CS-034
To be or not to be (efficient)

T. Shakespeare Doeppner

P. Hugo Van Hentenryck

C/C++ versus Java

Where does the time go?

- arrays
- virtual methods
- stack machine: JVM
- garbage collection

Concepts

- compiler optimizations
- x86 architecture
- Handles

x86 architecture: registers

Control Registers

- %ebp (frame ptr), %esp (stack ptr), %eis (pc)

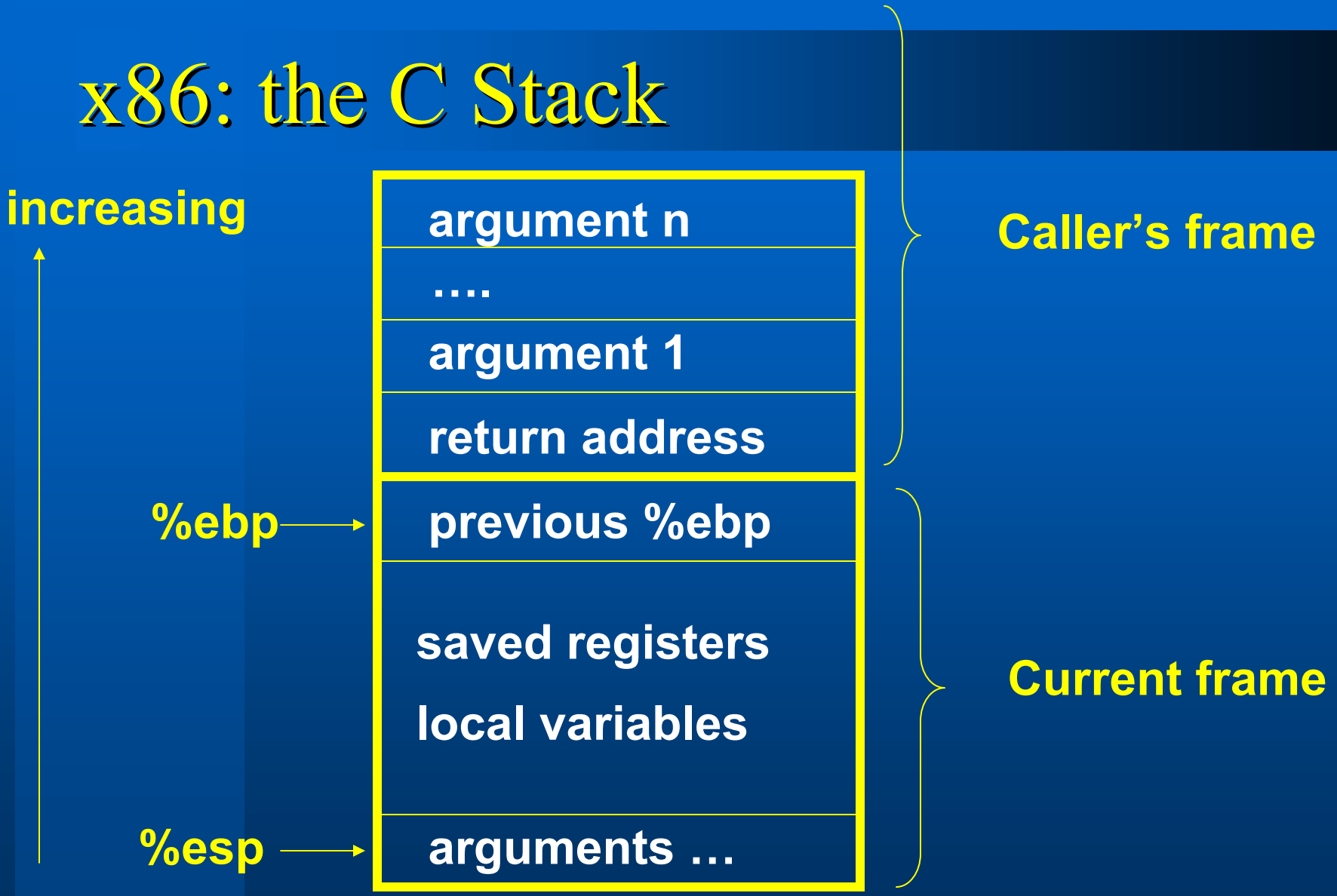
S-Registers (must be saved by subroutines)

- %ebx, %esi

T-Registers (must not be saved by subroutines)

