Chapter 3: Names, Scopes, and Bindings

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

High-level programming languages

- High-level features relative to assembly language
- 'Highness' = degree of abstraction
  - machine independence
    - efficient implementation does not depend on a specific machine instruction set
    - relatively easy goal
  - ease of programming
    - hard goal
    - more aesthetics(藝術的), trial and error than science ?
- Programming language design
  - find the right abstractions

Name

- Mnemonic character string to represent something else
  - usually identifiers, e.g., bull, cow, boy, girl, etc.
  - some languages allow names like ‘+’ & ‘:=‘
- Enables programmers to
  - refer to variables *etc.* using
    - symbolic names rather than (e.g., $r0, $s0, sp, or fp, etc)
    - low-level hardware addresses (e.g., 0x3F3D or 0x11AA)
  - abstract their control and data structures
    - express purpose or function instead of implementation
    - make programs manageable
    - control abstraction: subroutines
    - data abstraction: classes (for example)

Binding & Binding Time...

- Binding
  - an association between a name and the thing that is named
- Binding time
  - the time at which an *implementation decision* is made to create a binding
- The time spent in implementation decision
  - Language design time
    - the design of specific program constructs (syntax)
    - primitive types
    - meaning (semantics)
  - Language implementation time
    - fixation of implementation constants such as
      - numeric precision
      - run-time memory sizes
      - max identifier name length
      - number and types of built-in exceptions, etc.

...Binding times

- Program writing time
  - programmer’s choice of algorithms and data structures
- Compile time
  - translation of high-level constructs to machine code
  - choice of memory layout for objects
- Link time
  - multiple object codes (machine code files) and libraries are combined into one executable code
- Load time
  - operating system loads the executable code in memory
- Run time
  - program executes

Nature of bindings

- Static: things bound before execution
- Dynamic: execution-time bindings
- *Early binding*
  - efficiency (e.g. addressing a global variable in C)
  - languages tend to be compiled
- *Late binding*
  - flexibility (e.g. polymorphism in Smalltalk)
  - languages tend to be interpreted
- Our current interest
  - binding of *identifiers* to *variables* they name
  - note: all data has not to be named (*e.g.* dynamic storage)

Things we have to keep track of

- The differences between names and objects by identifying several key events following:
  - creation
    - of objects
    - of bindings
  - references to variables or subroutines, etc. (which use bindings)
  - deactivation and reactivation of bindings
  - destruction
    - of bindings
    - of objects
- If we don’t keep good track of them we get
  - garbage: object that outlives it's binding and
  - dangling references: bindings that outlive their objects

