Object implementation techiques

Key ideas

1. data hiding

2. inheritance

3. dynamic binding

Simple data hiding

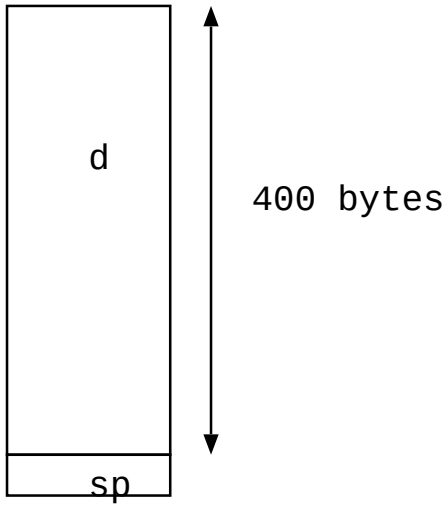
Consider a language in which you have objects which have private fields that can only be seen by their own methods.

Suppose it looks a bit like java

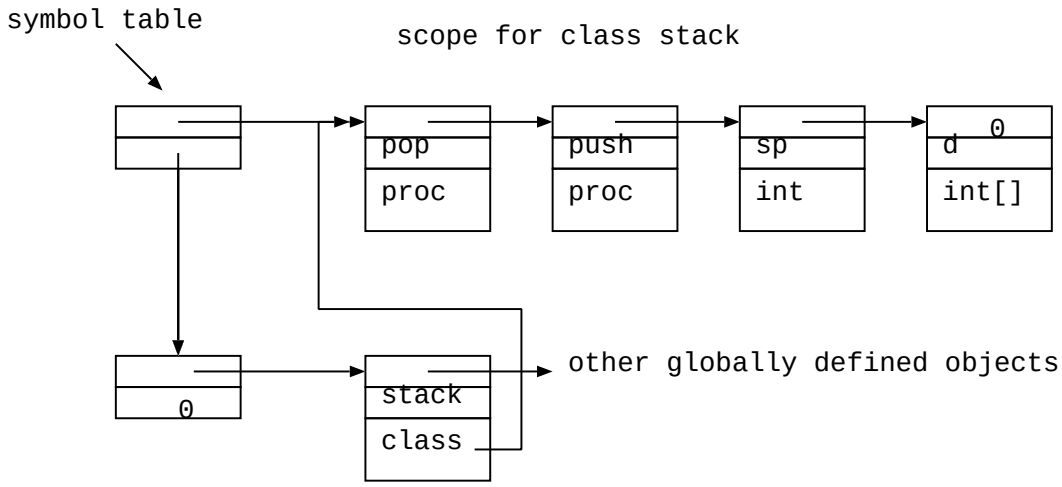
```
class stack{
 private int d[100],sp=0;
 public void push(int x){d[sp++]=x}
 public int pop(){return d[--sp];}
}
```

How could we implement this?

Well for a start we use record structures to represent the object. So in memory it looks like:



