Module 16: Programming in C++

static Members

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand `static` data member and member function

Module Outline

static Data Member

- A `static` data member
  - is *associated with class* not with object
  - is *shared by all the objects* of a class
  - needs to be *defined outside the class scope* (in addition to the *declaration within the class scope*) to avoid linker error
  - *must be initialized* in a source file
  - is constructed before `main()` starts and destructed after `main()` ends
  - can be `private` / `public`
  - can be accessed
    - with the class-name followed by the scope resolution operator (`::`)
    - as a member of any object of the class
  - **virtually eliminates any need for global variables in OOPs environment**
- We illustrate first with a simple example and then with a Print Task where:
  - There is a printer which can be loaded with a paper from time to time
  - Several print jobs (each requiring a number of pages) may be fired on the printer

Program 16.01: static Data Member: Example

Non static Data Member

```
#include<iostream>
using namespace std;
class MyClass { int x; // Non-static
public:
    void set() { x = 15; }
    void print() { x = x + 10;
        cout << "x =" << x << endl ;
    }
};

int main() {
    MyClass obj1, obj2; // Have distinct x
    obj1.set(); obj2.set();
    obj1.print(); obj2.print();
}
---
x = 25 , x = 25
```

- x is a non-static data member
- x cannot be shared between obj1 & obj2
- Non-static data members do not need separate definitions - instantiated with the object
- Non-static data members are initialized during object construction

static Data Member

```
#include<iostream>
using namespace std;
class MyClass { static int x; // Declare static
public:
    void set() { x = 15; }
    void print() { x = x + 10;
        cout << "x =" << x << endl;
    }
};
int MyClass::x = 0; // Define static data member
int main() {
    MyClass obj1, obj2; // Have same x
    obj1.set(); obj2.set();
    obj1.print(); obj2.print();
}
---
x = 25 , x = 35
```

- x is static data member
- x is shared by all MyClass objects including obj1 & obj2
- static data members must be defined in the global scope

- static data members are initialized during program start-up

Program 16.02: static Data Member: Print Task (Unsafe)


```
#include <iostream>
using namespace std;
class PrintJobs { int nPages_;  /* # of pages in current job */ public:
    static int nTrayPages_; /* # of pages in the tray */ static int nJobs_; // # of print jobs executing
    PrintJobs(int nP): nPages_(nP) { ++nJobs_; cout << "Printing " << nP << " pages" << endl;
        nTrayPages_ = nTrayPages_ - nP;
    }                           // Job started
    ~PrintJobs() { --nJobs_; } // Job done
};
int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0;        // Definition and initialization -- no job to start with
int main() {
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
    {
        PrintJobs job1(30), job2(20); // Different job1 in block scope
        cout << "Jobs = " << PrintJobs::nJobs_ << endl;
        cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
        PrintJobs::nTrayPages_ += 100; // Load 100 more pages
    }
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
}
```

Output:

```
Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3 // same nJobs_, nTrayPages_
Pages= 440
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 540
```

Program 16.03/04: Order of Initialization: Order of Definitions


```
#include <iostream>
#include <string>
using namespace std;
class Data { string id_; public:
    Data(const string& id) : id_(id)
    { cout << "Construct: " << id_ << endl; }
    ~Data()
    { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
    static Data d1_; // Listed 1st
    static Data d2_; // Listed 2nd
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd

int main() { }
```

```
-----
Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1
```

```
#include <iostream>
#include <string>
using namespace std;
class Data { string id_; public:
    Data(const string& id) : id_(id)
    { cout << "Construct: " << id_ << endl; }
    ~Data()
    { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
    static Data d2_; // Order of static members swapped
    static Data d1_;
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd

int main() { }
```

```
-----
Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1
```

- *Order of initialization of static data members does not depend on their order in the definition of the class. It depends on the order their definition and initialization in the source*

static Member Function

- A `static` member function
  - does not have `this` pointer – not associated with any object
  - *cannot access* non-`static` data members
  - *cannot invoke* non-`static` member functions
  - can be accessed
    - with the class-name followed by the scope resolution operator (`::`)
    - as a member of any object of the class
  - is needed to read / write `static` data members
    - Again, for encapsulation `static` data members should be `private`
    - `get()-set()` idiom is built for access (`static` member functions in `public`)
  - *may initialize* `static` data members *even before any object creation*
  - *cannot co-exist with a non-*`static` *version* of the same function
  - *cannot be declared* as `const`
- We repeat the Print Task with better (safer) modeling and coding

