

Data Structures & Algorithms in C++

WEEK 02

Object-Oriented Design

- Inheritance and Polymorphism
- Templates
- Exceptions

Procedure-oriented vs. Object-oriented

➤ Comparison between POD and OOD

Feature	Procedure-Oriented Design (POD)	Object-Oriented Design (OOD)
Focus	Functions or procedures	Objects
Modularity	Divides program into small functions	Organizes program into objects
Data	Data is secondary to functions	Data and behavior are encapsulated
Approach	Top-down approach	Bottom-up approach
Examples	Languages like C, Pascal, Fortran	Languages like Java, C++, Python
Design Philosophy	Emphasizes functions and action sequences	Emphasizes objects and interactions
Data Handling	Data is passed around between functions	Data and methods are within objects
Reusability	Functions can be reused	Objects and classes are highly reusable
Maintenance	Can be challenging to maintain as size grows	Easier to maintain due to encapsulation

OOD Goals

➤ Objects

- Main “actors” in the object-oriented design paradigm
- Instances of the class including data members and member functions

➤ Goals

- **Robustness** : capability of handling unexpected inputs that are not explicitly defined for its application (more importantly in life-critical applications)
- **Adaptability** : ability of software to run with minimal change on different H/W and OS platforms (Web browsers and Internet search engines)
- **Reusability** : reusing of the same code can reduce the cost of developing high-quality software

OOD Principles

- ① Abstraction

➤ Abstraction

- Simplifies a complicated system by focusing on its **fundamental parts**
- Involves :
 - **Naming** system components
 - **Explaining their functionality** in simple, precise terms

➤ Abstract Data Types (ADTs)

- A mathematical model of a data structure
- Specifies :
 - **Data types** stored
 - **Supported operations** on the data
 - **Parameter types** for the operations
- ADTs define **what the operations do**, not how they do it

OOD Principles

- ① Abstraction

➤ Abstraction in C++

- Achieved through the **public interface** of a class :
 - **Signatures** : Names, return types, argument types of public member functions
 - The **only accessible part** of a class for its users
- ADT is modeled in C++ by a class; classes specify **how the operations are performed** in the body of each function

OOD Principles

- ① Abstraction

➤ Example

```
class Shape { // Abstract Base Class
public:
    virtual double getArea() const = 0;
};

class Circle : public Shape { // Derived Class: Circle
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getArea() const override {
        return M_PI * radius * radius; // Area of a circle
    }
};
```

```
class Rectangle : public Shape { // Derived Class: Rectangle
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() const override {
        return width * height; // Area of a rectangle
    }
};

int main() {
    // Create objects of Circle and Rectangle
    Shape* circle = new Circle(5.0);
    Shape* rectangle = new Rectangle(4.0, 6.0);
    return 0;
}
```

OOD Principles

- ② Encapsulation

➤ Encapsulation

- Ensure that **internal details** of a component or system are **hidden**
- Only necessary information is exposed through an **abstract interface**

➤ Advantages

- Freedom to modify the internal implementation
- Change to implementation are localized, reducing system-wide impacts
- Protected from unintended interference

➤ Constraints

- Maintain the **abstract interface** visible to outside components
- Ensure **consistency** in interactions

OOD Principles

- ② Encapsulation

➤ Example

```
class EncapsulatedClass {  
    private:  
        int hiddenData; // Internal detail  
  
    public:  
        // Abstract interface  
        void setData(int value) {  
            hiddenData = value; // Controls access  
        }  
  
        int getData() const {  
            return hiddenData; // Provides controlled access  
        }  
};
```

OOD Principles

- ③ Modularity

➤ Modularity

- Divide a software system into separate, function units
- Enhance clarity and maintainability by keeping components well-organized

➤ Importance of Modularity

- Ensure that components in complex systems interact properly
- Keep components logically divided, simplifying understanding and debugging

➤ Benefits

- Improved maintainability
- Code reusability
- Scalability

Principles

- ③ Modularity

➤ Example

// Module 1: Class for User Authentication

```
class AuthModule {
```

```
public:
```

```
    void authenticateUser() {  
        cout << "User authenticated." << endl;  
    }  
};
```

// Module 2: Class for Database Operations

```
class DatabaseModule {
```

```
public:
```

```
    void fetchData() {  
        cout << "Data fetched from the database." << endl;  
    }  
};
```

// Main Application

```
int main() {
```

```
    AuthModule auth;  
    DatabaseModule db;
```

```
    auth.authenticateUser();  
    db.fetchData();
```

```
    return 0;
```

```
}
```

Principles

- Hierarchical Organization

➤ Hierarchical Organization

- A tree-like structure : base classes (parents), derived classes (children)
- Derived classes inherit properties and behaviors from their base classes

