Programming Language & Compiler

Names, Scopes, and Bindings

Hwansoo Han

Name, Scope, and Binding

- Name
  - A mnemonic character string to represent something else
  - Names in most languages are identifiers
  - Symbols (like '+') can also be names
    - ‘+’ represents an add operation
- Binding
  - An association between a name and the thing it names
- Scope of a binding
  - The part of the program in which the binding is active

Named vs. Unnamed Data

- Programming languages have ability to name data
  - Refer to data using symbolic identifiers rather than addresses

- Not all data is named!
  - Dynamic storage in C is referenced by pointers, not by names

Binding Time

- Binding time
  - Time at which a binding between two things is made
  - All implementation decisions in PL

- Representative two bindings
  - Static binding - binding is made before run time
  - Dynamic binding - binding is made during run time

Binding – efficiency vs. flexibility

- Early vs. later
  - Early binding times – generally lead to greater efficiency
  - Later binding times – generally lead to greater flexibility

- Compiled vs. interpreted
  - Compiled languages – tend to have early binding times
  - Interpreted languages – tend to have later binding times

Object Lifetime (1)

- Key events for objects
  - Creation of objects
  - Creation of bindings
  - References to variables (which use bindings)
  - (Temporary) deactivation of bindings
  - Reactivation of bindings
  - Destruction of bindings
  - Destruction of objects

- Lifetime of an object
  - Period between creation and destruction of the object

Object Lifetime (2)

- Lifetime of a binding
  - Period from creation to destruction of a binding
  - If an object outlives binding, it becomes a garbage
  - If binding outlives object, it becomes a dangling reference
- Scope of a binding
  - Textual region of a program where binding is *active*
- Object lifetime generally corresponds to storage allocation mechanisms
  - Static object, Stack object, Heap object

Storage Management: Static

- Static allocation
  - Live the same lifetime as the program
- Global segments for
  - Code
  - Global variables
  - Static variables
  - Explicit constants (including strings, sets, *etc*.)
  - Scalars (constant numbers)
    - Small scalars may be stored in the immediate fields of instructions
    - E.g. ADD r1, r2, 4

Storage Management: Stack-Based

- Stack-Based Allocation
  - Local variables for functions

- Central stack for
  - Local variables
  - Parameters
  - Temporaries

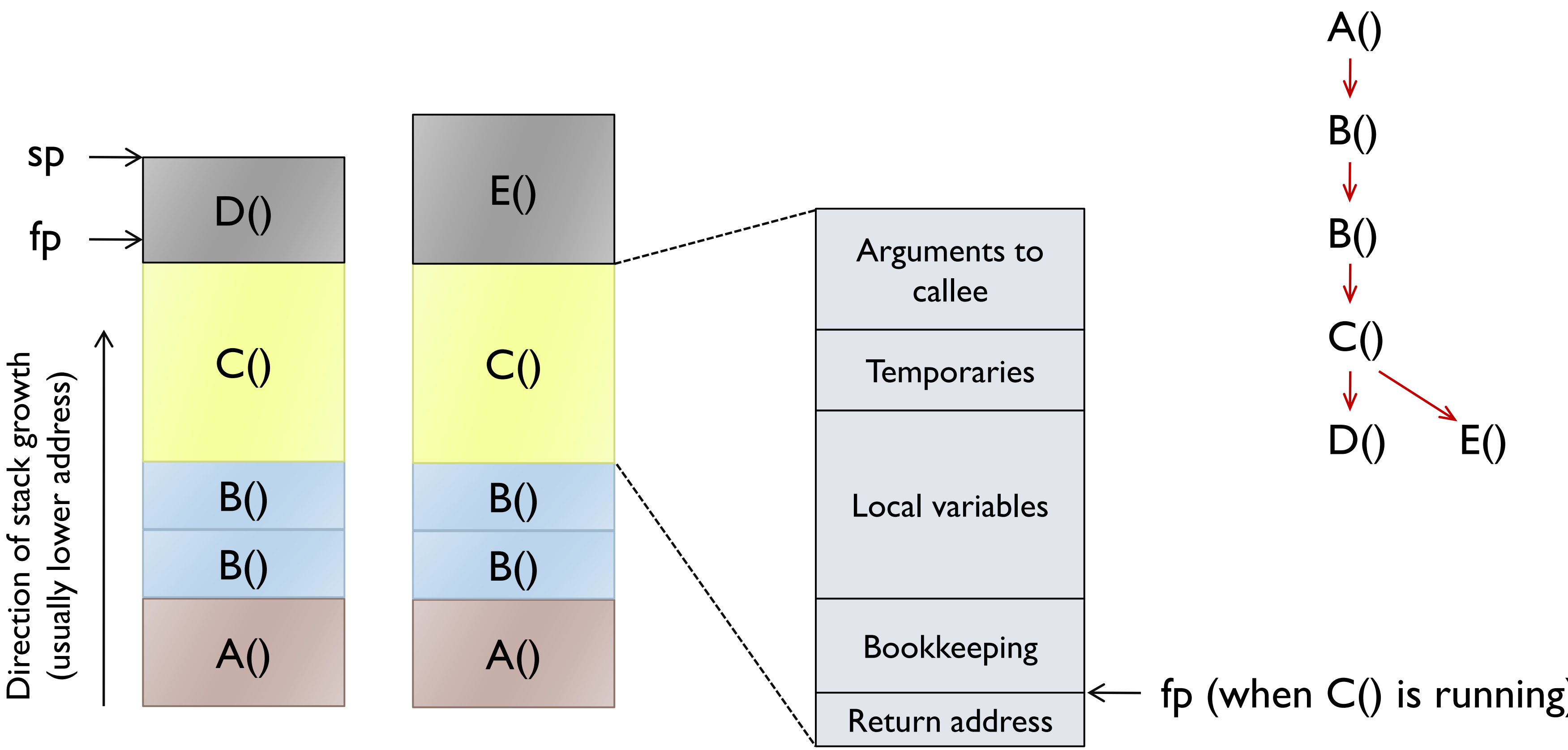
- Why a stack?
  - Allocate space for recursive routines
  - Reuse space

