Name, Scope and Binding (2)

In Text: Chapter 5

N. Meng, F. Poursardar

Variable

- A program variable is an abstraction of a memory cell or a collection of cells
- It has several attributes
  - Name: A mnemonic character string
  - Address
  - Type

Variable Attributes (continued)

- Storage Bindings
  - Allocation
    - Getting a memory cell from a pool of available memory to bind to a variable
  - Deallocation
    - Putting a memory cell that has been unbound from a variable back into the pool
- Lifetime
  - The lifetime of a variable is the time during which it is bound to a particular memory cell

Object Lifetime and Storage Management

- **Key events**: creation of objects, creation of bindings, references to variables (which use bindings), (temporary) deactivation of bindings, reactivation of bindings, destruction of bindings, and destruction of objects.

- Binding lifetime: the period of time from creation to destruction of a name-to-object binding.
- Object lifetime: the time between the creation and destruction of an objects is the object's lifetime:
  - If object outlives binding it's garbage.
  - If binding outlives object it's a dangling reference.

- Scope: the textual region of the program in which the binding is active; we sometimes use the word scope as a noun all by itself, without an indirect object.

Lifetime

- If an object's memory binding outlives its access binding, we get **garbage**
- If an object's access binding outlives its memory binding, we get a **dangling reference**

- Variable lifetime begins at allocation, and ends at deallocation either by the program or garbage collector

Categories of Variables by Lifetimes

- Static
- Stack-dynamic
- Explicit heap-dynamic
- Implicit heap-dynamic

Storage Allocation Mechanisms

- Static: objects are given an absolute address that is retained throughout the program's execution.
- Stack: objects are 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. They require a more general (and expensive) storage management algorithm.

