CSE1121: Structured & OOP Language

Sumaya Kazary
Assistant Professor
Department of Computer Science and Engineering
DUET, Gazipur.

Acknowledgement
Thanks to the authors of all the books and online tutorials used in this slide.

A CLOSER LOOK AT CLASSES

Ref: Ch-3, Teach yourself C++

ASSIGNING OBJECTS

- One object can be assigned to another provided that both objects are of the same type.
- By default, when one object is assigned to another, a bitwise copy of all the data members is made. Including compound data structures like arrays.
- Creates problem when member variables point to dynamically allocated memory and destructors are used to free that memory.
- Solution: **Copy constructor** (to be discussed in Next Ch.)
- **Example:** All examples attached with the topic.

Example……….

```
class myclass {
                    int i;
   public:
          void set_i(int n) { i=n; }
          int get_i() { return i; }
};
int  main( ){
   myclass ob1, ob2;
   ob1.set_i(99);
   ob2 = ob1;          // assign data from ob1 to ob2
    cout << "This is ob2's i: " << ob2.get_i();
return 0;
}
```

This is ob2's i: 99

Department of CSE,DUET

Is It Correct?????Example……….

```
class myClass {
            int i;
   public:
        void set_i(int n) { i=n; }
        int get_i() { return i; }
};
```

```
class yourClass {
        int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};
```

It is not sufficient that the types just be physically similar

their type names must be the same.

Department of CSE,DUET

PASSING OBJECTS TO FUNCTIONS

- Objects can be passed to functions as arguments in just the same way that other types of data are passed.
- By default all objects are passed by value to a function.
- Address of an object can be sent to a function to implement call by reference.
- **In call by reference, as no new objects are formed, constructors and destructors are not called.**
- In case of call by value, while making a copy, constructors are not called for the copy but destructors are called.
- Can this cause any problem in any case?
- Yes. Solution: **Copy constructor** (discussed later)
- **Example**: All examples in Book.

Department of CSE,DUET

Example ~~~~~~

```
class myclass {
        int i;
public:
        myclass(int n);
        ~myclass();
        void set_i(int n) { i=n; }
        int get_i() { return i; }
};
myclass::myclass(int n){
        i = n;
        cout << "Constructing " << i << "\n";
}
myclass::~myclass(){
cout << "Destroying " << i << "\n";
}
```

```
void f(myclass ob) {
    ob.set_i(2);
    cout << "This is local i: " << ob.get_i()<<endl;
}
```

```
void main(){
    myclass o(1);
    f(o);
    cout << "This is i in main: ";
    cout << o.get_i() << "\n";
}
```

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

Department of CSE,DUET

Example ~~~~~~

```
class myclass {
        int i;
public:
        myclass(int n);
        ~myclass();
        void set_i(int n) { i=n; }
        int get_i() { return i; }
};
myclass::myclass(int n){
        i = n;
        cout << "Constructing " << i <<
  "\n";
}
```

```
void f(myclass ob) {
   ob.set_i(2);
   cout << "This is local i: " << ob.get_i()<<endl;
}
```

```
void main(){
   myclass o(1);
   f(o);
   cout << "This is i in main: ";
   cout << o.get_i() << "\n";
}
```

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

LOOK at HERE: the copy of **o** (in **main()**) is passed to **ob** (within **f()**). That's why the constructor isn't called !!!!!!!!!!!! THINK !!!

Example ~~~~~~

```
class myclass {
        int i;
public:
        myclass(int n);
        ~myclass();
        void set_i(int n) { i=n; }
        int get_i() { return i; }
};
myclass::myclass(int n){
        i = n;
        cout << "Constructing " << i <
    "\n";
}
myclass::~myclass(){
```

```
void f(myclass ob) {
    ob.set_i(2);
    cout << "This is local i: " <<
ob.get_i()<<endl;
}
```

```
void main(){
    myclass o(1);
    f(o);
    cout << "This is i in main: ";
    cout << o.get_i() << "\n";
}
```

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

LOOK at HERE: It is necessary to call the destructor when the copy is destroyed.

