x86/Linux Primitives for Object Orientation

- Object allocations need to be done on the heap.
  - The language run-time needs to provide explicit dynamic memory management primitives.
  - Rather than write one from scratch, may just be easier to use the standard `malloc()` and `free()` routines in the C standard library.
    - Cross-language interoperability issues.
- Instance methods need to be dynamically dispatched.
  - The `CALLQ *Opnd` form of the `CALLQ` instruction supports an indirect procedure call, similar to the indirect unconditional jump instruction.

LiveOak-3 Implementation on x86

- Given the greater flexibility of the architecture, it makes sense to implement the reference "object-record/class-record" model.
  - The class record variables can be allocated statically as global variables and initialized with the appropriate function pointers.
  - This is possible because we have global visibility of all program modules at compile-time.
    - If we had separate modules that were linked post-compilation, then we would need another level of indirection in the generated code, with placeholders for the appropriate variables that are allocated and initialized at link-, load-, or run-time.
  - To be complete, we should also have an `Object` class as the root of the object hierarchy.
- Any methods that cannot be overridden in sub-classes can be dispatched statically and need not be allocated in class method tables.
  - This may be indicated, e.g., by a keyword like `final`.
  - Name mangling is still required for method disambiguation.

Run-Time Data Structures

- Class objects
  - Layout, allocation, placement.

```
class Point2D {
  int x, y;
  Point2D(int, int);
  int get_x(void);
  int get_y(void);
  double dist(void);
};
class Point3D extends Point2D {
  int z;
  Point3D(int, int, int);
  int get_z(void);
  double dist(void); // overrides
};
Point2D p1, p2;
Point3D q = new Point3D(10,10,10);
```

Run-Time Data Structures

- Class objects
  - Layout, allocation, placement.
- Class instance objects
  - Layout, allocation, placement.
  - Interactions with inheritance.

```
class Point2D {
  int x, y;
  Point2D(int, int);
  int get_x(void);
  int get_y(void);
  double dist(void);
};
class Point3D extends Point2D {
  int z;
  Point3D(int, int, int);
  int get_z(void);
  double dist(void); // overrides
};
Point2D p1, p2;
Point3D q = new Point3D(10,10,10);
```

Run-Time Data Structures

- Class objects
  - Layout, allocation, placement.
- Class instance objects
  - Layout, allocation, placement.
  - Interactions with inheritance.
- Method tables.
  - Inline/out-of-line.
  - Hierarchical/flattened.

```
class Point2D {
  int x, y;
  Point2D(int, int);
  int get_x(void);
  int get_y(void);
  double dist(void);
};
class Point3D extends Point2D {
  int z;
  Point3D(int, int, int);
  int get_z(void);
  double dist(void); // overrides
};
Point2D p1, p2;
Point3D q = new Point3D(10,10,10);
```

Run-Time Data Structures

- Class objects
  - Layout, allocation, placement.
- Class instance objects
  - Layout, allocation, placement.
  - Interactions with inheritance.
- Method tables.
  - Inline/out-of-line.
  - Hierarchical/flattened.
- Constructors.

```
class Point2D {
  int x, y;
  Point2D(int, int);
  int get_x(void);
  int get_y(void);
  double dist(void);
};
class Point3D extends Point2D {
  int z;
  Point3D(int, int, int);
  int get_z(void);
  double dist(void); // overrides
};
Point2D p1, p2;
Point3D q = new Point3D(10,10,10);
```