Names, Bindings, Scopes

COS 301: Programming Languages

School of Computing and Information Science, University of Maine

Variables

- In imperative languages
  - Language: abstractions of von Neumann machine
  - **Variables**: abstraction of memory cell or cells
  - Sometimes close to machine (e.g., integers), sometimes not (e.g., arrays, etc.)
- In functional languages
  - Pure functional: no variables — but can have named expressions
  - Most have variables – more like pointers than true variables
- In OO languages (pure)
  - **Instance variables** only

Variable properties

- Name
- Type
- Scope & lifetime

Names

Names

- **Name** = identifiers (more or less)
- Names not just for variables, of course
  - subprograms
  - modules
  - classes
  - parameters
  - types
  - program constructs
  - …

"What's in a name?"

- Name: string of characters that identifies some program entity
- Which characters?
- Restrictions on how name begins, other implicit typing?
- Is beginning of name meaningful?
- Any special characters allowed for readability?
- **Case-sensitive** or not?
- What's allowed vs "culture" of language
  - Underscores/hyphens
  - **Camel case** (camel notation)

Length

- Early languages: 1-character names
- Too short, not meaningful
- Fortran – 6 characters (initially; 31 as of ’95)
- C – no limit, but only 63 significant
- Java, C#, Ada, Lisp – no limit, all significant
- C++ varies by implementation

Special words in the language

- Reserved words vs keywords
- **Keywords:** part of the syntax, special meaning
  - E.g., Fortran “Integer”
  - E.g., in Lisp: t, nil (cf. keyword package; package locks)
- **Reserved words:** cannot be used as keyword
  - Eliminates some confusion with multiple meanings of keywords
  - Keywords usually reserved and vice versa — but not always
  - Too many ⇒ difficult for programmer
    - E.g., Cobol has 300!
  - But some may have too few: Fortran, PL/I: no reserved words!

```
if if = then then then = else else else = then
```

- **Imported names** (packages, libraries) – function as reserved words locally

Variables

Variables

- Here: concentrate on imperative languages
- Variable: abstraction of memory cell(s)
- More than just a value!
  - Value is one **attribute** of the variable
  - Others: address, type, lifetime, scope
- I.e., variable = <name,address,value,type,lifetime,scope>

Names

- **Binding** of an identifier to a memory address
- Not all variables have names!
  - **Heap dynamic variables**
  - E.g.:

```
int *foo;

foo = new int;

*foo = 0;
```

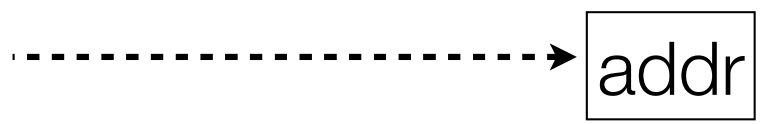
Names

- **Binding** of an identifier to a memory address
- Not all variables have names!
  - **Heap dynamic variables**
  - E.g.:

```
int *foo;

foo = new int;

*foo = 0;
```



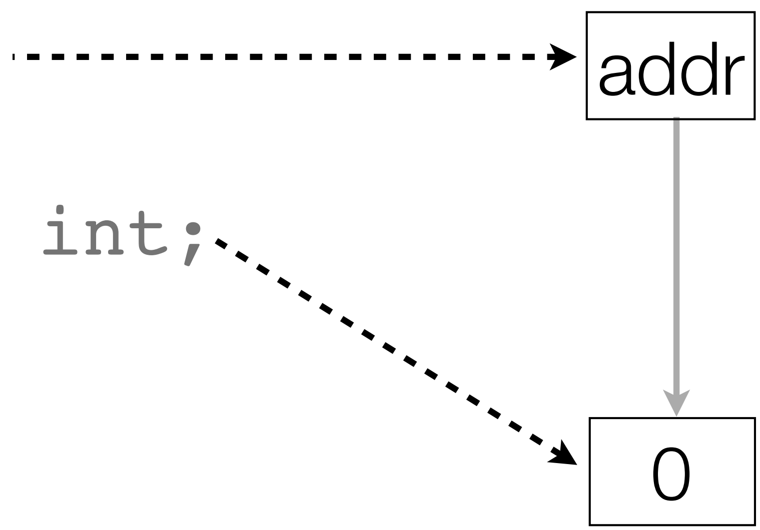
Names

- **Binding** of an identifier to a memory address
- Not all variables have names!
  - **Heap dynamic variables**
  - E.g.:

```
int *foo;

foo = new int;

*foo = 0;
```



Addresses

- **Address**: where variable is (begins) in memory
- **L-value** = address
- Not that simple, though:
  - Different addresses at different times – for the same variable
  - Different addresses in different parts of the program for the same name
  - Same address, multiple names (**aliases**)
    - **pointers**
    - **reference variables**
    - **unions** (C, C++)
    - decreases readability

Type of variable

- **Type** determines
  - size of variable (⇒ range of values possible)
  - how to interpret bits
  - which operations can be applied
- Much more about types later

Value

- Value = **r-value**
  - l-value ⇒ address
- **Abstract memory cell:**
  - Real memory cells: usually a byte
  - Abstract memory cell: size required by the type
  - E.g.: `float` may be 4 bytes ⇒ 1 (abstract) memory cell

Pointers

- **Pointers** – indirect addressing
- **Dereferencing**
- C:

Pointers

- **Pointers** – indirect addressing
- **Dereferencing**
- C:

```
int b;
b = 3;
int* ptr, other_ptr;
ptr = malloc(sizeof(int));
other_ptr = ptr;
*ptr = b;
*other_ptr = ?
```