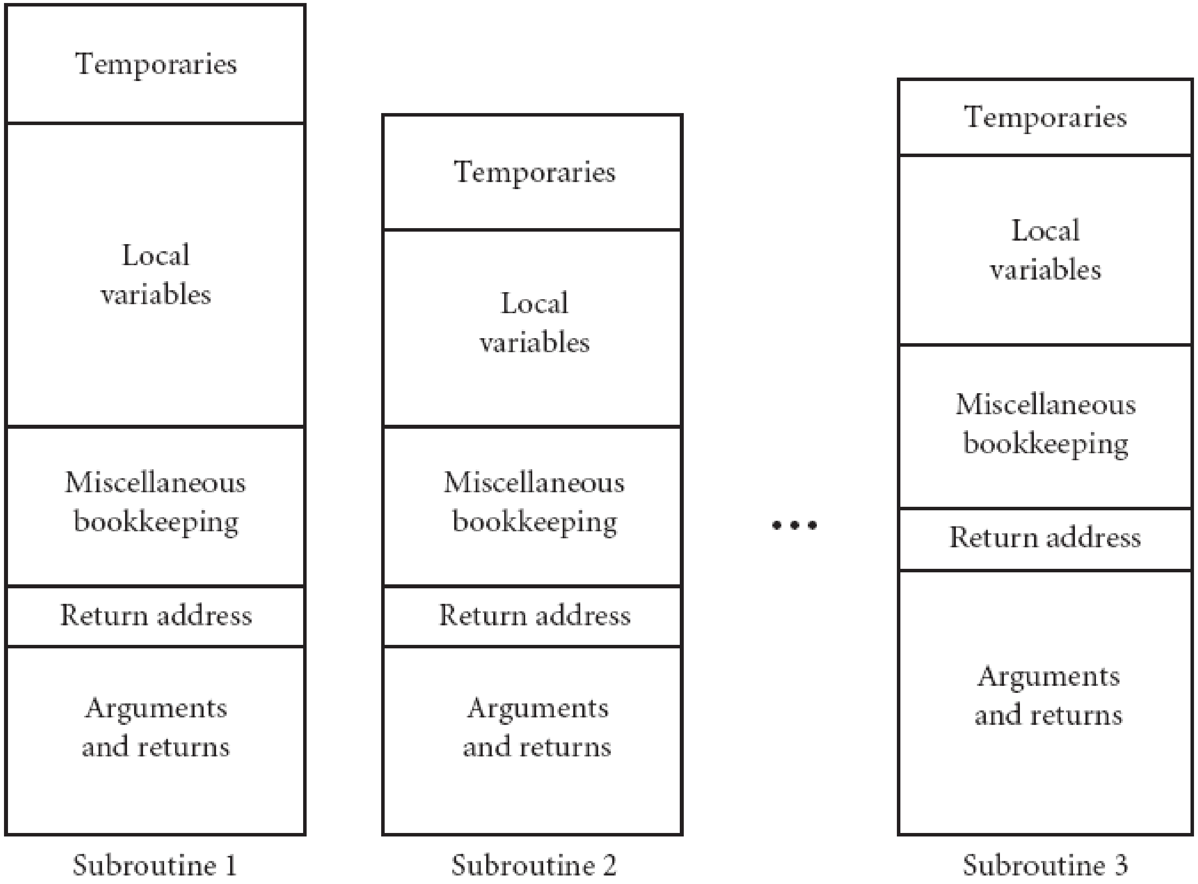
Lifetime

- Binding lifetime
  - time between the creation and destruction
- Object lifetime
  - defined similarly, but not necessarily the same as binding lifetime
  - e.g. objects passed as reference parameters
  - generally corresponds to the *storage allocation* mechanism that is used to allocate/deallocate object’s space

Storage allocation

- *Static* objects
  - have absolute (and same) address through the program execution
  - space is ‘part of the program’
- *Stack* objects
  - allocated/deallocated in LIFO order (usually) in conjunction of subroutine calls/exits
- *Heap* objects
  - can be allocated/deallocated at arbitrary times
  - storage management more complex than with stack

An Example of Static Allocation



Static objects

- Global variables
- Translated machine code
    - subroutine locations in particular
- Constants
    - large ones in ‘constant pool’
    - small ones stored as part of instructions
- Subroutine variables that are declared static
- Run-time support tables (produced by the compiler)
    - symbol table, dynamic type checking, exception handling, garbage collection, ...
- Note: processor-supported memory protection is possible for constant data

Static or non?

- Local variables in non-recursive languages
  - e.g. early Fortrans
  - all data (subroutines included) can be allocated statically
  - pros: faster execution
  - cons: wastes space, bad programming practices
- Local constants
  - *compile-time constants* can be allocated statically
  - *elaboration-time constants* must be allocated from stack
    - each invocation may have a different value

Other information associated with subroutines

- Arguments and return values
  - compilers try to place these in registers if possible
  - if not, then the stack is used
- Temporary variables
  - hold intermediate values of complex calculations
  - registers / stack
- Bookkeeping information
  - return address (dynamic link)
  - saved registers (of the caller)
  - ...

Why a stack?

- Subroutine call / return is ‘stack-like’
  - → stack is the natural data structure to support data allocation & deallocation
- Allows recursion
  - several *instances* of same subroutine can be active
- Allows re-using space
  - even when no recursion is used

Maintaining the Stack

- Each subroutine call creates a *stack frame*
  - also called an *activation record*
  - bottom of the frame: arguments and returns
    - easy access for the caller
    - always at same offset
  - top of the frame: local variables & temporaries
    - compiler decides relative ordering
- Maintenance of stack is done by (see Section 8.2)
  - *prologue (code executed at the beginning) & epilogue (code executed at the end)* in the subroutine
    - saves space to do much here
  - *calling sequence* in the caller
    - instructions immediately before/after call/return
    - *may* save time to do much here
    - interprocedural optimizations possible

Addressing stack objects

- Offsets of variables *within* a frame can be decided at compile-time
- Locations of frames themselves may vary during execution
- Many machines have
  - a special *frame pointer* register (fp)
  - load/store instructions supporting indirect addressing via fp
- Address = fp + offset (Fig. 3.2)
  - locals, temps: negative offset
  - arguments, returns: positive offset
  - stack grows ‘downward’ from high addresses to low ones
  - push/pop instructions to manipulate both fp and data

