Lesson 19

• Names in programming languages
• Binding times
• Scopes
Names, Binding and Scope: Summary

• Abstractions and names
• Binding time
• Object lifetime
• Object storage management
  – Static allocation
  – Stack allocation
  – Heap allocation
• Scope rules
• Static versus dynamic scoping
• Reference environments
• Overloading and polymorphism
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
• Bindings are temporarily invisible when code is executed where the binding (name ↔ object) is out of scope
• **Memory leak**: object never destroyed (binding to object may have been destroyed, rendering access impossible)
• **Dangling reference**: object destroyed before binding is destroyed
• **Garbage collection**: prevents these allocation/deallocation problems
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 an expensive storage management algorithm
  – Example: Lisp lists
  – Example: Java class instances are always stored on the heap
Typical Program and Data Layout in Memory

- 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 data members are “shared” by objects and keep state during program lifetime

• Advantage of statically allocated object is the fast access due to absolute addressing of the object
  – So why not allocate local variables statically?
  – Problem: static allocation of local variables cannot be used for recursive subroutines: each new function instantiation needs fresh locals
Static Allocation in Fortran 77

- 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

<table>
<thead>
<tr>
<th>Temporary storage (e.g. for expression evaluation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Bookkeeping (e.g. saved CPU registers)</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Subroutine arguments and returns</td>
</tr>
</tbody>
</table>

Typical static subroutine frame layout
Stack Allocation

• Each instance of a subroutine that is active has an activation record (or subroutine frame) on the run-time stack
  – Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards
  – Method invocation works the same way, but in addition methods are typically dynamically bound

• Activation record layouts vary between languages, implementations, and machine platforms
Typical Stack-Allocated Activation Record

<table>
<thead>
<tr>
<th>Lower addr</th>
<th>Temporary storage (e.g. for expression evaluation)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local variables</td>
</tr>
<tr>
<td></td>
<td>Bookkeeping (e.g. saved CPU registers)</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Subroutine arguments and returns</td>
</tr>
</tbody>
</table>

- 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.
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

```
<table>
<thead>
<tr>
<th></th>
<th>Temporaries</th>
<th>Local variables</th>
<th>Bookkeeping</th>
<th>Return address</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Stack growth
• The size of the types of local variables and arguments determines the \texttt{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
```
Heap Allocation

• **Implicit heap allocation:**
  – Done automatically
  – Java class instances are placed on the heap
  – Scripting languages and functional languages make extensive use of the heap for storing objects
  – Some procedural languages allow array declarations with run-time dependent array size
  – Resizable character strings

• **Explicit heap allocation:**
  – Statements and/or functions for allocation and deallocation
  – Malloc/free, new/delete
Heap Allocation Algorithms

- Heap allocation is performed by searching the heap for available free space
- For example, suppose we want to allocate a new object E of 20 bytes, where would it fit?

<table>
<thead>
<tr>
<th>Object</th>
<th>Free</th>
<th>Object</th>
<th>Free</th>
<th>Object</th>
<th>Free</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30 bytes</td>
<td>B</td>
<td>10 bytes</td>
<td>C</td>
<td>24 bytes</td>
<td>D</td>
</tr>
</tbody>
</table>

- 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
Unlimited Extent

• An object declared in a local scope has **unlimited extent** if its lifetime continues indefinitely

• A local stack-allocated variable has a lifetime limited to the lifetime of the subroutine
  – In C/C++ functions should never return pointers to local variables

• Unlimited extent requires static or heap allocation
  – Issues with static: limited, no mechanism to allocate more variables
  – Issues with heap: should probably deallocate when no longer referenced (no longer bound)

• Garbage collection
  – Remove object when no longer bound (by any references)
Garbage Collection

• 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
  – Used in Lisp, Scheme, Prolog, Ada, Java, Haskell
  – Disadvantage is GC overhead, but GC algorithm efficiency has been improved
  – Not always suitable for real-time processing
## Comparison of Storage Allocation

