Object-Oriented Types

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

Object-Oriented Types

“The important thing about a language is what programs it can’t describe.”

—Nicklas Wirth

Inventor of Pascal

Three Attributes of OO Langauges

1. Encapsulation

   Hides data and procedures from other parts of the program.

2. Inheritance

   Creates new components by refining existing ones.

3. Dynamic Method Dispatch

   The ability for a newly-refined object to display new behavior in an existing context.

Encapsulation

How do you keep a large program partitioned?

Running Example: Linked List Stack

```
typedef struct node {
  int val;
  struct node *next;
} node_t;

node_t *list;

node_t *n =
  (node_t*) malloc(sizeof(node_t));
n->val = 3;
n->next = list;
list = n;
```

Linked List: Problems

Implementation exposed:

Malloc must be called explicitly every time node created.

Code that creates a node needs to know implementation of node.

Easy to forget part of the initialization of a node.

Difficult to change implementation: much code to update.

Global variable used for list: can’t have two.

Advantage: fast.

Linked List: Second Try

Put node creation into a function.

```
void push(int v)
{
  node_t *n =
    (node_t*) malloc(sizeof(node_t));
  n->val = v;
  n->next = list;
}

push(3);
```

Linked List 2

Advantages:

Easier-to-use: push operation free of implementation details.

Changes lead to less code to update.

Disadvantages:

Other program code can still create and modify list nodes.

Still using a global variable for the list.

Difficult to reuse this code.

Slower.

Linked List: Third Try

```
node *new_list() { ... }
void push(node *list, int v) { ... }
void destroy_list(node *list) { ... }

node *l = new_list();
push(l, 3);
destroy_list(l);
```

Linked List 3

Advantages:

- Even easier to use.
- More flexible: can manage multiple lists.
- Changes lead to less code to update.

Disadvantages:

- Other program code can still create and modify list nodes: really want to hide the node type more.
- Implementation not completely hidden.
- “push” is probably too popular an identifier.
- Slow.

Encapsulation

A key technique for complexity management is isolation.

Put a simple interface on a complex object:

Reduces conceptual load: easier to think of the interface.

Provides fault containment: when something goes wrong, it’s easier to isolate.

Provides independence: implementation can be modified without affecting the rest of the program.

Linked List: Fourth Try in C++

```
class List {
  struct Node {
    Node(int v, Node *n) { val=v; next=n; }
    int val;
    Node * next;
  };
  Node * head;
public:
  List() { head = 0; }
  void push(int v) { head = new Node(v, head); }
  int pop() { int v = head->val;
     head = head->next; return v; }
};
```

Linked List 4

```
List l;
l.push(3);
l.push(2);
int a = l.pop(); // 2
int b = l.pop(); // 3
```

Linked List 4

Advantages:

- Implementation hidden: other parts of the program can’t see Node.
- Push, pop operations inextricably bound to the List class.
- “Constructor” guarantees List objects always initialized correctly.

Disadvantages:

- Implementation tied to integers.
- Adding functionality appears to require copying and rewriting.

Managers vs. Types

```
List *new_list();
void push_list(List*, int);
int pop_list(List*);

class List {
public: List();
  ~List();
  void push(int);
  int pop();
};
```

Constructors/destructors made explicit.

Operations implicitly bound to objects.

Inheritance

How do you modify reused code?

Inheritance

Say you want to use the linked list as a queue, not just a stack.

Common problem: have something almost, but not quite, what you need.

In C++, classes are closed: can’t be amended once defined.

Manager approach may or may not have this problem.

(e.g., Java’s packages can be extended)

Inheritance

```
class List {
         struct Node {... }; Node * head;
public:  List(); void push(int); int pop(); };

class CountedList : public List {
  int count;
public:
  CountedList() { count = 0; }
  void push(int v) { List::push(v); ++count; }
  int pop(int v)
     { --count; return List::pop(v); }
  int count() { return count; }
};
```

Inheritance

```
class List {
         struct Node {... }; Node * head;
public:  List(); void push(int); int pop();
};

class CountedList : public List {
public:
  CountedList() {}
  int count() { int c = 0; Node * t = head; while
       ++c; t=t->next; }
    return c;
  }
};
```

Inheritance

This doesn't work:

```
class List {
  struct Node {...};  // private: by default
  ...
public:
};

class CountedList : public List {
 int count()
   { int c = 0; Node * t = head; ... }
};
```

Inheritance and Encapsulation

Elements of a class can be

private	visible only to members of the class
protected	visible to class members and derived classes
public	visible to everybody

Encapsulation

```
class Ex { int pri1;  // Private by default
private:   int pri2;
protected: int pro;
public:    int pub;
  void foo() { pri1=1; pri2=2; pro=3; pub=4; }
};

Ex e;

e.pri1 = 3;  // Error: private
e.pri2 = 4;  // Error: private
e.pro = 2;   // Error: protected
e.pub = 1;   // OK
```

Encapsulation

```
class Ex { int pri1;  // Private by default
private:   int pri2;
protected: int pro;
public:    int pub;
  void foo() { pri1=1; pri2=2; pro=3; pub=4; }
};
class Ex2 : public Ex {
public: void bar() {
    pri1=1; pri2=2; // Error: private
    pro=3;          // OK: protected
    pub=2;          // OK
  }
};
```

Friends

C++ has a “friend” mechanism for bending the rules.

```
class Ex {
  friend class Foo;
  int priv;  // private
};
class Foo {
public: Foo(Ex e) { e.priv = 1; } // OK
};
class Bar {
public:  Bar(Ex e)
   { e.priv = 0; } // Error: priv is private
};
```

Access Control over Parents

```
class Parent {
public: int x;
};

class PubChild : public Parent {};

PubChild puc;
puc.x = 1; // OK

class PrivChild : private Parent {};

PrivChild pvc;
pvc.x = 1; // Error: x is private
```

Dynamic Method Dispatch

How do you mix new code with old?