Stack-based Allocation of Space for Subroutines

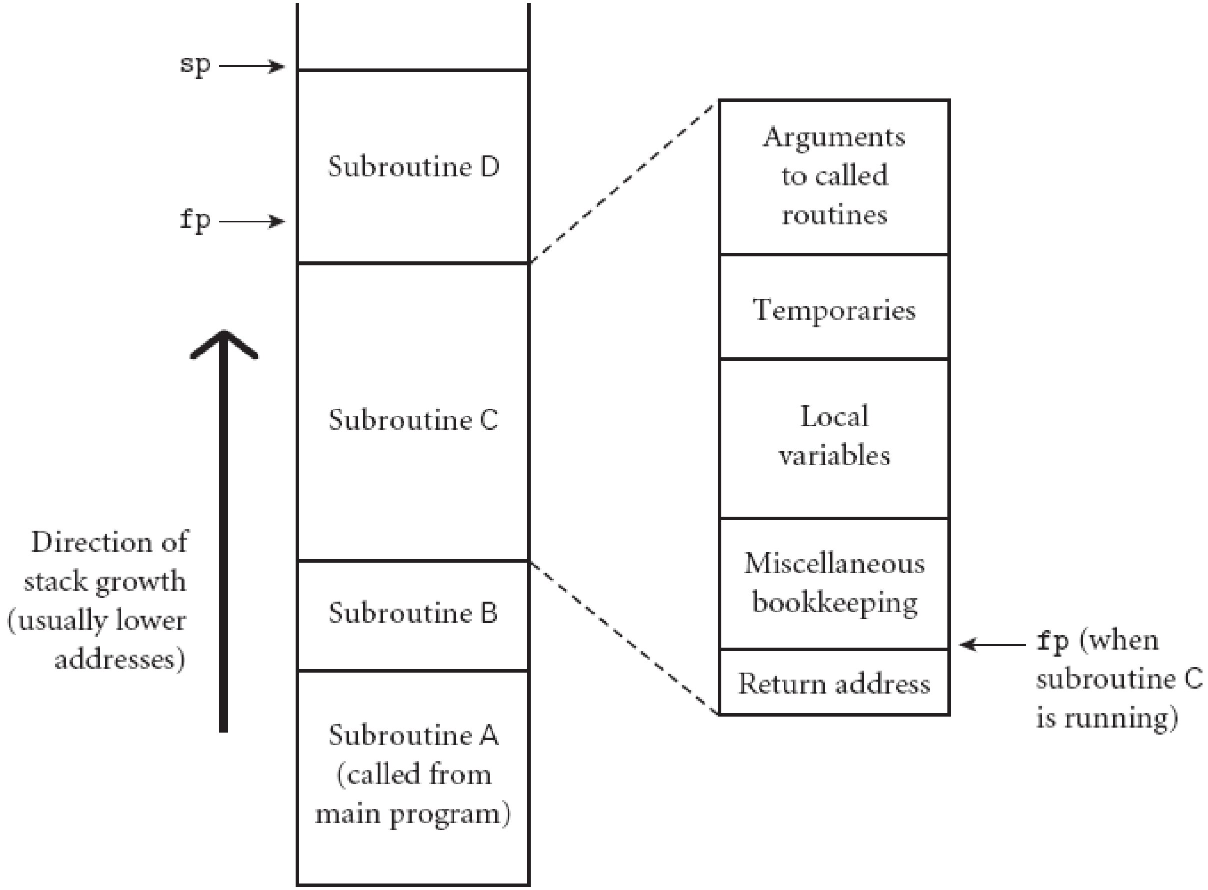
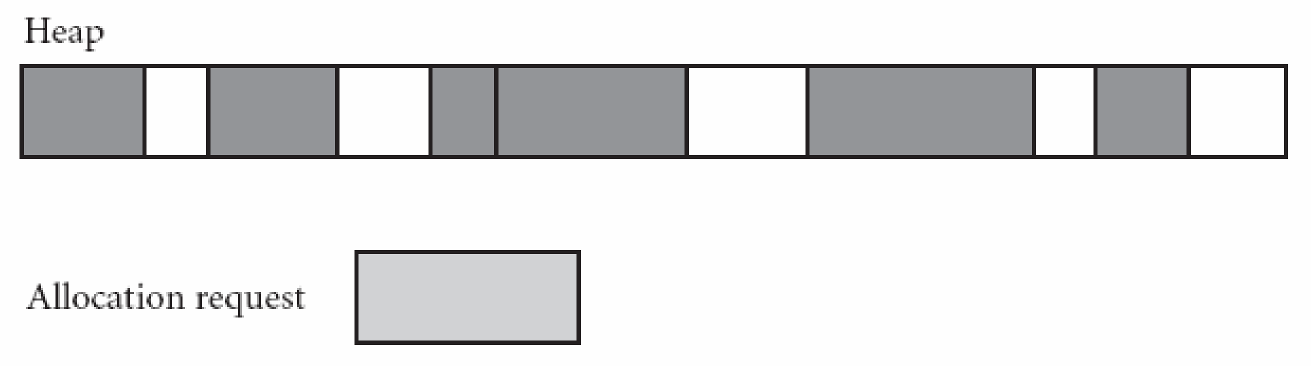


