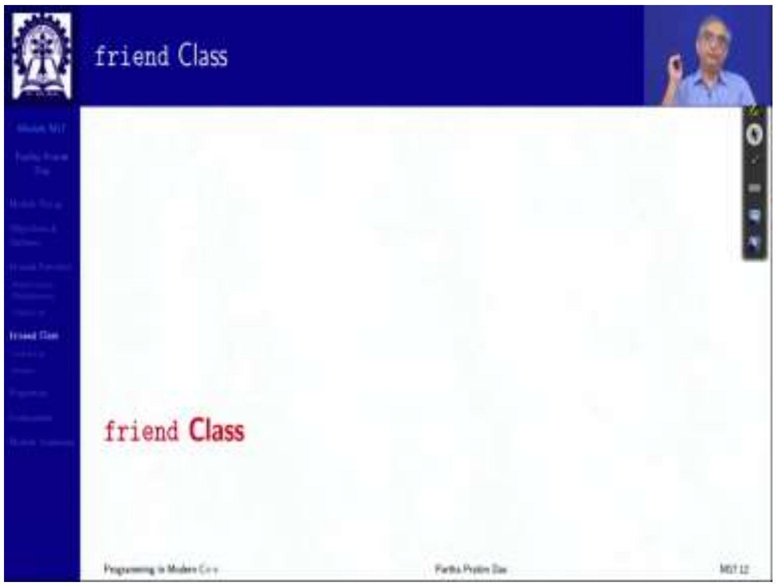
Now, the question is this function display is semantically logically a member function of list because there is no sense of displaying a node as such. So, this is a member function of list and it is trying to access, it is trying to access the private data of node again evaluation. So, to allow that, what I do is, I make this member function this member function is list, colon, colon, display.

I make that a friend of node when I make it a friend of node, the compiler will allow this function to access the private data members or private member functions here that is not their private data members of the node class. Similarly, I have an append to do. I am given a node pointer and have to append it to the list. So, I put this, then I put tail pointer, next as p and tail pointer next as tail, this we will insert.

So, I have made this also a friend of node. It is not, you can say that this is not necessary. Is it or, it is not? Why is it necessary? Is necessary because I need to manipulate the next pointer, which is private data member of node. So, without that I will not be able to without being a friend I will not be able to allow that access this will not compile.

So, I also make this as a friend function. So, you can see the earlier example of vector matrix multiplication was use of a using a global function to make friend of two classes here we are making the member functions of one class as friend of the other class, so, that this can cross communicate in this manner. So, these are typical situations where friend really can help you and you can run this example to see.

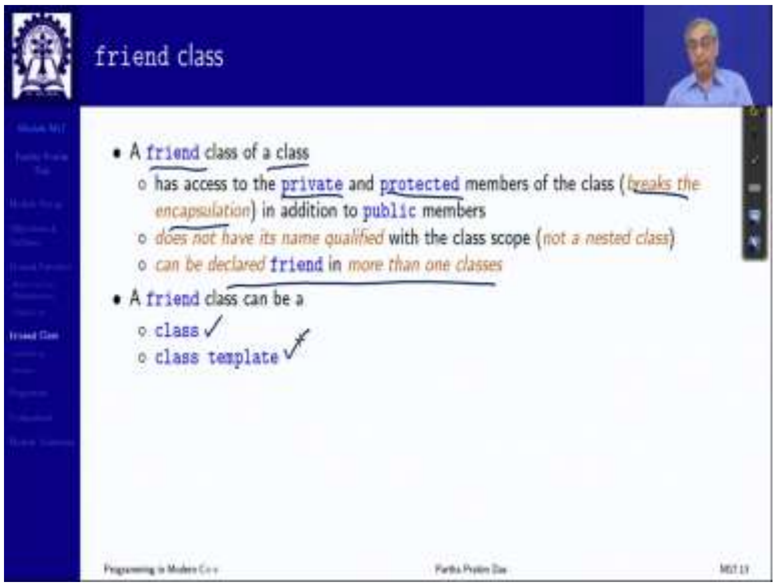
(Refer Slide Time: 16:27)



friend Class

friend **Class**

Programming in Modern C++ Partha Pratim Das M17.12



friend class

- A friend class of a class
  - has access to the private and protected members of the class (*breaks the encapsulation*) in addition to public members
  - *does not have its name qualified* with the class scope (*not a nested class*)
  - *can be declared* friend in *more than one classes*
- A friend class can be a
  - class ✓
  - class template ✓

