CSCI-C311 Programming Languages

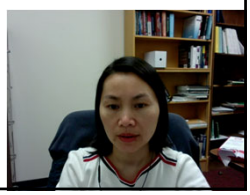
Scope and Binding

Dr. Hang Dinh

1

Outline and Reading

- After this lecture, you will learn
  - Concepts related to binding and scope
  - Static vs. Dynamic scope
  - Implementing scope
- Reading
  - Scott 4e - Sections 3.1 – 3.4



2

Name, Scope, and Binding

- A name is a character string used to represent something else
  - to refer to data using symbolic identifiers rather than addresses
  - Most names are identifiers
  - Symbols (like '+') can also be names (e.g., in Racket)
- A binding is an association between two things, such as a name and the entity (function, variable, etc) it names
- The scope of a binding is the part of the program (textually) in which the binding is active

3

Binding Time

- Binding time is
  - the time at which a binding is created
  - or more generally, the time at which any implementation decision is made
- Times at which implementation decisions may be made:
  - Language design time
  - Language implementation time
  - Program writing time
  - Compile time
  - Link time
  - Load time
  - Runtime

4

Binding Time

- *Static* (*lexical*) binding: things bound before runtime (typically at compile time)
- *Dynamic* binding: things bound at runtime.
- Early vs. Late binding
  - In general, early binding times are associated with greater efficiency
  - Later binding times are associated with greater flexibility
  - Compiled languages tend to have early binding times
  - Interpreted languages tend to have later binding times

5

Lifetime of Bindings and Objects

- Binding lifetime: time period from creation to destruction of binding
- Object lifetime: time period from creation to destruction of an object
- If object outlives binding it's *garbage*
  - Example (in C++): making a pointer NULL without deleting it first

```
int *p = new int[5];
p = NULL;
```

- If binding outlives object it's a *dangling reference*
  - Example (in C++): deleting a pointer but not making it NULL

```
int *p = new int[5];
delete p;
```

6

Storage Allocation of Objects

- Static objects
  - have absolute address that is retained throughout the program's execution
  - Examples: global or static local variables, numeric and string literals, code
- Stack objects
  - allocated and deallocated in last-in, first-out order,
  - usually in conjunction with subroutine calls and returns
- Heap objects
  - may be allocated and deallocated at arbitrary times
  - require a more general and expensive storage management

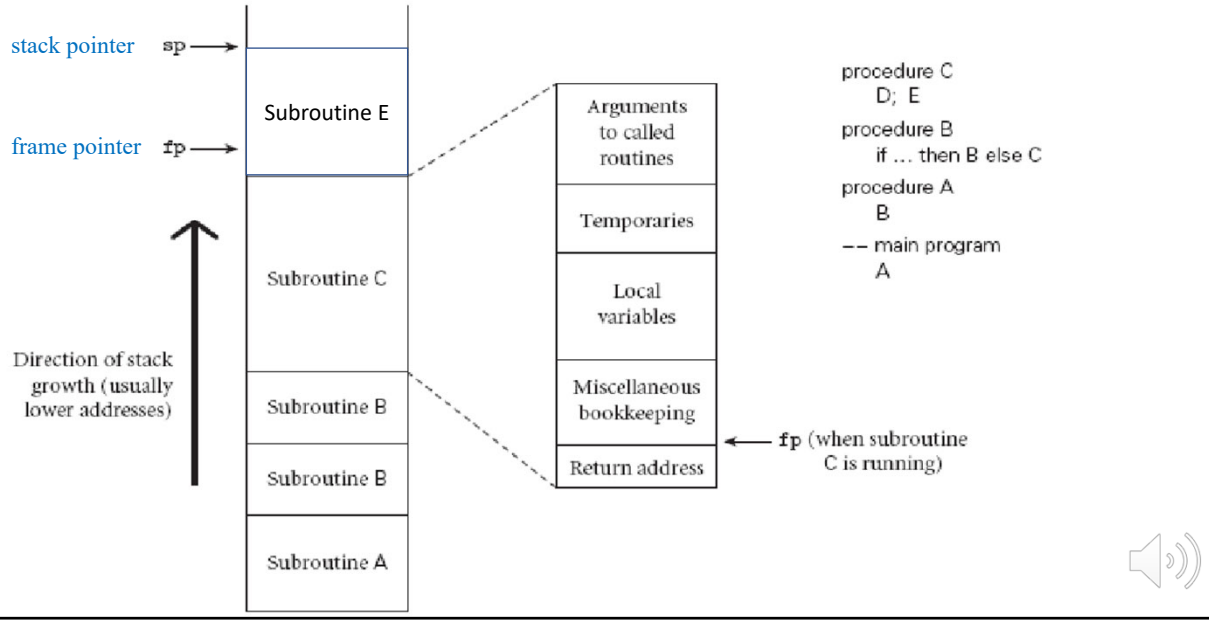
7

Stack-Based Allocation

- Why a stack?
  - allocate space for recursive routines
  - reuse space
- Contents of a stack frame (or activation record)
  - arguments and local variables
  - return address
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Each stack frame is allocated for each active subroutine.

