Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

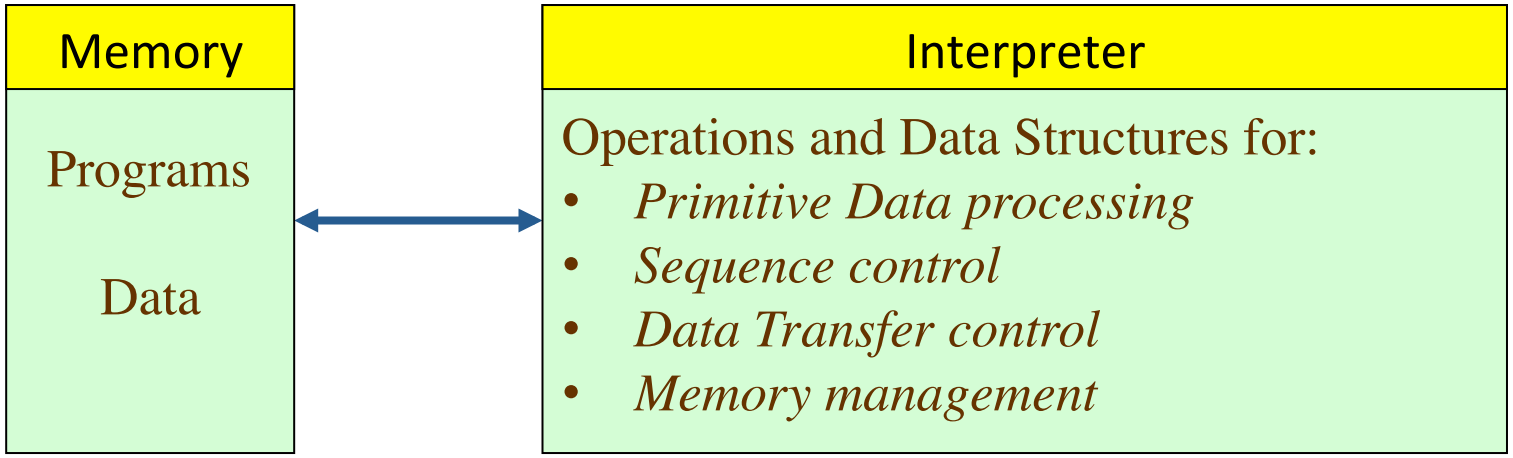
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 18

- Names and bindings
- Storage organization
- Scopes and static scoping

(Recap) Abstract Machine for a Language **L**

- Given a programming language **L**, an **Abstract Machine M_L for L** is *a collection of data structures and algorithms which can perform the storage and execution of programs written in **L***
- Structure of an abstract machine:



Programming Languages and Abstraction

- High Level PL's have been developed to
  - abstract from machine languages (→ portability)
  - make programming easier for software developers
- Each PL generation abstracts more
- Modern languages include sophisticated abstraction mechanisms to be used by programmers
- The study of PL's focuses on the study of abstractions, abstraction mechanisms, and how they can be implemented (efficiently…)

Abstraction mechanisms

Abstraction Mechanisms are independent of the concept to be abstracted

- Naming
  - Associate a name with a possibly complex entity
- Abstraction by Parametrization
  - Abstracts from identity of data by using parameters
    - Procedures with parameter
    - Generic types, …
- Abstraction by Specification
  - Abstracts from implementation details
    - Interfaces
    - Function prototypes, …
- Discussed in depth in
  - [B. Liskov, J. Guttag] Program Development in Java: Abstraction, Specification, and Object-Oriented Design

PL's defining Abstract Machines

The definition of a PL mainly consists of defining

- the Abstract Machine components, and
- the abstraction mechanism **[abs]** to extend them:
  - **Primitive Data processing [DP]**
    - Data types and operations
    - Procedures, …
  - **Sequence control [SC]**
    - Control structures, …
  - **Data Transfer control [DTC]**
    - Parameter passing mechanisms
    - Scoping rules, …
  - **Memory management [MM]**
    - Static / stack / heap allocation mechanisms, …

Programming Language Concepts

- The proposed view allows us to relate the typical concepts of Programming Languages in a unifying framework
  - Names, bindings and scope **[abs]**
  - Values and data types **[DP]**
  - Variables and storage management **[MM]**
  - Control abstraction **[abs, SC]**
  - Data abstraction **[abs, DP]**
  - Generic abstraction **[abs, DP]**
  - Concurrency **[SC]**
- We will discuss some of them in generality, making reference to concrete examples when useful