Programming in Modern C++ Partha Pratim Das M17.13

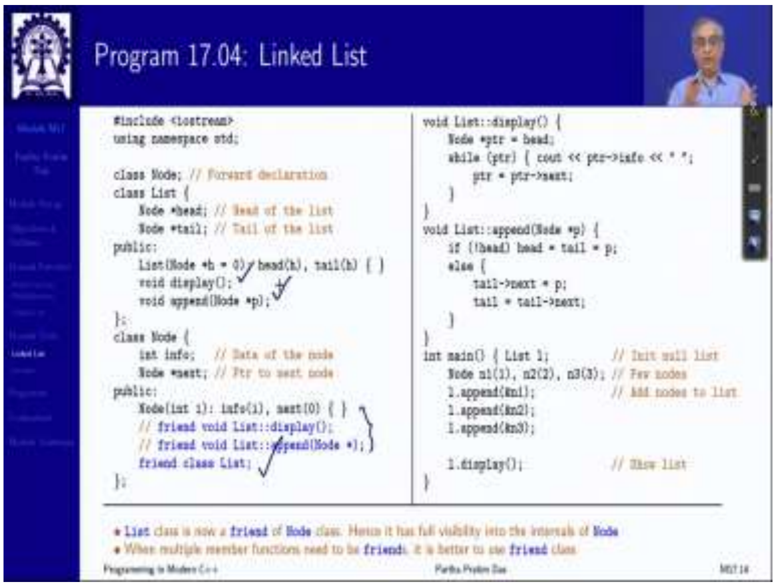
Now, as you can see, here, we had two member functions, which had to be friend of another class in several situations, there may be many member functions of a class which need to be friend of another class. So, in that case, we have an added feature by which we can make a class as a whole a friend of another class.

So, it is like a like a family friend, where you do not say that individual member is a friend is one way of saying but you say, Mr. Vardhan is our family friend, which includes that Mr. Vardhan is a friend, Mrs. Vardhan is a friend, their son is a friend, their daughter is a friend, their daughter in law is a friend, their son in law is a friend, the entire thing.

So, similarly, so, I say that friend class of a class in the similar manner has access to private and protected members of the class breaks encapsulation in the same way and does not have a name which is qualified in this class is not a nested class, there is a concept of a nested class again we have not done a class within a class, this is not done yet just think there are two independent classes this class says that class is my friend.

So, that does can have all kinds of private accesses. And it can be declared friend in one or more classes, what can be a friend a class or a class template, when we do template.

(Refer Slide Time: 18:06)



Program 17.04: Linked List

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

class Node; // Forward declaration
class List {
    Node *head; // Head of the list
    Node *tail; // Tail of the list
public:
    List(Node *h = 0): head(h), tail(h) { }
    void display();
    void append(Node *p);
};
class Node {
    int info;    // Data of the node
    Node *next; // Ptr to next node
public:
    Node(int i): info(i), next(0) { }
    // friend void List::display();
    // friend void List::append(Node *);
    friend class List;
};
```

