

Names, scopes, and bindings

Amer Diwan

Bindings

- An association between two things
- Typically a (name, thing) pair
- E.g.,
 - (x, 5)
 - (x, address of x)
 - (procedure name, procedure code)
 - (dll name, dll code)
- Bindings come in all shapes and sizes!

“Shapes and sizes”

- **Binding time**
 - When does the binding happen?
- **Binding scope**
 - When is the binding “active”?
- **Object lifetime**
 - What is the lifetime of a bound value?
- **We will examine the above in detail along with some implementation possibilities**

Binding time

Time	Example
Language design	Control flow constructs
Language implementation	Precision
Program writing	Algorithm
Compile	Layout of data
Link	Layout of modules
Load	Instruction addresses
Run	Variable values

Why is early binding time important?

- **Earlier binding ? better performance (usually)**
- `const x = 10` (can evaluate `x+10` at compile time)
versus `int x;` (`x+10` must be evaluated at run time)
- `o.m(10)` (requires run-time method dispatch)
versus `foo(10)` (a direct call!)

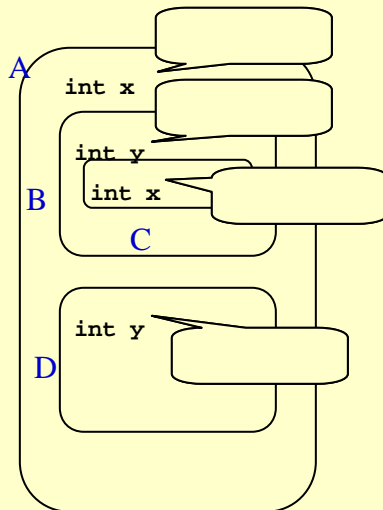
Why is late binding time important?

- **Late binding ? Greater flexibility (usually)**
- **Dynamically-linked** (can pick a different implementation for every run)
versus **Statically** (fixed implementation for all runs)
- `o.m(10)` (called procedure changes with type of “o”) versus
`foo(o, 10)` (same procedure called every time)

Binding scope

- The textual region of the program in which the binding is active
 - Static scope: determined on inspection of code
 - Dynamic scope: determined at run time

Static scope



When is a new scope created?

Scopes	Example languages
Global	Some BASIC
+ Procedure	Some FORTRAN
+ Block statements (BEGIN/END, {})	Pascal
+ File/Module	Modula-2, C
+ Classes	Java, Modula-3, C++

Advantages and disadvantages of having more scopes

- Expressiveness
- Simplicity
- Ease of implementation

What can go in a particular scope?

- Variables
- Types
- Procedures (sometimes)

What can go in a particular scope?

```
void foo(int cond) {  
  if (cond) {  
    int i;  
    void bar() {...}  
    ...  
  }  
}
```

What does it mean to declare a function inside another?

- ```
void foo(int i) {
 int j;
 void bar(int k) {
 int l;
 l = j + k;
 j = l;
 }
 bar(i);
 j = j - 1;
}
```
- bar can see its own variables and all variables in outer scopes: l, i, and j  
foo can see: bar, j, and l

## Pros and cons of allowing nested procedures

- Expressiveness
- Simplicity
- Ease of implementation

## Dynamic scope

- The run-time flow of the program and not static nesting determines the scope of a binding
- `a: integer`  
`procedure first`  
    `a := 1`  
`procedure second`  
    `a: integer`  
    `first()`  
`a := 2`  
`if cond then second() else first() end`

## Why have dynamic scoping?

- Makes more sense in macros rather than in calls
- Can use them to emulate parameters

```
print(num) {
 ...num/base...
```

```
foo() {
 VAR base: int := 10;
 print(10);
 bar();
 print(10);
}

bar() {
 VAR base: int := 16
 print(20)
}
```

## Static versus dynamic scoping (all else being equal...)

- Expressiveness
  - incomparable
- Simplicity
  - Easier to understand code with static scoping
- Ease of implementation
  - Similar
- Performance
  - Static scoping has overhead at compile time, dynamic has overhead at run time

