Run-time Environments
Status

• We have so far covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
• Next come the back-end phases
  - Code generation
  - Optimization
  - Register allocation
  - Instruction scheduling
• We will examine code generation first . . .
Run-time environments

• Before discussing code generation, we need to understand what we are trying to generate

• There are a number of standard techniques for structuring executable code that are widely used
Outline

• Management of run-time resources

• Correspondence between static (compile-time) and dynamic (run-time) structures

• Storage organization
Run-time Resources

• Execution of a program is initially under the control of the operating system (OS)

• When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of this space
  - The OS jumps to the entry point of the program (i.e., to the beginning of the “main” function)
Memory Layout

Memory

Low Address

Code

Other Space

High Address
Notes

• By tradition, pictures of run-time memory organization have:
  - Low addresses at the top
  - High addresses at the bottom
  - Lines delimiting areas for different kinds of data

• These pictures are simplifications
  - E.g., not all memory need be contiguous
Organization of Code Space

- Usually, code is generated one function at a time. The code area thus is of the form:

```
Code for function 1
Code for function 2
...
Code for function n
```

- Careful layout of code within a function can improve i-cache utilization and give better performance.
- Careful attention in the order in which functions are processed can also improve i-cache utilization.
What is Other Space?

- Holds all data needed for the program’s execution
- Other Space = Data Space

- Compiler is responsible for:
  - Generating code
  - Orchestrating the use of the data area
Code Generation Goals

• Two goals:
  - Correctness
  - Speed

• Most complications in code generation come from trying to be fast as well as correct
Assumptions about Flow of Control

(1) Execution is sequential; at each step, control is at some specific program point and moves from one point to another in a well-defined order.

(2) When a procedure is called, control eventually returns to the point immediately following the place where the call was made.

Do these assumptions always hold?
Language Issues that affect the Compiler

- Can procedures be recursive?
- What happens to the values of the locals on return from a procedure?
- Can a procedure refer to non-local variables?
- How are parameters to a procedure passed?
- Can procedures be passed as parameters?
- Can procedures be returned as results?
- Can storage be allocated dynamically under program control?
- Must storage be deallocated explicitly?
Activations

• An invocation of procedure $P$ is an *activation* of $P$

• The *lifetime* of an activation of $P$ is
  - All the steps to execute $P$
  - Including all the steps in procedures $P$ calls
Lifetimes of Variables

• The *lifetime* of a variable $x$ is the portion of execution in which $x$ is defined

• Note that:
  - Lifetime is a dynamic (run-time) concept
  - Scope is (usually) a static concept
Activation Trees

- Assumption (2) requires that when $P$ calls $Q$, then $Q$ returns before $P$ does

- Lifetimes of procedure activations are thus either disjoint or properly nested

- Activation lifetimes can be depicted as a tree
Example 1

```c
int g() { return 42; }
int f() { return g(); }
main() { g(); f(); }
```
Example 2

g(): int { return 42; }
f(x:int): int {
    if x = 0 then return g();
    else return f(x - 1);
}
main() { f(3); }

What is the activation tree for this example?
Notes

• The activation tree depends on run-time behavior
• The activation tree may be different for every program input

Since activations are properly nested, a (control) stack can track currently active procedures
  - push info about an activation at the procedure entry
  - pop the info when the activation ends; i.e., at the return from the call
Example

g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }

main                      Stack
   main
Example

g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
Example

g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
Example

g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
Example

g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
Revised Memory Layout

Memory

Code

Stack

Low Address

High Address
Activation Records

• The information needed to manage a single procedure activation is called an *activation record* (AR) or a *stack frame*

• If a procedure F calls G, then G’s activation record contains a mix of info about F and G
What is in G's AR when F calls G?

• F is “suspended” until G completes, at which point F resumes. G’s AR contains information needed to resume execution of F.

