Object-Oriented Programming

Chapter 10

Object-Oriented Programming

- Key elements:
  - Data hiding / Encapsulation
  - Inheritance
  - Dynamic method binding

Data hiding

- Data abstraction: control large software complexity
- *Data hiding*:
  - objects visible only where necessary
  - reduce cognitive load on programmer
  - global variables – no hiding
  - local variables – subroutines only but limited life
  - static variables – retained between invocations
  - modules as abstractions – *encapsulation*
    - subroutines, variables, types, etc. visible only inside module
    - *export* / *import* types
    - Java: `package`, C++: `namespace`
  - modules as types: the module *is* the type

Classes

- Class:
  - module as type
  - + inheritance
  - + dynamic method binding
- Object
  - instance of a class
  - object-oriented programming

Classes: Example

```
class list_err {                                   // exception
public:
    const char *description;
    list_err(const char *s) {description = s;}
};

class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;                                       // the actual data in a node
    list_node() {                                  // constructor
        prev = next = head_node = this;            // point to self
        val = 0;                                   // default value
    }
    list_node* predecessor() {
        if (prev == this || prev == head_node) return nullptr;
        return prev;
    }
    list_node* successor() {
        if (next == this || next == head_node) return nullptr;
        return next;
    }
```

Classes: Example (cont'd)

```
    bool singleton() {
        return (prev == this);
    }
    void insert_before(list_node* new_node) {
        if (!new_node->singleton())
            throw new list_err("attempt to insert node already on list");
        prev->next = new_node;
        new_node->prev = prev;
        new_node->next = this;
        prev = new_node;
        new_node->head_node = head_node;
    }
    void remove() {
        if (singleton())
            throw new list_err("attempt to remove node not currently on list");
        prev->next = next;
        next->prev = prev;
        prev = next = head_node = this;     // point to self
    }
    ~list_node() {                          // destructor
        if (!singleton())
            throw new list_err("attempt to delete node still on list");
    }
};
```

Classes: Example (cont'd)

```
class list {
    list_node header;
public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices
    int empty() {
        return header.singleton();
    }
    list_node* head() {
        return header.successor();
    }
    void append(list_node *new_node) {
        header.insert_before(new_node);
    }
    ~list() {                       // destructor
        if (!header.singleton())
            throw new list_err("attempt to delete nonempty list");
    }
};
```

- create an empty list:

```
list* my_list_ptr = new list
```

Classes

- Data members – *fields*:
  - `prev, next, head_node, val`
- Subroutine members – *methods*:
  - `predecessor, successor, insert_before, remove`
- Accessing current object:
  - `this` (C++), `self` (Objective-C), `current` (Eiffel)
- Object creation / destruction:
  - *constructors*: `list_node()` (same name as the class)
  - *destructors* (C++): `~list_node()`

Visibility

- `public`: visible to users
- `private`: invisible to users
- C++: what is not public is private

Inheritance

- Derived class – *inherits* base class's fields and methods

```
class queue : public list {       // queue derived from list
public:
    // no specialized constructor/destructor required
    void enqueue(int v) {
        append(new list_node(v));      // append inherited
    }[

    int dequeue()
        if (empty())
            throw new list_err("dequeue from empty queue");
        list_node* p = head();           // head inherited
        p->remove();
        int v = p->val;
        delete p;
        return v;
    }
};
```

Inheritance

- `queue`: *derived class, child class, subclass*
- `list`: *base class, parent class, superclass*
- public members of the base class are always visible to methods of the derived class
- public members of the base class are visible to users only if the class is publicly derived
- we can hide public members by `private` derivation
  - exceptions made with `using`

```
class queue : private list { ...
public:
    using list::empty;
```

Inheritance

- the opposite is also possible with `delete`:

```
class queue : public list { ...
     ...
     void append(list_node *new_node) = delete;
```

- C++ `protected`
  - visible to members of its class and classes derived from it

```
class derived : protected base { ...
```

Visibility – C++ rules

- Any class can limit visibility of its members:

member	class's methods	class's and descendant's methods	anywhere (class scope)
public	✓	✓	✓
protected	✓	✓	✗
private	✓	✗	✗

- A derived class can restrict visibility of base class members but can never increase it:
  - Exceptions: `using`, `delete`

member \ derived class	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Visibility

- Java, C#
  - `private, protected, public`
  - no `protected` or `private` derivation
  - derived class can neither increase nor restrict visibility
  - can hide a field or override a method by defining a new one with the same name
    - cannot be more restrictive than the base class version
  - Java `protected`: visible in the entire package

