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

<table>
<thead>
<tr>
<th>Time</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language design</td>
<td>Control flow constructs</td>
</tr>
<tr>
<td>Language implementation</td>
<td>Precision</td>
</tr>
<tr>
<td>Program writing</td>
<td>Algorithm</td>
</tr>
<tr>
<td>Compile</td>
<td>Layout of data</td>
</tr>
<tr>
<td>Link</td>
<td>Layout of modules</td>
</tr>
<tr>
<td>Load</td>
<td>Instruction addresses</td>
</tr>
<tr>
<td>Run</td>
<td>Variable values</td>
</tr>
</tbody>
</table>
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
When is a new scope created?

<table>
<thead>
<tr>
<th>Scopes</th>
<th>Example languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Some BASIC</td>
</tr>
<tr>
<td>+ Procedure</td>
<td>Some FORTRAN</td>
</tr>
<tr>
<td>+ Block statements</td>
<td>Pascal</td>
</tr>
<tr>
<td>(BEGIN/END, {})</td>
<td></td>
</tr>
<tr>
<td>+ File/Module</td>
<td>Modula-2, C</td>
</tr>
<tr>
<td>+ Classes</td>
<td>Java, Modula-3, C++</td>
</tr>
</tbody>
</table>

Advantages and disadvantages of having more scopes

- **Expressiveness**

- **Simplicity**

- **Ease of implementation**
What can go in a particular scope?

• Variables
• Types
• Procedures (sometimes)

```c
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 - l;
}

- `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
• \texttt{a: integer}
  \begin{verbatim}
  procedure first
  a := 1
  procedure second
  a: integer
  first()
  a := 2
  if cond then second() else first() end
  \end{verbatim}

Why have dynamic scoping?

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

\begin{verbatim}
print(num) {
  ...num/base...
}

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

bar() {
  VAR base: int := 16
  print(20)
}
\end{verbatim}
**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 called, it uses the copy of its variables
  - No work necessary when a procedure returns

```c
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
---|---|---
| h’s local vars | glob1 | |
| | glob2 | |
| main’s local vars | glob3 | |

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

```c
int gl;
void f() {
    int gl;
    ...
}
```

```c
MODULE M1;
IMPORT I;
BEGIN
    I.gl := 10
END M1.

MODULE M2;
BEGIN
    I.gl := 10
    (*ERROR*)
END M2.

INTERFACE I;
VAR gl: INTEGER;
END I.
```

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
Forward declarations and symbol tables

- { int i;
  { int j = i + 1;
   int i;
  } 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

```c
{ int i;
 { int j = i + 1;
  int i;
 }
}
```

After name resolution

```c
{ int i;
 { int j = i + 1;
  int i;
 }
}
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)