Pointers

- Some languages: **explicit dereferencing**
  - C: x = *y + 1;
  - ML: x := !y + 1
  - Pascal: x := ^y + 1
- Other languages: **implicit dereferencing**
  - Java
  - Lisp
  - Python

Binding

Binding

- **Binding** = association between attribute and entity
  - E.g.: variable's value attribute ⇔ value
  - E.g., variable's type attribute ⇔ data type
- Binding time:
  - **Static binding**:
    - Association happens prior to run-time
    - Compiled languages, e.g.
  - **Dynamic binding**:
    - Association happens at run-time
    - Interpreted languages, e.g., some things in compiled languages

Binding times

- Language **design time**: e.g., operators ⇔ functions (operations)
- Language **implementation time**: e.g., data types ⇔ range of values
- **Compile time**: variable ⇔ type
- **Link time**: library subprogram name ⇔ code
- **Load time**: variable ⇔ address
- **Run time**:
  - variables ⇔ values – via (e.g.) assignment
  - variable ⇔ address in interpreted languages
  - variable ⇔ address via `malloc(), new`
  - instance variable ⇔ address in Java

Example

- Statement (assume PI is a constant):

```
a = b + PI + 3
```

- Bindings:
    - Types of `a`, `b`:
        - Compiled languages: compile time
        - Interpreted languages: run time
    - *Possible* values of `a`, `b`: design time (in Java; implementation time in C)
    - Value of PI: compile time or load time
    - Value of `a`, `b`: runtime
    - `+`: compile time or design time (or even run time)
    - Meaning (representation) of 3: *compiler* design time

Binding times – again

- Static binding, dynamic binding – but more complicated (of course)
- Virtual memory complicates things
  - Even with static binding, it's to a *virtual* address
  - Paging ⇒ physical address changes
  - Transparent to the program, user
- **Garbage collecting** systems (Lisp, Java, .NET, Objective C, …)
  - Some GC systems: copy active memory to another chunk of memory
  - Addresses of variables change over time
  - E.g.: Lisp has no pointers, but **references** (sometimes called *locatives*), for this reason

Type Bindings

Type bindings

- Static bindings:
  - **Explicit declaration**: statement specifies types
  - **Implicit declaration**: binding via conventions
- Pros/cons of implicit declaration:
  - Pro: writability
  - Con: reliability (and possibly readability)
- E.g.: Fortran, VB: implicit declarations
  - Fortran: I–N as first char ⇒ integer
  - Currently can change this in Fortran (`Implicit None`) and VB (`Option Explicit`)

Type bindings

- Some languages set up different **namespaces** for different types – e.g., Perl
  - `$foo` ⇒ *scalar*
  - `@foo` ⇒ array
  - `%foo` ⇒ *hash*

Type bindings

- **Type inferencing:** context ⇒ type
  - VB, Go, ML, Haskell, OCaml, F#, C#, Swift,…
  - C#: infers type from setting in `var` statement (Swift similar)

```
var foo = 3.0
var bar = 4
var baz = "a string"
```

  - ML: compiler determines from context of reference
    - `fun degToRad(d) = d * 3.1415926 / 180;`
    - `fun square(x) = x * x;`
      - int is default type
      - call square(3.5) ⇒ error
      - can fix: fun square(x) : real = x * x;

Dynamic type binding

- **Dynamic binding**: no declarations, variable assigned type based on what value it's assigned
- Rare until relatively recently
  - Lisp – early instance of dynamic binding
  - More recently: JavaScript, Ruby, PHP, Python…
  - Perl: scalar's type is dynamically bound as are types of elements of arrays and hashes

Dynamic type binding

- **Dynamic binding**: no declarations, variable assigned type based on what value it's assigned
- Rare until relatively recently
  - Lisp – early instance of dynamic binding

```
(setq a 'foo)   (setq a "hi")
(setq a 3.14159) (setq a 5/16)
```

  - More recently: JavaScript, Ruby, PHP, Python…
  - Perl: scalar's type is dynamically bound as are types of elements of arrays and hashes

Dynamic type binding

- **Dynamic binding**: no declarations, variable assigned type based on what value it's assigned
- Rare until relatively recently
  - Lisp – early instance of dynamic binding

```
(setq a 'foo)   (setq a "hi")
(setq a 3.14159) (setq a 5/16)
```

  - More recently: JavaScript, Ruby, PHP, Python…

```
list = 3
list = [3, 4.5]
```

  - Perl: scalar's type is dynamically bound as are types of elements of arrays and hashes

Dynamic type binding

- **Dynamic binding**: no declarations, variable assigned type based on what value it's assigned
- Rare until relatively recently
  - Lisp – early instance of dynamic binding

```
(setq a 'foo)   (setq a "hi")
(setq a 3.14159) (setq a 5/16)
```

  - More recently: JavaScript, Ruby, PHP, Python…

```
list = 3
list = [3, 4.5]
```

  - Perl: scalar's type is dynamically bound as are types of elements of arrays and hashes

```
$foo = 3;   $foo = 'a';
@foo=[3, "foo",3.54];
%foo = ("a" => 4, 3 => "b", "pi" => 3);
$foo{"pi"} = 3.1415926;
```

Dynamic type binding

- C# (2010) allows dynamic binding

```
dynamic foo;
```

- OOP
  - In pure OO languages: **all** variables are dynamic and can reference any object (Smalltalk, Ruby)
  - In Java: restricted to referencing particular kind(s) of object