```
void List::display() {
    Node *ptr = head;
    while (ptr) { cout << ptr->info << " ";
        ptr = ptr->next;
    }
}
void List::append(Node *p) {
    if (!head) head = tail = p;
    else {
        tail->next = p;
        tail = tail->next;
    }
}
int main() { List l;         // Init null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1);            // Add nodes to list
    l.append(&n2);
    l.append(&n3);

    l.display();              // Show list
}
```

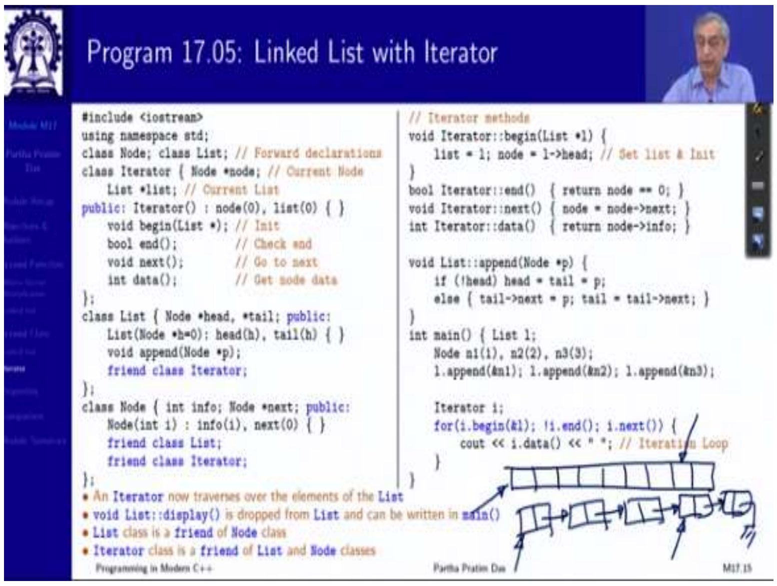
- List class is now a friend of Node class. Hence it has full visibility into the internals of Node
- When multiple member functions need to be friends, it is better to use friend class

Programming in Modern C++ Partha Pratim Das M17.14

So, we will rework on the previous example using this. So, all that we do is in the same link list, now in the node class I do not say that a list, colon, colon, void is a friend function. List, colon, colon, append is a friend function, rather they say, that the friend class list, class list as a whole is a friend. So, right now class list has only two member functions other than the constructor destructor etcetera. If I add another member functions, I do not have to say that it is a friend of node it will become a friend.

If I add a preprint function, if I add a search function to the class list, they all will automatically become friend of the node because the class as a whole is a friend. So, this is just a kind of it is not only a syntactic sugar, I should say but it is kind of a semantic sugar that will do that to say that I have a concept of the friends family and the friends family is a friend. Everybody is a friend within that.

(Refer Slide Time: 19:16)

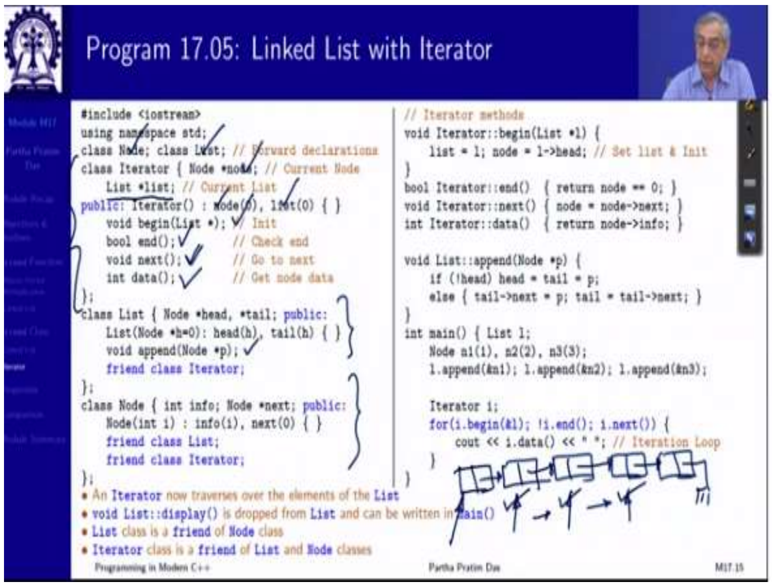


Program 17.05: Linked List with Iterator

