

Overview

- What are names?
- How are names specified?
- What are attributes of names? (memory locations, types, values, scopes, lifetime)
 - How are they defined?
What programming language rules govern their existence?
 - How are they associated with names?
 - How long do they exist?
- Scope:
 1. Static scoping and scoping tree
 2. Dynamic scoping
- Storage allocation
 - Global
 - Stack
 - Heap

Names and Identifiers in Programming Languages

- Name: fundamental abstraction in programming languages. Used to name values, locations, functions, parameters, constants etc.
Four design issues in naming:
 1. Maximum length of name?
 2. Can connector characters be used in names?
 3. Are names case sensitive?
 4. Are special words reserved words?
- Name length: FORTRAN 77: up to 6, C++, Java: no restrictions.
- Some combination of letters, digits, and/or `_`. For instance, in C, names defined as:

```
(Letter|_) { (Letter | _ | digit) }
```
- Pascal, Modula-2, FORTRAN etc. do not allow `'_'` characters in names. ADA id cannot end with `'_'`.
- Some distinguish between upper case and lower case. e.g., C++) Some do not. e.g., ADA
- **Keyword**: has special meaning only in a certain context.
Example: REAL in FORTRAN has two meanings: if in beginning then type, if followed by assignment then variable.
- **Reserved word**: Special word that cannot be used as a name.

Names and Attributes

- Process of programming: defining names and associating attributes with names.
For instance: program variables, function names, constants.
- Program variable: Named abstraction of memory cells.
Translation between variables and actual memory location performed by compiler or interpreter
- A variable is defined as *name, attribute*
Example: `int X;`
- Typically five kinds of attributes:
 1. Address in memory (location)
 2. Value,
 3. Type,
 4. Scope, and
 5. Lifetime.
- **Binding**: associate a name with its attributes.
- **Binding time**: How and when are attributes bound?
 - Language design time
 - Language implementation time
 - Compile time
 - Link time
 - Load time, or
 - Runtime.
- **Static binding**: before execution. **Dynamic binding**: during execution of program.

Concept of binding - cont'd.

- Example:

```
int count;
...
count = count + 5;
```

Typical bindings:
 - **Language design**: Set of possible types for `count`
 - **Language definition time**: Set of possible meanings for `+`
 - **Compiler design time**: Set of possible values of `count`; Internal representation of 5
 - **Compile time**: Type of `count`; Meaning of `+`
 - **Execution time**: Value of `count`
- *What does it mean to bind early vs. late?*
 - Flexibility but impacts on efficiency: Ability to bind early (say, at compile time) leads to efficient execution while binding that is delayed (till runtime) leads to more flexibility
What if `count` is not bound to its type until runtime?
- *How are bindings maintained?* Typically in a symbol table that keeps track of (name, attribute) tuple.
C++ compiler: Symbol table for C++ program. No symbol information regarding program at runtime
Java: Virtual machine stores information about program: classes, methods, etc.
- *Who maintains bindings?* Depends on how names will be used during processing.

(1) Attributes of variables: Address

- Names get mapped (bound) to memory location
 - Variables with same name in different functions can get mapped to different locations.
 - A variable in same function can get mapped to different location during different invocations of same functions
 - Static binding
 - Dynamic binding
- *Aliasing*: a memory location bound to two or more names
 - aliasing created through: (i) Variant record structures (ii) Program parameters (iii) Assignment of pointers or references.
 - Readability problems due to aliasing (who changed the value?)
 - Dangling references**: location that has been deallocated but can still be accessed

(2) Attributes of variables: value

- Expression $X := X + 1$ says: increment the value stored in X, and store it in location specified in X.
 - lhs: reference, rhs: value/contents
 - **l-value**: location of a variable, **r-value**: value stored in the location of a variable
 - Some languages (Algol68 and Bliss) make a clear distinction between l-value and r-value: to access r-value, l-value must always be dereferenced
 - In C, C++, use "&" operator to access location of variable. Also "*" operator for explicitly dereferencing
 - l-value can itself be an expression: $a[i] := 5$.

ECS140A, Winter'06 Names and bindings, slide 5 ©Raju Pandey

(3) Attributes of Variables: Type

- How are types specified and when does binding take place. (More on types later.)
- **Static type binding** thorough variable declaration:
 - **Explicit**: e.g., `int x;`
 - **Implicit**: Bind type with a variable implicitly. Example: FORTRAN (I, J, K, etc \Rightarrow integer)
 - Both explicit and implicit declarations create static bindings of names.
 - Is explicit declaration a good idea? Certain class of errors can be determined at compiled time.

```
XX = XY + 1; /* suppose XY is declared,
A = XX + 1; /* mistype again */
```
- **Dynamic type binding**: A variable is bound to a type when assigned to a value.
 1. +ve: supports great deal of flexibility. (For instance, think of an array of any types.)
 2. -ves: (i) error detection capacity diminished. (ii) less efficient because type checking done at runtime.Example languages: LISP, SmallTalk.
- **Type inference**: Determine types of expressions automatically. No need to bind them explicitly. Example (written in ML language)

```
fun area(r) = 3.14159 * r * r
```