Stack Frame

- Contents of a stack frame (activation record)
  - Arguments and return values
  - Local variables
  - Temporaries
  - Bookkeeping data (saved registers, static link, etc.)

- Reference mechanism
  - Fixed locations within a stack frame
  - Locations are assigned at compile time
  - Access with *displacement addressing* mode (base-offset)
    - *fixed offsets* from the stack pointer (sp), or frame pointer (fp)
    - Can generate code to access data: mov r3, [fp, 10]

Stack Frame (cont'd)



Stack Maintenance

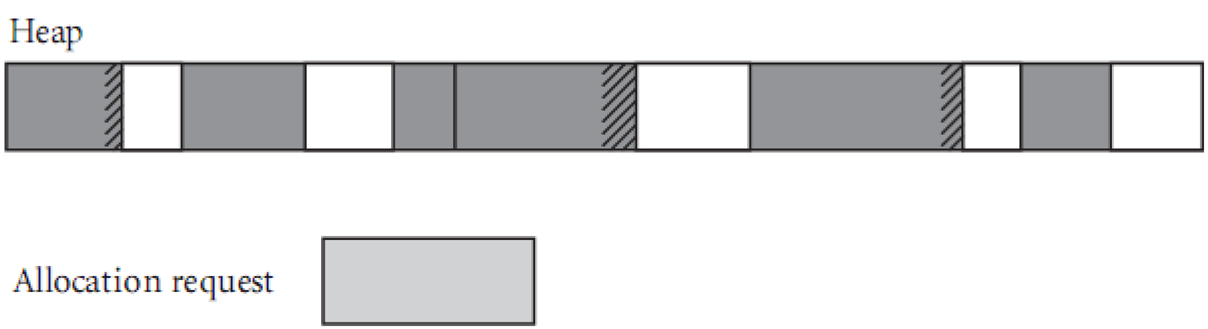
- Maintenance of stack is responsible for
  - *Calling sequence* at call site (caller)
  - Subroutine (callee) *prolog* and *epilog*

- Save *space*
  - Putting as much stuff in prolog & epilog of callee as possible
- Save *time*
  - Share responsibility in the caller and callee
    (e.g., caller-saved registers, callee-saved registers)
  - Passing data directly via registers from both caller and callee
    (e.g., passing arguments/return value in registers)