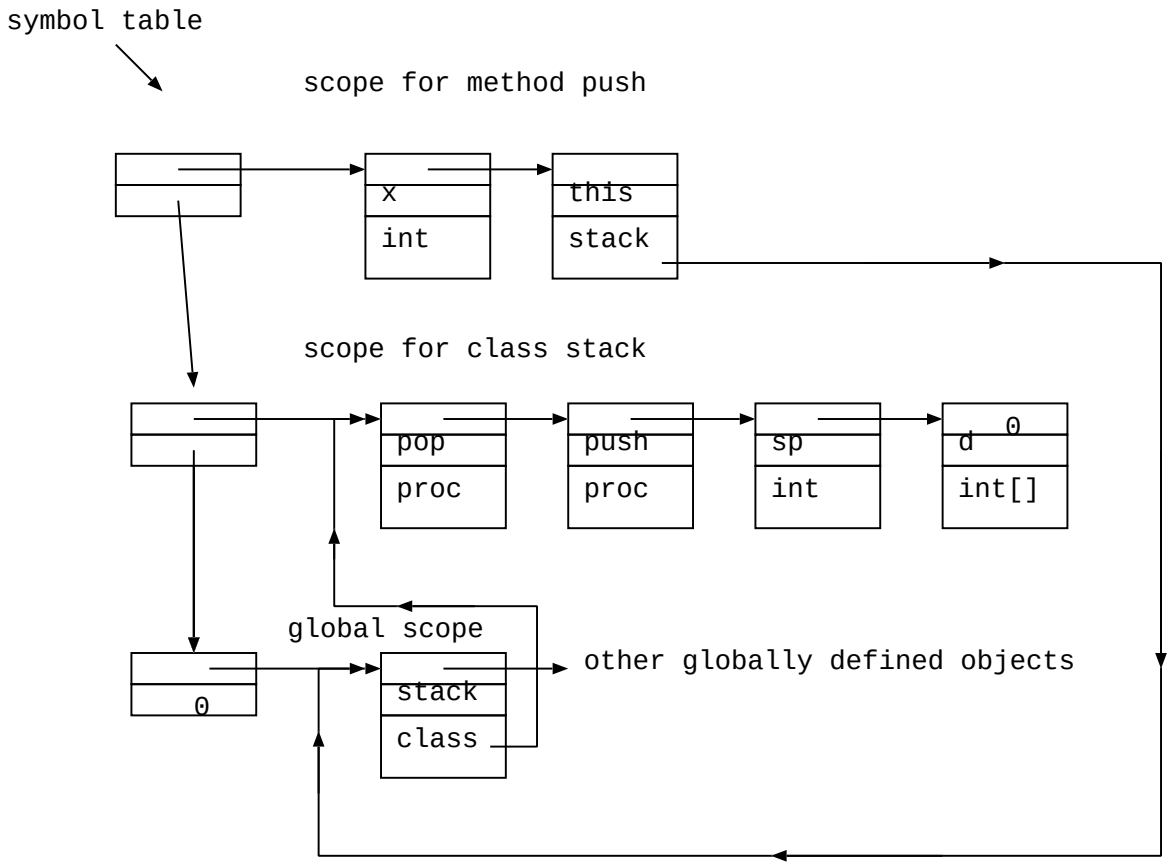
In the symbol table we enter a new scope level as we parse the class. This can be done by for instance structuring the symbol table as a linked list of linked list of identifier/type associations.



The procedural/method fields are stored in the symbol table with an additional hidden parameter:

```
int pop(stack this)
void push(stack this, int x)
```

When compiling push the symbol table looks like



Note that all fields of the stack object are available as subscripts of `this`. So we can translate all references to fields of the `stack` object, such as `d`, or `sp` to subscripted references.

the body of push is re-written

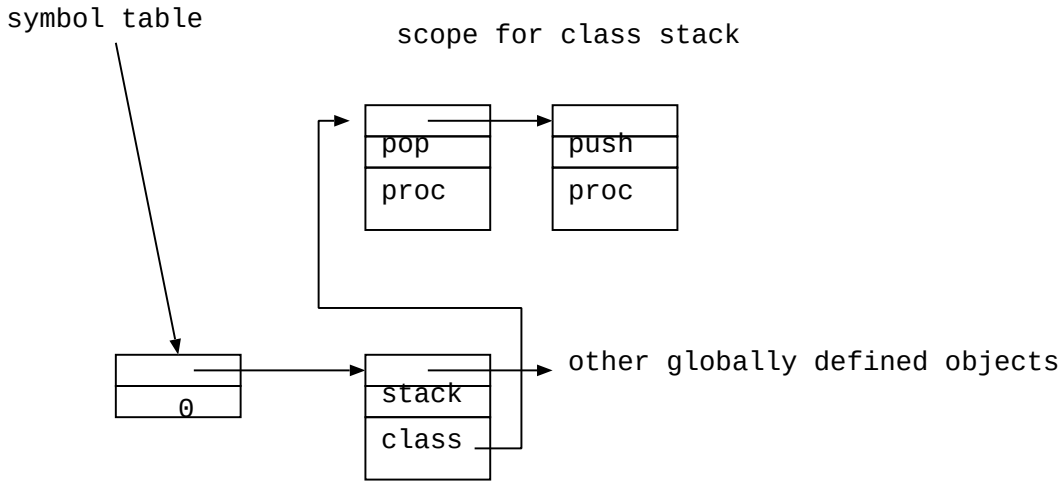
`d[sp++]=x` $\rightarrow$`this.d[this.sp++]=x`

We can do this because in this language (like C++) there are only 3 scope levels, global, class and local. If any identifier is found at class level it is rewritten with the `this` prefix.

