Chap 8 Symbol Table

name	attributes
:	:

- Collect attributes when a name is declared.
- Provide information when a name is used.

Two issues:

1. interface: how to use symbol tables

2. implementation: how to implement it.

§ 8.1 A symbol table class

A symbol table class provides the following functins:

1. create() : symbol_table

2. destroy (symbol_table)

3. enter (name, table) : pointer to an entry

4. find (name, table) : pointer to an entry

5. set_attributes (*entry, attributes)

6. get_attributes (*entry) : attributes

§ 8.2 basic implementation techniques

- basic operations: enter() and find()
- considerations: number of names
  storage space
  retrieval time
- organizations:

<1> unordered list (linked list/array)

<2> ordered list
  » binary search on arrays
  » expensive insertion
  (+) good for a fixed set of names
  (e.g. reserved words, assembly opcodes)

<3> binary search tree
  » On average, searching takes O(log(n)) time.
  » However, names in programsare not chosen randomly.

<4> hash table: most common
  (+) constant time

§ 8.2.1 binary search tree

- For balanced tree, search takes O(log(n)) time.

- For random input, search takes O(log(n)) time.
  - » However, average search time is 38% greater than that for a balanced tree.

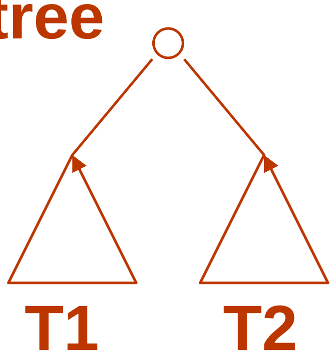
- In worst case, search takes O(n) time.

e.g. (A B C D E) ⇨ linear list

(A E B D C) ⇨ linear list

- Solution : keep the tree approximately balanced.

e.g. AVL tree



$$| \text{height}(T1) - \text{height}(T2) | \le 1$$

- Insertion/deletion may need to move some subtrees to keep the tree approximately balanced.

(+) space = O(n)

[compare] hash table needs a fixed size regardless of the number of entries.

Application: Sometimes, we may use multiple symbol tables. AVL is better than hashing in this case.

§ 8.2.2 hash tables

hash : name $\longrightarrow$ address

- **hash is easy to compute**
- **hash is uniform and randomizing.**

Ex. $(C_1 + C_2 + \ldots\ldots + C_n) \bmod m$

$(C_1 * C_2 * \ldots\ldots * C_n) \bmod m$

$(C_1 + C_n) \bmod m$

$(C_1 \oplus C_2 \oplus \ldots\ldots \oplus C_n) \bmod m$

- conflicts :

<1> linear resolution

- Try h(n), h(n)+1, h(n)+2, ...
- Problematic if the hash table did not reserve enough empty slots.

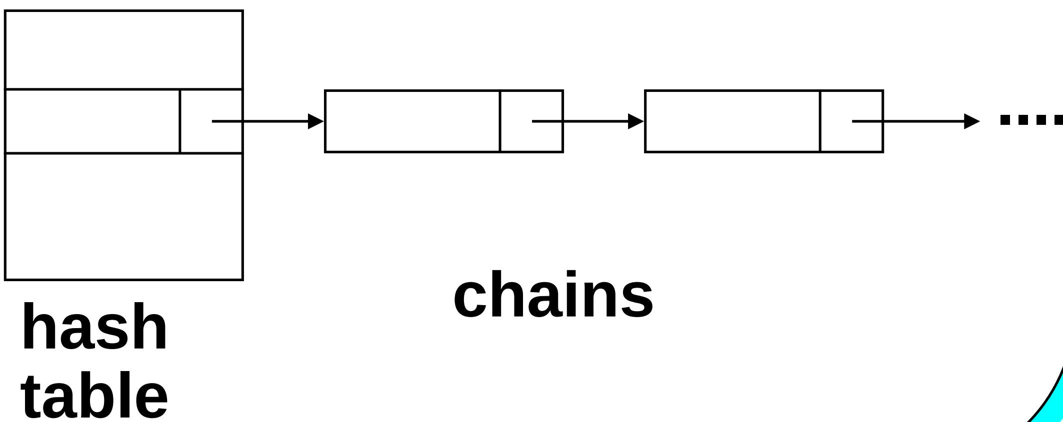
<2> add-the-hash rehash

- Try h(n), (2*h(n)) mod m, (3*h(n)) mod m, ...
- m must be prime.

<3> quadratic rehash

- Try h(n), (h(n)+1) mod m, (h(n)+4) mod m, ...