Run-time environment

- The compiler must implement the abstractions of the source programming languages, including *names*, *scopes*, *procedures*, *parameters*, …
- It creates and manages a *run-time environment* for providing such abstractions during execution
  - Layout and allocation of memory
  - Mechanism for accessing variables
  - Linkage between procedures, parameter passing mechanisms
  - Interface to Operating System, I/O, …

Names and abstraction

- Names are a fundamental abstraction mechanism
- Used by programmers and language designers to refer to variables, constants, operations, types, …
- Their use applies to all dimensions of programming languages:
  - Control abstraction:
    - Control flow constructs (if-then, while, for, return) hide low-level machine ops
    - Subroutines (procedures and functions) allow programmers to focus on manageable subset of program text, hiding implementation details
  - Data abstraction:
    - User-defined data types
    - Object-oriented classes hide data representation details behind a set of operations

Bindings and Binding Time

- A **binding** is an association between a **name** and an **entity**
- An entity that can have an associated name is called **denotable**
- **Binding time** is the time at which a ***decision is made*** to create a name ↔ entity binding (the actual binding can be created later):
  - Language design time
  - Language implementation time
  - Program writing time
  - Compile time
  - Link time
  - Load time
  - Run time

Binding Time Examples

- **Language design time**: the design of specific program constructs (syntax), primitive types, and meaning (semantics)
  - Syntax (names ↔ grammar)
    - `if (a>0) b:=a;` (C syntax style)
    - `if a>0 then b:=a end if` (Ada syntax style)
  - Keywords (names ↔ builtins)
    - `class` (C++ and Java), `endif` or `end if` (Fortran, space insignificant)
  - Reserved words (names ↔ special constructs)
    - `main` (C), `writeln` (Pascal)
  - Meaning of operators (operator ↔ operation)
    - `+` (add), `%` (mod), `**` (power)
  - Built-in primitive types (type name ↔ type)
    - float, short, int, long, string

Binding Time Examples (cont'd)

- **Language implementation time**: fixing implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc. (if not fixed by the language specification)
  - Internal representation of types and literals (type ↔ byte encoding, if not specified by language)
    - 3.1 (IEEE 754) and "foo bar" (\0 terminated or embedded string length)
  - Storage allocation method for variables (static/stack/heap)
- **Program writing time**: the programmer's choice of algorithms and data structures

Binding Time Examples (cont'd)

- **Compile time**: the time of translation of high-level constructs to machine code and choice of memory layout for data objects
  - The specific type of a variable in a declaration (name↔type)
  - Storage allocation mechanism for a global or local variable (name↔allocation mechanism)
- **Link time**: the time at which multiple object codes (machine code files) and libraries are combined into one executable (e.g. external names are bound)
  - Linking calls to static library routines (function↔address)
    - **`printf`** (in libc)
  - Merging and linking multiple object codes into one executable

Binding Time Examples (cont'd)

- **Load time**: when the operating system loads the executable in memory (e.g. physical addresses of static data)
  - Loading executable in memory and adjusting absolute addresses
    - Mostly in older systems that do not have virtual memory
- **Run time**: when a program executes
  - Dynamic linking of libraries (library function↔library code)
    - DLL, dylib
  - Nonstatic allocation of space for variable (variable↔address)
    - Stack and heap
  - Type of a variable (variable↔address) in dynamically typed languages