- %eax, %ecx, %edx

x86: the C Stack



x86 assembly language

movel

- move from register/memory to register/memory

pushl/popl

- push and pop from the stack

addl, subl, multl, xorl, andl

- arithmetic expression

leal

- load immediate (load effective address)

jmp

- jump

x86 assembly language

Conditional jumps

- two steps
- first compare and set flags
- then jump based on the flags

Function calls

- **call:** pushes %esi on the stack and jump to subroutine
- **ret:** pops the address on the stack and jump there
- **leave:** restore %ebp and %esp

Compiler Optimizations

- **Register allocation**
- **Inlining**
- **Deadcode elimination**
- **Subexpression factorization**
- **peephole optimization**
- **…**

Compiler Optimizations

Register allocation

- compilers try to allocate local variables in registers
- very useful for expressions

Not always possible

- no enough registers
- arrays and structures
- & may take the address of a local variable

Compiler Optimizations

Inlining

- compilers try to remove function calls
- replace them with the body

Not always possible

- recursion
- size of the code

C 2 x86

```
int sum(int a,int b)
{
   return a + b;
}
```

```
sum:
    pushl   %ebp                push ebp
    movl    %esp, %ebp          ebp <- esp
    movl    12(%ebp), %eax      eax <- b;
    movl    8(%ebp), %edx       edx <- a;
    popl    %ebp                ebp <- pop
    addl    %edx, %eax          eax <- eax+edx
    ret                         return
```

Register allocation: where is the return address?

How to get assembly code?

```
/Users/pvh/courses/cs034 [48] -> gcc -O2 -S fact.c
/Users/pvh/courses/cs034 [49] -> ls -l
-rw-r-----  1 pvh fac   516 Mar 12 12:45 fact.s
…
/Users/pvh/courses/cs034 [50] -> gcj -O2 -S Fact.java
-rw-r-----  1 pvh fac   516 Mar 12 12:45 Fact.s
…
```

Native compiler for java

C 2 x86

```
int sum(int a,int b)
{
   int s = 0;
   s = a + b;
   s += a + s;
   return s;
}
```

```
sum:
   pushl   %ebp                    push ebp
   movl    %esp, %ebp              bp <- esp
   movl    8(%ebp), %edx           edx <- a;
   movl    12(%ebp), %ecx          ecx <- b;
   popl    %ebp                    ebp <- pop
   movl    %edx, %eax              eax <- edx
   addl    %ecx, %eax              eax <- eax+ecx
   leal    (%edx,%eax,2), %eax     eax <- edx+2* eax
   ret                             return
```

- Register allocation: where is s?

C 2 x86

```
int sum(int a,int b)
{
   int s = 0;
   s = a + b;
   s += a + s;
}
```

```
sum:
        pushl   %ebp
        movl    %esp, %ebp
        popl    %ebp
        ret
```

- Deadcode elimination

Java 2 x

```
int sum(int a,i
{
  int s = 0;
  s = a + b;
  s += a + s;
  return s;
}
```

```
sum:
    pushl  %ebp                  push ebp
    movl   %esp, %ebp            ebp <- esp
    subl   $4, %esp              esp <- esp - 4;
    movl   16(%ebp), %eax        eax <- b
    addl   12(%ebp), %eax        eax <- eax + a
    movl   %eax, -4(%ebp)        s <- eax
    movl   -4(%ebp), %edx        edx <- s
    movl   -4(%ebp), %eax        eax <- s
    addl   12(%ebp), %eax        eax <- eax + a
    leal   (%eax,%edx), %eax     eax <- eax + edx
    movl   %eax, -4(%ebp)        res <- eax
    movl   -4(%ebp), %eax        eax <- s
    leave                        restore esp & ebp
    ret                          return
```

Arrays

Big differences between C/C++ and Java

C arrays are just pointers

Java arrays

- **objects**
- **bound checking**

C 2 x86

```
int sum(int* a) {
  int s = 0;
  int i;
  for(i = 0; i < 10; ++i)
    s += a[i];
  return s;
}
```

```
sum:
 pushl   %ebp                  push ebp
 xorl    %eax, %eax            eax <- 0
 movl    %esp, %ebp            ebp <- esp
 movl    8(%ebp), %ecx         ecx <- a
 xorl    %edx, %edx            edx <- 0
.L6:
 addl    (%ecx,%edx,4), %eax
         eax <- eax + M[ecx + 4 * edx]
incl     %edx                  edx <- edx+1
cmpl     $9, %edx
jle      .L6                   jmp if edx <= 9
popl     %ebp                  ebp <- pop
ret
```