Figure 3.2

Heap-Based allocation

- Heap is here not a priority queue
  - region of storage to support allocation/deallocation at arbitrary times
  - necessity for dynamic data structures
- Many space-management strategies
  - should be part of your data structures course?
  - space & speed concerns
  - space: internal & external *fragmentation* (O.S.)
- Internal
  - allocation of a block larger than required
  - happens because of standard-sized blocks
- External
  - blocks are scattered around the heap
  - lot of free space but in small pieces

An Example of External Fragmentation



Block allocation

- Maintain all unallocated blocks in one list
  - also call a *free list*
  - at each allocation request, look for a block of appropriate size
- Strategies
  - First-fit: return first large enough
  - Best-fit: return smallest block large enough
  - Neither is better than other (depends on requests)
- Both cases:
  - if the block found is much larger than the request, split it in 2 and return the other half to free list
- Deallocation
  - add to free list and merge with adjacent regions if possible

Reducing Allocation Time

- Scanning the free list
  - takes linear time in the # of free blocks
  - → maintain separate lists for different (standard) sizes
- Fast allocation
  - find appropriate size (constant time)
  - return the first block (constant time)
- Buddy systems & Fibonacci heaps
  - block sizes powers of 2 or Fibo numbers

Garbage collection

- External fragmentation can not be avoided
  - ‘checkerboard’ the heap
  - we end up with a situation where
    - we have a lot of free space
    - but no blocks large enough
    - → heap must be compacted by moving the allocated blocks
    - complicated because all the references to these blocks must be updated, too!

Explicit and implicit deallocation

- Allocation is always triggered by some program action
- Deallocation can be either
  - explicit
    - Pascal, C
    - simple, efficient, immediate
    - allows ‘hand-tailored’ storage management
  - or implicit
    - Java, C++, functional languages
    - garbage collector starts whenever space gets low
    - complex, time-consuming

Garbage Collection (GC) or not to GC?

- Explicit memory management is more efficient
- But
  - manual deallocation errors are among the most common and costly bugs
  - too fast deallocation → dangling references
  - forgotten deallocation → memory leaks
- Automatic GC is considered an essential feature of modern languages
  - GC algorithms have improved
  - implementations are more complex in general (so adding GC plays not a big role)
  - large applications make benefits of GC greater

Scope

- Scope (of a binding)
  - the (textual) part of the program where the binding is active
- Scope (in general) is a
  - program section of maximal size in which
    - no bindings change or at least
    - no re-declarations are permitted
- Lifetime and scope are not necessarily the same

Elaboration

- Subroutine entrance → new scope *opens*
  - create bindings for new local variables
  - deactivate bindings for global variables that are redeclared
- Subroutine exit → scope closes
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
- *Elaboration*
  - process of creating bindings when entering a new scope
  - also other tasks, for example in Ada:
    - storage allocation
    - starting tasks (processes)
    - propagating exceptions

Scope rules

- *Referencing environment* of a statement
  - the set of active bindings
  - corresponds to a collection of scopes that are examined (in order) to find a binding
- *Scope rules*
  - determine that collection and its order
- Static (lexical) scope rules
  - scope is defined in terms of the physical (lexical) structure of the program
  - typically the most recent (active) binding is chosen
  - we study mostly (and first) these here
- Dynamic scope rules
  - bindings depend on the current state of program execution

Static scope

- Related to program structure
  - Basic: one scope (static & global)
  - Fortran: global & local scopes
    - `COMMON` blocks
      - aim: share global data in separately compiled subroutines
      - typing errors possible
    - `EQUIVALENCE` of variables
      - aim: share (and save) space
      - predecessor of variant/union types
    - lifetime of a local variable?
      - semantically: execution of the subroutine
      - possible to SAVE variables (static allocation)
      - in practice *all* variables *may* behave as if SAVEd → bad programming