## Object lifetime

- When do objects die?
  - Object lifetime determines how to allocate objects
- Local variables
  - When enclosing procedure returns (FIFO)
- Global variables
  - When program ends
- Dynamically allocated variables
  - At any time

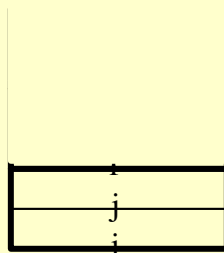
## Local variables and memory

- **Allocated** with enclosing procedure is called
- **Deallocated** when enclosing procedure returns
- **How about allocating a single copy of every local variable?**
  - Each time a procedure is calls, it uses the copy of its variables
  - No work necessary when a procedure returns

## Local variables and memory

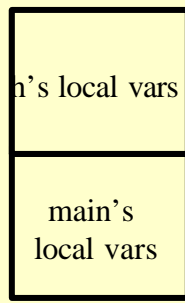
```
main() {
 int i = ...; int j = ...;
 foo(i)
}

foo(int f) {
 int k;
 if (f>10) {
 foo(f-1)
 }
}
```

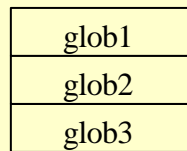


## Organization of memory

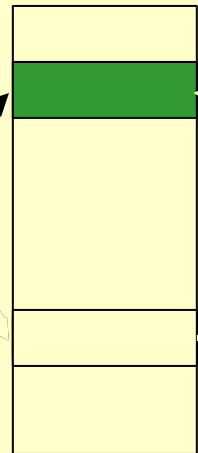
Local variables



Global variables



Dyn. alloc. variables



main calls `g`

`g` allocates heap object, assigns it to global

`g` returns

main calls `h`, ...

## Global variables and memory

- Live for the entire lifetime of the program
  - Allocated in a separate area (*static*)
  - May not be visible everywhere, however!

## Global variables, lifetime, and visibility

```
int g1;
void f() {
 int g1;
 ...
}
```

```
MODULE M1; MODULE M2; INTERFACE I;
IMPORT I; BEGIN VAR g1: INTEGER;
BEGIN I.g1 := 10 END I.
 I.g1 := 10 (*ERROR*)
END M1. END M2.
```

## Heap variables and memory

- Heap variables may live for any portion of the program execution
  - main() { v = new T; ...; free T; }
  - main() { v = new T; foo(); free T; bar(); }
- They are allocated in a separate “unstructured” area of memory

## More on memory organization

- We will discuss organization of local variables with Chapter 8 (subroutines)
- We will discuss organization of dynamically allocated memory with Chapter 7 (types)

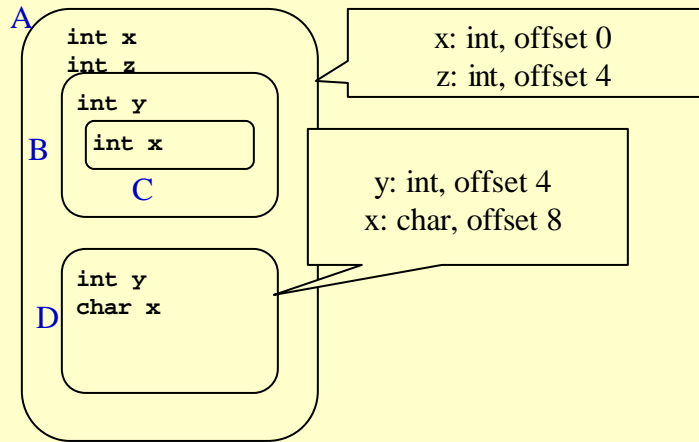
## Implementation of bindings: Symbol tables

- Issue: **How to enforce binding scope?**
- In class we will look only at mechanisms for static binding in a compiler: see text for dynamic binding