Program 16.05: static Data & Member Function: Print Task (Safe)


```
// #include <iostream> using namespace std;
class PrintJobs { int nPages_; // # of pages in current job
    static int nTrayPages_; /* # of pages in the tray */ static int nJobs_; // # of print jobs executing
public: PrintJobs(int nP) : nPages_(nP) { ++nJobs_; cout << "Printing " << nP << " pages" << endl;
            nTrayPages_ = nTrayPages_ - nP; } // Job started
    ~PrintJobs() { --nJobs_; }                // Job done
    static int getJobs()          { return nJobs_; }      // get on nJobs_. Readonly. No set provided
    static int checkPages()       { return nTrayPages_; } // get on nTrayPages_
    static void loadPages(int nP) { nTrayPages_ += nP; }  // set on nTrayPages_
};
int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0;        // Definition and initialization -- no job to start with
int main() { cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    {
        PrintJobs job1(30), job2(20); // Different job1 in block scope
        cout << "Jobs = " << PrintJobs::getJobs() << endl;
        cout << "Pages= " << PrintJobs::checkPages() << endl;
        PrintJobs::loadPages(100);    // Load 100 more pages
    }
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
}
```

Output:

```
Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3 // same nJobs_, nTrayPages_
Pages= 440
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 540
```

Counting Objects

- We illustrate another example and use for `static` data member and member function
  - Here we want to track the number of objects created and destroyed for a class at any point in the program
  - Naturally no object can keep this information. So we hold two `static` data members
    - `nObjCons_`: Number of objects created since beginning. It is *read-only and incremented in every constructor*
    - `nObjDes_`: Number of objects destroyed since beginning. It is *read-only and incremented in the destructor*
  - At any point (`nObjCons_ - nObjDes_`) gives the number of *Live* objects
  - In an alternate (less informative model) we may just maintain `static` data member `nLive_` which is *incremented in every constructor and decremented in the destructor*

Program 16.06: Count Objects


```
#include <iostream>
#include <string>
using namespace std;
class MyClass { string id_; // Object ID
    static int nObjCons_, nObjDes_; // Object history
public:
    MyClass(const string& id) : id_(id)
    { ++nObjCons_;
    cout << "ctor: " << id_ << " "; getObjLive(); }
    ~MyClass() { ++nObjDes_;
    cout << "dtor: " << id_ << " "; getObjLive(); }
    static int getObjConstructed()
    { return nObjCons_; }
    static int getObjDestructed()
    { return nObjDes_; }
    // Get number of live objects
    static int getObjLive() {
        int nLive = nObjCons_ - nObjDes_;
        cout << "Live Objects = " << nLive << endl;
        return nLive;
    }
};
int MyClass::nObjCons_ = 0;
int MyClass::nObjDes_ = 0;
```

```
int dummy1(MyClass::getObjLive()); // Before (main())
MyClass sObj("sObj");
int dummy2(MyClass::getObjLive()); // Before (main())
int main() { MyClass::getObjLive();
    MyClass aObj("aObj");
    MyClass *dObj = new MyClass("dObj");
    {
        MyClass bObj("bObj");
        delete dObj;
    }
    MyClass::getObjLive();
}
```

```
Live Objects = 0 // Before any object (dummy1)
ctor: sObj Live Objects = 1
Live Objects = 1 // Before main() (dummy2)
Live Objects = 1 // Enter main()
ctor: aObj Live Objects = 2
ctor: dObj Live Objects = 3
ctor: bObj Live Objects = 4
dtor: dObj Live Objects = 3
dtor: bObj Live Objects = 2
Live Objects = 2 // Exit main()
: aObj Live Objects = 1
dtor: sObj Live Objects = 0 // After all objecst
```