SUMMARY: Passing Objects To Functions

- When a copy of an object is generated because it is passed to a function, the object's constructor function is not called.
- BUT, when the copy of the object inside the function is destroyed, its destructor function is called.
- By default, when a copy of an object is made, a bitwise copy occurs. This means that the new object is an exact duplicate of the original.

RETURNING OBJECTS FROM FUNCTIONS

- A function may return an object to the caller.
  - The function must be declared as returning a class type.
  - Return an object of that type using **return** statement.
- When an object is returned by a function, a temporary object (**invisible to us**) is automatically created which holds the return value.
- While making a copy, constructors are not called for the copy but destructors are called
- **After the value has been returned, this object is destroyed.**
- The destruction of this temporary object might cause unexpected side effects in some situations. [for memory allocated object]
  - Solution: **Copy constructor** (to be discussed later)
- **Example:** All examples attached with the topic.

Example…………

```
class myclass {
        int i;
public:
   void set_i(int n) { i=n; }
   int get_i() { return i; }
};
myclass f(); // return object of type myclass
void main(){
   myclass o;
   o = f();
   cout << o.get_i() << "\n";
}
myclass f(){
   myclass x;
   x.set_i(1);
   return x;}
```

FRIEND FUNCTIONS

- A friend function is a non-member function of a class but still has access to its private elements.

- A friend function can be
  - A global function not related to any particular class
  - A member function of another class

- To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**.

- **Why friend functions ?**
  - Operator overloading
  - Certain types of I/O operations
  - Permitting one function to have access to the private members of two or more different classes

FRIEND FUNCTIONS-example

```
class MyClass
{
  int a; // private member
public:
  MyClass(int n) {    a = n;   }
  friend void f1(MyClass obj);
};
```