<4> chaining



- **Advantages of chaining:**
  - **(+) less space overhead of the hash table**
  - **(+) does not fail catastrophically when the hash table is almost full.**

- **The chains may be organized as search trees, rather than linear lists.**

- **More importantly, we can remove all names defined in a scope when the scope is closed.**

```
{  var  a;

      {   var a, b, c;
             ...
             ...
      }

}
```

§ 8.2.3 String space

Should we store the identifiers in the symbol table?

- Name lengths differ significantly.
  e.g. i, account_receivable,
  a_very_very_long_name

- Space is wasted if space of max size is reserved.

- Solution : store the identifiers in string space, and store an index and length in the symbol table.

Ex.

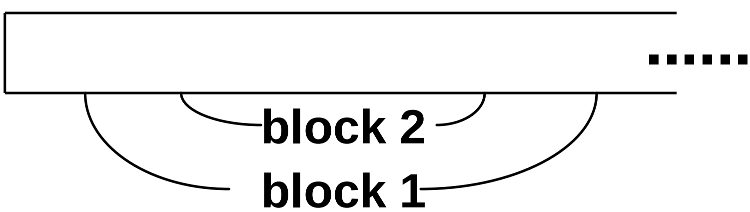
1							8					13							20
a	c	c	o	u	n	t	t	e	m	p	l	b	a	l	a	n	c	e	......

string space

symbol table

1	7	.....
13	7	.....
8	5	.....

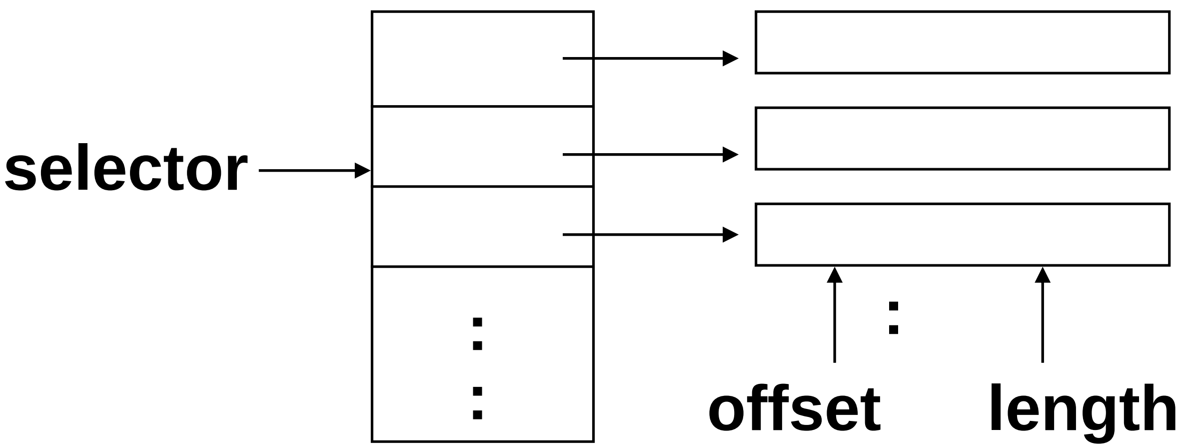
- **Usually, symbol table manages the string space.**
- **Names in string space may be re-used.**
- **Individual scopes may use different string space. However, in block-structured languages, a single string space is good for re-claiming space (when the name becomes invisible).**



- **How large is the string space?**
  - **too big — waste space**
  - **too small — run out of space**

Solution : segmented string space

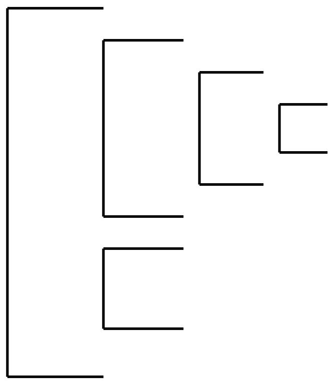
dynamically allocate 1 segment at a time.



We need (selector, offset, length) to identify a name.

§ 8.3 block-structured symbol table

- Scopes may be nested.



- Typical visibility rules specify that a use of name refers to the declaration in the innermost enclosing scope.