Type inferred from operators and other constants. ML rejects following because it cannot infer. User can provide hints.

```
fun square(r) = r * r
```

ECS140A, Winter'06 Names and bindings, slide 6 ©Raju Pandey

(4) Attributes of a name: Scope

- **Scope**: Used to determine the portion of code within which a name is defined and meaningful.
- Nature of scope:
 - *Global*: a name is visible in every block, module, or procedure
 - *Local*: a name is visible only within the block it is defined.
 - *Relatively global*: A local name is visible to a nested block.
- Create bindings for local objects and deactivate bindings for global (or relatively global) objects that are "hidden"
 - On exit from a scope, destroy bindings for local variables, and reactivate bindings for any "hidden" objects
- *Reference environment*: Set of active bindings
 - Determined by scoping rules
- How is a name V in scope S written?
 - V if S can be implicitly defined, or
 - S.V when S is needed to differentiate different V's in different scopes.
 - Examples: Java's package mechanism or Ada's package mechanism.
- Two kinds:
 - **Static or Lexical**: Use the program text to determine where the definition of names come from. A compiler can do that.
 - **Dynamic**: Definition of a name determined only at runtime.

ECS140A, Winter'06 Names and bindings, slide 7 ©Raju Pandey

Scope: static or lexical scoping

- Matching between usage of a name and its definition can be done at compile time (hence, called static scoping).
- Simplest model (BASIC): A single global scope.
- FORTRAN:
 - Local scope within subroutines
 - Global scopes
- Nested subroutines (Introduced in Algol 60, found in Pascal, Ada, JavaScript)
 - Example: (Pascal)

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

ECS140A, Winter'06 Names and bindings, slide 8 ©Raju Pandey

Scope: static or lexical scoping - cont'd.

- Block: Ability to introduce a scope within program units
- Block-structured languages:

Language	Feature	Declare Within?
Algol	begin — end	Yes
	Nested procedures	yes
Algol-68	begin — end	Yes
	then	yes
	do ...	yes
C,C++	{ }	yes
	global	yes
	nested procedures	no
Java	{ }	yes
	global	class statics
	nested procedures	no
Pascal	begin — end	Yes
	Nested procedures	yes
Fortran	procedure	Yes
	Nested procedures/blocks	No

- How to determine the scope of a name, especially within nested blocks?
 - Suppose a reference is made for a variable x within a block B .
 - First search for a declaration for x within B .
 - Next search for x 's declaration in block that declares B , called *static parent* of B .
 - Continue search until a declaration for x is found.

Static scope - cont'd.

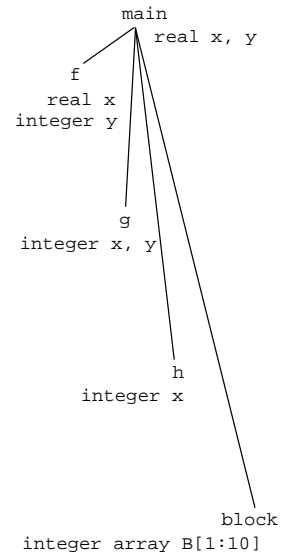
- Represent static scope through a tree: node of a tree contains the block or procedure names, and edge represents the nesting relationship.

Determine the scope of a name by traversing the tree from the node that it appears in to the root node.

- Example:

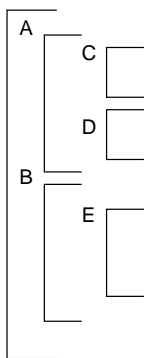
```

program main.
begin
  real x, y;
  real procedure f(x);
  real x;
  begin
    integer y;
    ...
    f := ...
  end
  procedure g(y, z);
  integer y, z;
  begin
    real array A[1:y];
    ...
  end
  integer procedure h(x);
  integer x;
  begin
    h := x*2 - 4;
  end
  begin
    integer array B[1:10];
  end ...
end
    
```

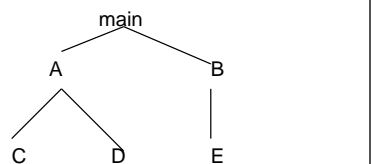


Scope: static or lexical scoping - cont'd.

