Topics

1. OOP Introduction
2. Type & Subtype
3. Inheritance
4. Overloading and Overriding

Section 1

OOP Introduction
So far the languages that we encountered treat data and computation separately.

In OOP, the data and computation are combined into an “object”.

Benefits of OOP

- more convenient: collects related information together, rather than distributing it.
  - Example: C++ iostream class collects all I/O related operations together into one central place.
  - Contrast with C I/O library, which consists of many distinct functions such as getchar, printf, scanf, sscanf, etc.

- centralizes and regulates access to data.
  - If there is an error that corrupts object data, we need to look for the error only within its class
  - Contrast with C programs, where access/modification code is distributed throughout the program

Benefits of OOP (Continued)

- Promotes reuse.
  - by separating interface from implementation.
    - We can replace the implementation of an object without changing client code.
    - Contrast with C, where the implementation of a data structure such as a linked list is integrated into the client code
  - by permitting extension of new objects via inheritance.
    - Inheritance allows a new class to reuse the features of an existing class.
    - Example: define doubly linked list class by inheriting/reusing functions provided by a singly linked list.
Encapsulation & Information hiding

- Encapsulation
  - centralizing/regulating access to data
- Information hiding
  - separating implementation of an object from its interface
- These two terms overlap to some extent.

Classes and Objects

- Class is an (abstract) type
  - includes data
    - class variables (aka static variables)
      - shared (global) across all objects of this class
    - instance variables (aka member variables)
      - independent copy in each object
      - similar to fields of a struct
  - and operations
    - member functions
      - always take object as implicit (first) argument
    - class functions (aka static functions)
      - don't take an implicit object argument
- Object is an instance of a class
  - variable of class type

Access to Members

- Access to members of an object is regulated in C++ using three keywords
  - Private:
    - Accessibly only to member functions of the class
    - Can't be directly accessed by outside functions
  - Protected:
    - allows access from member functions of any subclass
  - Public:
    - can be called directly by any piece of code.
Member Function

- Member functions are of two types
  - statically dispatched
  - dynamically dispatched.
- The dynamically dispatched functions are declared using the keyword “virtual” in C++
  - all member function functions are virtual in Java

C++

- Developed as an extension to C by adding object oriented constructs originally found in Smalltalk (and Simula67).
- Most legal C programs are also legal C++ programs
  - “Backwards compatibility” made it easier for C++ to be accepted by the programming community
  - ... but made certain features problematic (leading to “dirty” programs)
- Many of C++ features have been used in Java
  - Some have been “cleaned up”
  - Some useful features have been left out

Example of C++ Class

- A typical convention is C++ is to make all data members private. Most member functions are public.
- Consider a list that consists of integers. The declaration for this could be:

```cpp
class IntList {
    private:
        int elem; // element of the list
        IntList *next; // pointer to next element
    public:
        IntList (int first); // "constructor"
        ~IntList (); // "destructor".
        void insert (int i); // insert element i
        int getval (); // return the value of elem
        IntList *getNext (); // return the value of next

    }
```
Example of C++ Class (Continued)

- We may define a subclass of IntList that uses doubly linked lists as follows:

```cpp
class IntDList: IntList {
  private:
    IntList *prev;
  public:
    IntDList(int first);
    // Constructors need to be redefined
    ~IntDList();
    // Destructors need not be redefined, but
    // typically this is needed in practice.
    // Most operations are inherited from IntList.
    // But some operations may have to be redefined.
    insert (int);
    IntDList *prev();
}
```

C++ and Java: The Commonalities

- Classes, instances (objects), data members (fields) and member functions (methods).
- Overloading and inheritance.
  - base class (C++) → superclass (Java)
  - derived class (C++) → subclass (Java)
- Constructors
- Protection (visibility): private, protected and public
- Static binding for data members (fields)

A C++ Primer for Java Programmers

Classes, fields and methods:

**Java:**

```java
class A extends B {
  private int x;
  protected int y;
  public int f() {
    return x;
  }
  public void print() {
    System.out.println(x);
  }
}
```

**C++:**

```cpp
class A : public B {
  private: int x;
  protected: int y;
  public: int f() {
    return x;
  }
  void print() {
    std::cout << x << std::endl;
  }
}
```
Declaring objects:
- In Java, the declaration `A va` declares `va` to be a *reference* to object of class `A`.
  - Object creation is always via the `new` operator
- In C++, the declaration `A va` declares `va` to be an object of class `A`.
  - Object creation may be automatic (using declarations) or via new operator:
    ```cpp
    A *va = new A;
    ```

**Objects and References**

- In Java, all objects are allocated on the heap; references to objects may be stored in local variables.
- In C++, objects are treated analogous to `C structs`: they may be allocated and stored in local variables, or may be dynamically allocated.