Dynamic type binding

- Advantage: flexibility
  - E.g., write a Perl, Lisp, etc., program to average numbers without knowing what kind of numbers they are
  - Cannot do this in C, e.g. (without using pointers)

Dynamic type binding

- Disadvantages:
  - **Reliability** issues: compiler can't check types

```
i = 3;    j = "hi there"
…
foo = j;    ← typo - meant i
```

  - **Costs:**
    - Dynamic type checking ⇒ extra code/time
    - ⇒ maintain type information (runtime descriptor) ⇒ symbol table at runtime
    - Variable-sized values ⇒ heap storage, GC
    - Often interpreted languages (but can compile some [e.g., Lisp])

Storage Bindings, Lifetime

Storage bindings, lifetime

- Every variable has some **storage** bound to it
- **Allocation**: taking storage from pool of storage locations ⇒ variable
- **Deallocation**: returning storage to pool
- Variable **lifetime**: time variable is bound to storage – for scalars:
  - static
  - stack-dynamic
  - explicit heap-dynamic
  - implicit heap-dynamic

Static variables

- Storage (addresses) bound prior to run-time
- Lifetime: entire program lifetime
- Used for:
  - Global variables
  - Subroutine variables that need to exist across invocations (e.g., C/C++ static variable type)

```
int incCounter() {
  static int counter = 0;
  return ++count;
```

- "Static" variables in Java, C#, C++ classes – **class variables**

Static variables

- Efficient:
  - direct memory addressing
  - unless implementation uses a base register
- But:
  - No recursion (if only static variables)
  - No storage sharing among subprograms

Stack-dynamic variables

- Storage is on the **run-time stack**
- Type: statically bound
- Storage created at time of declaration **elaboration**:
  - Elaboration: when execution reaches declaration
  - Allocation of storage
  - Binding of storage
- Examples:
  - Parameters
  - Local variables in subroutines/methods

Stack-dynamic variables

- Everything static but address
  - **Indirect addressing**…
  - …but offset into stack is static
- Advantages:
  - **Recursion**

```
(defun fact (n)
   (cond
     ((<= n 1) 1)
     (t (* n (fact (1- n))))))
```

  - **Shared memory space** for all subprograms

Stack-dynamic variables

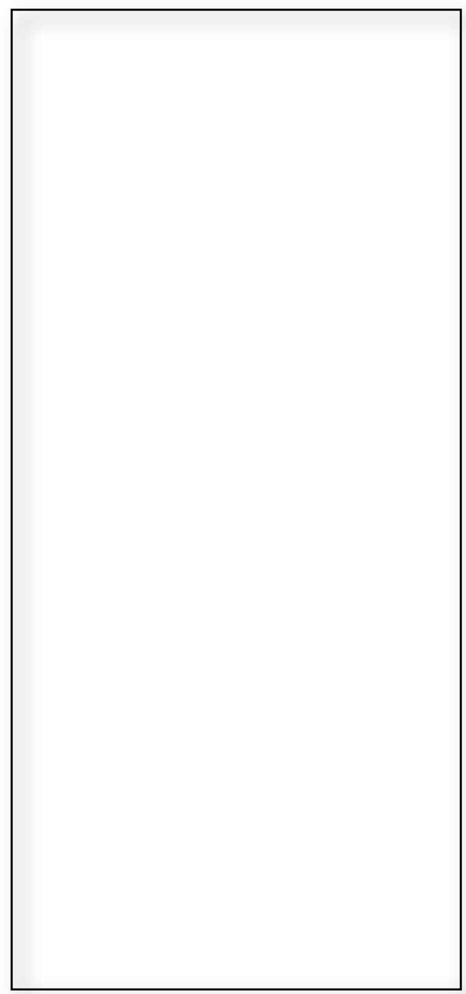
- Disadvantages:
  - Speed of access – indirect addressing
  - Time to allocate/deallocate variables (but done as a block)

Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused

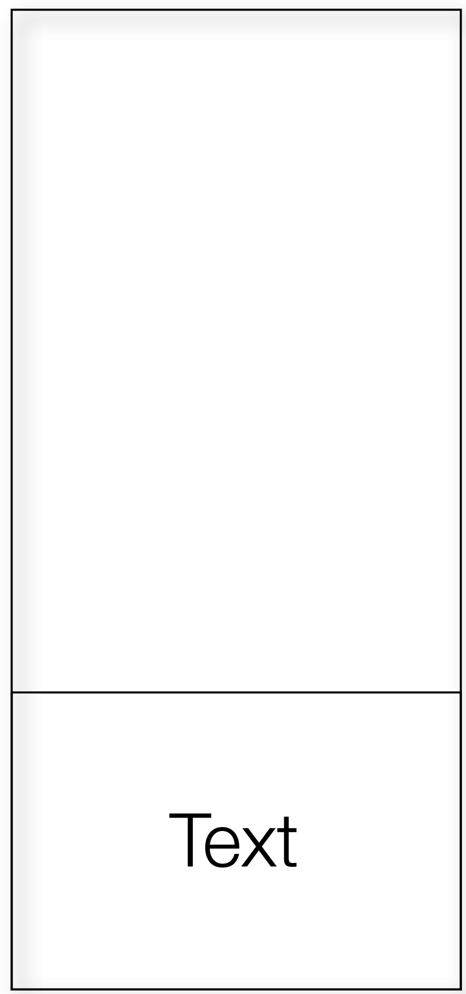
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



