Object-Orientation

CSCI 3136
Principles of Programming Languages

Faculty of Computer Science
Dalhousie University

Winter 2012

Reading: Chapter 9
What is a Object-Oriented Programming?

Elements of object-oriented programming:

- Data items to be manipulated are *objects*.
- Objects are members of *classes*, that is, classes are types.
- Objects store data in *fields* and behaviour in *methods* specified by their classes.

Main characteristics of most object-oriented programming systems:

- *Encapsulation* by hiding internals of an object from the user of the object.
- Customization of behaviour through *inheritance*.
- Polymorphism through *dynamic method binding*.
Advantages of Object-Oriented Programming

It *reduces conceptual load*:

- It reduces the amount of detail the programmer must think about at the same time.

It provides *fault and change containment*:

- It limits the portion of a program that needs to be looked at when debugging.
- It limits the portion of a program that needs to be changed when changing the behaviour of an object without changing its interface.

It provides *independence of program components* and thus *facilitates code reuse*.
Advantages of Object-Oriented Programming

It reduces conceptual load:

- It reduces the amount of detail the programmer must think about at the same time.

It provides fault and change containment:

- It limits the portion of a program that needs to be looked at when debugging.
- It limits the portion of a program that needs to be changed when changing the behaviour of an object without changing its interface.

It provides independence of program components and thus facilitates code reuse.

Note: Most of these are consequences of encapsulation and thus apply also to programming using modules.
Some Object-Oriented Languages

- SIMULA 67
- Smalltalk 72
- C++, 80s
- Modula-3, late 80s
- CLOS, 88
- Eiffel, 92
- Oberon, 90s (last version 95)
- Java, 95
- Ada 95
class list_node {
    list_node *prev, *next, *head;

public:

    int val;
    list_node();
    ~list_node();
    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();
};

void list_node::insert_before(list_node *new_node) {
    if (new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
Class Example in C++

class list_node {
    list_node *prev, *next, *head;

public:

    int val;

    list_node();
    ~list_node();

    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();
};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
class list_node {

    list_node *prev, *next, *head;

public:

    int val;

    list_node();
    ~list_node();

    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();
};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
Class Example in C++

class list_node {
    list_node *prev, *next, *head;

public:

    int val;

    list_node();
    ~list_node();

    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();
};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
class list_node {
    list_node *prev, *next, *head;

public:
    int val;

    list_node();
    ~list_node();

    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();
};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
class list_node {

// Private fields
    list_node *prev, *next, *head;

public:

// Public field
    int val;

// Constructor
    list_node();
    ~list_node();

// Public methods
    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();

};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
class list_node {

    list_node *prev, *next, *head;

public:

    int val;

    list_node();
    ~list_node();

    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();

};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
```cpp
class list_node {

    list_node *prev, *next, *head;

public:

    int val;

    list_node();
    ~list_node();

    list_node *predecessor();
    list_node *successor();
    bool singleton();
    void insert_before(list_node *new_node);
    void remove();

};

void list_node::insert_before(list_node *new_node) {
    if (!new_node->singleton())
        throw new list_err("inserting more than a single node");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
```
Inheritance

Using inheritance we can define a new *derived* or *child class* based on an existing *parent class* or *superclass*.

The derived class

- Inherits all fields and methods of the superclass,
- Can define additional fields and methods, and
- Can override existing fields and methods.

**Purpose:** Extend or specialize the behaviour of the superclass.
Inheritance

Using inheritance we can define a new *derived* or *child class* based on an existing *parent class* or *superclass*.

The derived class

- Inherits all fields and methods of the superclass,
- Can define additional fields and methods, and
- Can override existing fields and methods.

**Purpose:** Extend or specialize the behaviour of the superclass.

This allows us to define a *class hierarchy*.

- If only single inheritance is allowed, the hierarchy is a tree.
- If multiple inheritance is allowed, the hierarchy is a lattice.
Syntax of Inheritance

C++

    class push_button : public widget { ... }

Java

    public class push_button extends widget { ... }

Ada

    type push_button is new widget with ...
Syntax of Inheritance

C++

    class push_button : public widget { ... }

Java

    public class push_button extends widget { ... }

Ada

    type push_button is new widget with ...

Bad example in the textbook (C++)

    class queue : public list { ... }

Why is this a bad example?
Overriding Methods of a Base Class

To replace a method of a base class, redefine it in the derived class:

class widget {
    ...
    void paint();
    ...
};

class push_button : public widget {
    ...
    void paint();
    ...
};
To replace a method of a base class, redefine it in the derived class:

class widget {
    ...
    void paint();
    ...
};

class push_button : public widget {
    ...
    void paint();
    ...
};

Methods of the base class are still accessible in the derived class:

- Using scope resolution (::) in C++
- Using the super keyword in Java or Smalltalk
- Using explicit renaming in Eiffel
Syntax of Accessing Members of the Base Class

C++:     widget::paint()
Java:     super.paint()
C#:      base.paint()
Smalltalk:  super paint.
Objective C:  [super paint]
Eiffel:   class queue inherit list
          rename remove as old_remove
Encapsulation

Using modules:

• Define an *opaque* module type, a type whose definition is not exported by the module.