  - `static` fields and methods
    - orthogonal to the visibility by public/protected/private
    - belong to the class as a whole: *class* fields and methods

Generics

- Previous `list` has integers only
- *Generics* allow list of any type
  - C++: *templates*

```
template<typename V>
class list_node {
    list_node<V>* prev;
    list_node<V>* next;
    list_node<V>* head_node;
public:
    V val;
    list_node<V>* predecessor() { ...
    list_node<V>* successor() { ...
    void insert_before(list_node<V>* new_node) { ...
    ...
};
```

Generics

```
template<typename V>
class list {
    list_node<V> header;
public:
    list_node<V>* head() { ...
    void append(list_node<V> *new_node) { ...
    ...
};

template<typename V>
class queue : private list<V> {
    list_node<V> header;
public:
    using list<V>::empty;
    void enqueue(const V v) { ...
    V dequeue() { ...
    V head() { ...
};
```

Generics

```
typedef list_node<int> int_list_node;
typedef list_node<string> string_list_node;
typedef list<int> int_list;
...
int_list_node n(3);
string_list_node s(“boo!”);
int_list L;
L.append(&n);        // ok
L.append(&s);        // error
```

Initialization and Finalization

- Initialize – *Constructor*
- Choosing a constructor
  - Can specify several constructors – C++, Java, C#
  - overloading: differentiate by number and types of parameters

```
class list_node {
    ...
    list_node(int v) {
        prev = next = head_node = this;
        val = v;
    }
...
list_node element1(1);      // int val
list_node *e_ptr = new list_node(5) // heap
list_node element0();       // default; val=0
```

Initialization and Finalization

- References and Values
  - Python, Java: variables refer to objects
    - every object is created explicitly
  - C++: variable has an object as value
    - objects created explicitly or implicitly, as result of elaboration
    - C++ requires all objects initialized by constructors

```
foo b;        // calls 0-arg constructor foo::foo()
foo b(10, ‘x’); // calls foo::foo(int, char)

foo a;
foo b(a);   // calls copy constructor foo::foo(foo&)
foo b = a; // same thing (‘=‘ is not assignment)

foo a, b;   // calls foo::foo() twice
b = a; // assignment; calls foo::operator=(foo&)
```

Initialization and Finalization

- Execution order for constructors (C++)
  - base class constructor executed first
  - also constructors of member classes
  - can specify arguments in constructor's header

```
class foo : bar {
    mem1_t member1;      // mem1_t and
    mem2_t member2;      // mem2_t are classes
    ...
}
foo::foo (foo_param) : bar (bar_args),
    member1 (mem1_init_val), member2 (mem2_init_val) {
    ...
```

Initialization and Finalization

- Finalize – *Destructor*
  - destructor of derived class called first, then base
  - C++: used for storage reclamation (manual storage)
  - Example: queue derived from list
    - default destructor calls `~list` (throws exception if non-empty)
  - If we wish destruction of non-empty queue:

```
~queue() {
    while (!empty()) {
        list_node* p = contents.head();
        p->remove();
        delete p;
    }
}       // or

~queue() {
    while (!empty()) {
        int v = dequeue();
    }
}
```

Dynamic Method Binding

- Subtype
  - Class D derived from C such that D doesn't hide any publicly visible member of C
  - a D-object can be used anywhere a C-object is expected
  - derived class is a *subtype* of base class

```
class person { ...
class student : public person { ...
class professor : public person { ...
...
student s;
professor p;
...
person *x = &s;
person *y = &p;
```

Dynamic Method Binding

- Polymorphic subroutine

```
class person { ...
void person::print_label { ...
...
s.print_label(); // print_label(s)
p.print_label(); // print_label(p)
```

- What if we redefine `print_label` in the derived classes?

```
s.print_label(); // student::print_label(s)
p.print_label(); // professor::print_label(p)
```

Dynamic Method Binding

- What about this?

```
x->print_label(); // ??
y->print_label(); // ??
```

- *Static method binding*: use the types of the variables `x` and `y`
- *Dynamic method binding*: use the classes of objects `s` and `p` to which the variables refer
- Example:
  - list of students and professors
  - print label correctly for each – dynamic method binding
  - derived class definition *overrides* the base class definition

Dynamic Method Binding

- Dynamic method binding
  - run-time overhead
  - Python, Objective-C, Ruby, Smalltalk – all methods
  - Java, Eiffel – dynamic default
    - `final` (Java) or `frozen` (Eiffel) cannot be overridden
  - C++, C#, Ada95, Simula – static default
    - static: *redefining* method
    - dynamic: *overriding* method – `virtual`

```
class person {
public:
    virtual void print_label();
    ...
```