Nested program structure

- Subroutines within subroutines within ...
  - → scopes within scopes within ...
  - thing X declared inside a subroutine → thing X is *not* visible outside that subroutine
  - Algol 60, Pascal, Ada, C/C++, ...
- Resolving bindings
  - closest nested scope rule
  - subsequent declarations may *hide* surrounding ones (temporarily)
  - built-in/predefined bindings: special outmost scope
- Example in Figure 3.4

Non-local references

- Nested subroutine may refer to objects declared in other subroutines (surrounding it)
  - example 3.4: P3 can access A1, X and A2
  - how to access these objects (they are in other stack frames)?
  - → need to find the corresponding frame at run-time
- Difficulty
  - deeply nested routine (like P3) can call any visible routine (P1)
  - → caller's scope is *not* (always) the lexically surrounding scope
  - however, that surrounding scope *must* have a stack frame somewhere below in the stack
    - we can get to P3 only by making it visible first
    - P3 gets visible after P1 & P2 have been called

```
procedure P1 (A1 : T1);
var X : real;
    …
    procedure P2 (A2 : T2);
           …
           procedure P3 (A3 : T3);
           …
           begin
                     … (* body of P3 *)
           end;
           …
    begin
           …         (* body of P2 *)
    end;
    …
    procedure P4 (A4 : T4);
           …
           function F1 (A5 : T5) : T6;
           var X : integer;
           …
           begin
              …      (* body of F1 *)
           end;
    begin
           …         (* body of P4 *)
    end;
    …
begin
    …                (* body of P1 *)
end;
```

Accessing non-local stack objects

- Parent frame
  - most recent invocation of the lexically surrounding subroutine
- Augment stack frame with *static link*
  - pointer to parent frame
  - outermost frame: parent = nil
  - links form a *static chain* through all scopes
- Accessing
  - routine at nesting depth k refers to an object at depth j
  - follow k - j static links (k - j is known at compile-time)
  - use offset in that ancestor frame as usual
- Figure 3.5

Static Chains

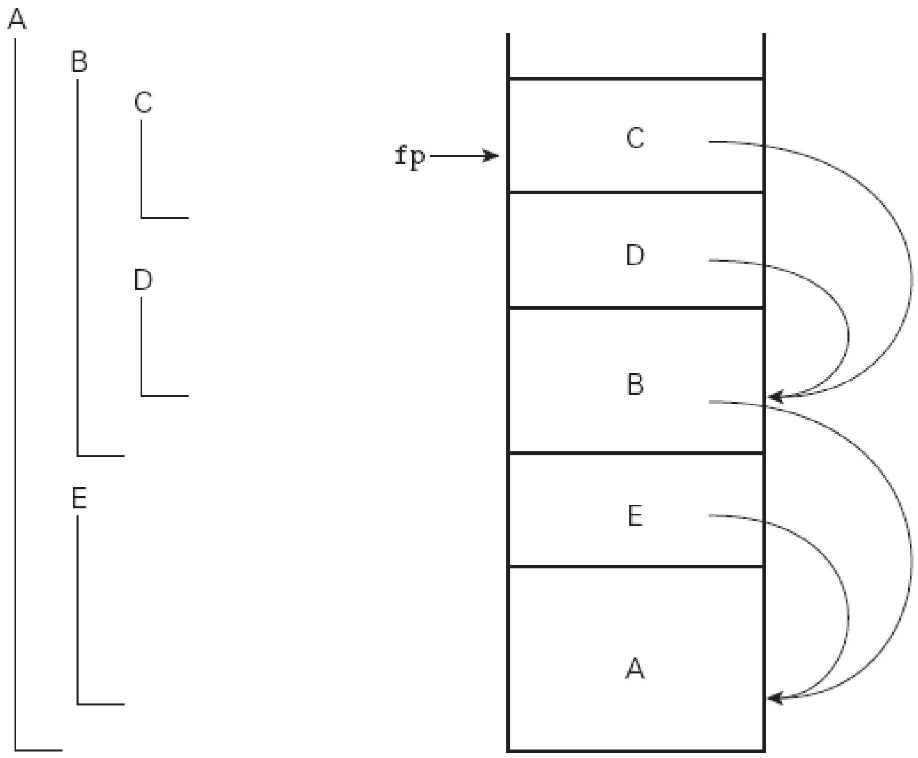


Figure 3.5

Holes in scopes

- Name-object binding N-O1
  - hidden by a nested declaration N-O2
  - has a *hole* in it's scope (for the lifetime of that nested declaration)
  - object O1 is inaccessible via N
- Scope resolution operators
  - allow programmer to explicitly use 'hidden' bindings
  - also call a *qualifiers*
  - Ada: **`My_Proc.X`** (**`X`** declared in **`My_Proc`**)
  - C++: **`::X`** (global X)