• Export subroutines to manipulate objects of the type. The implementation of these subroutines is not visible to the module’s user.

Using classes:

• *Public* methods are accessible to the class’s user, *private* methods are not.

• Private methods are accessible to other objects of the same class.

• Effective use of inheritance requires more fine-grained control over visibility of methods than sufficient when using modules.
Visibility in C++

Three visibility levels:

- **Private** methods/fields are visible to members of objects of the same class and to friends.
- **Protected** methods/fields are visible to members of objects of the same class or derived classes and to friends.
- **Public** methods/fields are visible to the whole world.
Visibility in C++

Three visibility levels:

- **Private** methods/fields are visible to members of objects of the same class and to friends.
- **Protected** methods/fields are visible to members of objects of the same class or derived classes and to friends.
- **Public** methods/fields are visible to the whole world.

Friends:

- A class can declare other classes and functions to be its friends, thereby providing them with access to its private and protected members.

```cpp
class X {
    int a;
    friend void f(int);
    friend class Y;
};
```
Derived classes can restrict (but not increase) the visibility of its base class’s members in objects of the derived class.

```c++
class A : public B { ... }
```
- All methods have the same visibility in the derived class as in the base class.

```c++
class A : protected B { ... }
```
- Public and protected members of the base class become protected in the derived class. Private members remain private.

```c++
class A : private B { ... }
```
- All members of the base class become private in the derived class.
Altering Visibility of Individual Members

class A {
    public:
        void a();
        void b();
    private:
        void c();
};

class B : private A {
    public:
        using A::a();
        using A::c();
};
Altering Visibility of Individual Members

```cpp
class A {
    public:
        void a();
        void b();
    private:
        void c();
};

class B : private A {
    public:
        using A::a();
        using A::c();
};

• a() is public in B.
• b() is private in B.
• The second `using` statement is illegal because it would increase the visibility of a private member of A.
```
Visibility Rules in Other Languages

Eiffel

- Derived classes can both restrict and increase the visibility of members of base classes.

Java

- Similar to C++, with the following exceptions.
- Base classes are always public.
- Protected members are visible in derived classes and in the same package.
- No notion of friends.

Python

- All class members are public.

Smalltalk, Objective C

- All methods are public.
- All fields are private.
Constructors

A constructor does not allocate the space for an object; it initializes ("constructs") the object in the allocated space.

Execution order of constructors:

• Constructor(s) of base class(es).
• Constructors of class members.
• Constructor of the class itself.

```cpp
class A {
public:
    A() { cout << "A"; }
};

class B {
public:
    B() { cout << "B"; }
}

class C : A {
    B b;
public:
    C() { cout << "C"; }
};

int main() {
    C c;
}
```

This prints "ABC".
Constructor and Method Overloading (1)

class A {
    ...

public:
    A() { ... } // Constructor 1
    A(int x) { ... } // Constructor 2

    void f(float x) { ... } // Method 1
    void f(int x) { ... } // Method 2
    void f(int x) const { ... } // Method 3

};

int main() {
    A x; // Calls constructor 1
    const A y(5); // Calls constructor 2

    x.f(3.4); // Calls method 1
    x.f(3); // Calls method 2
    y.f(3); // Calls method 3
    y.f(4.5); // Error: non-const method applied to const object
}

class A {
    ...
    public:
        A() { ... }

        void f(int x) { ... } // Method 1
        void f(int &x) { ... } // Method 2
    
};

int main() {
    A x;
    int y = 3;

    x.f(y); // Error: cannot decide which method to call
class A {
    int x;

public:
    A() : x(0) { cout << "C1"; }
    A(const A& a) : x(a.x) { cout << "C2"; }
    const A& operator =(const A& a) { x = a.x; cout << "A"; }
};

int main() {
    A u;  // Prints "C1"
    A v(u); // Prints "C2"
    A w = u; // Prints "C2"
    A x; // Prints "C1"
    x = u; // Prints "A"
}
Copy Constructors and Assignment

class A {
  int x;
public:
  A() : x(0) { cout << "C1"; }  
  A(const A& a) : x(a.x) { cout << "C2"; }
  const A& operator =(const A& a) { x = a.x; cout << "A"; }
};

int main() {
  A u;   // Prints "C1"
  A v(u); // Prints "C2"
  A w = u; // Prints "C2"
  A x;   // Prints "C1"
  x = u;  // Prints "A"
}

A similar analysis applies to

class A {
  int x;
public:
  A() : x(1) {} 
};

class A {
  int x;
public:
  A() { x = 1; }
};

Object-Orientation
CSCI 3136: Principles of Programming Languages
In languages with a reference model of variables or when using pointers in C++, we can use an object of a derived class where an object of the base class is expected.

Assume the derived class overrides a method of the base class.