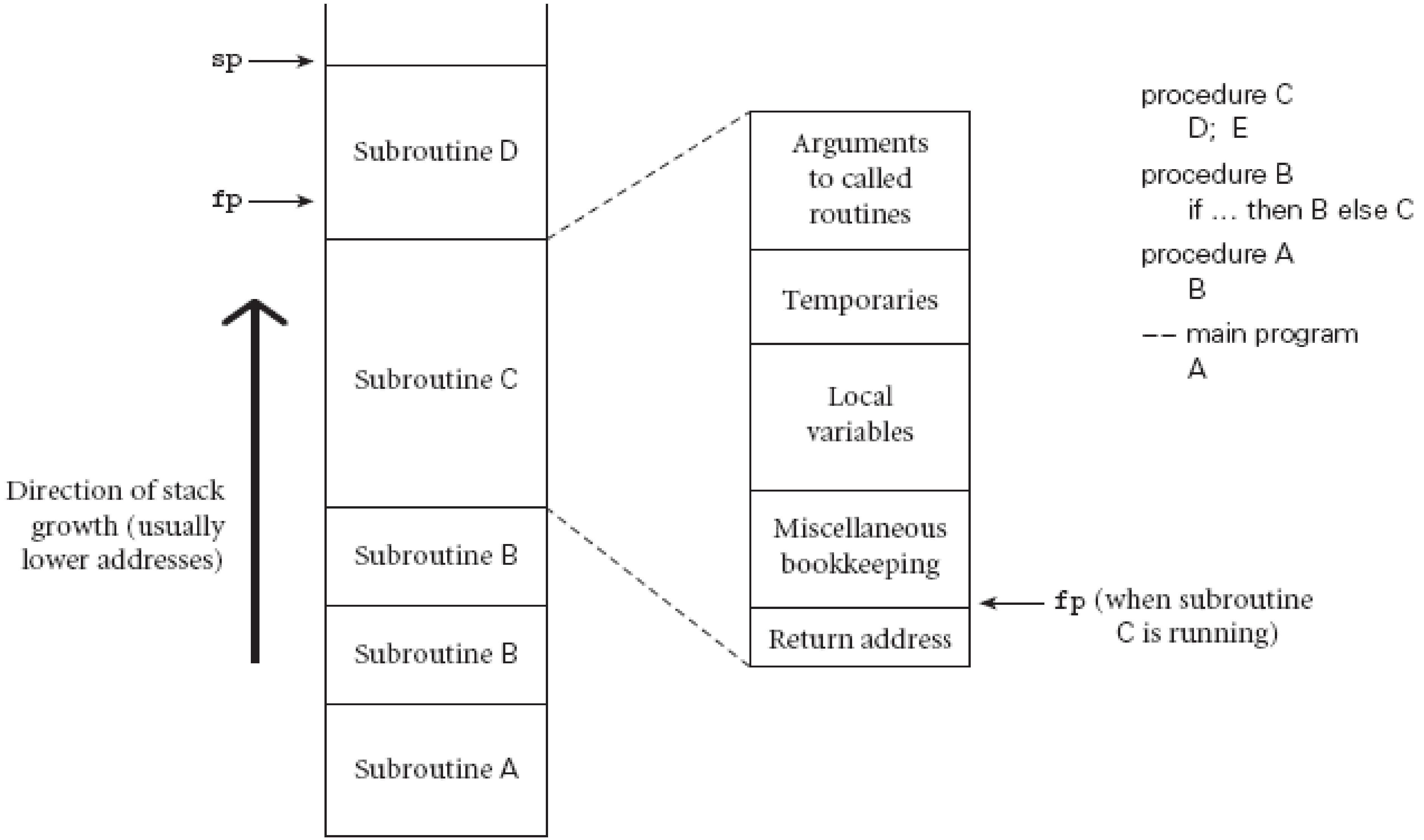
Static Allocation

- Static memory allocation is the allocation of memory at compile time before the associated program is executed
- When the program is loaded into memory, static variables are stored in the data segment of the program's address space
- The lifetime of static variables exists throughout program execution
  - E.g., static int a;

Static Allocation

- Advantage
  - Efficiency
- Disadvantage
  - Reduce flexibility
  - No support for recursive subprograms
  - No memory sharing among variables

Stack-Based Allocation for Subroutines



Stack-based Allocation

- The location of local variables and parameters can be defined as negative offsets relative to the base of the frame (fp), or positive offsets relative to sp
- The displacement addressing mechanism allows such addition to be specified implicitly as part of an ordinary `load` or `store` instruction
- Variable lifetime exists through the declared method

Heap-based Allocation

- Heap
  - A region of storage in which subblocks can be allocated and deallocated at arbitrary time
  - Its organization is highly disorganized because of the unpredictability of its use
- Heap space management
  - Different strategies achieve different trade-offs between speed and space

Heap-based Allocation

- **Explicit heap-dynamic variables** are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer.
- An example (C++):

```
int *intnode;          // Create a pointer
intnode = new int;  // Create the heap-dynamic variable
. . .
delete intnode;      // Deallocate the heap-dynamic variable
                     // to which intnode points
```

- Usage: to construct dynamic structures,
  - such as linked lists and trees, that need to grow and/or shrink during execution

Heap-based Allocation

- **Implicit heap-dynamic variables** are bound to heap storage only when they are assigned values.
- All their attributes are bound every time they are assigned.
- Example (JavaScript):

```
highs = [74, 84, 86, 90, 71];
```

- Advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written.
- One disadvantage of implicit heap-dynamic variables is the run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others

Garbage Collection

- Allocation of heap-based objects: triggered by some specific operation in a program (e.g., object instantiation).
- Deallocation: explicit in some languages (e.g., C++), implicit in others (e.g., Java).
- Garbage collection mechanism identifies and reclaims unreachable objects (implicitly deallocated).
- Explicit deallocation benefits: simplicity and execution speed provided that the programmer can correctly identify the end of an object's lifetime.
- Implicit deallocation (automatic garbage collection) benefits: eliminates manual allocation errors such as dangling reference and memory leak.

Garbage Collection Algorithms

- Reference Counting
  - Keep a count of how many times you are referencing a resource (e.g., an object in memory), and reclaim the space when the count is zero
  - It cannot handle cyclic structures
  - It causes very high overhead to maintain counters

Garbage Collection Algorithms

- Mark-Sweep
  - Periodically marks all live objects transitively, and sweeps over all memory and disposes of garbage
  - Entire heap has to be iterated over
  - Many long-lived objects are iterated over and over again, which is time-consuming