Scopes without subroutines

- Variable definitions in block statements
  - either at the beginning of the block or
    - C, Ada
  - anywhere where a statement may appear
    - C++, Java
    - scope extends to the end of the current block
  - space allocated from the stack frame
    - no extra runtime operations needed
    - space saved by letting allocations overlap each other

Re-declaring bindings

- Change bindings ‘on the fly’
  - e.g. to fix bugs (rapid prototyping)
  - interactive languages
- New meaning replaces the old one immediately everywhere
  - implemented using some search structure (name → meaning)
- Problems with ‘half-compiled’ languages (ML)
  - old (compiled) bindings may be preserved
  - in subroutines that are already elaborated (using the old binding)

Modules

- Great tool to divide programming effort
  - *information hiding*
    - details are visible only to parts that really need them
  - reduces the ‘cognitive load’ of programmers
    - minimize the amount of information required to understand any part of the system
  - changes & updates localized within single modules
- Other benefits
  - reduces name clashes
  - data integrity: only routines inside a module update certain object
  - errors are localized

Information hiding using subroutines?

- Hiding is limited to objects defined inside a routine
  - lifetime = execution of the routine
- Partial solution: use static local objects
  - C: **`static`**, Algol: **`own`**, ...
  - Example: Figure 3.6
  - ‘subroutines with memory’
  - ‘single-routine data abstractions’

Module: multiple-routine abstraction

- Combine and hide several routines & data structures
  - e.g. stack type, push and pop operations
  - Ada: package, Clu: cluster, Modula-2: module
  - objects inside a module are
    - visible to each other
    - *not* visible to the outside *unless* explicitly *exported*
  - objects outside a module are
    - *not* visible to the inside *unless* explicitly *imported*
- Example: Figure 3.7

Modules and bindings

- Bindings made inside a module
  - are inactive outside of it
  - but *not* destroyed
  - module-level objects have ‘same lifetime they would have without the enclosing module’
    - same as the scope they appear in
- Restrictions on export declarations
  - possible in many module-based languages
  - variables exported read-only
  - types exported as opaque (Modula-2)
    - only certain primitive operations allowed

Headers and bodies

- Modules are often divided into
  - header/declaration part
    - definitions for users of the module
    - the *public* interface of the module
    - may also contain private information (for compilation)
  - body/implementation part
    - definitions for the implementation of the module
- Header and body parts can be compiled separately
  - especially header can be compiled even if body does not exist (yet)
  - and so can the users of the header
  - total recompilation unnecessary if only some modules are updated

Open and closed scopes

- Open scope
  - no imports required (scope rules apply)
  - Ada packages, nested subroutines in most Algol family languages
- Closed scope
  - all names must be explicitly imported
  - Modula-2 modules, Euclid subroutines
  - Clu: nonlocal references not possible
  - import lists
    - document the program (nonlocal references are part of the interface)
    - help the compiler to detect *aliasing* (Euclid, Turing)

Aliasing

- Alias
  - extra name for something that already has a name (in the current scope)
  - we may have several (e.g., 陳仁暉, 阿暉, 陳老師, etc)
- How are they created?
  - Fortran: explicit declarations (equivalence)
  - variant/union structures
  - languages using pointers
  - reference parameters
- They are considered bad because
  - they create confusion and hard-to-track errors (Fig. 3.8)
  - compilers can optimize much better if they know there’s no aliasing

Type manager modules

- Modules support ‘naturally’ only the creation of one single instance of a given abstraction
  - Figure 3.7 creates only one stack
  - → replicate code?
- Alternative organization
  - module is a *manager* for the instances of the type it implements (Fig. 3.9)
  - additional routines to create/destroy instances
  - additional parameter to each operation (the object in question)
  - Clu: every module is a manager of some type

Module types

- Each module creates a new type
  - possible to declare (any number of) variables of that type
  - Euclid, Simula, Clu
  - Automatic
    - initialization code and
    - finalization code (e.g. to return objects to heap)
- Types and their operations are tightly bound to each other
  - operations ‘belong’ to objects of the module type
  - conceptually
    - type approach has a separate push for every stack
    - manager approach has one parameterized push for all stacks
  - same implementation in practice

Classes

- Object-oriented programming
- Module types augmented with *inheritance mechanism*
  - possible to define new modules ‘on top’ of existing ones (refinements, extensions)
  - objects inherit operations of other objects (no need to rewrite the code)

Module types and scopes

- Note: applies also to classes if we forget inheritance
- Every instance A of a module type has
  - a separate copy of the module variables
  - which are visible when executing one of A’s operations
- Indirect visibility (within same type)
  - the *instance variables* of B may be visible
  - to another instance A of the same type
  - if B is passed as a parameter of A’s operation
  - → binary operations of C++
  - opinions vary whether this is a good thing or not

Dynamic scope...