Comparison of `static` vis-a-vis non-`static`

`static` Data Members	Non-`static` Data Members
• Declared using keyword `static`	• Declared *without* using keyword `static`
• All objects of a class share the *same copy / instance*	• Each object of the class gets its *own copy / instance*
• *Accessed* using the *class name or object*	• *Accessed* only through an *object* of the class
• May be `public` or `private`	• May be `public` or `private`
• Belongs to the `namespace` of the class	• Belongs to the `namespace` of the class
• May be `const`	• May be `const`
• Are *constructed before* `main()` is invoked	• Are *constructed during* object construction
• Are *destructed after* (in reverse order) `main()` returns	• Are *destructed during* object destruction
• Are *constructed* in the order of definitions in source	• Are *constructed* in the order of listing in the class
• Has a *lifetime* encompassing `main()`	• Has a *lifetime* as of the lifetime of the object
• *Allocated* in *static* memory	• *Allocated* in *static*, *stack*, or *heap* memory as of the object

`static` Member Functions	Non-`static` Member Functions
• Declared using keyword `static`	• Declared *without* using keyword `static`
• *Has no* `this` pointer parameter	• *Has an implicit* `this` pointer parameter
• *Invoked* using the *class name or object*	• *Invoked* only through an *object* of the class
• May be `public` or `private`	• May be `public` or `private`
• Belongs to the `namespace` of the class	• Belongs to the `namespace` of the class
• *Can access* `static` data members and methods	• *Can* access non-`static` data members and methods
• *Cannot access* non-`static` data members or methods	• Can be invoked only during *lifetime* of the object
• Can be invoked anytime during program execution	• May be `virtual` and / or `const`
• Cannot be `virtual` or `const`	• *There cannot be* a non-`static` Constructor
• *Constructor* is `static` though not declared `static`	

Singleton Class

- **Singleton** is a *creational design pattern*
  - ensures that *only one object* of its kind exists and
  - provides a *single point of access* to it for any other code
- A class is called a Singleton if it satisfies the above conditions
- Many classes are singleton:
  - President of India
  - Prime Minister of India
  - Director of IIT Kharagpur
  - CEO of a Company
  - ...
- How to implement a Singleton Class?
- How to restrict that user can created *only one* instance?

Program 16.07: static Data & Member Function Singleton Printer


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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
private: bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; } // Private -- Printer cannot be constructed!
    static Printer *myPrinter_;               // Pointer to the Instance of the Singleton Printer
public: ~Printer() { cout << "Printer destructed" << endl; }
    static const Printer& printer(bool bw = false, bool bs = false) { // Access the Printer
        if (!myPrinter_) myPrinter_ = new Printer(bw, bs);            // Constructed for first call
        return *myPrinter_;                                           // Reused from next time
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};

Printer *Printer::myPrinter_ = 0;

int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);

    delete &Printer::printer();
}
```

Output:

```
Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed
```

Program 16.08: Using function-local `static` Data
Singleton Printer


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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
    bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; }
    ~Printer() { cout << "Printer destructed" << endl; }
public:
    static const Printer& printer(bool bw = false, bool bs = false) {
        static Printer myPrinter(bw, bs); // The Singleton -- constructed the first time

        return myPrinter;
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};
int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);
}
```

Output:

```
Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed
```

- Function local `static` object is used
- No memory management overhead – so destructor too get `private`
- This is called **Meyer's Singleton**

Module Summary

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

Module 17: Programming in C++

friend Functions and friend Class

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand `friend` function and class

Module Outline

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

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`

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*

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 + j;
    }
    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

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

Program 17.03: 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 *);
};
```

```
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

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`

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 `friend`s, it is better to use `friend` class

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

Properties of `friend`

- `friend`ship 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. `friend`s 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)

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`
• Breaks *encapsulation*	• Ensures *encapsulation*

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

Module 18: Programming in C++

Overloading Operator for User-Defined Types: Part 1

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir*, *jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by global function and member

Module Outline

1. Operator Function
   - Non-Member Function
   - Member Function
   - Operator Overloading Rules
2. Using Global Function
   - `public` data members
   - `private` data members
3. Using Member Function
   - `operator+`
   - `operator=`
   - Unary Operators
4. Module Summary

How can operator functions help?