When accessing an object of the derived class through a variable whose type is the base class, which method should we call?
Static vs Dynamic Method Binding (2)

Static method binding:

• The method invoked is determined by the type of the variable through which the object is accessed.
• Languages with static method binding: Simula, C++, Ada 95

Dynamic method binding:

• The method invoked is determined by the type of the accessed object.
• Languages with dynamic method binding: Smalltalk, Modula 3, Java, Eiffel

Which is more efficient: static or dynamic method binding?

Which is more natural?
Static and Dynamic Method Binding in C++

Given C++’s focus on efficiency, its default is static method binding.

Dynamic method binding is available by declaring the method to be `virtual`.

```cpp
class A {
public:
    void f();
    virtual void g();
};

class B : public A {
public:
    void f();
    void g();
};

int main() {
    B b;
    A *a = &b;
    
    b.f(); // B::f
    b.g(); // B::g
    a->f(); // A::f
    a->g(); // B::g
}
```
Abstract Classes

An abstract method is a method that is required to be defined only in derived classes.

C++

class person {
    ... 
    virtual void print_mailing_label() = 0; 
    ... 
};

Java

class person {
    ... 
    abstract void print_mailing_label(); 
    ... 
};

An abstract class has at least one abstract method and thus cannot be instantiated.

If all methods are abstract, then all the class does is define an interface.
The **virtual method table** or **vtable** is an array of addresses of the virtual methods of the object.

**Overhead:** Two extra memory accesses.
Implementation of Single Inheritance

Record of derived class

- Append extra data members to the record of the base class.
- Provides trivial access to these members through pointers whose type is the base class.

Vtable of derived class

- Copy vtable of base class.
- Replace entries of overridden virtual methods.
- Append entries of virtual methods declared in derived class.

```
class A {
    int a;
    double b;
    char c;
public:
    virtual void f();
    int g();
    virtual int h();
    double k();
};

class B : public A {
    int d;
    public:
    void f();
    virtual double l();
    virtual double *m();
};
```
Inheritance and Type Checks

class A { ... }
class B : public A { ... }

A a;
B b;
A *x;
B *y;

x = \&b; // ok; references through q will use prefixes of b’s
  // data space and vtable

y = \&a; // static semantic error; a lacks the additional data and vtable
  // entries of an object of class B

y = x; // error, but q actually does point to an instance of B
class A { ... };  
class B : public A { ... };  

A a;  
B b;  
A *x;  
B *y;  

x = &b;  // ok; references through q will use prefixes of b’s  
     // data space and vtable  

y = &a;  // static semantic error; a lacks the additional data and vtable  
     // entries of an object of class B  

y = x;  // error, but q actually does point to an instance of B  

Is there a way to resolve the second error? It is not actually an error, but as it is, the compiler cannot tell.
Dynamic Cast

C++

- $y = \text{dynamic\_cast}<B*>(x)$;
- Permits the assignment if $x$ points to an object of class $B$ or a derived class. Returns a null pointer otherwise.
Dynamic Cast

C++

- $y = \text{dynamic\_cast<B*>(x)}$;
- Permits the assignment if $x$ points to an object of class $B$ or a derived class. Returns a null pointer otherwise.

Java

- Same semantics but with C-style cast syntax:
  
  $y = (B) x$;
Dynamic Cast

C++

- \( y = \text{dynamic\_cast}\langle\text{B*}\rangle(x); \)
- Permits the assignment if \( x \) points to an object of class \( \text{B} \) or a derived class. Returns a null pointer otherwise.

Java

- Same semantics but with C-style cast syntax:
  \[
  y = (\text{B}) \ x;
  \]

**Implementation:** Include in each vtable the address of a run-time type descriptor.
Dynamic Cast

C++

• \( y = \text{dynamic
d
cast}<B*>(x); \)
• Permits the assignment if \( x \) points to an object of class \( B \) or a derived class. Returns a null pointer otherwise.

Java

• Same semantics but with C-style cast syntax:
  \[ y = (B) x; \]

Implementation: Include in each vtable the address of a run-time type descriptor.

Note: C++ also supports C-style casts without type checks. This is more efficient but less safe.
Type Casting in C++

dynamic_cast<T*> p

- Converts to type T* if the object pointed to by p is of class T or of a derived class. Returns a null pointer otherwise.
- Possible only for classes derived from polymorphic base classes and only when run-time type information (RTTI) is enabled.

static_cast<T*> p and reinterpret_cast<T*> p

- Perform conversions between unrelated types.
- static_cast performs some minimal type checking, while reinterpret_cast makes a bit-for-bit copy.

const_cast<T*> p

- Does not perform any type conversion other than removing the const-ness of a pointer.
Multiple Inheritance

Multiple inheritance allows a derived class to have multiple baseclasses:

```cpp
class A : public B, public C { ... }
```

Implementation issues

- How to access objects of A through a baseclass pointer.
- How to allow overriding of methods of different base classes.
- ...

Semantic issues

- If a method m is defined in more than one base class, which method is invoked by `a.m()`, where a is of class A?
- If B and C are derived classes of a common base class D, does A have two or only one copy of each data member of D?
- ...

DALHOUSSIE UNIVERSITY

CSCI 3136: Principles of Programming Languages