**Parameters to methods:**
- Java distinguishes between two sets of values: primitives (e.g. `int`, `float`, etc.) and objects (e.g `String`, `Vector`, etc.)
  - Primitive parameters are passed to methods *by value* (copying the value of the argument to the formal parameter)
  - Objects are passed *by reference* (copying only the reference, not the object itself).
- C++ passes all parameters *by value* unless specially noted.
Type

- **Apparent Type**: Type of an object as per the declaration in the program.
- **Actual Type**: Type of the object at run time.

Let **Test** be a subclass of **Base**. Consider the following Java program:

```java
Base b = new Base();
Test t = new Test();
...

b = t;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Apparent type of object referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Base</td>
</tr>
<tr>
<td>t</td>
<td>Test</td>
</tr>
</tbody>
</table>

... throughout the scope of b and t’s declarations

Type (Continued)

Let **Test** be a subclass of **Base**. Consider the following Java program fragment:

```java
Base b = new Base();
Test t = new Test();
...

b = t;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Program point</th>
<th>Actual type of object referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>before b=t</td>
<td>Base</td>
</tr>
<tr>
<td>t</td>
<td>before b=t</td>
<td>Test</td>
</tr>
<tr>
<td>b</td>
<td>after b=t</td>
<td>Test</td>
</tr>
<tr>
<td>t</td>
<td>after b=t</td>
<td>Test</td>
</tr>
</tbody>
</table>

Type (Continued)

Things are a bit different in C++, because you can have both objects and object references. Consider the case where variables are objects in C++:

```cpp
Base b();
Test t();
...

b = t;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Program point</th>
<th>Actual type of object referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>before b=t</td>
<td>Base</td>
</tr>
<tr>
<td>t</td>
<td>before b=t</td>
<td>Test</td>
</tr>
<tr>
<td>b</td>
<td>after b=t</td>
<td>Base</td>
</tr>
<tr>
<td>t</td>
<td>after b=t</td>
<td>Test</td>
</tr>
</tbody>
</table>
Things are a bit different in C++, because you can have both objects and object references. Consider the case where variables are pointers in C++:

```cpp
Base *b = new Base();
Test *t = new Test();
...
b = t;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Program point</th>
<th>Actual type of object referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>before b=t</td>
<td>Base*</td>
</tr>
<tr>
<td>t</td>
<td>before b=t</td>
<td>Test*</td>
</tr>
<tr>
<td>b</td>
<td>after b=t</td>
<td>Test*</td>
</tr>
<tr>
<td>t</td>
<td>after b=t</td>
<td>Test*</td>
</tr>
</tbody>
</table>

### Subtype Principle

A key principle:

- "For any operation that expects an object of type \( T \), it is acceptable to supply object of type \( T' \), where \( T' \) is subtype of \( T \)."

The subtype principle enables OOL to support subtype polymorphism:

- client code that accesses an object of class \( C \) can be reused with objects that belong to subclasses of \( C \).
Subtype Principle (Continued)

- The following function will work with any object whose type is a subtype of IntList.
  
  ```cpp
  void q(IntList &i, int j) {
    ... 
    i.insert(j) ;
  }
  ```

  Subtype principle dictates that this work for IntList and IntDList.
  - This must be true even if the insert operation works differently on these two types.
  - Note that use of IntList::insert on IntDList object will likely corrupt it, since the prev pointer would not be set.

Subtype Principle (Continued)

- Hence, `i.insert` must refer to
  - IntList::insert when `i` is an IntList object, and
  - IntDList::insert function when `i` is an IntDList.

  Requires dynamic association between the name “insert” and the its implementation.

  - achieved in C++ by declaring a function be virtual.
  - definition of insert in IntList should be modified as follows: virtual void insert(int i);
  - all member functions are by default virtual in Java, while they are nonvirtual in C++
  - equivalent of “virtual” keyword is unavailable in Java.

Reuse of Code

- Reuse achieved through subtype polymorphism
  - the same piece of code can operate on objects of different type, as long as:
    - Their types are derived from a common base class
    - Code assumes only the interface provided by base class.

