The Effect of Binding Time

- *Early binding times* (before run time) are associated with greater efficiency
    - Compilers try to fix decisions that can be taken at compile time to avoid to generate code that makes a 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) are associated with greater flexibility
    - Interpreters allow programs to be extended 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*

Run time:

Run time is a very broad term that covers the entire span from the beginning to the end of the execution.

- Program start up time
- Module entry time
- Elaboration time (point at which declaration is first seen)
- Procedure entry time
- Block entry time
- Statement execution time

Exercise 1: Identify the binding time for the following programming events:

- **The asterisk symbol (*) is bound to the multiplication operation.**
- **A data type like *int* is bound to its range.**
- **A variable is bound to its type in Java.**
- **A variable bounded to a storage cell.**
- **A call to a library subprogram is bound to the subprogram code.**

Exercise 2: Consider the following line of code.

```
count = count + 5;
```

Identify all the possible bindings and their associated time.

Static and Dynamic Scoping

Scope rules define the visibility rules for names in a programming language. What if you have references to a variable named `k` in different parts of the program? Do these refer to the same variable or to different ones?

Most languages, including Algol, Ada, C, Pascal, Scheme, and Haskell, are *statically scoped*. A *block* defines a new scope. Variables can be declared in that scope, and aren't visible from the outside. However, variables outside the scope -- in enclosing scopes -- are visible unless they are overridden. In Algol, Pascal, Haskell, and Scheme (but not C or Ada) these scope rules also apply to the names of functions and procedures.

Static scoping is also sometimes called lexical scoping.

Scope in Lambda Calculus

The scope of a variable is where that variable can be mentioned and used. In static scoping, the places where a variable can be used are determined by the lexical structure of the program. An alternative to static scoping is dynamic scoping, in which a variable is bound to the most recent (in time) value assigned to that variable.

The difference becomes apparent when a function is applied. In static scoping, any free variables in the function body are evaluated in the context of the defining occurrence of the function; whereas in dynamic scoping, any free variables in the function body are evaluated in the context of the function call. The difference is illustrated by the following program:

$$\begin{array}{l} \mathsf{let}\ d = 2\ \mathsf{in} \\ \mathsf{let}\ f = \lambda x.\, x + d\ \mathsf{in} \\ \mathsf{let}\ d = 1\ \mathsf{in} \\ f\, 2 \end{array}$$

Static Scoping:

1. The outer d is bound to 2.

2. The f is bound to λx. x + d. Since d is statically bound, this is will always be equivalent to λx. x + 2

(the value of d cannot change, since there is no variable assignment in this language).

3. The inner d is bound to 1.

4. When evaluating the expression f 2, free variables in the body of f are evaluated using the environment in which f was defined. In that environment, d was bound to 2. We get 2 + 2 = 4.

Dynamic Scoping

1. The outer d is bound to 2.

2. The f is bound to λx. x+d. The occurrence of d in the body of f is not locked to the outer declaration of d.

3. The inner d is bound to 1.

4. When evaluating the expression f 2, free variables in the body of f are evaluated using the environment of the call, in which d is 1. We get 2 + 1 = 3.

Simple Static Scoping Example

```
    begin
    integer m, n;

    procedure hardy;
        begin
        print("in hardy -- n = ", n);
        end;

    procedure laurel(n: integer);
        begin
        print("in laurel -- m = ", m);
        print("in laurel -- n = ", n);
        hardy;
        end;

    m := 50;
    n := 100;
    print("in main program -- n = ", n);
    laurel(1);
    hardy;
    end;
```

The output is:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 100    /* note that here hardy is called from laurel */
in hardy -- n = 100    /* here hardy is called from the main program */
```

Blocks can be nested an arbitrary number of levels deep.

Dynamic Scoping

Dynamic scoping was used in early dialects of Lisp, and some older interpreted languages such as SNOBOL and APL. It is available as an option in Common Lisp. Using this scoping rule, we first look for a local definition of a variable. If it isn't found, we look up the calling stack for a definition. (See Lisp book.) If dynamic scoping were used, the output would be:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 1    ;; NOTE DIFFERENCE -- here hardy is called from laurel
in hardy -- n = 100  ;; here hardy is called from the main program
```

Static Scoping with Nested Procedures

```
begin
integer m, n;

procedure laurel(n: integer);
    begin

    procedure hardy;
        begin
        print("in hardy -- n = ", n);
        end;

    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
    end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);
/* can't call hardy from the main program any more */
end;
```

The output is:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 1
```

Lifetime and Storage Management

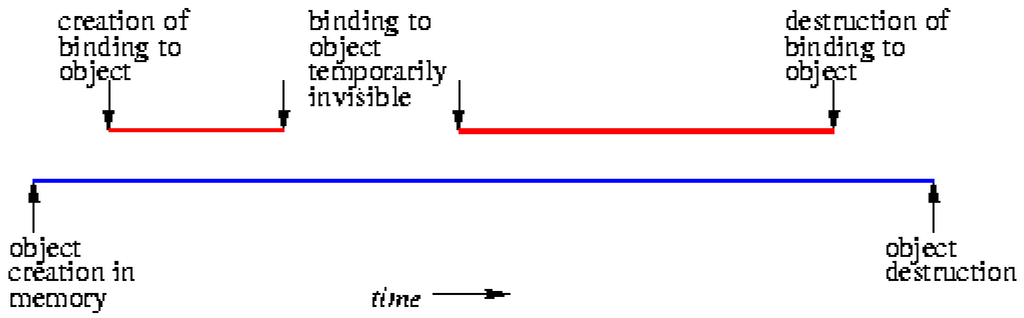
Bindings key Events

1. Creation of objects
2. Creation of bindings
3. Reference to variables (which use bindings)
4. (temporary) deactivation of bindings
5. Reactivation of bindings
6. Destruction of bindings
7. Destruction of objects

Object Lifetime Example

- Example C++ fragment:

```
{ SomeClass& myobject = *new SomeClass;
  ...
  { OtherClass& myobject = *new
OtherClass;
    ... myobject // is bound to other
object
    ...
  }
  ... myobject // is visible again
  ...
  delete myobject;
}
```

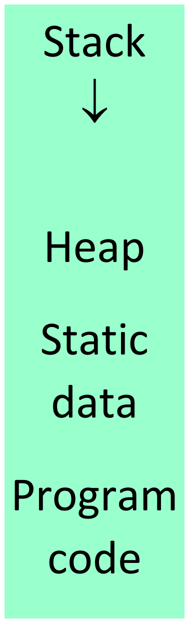

creation of binding to object
binding to object temporarily invisible
destruction of binding to object
object creation in memory
time
object destruction

Object Storage Management

- An object has to be stored in memory during its lifetime
- *Static objects* have an absolute storage address that is retained throughout the execution of the program
    - Global variables
    - Subroutine code
    - Class method code
- *Stack objects* are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns
    - Actual arguments of a subroutine
    - Local variables of a subroutine
- *Heap objects* may be allocated and deallocated at arbitrary times, but require an expensive storage management algorithm
    - E.g. 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)
- Static data objects are stored in static region (data section)
- Stack grows downward (data section)
- Heap grows upward (data section)



- The code section is protected from run-time modification

The period of time between the creation and the destruction of a name-to-object binding is called the binding's lifetime.

- If object outlives binding, it's garbage.
- If binding oulives objects it is a dangling reference.