Advanced Programming Lab
CS6150

Week 5

Polymorphism

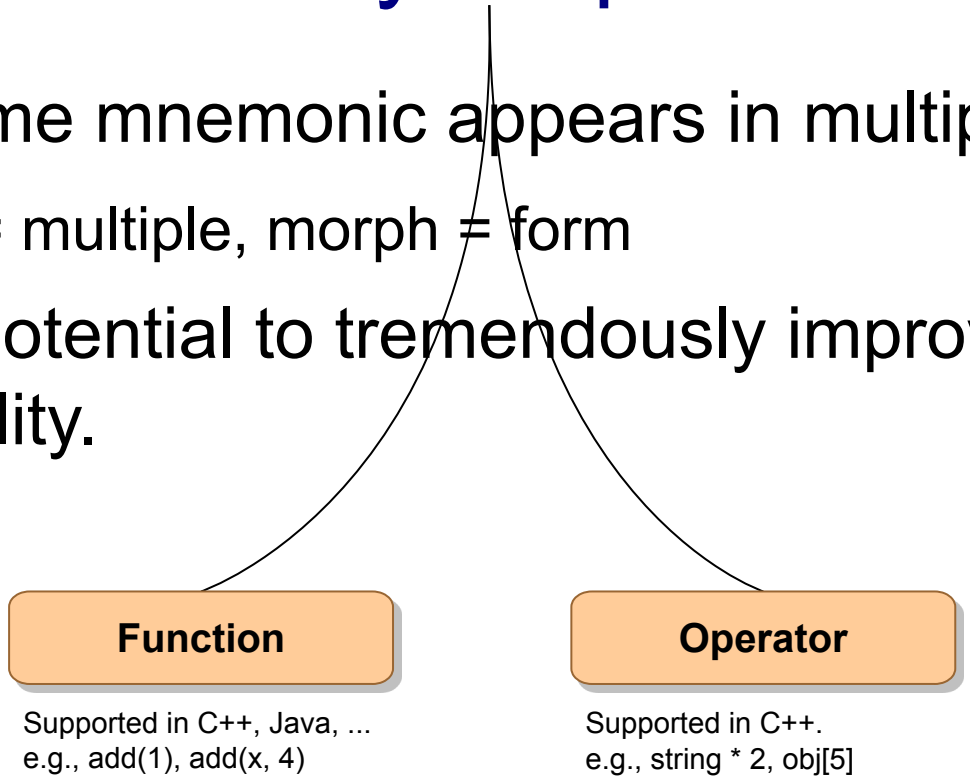
(Slides Courtesy : Rupesh Nasre)

Recap : Classes and Objects

- A class is a type
- An object is its instance
- A class declaration may have Methods and Variables.
- The class content may be public, private, or protected.
- Inheritance allows code reuse and generalization

Polymorphism

- The same mnemonic appears in multiple forms:
  - Poly = multiple, morph = form
- Bears potential to tremendously improve code readability.



Function Overloading : (without classes)

- A function can be re-implemented in many ways depending on the type / number of arguments.

- Cannot have two variations of the function that takes the same parameters in the same order.

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

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Function to add two strings
string add(string x, string y) {
    int i;
    string concat = x;
    for (i = 0; y[i] != '\0'; i++) {
        concat = concat + y[i];
    }
    return concat;
}

// Function to add three integers
int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    cout << "add(int, int): " << add(3, 4) << endl;        // calls first function
    cout << "add(string, string): " << add("Hello ", "World") << endl; // calls second function
    cout << "add(int, int, int): " << add(1, 2, 3) << endl;   // calls third function
    return 0;
}
```

Function Overloading and Inheritance

- A function in the base class can be re-implemented in the derived class.
- derived.method() calls the overloaded function.
- base.method() calls the base class method, provided base is not a derived class object.

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

class Base {
public:
    void display(string x) {
        cout << "Base display with string: " << x << endl;
    }
};

class Derived : public Base {
public:
    // Overload display() in derived class
    void display(double y) {
        cout << "Derived display with double: " << y << endl;
    }
};

int main() {
    Derived obj;
    obj.display("5");
    return 0;
}
```

Compile Time Error

Function Overloading and Inheritance

- A function in the base class can be re-implemented in the derived class.
- derived.method() calls the overloaded function.
- base.method() calls the base class method, provided base is not a derived class object.