- We have seen how overloading `operator+` a C-string wrapped in `struct` allows us a compact notation for concatenation of two strings (Module 09)
- We have seen how overloading `operator=` can define the deep / shallow copy for a UDT and / or help with user-defined copy semantics (Module 14)
- In general, operator overloading helps us to build complete algebra for UDT's much in the same line as is available for built-in types:
  - **Complex type**: Add (`+`), Subtract (`-`), Multiply (`*`), Divide (`/`), Conjugate (`!`), Compare (`==, !=, ...`), etc.
  - **Fraction type**: Add (`+`), Subtract (`-`), Multiply (`*`), Divide (`/`), Normalize (unary `*`), Compare (`==, !=, ...`), etc.
  - **Matrix type**: Add (`+`), Subtract (`-`), Multiply (`*`), Divide (`/`), Invert (`!`), Compare (`==`), etc.
  - **Set type**: Union (`+`), Difference (`-`), Intersection (`*`), Subset (`< <=`), Superset (`> >=`), Compare (`==, !=`), etc.
  - **Direct IO**: read (`<<`) and write (`>>`) for all types
- Advanced examples include:
  - **Smart Pointers**: De-reference (unary `*`), Indirection (`->`), Copy (`=`), Compare (`==, !=`), etc.
  - **Function Objects or Functors**: Invocation (`()`)

Operator Functions in C++: RECAP (Module 9)

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`c = a + b`	`operator=(c, operator+(a, b))`

- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct

// Operator function
MyType operator+(const MyType&, const MyType&);

a + b // Calls operator+(a, b)
```

- C++ allows users to define an operator function and overload it

Non-Member Operator Function

- A non-member operator function may be a
  - Global Function
  - `friend` Function
- **Binary Operator:**

```
MyType a, b; // An enum, struct or class
MyType operator+(const MyType&, const MyType&);        // Global
friend MyType operator+(const MyType&, const MyType&); // Friend
```

- **Unary Operator:**

```
MyType operator++(const MyType&);        // Global
friend MyType operator++(const MyType&); // Friend
```

- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`++a`	`operator++(a)`
`a++`	`operator++(a, int)  Special Case`
`c = a + b`	`operator=(c, operator+(a, b))`

Member Operator Function

- **Binary Operator:**

```
MyType a, b;                        // MyType is a class
MyType operator+(const MyType&); // Operator function
```

- The left operand is the invoking object – right is taken as a parameter
- **Unary Operator:**

```
MyType operator-();     // Operator function for Unary minus
MyType operator++();    // For Pre-Incrementer
MyType operator++(int); // For post-Incrementer
```

- The only operand is the invoking object
- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

Operator Expression	Operator Function
`a + b`	`a.operator+(b)`
`a = b`	`a.operator=(b)`
`++a`	`a.operator++()`
`a++`	`a.operator++(int) // Special Case`
`c = a + b`	`c.operator =(a.operator+(b))`

Operator Overloading – Summary of Rules: RECAP (Module 9)

- No new operator such as `**`, `<>`, or `&|` can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be change
  - *Preserves arity*
  - *Preserves precedence*
  - *Preserves associativity*
- These operators *can be overloaded*:

  `[] + - * / % ^ & | ~ ! = += -= *= /= %= ^= &= |=`
  `<< >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) [ ]`
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) *cannot be overloaded*
- The overloads of operators `&&`, `||`, and `,` (comma) *lose their special properties*: short-circuit evaluation and sequencing
- For a member operator function, invoking object is passed implicitly as the left operand but the right operand is passed explicitly
- For a non-member operator function (Global/friend) operands are always passed explicitly

Program 18.01: Using Global Function: public Data members (Unsafe)

Overloading + for complex addition

```
#include <iostream>
using namespace std;
struct complx { // public data member
    double re, im;
} ;
complx operator+ (complx &a, complx &b) {
     complx r;
     r.re = a.re + b.re;
     r.im = a.im + b.im;
     return r;
}
int main() { complx d1 , d2 , d;
    d1.re = 10.5; d1.im = 12.25;
    d2.re = 20.5; d2.im = 30.25;
    d = d1 + d2; // Overload operator +
    cout << "Real:" << d.re << ", ";
    cout << "Imag:" << d.im;
}
```

- **Output**: Real: 31, Imag: 42.5
- **operator+** is overloaded to perform addition of two complex numbers which are of **struct complx** type

Overloading + for string cat

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1, const String& s2) {
    String s;
    s.str = (char *) malloc(strlen(s1.str) +
                            strlen(s2.str) + 1);
    strcpy(s.str, s1.str); strcat(s.str, s2.str);
    return s;
}
int main() { String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das");
    name = fName + lName; // Overload operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
}
```