➤ Characteristics

- **Generalization** at higher levels
 - : base classes define broad, generic behaviors and attributes
- **Specialization** at lower levels
 - : derived classes provide more specific implementations or extend functionality

➤ Benefits

- Code reusability, Modularity, Ease of Maintenance, Extensibility

Design Patterns in OOD

- Advantages of OOD
 - Reusable, robust, and adaptable software
- Design Patterns
 - Describe a solution to a “**typical**” software design problem
 - Provide a general template for solutions in various situations
 - Elements
 - **Name** : identifies design pattern
 - **Context** : describes scenarios for pattern application
 - **Template** : describes how the pattern is applied
 - **Result** : describes and analyzes what the pattern produces

Design Patterns in OOD

➤ Algorithm Design Patterns

- Recursion
- Amortization
- Divide-and-conquer
- Prune-and-search (decrease-and-conquer)
- Brute force
- The greedy method
- Dynamic programming

➤ Software Engineering Patterns

- Position
- Adapter
- Iterator
- Template method
- Composition
- Comparator
- Decorator

Inheritance

➤ Object-Oriented Paradigm

- Provides a modular and hierarchical organizing structure

➤ Inheritance

- Specialized classes reuse the code from the generic class
(*derived class, child class, subclass*) (base class , *parent class, superclass*)
- Specialized class inherits general functions from the base class, and only defines the specialized functions

```
class Person {                                // Person (base class)
private:
    string    name;                          // name
    string    idNum;                         // university ID number
public:
    // ...
    void print();                            // print information
    string getName();                        // retrieve name
};
```

```
class Student : public Person {              // Student (derived from Person)
private:
    string    major;                        // major subject
    int       gradYear;                     // graduation year
public:
    // ...
    void print();                           // print information
    void changeMajor(const string& newMajor); // change major
};
```

Inheritance

➤ Public Members

- An object of derived class can access the public members of both base and derived classes. However, an object of base class cannot access members of the derived class

```
Person person("Mary", "12-345");    // declare a Person
Student student("Bob", "98-764", "Math", 2012); // declare a Student
```

```
cout << student.getName() << endl; // invokes Person::getName()
person.print();                     // invokes Person::print()
student.print();                    // invokes Student::print()
person.changeMajor("Physics");      // ERROR!
student.changeMajor("English");     // okay
```

➤ Protected Members

- “public” to all classes derived from this base class, but “private” to all other functions

Inheritance

- Illustrating Class Protection
 - **Protected** members : allow derived classes to access and modify certain members
 - **Private** members : used to encapsulate data and methods, not to accessed or modified directly from outside the class

```
class Base {
    private:    int priv;
    protected: int prot;
    public:     int publ;
};

class Derived: public Base {
    void someMemberFunction() {
        cout << priv;           // ERROR: private member
        cout << prot;           // okay
        cout << publ;           // okay
    }
};
```

```
class Unrelated {
    Base X;

    void anotherMemberFunction() {
        cout << X.priv;          // ERROR: private member
        cout << X.prot;          // ERROR: protected member
        cout << X.publ;          // okay
    }
};
```

Inheritance

➤ Constructors and Destructors

- Bottom-up construction : base class first, then its members, then the derived class

```
Person::Person(const string& nm, const string& id)
: name(nm),                // initialize name
  idNum(id) { }            // initialize ID number
```

```
Student::Student(const string& nm, const string& id,
                 const string& maj, int year)
: Person(nm, id),           // initialize Person members
  major(maj),              // initialize major
  gradYear(year) { }       // initialize graduation year
```

- Destruction in the reverse order : the derived classes destroyed before base classes
→ the derived class destructor does not need to call the base class destructor

```
delete s;                  // calls ~Student() then ~Person()
```

Inheritance

➤ Static Binding

- C++'s default action is to consider an object's declared type, not its actual type

```
Person* pp[100];           cout << pp[1]->getName() << '\n'; // okay
pp[0] = new Person(...);   pp[0]->print();           // calls Person::print()
pp[1] = new Student(...);  pp[1]->print();           // also calls Person::print() (!)
                           pp[1]->changeMajor("English"); // ERROR!
```

➤ Dynamic binding and Virtual Functions

- To use dynamic binding, the keyword “virtual” is added to the function's declaration

```
class Person {
    virtual void print() { ... }
    // ...
};
class Student : public Person {
    virtual void print() { ... }
    // ...
};

Person* pp[100];           // array of 100 Person pointers
pp[0] = new Person(...);   // add a Person (details omitted)
pp[1] = new Student(...);  // add a Student (details omitted)
pp[0]->print();             // calls Person::print()
pp[1]->print();             // calls Student::print()

⊗ Virtual destructor
```

