**Lê Trung Kiên \_ 23127075**

**OOP:** programming paradigm: objects concept:

- data (attributes / properties)

- methods (behavior)

**Class:** definition same kind objects (blueprint/template/prototype)

**OOP > procedural programming:** clear structure, keep DRY "Don't Repeat Yourself", easier to maintain, modify and debug

**Encapsulation:** *bundling data and methods into a single unit (class);*data hiding; access control; abstraction; maintainability & flexibility; data integrity (toàn vẹn) & avoid unintended issues; security

**UML (Unified Modeling Language)** **Class Diagram**: graphical notation construct & visualize object-oriented systems

A diagram of a company structure

Description automatically generated

**Obj lifecycle: Stack:** created statically → out of scope; **Heap:** allocate → ***deallocate*** (release)

**Friend class:**can access private & protected members of other classes **in which it is declared as a friend**

**Friend Function:** access private & protected members of a class: **declare** **global** & **other classes’** functionsas **“friend” inside that class** to access;non-member/ordinary function → can’t be called by object name & dot operator; violates data hiding; can’t do any run-time polymorphism in their members; **Friend:** not mutual, not herrited

**Constructor:** create, initialize objects and their variables: same class name, no return; call automatically (object instantiated); can be overloaded: multiple constructors - different parameter lists

**Default constructor:** no para; implicityly declared if not defined; must be defined when other constructors are defined

**Copy constructor:**

**When called?** Obj passed by valued to function/ returned by value from function/ initialized by another same class object

**Shallow copy:** default copy constructor; copy memory addr; **Dangling pointers:** point to invalid mem location; **Double deletion**

**Deep copy:** objects have pointers/ dynamically allocated; duplicate resources; independent

**Parameterized constructor:** constructor overloading; must define default constructor onced defined

**Overloading:** Many different “definitions” for the same name. Overloading functions are differentiated by their signatures (number/types of arguments). The return type is not considered in differentiating overloading functions.

**Move constructor:** steal resources (transfer ownership)

- avoid unnecessary deep copy (large objects/ dynamic resources) → efficient

- Safe state: after moving, org obj is left in a valid, empty state: nullptr

- no except

**Destructor:** deallocate, clean up

**When called?** Out of scope, delete

- Same name as class with prefix, no return, automatically called, default, 1/ class (not overloaded)

**Input & Output operators**

**friend** istream& operator>>(istream& i, TYPE &obj) {

in >> obj.value;

return i; // for chaining

}

**friend** ostream& operator<<(ostream& out, TYPE &obj) {

out << obj.value;

return out; // for chaining

}

**Arithmetic, bitwise operators**

TYPE operator+(const TYPE& other) const {

return TYPE(value + other.value);

}

TYPE operator%(int mod) const {

return TYPE(value % mod);

}

**Assignment operators**

TYPE& operator+=(const TYPE& other) {

value + other.value;

return \*this;

}

// If there is dynamic variables, clean up first

TYPE& operator=(const TYPE& other) {

If (this == &other)

return \*this;

value = other.value;

return \*this;

}

**Rule of three:** If a class defines any of the following then it should explicitly define all three: (1) destructor, (2) copy constructor, (3) assignment operator

**Relational operators**

bool operator == (const TYPE& other) const {

return value == other.value;

}

bool operator != (const TYPE& other) const {

return !(\*this == other);

}

**Logical operators**

bool operator&&(const TYPE& other) const {

return (value != 0) && (other.value != 0);

}

**Increment operators**

|  |  |
| --- | --- |
| **TYPE&** operator++() {  ++value;  return \*this;  } | **TYPE** operator++(int) {  TYPE tem = \*this;  ++value;  return temp;  } |

**Move assignment operators**

TYPE& operator=(TYPE&& other);

|  |  |
| --- | --- |
| **Cast operator**  operator float() const {  return static\_cast<float> (value);  } | **Index operator**  int& IntArray::operator[](int index) {  return arr[index];  } |

