CPSC 213

Introduction to Computer Systems

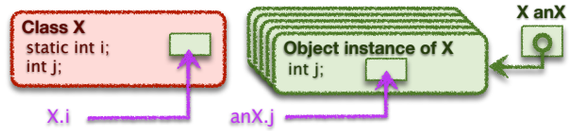
Unit 1c

Instance Variables and Structs

Reading

- Companion
  - 2.4.4-2.4.6
- Textbook
  - 2ed: 3.9.1
  - 1ed: 3.9.1

Instance Variables



- Variables that are an instance of a class or struct
  - created dynamically
  - many instances of the same variable can co-exist
- Java vs C
  - Java: ***objects*** are instances of non-static variables of a ***class***
  - C: ***structs*** are named variable groups, instance is also called a struct
- Accessing an instance variable
  - requires a reference to a particular object (pointer to a struct)
  - then variable name chooses a variable in that object (struct)

Structs in C (S4-instance-var)

```
struct D {
  int e;
  int f;
};
```

≈

```
class D {
  public int e;
  public int f;
}
```

- A struct is a
  - collection of variables of arbitrary type, allocated and accessed together
- Declaration
  - similar to declaring a Java class without methods
  - name is "struct" plus name provided by programer
  - static `struct D  d0;`
  - dynamic `struct D* d1;`
- Access
  - static `d0.e = d0.f;`
  - dynamic `d1->e = d1->f;`

Struct Allocation

```
struct D {
  int e;
  int f;
};
```

- Static structs are allocated by the compiler

`struct D  d0;`

Static Memory Layout

0x1000: value of d0.e
0x1004: value of d0.f

- Dynamic structs are allocated at runtime
  - the variable that stores the struct pointer may be static or dynamic
  - the struct itself is allocated when the program calls **malloc**

`struct D* d1;`

Static Memory Layout

0x1000: value of d1


```
struct D {
  int e;
  int f;
};
```

- runtime allocation of dynamic struct

```
void foo () {
  d1 = (struct D*) malloc (sizeof(struct D));
}
```

- assume that this code allocates the struct at address 0x2000

0x1000: 0x2000

0x2000: value of d1->e
0x2004: value of d1->f

Struct Access

```
struct D {
  int e;
  int f;
};
```

- Static and dynamic differ by an extra memory access
  - dynamic structs have dynamic address that must be read from memory
  - in both cases the offset to variable from base of struct is static

`d0.e = d0.f;`

`m[0x1000] ← m[0x1004]`

```
r[0]  ← 0x1000
r[1]  ← m[r[0]+4]
m[r[0]] ← r[1]
```

`d1->e = d1->f;`

`m[m[0x1000]+0] ← m[m[0x1000]+4]`

```
r[0]  ← 0x1000
r[1]  ← m[r[0]]
r[2]  ← m[r[1]+4]
m[r[1]] ← r[2]
```

load d1


```
struct D {
  int e;
  int f;
};
```

`d0.e = d0.f;`

```
r[0]  ← 0x1000

r[1]  ← m[r[0]+4]
m[r[0]] ← r[1]
```

```
ld $0x1000, r0  # r0 = address of d0
ld 4(r0), r1    # r0 = d0.f
st r1, (r0)     # d0.e = d0.f
```

`d1->e = d1->f;`

```
r[0]  ← 0x1000
r[1]  ← m[r[0]]
r[2]  ← m[r[1]+4]
m[r[1]] ← r[2]
```

load d1

```
ld $0x1000, r0  # r0 = address of d1
ld (r0), r1     # r1 = d1
ld 4(r1), r2    # r2 = d1->f
st r2, (r1)     # d1->e = d1->f
```

- The revised load/store base plus offset instructions
  - dynamic base address in a register plus a static offset (displacement)

`ld 4(r1), r2`

The Revised Load-Store ISA

- Machine format for base + offset
  - note that the offset will in our case always be a multiple of 4
  - also note that we only have a single hex digit in instruction to store it
  - and so, we will store offset / 4 in the instruction
- The Revised ISA

Name	Semantics	Assembly	Machine
load immediate	r[**d**] ← **v**	ld $**v**, **rd**	**0**d-- vvvvvvvv
load base+offset	r[**d**] ← m[r[**s**]+(o=p*4)]	ld o(**rs**), **rd**	**1psd**
load indexed	r[**d**] ← m[r[**s**]+4*r[**i**]]	ld (**rs**,**ri**,4), **rd**	**2sid**
store base+offset	m[r[**d**]+(o=p*4)] ← r[**s**]	st **rs**, o(**rd**)	**3spd**
store indexed	m[r[**d**]+4*r[**i**]] ← r[**s**]	st **rs**, (**rd**,**ri**,4)	**4sdi**