Arrays in C

Observe

- **no space on the stack**
- **i is register %edx**
- **s is register %eax (return value)**

Java 2 x

```
int sum(int[] a) {
   int s = 0;
   int i = 0;
   for(i = 0; i < 10; +
      s += a[i];
   return s;
}
```

```
      pushl   %ebp                          push ebp
      movl    %esp, %ebp                    ebp <- esp
      subl    $24, %esp                     esp <- esp - 24
      movl    $0, -4(%ebp)                  s <- 0  (stack)
      movl    $0, -8(%ebp)                  i <- 0  (stack)
.L2:
      cmpl    $9, -8(%ebp)                  compare i and 9
      jle     .L4                           jump to L4 if i <= 9
      jmp     .L7                           jump to L7
.L4:
      movl    -4(%ebp), %eax                eax <- s
      movl    %eax, -12(%ebp)               S[-12] <- eax       (s in S[-12])
      movl    12(%ebp), %edx                edx <- a
      movl    %edx, -16(%ebp)               S[-16] <- edx       (a in S[-16])
      movl    -8(%ebp), %ecx                ecx <- i
      movl    %ecx, -20(%ebp)               S[-20] <- ecx       (i in S[-20])
      movl    -16(%ebp), %edx               edx <- a
      movl    8(%edx), %eax                 eax <- a.length
      cmpl    %eax, -20(%ebp)               compare i and eax
      jb      .L5                           jump s[-20]/i < eax
      movl    -20(%ebp), %ecx               ecx <- i
      movl    %ecx, (%esp)                  S[esp] <- ecx
      call    _Jv_ThrowBadArrayIndex        exception
.L5:
      movl    -12(%ebp), %eax               eax <- s
      movl    -20(%ebp), %edx               edx <- i
      movl    -16(%ebp), %ecx               ecx <- a
      addl    12(%ecx,%edx,4), %eax         eax <- eax + M[exc + 4 * edx + 12]
      movl    %eax, -4(%ebp)                s <- eax;
      leal    -8(%ebp), %eax                eax <- &i
      incl    (%eax)                                    i <- i + 1;
      jmp     .L2
.L7:
      movl    -4(%ebp), %eax                eax = s;
      leave
      ret
```

Prologue

Checks

Access

```
pushl   %ebp                push ebp
movl    %esp, %ebp          ebp <- esp
subl    $24, %esp           esp <- esp - 24
movl    $0, -4(%ebp)        s <- 0  (stack)
movl    $0, -8(%ebp)        i <- 0  (stack)
.L2:
cmpl    $9, -8(%ebp)
jle     .L4                  L4 if i <= 9
jmp     .L7                  jump to L7
.L4
        ....
.L7:
movl    -4(%ebp), %eax       eax = s;
leave
 ret
```

stack space!

Java 2 x86

```
.L4:
movl    -4(%ebp), %eax          eax <- s
movl    %eax, -12(%ebp)         S[-12] <- eax       (s in S[-12])
movl    12(%ebp), %edx          edx <- a
movl    %edx, -16(%ebp)         S[-16] <- edx       (a in S[-16])
movl    -8(%ebp), %ecx          ecx <- i
movl    %ecx, -20(%ebp)         S[-20] <- ecx       (i in S[-20])
movl    -16(%ebp), %edx         edx <- a
movl    8(%edx), %eax           eax <- a.length
cmpl    %eax, -20(%ebp)         compare i and eax
jb      .L5                      jump s[-20]/i < eax
movl    -20(%ebp), %ecx         ecx <- i
movl    %ecx, (%esp)            S[esp] <- ecx
call    _Jv_ThrowBadArrayIndex  exception
```

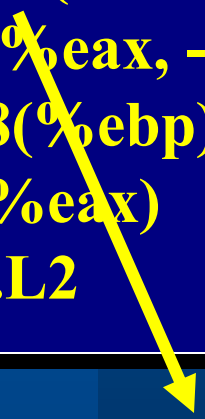
Java 2 x86

```
.L4:
...
movl   -16(%ebp), %edx      edx <- a
movl   8(%edx), %eax        eax <- a.length
...
```

The array in Java is an object!