Dynamic Method Binding

- Abstract classes
  - may omit the body of virtual functions – *abstract method*

```
abstract class person {     // Java, C#
    ...
    public abstract void print_label();
    ...
class person {                   // C++
    ...
public:
    virtual void print_label() = 0;
    ...
```

  - C++ – abstract method is called *pure virtual method*
  - *Abstract class* – has at least one abstract method
    - base for *concrete* classes
  - *Interface* – Java, C#
    - classes with abstract methods only

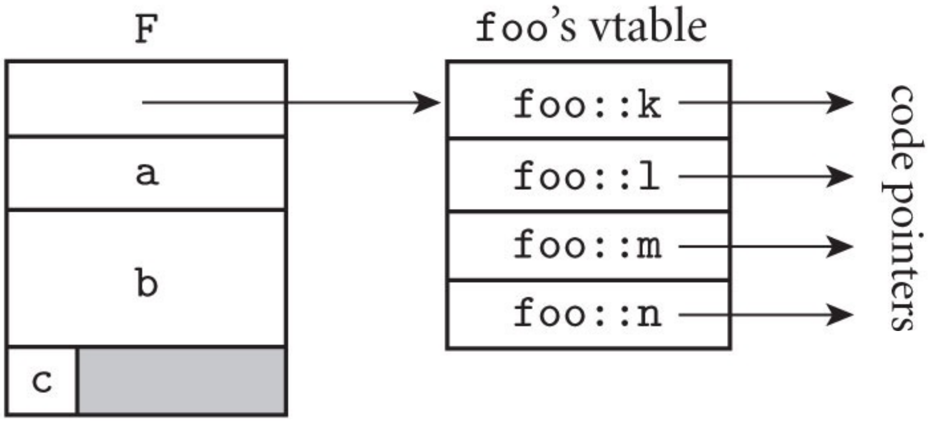
Dynamic member lookup

- Static method binding
  - the compiler knows which version of the method to call
- Dynamic method binding
  - reference variable must contain sufficient information for the code generated by compiler to find version at run time
- *Virtual method table* (*vtable*)
  - object implemented as a record whose first field contains the address of the vtable for the object's class
  - i^{th} entry of the vtable is the address of the code for the object's i^{th} virtual method

Dynamic member lookup

- Example

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```



Dynamic member lookup

- Dynamic method binding run-time overhead
- Example – code to call `f->m()`:
  - `f` is a pointer to an object of class `foo`
  - `m` is the third method of class `foo`

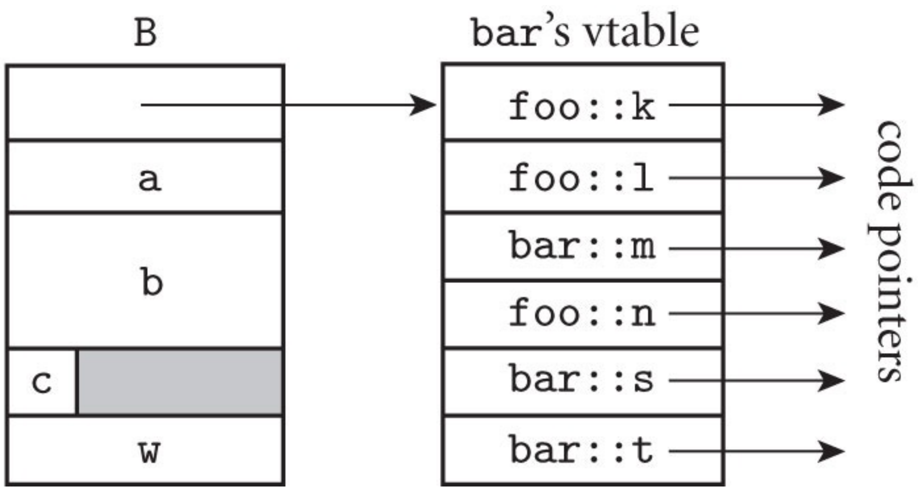
```
r1 := f
r2 := *r1           // vtable address
r2 := *(r2+(3–1)×4) // 4 = sizeof(address)
call *r2
```

  - this is two instructions longer than a call to statically identified method

Dynamic member lookup

- Inheritance

```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



Dynamic member lookup

- Example:

```
class foo { ...
class bar : public foo { ...
...
foo F;
bar B;
foo* q;
bar* s;
...
q = &B;      // ok; uses a prefix of B's vtable
s = &F;      // static semantic error
s = dynamic_cast<bar*>(q);    // run-time check
s = (bar*)(q);                // permitted but risky
                              // no run-time check
```