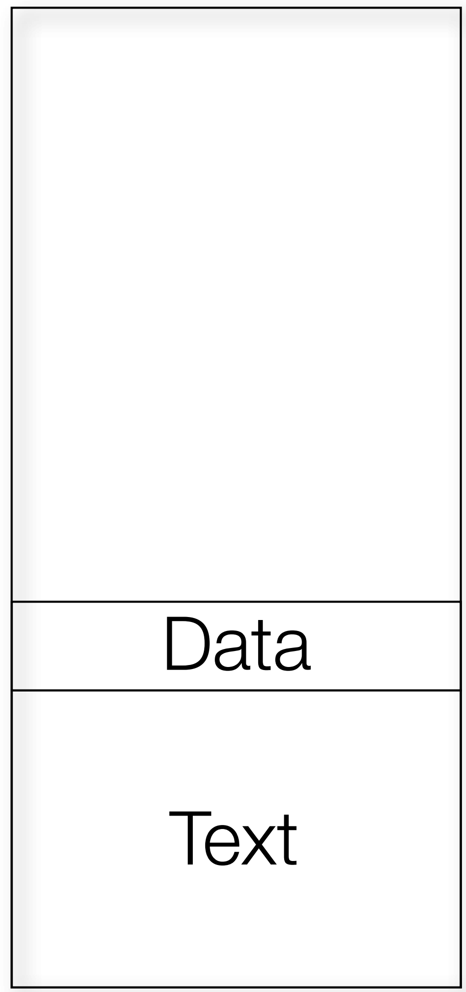
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



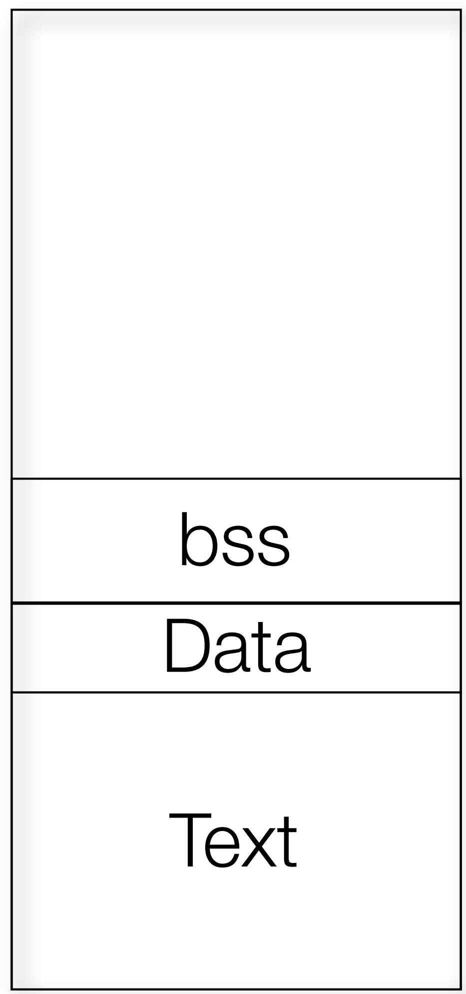
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



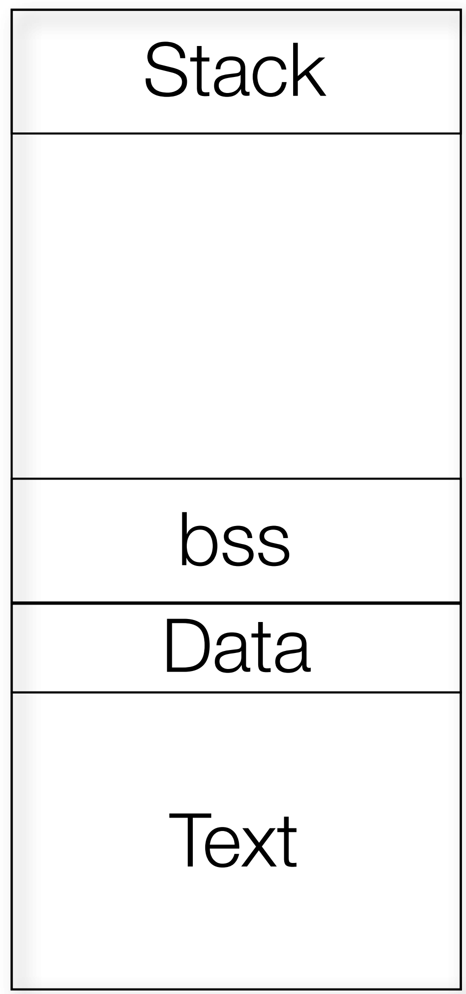
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



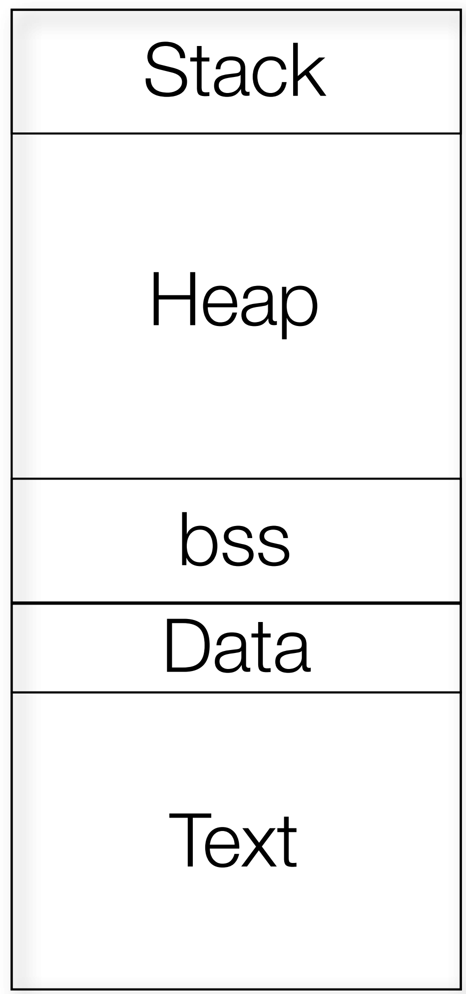
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