```
#include <iostream>
using namespace std;
class Node; class List; // Forward declarations
class Iterator { Node *node; // Current Node
    List *list; // Current List
public: Iterator() : node(0), list(0) { }
    void begin(List *); // Init
    bool end();         // Check end
    void next();        // Go to next
    int data();         // Get node data
};
class List { Node *head, *tail; public:
    List(Node *h=0): head(h), tail(h) { }
    void append(Node *p);
    friend class Iterator;
};
class Node { int info; Node *next; public:
    Node(int i) : info(i), next(0) { }
    friend class List;
    friend class Iterator;
};
```

```
// Iterator methods
void Iterator::begin(List *l) {
    list = l; node = l->head; // Set list & Init
}
bool Iterator::end()  { return node == 0; }
void Iterator::next() { node = node->next; }
int Iterator::data()  { return node->info; }

void List::append(Node *p) {
    if (!head) head = tail = p;
    else { tail->next = p; tail = tail->next; }
}
int main() { List l;
    Node n1(1), n2(2), n3(3);
    l.append(&n1); l.append(&n2); l.append(&n3);

    Iterator i;
    for(i.begin(&l); !i.end(); i.next()) {
        cout << i.data() << " "; // Iteration Loop
    }
}
```

- An Iterator now traverses over the elements of the List
- void List::display() is dropped from List and can be written in main()
- List class is a friend of Node class
- Iterator class is a friend of List and Node classes



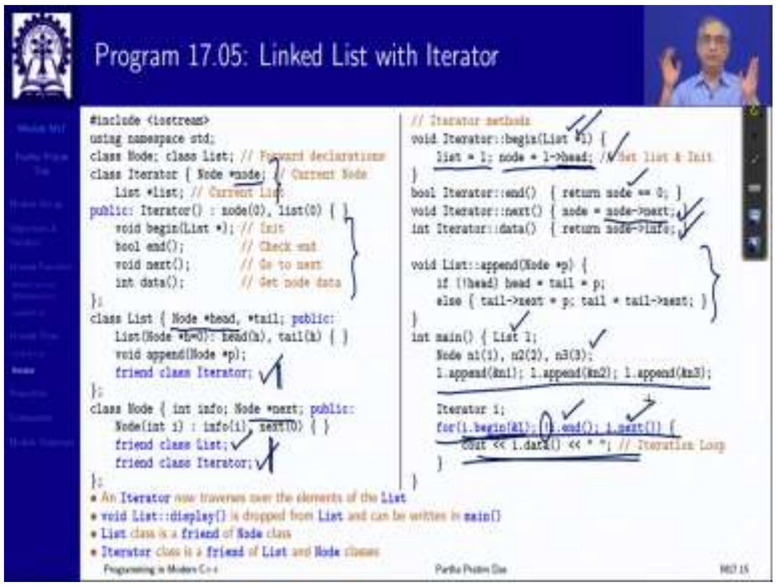
Program 17.05: Linked List with Iterator

```
#include <iostream>
using namespace std;
class Node; class List; // Forward declarations
class Iterator { Node *node; // Current Node
    List *list; // Current List
public: Iterator() : node(0), list(0) { }
    void begin(List *); // Init
    bool end();         // Check end
    void next();        // Go to next
    int data();         // Get node data
};
class List { Node *head, *tail; public:
    List(Node *h=0): head(h), tail(h) { }
    void append(Node *p);
    friend class Iterator;
};
class Node { int info; Node *next; public:
    Node(int i) : info(i), next(0) { }
    friend class List;
    friend class Iterator;
};
```

```
// Iterator methods
void Iterator::begin(List *l) {
    list = l; node = l->head; // Set list & Init
}
bool Iterator::end()  { return node == 0; }
void Iterator::next() { node = node->next; }
int Iterator::data()  { return node->info; }

void List::append(Node *p) {
    if (!head) head = tail = p;
    else { tail->next = p; tail = tail->next; }
}
int main() { List l;
    Node n1(1), n2(2), n3(3);
    l.append(&n1); l.append(&n2); l.append(&n3);