Ex.

```
begin      int x;
    begin    real  x;
         begin
              y:=x;
         end
    end
end
```

- **Implication: when a scope is closed, all declarations in that scope become inaccessible.**

Ex.

```
begin
        .........
          begin
                 int    x;
                 real   y;
                 type   z;
                   :
                   :
           end
      /* x,y,z are inaccessible here */
            :
            :
end
```

- Symbol tables in block-structured languages:

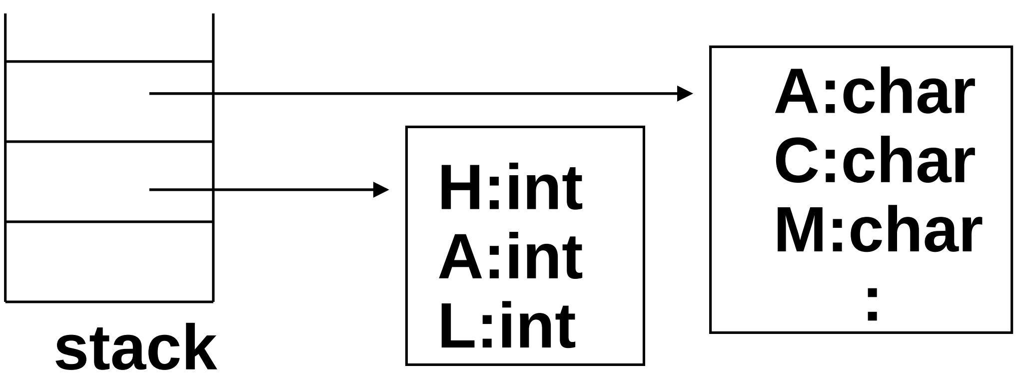
    1. many small tables
    2. one global table

<1> many small tables

- one symbol table per scope.
- use a stack of tables.
- The symbol table for the current scope is on stack top.
- The symbol tables for other enclosing scopes are placed under the current one.
- Push a new table when a new scope is entered.
- Pop a symbol table when a scope is closed.

Ex.

```
begin
    int  H,A,L;
    begin
        real   x,y;
         :
         :
    end
    begin
        char   A,C,M;
        print(M);
        H + A ..... ;
        X + L ...... ;
    end
end
```



symbol table

Chap 8 Symbol Table

- To search for a name, we check the symbol tables on the stack from top to bottom.

  (-) We may need to search multiple tables.

  E.g. A global name is defined in the bottom-most symbol table.

  (-) Space may be wasted if a fixed-sized hash table is used to implement symbol tables.

  - hash table too big -- waste
  - hash table too small -- collisions

<2> one global table

- All names are in the same table.
- What about the same name is declared several times?
    - Each name is given a scope number.
    - <name, scope number> should be unique in the table.

Ex.

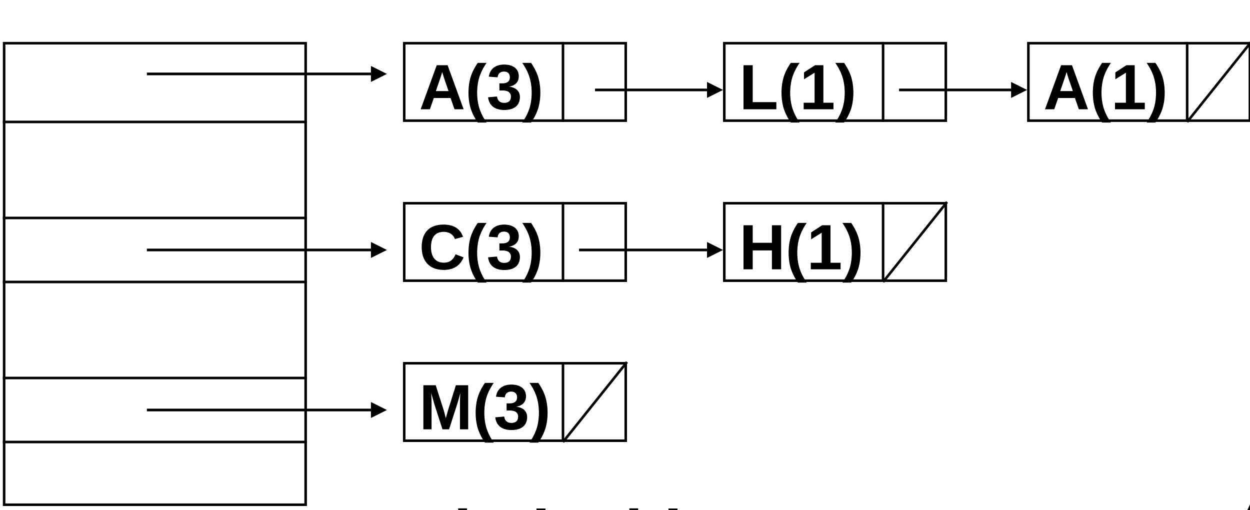
scope number

1

2

3

```
begin
    int  H,A,L;
        begin
            real  x,y;
               :
               :
         end
         begin
             char    A,C,M;
             print(M);
               H + A ....;
               X + L .... ;
         end
end
```