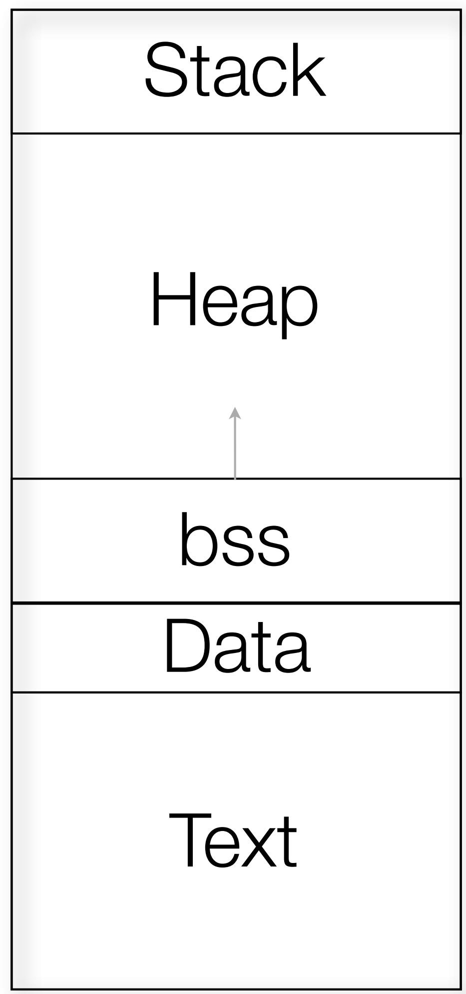
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



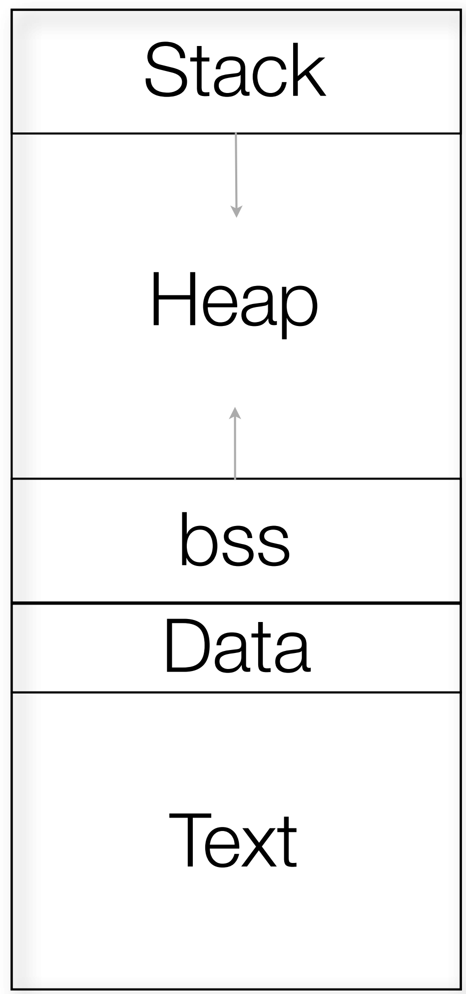
Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



Heap-dynamic variables

- **Heap**: portion of memory allocated to process, initially unused



Heap-dynamic variables

- Dynamic: allocated as needed by operator, system call (via subroutine)
- Referenced only via pointer
- Useful for:
  - data structures with size unknown at compile time
  - dynamic data structures (trees, linked lists)

Heap-dynamic variables

- Ex – C++:

```
int *foo;
foo = new int;
…
delete foo;
```

- Ex – C:

```
int *foo;
foo = malloc(sizeof(int));
…
free(foo);
```

Heap-dynamic variables

- Java:
  - **All objects** except primitive scalars → heap-dynamic
  - Created via `new`, accessed by ***reference variables***
  - No **destructor**: garbage collection
- C#:
  - Heap-dynamic and stack-dynamic variables
  - Also has pointers
- Lisp/CLOS – objects via `make-instance`

Heap-dynamic variables

- Advantage: flexibility
- Disadvantages:
  - Danger of pointers
  - Cost of reference, pointer access
  - Memory management
    - Garbage collection or manual
    - Fragmentation
    - Memory leaks

Implicit heap-dynamic variables

- Bound only when assigned variables (all attributes)
- JavaScript, Perl, Python…
- Lisp's **cons cells**
- Advantage: flexibility
- Disadvantages:
  - Those of other heap-dynamic variables
  - Also have to manage all attributes – maintain symbol table at runtime

Scope

Scope

- **Scope**:
  - Where the variable is **visible**
  - I.e., the statements in which it is visible/useable
- **Scope rules** of language:
  - Determine how references to names are associated with variables
  - Common error: inadvertently referencing a non-local variable
- **Local variables** – in program or block
- **Non-local variables**

