CS 1622: Object-Oriented Languages

Jonathan Misurda
jmisurda@cs.pitt.edu

What is an Object?

Objects combine data and code as well as provide three things:

1. Polymorphism
2. Encapsulation
3. Inheritance

We will focus first on data and then move on to the methods.

Without inheritance, object data can be seen as a **record type** much like a struct in C.

Record Types

```
struct person {
        char name[10];
        int age;
};
```

A record type is an aggregation of data contiguously laid out in memory with fields usually given distinct names (instead of numerical indices as with arrays).

Each field is located at an offset from the beginning of the memory allocation.

We can hold this information about a type in its symbol table entry.

Offsets

```
struct person {
        char name[10];
        int age;
};
```

For this struct, we might produce this memory layout:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
+0, name										+10, age			

However, `int age` would not begin at an address of a multiple of 4, which can be a problem on architectures like MIPS.

Padding for Alignment

```
struct person {
        char name[10];
        int age;
};
```

We can force this to fit our memory alignment requirements by adding wasted padding bytes:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+0, name										Padding		+12, age			

Allocation

The declaration of a record type or object is just a template that guides the construction of one during runtime.

In C, structs can be allocated in one of three areas: the global data segment, the stack, or the heap.

In Java/MiniJava, objects can only be allocated via new on the heap.

Since the record types are contiguous, the allocation can be done all at once, with the proper size including the padding bytes for alignment.

Dynamic instances of the records or objects should be cleaned up by returning from the method (for stack allocated structs) or by calling `free()` (for `malloc()`), or by doing **garbage collection**.

Initialization

Different languages have different rules about the default initialization of record types.

In C, a global is initialized to zero, stack and heap allocations are not initialized.

In Java, all objects are initialized to zero (0, 0.0, false, null).

Access

```
struct person {
        char name[10];
        int age;
};
```

person
Type: struct person
Size: 16

- **name**
  Type: char
  Size: 10
  Offset: 0
- **age**
  Type: int
  Size: 4
  Offset: 12

C Code:

```
struct person *bob;

bob = malloc(sizeof(struct person));
bob.age = 20;
```

MIPS code:

```
li      $a0, 16
call    malloc
li      $t0, 20
lw      $t0, 12($v0)
```

Inheritance

In Object Oriented Languages, we can extend an object (a base class) and add new fields to it to create a derived class.

Some OO Languages restrict the inheritance relationship so that a given class may only have one parent class. This is the **single inheritance** model of Java.

Other OO Languages allow the composition of many base classes into a derived class. This is **multiple inheritance** in languages like C++.

Single Inheritance

The base class of a derived class has a set of fields with their associated offsets in memory.

We can often refer to a derived class via a base class pointer and access those fields:

```
Base b = new Derived();
b.fieldFromBase = 7;
```

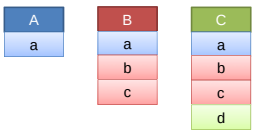
How should we lay out the derived class so that this is still possible?

Prefixing for Single Inheritance

```
class A {
        int a = 0;
}

class B extends A {
        int b = 0;
        int c = 0;
}

class C extends B {
        int d = 0;
}
```



Multiple Inheritance

In multiple inheritance, we may have several base classes for a single derived class. Since each base class may have its own fields, we cannot simply use prefixing to prepend the base classes to the derived class's data.

```
class A {
        int a = 0;
}

class B {
        int b = 0;
        int c = 0;
}

class C extends A, B {
        int d = 0;
}
```

Graph Coloring

Instead, we could create a set of non-overlapping offsets where certain spaces are left empty so that when overlaid, the derived class has the same offset locations as its base classes.

Since this is using the inheritance graph to determine interference, we can use graph coloring to "color" the offsets of the fields.

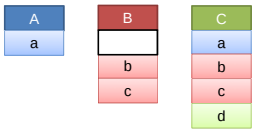
However, this only works if we know the fields at link time, dynamic loading can cause this to not work.

Graph Coloring

```
class A {
        int a = 0;
}

class B {
        int b = 0;
        int c = 0;
}

class C extends A, B {
        int d = 0;
}
```



Graph Coloring

Unfortunately, this approach can waste a lot of space per instance of an object.

Since there are likely to be many more instances than objects, we could instead use a descriptor table associated with the object to determine the field location in each instance.

This results in additional runtime work as we must load the descriptor's address, index it, and then use the resulting value to index the object in the instance.

Hopefully, however, the descriptor is loaded a small number of times and can be reused during any processing of an instance.

Static Data

Static data items exist independently of any instance of the class.

It then seems unlikely to store this data in the instances of the objects.

We could have a dedicated space to store the static fields (such as the global data segment) or we could use the descriptor table to point to a single location in memory where the static data lives.

Encapsulation

Public, private, protected, and package (default) scopes allow us to hide data items and ensure that their access goes through accessor and mutator functions.

Since these are about visibility and not a property of the machine, this is a compile-time construct that must be enforced by the type-checker during semantic analysis.

Methods

Methods are generally not stored in or near the object instances but rather turn into machine code as functions in non-OO languages do.

The major difference is that non-static methods then need to be able to find the instanced data that they are supposed to access and manipulate.

This is typically done through a hidden "this" pointer that is treated as an implicit parameter to the method:

`a.f(c)` becomes `f(a, c)` where `a` is referred to as `this` inside the body of `f`.

This can be considered a precolored node in the interference graph with the color of the first argument (such as $a0 in MIPS). It interferes with everything up until the last use (implicitly or explicitly) of this.

Implicit `this`

```
class A {
        int a = 0;

        void f() {
                a = 0;          //      this.a
                g();            //      this.g()  → g(this)
        }

        void g() {
                …
        }
}
```

Static Methods

If we wish to invoke a static method, it simply involves finding the right method based upon the type of object it is being called upon, not the type of the instance of the object.

At compile time, the call can simply be resolved to the appropriate label to jump to.

Dynamic Methods

When the type of the object determines which function should be called at runtime, we need a facility for the dynamic invocation of a method rather than depending on the compiler generating an appropriate label.

To this end, we can use our descriptor table as we did for object fields.

For multiple inheritance, global graph coloring works well.

Instanceof

At runtime, we may wish to determine whether a particular instance is a given object.

We can use the descriptors as a tag to determine the type.

Since we also have “is-a” membership for base classes, we potentially have to look in each of our base class’s descriptors for the type as well.

Typecasts

Static casts like C++ has may allow for unchecked runtime behavior that results in incorrect execution.

Runtime casts like Java and C++’s dynamic_cast require runtime checks to ensure that the type coercion is a safe one to perform.