Java 2 x86

```
.L5:
 movl   -12(%ebp), %eax              eax <- s
 movl   -20(%ebp), %edx              edx <- i
 movl   -16(%ebp), %ecx              ecx <- a
 addl   12(%ecx,%edx,4), %eax        eax<-eax+M[exc+4*edx+12]
 movl   %eax, -4(%ebp)               s <- eax;
 leal   -8(%ebp), %eax          eax <- &i
 incl   (%eax)                       i <- i + 1;
 jmp    .L2
```



What the $#$$% is this "12"?

Arrays

Big differences in efficiency

- **between C/C++ and Java**

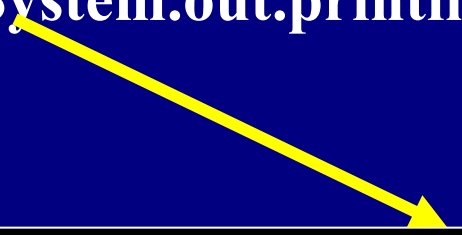
C arrays are just pointers

Java arrays

- **objects**
- **bound checking**

More on Arrays in Java

```
class Foo { int get() { return 3; }}
class Bar extends Foo { int get() { return 6;}}
public class Array {
    public static void main(String[] args) {
        Bar[] a = new Bar[10];
        Foo[] b = a;
        b[3] = new Foo();
        System.out.println(b[3].get());
    }
}
```



Java compiles this program!

More on Arrays in Java

```
movl    %ebx, 4(%esp)
movl    %esi, (%esp)
call    _Jv_CheckArrayStore
cmpb    $0, -9(%ebp)
movl    %ebx, 24(%esi)
je      .L8
```

Every store in an array has a runtime check!

C 2 x86

```
class Stack {
  int a[100];
  int top;
 public:
  Stack() : top(-1) {}
  void push(int i) { a[++top] = i; }
  int pop() { return a[top--]; }
};
```

```
void g(Stack* s) {
  int i = 0;
  while (i < 10)
    s->push(i++);
}
```

How is g compiled?

Why two incl?

```
    pushl   %ebp
    xorl    %ecx, %ecx              ecx <- 0
    movl    %esp, %ebp              ebp <- esp
    pushl   %ebx                    push ebx (save ebx)
    movl    8(%ebp), %ebx           ebx <- s
    movl    400(%ebx), %edx         edx <- top
.L6:
    movl    %ecx, %eax              eax <- ecx/i
    incl    %edx                    edx <- edx + 1
    incl    %ecx                    ecx <- ecx + 1
    movl    %eax, (%ebx,%edx,4)     eax/i -> S[ebx+4*edx/top]
    cmpl    $9, %ecx
    jle     .L6                     jump if ecx/i <= 9
    movl    %edx, 400(%ebx)         top <- edx;
    popl    %ebx                    ebx <- pop (restore ebx)
    popl    %ebp                    ebp <- pop
    ret
```