- **Output**: First Name: Partha, Last Name: Das, Full name: Partha Das
- **operator+** is overloaded to perform concat of first name and last to form full name. The data type is **String**

Program 18.02: Using Global Function: private Data members (Safe)


```
#include <iostream>
using namespace std;
class Complex { // Private data members
    double re, im;
public:
    Complex(double a=0.0, double b=0.0):
        re(a), im(b) { } ~Complex() { }
    void display();
    double real() { return re; }
    double img() { return im; }
    double set_real(double r) { re = r; }
    double set_img(double i) { im = i; }
} ;
void Complex::display() {
    cout << re << " +j " << im << endl;
}
```

```
Complex operator+(Complex &t1, Complex &t2) {
    Complex sum;
    sum.set_real(t1.real() + t2.real());
    sum.set_img(t1.img() + t2.img());
    return sum;
}
int main() {
    Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
    cout << "1st complex No:"; c1.display();
    cout << "2nd complex No:"; c2.display();
    c3 = c1 + c2; // Overload operator +
    cout << "Sum = "; c3.display();
}
```

- **Output**:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 8.3 +j 10.25
Sum = 12.8 +j 35.5
```

- Accessing private data members inside operator functions is clumsy
- Critical data members need to be exposed (get/set) violating encapsulation
- **Solution**: Member operator function or `friend` operator function

Program 18.03: Using Member Function


```
#include <iostream>
using namespace std;
class Complex { // Private data members
    double re, im;
public:
    Complex(double a=0.0, double b=0.0):
        re(a), im(b) { } ~Complex() { }
    void display();
    Complex operator+(const Complex &c) {
        Complex r;
        r.re = re + c.re;
        r.im = im + c.im;
        return r;
    }
} ;
```

```
void Complex::display() {
    cout << re;
    cout << " +j " << im << endl;
}
int main() {
    Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
    cout << "1st complex No:";
    c1.display();
    cout << "2nd complex No:";
    c2.display();
    c3 = c1 + c2; // Overloaded operator +
    cout << "Sum = ";
    c3.display();
    return 0;
}
```

- **Output**:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 8.3 +j 10.25
Sum = 12.8 +j 35.5
```

- Performing `c1 + c2` is equivalent to `c1.operator+(c2)`
- `c1` invokes the `operator+` function and `c2` is passed as an argument
- Similarly we can implement all binary operators (`%`, `-`, `*`, etc..)
- **Note:** No need of two arguments in overloading

Program 14.14: Overloading operator=: RECAP (Module 14)


```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { }        // ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
    ~String() { free(str_); }                                        // dtor
     String& operator=(const String& s) {
        if (this != &s) { free(str_); str_ = strdup(s.str_); len_ = s.len_; }
        return *this;
    }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket";
    s1.print(); s2.print();
    s1 = s1; s1.print();
}
(Football: 8)
(Cricket: 7)
(Football: 8)
```

- Check for self-copy `(this != &s)`
- In case of self-copy, do nothing

Notes on Overloading operator=: RECAP (Module 14)

- Overloaded operator= may choose between *Deep* and *Shallow Copy* for Pointer Members
  - *Deep copy* allocates new space for the contents and copies the pointed data
  - *Shallow copy* merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data
- If operator= is not overloaded by the user, compiler provides a free one.
- Free operator= can makes only a shallow copy

Program 18.04: Overloading Unary Operators

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

class MyClass { int data; public:
    MyClass(int d): data(d) { }

    MyClass& operator++()    { // Pre-increment:
        ++data;                // Operate and return the operated object
        return *this;
    }
    MyClass operator++(int) { // Post-Increment:
        MyClass t(data);       // Return the (copy of) object; operate the object
        ++data;
        return t;
    }
    void disp() { cout << "Data = " << data << endl; }
};
int main() {
    MyClass obj1(8); obj1.disp();
    MyClass obj2 = obj1++; obj2.disp(); obj1.disp();

