Introduction to Computer Systems

Unit 1b
Static Scalars and Arrays

1 Static Variables, Built-In Types (S1-global-static)

- **Java**
  - static data members are allocated when the class is compiled, not when the program starts
  - can store scalar, string, array, or object references

- **C++**
  - global variables and any other variable declared static
  - can be static scalars, arrays or structs or pointers (pointers later)

2 Static Variable Allocation

- **Java**
  - variables are allocated in the class file

- **C++**
  - static variable allocation is done at compile-time

3 Static Variable Access (scalars)

- **Java**
  - access static variables via the class name

- **C++**
  - access static variables via the class name

4 Static Variable Access (arrays)

- **Java**
  - access array elements using the array name followed by the index

- **C++**
  - access array elements using the array name followed by the index

5 Static Memory Layout

- **Java**
  - memory layout is fixed at compile-time

- **C++**
  - memory layout is fixed at compile-time

6 Static vs dynamic computation

- **Java**
  - static variables are initialized when the class is loaded

- **C++**
  - static variables are initialized when the class is loaded

7 Key Observation

- **Java**
  - static variables are initialized when the class is loaded

- **C++**
  - static variables are initialized when the class is loaded

8 Generalizing

- **Java**
  - static variables are initialized when the class is loaded

- **C++**
  - static variables are initialized when the class is loaded

9 When is space for a allocated (when is its address determined)?

- **Java**
  - during class loading

- **C++**
  - during class loading

10 The Big Picture

- **Java**
  - build model of execution
  - for Java and C programs
  - by examining language features
  - and deciding how they are implemented by the machine

- **C++**
  - build model of execution
  - for Java and C programs
  - by examining language features
  - and deciding how they are implemented by the machine

11 What is required

- **Java**
  - design an ISA into which programs can be compiled

- **C++**
  - design an ISA into which programs can be compiled

12 Our approach

- **Java**
  - examine code snippets that exemplify each language feature in turn
  - look at Java and C, passing to dig deeper when C is different from Java

- **C++**
  - examine code snippets that exemplify each language feature in turn
  - look at Java and C, passing to dig deeper when C is different from Java

13 The simulator is an important tool

- **Java**
  - machine execution is hard to visualize without it

- **C++**
  - machine execution is hard to visualize without it

14 Static Variable Allocation

- **Java**
  - variables are allocated in the class file

- **C++**
  - variables are allocated in the class file

15 Static Memory Layout

- **Java**
  - memory layout is fixed at compile-time

- **C++**
  - memory layout is fixed at compile-time

16 Static vs dynamic computation

- **Java**
  - static variables are initialized when the class is loaded

- **C++**
  - static variables are initialized when the class is loaded

17 Key Observation

- **Java**
  - static variables are initialized when the class is loaded

- **C++**
  - static variables are initialized when the class is loaded

18 Generalizing

- **Java**
  - what if it’s a = a + 2 or a = b? or a = foo? or a = 0?

- **C++**
  - what if it’s a = a + 2 or a = b? or a = foo? or a = 0?
**Static Variable Access (static arrays)**

- **Key Observation**
  - compiler does not know address of b[a] unless it can ... Memory Layout
  - 0x1000: value of a
  - 0x2000: value of b[0]
  - 0x2004: value of b[1]
  - ...
  - 0x2024: value of b[9]

- In assembly language
- Comparing static and dynamic arrays
- what is the benefit of static arrays?
- what is the benefit of dynamic arrays?

**Designing ISA for Static Variables**

- **Requirements for scalars**
  - load constant into register
  - store value in register/memory at constant address
  - load value in memory at address into a register

- **Additional requirements for arrays**
  - value in register/memory at address in register +4 plus constant
  - load value in memory at address in register +4 into a register

**The compiler’s semantic translation**

- It uses these instructions to compile the program snippet

**ISA Specification for these 5 instructions**

**In these instructions**

**Addressing Modes**

- **Immediate** constant value stored in instruction
- **Register** operand is register number, register stores value
- **Base-offset** operand in register number, register stores value of memory address
- **Indexed** two register number operands, store base memory address and value of value of b[k]

**Global Dynamic Array**

- **Java**
  - array variable stores reference to array allocated dynamically with new statement

- **C**
  - array variables can store static arrays or pointers to arrays allocated dynamically with malloc
to malloc library procedure

**How C Arrays are Different from Java**

- **Terminology**
  - use the term pointer instead of reference: they mean the same thing

- **Declaration**
  - the type is a pointer to the type of its elements, indicated with a *

- **Allocation**
  - malloc allocates a block of bytes; no type; no constructor

- **Type Safety**
  - any pointer can be cast to any pointer type

- **Bounds checking**
  - C performs no array bounds checking
  - out-of-bounds access manipulates memory that is not part of array
  - this is the major source of virus vulnerabilities in the world today

**Question: Can array bounds checking be performed statically?**

- what does this say about a tradeoff that Java and C take differently?

**C and Java Arrays and Pointers**

- **In both languages**
  - an array is a list of items of the same type
  - array elements are named by non-negative integers that start with 0

- **In Java**
  - variable a stores a pointer to the array
  - b[0] = 0 means m[0] = x * sizeOf(array-element) = 0

- **In C**
  - variable a can store a pointer to the array or the array itself
  - b[0] = 0 means m[0] = x * sizeOf(array-element) = 0
  - or m[0] = x * sizeOf(array-element) = 0

**Example**

- The following two C programs are identical

- For array access, the compiler would generate this code

- multiplying the index 4 by 4 (size of integer) to compute the array offset

- So, what does this tell you about pointer arithmetic in C?
Pointer Arithmetic in C

‣ Its purpose
• an alternative way to access dynamic arrays to the a[i]
‣ Adding or subtracting an integer index to a pointer
• results in a new pointer of the same type
• value of the pointer is offset by index times size of pointer’s referent
‣ for example:
  • adding 3 to an int* pointer is a pointer value 12 larger than the original
‣ Subtracting two pointers of the same type
• results in an integer
• gives number of referent-type elements between the two pointers
• for example:
  • if a[7]: `int i=7; &i - &a[7] = 5 == (a+7) - (a+2)
‣ other operators
• & X the address of X
• * X the value X points to

What Java does that C doesn’t
- static scalars and arrays
- the compiler knows the address of the scalar value or array
- the compiler does not know the address the array
What C does that Java doesn’t
- static arrays
- arrays can be accessed using pointer dereferencing operator
- arithmetic on pointers
What Java does that C doesn’t
- typesafe dynamic allocation
- automatic array-bounds checking

Looking more closely

And in assembly language