Storage Management: Heap-Based

- Heap is used for dynamic allocation
  - In-use blocks
  - Free blocks

- Fragmentation
  - Internal fragmentation (cross-hatched space) 
  - External fragmentation
    - Due to discontinuous free blocks, a request block cannot be allocated even if the total free blocks are more than the size of requested block



Garbage Collection

- Objects for heap-based allocation
  - Dynamic allocation is explicitly specified
  - Explicit deallocation (freeing object) may be omitted

- Garbage collection
  - Implicit deallocation of objects
  - Identify garbage (unreachable objects) and reclaim space

- Garbage
  - Objects no longer used – hard to determine at run-time
  - Unreachable objects – easier to determine
    - Guaranteed no use later (there are no ways to reference them)

Scope Rules

- A *scope* is textual region where bindings are active
  - A program section of maximal size
  - Bindings become active at the entry of the scope

  - No bindings change in the middle, or
  - No re-declarations are permitted in the middle at least

Scope Rules in Subroutines

- A subroutine opens a new scope on its entry
  - Create bindings for new local variables,
  - Deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)

- On subroutine exit
  - Destroy bindings for local variables
  - Reactivate bindings for global variables that were deactivated

- Term “elaboration” first used in Algol 68 and Ada
  - Process of creating bindings when entering a scope
  - Allocate space at stack, assign initial values, ...

Static Scoping

- Static scoping (= lexical scoping)
- A scope is defined in terms of the physical (lexical) structure of the program
  - Scopes can be determined at compile time
  - All bindings for IDs can be resolved by examining program
  - Typically, most recent, active binding made at compile time
    - Current binding is the declaration of most closely surrounding block
- Most compiled languages employ static scope rules
  - C/C++, Java, …

Static Scoping – nested blocks

- A classical example of static scope rules
  - The most closely nested rule
  - Used in block structured languages (Algol 60 and Pascal)

- Most closely nested rule
  - An identifier is known in local scope (where it is declared)
  - Also known in each enclosing scope from the closest, unless re-declared in an enclosed scope – "*hidden*"

- Resolving a reference to an identifier
  - Examine the local scope and statically enclosing scopes until a binding is found

Static Scoping - modules

- Object-oriented languages
  - More sophisticated, but static scope rules among classes

- Binding is not destroyed (– different from subroutine)
  - Modules in OOL (Modula, Ada, etc.) give you closed scopes without the limited lifetime
  - Bindings to variables declared in a module are inactive outside the module, not destroyed

  - Similar effect can be achieved in many languages
    - *static* (C term) variables
    - *own* (Algol term)

Static Links

- Access to non-local variables through Static Links
  - Each frame contains a static link to point to the *parent* frame
  - Parent frame means the most recent invocation of the lexically surrounding subroutine

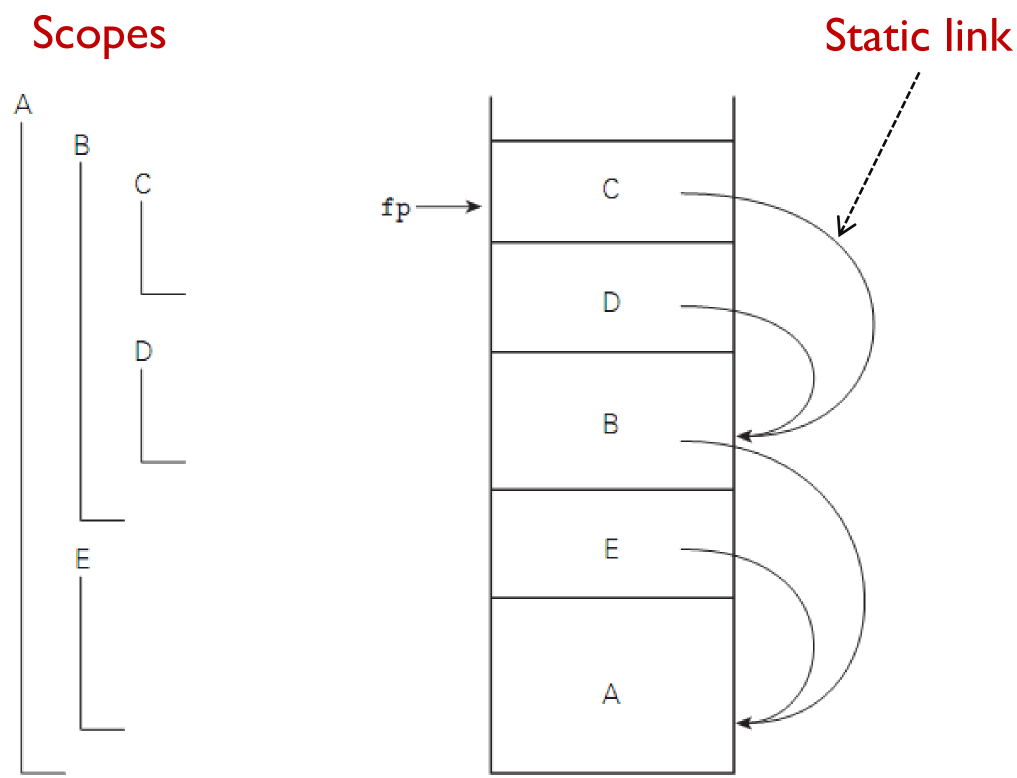
- You access a variable in a scope k levels outside
  - by following k static links and then using the known offset within that frame

