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 Languages

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);
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.

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
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)

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

```cpp
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:

```cpp
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

```cpp
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;
```

Encapsulation

```cpp
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
        pub=2; // OK
    }
};
```

Friends

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

```cpp
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

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

class PubChild : public Parent {};
```

Access Control over Parents

```cpp
class PrivChild : private Parent {};
```

Dynamic Method Dispatch

How do you mix new code with old?

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

Dynamic Method Dispatch

Say we had a routine that we wanted to use:

```cpp
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

```cpp
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

```cpp
void Foo_bar(Foo* this) { ... }  
```

```cpp
Foo f;  
Foo_bar(&f);  
```

Dynamic Method Dispatch
If we had a derived class,

```cpp
class List { ... };  
class Queue : public List { ... };  
void print_list(List *l) {  
    while ( !List_empty(l) ) {  
        printf("%d ", List_pop(l) );  
    }  
}  
```

Becomes

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

Actual type of `l` object should determine this.

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

```cpp
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;  
```

Initialization and Finalization
How do objects begin and end their lives?

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

```cpp
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.

```cpp
A *a = new B;  
a->Bar();  
```

```cpp
klanguage
A *a = new B;  
a->Foo();  
```
Constructors and Base Classes

```cpp
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,

```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

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

Destructors

Main uses:

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

Memory management in my favorite languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Manual malloc() and free()</td>
</tr>
<tr>
<td>C++</td>
<td>Semi-automatic in constructors, destructors</td>
</tr>
<tr>
<td>Java</td>
<td>Fully automatic garbage collection</td>
</tr>
<tr>
<td>Tiger</td>
<td>No garbage collection ever</td>
</tr>
</tbody>
</table>

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

Java has automatic garbage collection: language’s problem.

C++ needs destructors.