Name, Scope and Binding (2)

In Text: Chapter 5

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

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**

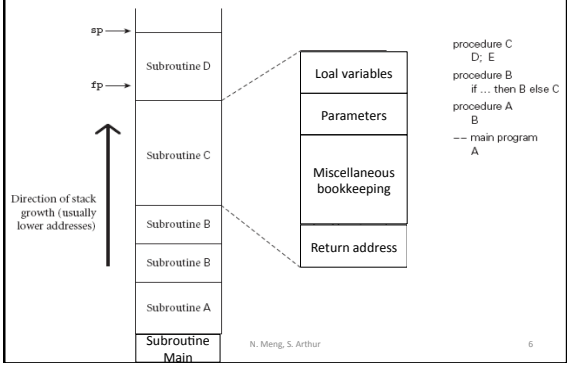
Storage Allocation Mechanism

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

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;

Stack-based Allocation



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
- Heap space management
  - Different strategies achieve different trade-offs between speed and space

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

Garbage Collection Algorithms

- Generational Garbage Collection
  - Studies show that
    - most objects live for very short time
    - the older an object is, the more likely it is to live quite long
- Concentrate on collections of young objects, and move surviving objects to older generations, which are collected less frequently

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 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 (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

Scope (continued)

- The scope rules of a language determine how a particular occurrence of a **name** is associated with a **variable**
- They determine how **references** to variables declared outside the currently executing subprogram or block are associated with their **declarations**
- Two types of scope
  - Static/lexical scope
  - Dynamic scope

Global Scope

- C, C++, PHP, and Python support a file to consist of function definitions
  - These languages allow variable declarations to appear outside function definitions
- C and C++ have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that it is defined in another file
  - E.g., extern int var;

Global Scope (continued)

- PHP
  - The scope of a variable (implicitly) declared in a function is local to the function
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the $GLOBALS array or by declaring it global

Global Scope (continued)

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

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, 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 (Common Lisp) [1]

```
(setq x 3)  ; declare lexical scoping with "setq"
(defun foo () x)
(let ((x 4)) (foo)) ; returns 3
```

```
(defvar x 3)  ; declare dynamic scoping with "defvar"
(defun foo () x)
(let ((x 4)) (foo)) ; returns 4
```

When foo goes to find the value of x,

- it initially finds the lexical value defined at the top level ("setq x 3" or "defvar x 3")
- it checks if the variable is dynamic
  - If it is, then foo looks to the calling environment, and uses 4 as x value

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 Environments

- **Referencing environments** of a statement is the collection of all variables that are visible in the statement

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

```
1. procedure Example is
2.    A, B : Integer;
3.    …   <----------------------1
4.    procedure Sub1 is
5.       X, Y: Integer;
6.       begin  -- of Sub1
7.          …      <-----------------2
8.       end;   -- of Sub1
9.    procedure Sub2 is
10.      X: Integer;
11.      begin  -- of Sub2
12.      …          <----------------3
13.      end;   -- of Sub2
14.   begin       -- of Example
15.   …   <----------------------4
16.   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

```
1. void sub1() {
2.    int a, b;
3.    …   <----------------------1
4. }   /* end of sub1 */
5. void sub2() {
6.    int b, c;
7.    …          <-------------------2
8.    sub1();
9. }   /* end of sub2 */
10.void main() {
11.   int c, d;
12.   …          <----------------3
13.  sub2();
14.}   /* 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

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