    obj2 = ++obj1;
    obj2.disp(); obj1.disp();
}
```

- Output

Data = 8
Data = 8
Data = 9
Data = 10
Data = 10

- The **pre-operator** should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be **MyClass&** and it should return ***this;**

- The **post-operator** should perform the operation (increment / decrement / other) after it returns the original value. Hence it should copy the original object in a temporary **MyClass t;** and then **return t;**. Its return type should be **MyClass** - by value

Program 18.05: Overloading Unary Operators: Pre-increment & Post Increment


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

class MyClass { int data;
public:
    MyClass(int d) : data(d) { }

    MyClass& operator++()   { // Pre-Operator
        data *= 2;
        return *this;
    }
    MyClass operator++(int) { // Post-Operator
        MyClass t(data);
        data /= 3;
        return t;
    }
    void disp() { cout << "Data = " << data << endl; }
};
int main() {
    MyClass obj1(12); obj1.disp();
    MyClass obj2 = obj1++; obj2.disp(); obj1.disp();

    obj2 = ++obj1;
    obj2.disp(); obj1.disp();
}
```

- Output

Data = 12
Data = 12
Data = 4
Data = 8
Data = 8

- The **pre-operator** and the **post-operator** need not merely increment / decrement
- They may be used for any other computation as this example shows
- However, it is a good design practice to keep close to the native semantics of the operator

Module Summary

- Introduced operator overloading for user-defined types
- Illustrated methods of overloading operators using global functions and member functions
- Outlined semantics for overloading binary and unary operators

Module 19: Programming in C++

Overloading Operator for User-Defined Types: Part 2

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir*, *jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by friend function and its advantages

Module Outline

1. Issues in Operator Overloading
2. `operator+`
3. `operator==`
4. `operator<<, operator>>`
5. Guidelines for Operator Overloading
6. Module Summary

Operator Function for UDT: RECAP (Module 18)

- Operator Function options:
  - Global Function
  - Member Function
  - `friend` Function
- **Binary Operator:**

```
MyType a, b; // An enum, struct or class
MyType operator+(const MyType&, const MyType&);         // Global
MyType operator+(const MyType&);                         // Member
friend MyType operator+(const MyType&, const MyType&); // Friend
```

- **Unary Operator:**

```
MyType operator++(const MyType&);         // Global
MyType operator++();                      // Member
friend MyType operator++(const MyType&); // Friend
```

- **Examples:**

Expression	Function	Remarks
`a + b`	`operator+(a, b)`	global / friend
`++a`	`operator++(a)`	global / friend
`a + b`	`a.operator+(b)`	member
`++a`	`a.operator++()`	member

Issue 1: Extending operator+

- Consider a `Complex` class. We have learnt how to overload `operator+` to add two `Complex` numbers:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

d3 = d1 + d2; // d3 = 4.1 +j 6.5
```

- Now we want to extend the operator so that a Complex number and a real number (no imaginary part) can be added together:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

d3 = d1 + 6.2; // d3 = 8.7 +j 3.2

d3 = 4.2 + d2; // d3 = 5.8 +j 3.3
```

- We show why global operator function is not good for this
- We show why member operator function cannot do this
- We show how `friend` function achieves this

Issue 2: Overloading IO Operators: operator<<, operator>>

- Consider a `Complex` class. Suppose we want to overload the streaming operators for this class so that we can write the following code:

```
Complex d;

cin >> d;

cout << d;
```

- Let us note that these operators deal with stream types defined in `iostream`, `ostream`, and `istream`:
  - `cout` is an `ostream` object
  - `cin` is an `istream` object
- We show why global operator function is not good for this
- We show why member operator function cannot do this
- We show how `friend` function achieves this

Program 19.01: Extending operator+ with Global Function


```
#include <iostream>
using namespace std;
class Complex { public: double re, im;
    explicit Complex(double r = 0, double i = 0): re(r), im(i) { } // No implicit conversion is allowed
    void disp() { cout << re << " +j " << im << endl; }
};
Complex operator+(const Complex &a, const Complex &b) { // Overload 1
    return Complex(a.re + b.re, a.im + b.im);
}
Complex operator+(const Complex &a, double d) {         // Overload 2
    Complex b(d); return a + b; // Create temporary object and use Overload 1
}
Complex operator+(double d, const Complex &b) {         // Overload 3
    Complex a(d); return a + b; // Create temporary object and use Overload 1
}
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

- Works fine with global functions - 3 separate overloading are provided
- A bad solution as it breaks the encapsulation – as discussed in Module 18
- **Let us try to use member function**
- **Note:** A simpler solution uses Overload 1 and implicit casting (for this we need to remove `explicit` before constructor). But that too breaks encapsulation. We discuss this when we take up cast operators

Program 19.02: Extending operator+ with Member Function