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

class Base {
public:
    void display(string x) {
        cout << "Base display with string: " << x << endl;
    }
};

class Derived : public Base {
public:
    // Overload display() in derived class
    void display(double y) {
        cout << "Derived display with double: " << y << endl;
    }
};

int main() {
    Derived obj;
    obj.display(5);
    return 0;
}
```

This is OK

Function Overloading and Inheritance

- How to access base.method() from a derived class?
- Declare them explicitly
- Once for every method

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

class Base {
public:
    void display(int x) {
        cout << "Base display with int: " << x << endl;
    }
};


class Derived : public Base {
public:
    using Base::display; // Bring all overloads of display from Base into scope
    void display(double y) {
        cout << "Derived display with double: " << y << endl;
    }
};

int main() {
    Derived obj;
    obj.display(5);     // Calls Base::display(int)
    obj.display(5.5);   // Calls Derived::display(double)
    return 0;
}
```

Function Overloading and Inheritance

- A derived class can redefine a method from the base class.
- If their signatures are the same, derived class method hides the base class method.
- A base class pointer calls the base method, while a derived class pointer calls the derived method.
- A base pointer pointing to derived class calls the base method.

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

class Base {
public:
    void showMessage() {
        cout << "Message from Base class" << endl;
    }
};

class Derived : public Base {
public:
    // This redefines (overrides) the Base class method
    void showMessage() {
        cout << "Message from Derived class" << endl;
    }
};

int main() {
    Base b;
    Derived d;

    b.showMessage();  // Calls Base version
    d.showMessage();  // Calls Derived version

    // But if we use a Base pointer pointing to a Derived object:
    Base* ptr = &d;
    ptr->showMessage(); // Still calls Base version (not virtual)

    return 0;
}
```

Function Overloading and Inheritance

- What if we want to invoke derived class method, while using a Base class pointer?

- Use Virtual Methods

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

class Base {
public:
    void showMessage() {
        cout << "Message from Base class" << endl;
    }
};

class Derived : public Base {
public:
    // This redefines (overrides) the Base class method
    void showMessage() {
        cout << "Message from Derived class" << endl;
    }
};

int main() {
    Base b;
    Derived d;

    b.showMessage();  // Calls Base version
    d.showMessage();  // Calls Derived version

    // But if we use a Base pointer pointing to a Derived object:
    Base* ptr = &d;
    ptr->showMessage(); // Still calls Base version (not virtual)

    return 0;
}
```

Virtual Functions

- If a function is virtual in the base class, it indicates that a derived class may want to override it.
- When a virtual method is invoked using a base class pointer, appropriate version of the method is invoked.
- In derived class definition, override is optional but recommended

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

class Base {
public:
    // Virtual function
    virtual void show() {
        cout << "Base class show function" << endl;
    }
};

class Derived : public Base {
public:
    // Override the virtual function
    void show() override {
        cout << "Derived class show function" << endl;
    }
};

int main() {
    Base* basePtr;      // Base class pointer
    Derived derivedObj; // Derived class object