The Effect of Binding Time

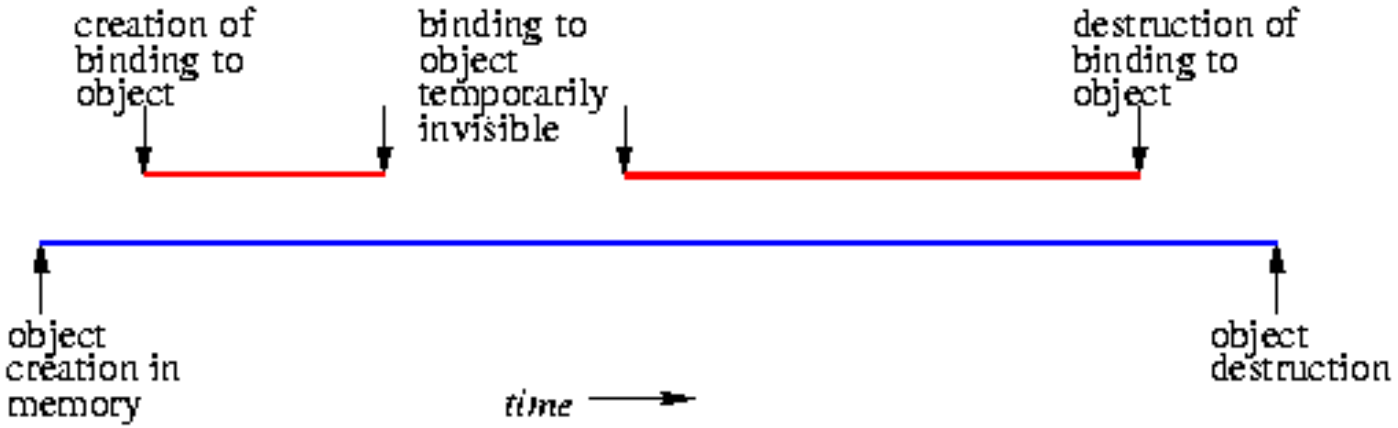
- ***Early binding times*** (before run time, "**static**") are associated with greater efficiency and clarity of program code
  - Compilers make implementation decisions at compile time (avoiding to generate code that makes the decision at run time)
  - Syntax and static semantics checking is performed only once at compile time and does not impose any run-time overheads
- ***Late binding times*** (at run time, "**dynamic**") are associated with greater flexibility (but sometimes make the effects of executing the program obscure)
  - Compilers must generate the code that makes the decision at run time
  - Languages such as Smalltalk-80 with polymorphic types allow variable names to refer to objects of multiple types at run time
  - Method binding in object-oriented languages must be late to support **dynamic binding**

Binding Lifetime versus Object Lifetime

- **Key events**:
  - Object creation ("Object" here is any entity in the programming language)
  - Creation of bindings
  - The object is manipulated via its binding
  - Deactivation and reactivation of (temporarily invisible) bindings
  - Destruction of bindings
  - Destruction of objects
- **Binding lifetime**: time between creation and destruction of binding to object. Examples:
  - a pointer variable is set to the address of an object
  - a formal argument is bound to an actual argument
- **Object lifetime:** time between creation and destruction of an object

Binding Lifetime versus Object Lifetime (cont'd)

- Bindings are visible in scopes, portions of the program that can be determine statically or dinamically
- Bindings are temporarily invisible when code is executed where the binding (name ⇔ object) is out of scope

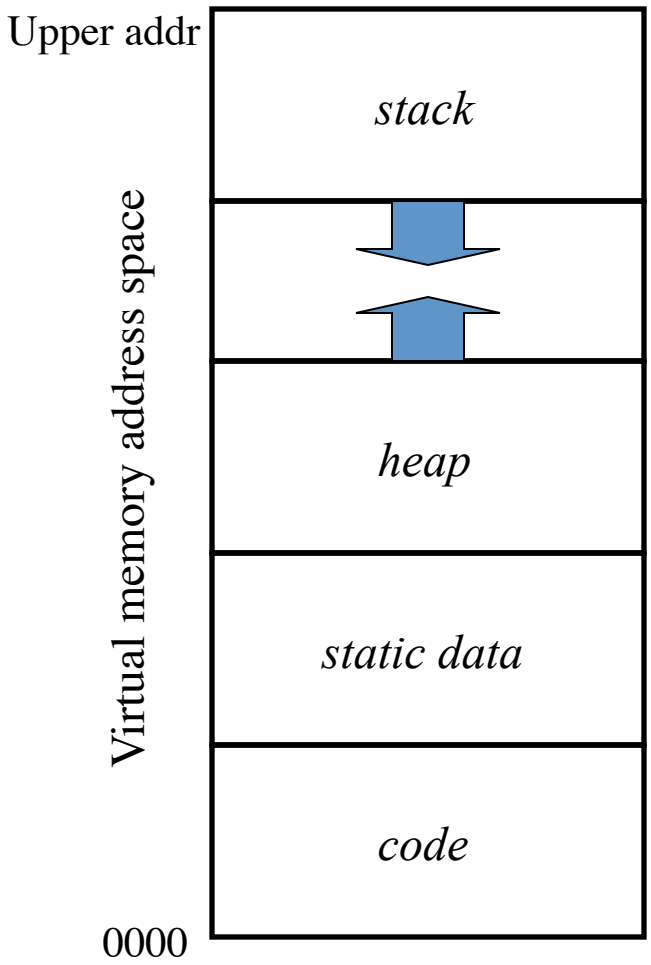


- Since scopes are typically nested, allocation of objects in memory can follow a stack based, LIFO policy

Object Storage

- Objects (program data and code) have to be stored in memory during their lifetime
- **Static objects** have an absolute storage address that is retained throughout the execution of the program
  - Global variables and data
  - Subroutine code and class method code
- **Stack objects** are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns
  - Actual arguments passed by value to a subroutine
  - Local variables of a subroutine
- **Heap objects** may be allocated and deallocated at arbitrary times, but require a storage management algorithm
  - Example: Lisp lists
  - Example: Java class instances are always stored on the heap