Compiler Optimizations

Inlining

- compilers remove function calls
- replace them with the body of the function

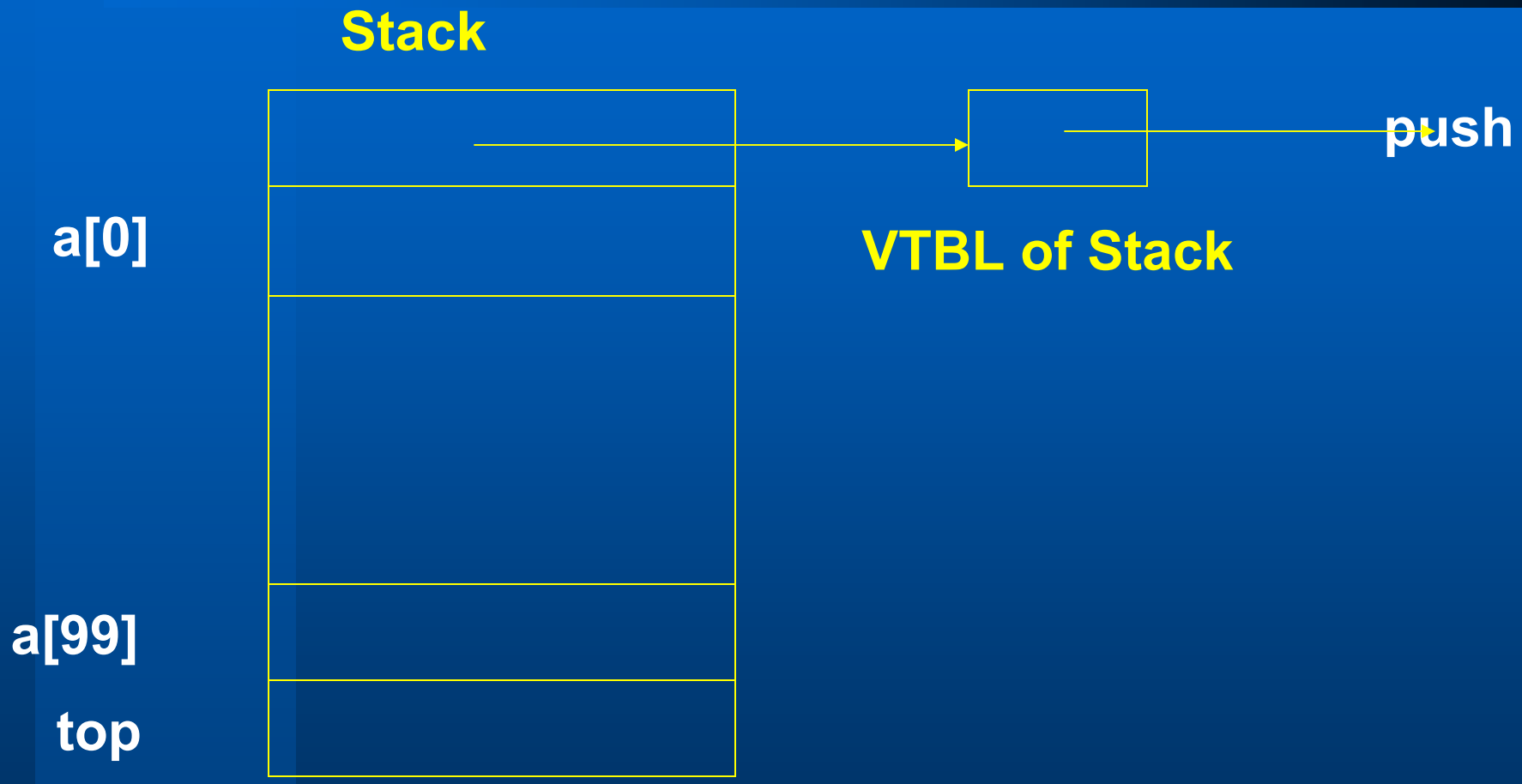
The call to push is removed

- direct manipulation of the stack
- top: offset 400

C 2 x86

```
class Stack {
  int a[100];
  int top;
 public:
  Stack() : top(-1) {}
  virtual void push(int i) { a[++top] = i; }
  int pop() { return a[top--]; }
};
```

VTBL



No inlining!

```
      pushl   %ebp                    p
       movl    %esp, %ebp             sp
       pushl   %esi                   push esi
       pushl   %ebx                   push ebx
       subl    $16, %esp              eps <- eps - 16
       xorl    %ebx, %ebx             ebx <- 0
       movl    8(%ebp), %esi          esi <- &s
.L5:
       movl    (%esi), %eax           eax <- *(&s) [vtbl]
       movl    %ebx, 4(%esp)          S[4] <- ebx/i
       incl    %ebx                   ebx <- ebx + 1;
       movl    %esi, (%esp)           S[0] <- esi/s
       call    *(%eax)                call the routine stored in %eax
       cmpl    $9, %ebx               compare ebx and $9
       jle     .L5                    jump
       addl    $16, %esp              restore the stack
       popl    %ebx                   restore ebx
       popl    %esi                   restore esi
       popl    %ebp                   restore ebp
       ret
```

```
      pushl   %ebp                    push ebp
      movl    %esp, %ebp              ebp <- esp
      pushl   %esi                    push esi
      pushl   %ebx                    push ebx
      subl    $16, %esp               eps <- eps - 16
      xorl    %ebx, %ebx              ebx <- 0
      movl    8(%ebp), %esi           esi <- &s
.L5:
      movl    (%esi), %eax            eax <- *(&s) [vtbl]
      movl    %ebx, 4(%esp)           S[4] <- ebx/i
      incl    %ebx                    ebx <- ebx + 1;
      movl    %esi, (%esp)            S[0] <- esi/s
      call    *(%eax)                 call the routine stored in %eax
      cmpl    $9, %ebx                compare ebx and $9
      jle     .L5                     jump
      addl    $16, %esp               restore the stack
      popl    %ebx                    restore ebx
      popl    %esi                    restore esi
      popl    %ebp                    restore ebp
      ret
```