    basePtr = &derivedObj;  // Base pointer points to derived object

    basePtr->show();   // Calls Derived's show() due to virtual function

    return 0;
}
```

Virtual Methods

- A virtual method declared in the base class makes the method virtual in base class, all the classes transitively derived from it.

- Signature must exactly match in derived classes
  - Otherwise, it's considered overloading, not overriding.

- Redefinition is optional in derived classes, but the base class declaration is mandatory.

- Constructors cannot be virtual.
- Destructors should be virtual, unless a class is not going to be used as a base class.
  - Otherwise only base destructor will be called

Why is this useful?

- Consider a hierarchy of students
  - Student ← IITM_Student ← CSE_Student ← MTech_Student
  - Student *s = new MTech_Student; s->display();
  - Which display() should be called?
  - Most specialized method is called.

Pure Virtual Functions and Abstract Classes

- A virtual function with *no definition* is pure.
- A class with at least one PVF is abstract.
- An abstract class cannot be instantiated.
  - But pointers and references of abstract type can be created and used.
- If a derived class does not implement all PVF, then it also becomes pure.

- A PVF can be specified even with a default definition.

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

// Abstract class with a pure virtual function
class Shape {
public:
    // Pure virtual function makes this class abstract
    virtual void draw() = 0;

    void description() {
        cout << "This is a shape." << endl;
    }
};

// Derived class must override the pure virtual function
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle." << endl;
    }
};

int main() {
    // Shape s; // Error! Cannot instantiate an abstract class

    Circle c;
    c.description();  // Calls base class non-virtual method
    c.draw();         // Calls overridden method in derived class

    Shape* shapePtr = &c;
    shapePtr->draw(); // Calls Circle's draw() due to virtual dispatch

    return 0;
}
```

Pure Virtual Functions and Abstract Classes

- Pure virtual functions do not contain any code in the base class; only the signature exists.
- Any class with one or more pure virtual functions becomes an abstract class, which cannot be instantiated (objects cannot be created directly from it).
- All derived classes must provide their own definitions for all pure virtual functions; otherwise, they themselves become abstract.
- Commonly used to enforce a contract where derived classes must implement essential functions, much like interfaces in other languages.

Overloading v/s Overriding

- Overloading lets you have the same function name multiple times in a class but with different parameter sets.
  - Can be disambiguated at compile time.

- Overriding means a derived class re-implements a base class function with the same signature to give it new behavior.
  - Function to call decision is based on the actual object type.

Operator Overloading

- Just like functions, operators can also be overloaded.
  - Example: ab + ba = abba
- Operator overloading lets you define functions that provide new meanings to operators when used with user-defined types.
- You create a special function with the keyword `operator` followed by the operator symbol you want to overload.
- Two ways to define:
  - As a member of the class
    - Define it in the left class
  - As a friend

Overloading +

(As a member of the class)

- How could we access other.real / other.imag inside the operator+ ?
- In C++, member functions have access to private members of all instances of the class, not just the current (`this`) object.

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

class Complex {
private:
    float real, imag;

public:
    // Constructor
    Complex(float r = 0, float i = 0) : real(r), imag(i) {}

    // Overload '+' operator as a member function
    Complex operator+( Complex& other)  {
        return Complex(real + other.real, imag + other.imag);
    }

    void display()  {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3.5, 2.5);
    Complex c2(1.5, 4.5);

    Complex sum = c1 + c2;  // '+' operator overloaded
    sum.display();          // Output: 5 + 7i

    return 0;
}
```

Overloading +

(As a friend)

- Typically used with the operator takes two different classes as parameters

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

class Celsius;

class Fahrenheit {
    double degreesF;
public:
    Fahrenheit(double f = 0.0) : degreesF(f) {}
    double getF() const { return degreesF; }
    // Declare friend operator+ to access private members
    friend Celsius operator+(Fahrenheit& f,  Celsius& c);
};

class Celsius {
    double degreesC;
public:
    Celsius(double c = 0.0) : degreesC(c) {}
    double getC() { return degreesC; }
    void display() {
        cout << degreesC << " °C" << endl;
    }
    // Declare friend operator+ to access private members
    friend Celsius operator+( Fahrenheit& f,  Celsius& c);
};

double fahrenheitToCelsius(double f) {
    return (f - 32) * 5.0 / 9.0;
}

// Overloaded + operator to add Fahrenheit and Celsius temps (converted to Celsius)
Celsius operator+(const Fahrenheit& f, const Celsius& c) {
    double celsiusFromF = fahrenheitToCelsius(f.degreesF);
    return Celsius(celsiusFromF + c.degreesC);
}

int main() {
    Fahrenheit fTemp(98.6);
    Celsius cTemp(37);

    Celsius result = fTemp + cTemp;  // Uses overloaded operator+
    cout << "Sum of temperatures in Celsius: ";
    result.display();

    return 0;
}
```