Static Chains

```
A() {
      B() {
            C() { … }
            D() { call C(); }

            call D();
      }
      E() {
            call B();
      }

      call E();
}
```



Dynamic Scoping

- With dynamic scope rules, bindings depend on the current state of program execution
  - Cannot always be resolved by examining the program, because they vary depending on calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time

Dynamic Scoping

- Dynamic scoping often used in interpreted languages
  - APL, Snobol, Tex, early dialects of LIPS, Perl

- No type checking at compile time
  - Type determination is not always possible at compile time, when dynamic scope rules are in effect

Accessing Variables in Dynamic Scope

- Two methods
  - Stack
  - Central table

- Stack (*association list*) of all active variables
  - To find a variable, hunt down from top of stack
  - Equivalent to searching the activation records on the dynamic chain
  - Slow accesses but fast calls

Accessing Variables in Dynamic Scope

- Central table with one slot for every variable name
  - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
  - Otherwise, you'll need a hash function to do lookup
  - Every subroutine changes the table entries for its locals at entry/exit
  - Slow calls but fast accesses

Example: Static vs. Dynamic

```
n : integer;          <——————— global

procedure main {

  procedure first {
    n := 1;
  }
  procedure second {
    n : integer;      <——————— local
    first();
  }

  n := 2;
  second();
  write(n);
}
```

Output

Static scoping: 1
Dynamic scoping : 2

Example: Static vs. Dynamic (cont'd)

- How dynamic scoping works for the prev. example
  - Create a binding for global `n` when we enter *main()*
  - Create another binding for local `n` when we enter *second()*
    - This is the most recent, active binding when *first()* is executed
  - In *first()*, modify `n` local to *second()*, not global `n`
  - In *main()*, *write()* uses global `n`,
    - `n` local to *second()* is no longer active in *main()*

Aliases

- Aliasing
  - Same address but multiple names
  - *Variant Record* in Pascal and *Union* in C
  - *Common* and *Equivalence* in FORTRAN
  - Parameter passing by reference to a subroutine

- What are aliases good for?
  - Space saving
  - Multiple representations
  - Pointer-based data structures

Overloading

- Overloading
  - The same name performs different things
    - functions, operators, enumeration constants, etc.

- Overloading happens in almost all languages
  - "Integer +" vs. "real +"
  - Enumeration constants in Ada

```
type autumn is (sep, oct, nov);
type base is (dec, bin, oct, hex);
mo : autumn;
pb :  base;

mo : = nov;
pb := oct;
print(oct);          -- error!
                      -- cannot decide type
```

Overloaded Functions

- Two different things with the same name (in C++)

```
int norm (int a){return a>0 ? a : -a;}
complex norm (complex c ) { sqrt(c.a*c.a + c.b*c.b); }

int i;
complex c;

norm(i);    // integer norm function
norm(c);    // complex norm function
```

Polymorphic Functions

- One thing that works in more than one way
  - Polymorphism ≠ Overloading – they are slightly different
- Parametric polymorphism
  - Code takes types as parameters explicitly or implicitly
  - Generic in Java (or templates in C++) explicitly takes types
  - Lisp, ML, Scheme, Haskell implicitly take types
- Subtype polymorphism
  - Code takes subtypes as well as original type in OOL
  - Inheritance in OOL provides this with *virtual methods*
  - Involves *dynamic binding* of *overriding* function
    - Overriding – a name in base class is redefined in a sub class,
      with the exactly same number/types of parameters

