BASIC OOP IN C++

Overview

- Object Models, Classes, Inheritance, and Polymorphism
- Interface-based Design: Abstract Classes

References

- Gary J. Bronson: C++ for Engineers and Scientists. 3rd Edition. Thomson (2010)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

C++ OBJECT MODELS

- C++ supports:
  - A value-based object model (stack-based objects)
  - A reference-based object model (heap-based objects)

VALUE-BASED OBJECT MODEL

- In the value-based object model:
  - Objects are stored on the stack.
  - Object are accessed through object variables.
  - C++ scoping mechanism controls the lifetime of objects.
  - An object's memory is implicitly released when the object's lifetime reaches its end.

VALUE-BASED OBJECTS

- Value-based objects look and feel like records (or structs):

```
Card AceOfDiamond( Diamond, 14 );

Card TestCard( Diamond, 14 );

std::cout << “The test card is ” << TestCard.getName() << std::endl;
```

REFERENCE-BASED OBJECT MODEL

- In the reference-based object model:
  - Objects are stored on the heap.
  - Objects are accessed through pointer variables.
  - Heap objects persist even if the scope in which they have been created is closed.
  - An object's memory must be explicitly released.

REFERENCE-BASED OBJECTS

- Reference-based objects require pointer variables and an explicit ***new*** and ***delete***:

```
Card* AceOfDiamond = new Card( Diamond, 14 );

Card* TestCard = new Card( Diamond, 14 );

std::cout << "The test card is " << TestCard->getName() << std::endl;

delete AceOfDiamond;

delete TestCard;
```

member dereference

release memory

INHERITANCE

- Inheritance lets us define classes that model relationships among classes, sharing what is common, and specializing only that which is inherently different.

- Inheritance is

  - A mechanism for specialization;

  - A mechanism for reuse;

  - Fundamental to supporting polymorphism.

CLASS ACCOUNT

```
class Account
{
private:
    uint64_t fNumber;
    double fBalance;

public:
    Account( uint64_t aNumber, double aBalance = 0.0 ) noexcept;

    virtual ~Account() {}

    uint64_t number() const noexcept { return fNumber; }
    double balance() const noexcept { return fBalance; }

    bool deposit( double aAmount ) noexcept;

    virtual bool withdraw( double aAmount ) noexcept;
};
```

- An account has a number(2^{64}- 1 values) and a balance.
- To create an account we need a number. Funds can be credited to an account once it has been created.
- Class Account defines virtual members. Hence, it must define a virtual destructor and class Account can be overridden using inheritance.

VIRTUAL MEMBER FUNCTIONS

- To assign a member function inherited from a base class new behavior in a derived class, one overrides it.
- To allow a member function in a base class to be overridden, one must declare the member function virtual.
- Note, non-virtual member functions are resolved with respect to the declared type of the object.
- Explicitly defining a (virtual) destructor affects which special member functions the compiler synthesizes (i.e., no default constructor is automatically created).

METHOD OVERRIDING

- Method overriding is an object-oriented programming mechanism that allows a subclass to provide a more specific implementation of a method, which is also present in one of its superclasses.

- The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has the same name, same parameter signature, and same return type as the method in the parent class. We say these methods belong to the same method family.

- Which (overridden) method is selected at runtime is determined by the receiver object used to invoke the member. In general, the most recent definition is chosen, if possible.

METHOD FAMILY

- A member function of a class always belongs to a specific set, called method family.
- If the elements of this set are virtual (default), then their invocation is governed by dynamic binding, a technique that makes polymorphism real.

VIRTUAL WITHDRAW METHOD

```
class BankAccount : public Account
{
private:
    double fInterestRate;

public:
    BankAccount( uint64_t aNumber, double aBalance = 0.0, double aInterestRate = 0.0 ) noexcept;

    virtual ~BankAccount() override {}

    double& interest() noexcept { return fInterestRate; }

    bool withdraw( double aAmount ) noexcept override;

    bool creditInterest( double aInterval = 31.0 / 365.0 ) noexcept;
    bool applyServiceFee( double aFee = 5.0 ) noexcept;

    // inherited from Account
    // uint64_t number() const noexcept;
    // double balance() const noexcept;
    // bool deposit( double aAmount ) noexcept;
    // bool withdraw( double aAmount ) noexcept;
};
```

overridden method

ACCESS LEVELS FOR INHERITANCE

- **public**:
  - Public members in the base class remain public.
  - Protected members in the base class remain protected.
  - Yields a "is a" relationship, that is, a subtype.
- **protected**:
  - Public and protected members in the base class are protected in the derived class.
  - Yields a "implemented in terms of" relationship, that is, a new type.
- **private**:
  - Public and protected members in the base class become private in the derived class.
  - Yields a stricter "implemented in terms of" relationship, that is, a new type.