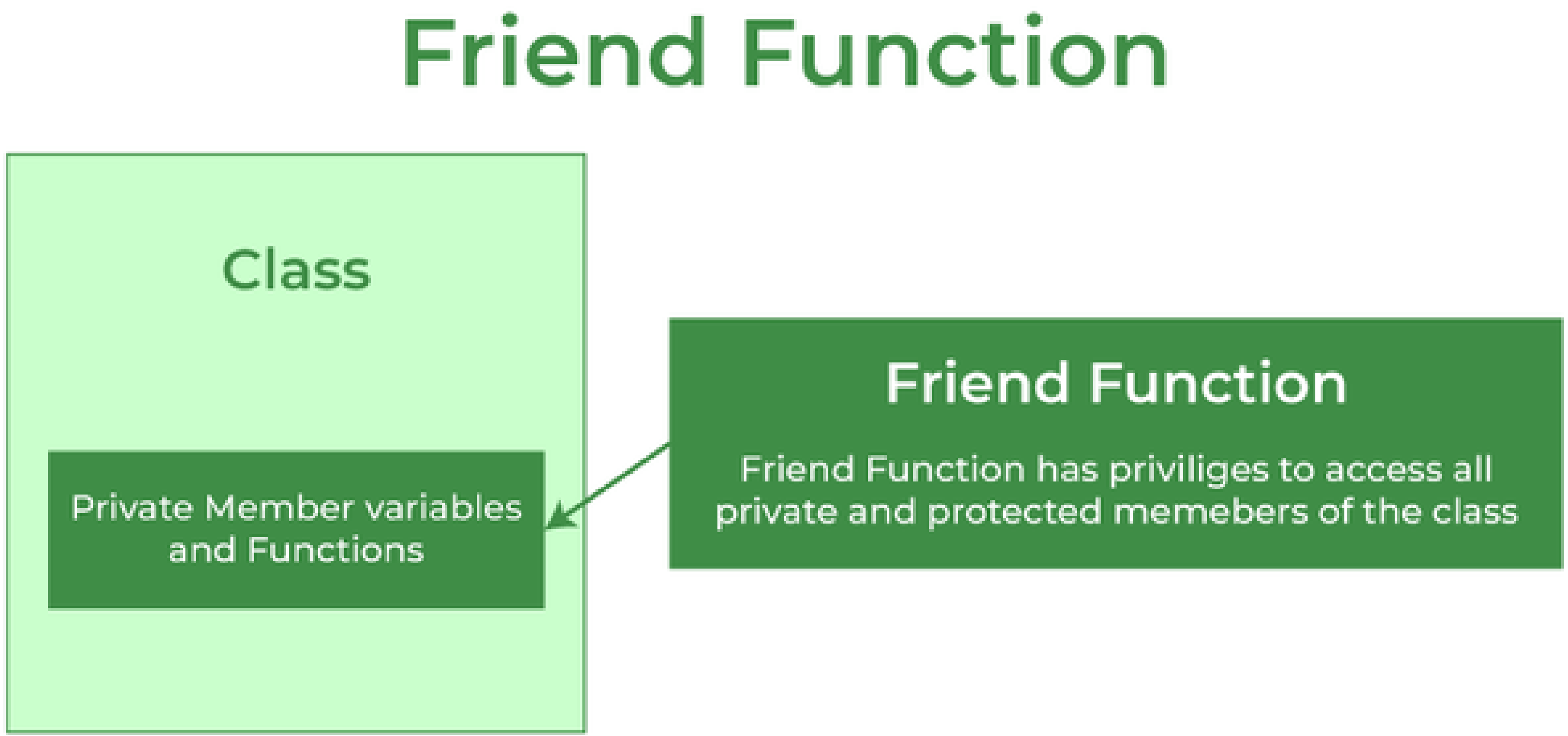
```
// friend keyword not used
void f1(MyClass obj) {
  cout << obj.a << endl;
   // can access private
   member ‘a’ directly
  MyClass obj2(100);
  cout << obj2.a << endl;  }
```

```
void main()
 {
   MyClass o1(10);
   f1(o1);
}
```

Department of CSE, DUET

FRIEND FUNCTIONS

- The concepts of encapsulation and data hiding dictate that **nonmember functions should not be able to access an object's private or protected data**.
  - The policy is, if you're not a member, you can't get in.
- However, there are situations where such rigid intolerance leads to considerable inconvenience.
- Friend functions need to access the members (private, public or protected) of a class **through** an object of that class.
- The object can be declared within or passed to the friend function.
  - **NOTE:** A member function can directly access class members.
- A function can be a member of one class and a friend of another.
- **Example : All examples in this topic .**



- A **friend function** can access the **private** and **protected** data of a class.

Friend Functions

```
class YourClass; // a forward declaration
class MyClass {
            int a; // private member
public:
   MyClass(int n) { a = n; }
   friend int compare (MyClass obj1, YourClass
    obj2);
};
class YourClass {
            int a; // private member
public:
   YourClass(int n) { a = n; }
   friend int compare (MyClass obj1, YourClass
    obj2);};
```

```
int compare (MyClass obj1, YourClass obj2) {
   return (obj1.a – obj2.a);  }
void main() {
   MyClass o1(10); YourClass o2(5);
   int x = compare(o1, o2); // x = 5   }
```

Friend Functions

```
class YourClass; // a forward declaration
class MyClass {
   int a; // private member
public:
   MyClass(int n) { a = n; }
   int compare (YourClass obj)
    {  return (a – obj.a)   }
};
```

```
class YourClass {    int a; // private member
public:       YourClass(int n) { a = n; }
   friend int MyClass::compare (YourClass obj);
};
void main() {
   MyClass o1(10);  Yourclass o2(5);
   int n = o1.compare(o2); // n = 5 }
```

A function can be a member of one class and a friend of another.