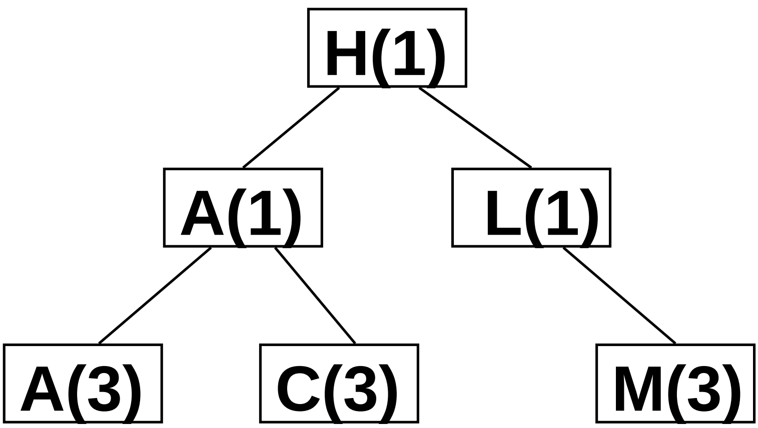


symbol table

- To search a name is easy.
- New names are placed at the front of lists.
- To close a scope, we need to remove all entries defined in that scope.
  (We need to examine each list.)

- One global table cannot be implemented with binary trees easily.
  - It is not easy to delete all entries declared in a scope when the scope is closed.
  - To find a name, we need to search the last entry (rather than the first entry).

Ex.



Consider the case when A is being searched for.

Comparisons:

many small tables	one global table
simpler	more complicated
slower	faster search
less efficient (for hash tables)	efficient storage use
	Need space for scope numbers
hash table or trees	Use hash tables
Good for keeping symbol tables of closed scopes (e.g. in multi-pass compilers).	Good for 1-pass compilers. Entries may be discarded after the scope is closed.

§ 8.4 Extensions to block-structured symbol tables

- issues :
  - different visibility rules
  - different search rules
  - multiple uses of a name in the same scope
  - implicit declaration
  - use-before-define

Ex.

1. qualified field name
   rec1.field1.field2
2. import/export rules
3. with statement in Pascal
4. use statement in Ada

- two implementation approaches:

<1> duplicate all the visible names in the current scope

(+) easy to implement

(-) significant space overhead

<2> use flags in symbol table entries to control visibility

(-) more complicated

(-) slower

(+) less space overhead

§ 8.4.1 fields and records

- The only restriction of field names is that they be unique within the record.

Ex.

```
A : record          (1)
    A : int;
    X : record      (2)
        A : real;
        B : boolean;
        end
    end
```

- References to fields must be completely qualified. Ex. A
  A.A
  A.X.A

- In PL/1 and COBOL, incomplete qualifications are allowed.

Chap 8 Symbol Table

- **HOW TO HANDLE FIELD NAMES?**

<1> one small table for each record type

The table is an attribute of the record type

‘‘‘‘’

Ex. R.A Find R

Find A in R s symbol table

(+) easy to implement

(-) waste space if hash table is used,
but good for a binary tree.

<2> treat field names like ordinary names.

- Put them in the same symbol table.

- Need a record number field
(like scope number).

- Each record type has a unique
record number.

Ex. R.A Find R

Find A with record
number of R s type.

- For ordinary names, the record
number is 0.

§ 8.4.2 export

- **Compare:**

  visibility rule: names are not visible outside.

  record rule: all names are visible outside with proper qualification.

  export rule: some, but not all, of the names are visible outside.

- **Export rules are for modularization.**

e.g. Ada packages, MODULA-2 modules, C++ classes.

Ex.

```
module   IntStack
 export    push,pop;  ←—— export rule
 const     max=25
 var       stack : array[1..max] of int;
           top : 1.. max
 procedure  push(int);
 procedure  pop() : int;
 begin   top:=1; end IntStack;
```

- How to handle export rules?

```
begin

    module
        export  A,B,C;
        int   A,B,C;
          :
          :
    end

end
```

When a scope is closed, move all exported names outside.

- How to find all exported names when a scope is closed?

For <u>export</u> located at the begining

1. When a hash table + collision chains is used -- scan each list.
2. When a binary tree, one for each scope, is used for each scope -- cluster at the root.