Typical Program and Data Layout in Memory

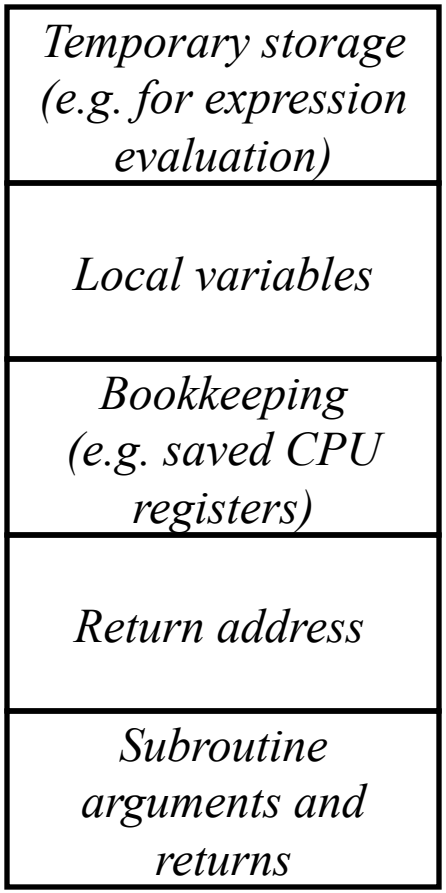


- The compiler assumes that the target program will run in a logical address space
- The operating system will map logical addresses to physical ones
- Program code is at the bottom of the memory region (code section)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward

Static Allocation

- Program code is statically allocated in most implementations of imperative languages
- Statically allocated variables are **history sensitive**
  - Global variables keep state during entire program lifetime
  - Static local variables in C functions keep state across function invocations
  - Static members in Java classes are “shared” by objects and keep state during program lifetime (but they are allocated in the heap…)
- Advantage of statically allocated object is the fast access due to absolute addressing of the object
  - Address determined by the compiler, does not need to be computed at runtime
  - Problem: static allocation of local variables cannot be used for *recursive* subroutines: each new function instantiation needs fresh locals

Static Allocation
A paradigmatic example: Fortran 77



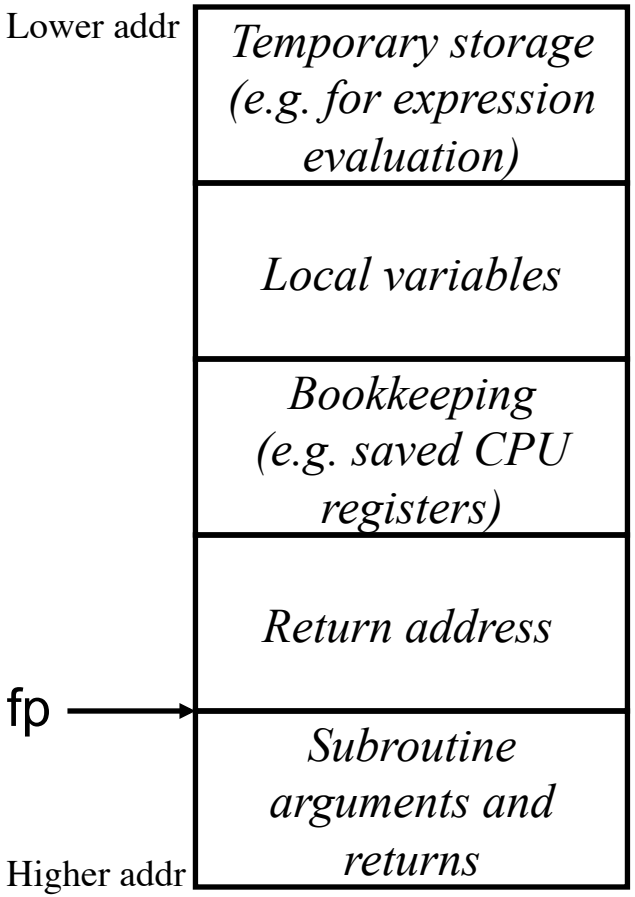
Typical static subroutine frame layout

- Fortran 77 has no recursion
- Global and local variables are statically allocated as decided by the compiler
- Global and local variables are referenced at absolute addresses
- Avoids overhead of creation and destruction of local objects for every subroutine call
- Each subroutine in the program has a ***subroutine frame*** that is statically allocated
- This subroutine frame stores all subroutine-relevant data that is needed to execute
- Inefficient use of memory, even in absence of recursion

