Inheritance

- Property of object-oriented programming that allows one class, known as a derived class, to share the structure and behavior of another class, known as the base class

- Allows newly created members to inherit members (attributes and methods) from existing classes

- Derived (or child) classes include their own members and members inherited from the base (or parent) class

- A collection of classes with common inherited members is viewed as a family of classes

Why use inheritance?

- Allows you to reuse the code from a previous programming project
  - You can enhance an existing class by adding new attributes and methods
  - Extremely practical approach in an environment with multiple programmers, working on the code written by others long ago
    * You do not have to get inside each method of an existing class to modify the data and code

- Allows you to build a hierarchy among classes
  - A general bank account is used to define attributes such as an account number and account balance, and member functions such as deposit that are common to all bank accounts
  - Classes that define checking account and savings accounts can be derived from the base class bank account
  - We could even derive another class called super-now account from the class checking account, leading to a grandchild class of bank account
  - Such a family of classes is referred to as a class hierarchy

- IS-A relationship
  - Important link between a derived class and its base class
  - Must exist for proper use of inheritance
  - A checking account IS-A bank account

Derived classes

- A derived class is declared using the following format:

```cpp
class <derived class> : public <base class>
{
    <derived class member data>
    <derived class member functions>
};
```

- The class for bank accounts
class bank_account
{
  protected:
    int    account_no;
    float  balance;

  public:
    void deposit ( float ); // Add deposit
    int    account_num ( void ); // Return account number
    float  curr_bal ( void ); // Return current balance
};

- There is no constructor function to initialize the attributes of the class because we are going to leave that for the derived classes
- The way things are structured, we will not be creating any objects for the base class
- When no objects are to be created for a class, the class is known as an abstract class
- The attributes of the class have been declared to be protected

• protected member
  - A member that is accessible to both the base class and any derived classes of the base class in which it is declared
  - A protected member in a base class is accessible to any class within the class family but not accessible from outside the class family
  - If a member is declared to be private to a base class, it is not accessible to a derived class

• Implementation of methods in the base class

  void bank_account::deposit ( float amount )
  {
    balance += amount;
  }

  int bank_account::account_num ( void )
  {
    return ( account_no );
  }

  float bank_account::curr_bal ( void )
  {
    return ( balance );
  }

• Derived class for checking account

class checking : public bank_account
{
  protected:
    float minimum;       // Minimum balance to avoid check charge
    float charge;        // Per check-charge

  public:
    // Constructor functions
    checking ( void );
The keyword `public` prior to the base class name makes the class `bank_account` a public base class to the derived class `checking`.

A public base class allows all public members of the base class to be public in the derived class.

The inherited members of the public base class (both attributes and methods) maintain their access level in the derived class.

Without the use of the keyword `public`, the public functions of the base class would not be accessible to any code using an object of the class `checking`, and the following will lead to a compile-time error:

```cpp
checking ca;
float bal = ca.curr_bal();
```

- `minimum` and `charge` are defined to be protected members because they will be inherited by the class `super_now`.
- Because of inheritance, `checking` has four attributes – two of its own and two inherited from `bank_account`.

**Implementation of the methods in the derived class `checking`**

```cpp
class checking {  
public:  
    checking ( int, float, float, float );  
    void cash_check ( float );  
private:  
    checking ( void ) // Default constructor  
    {  
        account_no = 0;  
        balance = 0.0;  
        minimum = 0.0;  
        charge = 0.50;  
    }  
    checking ( int acct_no = 0, float bal = 0.0, float min = 1000, float chg = 0.50 )  
    {  
        account_no = acct_no;  
        balance = bal;  
        minimum = min;  
        charge = chg;  
    }  
    void cash_check ( float amt ) // Cash a check  
    {  
        if ( amt > balance ) // Test for overdraft  
            cerr << "Cannot cash check, account will be overdrawn." << endl;  
        else  
            balance -= ( balance < minimum ) ? ( amt + charge ) : amt;  
    }  

class super_now : public checking {  
public:  
    float int_rate;  
    // Annual rate of interest  
    super_now ( void );  
    // Constructor functions
```
The `super_now` class has only one attribute, and inherits four attributes from its parent and grandparent:

- **Implementation of the methods in the derived class `super_now`**

```cpp
super_now::super_now ( void ) // Default constructor
{
    account_no = 0;
    balance = 0.0;
    minimum = 0.0;
    charge = 0.50;
    int_rate = 2.0;
}

super_now::super_now ( int acct_no = 0, float bal = 0.0, float min = 0.0, \n    float chg = 0.5, float rate = 2.0 ) : checking ( acct_no, bal, min, chg )
{
    int_rate = rate;
}

void super_now::add_interest ( void ) // Add interest to balance
{
    float interest;

    if ( balance >= minimum )
    {
        interest = balance * ( int_rate * 0.01 / 12 );
        balance += interest;
    }
}
```

- **The derived class `savings`**

```cpp
class savings : public bank_account
{
    float int_rate; // Annual rate of interest

public:
    savings ( void ); // Default constructor
    savings ( int, float, float );
    void add_interest ( void ); // Add interest to balance
    void withdraw ( float ); // Make a withdrawal
};
```