```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit conversion is allowed
    void disp() { cout << re << " +j " << im << endl; }
    Complex operator+(const Complex &a) {     // Overload 1
        return Complex(re + a.re, im + a.im);
    }
    Complex operator+(double d) {             // Overload 2
        Complex b(d);     // Create temporary object
        return *this + b; // Use Overload 1
    }
};
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2

    //d3 = 4.2 + d2;          // Overload 3 is not possible - needs an object on left
    //d3.disp();
}
```

- Overload 1 and 2 works
- Overload 3 cannot be done because the left operand is `double` – not an object
- **Let us try to use `friend` function**
- **Note:** This solution too avoids the feature of cast operators

Operator Overloading using `friend`

- Using global function, accessing `private` data members inside operator function is gets difficult
- It increases writing overhead, makes code complicated, else violates encapsulation
- As we saw till now most operators can actually be overloaded either by global function or member function, *But If the left operand is not an object of the class type then it cannot be overloaded through member function*
- To handle such situation, we require `friend` function
  - **Example:** For two objects `d1` & `d2` of the same class, we cannot overload (`constant + d2`) using member function. However, using `friend` function we can overload (`d1 + d2`), (`d1 + constant`), or (`constant + d2`)
  - **Reason:** While computing (`d1 + d2`) with member function, `d1` calls the `operator+()` and `d2` is passed as an argument. Similarly in (`d1 + constant`), `d1` calls the `operator+()` and `constant` is passed as an argument. But while calling (`constant + d2`) a `constant` cannot call the member function

    Similar analysis will also hold when `d1` & `d2` are objects of different classes and we cannot add the operator to the class of `d1`
- So operators like `<<`, `>>`, relational (`<`, `>`, `==`, `!=`, `<=`, `>=`) should be overloaded through `friend`

Program 19.03: Extending operator+ with friend Function


```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit conversion is allowed
    void disp() { cout << re << " +j " << im << endl; }
    friend Complex operator+(const Complex &a, const Complex &b) { // Overload 1
        return Complex(a.re + b.re, a.im + b.im);
    }
    friend Complex operator+(const Complex &a, double d) {         // Overload 2
        Complex b(d); // Create temporary object
        return a + b; // Use Overload 1
    }
    friend Complex operator+(double d, const Complex &b) {         // Overload 3
        Complex a(d); // Create temporary object
        return a + b; // Use Overload 1
    }
};
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

- Works fine with friend functions - 3 separate overloading are provided and Preserves the encapsulation too
- **Note:** A simpler solution uses only Overload 1 and implicit casting (for this we need to remove explicit before constructor) will be discussed when we take up cast operators

Program 19.04: Overloading operator== for strings with friend Function


```
#include <iostream>
#include <string>
#include <cstdlib>
#include <cstring>
using namespace std;
class MyStr { const char *name_; public:
    explicit MyStr(const char *s) : name_(strdup(s)) { } ~MyStr() { free((void *)name_); }
    friend bool operator==(const MyStr& s1, const MyStr& s2) { return !strcmp(s1.name_, s2.name_); }    // 1
    friend bool operator==(const MyStr& s1, const string& s2) { return !strcmp(s1.name_, s2.c_str()); } // 2
    friend bool operator==(const string& s1, const MyStr& s2) { return !strcmp(s1.c_str(), s2.name_); } // 3
};
int main() {
    MyStr mS1("red"), mS2("red"), mS3("blue"); string sS1("red"), sS2("red"), sS3("blue");
    if (mS1 == mS2) cout << "Match "; else cout << "Mismatch "; // MyStr, MyStr: Overload 1
    if (mS1 == mS3) cout << "Match "; else cout << "Mismatch "; // MyStr, MyStr: Overload 1
    if (mS1 == sS2) cout << "Match "; else cout << "Mismatch "; // MyStr, string: Overload 2
    if (mS1 == sS3) cout << "Match "; else cout << "Mismatch "; // MyStr, string: Overload 2
    if (sS1 == mS2) cout << "Match "; else cout << "Mismatch "; // string, MyStr: Overload 3
    if (sS1 == mS3) cout << "Match "; else cout << "Mismatch "; // string, MyStr: Overload 3
    if (sS1 == sS2) cout << "Match "; else cout << "Mismatch "; // string, string: C++ Lib
    if (sS1 == sS3) cout << "Match "; else cout << "Mismatch "; // string, string: C++ Lib
}
```

Output: `Match Mismatch Match Mismatch Match Mismatch Match Mismatch`

- `MyStr` is a user-defined string class while `string` is from C++ Standard Library. These are compared here by `operator==`.