8

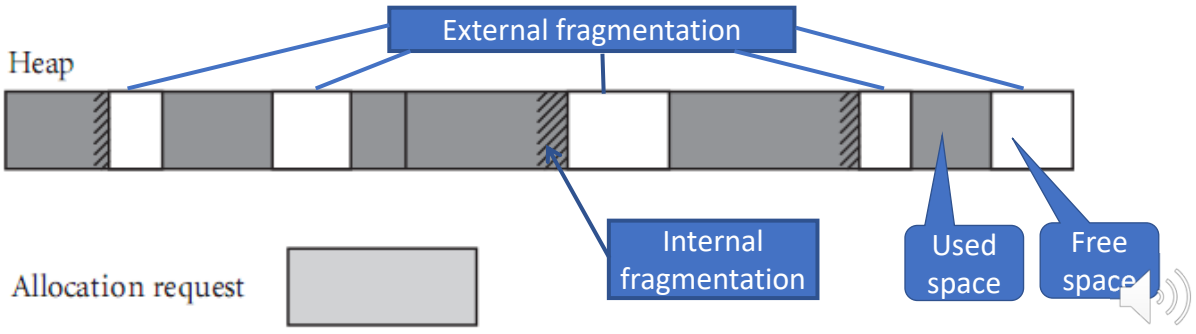
Stack-based Allocation of Space for Subroutines



9

Heap-Based Allocation

- Heap for dynamic allocation (e.g., linked lists, strings, sets)
- Principle concerns of managing a heap: *speed* and *space*
- Space concerns: internal and external *fragmentation*



10

Heap-Management Algorithms

- Maintain a single linked list – the free list – of free heap blocks
- Initially the free list contains a single block comprising the entire heap.
- At each allocation request, the algorithm searches the free list for a block of appropriate size
  - First fit algorithm: select the first block that is large enough
  - Best fit algorithm: select the smallest block that is large enough
  - If the chosen block is significantly larger than required, divide it into two and return the unneeded portion to the free list.
- When a block is deallocated and returned to free list
  - Check if both physically adjacent blocks are free; if so, coalesce them

11

Scope Rules

- Scope of a binding: The textual region of the program in which the binding is active.
- Scope: a program region of maximal size in which no bindings change (or at least none are destroyed)
  - Typically, a scope is the body of a module, class, subroutine, or structured control-flow statement, sometimes called a *block*.
- At any given point in a program's execution, the set of active bindings is called the current referencing environment.
  - The set is principally determined by *static* or *dynamic scope rules*.

12

Scope Rules

- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for local objects/variables,
  - deactivate bindings for global objects/variables that are re-declared (these variable are said to be hidden by local variables of the same name)
- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were hidden

13

Static (Lexical) Scope Rules

- In static scope rules, a scope is defined in terms of the physical (lexical) structure of the program
  - The determination of scopes can be made by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time
- Most compiled languages, C and Pascal included, employ static scope rules

14

Static Scope Rules: Example

- *Closest nested scope rules* in block structured languages
  - A name is known in the scope in which it is declared and in each internally nested scope, unless it is re-declared in a nested scope
  - To resolve a reference to a name, we start looking in the current, innermost scope and continue outward, examining successively surrounding scopes until a binding is found

```
void main(){
   int n=0;
   for(int n=1; n<100; n++)
      for(int i=1; i<=10; i++)
           cout << n <<endl;
}
```

15

Scope Rules: Static vs. Dynamic

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time
- Dynamic scope rules are usually encountered in interpreted languages

16

Scope Rules: Static vs. Dynamic Example in Pascal

```
program scopes (input, output );
var n : integer;

procedure first;
     begin n := 1; end;

procedure second;
   var n : integer;
   begin first; end;

begin
   n := 2; second; write(n);
end.
```

At issue is whether this assignment changes the variable n declared in the main program or the variable n declared in **procedure** second

- If **static** scope rules are in effect, the program prints a **1**
- If **dynamic** scope rules are in effect, the program prints a **2**

17

Scope Rules: Static vs. Dynamic Example in Pascal

```
program scopes (input, output );
var n : integer;

procedure first;
     begin n := 1; end;

procedure second;
   var n : integer;
   begin first; end;

begin
   n := 2; second; write(n);
end.
```

- Static scope rules require that the reference resolve to the most recent, compile-time binding,
  - namely the global variable **n**
- Dynamic scope rules, require that we choose the most recent, active binding at run time

19

Dynamic Scope Rules Example in Pascal

```
program scopes (input, output );
var n : integer;

procedure first;
     begin n := 1; end;

procedure second;
   var n : integer;
   begin first; end;

begin
   n := 2; second; write(n);
end.
```

At run time we create a binding for **n** when we enter the main program.

Then create another binding for n when we enter **procedure** second. This is the most recent, active binding when **procedure** first is executed. Thus, we modify the variable local to **procedure** second, not the global variable

we write the global variable because the variable n local to procedure second is no longer active

20

Use of Dynamic Scope Rules

- Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
  - This is generally considered bad programming practice nowadays
- Alternative mechanisms exist
  - static variables that can be modified by auxiliary routines
  - default and optional parameters
- Modern languages generally abandoned dynamic scoping
  - Dynamic scoping makes programs harder to understand
  - Still found in *environment variables* of the Unix programming environment.

21

Implementing Static Scope

- Static scoping is implemented using symbol table
- The symbol table is a dictionary
  - maps names to information compiler knows about them
- Basic operations on the symbol table:
  - insert a new mapping (a name-to-object binding)
  - look up information present for a given name
- Nothing is ever deleted from the table
  - It's retained throughout compilation,
  - saved for use by debuggers or runtime reflection mechanisms (type lookup)

22

Implementing Dynamic Scope

- Dynamic scoping is implemented using one of the two organizations:
- Association list (or A-list):
  - list of name/value pairs
  - functions as a stack: new declarations are pushed as they are encountered, and popped at the end of the scope in which they appeared
  - bindings found by searching down the list from the top
- Central reference table:
  - explicit mapping from names to their current meaning
  - faster lookup
  - more difficult to save a referencing environment for future use.

23