CONVERSION FUNCTION

- Used to convert an object of one type into an object of another type.

- *A conversion function automatically converts an object into a value that is compatible with the type of the expression in which the object is used.*

- **General form:** ***operator type() {return value;}***
  - ***type*** is the target type and
  - *value* is the value of the object after conversion.

- No parameter can be specified.

- Must be a member of the class for which it performs the conversion.

- **Examples**: From Book.

Conversion Function

6

112

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

class coord
{
   int x, y;
public:
coord(int i, int j){ x = i; y = j; }
operator int() { return x*y; }
};
```

```
void main
{
   coord o1(2, 3), o2(4, 3);
   int i;

   i = o1;
   // automatically converts to integer
   cout << i << '\n';

   i = 100 + o2;
   // automatically converts to integer
   cout << i << '\n';
}
```

STATIC CLASS MEMBERS

- A class member can be declared as ***static***
- Only one copy of a ***static*** variable exists – no matter how many objects of the class are created
  - All objects share the same variable
- It can be private, protected or public
- A ***static*** member variable <u>exists before any object of its class is created</u>
- In essence, a ***static*** class member is a global variable that simply has its <u>**scope** restricted to the class in which it is declared</u>

STATIC CLASS MEMBERS

- When we declare a ***static*** data member within a class, we are not defining it
  - Instead, we must provide a definition for it elsewhere, outside the class
- To do this, we re-declare the ***static*** variable, using the scope resolution operator to identify which class it belongs to
- All ***static*** member variables are initialized to **0** by default

STATIC CLASS MEMBERS

- The principal reason ***static*** member variables are supported by C++ is to avoid the need for global variables

- Member functions can also be ***static***
  - Can access only other ***static*** members of its class directly
  - Need to access ***non-static*** members through an object of the class
  - Does not have a ***this*** pointer
  - Cannot be declared as ***virtual***, ***const*** or ***volatile***

- ***static member functions*** can be accessed through an object of the class or can be accessed independent of any object, via the class name and the scope resolution operator
  - Usual access rules apply for all ***static*** members

- Example: static.cpp

STATIC CLASS MEMBERS

```
class shared {
    static int a;
    int b;
public:
    void set(int i, int j) {a=i; b=j;}
    void show();
} ;
int shared::a; // define a
void shared::show(){
    cout << "This is static a: " << a;
    cout << "\nThis is non-static b: "
    << b;
    cout << "\n";
}
```

```
void main(){
        shared x, y;
        x.set(1, 1); // set a to 1
        x.show();
        y.set(2, 2); // change a to
2
        y.show();
       x.show();
}
```

This program displays the following output when run.

This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1

Here, a has been changed for both x and y because a is shared by both objects.

STATIC CLASS MEMBERS

```
class myclass {
    static int x;
public:
    static int y;
    int getX() { return x; }
    void setX(int x) {
        myclass::x = x;
    }
};
int myclass::x = 1;
int myclass::y = 2;
```

```
void main ( ) {
    myclass ob1, ob2;
    cout << ob1.getX() << endl; // 1
    ob2.setX(5);
    cout << ob1.getX() << endl; // 5
    cout << ob1.y << endl; // 2
    myclass::y = 10;
    cout << ob2.y << endl; // 10
    // myclass::x = 100;
    // will produce compiler error
}
```

CONST MEMBER FUNCTIONS AND *MUTABLE*

- When a class member is declared as *const* it can't modify the object that invokes it.
- A *const* object can't invoke a non-*const* member function.
- But a *const* member function can be called by either *const* or non-*const* objects.
- If you want a *const* member function to modify one or more member of a class but you don't want the function to be able to modify any of its other members, you can do this using *mutable*.
- *mutable* members can modified by a *const* member function.
- Examples: From Book.

LECTURE CONTENTS

- **Teach Yourself C++**
  - Chapter 3 (Full, with exercises)
  - Chapter 13 (13.2,13.3 and 13.4)