Rules for Operator Overloading

- Must be overloaded for a user-defined class.
- Operator associativity remains the same.
- Operator precedence remains the same.
- Cannot define a new symbol as operator.

Overloading Unary Operator

(As a member of the class)

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

class Point {
private:
    int x, y;

public:
    // Constructor
    Point(int xVal = 0, int yVal = 0) : x(xVal), y(yVal) {}

    // Overload unary minus (-) operator
    Point operator-()  {
        return Point(-x, -y); // Negates both coordinates
    }

    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p1(5, -3);

    cout << "Original point: ";
    p1.display();

    Point p2 = -p1;  // Calls overloaded unary operator-
    cout << "After applying unary - : ";
    p2.display();

    return 0;
}
```

Overloading Input and Output

- Name = “Alice”;
- cout<<name;

- Prints the value of name (Alice)

- Can we have:
  - Date as a class that contains day, month and year
  - cout<<date;
    - Prints the dd/mm/yyyy
  - cin>>date;
    - Takes dd, mm and yyyy as inputs with appropriate message

- Possible by overloading << and >> operators

Overloading Input and Output

- ostream class is used to write output data to devices like the console or files.
  - The common instance is `std::cout`
  - Other instances include `std::cerr, std::clog …`

- istream class is used to read input data from devices like the keyboard or files.
  - The common instance is `std::cin`.
  - Other instances include `std::ifstream`

- Here `std` is the namespace

Overloading <<

- Overload << as a non-member friend function that takes an ostream& and a reference to the class object.
- Allows objects of the class to be used with output stream.
- The function returns ostream& so that multiple insertions can be chained.

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

class Date {
    int month, day, year;

public:
    Date(int m, int d, int y) : month(m), day(d), year(y) {}

    // Declare friend function to overload <<
    friend ostream& operator<<(ostream& os, const Date& dt);
};

// Definition of overloaded << operator
ostream& operator<<(ostream& os, const Date& dt) {
    os << dt.month << '/' << dt.day << '/' << dt.year;
    return os;  // Return ostream to allow chaining
}

int main() {
    Date date(8, 14, 2025);
    cout << "Today's date is: " << date << endl;
    return 0;
}
```

Overloading >> for cin

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

class Date {
    int day, month, year;

public:
    Date() : day(1), month(1), year(2000) {}

    // Friend function to overload >>
    friend istream& operator>>(istream& is, Date& dt);
    // Friend function to overload <<
    friend ostream& operator<<(ostream& os, const Date& dt);
};

// Overload extraction operator >>
istream& operator>>(istream& is, Date& dt) {
    cout << "Enter day month year: ";
    is >> dt.day >> dt.month >> dt.year;
    return is; // Return stream for chaining
}

// Overload insertion operator << (for display)
ostream& operator<<(ostream& os, const Date& dt) {
    os << dt.day << "/" << dt.month << "/" << dt.year;
    return os;
}

int main() {
    Date d;

    // Using overloaded >>
    cin >> d;

    // Using overloaded <<
    cout << "You entered: " << d << endl;

    return 0;
}
```

Non-overloadable Operators

- **.** member operator
- **.*** pointer to member operator

- **?:** ternary conditional operator
- **::** scope resolution operator
- **sizeof** data size operator
- **typeid** data type operator

Other Overloadable Operators

- New
- Delete
- [ ]
- ->

See you in the lab on Friday

Try out examples

Practise problems will be available by tomorrow