- Implementation:
  - Each name has a boolean exported attribute.
  - Use a close_scope() procedure to handle exported names.

- For Ada packages,

specifica-tion (exported)

```
package IntStack is
      procedure   push(int);
      function      pop(): int;
end   package
```

implemen-tation

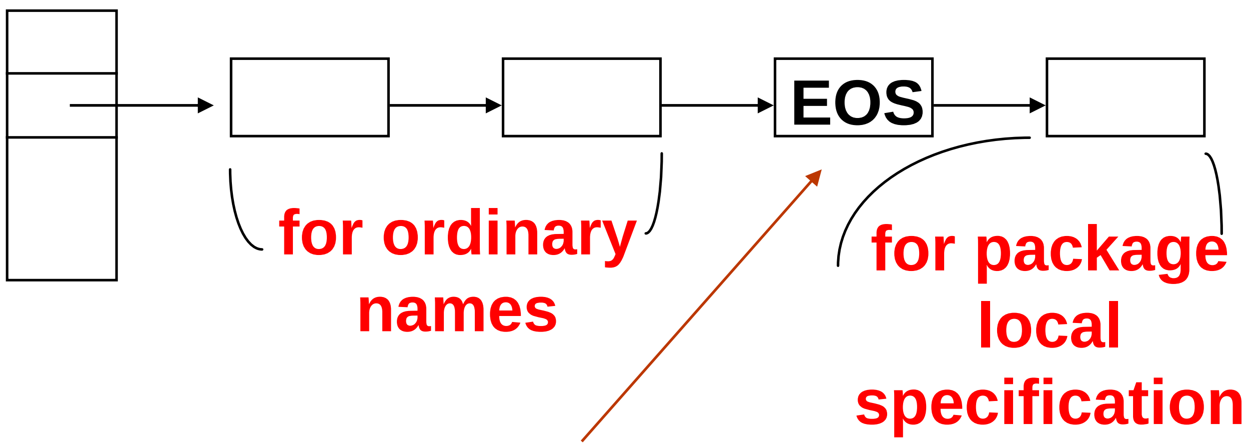
```
package body IntStack is
      max: const int := 25;
      stack: array ......
      top: int;
      procedure push() { .... }
      function pop ....
begin   top := 1;   end IntStack
```

Chap 8 Symbol Table

- The specification part of a package is exported.
- The exported names are used with qualifications, e.g. IntStack.push.
- They can also be imported with use clauses.
- How to handle packages?

  <1> one small table per package (like one table per record type).

  <2> If one big table is used, insert a end-of-search mark at each chain.



Use clauses and qualified references may pass beyond EOS during search.

- **Modular-2 is similar to Ada:**

```
definition  module  IntStack
    export  qualified  push, pop;
    procedure   push(int);
    procedure   pop: int;
end   IntStack

implementation  module  IntStack;
  .......
end
```

All exported names are accessed with qualifications, e.g. IntStack.push().

Chap 8 Symbol Table

- Separate compilation

• Packages/modules may be compiled separately.

• When package specification (or module definition) is compiled, the information is saved in a library.

• The information is used to compile package body or other packages.

• The information allows complete static checking.

• One table per package is most suitable for separate compilation.

• C uses #include
  - less efficient
  - sometimes inconsistent.

§ 8.4.3 import rules

- Two kinds of scopes:
  - importing scope: e.g. Pascal
  - non-importing scope: e.g. modules in Module-2

- In some languages, non-local objects can be imported with restrictions, e.g. read-only.

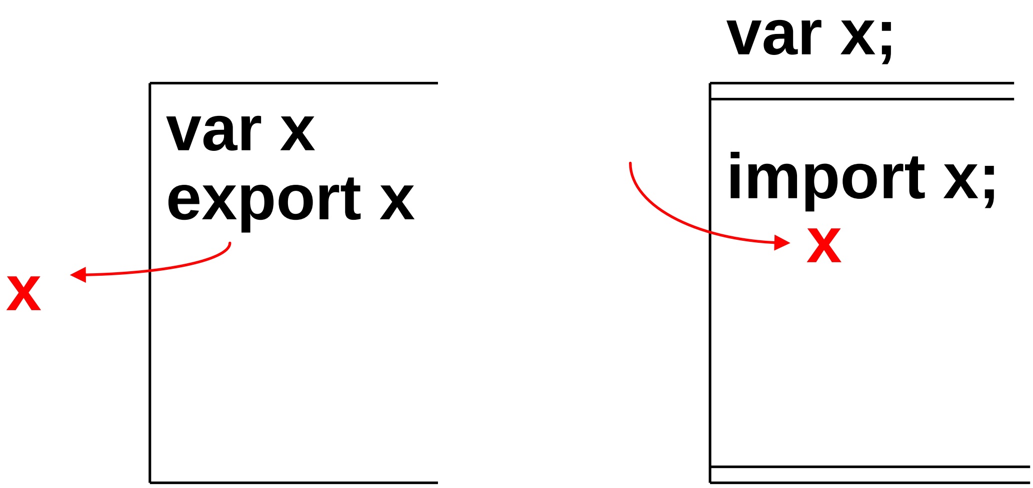
- Non-local objects must be imported level by level.