    Iterator i;
    for(i.begin(&l); !i.end(); i.next()) {
        cout << i.data() << " "; // Iteration Loop
    }
}
```

- An Iterator now traverses over the elements of the List
- void List::display() is dropped from List and can be written in main()
- List class is a friend of Node class
- Iterator class is a friend of List and Node classes

Programming in Modern C++ Partha Pratim Das M17.15



Program 17.05: Linked List with Iterator

```
#include <iostream>
using namespace std;
class Node; class List; // Forward declarations
class Iterator { Node *node; // Current Node
    List *list; // Current List
public: Iterator() : node(0), list(0) { }
    void begin(List *); // Init
    bool end();         // Check end
    void next();        // Go to next
    int data();         // Get node data
};
class List { Node *head, *tail; public:
    List(Node *h=0): head(h), tail(h) { }
    void append(Node *p);
    friend class Iterator;
};
class Node { int info; Node *next; public:
    Node(int i) : info(i), next(0) { }
    friend class List;
    friend class Iterator;
};
```

```
// Iterator methods
void Iterator::begin(List *l) {
    list = l; node = l->head; // Set list & Init
}
bool Iterator::end()  { return node == 0; }
void Iterator::next() { node = node->next; }
int Iterator::data()  { return node->info; }

void List::append(Node *p) {
    if (!head) head = tail = p;
    else { tail->next = p; tail = tail->next; }
}
int main() { List l;
    Node n1(1), n2(2), n3(3);
    l.append(&n1); l.append(&n2); l.append(&n3);