Garbage Collection Algorithms

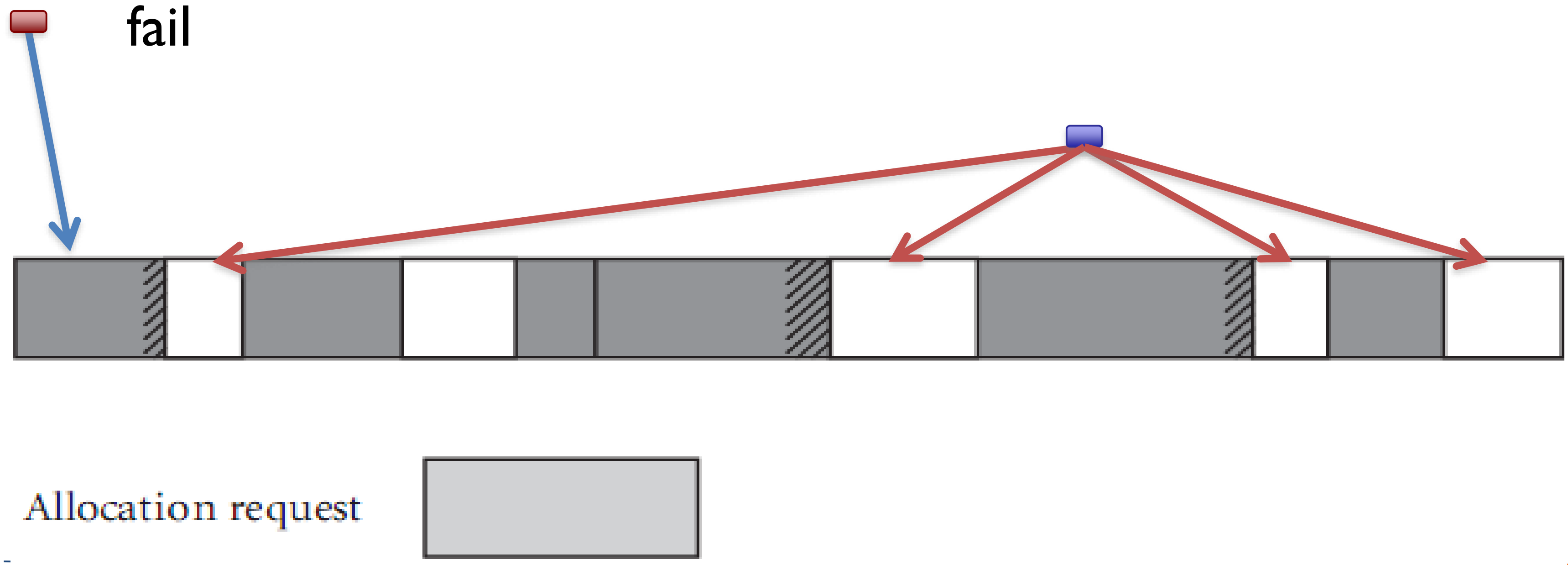
- Mark-Compact
  - Mark live objects, and move all live objects into free space to make live space compact
  - It takes even longer time than mark-sweep due to object movement

Garbage Collection Algorithms

- Copying
  - It uses two memory spaces, and each time only uses one space to allocate memory, when the space is used up, copy all live objects to the other space
  - Each time only half space is used

Space Concern

- Fragmentation
  - The phenomenon in which storage space is used inefficiently
  - E.g., although in total 6K memory is available, there is not a 4K contiguous block available, which can cause allocation to fail



Space Concern

- Internal fragmentation
  - Allocates a block that is larger than required to hold a given object
  - E.g., Since memory can be provided in chunks divisible by 4, 8, or 16, when a program requests 23 bytes, it will actually gets 32 (2^8) bytes
- External fragmentation
  - Free memory is separated into small blocks, and the ability to meet allocation requests degrades over time

Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block

Declaration Order

Examples (C#)

```
{int x;
. . .
    {int x; //illegal
    . . .
    }
. . .
}
```

```
void fun() {
. . .
for (int count = 0; count < 10;
count++){
. . .
}
. . .
}
```

Declaration Order (continued)

- In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
  - However, a variable still must be declared before it can be used
- In C++, Java, and C#, variables can be declared in for statements
  - The scope of such variables is restricted to the for construct

Scope

- The scope of a variable is the range of statements over which its declaration is visible
- A variable is visible in a statement if it can be referenced in that statement
- The **nonlocal** variables of a program unit or block are those that are visible but not declared in the unit
- Global versus nonlocal

- Two types of scope
  - Static/lexical scope
  - Dynamic scope

Scope Rules

- Scope: a program section of maximal size in which no bindings change, or at least no re-declarations are permitted.
- In most languages with subroutines, we open a new scope on subroutine entry:
  - Create bindings for new local variables.
  - Deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope).
  - Make references to variables.
- On subroutine exit destroy bindings for local variables and reactivate bindings for global variables that were deactivated.

Static Scope

- The scope of a variable can be statically determined, that is, prior to execution
- Two categories of static-scoped languages
  - Languages allowing nested subprograms: Ada, JavaScript, Python, and PHP
  - Languages which does not allow subprograms: C, C++, Java

Static Scope

- To connect a name reference to a variable, you must find the **appropriate declaration**
- Search process
  1. search the declaration locally
  2. If not found, search the next-larger enclosing unit (static parent or ancestors)
  3. Loop over step 2 until a declaration is found or an undeclared variable error is detected

An Example (Ada)

```
1. procedure Big is
2.    X : Integer;
3.    procedure Sub1 is
4.          X: Integer;
5.          begin  -- of Sub1
6.          …
7.          end;   -- of Sub1
8.    procedure Sub2 is
9.          begin   -- of Sub2
10.         … X …
11.         end;-- of Sub2
12. begin             -- of Big
13. …
14. end;      -- of Big
```

- Which declaration does X in line 10 refer to?