- C++ provides three classes of visibility
  - private
  - protected
  - public

private, protected — for subclasses and friends

public — for general public

Chap 8 Symbol Table



export: to the outside
import: to the inside

Ex. of import

```
module   thingStack
    import   Thing;
    var      stack : array [1..3] of Thing;
    ......
end
```

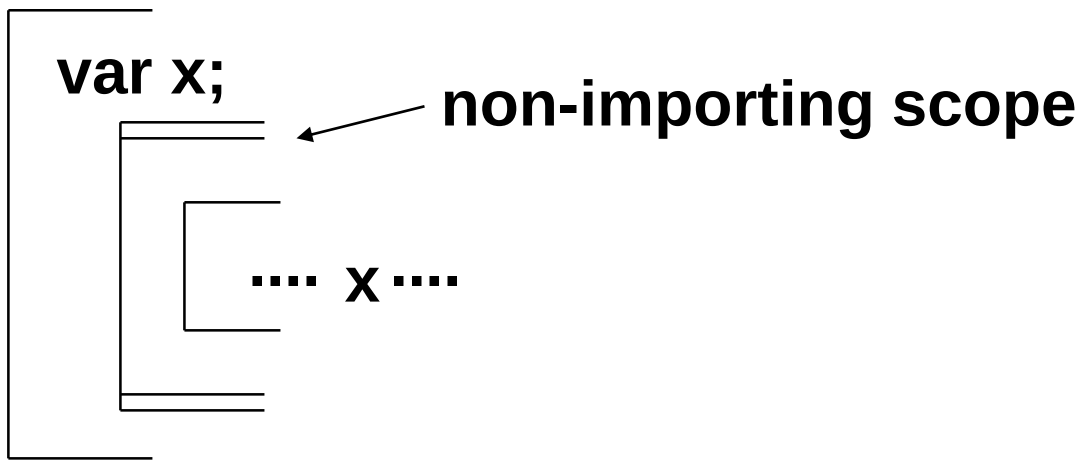
- Purpose of import rules:
  more precise control
  (for increasing reliability)

- Implementation

<1> modify search operation

Find the name

check the scope of the name definition against any non-importing scopes



<2> process import statements

For an import statement

import thing;

create entries for the imported names in the current scope.

- **The above method works well as long as not many names are imported, but is problematic when many names are imported. Consider**

```
var  A,B,C, ... ,Z;
    import  A,B,C, ... ,Z;
        import  A,B,C, ... ,Z;
            import  A,B,C, ... ,Z;
```

- Many entries will be created.

- Solution: use max-depth field for each name. (This is the maximum depth of nesting scope in which a name may be seen.)

- The max-depth field is set
  - when the name is declared, or
  - when the name is exported, or
  - when the importing scope is closed.

- difficulty: The entire symbol table must be searched to find and modify certain entries when a scope is closed.
  - Some production compilers actually do this!
  - We may move the imported names to the head of chains for easy modification.

§ 8.4.4 Altered search rules

1. with statement
2. qualified names
3. use clause in Ada

<1> with statement

Ex. **with R do <stmt>**

Within <stmt> a name is searched in the following order :

1. fields of record R
2. other scopes in the normal order

- To process with statements,

1. easy if each record and each scope has its own symbol table.
   (Use a stack of symbol tables.)
2. for a single big table,
   2.1. open a new scope
   2.2. keep a stack of all open with stmts

<2> qualified names

Ex. IntStack.A

Find IntStack's symbol table or scope number

Find A in that symbol table or in that scope

[similar to record's fields]

- In Ada/CS, packages do not nest.
  Things are simpler.

- In Ada, packages may nest.

<3> use clause in Ada

Ex. use pkg1,pkg2;

Names in pkg1 and pkg2 will become visible.