• G’s AR may also contain:
  - G’s return value (needed by F)
  - Actual parameters to G (supplied by F)
  - Space for G’s local variables
The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- (optional) Pointer to the previous activation record
  - The control link: points to the AR of caller of G
- (optional) Access link for access to non-local names
  - Points to the AR of the function that statically contains G
- Machine status prior to calling G
  - Return address, values of registers & program counter
  - Local variables
- Other temporary values used during evaluation
Example 2, Revisited

```cpp
g(): int { return 42; }
f(x:int): int {
    if x=0 then return g();
    else return f(x - 1);(**)
}
main() { f(3);(*) }
```

AR for f:

<table>
<thead>
<tr>
<th>result</th>
<th>argument</th>
<th>control link</th>
<th>return address</th>
</tr>
</thead>
</table>

Stack After Two Calls to f

main

f

(result)
3
(*)
(result)
2
(**)
Notes

• **main()** has no argument or local variables and returns no result; its AR is uninteresting

• (*) and (**) are return addresses (continuation points) of the invocations of `f()`
  - The return address is where execution resumes after a procedure call finishes

• This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.
The Main Point

- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record (as displacements from \textit{sp})

\textit{Thus, the AR layout and the code generator must be designed together!}
Example 2, continued

The picture shows the state after the call to the 2nd invocation of `f()` returns
Discussion

• The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame.

• There is nothing magical about this run-time organization:
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
Discussion (Cont.)

• Real compilers hold as much of the frame as possible in registers
  - Especially the function result and (some of) the arguments
Storage Allocation Strategies for Activation Records (1)

Static Allocation (Fortran 77)

- Storage for all data objects laid out at compile time
- Can be used only if size of data objects and constraints on their position in memory can be resolved at compile time ⇒ no dynamic structures
- Recursive procedures are restricted, since all activations of a procedure must share the same locations for local names
Storage Allocation Strategies for Activation Records (2)

**Stack Allocation** (Pascal, C)
- Storage organized as a stack
- Activation record pushed when activation begins and popped when it ends
- Cannot be used if the values of local names must be retained when the evaluation ends or if the called invocation outlives the caller

**Heap Allocation** (Lisp, ML)
- Activation records may be allocated and deallocated in any order
- Some form of garbage collection is needed to reclaim free space
Globals

• All references to a global variable point to the same object
  - Can’t store a global in an activation record

• Globals are assigned a fixed address once
  - Variables with fixed address are “statically allocated”

• Depending on the language, there may be other statically allocated values
  - e.g., static variables in C
Memory Layout with Static Data

Memory

Low Address

Code

Global/Static Data

Stack

High Address
Heap Storage

• A value that outlives the procedure that creates it cannot be kept in the AR
  ```
  foo() { new bar; }
  ```
  The `bar` value must survive deallocation of `foo`’s AR

• Languages with dynamically allocated data use a `heap` to store dynamic data
Review of Runtime Organization

- The **code area** contains object code
  - For most languages, fixed size and read only
- The **static area** contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The **stack** contains an AR for each currently active procedure
  - Each AR usually has fixed size, contains locals
- The **heap** contains all other data
  - In C, heap is managed explicitly by `malloc()` and `free()`
  - In Java, heap is managed by `new()` and garbage collection
  - In ML, both allocation and deallocation in the heap is managed implicitly
Notes

• Both the heap and the stack grow

• Must take care so that they don’t grow into each other

• Solution: start heap and stack at opposite ends of memory and let them grow towards each other
Memory Layout with Heap

- **Code** (Low Address)
- **Global/Static Data**
- **Stack**
- **Heap** (High Address)
Data Layout

• Low-level details of computer architecture are important in laying out data for correct code and maximum performance

• Chief among these concerns is alignment of data
Alignment

- Most modern machines are 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
- Data is *word-aligned* if it begins at a word boundary

Most machines have some alignment restrictions
(Or performance penalties for poor alignment)
Alignment (Cont.)

Example: A string:

"Hello"

Takes 5 characters (without the terminating \0)

• To word-align next datum on a 32-bit machine, add 3 “padding” characters to the string

• The padding is not part of the string, it’s just unused memory