    Iterator i;
    for(i.begin(&l); !i.end(); i.next()) {
        cout << i.data() << " "; // Iteration Loop
    }
}
```

- An Iterator now traverses over the elements of the List
- void List::display() is dropped from List and can be written in main()
- List class is a friend of Node class
- Iterator class is a friend of List and Node classes

Programming in Modern C++ Partha Pratim Das M17.15

Then linked list with iterator. What is an iterator? Iterator is like a for loop. You have any set of elements. It could be an array, it could be a link list. So, an iterator is something which has a starting point and it has an ending marker somewhere. Starting point and ending and from the starting to the end I keep on doing it, we have talked about forward iterator, reverse iterators all that in the context of the STL as well. So, now we are seeing how to how to build that kind of iterator.

So, let us see. So, what we have, we have the node class, the list class. So, this is the list class this a node class, you know other functions of that are not being highlighted only I have only included append here there could be several other function because the focus in this example is different and then I have an iterator class which has to iterate on the list.

So, what the iterator has to remember, it has to remember the list that it is iterating on, it has to know the list on which it is iterating so, it has a list pointer and while it is iterating it is a list like this. Is a list like this, this is the list the header then, after having seen this node, I will need to go to the next node then I will need to go to the next node, then I will need to go to the next node for a marker that keeps on traversing.

So, I need a node pointer which says which is a current node. This is a basic you know, information data of the iterator then what are the things you will need to do? You will have to initialize, that is tell the iterator that this is the list on which you will iterate, that is the initialization. See, this is that is different from the construction of the iterator I can construct a traitor once and use it to traverse multiple lists constructor all that it has to do is set the node on the list to null. But then I do begin to say that this is where my iteration starts for this particular list.

I need an end to check where to stop, that the list could have 20,000 elements and I want to iterate for 1000 only. So, I will have some condition or it may be the end of the list. Some condition to check when to stop the iteration. So, end is a function in the iterator if I call that it checks if the int has arrived, if the int have arrived, it will stop. Here I have modeled assuming that it goes up to the end of the list.

Then at any point what I need once I am done with visiting one node, I need to go to the next node. Once I am done with that I need to go to the next node and so on. So, I need the next function which takes me to the next node on the list that is the next member function here. Then when I am at a node, I will need to get the data of that node because otherwise why am

I iterating because I want that information of the node. So, these 4 functions begin end next and data form the interface for an iterator.

These two data members form the implementation of the data member of the iterator. So, now let us try to write this function. So, I want to write begin so I have list pointer pass to it. So, the list member of the iterator will be set to l, node member will have to be set to l pointer head the first node. So, if I have to do that, where is l pointer head, l pointer head is here node* head. So, it is a private data member of the list class I will not be able to use it, access it, what I do? I make class iterator as a friend, I made the class iterate as a friend. So, now it can freely use.

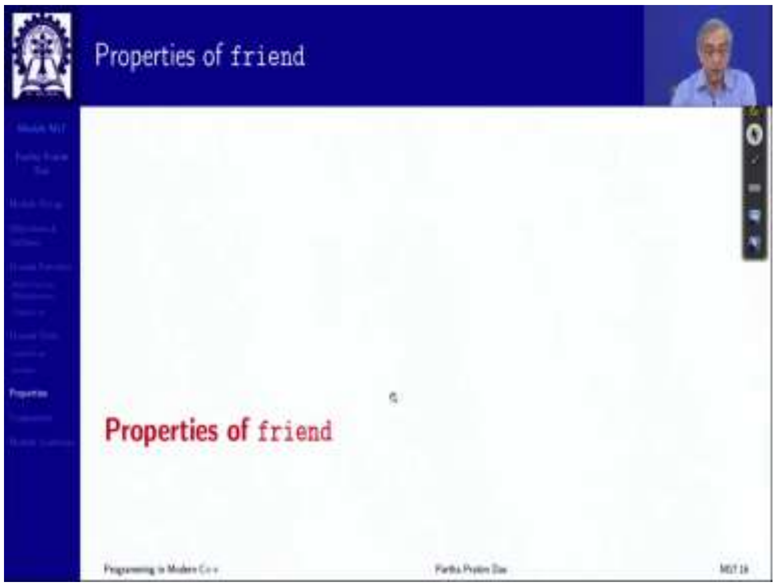