PUBLIC INHERITANCE

- Public inheritance enables inclusion polymorphism: A subclass is a subtype.
- The subtype relationship is a key ingredient in contemporary object-oriented software development.
- An object of a subclass can be used safely anywhere an object of a superclass is expected. Example, we can use a BankAccount object in lieu of an Account object.

INHERITANCE: OBJECT CONSTRUCTION

```
BankAccount::BankAccount( uint64_t aNumber, double aBalance, double aInterestRate ) noexcept :
    Account(aNumber, aBalance),
    fInterestRate(aInterestRate)
{}
```

- The subclass constructor must first initialize the base class object using a suitable base class constructor.
- We use a member initializer list to initialize both, the base class object and the subclass member variables.
- The base class constructor cannot be called from within the body of a subclass constructor.

OVERRIDING WITHDRAW

```
bool BankAccount::withdraw( double aAmount ) noexcept
{
    if ( (balance() - aAmount) > 0.0 )
    {
        return Account::withdraw( aAmount );
    }

    return false;
}
```

- BackAccount's withdraw() performs a balance check and allows withdraws only if sufficient funds are available.
- To call a base class method in C++, we must refer to the corresponding base class scope (e.g. Account::).

VIRTUAL METHOD CALLS

```
int main()
{
    BankAccount IAccount( 12345 );

    std::cout << "Balance: " << IAccount.balance() << std::endl;

    // test dynamic method invocation

    Account& IBankAccount = IAccount;

    if ( IBankAccount.withdraw( 50.0 ) )
    {
        std::cout << "Instant credit. Wow!" << std::endl;
    }
    else
    {
        std::cout << "Sorry, not enough funds available. :(" << std::endl;
    }

    return 0;
}
```

calls BankAccount::withdraw()

```
Balance: 0
Sorry, not enough funds available. :(
```

INTERFACE-BASED DESIGN

- Interface-based design is an architectural pattern for modular program development in object-oriented programming languages.
- From a conceptual standpoint, an interface is a contractual specification (i.e., protocol) between two parties. Ideally, if both parties adhere to the specification of the same interface, then correct interaction is guaranteed.
- In C++, we can use abstract classes as a means to create interfaces.
- Note, languages like C# or Java natively support interfaces as built-in abstraction. Interfaces and abstract classes coexist in these languages.
- An abstract class can define behavior, whereas an interface cannot (except for class-level features).

ABSTRACT CLASSES

- A class is abstract if it contains one or more pure virtual member functions.
- An abstract class cannot be instantiated.
- Derived classes must provide an implementation for the pure virtual member functions.
- An abstract class requires a virtual destructor. If one declares a pure virtual destructor, a definition for it must be given in a subclass eventually.

PURE VIRTUAL MEMBER FUNCTIONS

```
class AbstractClass
{
private:
    ...

public:

    AbstractClass();
    virtual ~AbstractClass();

    virtual type methodA() qualifier = 0;

    ...

    virtual type methodN() qualifier = 0;
};
```

- Pure virtual member functions declare what a class provides, but defer the implementation of that behavior to subclasses.
- In C++, we use = 0 to denote that a member function is abstract, that is, it does not have an implementation.

IMPLEMENTING PURE VIRTUAL MEMBER FUNCTIONS

```
class ConcreteClass : public AbstractClass
{
private:
    ...

public:

    ConcreteClass
    virtual ~ConcreteClass();

    virtual type methodA() qualifier override;

    ...

    virtual type methodN() qualifier override;
};
```

- In a subclass we override the pure virtual members.
- If all pure virtual members have been overridden, then we can create objects of this class.