**Inheritance:** mechanism of basing a class upon another class, retaining similar implementation/ deriving new classes (subclasses) from existing ones (super/ base class), forming them into a hierarchy of classes

A table with text on it

Description automatically generated

**Constructor inheritance:** not inherited; called implicitly or explicitly by the child constructor. No constructor in derived class, compiler provides default constructor for derived class, implicitly call the base class's default constructor (if exists). Base class: no default constructor - derived class no constructor → compilation error → explicitly declare a constructor in the derived class & initialize the base class accordingly

**Order of Construction**

(1) Base class and its member constructors are called in the order of declaration (2) Derived class and its member constructors are called in the order of declaration

|  |  |
| --- | --- |
| class C1 {  public: C1() { cout << "C1 ctor;"; }  };  class C2 {  public: C2() { cout << "C2 ctor;"; }  };  class Base {  private: C1 c1;  public: Base() { cout << "Base ctor;";}  }; | class Derived: public Base {  private: C2 c2;  public: Derived(): Base() { cout << "Derived ctor;";}  };  int main() {  Derived d;  }  **The output:**  C1 ctor;Base ctor;C2 ctor;Derived ctor |

**Order of Destruction** Default Destructor X::~X() {}

(1) The class’s destructor is called, and the body of the destructor function is executed

(2) Destructors for nonstatic member objects are called in the reverse order in which they appear in the class declaration

(3) Destructors for non-virtual base classes are called in the reverse order of declaration

(4) Destructors for virtual base classes are called in the reverse order of declaration

(\*) In the main function, the obj declared last will be the first one to be destroyed, and vice versa

**Polymorphism:** allows objects to be treated as instances of their base class rather than their actual class; enable a function to use objects of different classes through a common interface

**Override:** not mandatory; highly recommended, code more readable, help compiler catch errors

**Pure virtual method and Abstract class:** Pure virtual method = virtual method with no implementation (= 0) in base class. ≥ 1 pure virtual method → abstract classes & can’t be instantiated directly (Shape\* shape = new Square(); thì được)

**Virtual destructor:** ensures that the destructor of a derived class is called when deallocating

**Inheritance abuse (lạm dụng):** Creating an abstract base class for classes that should not necessarily share a common interface can lead to unnecessary complexity, use composition instead

**static\_cast:** safer than C-style cast; used for conversions between related types (converting pointers up and down an inheritance hierarchy). If types are incompatible or conversion does not make sense → compiler generate error → prevent potential issues before the program runs

**dynamic\_cast:** primarily for safely downcasting in inheritance hierarchy; checks at runtime whether the cast is valid and returns nullptr if it isn’t

**const\_cast:** add/ remove the ***const qualifier*** from a variable; should beused with caution (modifying a const object results in undefined behavior)

**reinterpret\_cast:** used for low-level casting that may involve converting any pointer type to any other pointer type; don’t check safty → dangerous

**Final:** prevent a class from being inherited; prevent function from being overridden in the derived classes

**Multiple inheritance:** allows a class to inherit from more than one base class

**Name ambiguities:** ≥ 2 base classes have members (variables or methods) with same name. Must be resolved explicitly (derived class) or error

**Diamond problem:**

Data Duplication: A has data members, D will have 2 separate copies of those members, potentially causing A diagram of a diagram

Description automatically generatedissues with data consistency. Solve by virrtual inheritance

Class A {… };

class B, C : virtual public A { … };

Class D : public B, public C { … };

**Multiple inheritance - Alternatives**

Composition (combining objects)/ using interfaces (pure virtual classes) can offer cleaner, more maintainable solutions → evaluate complexity and design implications before using multiple inheritance & prefer simpler alternatives like composition/ interfaces when possible

**Abstraction:** hiding the implementation details of anobject & exposing only the essential features that the user needs to interactwith; achieved by using abstract classes and purevirtual functions

**A blue and white rectangular box with black text

Description automatically generated**

**TEMPLATE**

Templates enable us to implement the function only once to be used for different argument data types

Templates enable C++ compiler work as an interpreter. Programs are interpreted at the compiling time, therefore having results at the compiling time. For example, loops or conditional checking can be replaced by recursive templates.