## Symbol tables

Maps names to the information that the compiler knows about them

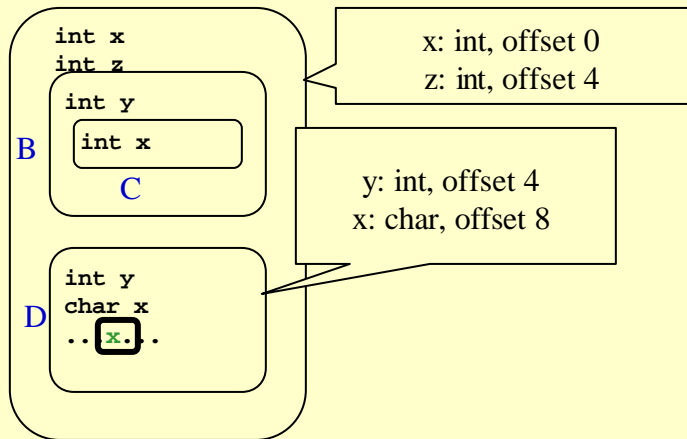
Simplest implementation: attach a symbol table to each scope



## More operationally

- Each scope node in a parse table has a symbol table node
- All declarations in the scope go in this symbol table
- When compiler encounters a name, it searches for its information by searching scopes starting with the innermost scope

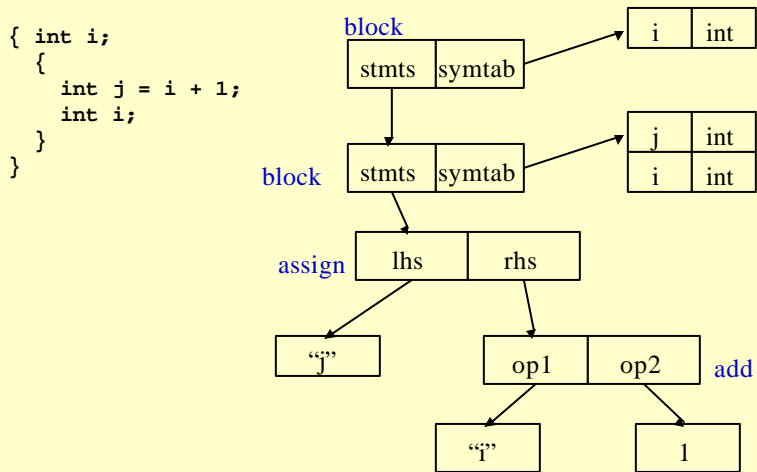
## Example



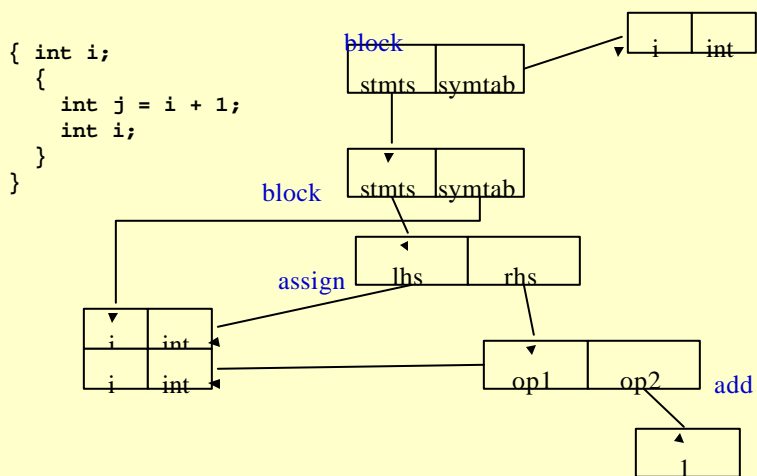
## Forward declarations and symbol tables

- ```
{ int i;  
  {  
    int j = i + 1;  
    int i;  
  }  
}
```
- Which “i” is used in the addition?
- If forward declarations are allowed, the compiler needs **two passes**:
 - pass 1: build all the symbol tables
 - pass 2: resolve all symbols

Putting it together: parsing and name analysis



After name resolution



Relationship to readings

- Text covers symbol table management in greater detail
- Text discuss some other issues with scopes
 - forward declaration
 - declarations at any point and not just at the beginning of blocks
- Both are important for project 1

Readings for next class

- Chapter 4 (except 4.5)