Stack Allocation

- Each instance of a subroutine that is active has a *subroutine frame* (or *activation record*) on a run-time stack
  - Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards
- Subroutine frame layouts vary between languages, implementations, and machine platforms
- Typically the stack is accessed through two pointers:
  - The *stack pointer* (sp) points to the next available free space on the stack
  - The *frame pointer* (fp) points to the activation record of the currently active subroutine
- Some languages use only the *stack pointer*, like C
- Stack and frame pointer are usually in registers

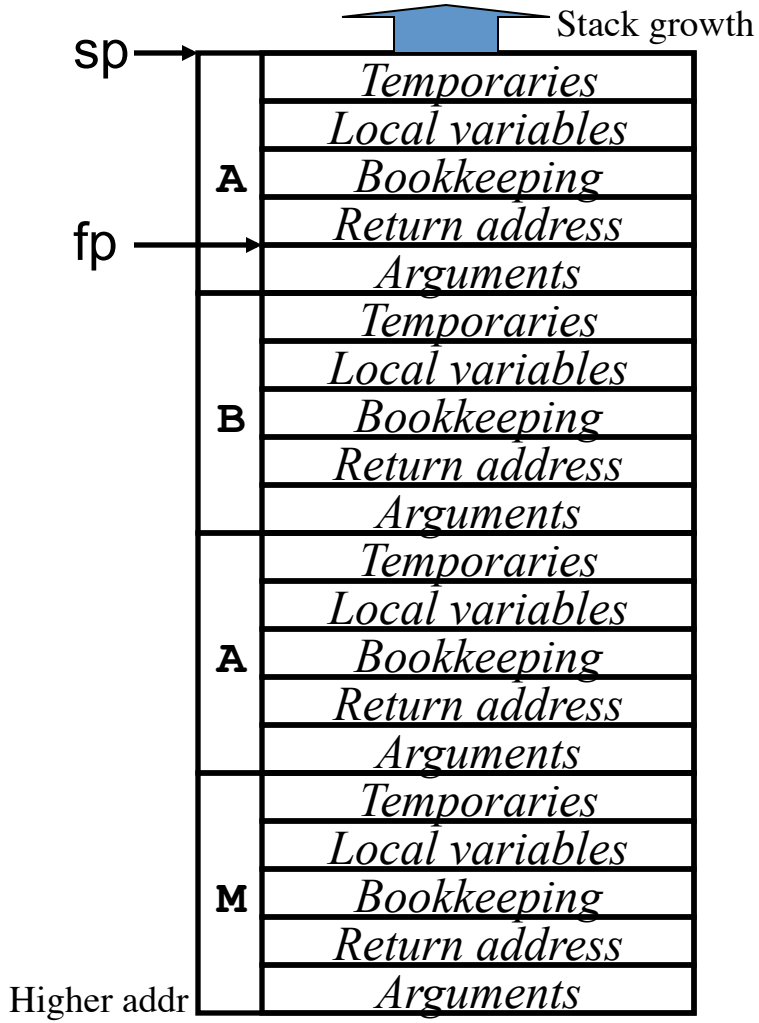
Typical Stack-Allocated Activation Record



Typical subroutine frame layout

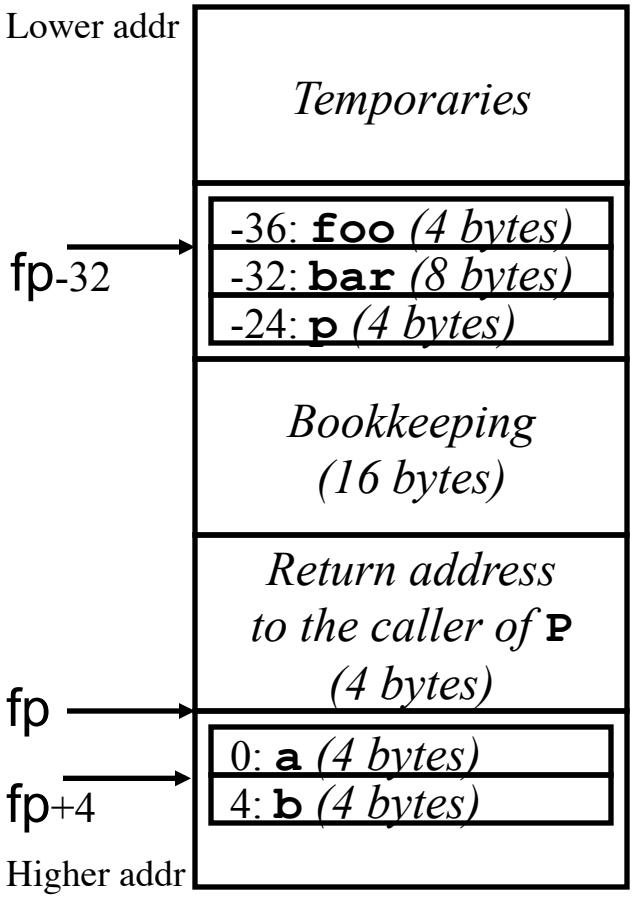
- A *frame pointer* (**fp**) points to the frame of the currently active subroutine at run time
- Subroutine arguments, local variables, and return values are accessed by constant address offsets from the **fp**
- Sometimes arguments and/or the return value are placed in registers