**Template class:** write generic programs

Template <class T, class U…>

Class MyClass { … };

// tránh compile error

*template* *class* MyVector<*int, float…*>;

**Use:**

MyClass<int, float…> intClass;

**Multiple catch (handlers):** can be chained; each one with a different parametertype. (như switch case)

**Handle any type of exceptions**

catch (char param) { cout << "char exception"; }

catch (...) { cout << "default exception "; }

**Exception specification (legacy)**

double myfunction (char param) throw (int);

→ throws an exception of some type other than int → calls std::unexpected instead of looking for a handler or calling std::terminate

int myfunction (int param) throw(); // all exceptions call unexpected

int myfunction (int param); // normal exception handling

**Standard exceptions:** declare objects to be thrown as exceptions: *std::exception*

- std::exception: base class all standard C++ exceptions

- std::runtime\_error - detectable only at runtime.

- std::logic\_error - detectable before runtime

- Others std::overflow\_error, std::underflow\_error, and std::invalid\_argument

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Exception Safety and Best Practices**

- Catch excpetion **by reference** avoid slicing & unnecessary copying

- Throw by value, catch by reference: When throw, C++ creates a temporary exception object.

- Only use exceptions for exceptional situations

- Use custom exception classes for specific error handling in complex applications.

- noexcept where appropriate → enhance performance

**Object slicing:** derived class object is assigned to a base class object, causing the "slicing off" of any derived-specific attributes/ methods → incomplete/ unexpected behavior, especially when using polymorphism → solve by passig by reference

**Upcasting:** converting a derived class reference or pointer to a base class. This is normal practice when using dynamic binding

**Downcasting:** converting a base class pointer to a derived class pointer. Explicit type cast is required when using downcasting

**DYNAMIC BINDING:** In the derived class, if the virtual function is not override, compiler will look for the latest definition of this function in the inheritance hierarchical structure

**DESIGN PATTERNS**

***SINGLETON:*** *creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.*

class C {

private:

static C\* instance;

C() { cout << "C created.\n"; }

public:

static C\* getInstance() {

if (instance == NULL) { instance = new C(); }

return instance; }

static void delInstance() {

if (instance) {

delete instance;

instance = NULL;

}

}

};

***Composite****: structural pattern lets you compose objects into tree structures and then work with these structures as if they were individual objects*

***Strategy*** *behavioral pattern lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable*

***Prototype****creational pattern lets you copy existing objects without making your code dependent on their classes.Ư*

**Factory Method** is a creational pattern provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created

**Abstract Factory** creational pattern lets you produce families of related objects without specifying their concrete classes

**Builder** creational pattern lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code

**Adapter** structural pattern allows objects with incompatible interfaces to collaborate

**Bridge** structural pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other

**Decorator** structural pattern lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

**Facade** structural pattern provides a simplified interface to a library, a framework, or any other complex set of classes.

**Flyweight** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

**Proxy** structural pattern lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

**Template Method** behavioral pattern defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

**Visitor** behavioral design pattern lets you separate algorithms from the objects on which they operate.

**Iterator** behavioral pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

**Observer** behavioral design that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

**Chain of Responsibility** behavioral design pattern lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

**A diagram of a strategy

Description automatically generated**

**RAII (Resource Acquisition Is Initialization)**

- To ensure that if a resource is used, it is released properly by attaching it into the life cycle of the object

- RAII helps to write exception-safe code easier

**Main applications**:

- Often used to manage thread lock of multi threading applications.

- Applications working with resources, such as dynamic memory allocating or file management to avoid leaking.