Runtime Environments
• Programs run in the context of a **system**
  − Instructions, registers, memory, I/O ports, etc.
• Compilers must emit code that uses this system
  − Must obey the rules of the hardware and OS
  − Must be interoperable with shared libraries compiled by a different compiler
• Memory conventions:
  − **Stack** (used for subprogram calls)
  − **Heap** (used for dynamic memory allocation)
Subprograms

- **Subprogram** general characteristics
  - Single entry point
  - Caller is suspended while subprogram is executing
  - Control returns to caller when subprogram completes
  - Caller/callee info stored on stack

- **Procedure vs. function vs. method**
  - Functions have return values
  - Methods have an associated object (the receiver)
Subprograms

- **New-ish terms**
  - **Header**: signaling syntax for defining a subprogram
  - **Parameter profile**: number, types, and order of parameters
  - **Signature/protocol**: parameter types and return type(s)
  - **Prototype**: declaration without a full definition
  - **Referencing environment**: variables visible inside a subprogram
  - **Name space / scope**: set of visible names
  - **Aliases**: different names for the same location
  - **Call site**: location of a subprogram invocation
  - **Return address**: destination in caller after call completes
Parameters

- **Formal vs. actual parameters**
  - Formal: parameter inside subprogram definition
  - Actual: parameter at call site

- **Semantic models:** \textit{in, out, in-out}

- **Implementations (key differences are when values are copied and exactly what is being copied)**
  - Pass-by-value (\textit{in, value})
  - Pass-by-result (\textit{out, value})
  - Pass-by-copy (\textit{in-out, value})
  - Pass-by-reference (\textit{in-out, reference})
  - Pass-by-name (\textit{in-out, name})
Parameters

- **Pass-by-value**
  - Pro: simple
  - Con: costs of allocation and copying
  - Often the default

- **Pass-by-reference**
  - Pro: efficient (only copy 32/64 bits)
  - Con: hard to reason about, extra layer of indirection, aliasing issues
  - Often used in object-oriented languages
Subprogram Activation

- Caller and callee must agree on calling conventions
- Standard calling contract:
  - Caller: **precall** sequence
    - Evaluate and store parameters
    - Save return address
    - Transfer control to callee
  - Callee: **prologue** sequence
    - Save & initialize base pointer
    - Allocate space for local variables
  - Callee: **epilogue** sequence
    - De-allocate activation record
    - Transfer control back to caller
  - Caller: **postreturn** sequence
    - Clean up parameters
General Stack Layout

- **Stack pointer (SP)**
  - Top of stack (lowest address)

- **Base pointer (BP)**
  - Start of current frame (i.e., saved BP)
  - $r_{arp}$ in EAC (CS 432)
  - EP in Sebesta (CS 430)

- **Stack frame / activation record per call**
  - Parameters
    - Positive offset from BP
  - Saved return address
  - Saved BP (dynamic link)
  - Local variables
    - Negative offset from BP
    - Allocated by decrementing SP

```c
void foo()
{
    int a, b;
    bar(a);
    return;
}
```
# Calling Conventions

<table>
<thead>
<tr>
<th>Calling Convention</th>
<th>Integral parameters</th>
<th>Base pointer</th>
<th>Caller-saved registers</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdecl (x86)</td>
<td>On stack (RTL)</td>
<td>Always saved</td>
<td>EAX, ECX, EDX</td>
<td>EAX</td>
</tr>
<tr>
<td>x86-64 (x64)</td>
<td>RDI, RSI, RDX, RCX, R8, R9, then on stack (RTL)</td>
<td>Saved only if necessary</td>
<td>RAX, RCX, RDX, R8-R11</td>
<td>RAX</td>
</tr>
<tr>
<td>ILOC</td>
<td>On stack (RTL)</td>
<td>Always saved</td>
<td>All virtual registers</td>
<td>RET</td>
</tr>
</tbody>
</table>
Prologue:

- **push** %ebp ; save old base pointer
- **mov** %esp, %ebp ; save top of stack as base pointer
- **sub** X, %esp ; reserve X bytes for local vars

Within function:

- +OFFSET(%ebp) ; function parameter
- -OFFSET(%ebp) ; local variable

Epilogue:

- <optional: save return value in %eax>
- **mov** %ebp, %esp ; restore old stack pointer
- **pop** %ebp ; restore old base pointer
- **ret** ; pop stack and jump to popped address

Function calling:

- <push parameters> ; precall
- **call** <fname> ; save return address and jump
- <deallocate parameters> ; postreturn

**Much of prologue & epilogue is optional in x86-64**
ILOC Calling Conventions

Prologue:

-push  BP ; save old base pointer
-i2i   SP => BP ; save top of stack as base pointer
-addI  SP, -X => SP ; reserve X bytes for local vars

(Required even if there are no local vars – X may need to be adjusted during register allocation for spilled registers)

Within function:

[BP+OFFSET] ; function parameter
[BP-OFFSET] ; local variable

Epilogue:

-<optional: save return value in RET>
i2i   BP => SP ; restore old stack pointer
-pop   BP ; restore old base pointer
-return ; pop stack and jump to popped address

Function calling:

-<push parameters> ; precall
call <fname> ; save return address and jump
-<dealloc parameters> ; postreturn

Described in detail in section 8 of Decaf reference
def void foo()
{
}

def int main()
{
    foo();
    return 0;
}
def int add(int x, int y)
{
    int sum;
    sum = x + y;
    return sum;
}

def int main()
{
    return add(3, 7);
}
Misc. Topics

• Macros
  – Call-by-name, “executed” at compile time
  – Often provide call-by-name semantics

• Closures
  – A subprogram and its referencing environment
  – Requires a more general structure than the stack

• Just-in-time (JIT) compilation
  – Defer compilation of each function until it is called
  – New chapter in EAC3e!
Heap Management

- Desired properties
  - Space efficiency
  - Exploitation of locality (time and space)
  - Low overhead

- Allocation (malloc/new)
  - First-fit vs. best-fit vs. next-fit
  - Coalescing free space (defragmentation)

- Manual deallocation (free/delete)
  - Dangling pointers
  - Memory leaks
Automatic De-allocation

- Criteria: overhead, pause time, space usage, locality impact
- Basic problem: finding reachable structures
  - Root set: static and stack pointers
  - Recursively follow pointers through heap structures
- Reference counting (incremental)
  - Memory/time overhead to track the number of active references to each structure
  - Catch the transition to unreachable (count becomes zero)
  - Has trouble with cyclic data structures
- Mark and sweep (batch-oriented)
  - Occasionally pause and detect unreachable structures
  - High time overhead and potentially undesirable "pause the world" semantics
  - Partial collection: collect only a subset of memory on each run
  - Generational collection: collect newer objects more often
Object-Oriented Languages

- Classes vs. objects
- Inheritance relationships (subclass/superclass)
  - Single vs. multiple inheritance
- Closed vs. open class structure
- Visibility: public vs. private vs. protected
- Static vs. dynamic dispatch
- Object-records and virtual method tables