Overloading IO Operators: operator<<, operator>>

- Consider `operator<<` for `Complex` class. This operator should take an `ostream` object (stream to write to) and a `Complex` (object to write). Further it allows to chain the output. So for the following code

```
Complex d1, d2;

cout << d1 << d2; // (cout << d1) << d2;
```

the signature of `operator<<` may be one of:

```
// Global function
ostream& operator<< (ostream& os, const Complex &a);

// Member function in ostream
ostream& ostream::operator<< (const Complex &a);

// Member function in Complex
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for `ostream` object is used so that chaining would work

Program 19.05: Overloading IO Operators with Global Function


```
#include <iostream>
using namespace std;
class Complex {
public: double re, im;
    Complex(double r = 0, double i = 0): re(r), im(i) { }
};
ostream& operator<<(ostream& os, const Complex &a) {
    os << a.re << " +j " << a.im << endl;
    return os;
}
istream& operator>>(istream& is, Complex &a) {
    is >> a.re >> a.im;
    return is;
}
int main() {
    Complex d;

    cin >> d;

    cout << d;
}
```

- Works fine with global functions
- A bad solution as it breaks the encapsulation – as discussed in Module 18
- **Let us try to use member function**

Overloading IO Operators with Member Function

- Case 1: `operator<<` is a member in `ostream` class:

```
ostream& ostream::operator<< (const Complex &a);
```

  This is not possible as `ostream` is a class in C++ standard library and we are not allowed to edit it to include the above signature
- Case 2: `operator<<` is a member in `Complex` class:

```
ostream& Complex::operator<< (ostream& os);
```

  In this case, the invocation of streaming will change to:

```
d << cout; // Left operand is the invoking object
```

  This certainly spoils the natural syntax
- IO operators cannot be overloaded by member functions
- **Let us try to use `friend` function**

Program 19.06: Overloading IO Operators with `friend` Function


```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    Complex(double r = 0, double i = 0): re(r), im(i) { }
    friend ostream& operator<<(ostream& os, const Complex &a);
    friend istream& operator>>(istream& is, Complex &a);

};
friend ostream& operator<<(ostream& os, const Complex &a) {
    os << a.re << " +j " << a.im << endl;
    return os;
}
friend istream& operator>>(istream& is, Complex &a) {
    is >> a.re >> a.im;
    return is;
}
int main() { Complex d;

    cin >> d;

    cout << d;
}
```

- Works fine with `friend` functions

Guidelines for Operator Overloading

- Use *global function* when encapsulation is not a concern. For example, using `struct String { char* str; }` to wrap a C-string and overload `operator+` to concatenate strings and build a String algebra
- Use *member function* when the left operand is necessarily an object of a class where the operator function is a member. E.g., `operator=`, `operator new` etc. must be member functions
- Use `friend` *function*, otherwise for operators like `<<`, `>>`, relational (`<`, `>`, `==`, `!=`, `<=`, `>=`) should be overloaded through `friend`
- While overloading an operator, try to *preserve its natural semantics* for built-in types as much as possible. For example, `operator+` in a `Set` class should compute union and NOT intersection
- Usually stick to the *parameter passing* conventions (built-in types by value and UDT's by constant reference)
- Decide on the *return type* based on the natural semantics for built-in types as illustrated in the examples
- Only overload the operators that you may need (*minimal design*)

Module Summary

- Several issues in operator overloading has been discussed
- Use of `friend` is illustrated in versatile forms of overloading with examples
- Discussed the overloading IO (streaming) operators
- Guidelines for operator overloading is summarized
- Use operator overloading to build algebra for:
  - Complex numbers
  - Fractions
  - Strings
  - Vector and Matrices
  - Sets
  - and so on ...