- Two attributes are inherited from the base class `bank_account`

- **Implementation of the methods in the derived class `savings`**

```cpp
savings::savings ( void ) // Default constructor
{
    account_no = 0;
```
balance = 0.0;
int_rate = 4.0;
}
savings::savings ( int acct_no = 0, float bal = 0.0, float rate = 4.0 )
{
    account_no = acct_no;
    balance = bal;
    int_rate = rate;
}
void savings::withdraw ( float amt ) // Make a withdrawal
{
    balance -= amt;
}
void savings::add_interest ( void ) // Add interest
{
    float interest;
    interest = balance * ( int_rate * 0.01 / 12 );
    balance += interest;
}

• Structure of the base class bank_account

| account_no | balance |

• Structure of the derived class checking

| account_no | balance | minimum | charge |

• Structure of the derived class super_now

| account_no | balance | minimum | charge | int_rate |

• Structure of the derived class savings

| account_no | balance | int_rate |

In each of the above classes, a pointer to the derived classes is also a pointer to the base class

Consider the following case:

    bank_account * b = new checking;
Inheritance

This will allocate space for an object of the class checking but ignore the attributes that are not visible in bank_account

- Difference between inheritance and using a base class as a subclass in the derived class
  - Both the cases have the same memory layout with the following differences:
    - A pointer to the derived class is not automatically a pointer to the base class
    - A public member function of the base class cannot be applied to the derived class (you will have to specify derived.base.foo())

Virtual functions

- Dynamic v. Static binding
  - Static binding
    - Static binding occurs when a polymorphic function is defined for several classes in a family and the code for the function is attached, or bound, at compile time
    - Overloaded functions are statically bound
  - Dynamic binding
    - Dynamic (or late) binding occurs when a polymorphic function is defined for several classes in a family but the actual code for the function is not attached, or bound, until execution time
    - A polymorphic function that is dynamically bound is called a virtual function.

- A pointer to a derived class automatically points to a base class
  - We can put pointers to a derived class objects and to base class objects in an array
  - We cannot put actual objects of the base class and derived classes in an array because of different memory size requirements
  - Imagine printing the bank statement for all checking accounts:

```cpp
for ( accts.reset(); !accts.at_end(); accts.advance() )
    accts.print_statement();
```
  - Because of differing memory requirements, there is no way of telling whether the account is a plain checking account or a super-now account and certainly, the print function for both the cases will have to be handled differently
  - We can add a type field to specify the type of account and use it to pick up the correct print function:

```cpp
switch ( acct->type )
{
    case 0: acct->print_statement(); break;
    case 1: ( ( super_now * ) acct ) -> print_statement(); break;
}
```

- A virtual function allows us to avoid writing such code and solves the problem by making the print_statement function virtual in base class checking

```cpp
class checking : public bank_account
{
    // ...
    virtual void print_statement();
};
```

- A virtual function alerts the compiler automatically to make a type field that distinguishes a plain checking account and all its derived classes, and to translate each call
Inheritance

 acct->print_statement();

 into run-time selection of the correct method

 • This is also known as run-time overloading of the function
 • Only member functions can be virtual

Multiple inheritance

 • Look at the following inheritance hierarchy:

```
Box

Text

Shaded box

Text box
```

– The inheritance is no longer a tree-shaped hierarchy
– Text box inherits from two classes: Box and Text
– It was possible to define the class Text_Box as
  ```
  class Text_Box : public Box
  {
    Text t;
    public:
      // ...
  }
  ```
  but then, it will not inherit from the class Text

• Solution: Multiple inheritance

• The declaration looks like:
  ```
  class Text_Box : public Box, public Text
  {
    // ...
  }
  ```

• Notice that there are two separate keywords public in front of each class from where the derived class inherits

• The constructor function for such a class looks like
  ```
  Text_Box :: Text_Box ( int x1, int y1, int x2, int y2, const char t[] )
  : Box ( x1, y1, x2, y2 ), Text ( t )
  {}
  ```

• With multiple inheritance, a pointer to a derived class no longer automatically points to the base class because the order of inclusion of the two classes is not guaranteed
• However, whenever a pointer to a derived class is converted to a pointer to any of the base classes, an appropriate offset is added by the compiler

• What if we now want to have a class called Shaded_Text_Box
  – It is natural to combine the two classes: Shaded_Box and Text_Box as
    ```cpp
class Shaded_Text_Box : public Shaded_Box, public Text_Box
{
    // ...
};
```
  – The inheritance looks like

  ![Inheritance Diagram]

  – There is a serious problem: The class Box is inherited via both Shaded_Box and Text_Box
  – If any of the attributes in the class Box is referred to, is the reference to be resolved through Shaded_Box or Text_Box
  – Wasting the space for both Boxes is certainly undesirable

• Solution: virtual base classes
  – The classes Shaded_Box and Text_Box can both declare the base class Box to be virtual
    ```cpp
class Text_Box : virtual public Box { ... };  
class Shaded_Box : public virtual Box { ... }; 
```
  – The order of public and virtual does not matter
  – A class inheriting from both these classes will get only one instance of Box
  – If there are multiple virtual classes, the common ones are coalesced to a single copy

• Restriction on base class construction
  – A virtual base class may be initialized only from the most derived class or with a constructor that requires no arguments
    * In the above example, the most derived class is Shaded_Text_Box