Implementing Object-oriented Features

Object-oriented languages permit procedures and data to be packaged together into objects.

Example (C++)

```cpp
class text {
    char *contents;
    int size;
    text (char *s) {contents = s; size = 12;}
    void show()
    {   ...display(contents,Courier,size);...}
    int changesize (int d) {return size += d;}
};

text x("abc");
... x.show(); ...
... x.changesize(-1) + 17 ...
```

Note that class functions (like show) have implicit access to the data elements of the class (like contents).
Naive Implementation

Just add implicit this pointer parameter to each class function. Then can represent class in ordinary C (or close).

struct text {
    char *contents;
    int size;
};

struct text text_constructor (char *s) {
    struct text this;
    this.contents = s;
    this.size = 12;
    return this;
}

void text_show (struct text *this) {
    {display(this->contents,Courier,
        this->size);}  
}

int text_changesize (struct text *this, int d) {
    {return this->size += d;}
}

struct text x = text_constructor("abc");
... text_show (&x);...
... text_changesize(&x,-1) + 17;...
Inheritance

If a class B inherits from another class A, B contains all the data fields and functions that A does, plus more. Any B object can be used wherever an A object can be used.

Example: fancypText is subclass of text.

```cpp
class fancypText : text {
    font fnt;
    fancypText(char *, font) {...};
    void emphasize() {fnt = Italic;}
};
```

- Represent each subclass object by record that extends superclass object.

```cpp
struct fancypText {
    char *contents;
    int size;
    font fnt;
}
```

(Multiple inheritance is considerably harder.)
Inheritance (cont.)

• Superclass functions work on subclass objects too, because offsets of data fields they can see are the same!

```c
struct fancytext y;
text_changesize((struct text *) &y, -1);
```

• Subclass functions work only on subclass objects, since they may reference fields that only exist in subclass:

```c
void fancytext_emphasize(struct fancytext *this)
{this->fnt = Italic;}
```

• Since superclass and subclass have different sizes, direct assignment or pass-by-value of object records won’t work properly. Instead, should always manipulate pointers to class records. (C++ is unusual in not requiring this implicitly anyhow.)

```c
struct text x, *p;
struct fancytext y, *q;
x = (struct text) y;  /* Bad */
p = (struct text *) q; /* OK */
```
Virtual Functions

Sometimes it is appropriate to redefine the behavior of a function within a subclass:

```cpp
class text {
    ...
    virtual void show()
        {display(contents,Courier,size);}
}

class fancytext : text {
    font fnt;
    virtual void show()
        {display(contents,fnt,size);}
}

text *x = new text("abc");
fancytext *y = new fancytext("def",Arial);
if (???) x = y;
x->show();
```

Invokes fancytext::show if x is actually a fancy text; otherwise invokes text::show.

We can't tell at compile time which function should be invoked!
Solution: Function Pointers in Objects
struct text {
    char *contents;
    int size;
    void (*show)(struct text *this);
};

struct fancytext {
    ..as above, followed by...
    font fnt;
};

void text_show()
    {display(contents,Courier,size);} 

void fancytext_show()
    {display(contents,fnt,size);} 

struct text text_constructor (char *s) {
    ...
    result.show = &text_show;
    ...
}

struct fancytext fancytext_constructor 
    (char *s, font f) {
    ...
    result.show = &fancytext_show;
    ...
}

struct text *x; 
struct fancytext *y; 
if (????) x = y;  
(*((x->show)) (x);
Dispatch Tables

If there are many virtual functions in class, it’s better to separate out the function pointers for each defined subclass into a function dispatch table, which is pointed to from each object record for that subclass.

- The tables for all related classes are organized so that a given function always appears at the same table offset.
- Advantage: saves space in object records (though still uses lots of duplicate space in deep class hierarchies).
- Disadvantage: virtual function calls require an extra dereference.

This is essentially the implementation model used by C++, Java, Smalltalk, etc.

Multiple inheritance (C++) or interfaces (Java) requires more work, because we can’t maintain field or function entries in the same slot as the superclass when there are multiple superclasses!