Lexical (static) scoping

- Lexical (static) scoping — most modern languages
  - Where name defined in program matters
  - Binding of name ⇔ variable can be determined prior to runtime
- Name bound to variable in a collection of statements
  - Subprograms
  - Blocks
- Nested functions/blocks
- Algol 60 introduced lexical scoping – including begin–end blocks, nested scoping
- Nested scopes: Common Lisp, Ada, JavaScript, Scheme, Fortran (2003 and newer)
- C, C++, Java – can't nest functions

Non-local names in lexical scope

- Look in local scope first for **declaration** of variable
- If not found ⇒ look in **static parent** scope
  - If not found there, look in *its* static parent scope, etc.
  - I.e., look in **static ancestors**
- Ultimately: look in **global scope**
- If not found ⇒ undeclared variable error

Example

```
function outer {
 function inner1 {
    var x = 1;
    inner2(x);
 }
 function inner2 (y){
    function inner3 (x){
     x = x * x;
    }
    x = y + 3;
 }

 var x = 2;
 inner1();
}
```

Example

```
function outer {
  function inner1 {
     var x = 1;
     inner2(x);
  }
  function inner2 (y){
     function inner3 (x){
      x = x * x;
     }
     x = y + 3;
  }

  var x = 2;
  inner1();
}
```

Example

```
function outer {
  function inner1 {
    var x = 1;
    inner2(x);
  }
  function inner2 (y){
    function inner3 (x){
      x = x * x;
    }
    x = y + 3;
  }

  var x = 2;
  inner1();
}
```

Example

```
function outer {
  function inner1 {
     var x = 1;
     inner2(x);
  }
  function inner2 (y){
     function inner3 (x){
       x = x * x;
     }
     x = y + 3;
  }

  var x = 2;
  inner1();
}
```

Blocks

- Algol 60 → **blocks** — with scope
- Many modern languages: **block-structured languages**
- Block's local variables ⇒ stack dynamic
- C-based languages: any compound statement can have declarations ⇒ new scope
- JavaScript does not allow non-function blocks (as scopes)
- Lisp, others: **let** construct

Block example: C

```
int* swap(int* foo) {
  int bigger;
  if (foo[0] > foo[1]) {
    int temp;
    bigger = foo[0];
    temp = foo[1];
    foo[1] = foo[0];
    foo[0] = temp;
  }
  printf("bigger=%d",bigger);
  return foo;
```

Block example: Lisp

```
(defun swap (a)
  (let ((bigger 0) (smaller 0))   ;; scope 1
    (if (> (first a) (second (a))
      (let ((temp (first a)))    ;; scope 2
        (setf bigger (first a)
              smaller (second a))
        (setf (first a) (second a))
        (setf (second a) temp))
      (setf bigger (second a)
            smaller (first a)))
    (format t “Bigger=~s, smaller=~s.~%”
              bigger smaller)
    )
  a
  )
```

Nesting scope

- Varying support: JavaScript, Perl, Ruby, Python
- Nested classes, blocks in C++, Java
- Nested blocks, not subprograms, in C
- Reusing names in nested scopes:

Nesting scope

- Varying support: JavaScript, Perl, Ruby, Python
- Nested classes, blocks in C++, Java
- Nested blocks, not subprograms, in C
- Reusing names in nested scopes:

```
int count;
…
while (…) {
  int count;
  count++;
  }
```

Nesting scope

- Varying support: JavaScript, Perl, Ruby, Python
- Nested classes, blocks in C++, Java
- Nested blocks, not subprograms, in C
- Reusing names in nested scopes:

```
int count;
…
while (…) {
  int count;
  count++;
  }
```

- Allowed in C, C++
- Not in Java, C#

Nesting in **`for`** loop

- Some languages: `for` loop has its own scope
- Scope includes variables declared in initialization of loop
- E.g., C:

```
int i;
…
for (int i = -100; i<100 ;i++) {
   …
   a = 3 * i;
   …
}
```

Nesting scope – Why?

- Saves memory – only allocate what is needed
- Encapsulation (cf. OO)
- Readability/writability: keeps names close to where they are used

Accessing hidden/shadowed variables

- Variable in local scope hides or **shadows** one with same name in outer scope(s)
- Some languages (Java, C#) don't allow this in general
- Some languages allow accessing hidden variables
  - E.g., Ada: `unit.name`

Lexical scope: Summary

	Algol	C	Java	Ada	Lisp
Package	n/a	n/a	yes	yes	yes (namespace)
Class	n/a	n/a	nested	yes	yes
Function	nested	yes	yes	nested	yes
Block	nested	nested	nested	nested	nested
For Loop	no	post '89	yes	automatic	automatic

Declaration order

- Some languages: declaration can appear anywhere
  - E.g., C (99+), C++, Java, VB, C#
  - C, C++, Java – scope from declaration → end of block
  - C# – scope is whole block (but must be declared prior to use)
- Other languages:
  - Variables have to be defined prior to executable statements (e.g., Pascal)
  - Readability? Writability?