  Polymorphism arises due to the fact that the implementation of operations may differ across subtypes.
Reuse of Code (Continued)

- Example:
  - Define a base class called DrawableObject
    - supports draw() and erase().
  - DrawableObject just defines an interface
    - no implementations for the methods are provided.
    - this is an abstract class — a class with one or more abstract methods (declared but not implemented).
    - also an interface class — contains only abstract methods subtypes.

Reuse of Code: example (Continued)

- The hierarchy of DrawableObject may look as follows:

```
DrawableObject
    |__________________________|
    |                          |
    | BitMaps                  |
    |                           |
    | GeometricShapes          |
    |                           |
    | ClosedFigures            |
    |                           |
    | Polygon                  |
    |                           |
    | Rectangle                |
    |                           |
    | Square                   |
    |                           |
    | OpenFigures              |
    |                           |
    | Ellipse                  |
    |                           |
    | Triangle                 |
    |                           |
    | Circle                   |
    |                           |
```

- The subclasses support the draw() and erase() operation supported by the base class.

- Given this setting, we can implement the redraw routine using the following code fragment:

```c
void redraw(DrawableObject* objList[], int size){
    for (int i = 0; i < size; i++)
        objList[i]->draw();
}
```
Reusing Code: example (Continued)

- objList[i].draw will call the appropriate method:
  - for a square object, Square::draw
  - for a circle object, Circle::draw

- The code need not be changed even if we modify the inheritance hierarchy by adding new subtypes.

Compare with implementation in C:
```c
void redraw(DrawableObject *objList[], int size) {
    for (int i = 0; i < size; i++){
        switch (objList[i]->type){
            case SQUARE: square_draw((struct Square *)objList[i]);
            break;
            case CIRCLE: circle_draw((struct Circle *)objList[i]);
            break;
            ........
            default: ....
        }
    }
}
```

Differences:
- no reuse across types (e.g., Circle and Square)
- need to explicitly check type, and perform casts
- will break when new type (e.g., Hexagon) added

Reusing Code (Continued)

- Reuse achieved through subtype polymorphism
  - the same piece of code can operate on objects of different type, as long as:
    - Their types are derived from a common base class
    - Code assumes only the interface provided by base class.

- Polymorphism arises due to the fact that the implementation of operations may differ across subtypes.
Dynamic Binding

- Dynamic binding provides overloading rather than parametric polymorphism.
  - the draw function implementation is not being shared across subtypes of DrawableObject, but its name is shared.
- Enables client code to be reused
- To see dynamic binding more clearly as overloading:
  - Instead of a.draw(),
  - view as draw(a)

Reuse of Code (Continued)

- Subtype polymorphism = function overloading
- Implemented using dynamic binding
  - i.e., function name is resolved at runtime, rather than at compile time.
- Conclusion: just as overloading enables reuse of client code, subtype polymorphism enables reuse of client code.

Section 3

Inheritance
Inheritance

- language mechanism in OO languages that can be used to implement subtypes.
- The notion of interface inheritance corresponds conditions (1), (2) and (3) in the definition of Subtype
- but provision (3) is not checked or enforced by a compiler.

Subtyping & interface inheritance

- The notion of subtyping and interface inheritance coincide in OO languages.
  OR
- Another way to phrase this is to say that “interface inheritance captures an 'is-a' relationship”
  OR
- If A inherits B's interface, then it must be the case that every A is a B.

Implementation Inheritance

- If A is implemented using B, then there is an implementation inheritance relationship between A and B.
  - However A need not support any of the operations supported by B
    OR
  - There is no 'is-a' relationship between the two classes.
- Implementation inheritance is thus “irrelevant” from the point of view of client code.
- Private inheritance in C++ corresponds to implementation-only inheritance, while public inheritance provides both implementation and interface inheritance.
Implementation Inheritance (Continued)

- Implementation-only inheritance is invisible outside a class
  - not as useful as interface inheritance.
  - can be simulated using composition.

```java
class B{
    op1(...)  // Some operations supported by B may also be supported in A (e.g., op1), while others (e.g., op2) may not be
    op2(...)  // New operations supported by A
}
class A: private class B {
    op1(...)  // Some operations supported by B may also be supported in A (e.g., op1), while others (e.g., op2) may not be
        // New operations supported by A
}
```

- The implementation of op1 in A has to explicitly invoke the implementation of op1 in B:

```java
A::op1(...){
    B::op1(...)
}
```

- So, we might as well use composition:

```java
class A{
    B b;
    op1(...) {  b.op1(...)  }
    op3(...)
}
```

Polymorphism

"The ability to assume different forms"

- A function/method is polymorphic if it can be applied to values of many types.
- Class hierarchy and inheritance provide a form of polymorphism called subtype polymorphism.
- As discussed earlier, it is a form of overloading.
  - Overloading based on the first argument alone.
  - Overloading resolved dynamically rather than statically.
- Polymorphic functions increase code reuse.
Polymorphism (Continued)

- Consider the following code fragment: 
  \[(x < y)? x : y\]
- “Finds the minimum of two values”.
- The same code fragment can be used regardless of whether \(x\) and \(y\) are:
  - integers
  - floating point numbers
  - objects whose class implements operator “\(<\)”.
- **Templates** lift the above form of polymorphism (called parametric polymorphism) to functions and classes.