When we leave the scope, we elide all the entries that are private, and set

symbol table ←symbol table. next

So that the smbol table now looks like:



We have now hidden the data fields of the object.

When we encounter a sequence like

```
stack s;
s .push(7);
```

The first line causes the compiler to reserve space in the local context for the stack s, and then to emit code to initialise it. This can be viewed in C terms as generating code for:

```
struct stack{int d[100],sp;}s;
s.sp=0;
```

the call `s.push(7)` is resolved as follows:

1. look up s to determine its type `stack`

2. encounter the . which indicates field dereference,

3. this tells us to open the scope for `stack`.

4. Search the scope for `stack` to find `push`

5. Substitute `s` as the first parameter of `push` to get: `stack$push(s,7)`

Note that all of this can be done statically. Thus in principle we can pre-process this sort

of object oriented code into a non object oriented language like C. Early C++ compilers were done this way.

Names of methods have to be prefixed by the class name to disambiguate them in the assembler code. Thus push becomes stack$push, where $ is some character or character sequence that is not allowed in source names but is allowed by the assembler.

Inheritance

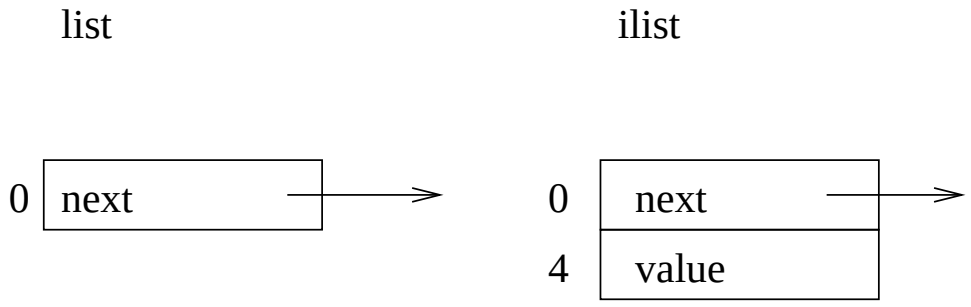
Let us now look at how we can define a class as extending another class. We first define a generic list processing class.

```
class list{
 private list next=null;
 public list getnext(){return next;}
 public boolean atend(){return next==null;}
 public void append(list x)
 {
   if (atend())next=x;
   else next.append(x);
 }
}
```

And now we make an extension that handles integer lists:

```
class ilist extends list{
  public int value;
  public ilist(int v,list n)
  { value=v; next=n;}
}
```

In terms of the data structure we have:



The field(s) of the child class come after the field(s) of the parent class in memory.

Constructors

Next consider the problem of constructors. Let us assume for now that all objects are allocated on the heap. This is not true for all object oriented languages (eg C++). Thus instances are created by some form of memory allocation statement:

```
list x = ilist(9,y);
```

How do we translate this?

```
list x=(list) malloc(8);
ilist.ilist(x,9,y);
```

and in the body of the ilist constructor

```
{ value=v; next=n;} becomes
{list.list(this);this.value=v;this.next=n;}
```

or in assembler

```
ilist$ilist: enter
        push dword[ebp+8]   ; push this
        call list$list
        pop eax             ; eax<-this
        mov ebx,[ebp+12]    ; ebx<-v
        mov [eax+4],ebx     ; this.value<-ebx
        mov ebx,[ebx+16]    ; ebx<-n
        mov [eax],ebx       ; this.next <- ebx
        ret
```

There is an implicit constructor in the list class indicated by the statement

```
next=null;
```

We have to create an anonymous function to initialise the list object, something like

```
list$list: enter
           mov eax, [ebp+8]
           mov [eax],0
           ret
```

Dynamic Binding

Up to now we have assumed that the linker knows where to find all of the methods of a class. This will not in general be true.

If you allow method m of class A to be over-riden in class B then when an instance x of B is passed into a context which sees it as an A and a call is made

x.m()

then the compiler and linker have now way of determining if the method m of class A or the method m of class B is to be called.