- Name-object binding decided at run-time
  - usually the last active declaration
  - thus, derived from the order in which subroutines are called (Fig. 3.11)
  - flow of control is unpredictable → compilation impossible
- Example languages
  - early functional languages (Lisp)
  - Perl (v5.0 gives also static scope)
  - environment variables in command shells
- Static scope rules require that the reference resolve to the closest lexically enclosing declaration.

...Dynamic scope

- Dynamic scope → dynamic semantics
  - type checking in expressions and parameter passing must be deferred to run-time
- Simple implementation
  - maintain declarations in a stack
  - search stack top → bottom to find bindings
  - push/pop bindings when entering/leaving subroutines
  - quite slow

Dynamic scope is a bad idea?

- Cons
  - High run-time costs
  - Non-local references are ‘unpredictable’
  - Dynamic programs are hard to understand
- Pros
  - Easy to customize subroutines ‘on the fly’
  - book example
    - print integers in different bases
    - base = non-local (dynamic) reference

Simulating dynamic scope

- Workaround 1
  - make separate routines for separate cases
  - default parameters (Ada) → one interface
  - overloading (C++) → same name
  - but: calls made under the emulated ‘dynamic scope’ do not ‘inherit’ the mimicked non-local binding
- Workaround 2
  - use a global/static variable instead of a non-local reference
  - store/restore before/after use of the routine
- Jenhui says
  - pass all stuff in parameters
  - forget that non-local references even exist

Binding of referencing environments

- Note: we skip section 3.3.3 & 3.3.4
- WHEN scope rules should be applied?
  - usually no problem (just apply scope rules)
  - problematic case: references to subroutines
    - e.g. function parameters
    - these may have non-local references, too!
  - when the reference was created?
  - when the referred routine is (finally) used?
- In other words
  - WHAT is the referencing environment of a subroutine reference?

Shallow & deep binding

- Consider example of Fig. 3.16 (dynamic scoping)
- **`print_routine`**
  - should create its environment just before it's used
  - otherwise the 'format trick' would not work
  - this late binding is called *shallow binding*
  - default in dynamically scoped languages
- **`older_than`**
  - is designed to use the global **`threshold`** variable
  - i.e. it is meant to be used in that one and only environment
  - referencing environment
    - should be bound when **`older_than`** is passed as a parameter
    - and used when it is finally called
  - this early binding is called *deep binding*

Implementing deep binding

- Subroutine *closure* bundles together
  - pointer to subroutine code and it’s
  - referencing environment
- Dynamic scoping
  - environment implemented as a binding stack
    - book calls this an *association list* (section 3.3.4)
    - current top of stack defines the environment
  - when a routine is passed as a parameter
    - save current top of the stack in the closure & pass closure
  - when the referenced routine is called
    - use saved pointer as the top of stack
    - if other functions are called, grow another ‘top’ for the stack from this point (list-implemented stack)

Deep binding & static scope

- Deep binding is default in static scope
  - shallow binding does not make much sense
- Does the binding time matter at all?
  - generally not
    - name → lexical nesting
    - nesting does no change
  - recursion!
    - we must find the correct *instance*, too
    - closure must capture the *current* instance of every visible object when it is created
    - this saved closure is then used when routines are used
    - example in Fig. 3.17

Some notes

- Binding rules matter (with static scoping) *only*
  - when referencing objects that are *not* local *neither* local
  - irrelevant in
    - C: no nested structure
    - Modula-2: only top-level routines can be passed as parameters
    - and in all languages that do not permit passing subroutines as arguments
  - Jenhui: good programmer doesn’t make such programs anyway

Implementing deep binding

- Static links define referencing environments
  - pass: create closure with current static link
  - call: use the saved static link (instead of creating a new one) when creating the frame record
    - static chain is now the same as in the time the parameter was passed

Classes of objects (values)