Dynamic Method Dispatch

Say we had a routine that we wanted to use:

```
void print_list(List *l) {
  while ( !(l->empty()) ) {
    printf("%d ", l->pop());
  }
}
```

The code would be the same if we passed it an object derived from the List class.

The only difference would be the functions called by

```
l->empty()
l->pop()
```

Method Dispatch

What happens when you write

```
class Foo { public: void bar() { ... } };
Foo f;
f.bar();
```

The type of **f** is the class **Foo**.

Lookup member "**bar**," which is a method.

Generated code looks like

```
void Foo_bar(Foo* this) { ... };
Foo f;
Foo_bar(&f);
```

Method Dispatch

```
void print_list(List *l) {
  while ( !( l->empty()) ) {
    printf("%d ", l->pop() );
  }
}
```

becomes

```
void print_list(List *l) {
  while ( ! List_empty(l) ) ) {
    printf("%d ", List_pop(l) );
  }
}
```

Dynamic Method Dispatch

If we had a derived class,

```
class List { ... };
class Queue : public List { ... };
```

```
void print_list(List *l) {
  while ( ! List_empty(l) Queue_empty(l) ) ) {
    printf("%d ", List_pop(l) Queue_pop(l) );
  }
}
```

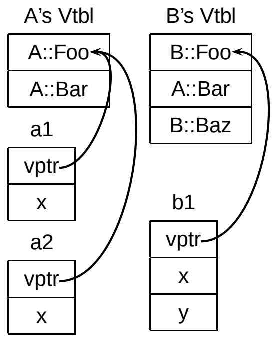
Actual type of **l** object should determine this.

Virtual Functions

The Trick: Add a "virtual table" pointer to each object.

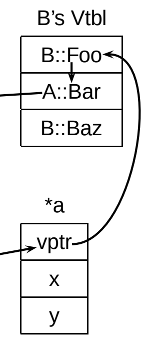
```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1, a2; B b1;
```



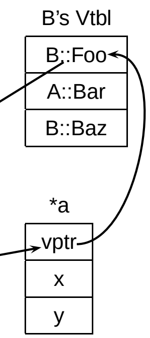
Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar()
    { do_something(); }
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};
A *a = new B;
a->Bar();
```



Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo()
    { something_else(); }
  virtual void Baz();
};
A *a = new B;
a->Foo();
```



Initialization and Finalization

How do objects begin and end their lives?

Initialization and Finalization

Most objects have some notion of a "consistent state."

```
class Box {
  int n, s, e, w;
  char *name;
public:
};
```

E.g., **n** $>$ **s**, **e** $>$ **w**, **name** is non-zero.

Information hiding intends to let us make the guarantee

> If the object is in a consistent state, applying any method leaves the method in a consistent state.

This is an inductive proof: need to start somewhere.

Initialization and Finalization

The idea of a constructor is to guarantee the object begins life in a consistent state.

Most OO languages guarantee that at least one constructor will be called on any new object from its class.

Often more than one constructor:

```
class Foo {
  int x, y;
public:
  Foo() { x = 0; y= 0; }
  Foo(int a, int b) { x = a; y = b; }
};
```

Constructors and Base Classes

```
class Foo { ...
public: Foo(int x) { ... }
};

class Bar : public Foo { ...
public: Bar() { ... } // Error: Foo(int)?
};
```

Need to specify arguments if the constructor demands it:

```
class Bar : public Foo { ...
public: Bar(int x) : Foo(x) { ... } // OK
};
```

Constructors and Base Classes

In Java,

```
class Foo {
  public Foo(int x) { ... }
}

class Bar extends Foo {
  public Bar(int x) { super(x); ... }
}
```

Easier in Java: guaranteed there's at most one base class.

Sort of odd: **`super(x)`** looks like a function call, but it can only be at the beginning of a constructor body.

Destructors

Memory management in my favorite languages:

C	Manual malloc() and free()
C++	Semi-automatic in constructors, destructors
Java	Fully automatic garbage collection
Tiger	No garbage collection ever

C, Tiger don't have objects: don't need destructors.

Java has automatic garbage collection: language's problem.

C++ needs destructors.

Destructors

```
class Foo {
  int *a;
public:
  Foo(int n) { a = new int[n]; }
  ˜Foo() { delete[] a; }
};
```

Storage for object automatically freed from the heap.

Anything you asked for explicitly needs to be freed explicitly.

Destructors

Main uses:

- Freeing resources (memory, file descriptors, etc.)
- Tracking statistics (how many things are "live")
- Maintaining consistency (informing owners)