Polymorphism

➤ **Polymorphism**

- Allows a variable to take on different types
- Typically applied using pointer variables

➤ **Dynamic binding**

- Both S and T define a virtual member function a
- Invocation `p->a()` uses dynamic binding : `T::a` or `S::a` (`p` : pointer of the object)
- Caller of `p->a()` doesn't need to know if `p` refers to T or S

* Software reuse support : inheritance, polymorphism, function overloading

➤ **Specialization** : override for specialized functions

➤ **Extension** : add new functions

Examples of Inheritance

firstValue(): Reset the progression to the first value and return it.

nextValue(): Step the progression to the next value and return it.

printProgression(*n*): Reset the progression and print its first *n* values.

```
class Progression {                // a generic progression
public:
    Progression(long f = 0)         // constructor
        : first(f), cur(f) { }
    virtual ~Progression() { };    // destructor
    void printProgression(int n);   // print the first n values
protected:
    virtual long firstValue();      // reset
    virtual long nextValue();      // advance
protected:
    long first;                    // first value
    long cur;                      // current value
};
```

```
void Progression::printProgression(int n) { // print n values
    cout << firstValue();                 // print the first
    for (int i = 2; i <= n; i++)          // print 2 through n
        cout << ' ' << nextValue();
    cout << endl;
}

long Progression::firstValue() {          // reset
    cur = first;
    return cur;
}

long Progression::nextValue() {           // advance
    return ++cur;
}
```

Examples of Inheritance

```
class ArithProgression : public Progression { // arithmetic progression
public:
    ArithProgression(long i = 1);           // constructor
protected:
    virtual long nextValue();               // advance
protected:
    long inc;                               // increment
};
```

```
class GeomProgression : public Progression { // geometric progression
public:
    GeomProgression(long b = 2);           // constructor
protected:
    virtual long nextValue();               // advance
protected:
    long base;                              // base value
};
```

```
ArithProgression::ArithProgression(long i) // constructor
: Progression(), inc(i) { }

long ArithProgression::nextValue() {       // advance by adding
    cur += inc;
    return cur;
}
```

```
GeomProgression::GeomProgression(long b) // constructor
: Progression(1), base(b) { }

long GeomProgression::nextValue() {       // advance by multiplying
    cur *= base;
    return cur;
}
```

Examples of Inheritance

```
/** Test program for the progression classes */
int main() {
    Progression* prog;

    // test ArithProgression
    cout << "Arithmetic progression with default increment:\n";
    prog = new ArithProgression();
    prog->printProgression(10);
    cout << "Arithmetic progression with increment 5:\n";
    prog = new ArithProgression(5);
    prog->printProgression(10);

    // test GeomProgression
    cout << "Geometric progression with default base:\n";
    prog = new GeomProgression();
    prog->printProgression(10);
    cout << "Geometric progression with base 3:\n";
    prog = new GeomProgression(3);
    prog->printProgression(10);
}
```

```
Arithmetic progression with default increment:
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45
Geometric progression with default base:
1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683
```

Multiple Inheritance and Class Casting

➤ Multiple Inheritance

- C++ allows deriving a class from multiple base classes
- Useful for defining interfaces, but introduces complexities
- Single inheritance is used almost exclusively to avoid these complications

```
class Car { // Base class: Car
public:
    void drive() {...}
    void park() {...}
};
```

```
class Boat { // Base class: Boat
public:
    void sail() {...}
    void anchor() {...}
};
```

```
class HybridVehicle : public Car, public Boat { // Derived class :
public:
    void hybridMode() {...}
};

int main() {
    HybridVehicle hybrid;
    hybrid.drive(); hybrid.park();
    hybrid.sail();  hybrid.anchor();
    hybrid.hybridMode();
    return 0;
}
```


Multiple Inheritance and Class Casting

➤ Strong Typing

- Enforces that all variables be typed and operations declare the types they expect
- Helps prevent bugs by ensuring correct usage and function behavior

➤ Type Casting

- Sometimes necessary to explicitly change a variable from one type to another
- If an illegal pointer cast is attempted, then the result is a null pointer

```
Person* pp[100];  
pp[0] = new Person(...);  
pp[1] = new Student(...);
```

```
for (int i = 0; i < 100; i++) {  
    Student *sp = dynamic_cast<Student*>(pp[i]);  
    if (sp != NULL)                                // cast succeeded?  
        sp->changeMajor("Undecided");              // change major  
}
```

Interfaces and Abstract Classes

➤ **Abstract Class**

- Used only as a base class for inheritance; cannot be instantiated directly
- Useful for defining generic base classes for related subclasses

➤ **Pure Virtual Function**

- Declared by giving “ = 0” in place of its body
- Base class cannot be instantiated if it has pure virtual functions