Variable Hiding

- Variables can be hidden from a unit by having a "closer" variable with the same name
  - "Closer" means more immediate enclosing scope
  - C++ and Ada allow access to the "hidden" variables (using fully qualified names)
    - scope.name
- Blocks can be used to create new static scopes inside subprograms

Dynamic Scope

- Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other
- Dynamic scope can be determined only at runtime
- Always used in interpreted languages, which does not have type checking at compile time

An Example

```
program foo;
   var x: integer;

   procedure f;
   begin
      print(x);
   end f;

   procedure g;
      var x: integer;
   begin
      x := 2;
      f;
   end g;

begin
   x := 1;
   g;
end foo.
```

What value is printed?

Evaluate with **static scoping**:

x = 1

Evaluate with **dynamic scoping**:

x = 2

Static vs. Dynamic Scoping

	Static scoping	Dynamic scoping
Advantages	1. Readability 2. Locality of reasoning 3. Less runtime overhead	Some extra convenience (minimal parameter passing)
Disadvantages	Less flexibility	1. Loss of readability 2. Unpredictable behavior 3. More runtime overhead

Another Example

```
void printheader() {
   …
}
void compute() {
   int sum;
   …
   printheader();
}
```

What is the static scope of sum?

What is the lifetime of sum?

Referencing Environment

- At any given point in a program's execution, the set of active bindings is called the current referencing environment.
- The referencing environment is principally determined by static or dynamic scope rules.
- Sometimes it may depend on deep and shallow binding related to the passing of parameters to subroutines.

Referencing environments in static-scoped languages

- The variables *declared in the local scope* plus the collection of all variables of *its ancestor scopes that are visible*, excluding variables in nonlocal scopes that are hidden by declarations in nearer procedures

An Example

```
procedure Example is
   A, B : Integer;
   …    ←------------------------1
    procedure Sub1 is
       X, Y: Integer;
       begin  -- of Sub1
          …        ←---------------2
      end;    -- of Sub1
   procedure Sub2 is
      X: Integer;
      begin    -- of Sub2
      …           ←---------------3
      end;     -- of Sub2
   begin              -- of Example
   …    ←----------------------4
   end;               -- of Example
```

What are the referencing environments of the indicated program points?

Point RE

1. A and B of Example
2. A and B of Example, X and Y of Sub1
3.
4.

Referencing environments in dynamic-scoped languages

- A subprogram is **active** if its execution has begun but has not yet terminated
- The referencing environments of a statement in a dynamically scoped language is the *locally declared variables*, plus the variables of *all other subprograms that are currently active*
  - Some variables in active previous subprograms can be hidden by variables with the same names in recent ones

An Example

```
void sub1() {
    int a, b;
    ...   <------------------------1
}   /* end of sub1 */
void sub2() {
    int b, c;
    ...          <------------------2
    sub1();
}   /* end of sub2 */
void main() {
    int c, d;
    ...          <----------------3
    sub2();
}   /* end of main */
```

What are the referencing environments of the indicated program points?

The meaning of names within a scope

- Within a scope,
  - Two or more names that refer to the same object at the same program point are called **aliases**
    - E.g., int a =3; int* p = &a, q = &a;
  - A name that can refer to more than one object at a given point is considered **overloaded**
    - E.g., print_num(){…}, print_num(int n){…}
    - E.g., complex **+** complex, complex **+** float

Named Constants

- A named constant is a variable that is bound to a value only once
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called manifest constants) or dynamic

Parameterize a Program

```
void example() {
int[] intList = new int[100];
String[] strList = new String[100];
. . .
for (index = 0; index < 100;
index++) {
. . .
}
. . .
for (index = 0; index < 100;
index++) {
. . .
}
. . .
average = sum / 100;
. . .
}
```

```
void example() {
final int len = 100;
int[] intList = new int[len];
String[] strList = new String[len];
. . .
for (index = 0; index < len;
index++) {
. . .
}
. . .
for (index = 0; index < len;
index++) {
. . .
}
. . .
average = sum / len;
. . .
}
```

Using a named constant as a program parameter

Named Constants (continued)

- Languages:
  - C++ and Java: allow dynamic binding of values to named variables
    - final int result = 2 * width + 1; (Java)
  - C# has two kinds, `readonly` and `const`
    - the values of `const` named constants are bound at compile time
    - the values of `readonly` named constants are dynamically bound