- two rules:
  1. local definition has precedence if a name is defined both locally and in pkgs.
  2. If a name is defined in both pkg1 and pkg2, the name is not directly visible.

Solutions :

1. **Enter names in packages into symbol table -- expensive if package is big.**

2. **Search a name in symbol table. If the name is not found in local table, search all packages.**

3. **similar to (2). After a name is found, put it in local symbol table to avoid repeated search.**

§ 8.5 Implicit Declaraton

Ex. FORTRAN implicitly declares variables (plus their types).

Ex. Algol 60 implicitly declares labels.

Ex. Ada implicitly declares for-loop indices and open a new scope.

different i

```
i : integer;
for  i  in  1..9  loop
     ........
     end  loop;
```

Solution for Ada :

Actually create a new scope.

or

Put a loop index in the same scope but possibly hide anexisting name temporarily.

Ex. labels in Pascal

- **In Pascal, label declarations and label usages do not mesh well. Specifically, we may not be able to jump to a label within the scope of its declaration.**

Ex.

```
label   99;
begin
      ......
      for  i:=1  to  n do begin
            99:  x:=x+1;
         end
      goto   99;   <------  illegal
   99.....
end
```

Solution: Mark the label (99) as inaccessible outside the for-loop.

§ 8.6 Overloading

• In most languages, a name may denote different objects.

Ex. In Pascal, a name may denote a function and its return value.

```
function   abc() : integer
begin
      abc := abc + 1;
end
```

as return value

as a function

• Ada allows much general overloading.

procedure names
function names
operators
enumeration literals

may be overloaded

Ex. function "+" (X,Y:complex):complex
function "+" (U,V:polar):polar

Ex.

type month is (Jan,Feb,...,Oct,Nov,Dec);
type base is (Bin,Oct,Dec);

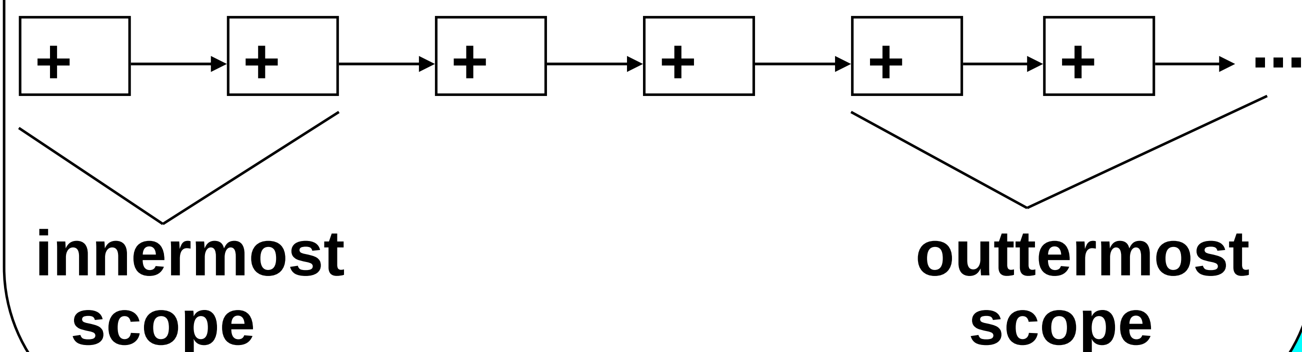
- The new Oct overloads, rather than hides, the old Oct.
- There are algorithms to determine the meaning of an overloaded name (chap 11) in a given context.
- C++ allows operators (+,-,[ ], ...) to be overloaded.

- **How to accommodate overloading in symbol table?**

  Link together all visible definitions of an overloaded name.

  When an overloaded name is used, all the definitions are available. Then we can choose an appropriate one.

  » **Establish the link when an overloaded name is defined.**
  » **Purge the link when a scope is closed**
  » **To ease purging, order the link by scope.**



Ex. in Pascal

```
program

    function   abc() : integer;
        begin   •  ←  At this point,
            :
            :     | abc | int |  →  | abc | func |

        end
        •  ←  At this point,
begin
        :         | abc | func |
        :
end.
```

§ 8.7 Forward Reference

In Pascal, pointer types may introduce forward references.

```
type  T = integer;
procedure   abc();
    type    P = ↑T;
                :
                :
    type    T = real;
```

which T does this T refer?

P is a pointer-to-real type.

Same problem for non-local goto in Algol 60.