Global scope

- **Global variables** — e.g., C, C++, Lisp, Python, etc.)
  - No enclosing scope
  - Globals appear outside any function
- C/C++: one definition, but multiple declarations
  - Definition ⇒ where storage is allocated
  - Definition often also initializes the variable
  - Declarations:

```
extern int sum;
```

Global variables – accessing

- Last place to look in lexical scoping (most languages)
- Some languages: can explicitly access them – e.g., `::foo (in C++)`
- PHP: globals aren't accessible by default
  - Access via `$GLOBALS` (associative) array…
  - …or explicitly declare in function: `global $foo`
- Python:
  - Can access (read) globals inside function unless you also try to set them
  - Can set them only if declared — e.g., `global foo`
  - Can only access variables in nonlocal scope with `nonlocal`

Example – Python (v.2)

```
day = "Monday"
def tester():
  print "The global day is: ",day   #reading ok
tester()
```

output:
The global day is: Monday

```
day = "Monday"
def tester():
  print "The global day is: ",day   #reading OK
  day = "Tuesday“     #oops! Writing not OK
  print "The new value of day is: ",day
tester()
```

```
output:
UnboundLocalError: local variable 'day' referenced before assignment
```

```
day = "Monday"
def tester():
  global day
  print "The global day is: ",day
  day = "Tuesday"
  print "The new value of day is: ",day
tester()
```

```
output:
  The global day is: Monday
  The new value of day is: Tuesday
```

Globals and compilation units

- **Compilation unit:** file (e.g.) compiled separately
- Most languages: declarations at compilation unit level
- Multiple compilation units ⇒ need mechanism to make variables truly global
- C: header files – `#include <foo>`
- Or use `extern` and allow linker to resolve

Advantages of static scoping

- Static type checking is possible — at compile time
- Can directly translate references → addresses
- Does not require maintenance and traversal of binding stacks (or even symbol tables for compiled languages) at runtime

Problems with static scoping

- May provide more access to variables, functions, than necessary
- As programs evolve:
  - Initial static structure may become cumbersome
  - Tempts programmers toward making more things global over time
- Alternative: **encapsulation** (construct or objects)

Dynamic Scope

- Static (lexical) scope: depends on how program units are written
- **Dynamic scope**: depends on how they are called

Dynamic is temporal, static is spatial

- To find which variable is being referenced: Look back through chain of subprogram calls

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...


     ...
     call Sub1
     call Sub2
                  …
```

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...

     ...
     call Sub1
     call Sub2
                   …
```

Static scoping:
Sub2's X always...

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...

     ...
     call Sub1
     call Sub2
                    …
```

Static scoping:
Sub2's X always...

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...


     ...
     call Sub1
     call Sub2
                 …
```

Static scoping:
  Sub2's X always...

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...

     ...
     call Sub1
     call Sub2
                ...
```

Static scoping:
Sub2's X always...

Dynamic scoping:
Big → Sub1 → Sub2...

Scope Example

```
Big
    declaration of X
    Sub1
        declaration of X -
            ...
        call Sub2
            ...
    Sub2
        ...
        reference to X -
        ...

      ...
    call Sub1
    call Sub2
                  …
```

Static scoping:
Sub2's X always...

Dynamic scoping:
Big → Sub1 → Sub2...

Scope Example

```
Big
    declaration of X
   Sub1
      declaration of X -
          ...
      call Sub2
          ...

   Sub2
      ...
      reference to X -
      ...

    ...
    call Sub1
    call Sub2
                  …
```

Static scoping:
  Sub2's X always...

Dynamic scoping:
  Big → Sub1 → Sub2...

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...

     ...
     call Sub1
     call Sub2
               …
```

Static scoping:
  Sub2's X always...

Dynamic scoping:
  Big → Sub1 → Sub2...
  Big → Sub2...

Scope Example

```
Big
    declaration of X
    Sub1
       declaration of X -
           ...
       call Sub2
           ...

    Sub2
       ...
       reference to X -
       ...

     ...
     call Sub1
     call Sub2
                ...
```

Static scoping:
  Sub2's X always...

Dynamic scoping:
  Big → Sub1 → Sub2...
  Big → Sub2...

Dynamic scoping

- Examples:
  - APL, SNOBOL, some (early) Lisp dialects
  - Perl, Common Lisp: can declare some variables to be dynamic – e.g.:

```
(defvar *foo* 3)  ;; special (dynamic) variable
```

Dynamic scoping

- Advantage: convenience – e.g., no need for some parameter passing

- Disadvantages:

  1. While a subprogram is executing, its variables are visible to all subprograms it calls

  2. Impossible to statically type check

  3. Poor readability

Scope and Lifetime

- Scope: *where* the variable is visible
- **Lifetime**: *when* the variable has storage bound
- Often appear related – parameters, e.g.
- Often not, however – e.g., a `static` variable in C
- Scope is lexical, lifetime is temporal

Scope and Lifetime

- Fortran, COBOL:
  - static allocation to global memory area
  - ⇒ lifetime of all variables = life of program
  - memory management, ensuring unique names: programmer's responsibility
- Why?
  - Early machines had limited memory:
    - E.g., IBM 1130: 32 KB; IBM 360: 64 KB
  - Also lacked support for a call stack!
  - Could argue: use dynamic storage, but…
  - ...static gives programmer control of memory

Recall: Stack-dynamic allocation

- Algol: memory allocated/deallocated at scope entry/exit
- Allowed recursion
- Almost all modern languages do this
- **Stack frame:** What is pushed onto stack when subroutine called
  - Return address
  - Parameters!
  - Local variables
  - Pointers to stack frames for caller &/or outer scope
- On exit: pop stack frame

When Scope ≠ Lifetime

- Static scope: sometimes variable alive when out of scope

```
sub A (x)
    B(3);
    return x;
sub B (y)
    return 4*y;
```

- Static allocation (e.g., C, C++, …)
- Closures

When Scope ≠ Lifetime

- Static allocation (e.g., C, C++, …)
  - Suppose we want to count times subroutine called:

```
void foo () {
    int counter = 0;
    counter++;
    … }
```

  - Problem – counter created and destroyed
  - Solution:

```
void foo () {
    static int counter = 0;
    counter++;
    … }
```