Activation Records on the Stack



- Activation records are pushed and popped onto/from the runtime stack
- The *stack pointer* (sp) points to the next available free space on the stack to push a new activation record onto when a subroutine is called
- The *frame pointer* (fp) points to the activation record of the currently active subroutine, which is always the topmost frame on the stack
- The fp of the previous active frame is saved in the current frame and restored after the call
- In this example:
  M called **A**
  A called **B**
  B called **A**

Example Activation Record



- The size of the types of local variables and arguments determines the **fp** offset in a frame
- Example Pascal procedure:

```
procedure P(a:integer,
            var b:real)
(* a is passed by value
   b is passed by reference,
     = pointer to b's value
*)
var
  foo:integer;(* 4 bytes *)
  bar:real;   (* 8 bytes *)
  p:^integer; (* 4 bytes *)
begin
  ...
end
```

Generating Code for Stack Allocation of Activation Records

```
t1 := a + b
param t1
param c
t2 := call foo,2
…



func foo
…
return x
```

```
int foo(int a,int b){
…
return x}
…
d = foo(a+b,c)
…
```

```
100: ADD #16,SP
108: MOV a,R0
116: ADD b,R0
124: MOV R0,4(SP)
132: MOV c,8(SP)
140: MOV #156,*SP
148: GOTO 500
156: MOV 12(SP),R0
164: SUB #16,SP
172: …

500: …
564: MOV R0,12(SP)
572: GOTO *SP
```

Push frame

Store a+b

Store c

Store return address

Jump to foo

Get return value

Remove frame

Store return value

Return to caller

Note: Language and machine dependent
Here we assume C-like implementation with SP and no FP

Heap Allocation

- **Objects allocated on heap if**
  - **Lifetime**: not bound to a subroutine
    - Eg: **Unlimited extent**
  - **Multiplicity**: more than one
    - Records, class instances, …
  - **Size**: dynamic
    - Dynamic arrays
    - Variable length strings
    - Sets, Lists, Queues, …
- Some languages use heap allocation almost by default
  - Scripting and functional languages
- **Implicit heap allocation**
  - Done automatically when creating data structures
- **Explicit heap allocation**:
  - Statements and/or functions for allocation and deallocation
  - Malloc/free, new/delete

Heap Allocation Algorithms

- The heap is organized as a list of free blocks of memory
- Heap allocation (triggered e.g. by *malloc* in C, or by *new* in Pascal, Java, C++) is performed by searching the heap for available free space
- Deletion of objects leaves free blocks in the heap that can be reused
- *Internal heap fragmentation*: if allocated object is smaller than the free block the extra space is wasted
- *External heap fragmentation*: smaller free blocks cannot always be reused resulting in wasted space

Heap Allocation Algorithms (cont'd)

- Maintain a linked list of free heap blocks
- **First-fit**: select the first block in the list that is large enough
- **Best-fit**: search the entire list for the smallest free block that is large enough to hold the object
- If an object is smaller than the block, the extra space can be added to the list of free blocks
- When a block is freed, adjacent free blocks are merged
- **Buddy system**: use heap pools of standard sized blocks of size 2^k
  - If no free block is available for object of size between $2^{k-1}+1$ and 2^k then find block of size 2^{k+1} and split it in half, adding the halves to the pool of free 2^k blocks, etc.
- **Fibonacci heap**: use heap pools of standard size blocks according to Fibonacci numbers
  - More complex but leads to slower internal fragmentation

Problems with **explicit** heap allocation

- Explicit manual deallocation errors are among the most expensive and hard to detect problems in real-world applications
  - If an object is deallocated too soon, a reference to the object becomes a **dangling reference**
  - If an object is never deallocated, the program **leaks memory**
- Automatic garbage collection removes all objects from the heap that are not accessible, i.e. are not referenced, putting them in the free list
  - Disadvantage is GC overhead, but GC algorithm efficiency has been improved
  - Not always suitable for real-time processing
  - Several algorithms available