➤ **Interfaces and Abstract Base Classes**

- C++ does not provide a direct mechanism for defining interface for ADTs
(cf. Java's interface : a collection of function declarations with no data and no bodies)
- Use abstract base classes with all functions as pure virtual

Interfaces and Abstract Classes

```
class Stack {                                     // stack interface as an abstract class
public:
    virtual bool isEmpty() const = 0;           // is the stack empty?
    virtual void push(int x) = 0;               // push x onto the stack
    virtual int pop() = 0;                      // pop the stack and return result
};

class ConcreteStack : public Stack {             // implements Stack
public:
    virtual bool isEmpty() { ... }              // implementation of members
    virtual void push(int x) { ... }            // ... (details omitted)
    virtual int pop() { ... }
private:
    // ...                                       // member data for the implementation
};
```

Templates

- Function Templates

➤ **Function Template**

- Automatic mechanism in C++ for generic functions
- A well-defined pattern; allows instantiation of a concrete function for a specified type
- Compiler checks the argument types at function call

➤ **Advantages of Function Template**

- Reduce redundancy
- Simplify the codebase
- Easy to scale for multiple data types

Templates

- Function Templates

➤ Example

```
int integerMin(int a, int b)           // returns the minimum of a and b
{ return (a < b ? a : b); }
```

```
template <typename T>
T genericMin(T a, T b) {               // returns the minimum of a and b
    return (a < b ? a : b);
}
```

```
cout << genericMin(3, 4) << ' '      // = genericMin<int>(3,4)
     << genericMin(1.1, 3.1) << ' '  // = genericMin<double>(1.1, 3.1)
     << genericMin('t', 'g') << endl; // = genericMin<char>('t','g')
```

Templates

- Class Templates

➤ **Class Template**

- Allows templating of C++ classes to work with various data types
- Provides a single declaration for multiple types
- Extensively used in the STL (Standard Template Library)

➤ **Benefits** of Class Template

- Code Reusability : define once, apply to various types
- Simplified Syntax : reduces redundancy
- Extensibility : easily adapt to new data types

➤ **Usages** in STL

- `std::vector<T>`
- `std::map<Key, Value>`

Templates

- Class Templates

➤ Example

```
template <typename T>
class BasicVector {
public:
    BasicVector(int capac = 10);
    T& operator[](int i)
        { return a[i]; }
    // ... other public members omitted
private:
    T* a;
    int capacity;
};
```

```
template <typename T> // constructor
BasicVector<T>::BasicVector(int capac) {
    capacity = capac;
    a = new T[capacity]; // allocate array storage
}

BasicVector<int> iv(5); // vector of 5 integers
BasicVector<double> dv(20); // vector of 20 doubles
BasicVector<string> sv(10); // vector of 10 strings
```

Exceptions

➤ Exceptions

- Unexpected events during program execution
- The result of an error condition or simply an unanticipated input

➤ “Thrown” by

- The code itself when encountering a problem
- C++ runtime environment (e.g., out of memory)

➤ Why use Exceptions

- Cleaner, more efficient error management
- Improves code readability and maintainability

Exceptions

➤ Example

```
class MathException {  
public:  
    MathException(const string& err)  
        : errMsg(err) { }  
    string getError() { return errMsg; }  
private:  
    string errMsg;  
};
```

```
class ZeroDivide : public MathException {  
public:  
    ZeroDivide(const string& err)           // divide by zero  
        : MathException(err) { }  
};  
  
class NegativeRoot : public MathException {  
public:  
    NegativeRoot(const string& err)         // negative square root  
        : MathException(err) { }  
};
```

Exceptions

➤ Try-Catch Block

- Try block contains code that may throw exceptions
- Catch blocks handle specific exception types

➤ Benefits of Exception Handling

- Cleaner error management compared to special return values
- Provides flexibility and reusability through catch blocks
- Enables robust application development

Exceptions

➤ Example

```
class MathException {  
  public :  
    MathException(const string& err)  
      : errMsg(err) {}  
    string getError() { return errMsg; }  
  private :  
    string errMsg;  
};  
  
class ZeroDivide : public MathException {  
  public :  
    ZeroDivide(const string& err)  
      : MathException(err) {}  
};
```

```
cout << "Enter two numbers : ";  
cin >> num >> denom;  
  
try {  
  if ( denom == 0 )  
    throw ZeroDivide("Division by Zero in main routine");  
  result = num / denom;  
  cout << "Division result = " << result << endl;  
} catch (ZeroDivide &e) {  
  cerr << "Exception caught : " << e.getError() << endl;  
} catch (...) {  
  cerr << "Unknown exception error " << endl;  
}
```