When Scope ≠ Lifetime

- Closures
  - A function with **free (nonlocal) variables**
  - Plus an **environment** that *closes* the function
  - E.g., in Python (3.0):

When Scope ≠ Lifetime

- Closures
  - A function with **free (nonlocal) variables**
  - Plus an **environment** that *closes* the function
  - E.g., in Python (3.0):

```
def makeCounter (init):
   counter = init
   def increment():
      nonlocal counter
      counter += 1
      return counter
   return increment
```

When Scope ≠ Lifetime

- Closures
  - A function with **free (nonlocal) variables**
  - Plus an **environment** that *closes* the function
  - E.g., in Python (3.0):

```
def makeCounter (init):
   counter = init
   def increment():
      nonlocal counter
      counter += 1
      return counter
   return increment
```

```
>>> c = makeCounter(0)
>>> c()
1
>>> c()
2
>>>
```

When Scope ≠ Lifetime

- Closures
  - A function with free (nonlocal) variables
  - Plus an environment that closes the function
  - E.g., in Lisp

When Scope ≠ Lifetime

- Closures
  - A function with free (nonlocal) variables
  - Plus an environment that closes the function
  - E.g., in Lisp

```
(let ((counter 0))
   (defun count ()
     (incf counter)
     counter))
```

When Scope ≠ Lifetime

- Closures
  - A function with free (nonlocal) variables
  - Plus an environment that closes the function
  - E.g., in Lisp

```
(let ((counter 0))
   (defun count ()
     (incf counter)
     counter))
```

```
CL-USER> (count)
1
CL-USER> (count)
2
```

Referencing environments

- Referencing environment: All the names visible at some point in a program (e.g., at a statement)
- Static scoping: local vars + vars in all enclosing lexical scopes (ancestor scopes)
- Dynamic scoping: local vars + all visible vars in all active subprograms

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
- At point 2:
- At point 3:
- At point 4:

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 1:

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  X and Y of Sub1, A and B of Example

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  - X and Y of Sub1, A and B of Example
- At point 2:

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  - X and Y of Sub1, A and B of Example
- At point 2:
  - X of Sub3 (X of Sub 2 is hidden), Z of Sub3, A and B of Example

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  - X and Y of Sub1, A and B of Example
- At point 2:
  - X of Sub3 (X of Sub 2 is hidden), Z of Sub3, A and B of Example
- At point 3:

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  X and Y of Sub1, A and B of Example
- At point 2:
  X of Sub3 (X of Sub 2 is hidden), Z of Sub3, A and B of Example
- At point 3:
  X and Z of Sub 2, A and B of Example

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  X and Y of Sub1, A and B of Example
- At point 2:
  X of Sub3 (X of Sub 2 is hidden), Z of Sub3,
  A and B of Example
- At point 3:
  X and Z of Sub 2, A and B of Example
- At point 4:

Static scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  X and Y of Sub1, A and B of Example
- At point 2:
  X of Sub3 (X of Sub 2 is hidden), Z of Sub3, A and B of Example
- At point 3:
  X and Z of Sub 2, A and B of Example
- At point 4:
  A and B of Example

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 3:
- At point 2:
- At point 1:

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

Dynamic scope example

- Referencing Environments

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 3:

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 3:

  c and d of main

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 3:
  - c and d of main
- At point 2:

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 3:
  - c and d of main
- At point 2:
  - b and c of sub2, d of main (c of main is hidden)

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <------------ 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <------------ 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <------------ 3
    end -- of Sub2
  begin -- of Example
    ... <------------- 4
  end   -- of Example
```

- Referencing Environments
- At point 3:
  - c and d of main
- At point 2:
  - b and c of sub2, d of main (c of main is hidden)
- At point 1:

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 3:
  - c and d of main
- At point 2:
  - b and c of sub2, d of main (c of main is hidden)
- At point 1:
  - a and b of sub1, c of sub2, d of main (c of main and b of sub2 are hidden)

Dynamic scope example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <----------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <----------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <----------- 3
    end -- of Sub2
  begin -- of Example
    ... <------------ 4
  end   -- of Example
```

- Referencing Environments
- At point 3:
  - c and d of main
- At point 2:
  - b and c of sub2, d of main (c of main is hidden)
- At point 1:
  - a and b of sub1, c of sub2, d of main (c of main and b of sub2 are hidden)

Named constants

- Named **constant**: a "variable" bound only once to a value
- Advantages:
  - Readability: e.g., `pi` rather than 3.14159…
  - Parameterization/modifiability: e.g., `#define numAnswers 40`
- Binding:
  - Static (**manifest constants**): bound at compile time
  - Dynamic:
    - bound to value when storage is created
    - useful to bind to an expression whose value is not known until runtime

Named constants

- Example static binding in some languages:
  - Constant-valued expressions only
  - E.g., Fortran 95, C, C++ (`#define`)
  - Often no storage needed (why not?)
- Dynamic binding:
  - Example: C++

```
const int numElements = rows * columns
```

  - Ada, C++, and Java: expressions of any kind
- C# has two kinds, `readonly` and `const`
  - `const` – static
  - `readonly` – dynamic

Initialized data

- Variables can be initialized statically or dynamically
  - Static: at compile time
  - Dynamic: at runtime
- Ex:

```
int x = 0;
int  c[5] = {10,20,30,40,50}
int * foo = c;  /* foo ⇒ alias of c */
```

- Static initialization: literal values/expressions known at compile time
- Compiled language: statically-initialized variables reside in the data section of the executable file