You are here
Representing Data at Runtime

- Source language types
  - int, boolean, string, object types

- Target language types
  - Single bytes, integers, address representation

- Compiler should map source types to some combination of target types
  - Implement source types using target types
Basic Types

- int, boolean, string, void
- Arithmetic operations
  - Addition, subtraction, multiplication, division, remainder
- Could be mapped directly to target language types and operations
Pointer Types

- Represent addresses of source language data structures
- Usually implemented as an unsigned integer
- Pointer dereferencing – retrieves pointed value

- May produce an error
  - Null pointer dereference
  - when is this error triggered?
  - how is this error produced?
Object Types

- Basic operations
  - Field selection
    - computing address of field, dereferencing address
  - Copying
    - copy block or field-by-field copying
  - Method invocation
    - Identifying method to be called, calling it

- How does it look at runtime?
Object Types

class Foo {
   int x;
   int y;

   void rise() {...}
   void shine() {...}
}

Runtime memory layout for object of class Foo

Compile time information
Field Selection

```c
Foo f;
int q;
q = f.x;
```

![Diagram showing field selection and memory layout](image)

**Runtime memory layout** for object of class Foo

- `DispacthVectorPtr`
- `x`
- `y`

**Compile time** information

- `base pointer`
- `field offset from base pointer`

- `MOV f, %EBX`
- `MOV 4(%EBX), %EAX`
- `MOV %EAX, q`
Object Types - Inheritance

class Foo {
    int x;
    int y;

    void rise() {...}
    void shine() {...}
}

class Bar extends Foo{
    int z;
    void twinkle() {...}
}
Object Types - Polymorphism

class Foo {
    ...
    void rise() {...}
    void shine() {...}
}

class Bar extends Foo{
    ...
}

class Main {
    void main() {
        Foo f = new Bar();
        f.rise();
    }
}
Dynamic Binding

- Finding the right method implementation
- Done at runtime according to object type
- Using the Dispatch Vector (a.k.a. Dispatch Table)

```java
class Foo {
    ...
    void rise() {...}
    void shine() {...}
}

class Bar extends Foo {
    void rise() {...}
}

class Main {
    void main() {
        Foo f = new Bar();
        f.rise();
    }
}
```
Dispatch Vectors in Depth

- Vector contains addresses of methods
- Indexed by method-id number
- A method signature has the same id number for all subclasses

```java
class Foo {
    ...
    void rise() { ... } 0
    void shine() { ... } 1
}

class Bar extends Foo {
    void rise() { ... } 0
}

class Main {
    void main() {
        Foo f = new Bar();
        f.rise();
    }
}
```
Dispatch Vectors in Depth

class Foo {
    ...
    void rise() {...} 0
    void shine() {...} 1
}

class Bar extends Foo{
    void rise() {...} 0
}

class Main {
    void main() {
        Foo f = new Foo();
        f.rise();
    }
}

Pointer to Foo

f → DVPtr
    x
    y

Object layout

Using Foo’s dispatch table

Diagram showing the dispatch vector and method code.
Representing dispatch tables

class A {
    void rise() {...}
    void shine() {...}
    static void foo()
    {...}
}
class B extends A {
    void rise() {...}
    void shine() {...}
    void twinkle() {...}
}

# data section
.data
    .align 4
_DV_A:
    .long _A_rise
    .long _A_shine
_DV_B:
    .long _B_rise
    .long _B_shine
    .long _B_twinkle
Multiple Inheritance

class C {
    field c1;
    field c2;
    void m1() {...}
    void m2() {...}
}
class D {
    field d1;
    void m3() {...}
    void m4() {...}
}
class E extends C,D{
    field e1;
    void m2() {...}
    void m4() {...}
    void m5() {...}
}
Runtime checks

- generate code for checking attempted illegal operations
  - Null pointer check
    - MoveField, MoveArray, ArrayLength, VirtualCall
    - Reference arguments to library functions should not be null
  - Array bounds check
  - Array allocation size check
  - Division by zero
  - ...

- If check fails jump to error handler code that prints a message and gracefully exists program
Null pointer check

# null pointer check
cmp $0, %eax
je labelNPE

Single generated handler for entire program

labelNPE:
push $strNPE   # error message
call __println
push $1        # error code
call __exit
Array bounds check

```assembly
# array bounds check
mov -4(%eax),%ebx  # ebx = length
mov $0,%ecx       # ecx = index
cmp %ecx,%ebx
jle labelABE      # ebx <= ecx ?
cmp $0,%ecx
jl  labelABE      # ecx < 0 ?

Single generated handler for entire program

labelABE:
push $strABE       # error message
call __println
push $1             # error code
call __exit
```
Array allocation size check

# array size check
cmp $0,%eax   # eax == array size
jle labelASE  # eax <= 0 ?

Single generated handler for entire program

labelASE:
   push $strASE  # error message
call __println
   push $1       # error code
call __exit
Automatic Memory Management

- automatically free memory when it is no longer needed
- not limited to OO programs, we show it here because it is prevalent in OO languages such as Java
  - also in functional languages
- approximate reasoning about object liveness
- use reachability to approximate liveness
- assume reachable objects are live
  - non-reachable objects are dead

- Three classical garbage collection techniques
  - reference counting
  - mark and sweep
  - copying
GC using Reference Counting

- add a reference-count field to every object
  - how many references point to it
- when \( rc == 0 \) the object is non reachable
  - non reachable => dead
  - can be collected (deallocated)
Managing Reference Counts

- Each object has a reference count o.RC
- A newly allocated object o gets o.RC = 1
  - why?

- write-barrier for reference updates
  update(x,old,new) {
    old.RC--;
    new.RC++;
    if (old.RC == 0) collect(old);
  }

- collect(old) will decrement RC for all children and recursively collect objects whose RC reached 0.
Cycles!

- cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
- several approaches for dealing with cycles
  - ignore
  - periodically invoke a tracing algorithm to collect cycles
  - specialized algorithms for collecting cycles
GC Using Mark & Sweep

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object

- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects
GC Using Mark & Sweep

mark_sweep() {
  for Ptr in Roots mark(Ptr)
  sweep()
}

mark(Obj) {
  if mark_bit(Obj) == unmarked {
    mark_bit(Obj)=marked
    for C in Children(Obj) mark(C)
  }
}

Sweep() {
  p = Heap_bottom
  while (p < Heap_top)
    if (mark_bit(p) == unmarked) then free(p)
    else mark_bit(p) = unmarked;
    p=p+size(p)
}
Copying GC

- partition the heap into two parts: old space, new space

- GC
  - copy all reachable objects from old space to new space
  - swap roles of old/new space
Example

```
old  new
A    B
D    C
      E
```

Roots
Example
The End