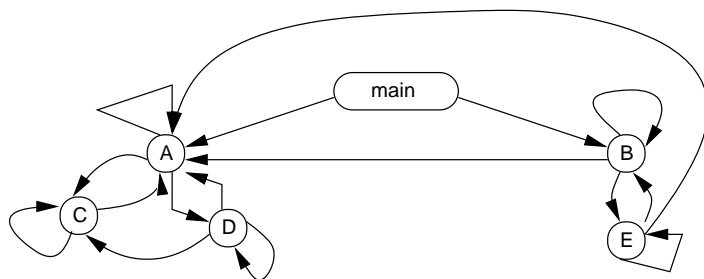
main



main, A, B, C, D, E: procedures



Allowable calls



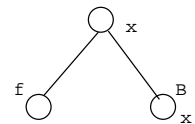
Scope: dynamic scoping

- Used in APL, SNOBOL4, and earlier versions of LISP.
- Scope of a name is based on the calling sequence of subprograms, not on their spatial relationships.

- Example:

```

begin
  integer x;
  procedure f;
  begin
    x := x + 3;
  end
  x := 20;
  begin -- block B
  integer x;
  x := 10;
  f;
  write x;
  end
  f;
  write x;
end
    
```



	x=20
B	x=10
f	x?

- Values printed
 - Static scoping: 10, 26
 - Dynamic scoping: 13, 23

Scope: static scoping vs. dynamic scoping

- Dynamic scoping useful in some cases: Example: find sum ($f(x)$), $x := 1$ to 10.

```
integer procedure sum
begin
  integer i;
  sum := 0;
  for i := 1 to 10 sum := sum + f(i)
end

begin
  integer procedure f(i);
  integer i;
  begin
    f := i*2 + 1;
  end
end
write (sum); end

begin
  integer procedure f(i);
  integer i;
  begin
    f := i*3 - 1;
  end
end
write (sum); end
```

Each invocation of sum involves using the function f defined in the current scope. Similar to passing functions..

- Local variables of a procedure are visible to all procedures that will be executed after the procedure.
- Dynamic scoping can require runtime type checking:

```
begin
  integer x = 10;
  procedure P; begin write (x) end
  begin
    boolean x := true;
    P
  end
  P
end
```

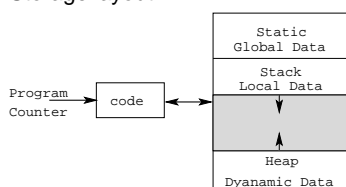
What is the type of x in procedure P?

Symbol Table

- A mechanism for associating values (or attributes) with names of programs.
- How is it used?
 - Use declarations to add name/attribute pair in the table.
 - Every time a name is used, access its attributes through symbol table.
- A symbol table data structure will provide support for storing, removing, and searching names in a table.
- Implementation mechanisms for symbol tables: Unordered list, Ordered list, Binary search trees, Hash tables (most common means of implementing symbol tables).
- **Scope stack:**
 - Every time one enters a scope, create a symbol table for the scope and push it on the stack.
 - Every time one exits a scope, pop off top symbol table stack.
 - Searching a name: Search top symbol table first, then second from top, and so on until name is found.
- **Single symbol table:** All names for all scopes appear in a single table. Each name scope is given a unique *scope number*.
 - A name may appear more than once, except with different scope numbers.
 - Provides slightly faster searching. Also tends to use space more efficiently. However, need to add extra information (scope number).

(5) Attributes of variables: Storage bindings and lifetime

- Key events:
 - Creation of data objects
 - Creating of binding
 - References to names that use bindings
 - Deactivation and re-activation of bindings that may be temporarily unusable
 - Destruction of bindings
 - Destruction of objects
- **Lifetime or extent of name:** Period of time between creation and destruction of a name-to-object binding
- **Lifetime or extent of object:** Period of time between creation and destruction of an object
A binding to an object that is no longer live is called a *dangling reference*
- Divide variables into four kinds (on basis of their lifetime):
 - (i) **Static**,
 - (ii) **Stack-dynamic**,
 - (iii) **explicit heap-dynamic**, and
 - (iv) **implicit heap-dynamic**.
- Storage layout:



Storage management

- **Static:** Bound to memory cells before program execution begins and remain bound until program execution terminates.
Examples:
 - Global variables
 - Strings in languages such as C++
 - Tables that compiler generate during program execution
 - Constants may be allocated in read only memory.Very efficient. No allocation/deallocation overhead.
- **Stack-dynamic:** Storage binding created when declaration statements for variables are reached and whose types are static. Storage allocation for blocks and procedures through stacks
As a block is entered, variables declared in the block are allocated on the stack. As a block is exit, space is deallocated by popping the stack.
- **Explicit heap-dynamic:** Storage allocation for dynamically allocated objects on heap. Can be accessed through pointers or references.
Used often for dynamic structures such as linked lists etc.
- **Implicit dynamic variables:** For certain variables, memory allocated only when assigned certain values. APL and Algol68 have this facility.