Scope of a binding

- The **scope of a binding** is the textual region of a program in which a name-to-object binding is active
- **"Scope"**: textual region of maximal size where bindings are not destroyed
  - Module, class, subroutine, block, record/object
- **Statically scoped language**: the scope of bindings is determined at compile time
  - Used by almost all but a few programming languages
  - More intuitive to user compared to dynamic scoping
- **Dynamically scoped language**: the scope of bindings is determined at run time
  - Used e.g. in Lisp (early versions), APL, Snobol, and Perl (selectively)
- The difference is only relevant for *non-local bindings*: those which are neither global nor created locally in the scope

Static (lexical) scoping

- The bindings between names and objects can be determined by examination of the program text
- **Scope rules** of the language define the scope of bindings
  - Early Basic: all variables are global and visible everywhere
  - Fortran 77:
    - scope of local variables limited to the subroutine (unless "save"-ed, like "static" in C);
    - scope of global variable is the whole program text unless hidden
  - Algol 60, Pascal, Ada, … : allow ***nested subroutine definitions***
  - Java, … : allow ***nested classes***
    - Adopt the **closest nested scope rule**

Closest Nested Scope Rule

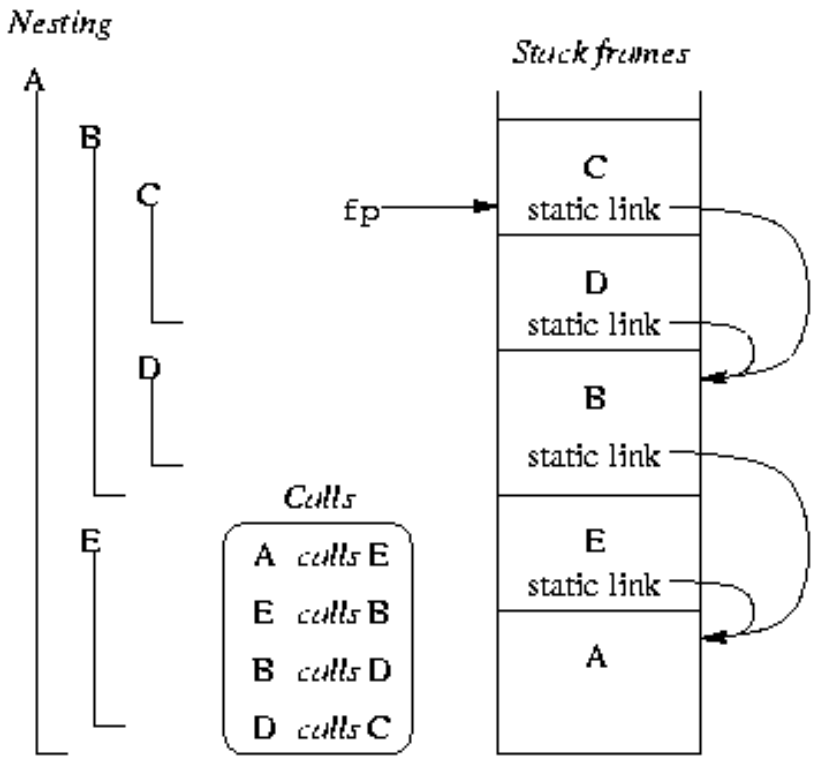
```
procedure P1(A1:T1)
var X:real;
...
  procedure P2(A2:T2);
  ...
    procedure P3(A3:T3);
    ...
    begin
    (* body of P3: P3,A3,P2,A2,X of P1,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P2: P3,P2,A2,X of P1,P1,A1 are visible *)
  end;
  procedure P4(A4:T4);
  ...
    function F1(A5:T5):T6;
    var X:integer;
    ...
    begin
    (* body of F1: X of F1,F1,A5,P4,A4,P2,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P4: F1,P4,A4,P2,X of P1,P1,A1 are visible *)
  end;
...
begin
(* body of P1: X of P1,P1,A1,P2,P4 are visible *)
end
```

- To find the object referenced by a given name:
  - Look for a declaration in the current innermost scope
  - If there is none, look for a declaration in the immediately surrounding scope, etc.
- Built-ins or predefined objects as defined in outermost scope, external to the "global" one
  - I/O routines, mathematical functions