Think about the end, all that end needs to decide is your current node is null. It does not need to look into list or node classes. But as an ensemble of list class, it has become friend of list as ensemble of the iterator class, which is a friend of list it is also become a friend of this member function also become a friend. In the next one, I need to access node pointer next, which is a private data of the node class. So, iterator must be a friend of node.

So, iterator is a friend of list as well as the friend of node and friend of node is needed for these two member functions of iterator and friend of list is needed for this one. Similarly, I have an append function, which I have seen before which needs that list is a friend of node. So, you can see that list has a friend which is iterative. And node has two friend classes. One is a list in other is a iterator.

And with that, I can write this whole iteration process in a very compact manner, this is a list I have created some arbitrary node, I have appended them to this list, which I did in the previous example also. And then I do a begin that is initialization, which sets everything here then I check if the end has been reached, if not, if not, I will continue what I will do in the continuation, I will print the data, the information of that node, and then I will go to the next node.

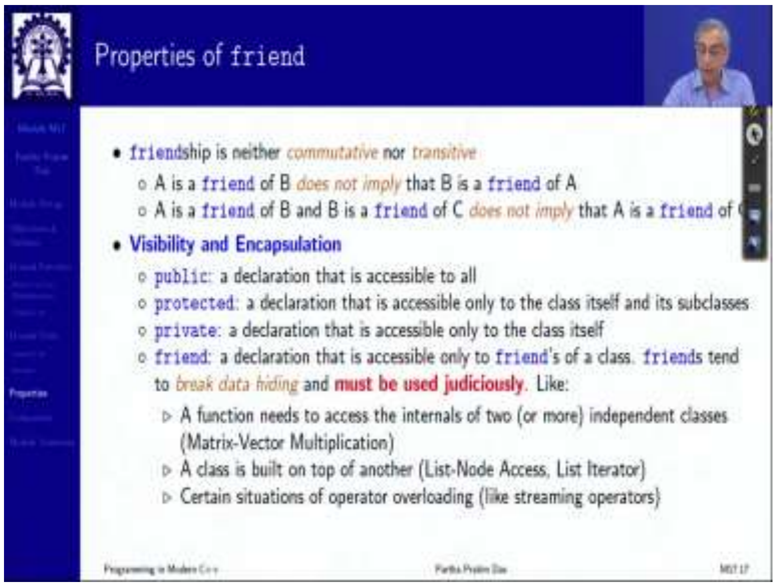
See, the beauty of this whole design is and that is the beauty of iterator and again, again, it is a design pattern and we will talk about that more. The beauty of this design is it does not need to know what is there in the list what is there in the node, all that none of those details, but this simple code is like a simple for loop, which will always work and to be able to make such things work we need the friend class friend function.

(Refer Slide Time: 27:10)


Properties of friend

Properties of friend

Programming in Modern C++ Partha Pratim Das M17.16


Properties of friend

- friendship is neither *commutative* nor *transitive*
  - A is a friend of B *does not imply* that B is a friend of A
  - A is a friend of B and B is a friend of C *does not imply* that A is a friend of C
- **Visibility and Encapsulation**
  - public: a declaration that is accessible to all
  - protected: a declaration that is accessible only to the class itself and its subclasses
  - private: a declaration that is accessible only to the class itself
  - friend: a declaration that is accessible only to friend's of a class. friends tend to *break data hiding* and **must be used judiciously**. Like:
    - A function needs to access the internals of two (or more) independent classes (Matrix-Vector Multiplication)
    - A class is built on top of another (List-Node Access, List Iterator)
    - Certain situations of operator overloading (like streaming operators)

Programming in Modern C++ Partha Pratim Das M17.17

So, the basic properties of friend if you must remember is, first thing is friendship is neither commutative nor transitive, that is if A is a friend of B, B is not necessarily a friend of A. So, node has said that the list is its friend, but by this node does not become a friend of list. If this is to be said this is to be said explicitly.

Second is not transitive, that is if A is a friend of B and B is a friend of C does not mean that A is a friend of C, none of this will hold. In terms of visibility and encapsulation, as you can see public visible to everyone protected as we will see subsequently is visible to the class as well as to its sub classes. Private visible to only the class itself. Friend, anything which is a friend will have the visibility like the private.

So, but it is an external function or a global function or a member function or a template or any class or a class template. So, the friend will break the encapsulation, it will break the data hiding property and therefore, it must be used very judiciously. And just few scenarios and variants of that are where you use friend, like when the function needs to access the internals of two classes two independent classes as we have seen matrix and vector when one class is built on top of the other, there is a node on top of that you have list, on top of that you have iterator right a 3 stage hierarchy.