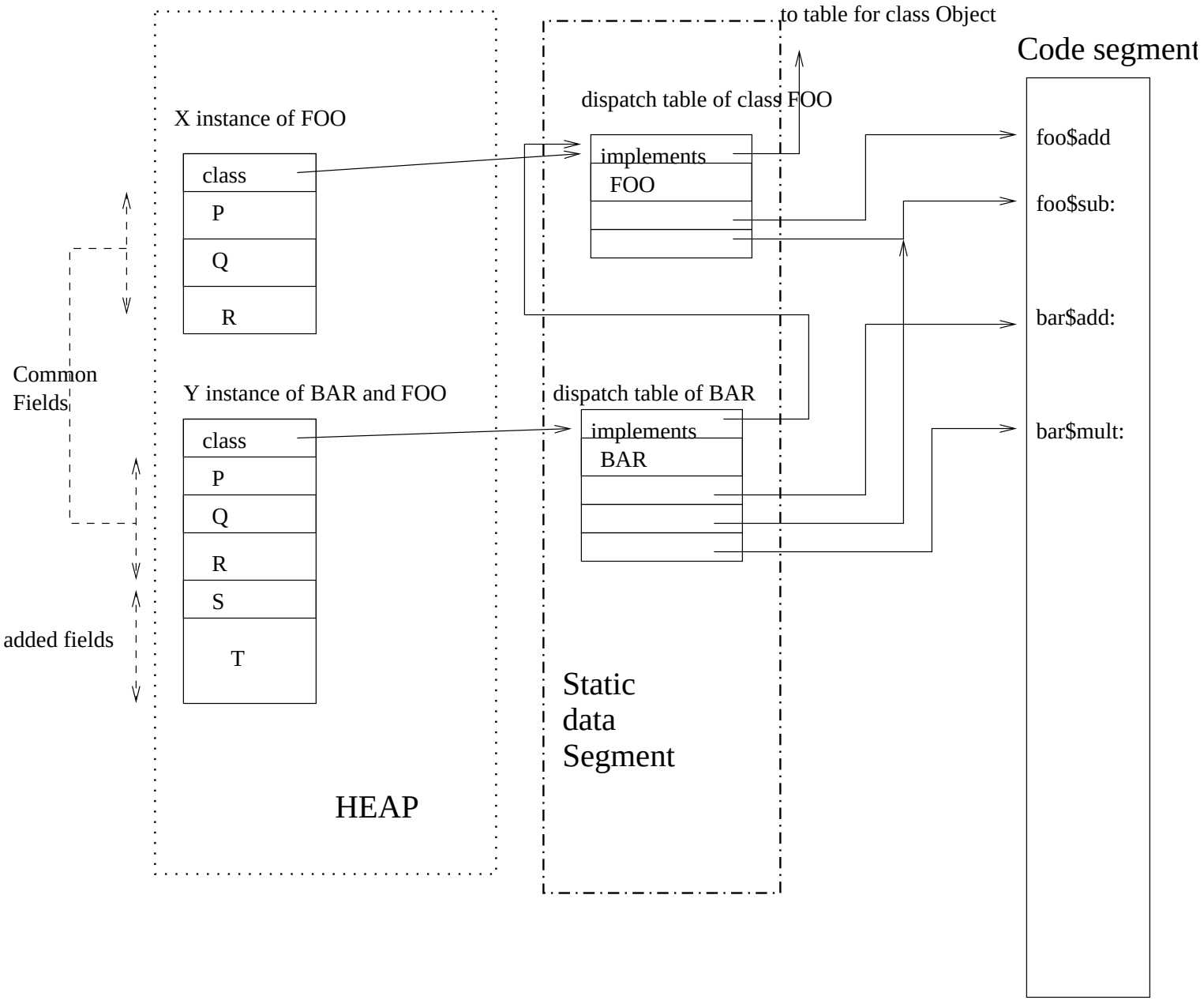
The answer to this is to provide at the top of each member of the class a pointer to a dispatch table.

1. Each class has a dispatch table

2. Each instance points at its dispatch table

3. The dispatch table points at the methods

Every problem in computing can be solved by enough indirection. (Willis)


to table for class Object
Code segment
dispatch table of class FOO
X instance of FOO
class
P
Q
R
implements
FOO
foo$add
foo$sub:
bar$add:
bar$mult:
Common
Fields
Y instance of BAR and FOO
dispatch table of BAR
class
P
Q
R
S
T
implements
BAR
added fields
Static
data
Segment
HEAP

Interfaces

How do we handle the situation in languages like Java where we have