Parametric Polymorphism

- Explicit parametric polymorphism

```
// Java generic with interface java.lang.Comparable<T>
public static <T extends Comparable<T>> T max(T a, T b) {
    if (a.compareTo(b) >= 0) return a;
     else return b;
}

max(1, 5);          // T is Integer (autoboxing int)
max(1.4, 5.6);      // T is Double  (autoboxing double)
```

- Implicit parametric polymorphism
  - Interpreted languages determine operators at run time

```
(define min (lamda (a b) (if (< a b) a b )))  // Scheme

min a b = if a < b then a else b   // Haskell
```

Generic Functions

- A syntactic template that can be instantiated in more than one way *at compile time*
  - Via macro processors in C/C++
  - Built-in in C++ and Ada

```
// C++ template
template<class X> X max(X a, X b) {
  return a>b ? a : b;
}

void g(int a, int b, char c, char d) {
  int m1 = max(a,b);
  char m2 = max(c,d);
}
```

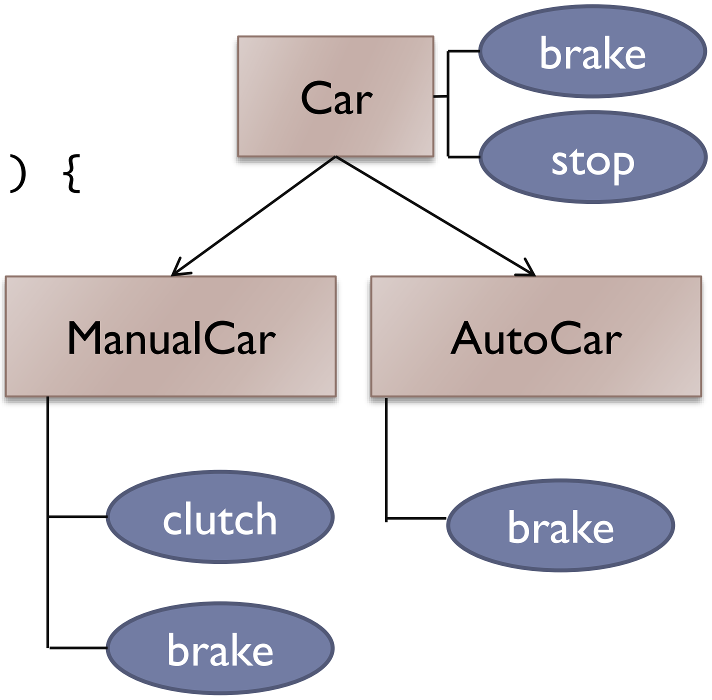
Subtype Polymorphism

```
// Java subtype with virtual method
public class Car {
  public void brake() {}
  public void stop() { brake(); }

  public static void main(String args[]) {
    ManualCar m = new ManualCar();
    AutoCar a = new AutoCar();
    m.stop();
    a.stop();
  }
}

class ManualCar extends Car {
  public void clutch() { … }
  public void brake() { clutch(); … }
}

class AutoCar extends Car {
  public void brake() { … }
}
```



Coercion

- Coercion allows implicit type conversion
  - Compiler automatically converts a value of one type into a value of another type, when the context requires it
  - Could cause performance problem

```
double min(double x, double y) { … }

double  f, g, h;
int     i, j, k;

f = min(g, h);
i = min (j, k);
```

Type conversion operations are inserted for parameters and return value.

Language Features

- Language features can be surprisingly subtle

- A language that is easy to compile often leads to
  - A language that is easy to understand
  - More good compilers on more machines (compare Pascal and Ada!)
  - Better (faster) code
  - Fewer compiler bugs
  - Smaller, cheaper, faster compilers
  - Better diagnostics

Summary

- Binding times: Static-binding vs. Dynamic-binding
- Object lifetime and storage management
  - Static, stack-based, heap-based management
  - Storage determines lifetime of objects

- Scope rules: Static scope vs. dynamic scope
- Meaning of names within a scope changes
  - Aliases
  - Overloading
  - Polymorphism