- First-class objects can be
  - passed as parameters
  - returned from a subroutine
  - assigned to variables
  - e.g. integers in most languages
- Second-class objects
  - can only be passed as parameters
  - e.g. arrays in C/C++
- Third-class objects
  - can not even be passed as a parameter
  - e.g. jump labels (in most languages that have a goto-statement)

Subroutines & classes

- First-class
  - functional languages
    - note: these can even *create* new subroutines
  - Modula-2 & -3, Ada 95, C, C++
    - language-specific restrictions on use
- Second-class
  - almost all other languages
  - Ada 83: third-class

Problem with first-class subroutines…

- Reference to a subroutine
  - may live *longer* than the scope that created it
  - → referencing environment no longer exists when routine is called
- Functional languages
  - *unlimited extent* of local objects
  - frames are allocated from the heap (not stack)
  - garbage collected when no references remain

...Problem with first-class subroutines

- Imperative languages *want* to use the stack
  - *limited extent* of local objects
  - frames are deleted from the stack when execution leaves the subroutine
    - → dangling references if 1st class subroutines
- Algol-family languages have different workarounds
  - Modula-2: only top-level subroutines can be referenced to
  - Modula-3
    - only top-level subroutines are 1st class, others are 2nd class
  - Ada 95
    - a nested routine can be returned (by a function) only to a scope that contains that routine
    - → referencing environment will always be alive
- C/C++: no problem (because they have no nested scopes)

Overloading

- Aliasing: multiple names for same object
- Overloading: one name for several objects
  - in the same scope
  - semantic rules: context of the name must give enough information to resolve the binding
  - most programming languages support at least some overloading (arithmetic operations)

Overloading of ...

- Enumeration constants
  - Ada example in Figure 3.18
    - dec & oct overloaded
    - print has not sufficient context → explicit qualification required
  - Modula-3: each occurrence must be qualified
- Subroutines
  - Ada, C++
    - arbitrary number as long as parameter types/number are different
    - also arithmetic operators (syntactic sugar)
  - Figure 3.19

Overloading is NOT coercion

- Coercion
    - process in which the compiler *automatically*
        - converts an object X:T1 to another type T2
        - when X is used in a context where T2 is expected
- In overloading
    - separate functions are selected by the compiler for different uses
- In coercion
    - there is only one function
    - compiler makes the necessary type transformations

Overloading is NOT polymorphism

- Polymorphism
  - polymorphic objects may represent objects of more than one type
  - subroutines can manipulate the polymorphic parameters without any conversions
    - either all objects have some common characteristics (and only these are used)
    - or objects contain other information so the subroutine can customize itself appropriately

Examples of polymorphism

- abs(x) for any type that supports
  - comparison ‘> 0’ and
  - negation
- counting the length of a list (of any type)
  - only succ & empty –tests required
- mergesort
  - comparison, succ, empty, cons
- conformant array parameters
  - very limited form of polymorphism (Pascal, Ada)

Overloading is NOT generics

- Generic subroutines (or modules)
  - parameterized *templates* that can be *instantiated* to create *concrete* subroutines
    - early C++ versions used cpp to do this
  - Ada example in Fig. 3.20
- Generics is not polymorphism
  - polymorphic routine is a *single* object capable of accepting multiple types
    - compiled to a single body of code
  - generic routines are instantiated to create an own concrete routine for each different use
    - compiled to several copies of the code
    - Ada allows these instance names to be overloaded
    - C++ requires them to do so

Naming-related pitfalls...

- Redefinition of function name inside the function
  - recursion impossible
  - Pascal: function name is used to define the return value
    - → strange problems
    - most current languages use return-statement or some special pseudo-variable for return values
- Scope of a name
  - *entire* block it is declared in (Pascal)
    - names must be declared before they are used
    - strange consequences when names refer to each other
    - do we use ‘external’ or ‘internal’ name?
  - or from the declaration to the end of the block? (Ada)
  - or either to the end of the block or next re-definition (ML)

…Naming-related pitfalls

- Recursive data types
  - need to reference to the not-yet-defined type
  - Pascal: pointers are an exception to the general ‘declare before use’ rule
    - pointer declaration makes a *forward reference*
  - C, C++, Ada
    - forward references are forbidden but *incomplete* type definitions are allowed
- Mutually recursive subroutines
  - need to reference to the not-yet-defined subroutine
  - Pascal: forward declarations
  - Modula-3: order of declarations does not matter
  - Java, C++
    - variables ‘declared before used’
    - classes can be used before they are (totally) declared
    - order of routines does not matter (inside a class)