Parametric polymorphism Vs Interface Inheritance

- In C++,
  - template classes support parametric polymorphism
  - public inheritance support interface + implementation inheritance.
  - Parametric polymorphism is more flexible in many cases.

```cpp
template class List<class ElemType>{
  private:
    ElemType *first; List<ElemType> *next;
  public:
    ElemType *get(); void insert(ElemType *e);
};
```

Now, one can use the List class with any element type:

```cpp
void f(List<A> alist, List<B> blist){
  A a = alist.get();
  B b = blist.get();
}
```

Parametric polymorphism Vs Inheritance (Continued)

- If we wanted to write a List class using only subtype polymorphism:
  - We need to have a common base class for \(A\) and \(B\)
  - e.g., in Java, all objects derived from base class “Object”

```cpp
class AltList{
  private:
    Object first; AltList next;
  public:
    Object get(); void insert(Object o);
};
```

```cpp
void f(AltList alist, AltList blist) {
  A a = (A)alist.get();
  B b = (B)blist.get();
}
```
Parametric polymorphism Vs Interface Inheritance (Continued)

- Note: get() returns an object of type Object, not A.
- Need to explicitly perform runtime casts.
  - type-checking needs to be done at runtime, and type info maintained at runtime
  - potential errors, as in the following code, cannot be caught at compile time

```c
List alist, blist;
A a; A b; //Note b is of type A, not B
alist.insert(a);
blist.insert(b);
f(alist, blist); //f expects second arg to be list of B's, but we are giving a list of A's.
```

Section 4

Overloading and Overriding

Overloading, Overriding, and Virtual Functions

- Overloading is the ability to use the same function NAME with different arguments to denote DIFFERENT functions.
  - In C++
    - void add(int a, int b, int& c);
    - void add(float a, float b, float& c);
  - Overriding refers to the fact that an implementation of a method in a subclass supersedes the implementation of the same method in the base class.
Overloading, Overriding, and Virtual Functions (Continued)

- Overriding of non-virtual functions in C++:

  ```cpp
class B {
  public:
    void op1(int i) { /* B's implementation of op1 */ }
  }

class A: public class B {
  public:
    void op1(int i) { /* A's implementation of op1 */ }
  }

main() {
  B b; A a;
  int i = 5; b.op1(i); // B's implementation of op1 is used
  a.op1(i); // Although every A is a B, and hence B's implementation of
  // op1 is available to A, A's definition supercedes B's defn,
  // so we are using A's implementation of op1.
  ((B)a).op1(); // Now that a has been cast into a B, B's op1 applies.
  a.B::op1(); // Explicitly calling B's implementation of op1
}
```

In the above example the choice of B's or A's version of op1 to use is based on
compile-time type of a variable or expression. The runtime type is not used.

- Overloaded (non-member) functions are also resolved using compile-time type
  information.

Overriding In The Presence Of Virtual Function

```cpp
class B {
  public:
    virtual void op1(int i) { /* B's implementation of op1 */ }
  }

class A: public class B {
  public:
    void op1(int i) { /* A's implementation of op1 */ }
  }

main() {
  B b; A a;
  int i = 5;
  b.op1(i); // B's implementation of op1 is used
  a.op1(i); // A's implementation of op1 is used.
  ((B)a).op1(); // Still A's implementation is used
  a.B::op1(); // Explicitly requesting B's definition of op1
}
```
Overriding In The Presence Of Virtual Function (Continued)

```cpp
void f(B x, int i) {
    x.op1(i);
}

which may be invoked as follows:

B b;
A a;
f(b, 1); // f uses B's op1
f(a, 1); // f still uses B's op1, not A's
```

```cpp
void f(B& x, int i) {
    x.op1(i);
}

which may be invoked as follows:

B b;
A a;
f(b, 1); // f uses B's op1
f(a, 1); // f uses A's op1
```