Static Scope Implementation with Static Links

- Access to **global variable**: compiled using constant address
- Access to **local variable**: compiled using *frame pointer* (stored in a register) and *statically known offset*: **`<offset>(FP)`**
- Access to **nonlocal variable?**
- Scope rules are designed so that we can only refer to variables that are *alive*: the variable must have been stored in the activation record of a subroutine
- If a variable is not in the local scope, we are sure there is an activation record for the surrounding scope already allocated on the stack:
  - The current subroutine can only be called when it was visible
  - The current subroutine is visible only when the surrounding scope is active
- Each frame on the stack contains a *static link* pointing to the frame of the **static parent**

Example Static Links



- Subroutines C and D are declared nested in B
  - B is static parent of C and D
- B and E are nested in A
  - A is static parent of B and E
- The **fp** points to the frame at the top of the stack to access locals
- The static link in the frame points to the frame of the static parent

A Typical Calling Sequence

- The caller
  - Saves (in the dedicated area in its activation record) any **registers** whose values will be needed after the call
  - Computes values of **actual parameters** and moves them into the stack or registers
  - Computes the **static link** and passes it as an extra, hidden argument
  - Uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the **return address** on the stack or in a register
- In its prologue, the callee
  - allocates a frame by subtracting an appropriate constant from the **sp**
  - saves the old **fp** into the stack, and assigns it an appropriate new value
  - saves any **registers** that may be overwritten by the current routine (including the **static link** and **return address**, if they were passed in registers)

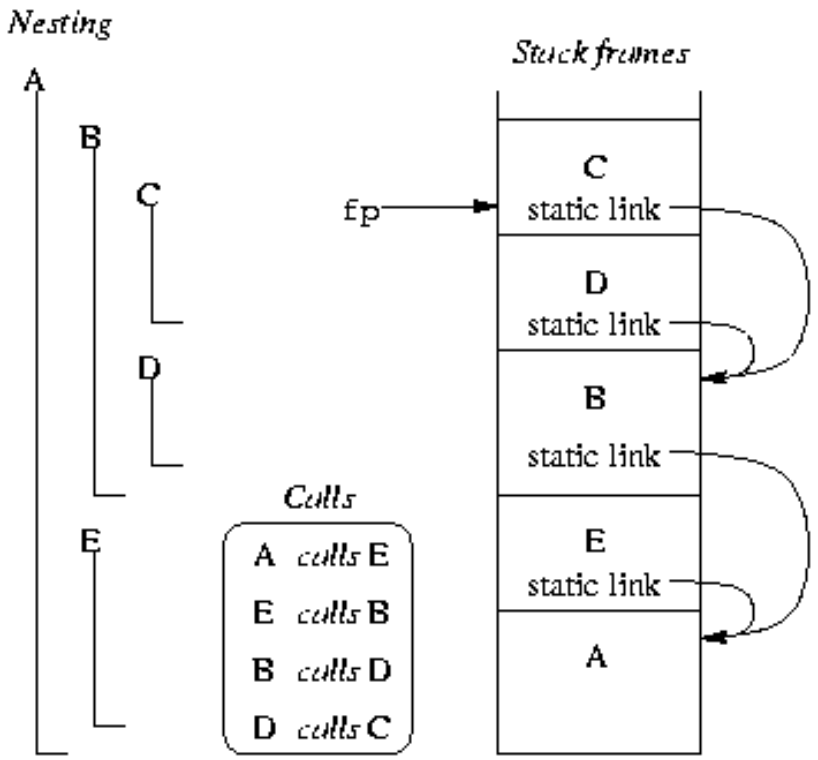
A Typical Calling Sequence (cont'd)

- After the subroutine has completed, the callee
  - Moves the return value (if any) into a register or a reserved location in the stack
  - Restores registers if needed
  - Restores the **fp** and the **sp**
  - Jumps back to the return address
- Finally, the caller
  - Moves the return value to wherever it is needed
  - Restores registers if needed

Static Chains

- How do we access non-local objects?
- The static links form a static chain, which is a linked list of static parent frames
- When a subroutine at nesting level ***j*** has a reference to an object declared in a static parent at the surrounding scope nested at level ***k***, then ***j-k*** static links form a static chain that is traversed to get to the frame containing the object
- The compiler generates code to make these traversals over frames to reach non-local objects

Example Static Chains



- Subroutine A is at nesting level 1 and C at nesting level 3
- When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object

Displays

- Access to an object in a scope *k* levels out requires that the static chain be dereferenced *k* times.
- An object *k* levels out will require *k* + 1 memory accesses to be loaded in a register.
- This number can be reduced to a constant by use of a **display**, a vector where the *k*-th element contains the pointer to the activation record at nesting level *k* that is currently active.
- Faster access to non-local objects, but bookeeping cost larger than that of static chain