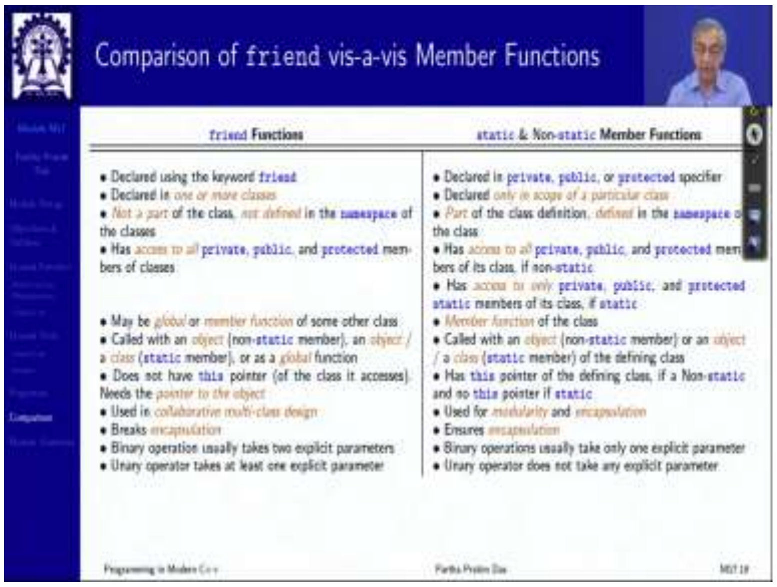
So, you need to go deep and see and that visibility requires friend to be used and we will have, we will show you some examples of specific streaming operators and so on, which make use a friend in a very specific way. And please keep those in mind and do not use the friend feature arbitrarily in your design.

(Refer Slide Time: 29:17)

Comparison of friend vis-a-vis Member Functions

Comparison of friend vis-a-vis Member Functions

Programming in Modern C++ Partha Pratim Das M17.18



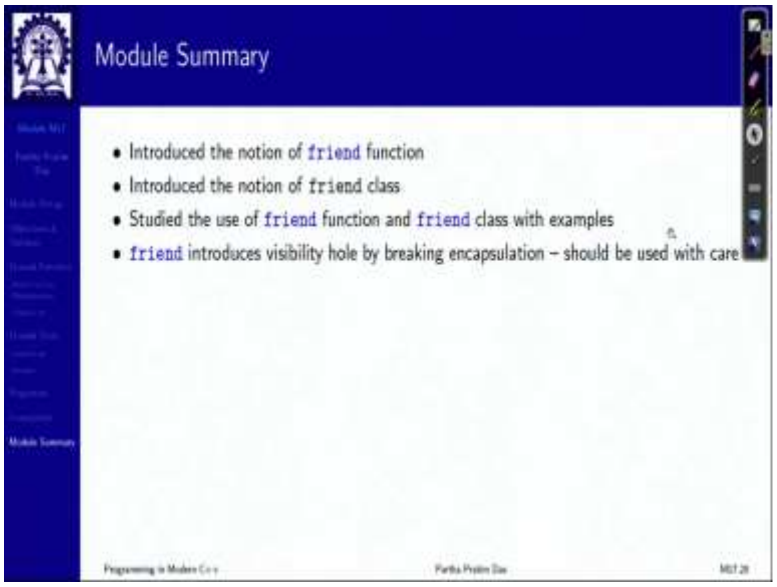
Comparison of friend vis-a-vis Member Functions

friend Functions	static & Non-static Member Functions
Declared using the keyword friend	Declared in private, public, or protected specifier
Declared in *one or more classes*	Declared *only in scope of a particular class*
Not a part of the class, *not defined* in the namespace of the classes	*Part* of the class definition, *defined* in the namespace of the class
Has *access to all* private, public, and protected members of classes	Has *access to all* private, public, and protected members of its class, if non-static
	Has *access to only* private, public, and protected static members of its class, if static
May be *global* or *member function* of some other class	*Member function* of the class
Called with an *object* (non-static member), an *object / a class* (static member), or as a *global* function	Called with an *object* (non-static member) or an *object / a class* (static member) of the defining class
Does not have this pointer (of the class it accesses). Needs the *pointer to the object*	Has this pointer of the defining class, if a Non-static and no this pointer if static
Used in *collaborative multi-class design*	Used for *modularity* and *encapsulation*
Breaks *encapsulation*	Ensures *encapsulation*
Binary operation usually takes two explicit parameters	Binary operations usually take only one explicit parameter
Unary operator takes at least one explicit parameter	Unary operator does not take any explicit parameter

Programming in Modern C++ Partha Pratim Das M17.19

So, here the whole of friend is summarized for friend functions, particularly we have made given I have given a comparison with the static and non-static member functions you have. If you have understood the basic concept these all will become like obvious for you, but just that summarized so that you can refer to them and in case we have a confusion that well what happens in this case, how does a friend behave and how does the static function behave or a non-static member function behave? You can refer to this comparative chart.

(Refer Slide Time: 29:58)



Module Summary

- Introduced the notion of friend function
- Introduced the notion of friend class
- Studied the use of friend function and friend class with examples
- friend introduces visibility hole by breaking encapsulation – should be used with care

Programming in Modern C++ Partha Pratim Das M17.20

So, that brings us to the end of this module where we have introduced the notion of friend function, and friend class, and discussed several very specific examples, and you will see that these examples keep on coming and even in your placement interviews and so on, it is very likely that you may be asked one of these some of these questions because they are very favorite of various most people.

And just remember that friend introduces visibility hole by breaking the encapsulation it must be used with care. And that is another point that most of the time interviewers would like to know that well, if it does, so, then why do you still have friend or do you use friend in such a situation and so on, you have to study and give response in a very judicious manner. Thank you very much for your patience and see you in the next module.