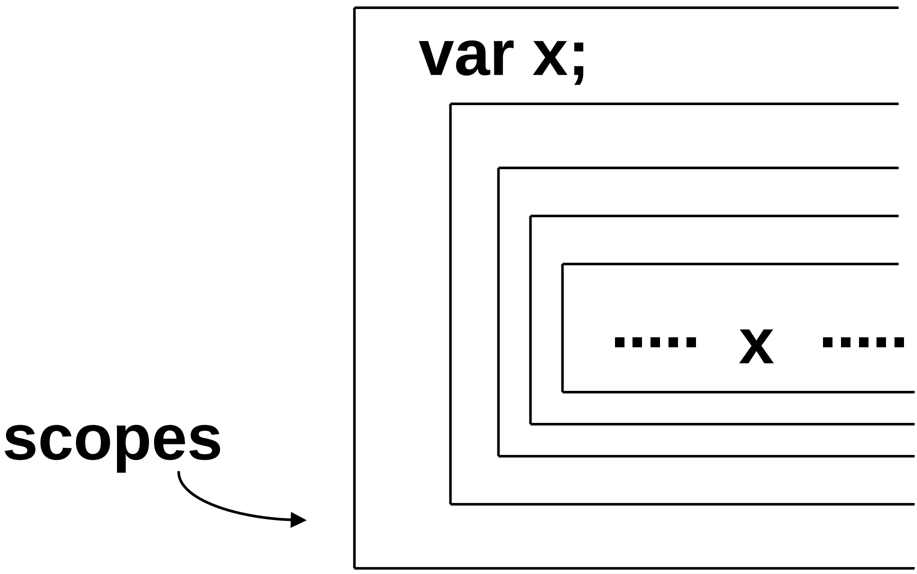
```
L :   .....

    proc   abc();
    begin
        goto  L;
             :
             :
      L :  .....

    end
```

which L does this L refer to?

Chap 8 Symbol Table

• **In the most general case**

Suppose x may be a forward reference.



Need to check scopes from inside out.

• **The Pascal label problem is simpler since labels have to be declared.**

• **This is why most languages require declarations proceede statements.**

- **How to forbid forward reference?**

In Pascal, constants cannot be forward referenced. Then consider

```
const    c=10
procedure  abc();
const   D=C;
           :
           :
const   C=20;
begin ...  end
```

- So what is the value of D?   error!

- It is not easy to detect this error.

- Most Pascal compilers cannot detect this error, including SUN pc.

- Due to this difficulty, Ada changes the visibility rule.

- How to handle forward reference?
  We need more than one pass.
  » Collect all definitions.
  » Process declarations and stmts.

- For Pascal pointer type,
  » link together all references to type T.
  » determine the real T after the type section is completely examined.

- For goto label, use backpatch in one pass.

- More general forward references,

e.g. in PL/1,

```
A = B + C
INT  A, B, C;
```

must be processed in multiple passes.

Chap 8 Symbol Table

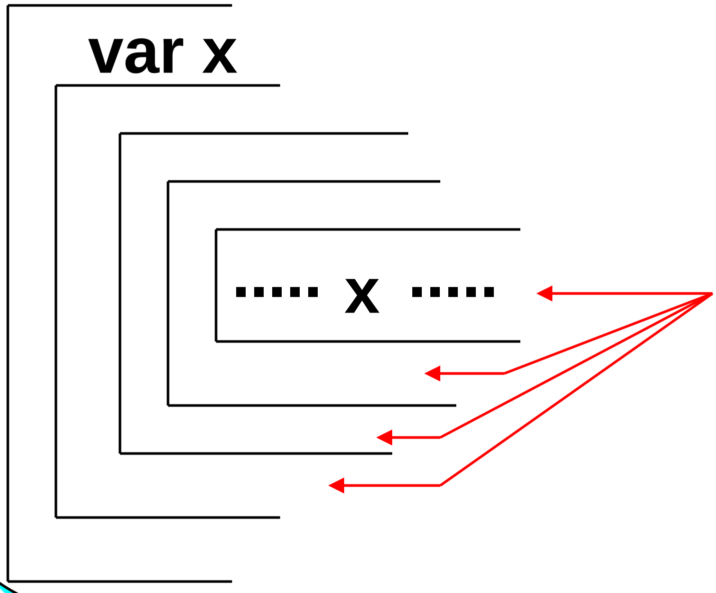
- To detect illegal forward reference,

```
const   c=10;

procedure   abc;
      const    d=c;

      const    c=20;
```

at this point, assume c is imported.

at this point, we find c is in conflict with the imported c.

In general, assume x may be forward referenced.



x is implicitly imported at these points.