C 2 x86

why the 4?

```
push:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax           eax <- &s
        movl    12(%ebp), %edx          edx <- i
        movl    404(%eax), %ecx         ecx <- top;
        incl     %ecx                   ecx++;
        movl    %ecx, 404(%eax)         top <- ecx
        movl    %edx, 4(%eax,%ecx,4)    a[4*ecx] <- i
        popl    %ebp
        ret
```

```
void swap(int& a,int& b)
{
   int tmp = a;
   a = b;
   b = tmp;
}
```

C 2 x86

where is tmp?

```
swap
        pushl   %ebp            push ebp
        movl    %esp, %ebp      ebp <- esp
        movl    8(%ebp), %edx   edx <- &a
        pushl   %ebx            push ebx
        movl    12(%ebp), %ecx  ecx <- &b
        movl    (%edx), %ebx    ebx <- a
        movl    (%ecx), %eax    eax <- b
        movl    %eax, (%edx)    *a <- (*edx)/b
        movl    %ebx, (%ecx)    *b <- (*ecx)/a
        popl    %ebx            pop ebx
        popl    %ebp            pop ebp
        ret
```

Class Handles

```
class StackI {
  int a[100]; int top;
 public:
  StackI() : top(-1) {}
  void push(int i) { a[++top] = i; }
  int pop() { return a[top--]; }
};
```

```
class Stack {
  StackI* s;
 public:
  Stack() { st = new StackI(); }
  void push(int i) { s->push(i) }
  int pop() { return s->pop(); }
};
```

```
void g(StackI* s) {
  int i = 0;
  while (i < 10)
    s->push(i++);
}
```

```
void h(Stack s) {
  int i = 0;
  while (i < 10)
    s.push(i++);
}
```

Class Handles are free

The handle is an object

- only one field: a pointer that takes a word

Passing the handle as a parameter

- is similar to passing the pointer

Inlining

- removes the two method calls

The assembly of g and h is the same.

Memory Management

Heap memory in C/C++

- **What you see is what you get**
- **Cost of delete and free**
- **Do not touch the "good stuff"**

Heap memory in Java

- **Mark/copy the good stuff**
- **Collect the garbage**

Memory Management

Stack allocation in C/C++

- **automatic deallocation at no cost**
- **esp <- pop()**
- **not available in Java**

Memory Management

Locality

- typically more difficult with garbage collection

C/C++

- the stack has a lot of locality
- deallocation occurs as early as possible

Java

- garbage collection occurs when space is running tight

The JVM

Java is often compiled in two steps

- the first step generates JVM code (class file)
- the second step may generate native code from the JVM (JIT compiler)

Advantages

- The JVM code is portable across different machines

Note that the source code is also portable!

The JVM

The JVM is a stack machine

- no registers
- all arithmetic operations act on the JVM stack

Main Inconvience

- it is difficult to optimize the code on a register machine
- the structure is somehow lost

The JVM

```
int sum(int a,int b)
{
   return a + b;
}
```

```
Method sum
        iload_1
        iload_2
        iadd
        ireturn
```

The JVM

```
iload_1
        movl    8(%ebp), %edx
        pushl   %edx
iload_2
        movl    12(%ebp), %edx
        pushl   %edx
iadd
        popl    %ecx
        popl    %edx
        addl    %ecx,%edx
        pushl   %edx
ireturn
        …
```

The C code

```
sum:
    pushl   %ebp                    push ebp
    movl    %esp, %ebp              ebp <- esp
    movl    12(%ebp), %eax          eax <- b;
    movl    8(%ebp), %edx           edx <- a;
    popl    %ebp                    ebp <- pop
    addl    %edx, %eax              eax <- eax+edx
    ret                             return
```

To be or not to be efficient

- Arrays are very efficient in C/C++
  - slow in Java: bound checks, type checks, objects
- Use virtual methods only with polymorphism
  - inlining of other methods
- Handles are very efficient (no cost)
- Discovery cost of garbage collection
- Cost of the JVM
- Exceptions are cheap: use them

Questions…