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>N/A</td>
<td>local variables and subroutine arguments</td>
<td><em>implicit</em>: local variables of variable size; <em>explicit</em>: new (destruction with garbage collection or explicit with unchecked deallocation)</td>
</tr>
<tr>
<td>C</td>
<td>global variables; static local variables</td>
<td>local variables and subroutine arguments</td>
<td><em>explicit</em> with <code>malloc</code> and <code>free</code></td>
</tr>
<tr>
<td>C++</td>
<td>Same as C, and static class members</td>
<td>Same as C</td>
<td><em>explicit</em> with <code>new</code> and <code>delete</code></td>
</tr>
<tr>
<td>Java</td>
<td>N/A</td>
<td>only local variables of primitive types</td>
<td><em>implicit</em>: all class instances (destruction with garbage collection)</td>
</tr>
<tr>
<td>Fortran77</td>
<td>global variables (in common blocks), local variables, and subroutine arguments (implementation dependent); <code>SAVE</code> forces static allocation</td>
<td>local variables and subroutine arguments (implementation dependent)</td>
<td>N/A</td>
</tr>
<tr>
<td>Pascal</td>
<td>global variables (compiler dependent)</td>
<td>global variables (compiler dependent), local variables, and subroutine arguments</td>
<td>Explicit: <code>new</code> and <code>dispose</code></td>
</tr>
</tbody>
</table>
The scope of a binding is the textual region of a program in which a name-to-object binding is active

- **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 in Lisp (early versions), APL, Snobol, and Perl (selectively)
Effect of Static Scoping

Program execution:

```plaintext
a:integer
main()
a:=2
second()
a:integer
first()
a:=1
write_integer(a)
```

Program prints “1”

• The following pseudo-code program demonstrates the effect of scoping on variable bindings:

```plaintext
a:integer
procedure first
  a:=1
procedure second
  a:integer
  first()
  a:=1
procedure main
  a:=2
  second()
  write_integer(a)
```
Effect of Dynamic Scoping

The following pseudo-code program demonstrates the effect of scoping on variable bindings:

```plaintext
a: integer
main()
  a := 2
  second()
    a: integer
      first()
        a := 1
      write_integer(a)
  write_integer(a)
```

Program execution:

Program prints “2”
Static Scoping

• The bindings between names and objects can be determined by examination of the program text

• *Scope rules* of a program language define the scope of variables and subroutines, which is the region of program text in which a name-to-object binding is usable
  – Early Basic: all variables are global and visible everywhere
  – Fortran 77: the scope of a local variable is limited to a subroutine; the scope of a global variable is the whole program text unless it is hidden by a local variable declaration with the same variable name
  – Algol 60, Pascal, and Ada: these languages allow nested subroutines definitions and adopt the *closest nested scope rule* with slight variations in implementation
Closest Nested Scope Rule

- 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.
Static Scope Implementation with Static Links

• 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 somewhere below 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
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 forms 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
A Typical Calling Sequence

• The caller
  – Saves any registers whose values will be needed after the call
  – Computes values of arguments 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 epilogue
  – 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
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
Out of Scope

• Non-local objects can be hidden by local name-to-object bindings and the scope is said to have a hole in which the non-local binding is temporarily inactive but not destroyed
• Some languages, like Ada, C++ and Java, use qualifiers or scope resolution operators to access non-local objects that are hidden
  – P1.X in Ada to access variable X of P1
  – ::X to access global variable X in C++
  – this.x or super.x in Java
Out of Scope Example

• P2 is nested in P1
• P1 has a local variable X
• P2 has a local variable X that hides X in P1
• When P2 is called, no extra code is executed to inactivate the binding of X to P1

```plaintext
procedure P1;
var X: real;
    procedure P2;
    var X: integer
    begin
        ... (* X of P1 is hidden *)
    end;
begin
    ... 
end
```
Dynamic Scope

• Scope rule: the “current” binding for a given name is the one encountered most recently **during execution**
• Typically adopted in (early) functional languages that are interpreted
• Perl v5 allows you to choose scope method for each variable separately
• With dynamic scope:
  – Name-to-object bindings *cannot* be determined by a compiler in general
  – Easy for interpreter to look up name-to-object binding in a stack of declarations
• Generally considered to be “a bad programming language feature”
  – Hard to keep track of active bindings when reading a program text
  – Most languages are now compiled, or a compiler/interpreter mix
• Sometimes useful:
  – Unix environment variables have dynamic scope
Dynamic Scoping Problems

• In this example, function `scaled_score` probably does not do what the programmer intended: with dynamic scoping, `max_score` in `scaled_score` is bound to `foo`'s local variable `max_score` after `foo` calls `scaled_score`, which was the most recent binding during execution:

```plaintext
max_score: integer
function scaled_score(raw_score: integer): real
  return raw_score/max_score*100
...
procedure foo
  max_score: real := 0
  ...
  foreach student in class
    student.percent := scaled_score(student.points)
    if student.percent > max_score
      max_score := student.percent
```
Dynamic Scope Implementation with Bindings Stacks

• Each time a subroutine is called, its local variables are pushed on a stack with their name-to-object binding
• When a reference to a variable is made, the stack is searched top-down for the variable's name-to-object binding
• After the subroutine returns, the bindings of the local variables are popped
• Different implementations of a binding stack are used in programming languages with dynamic scope, each with advantages and disadvantages