Naming, scoping, binding, etc.

Instructor: Dr. B. Cheng
Fall 2004

Imperative Programming

- The central feature of imperative languages are variables
- Variables are abstractions for memory cells in a Von Neumann architecture computer
- Attributes of variables
  - Name, Type, Address, Value, ...
- Other important concepts
  - Binding and Binding times
  - Strong typing
  - Type compatibility rules
  - Scoping rules

Preliminaries

- **Name**: representation for something else
  - E.g.: identifiers, some symbols
- **Binding**: association between two things;
  - Name and the thing that it names
- **Scope of binding**: part of (textual) program that binding is active
- **Binding time**: point at which binding created
  - Generally: point at which any implementation decision is made.
Names (Identifiers)

- Names are not only associated with variables
  - Also associated with labels, subprograms, formal parameters, and other program constructs

- Design issues for names:
  - Maximum length?
  - Are connector characters allowed? ("_")
  - Are names case sensitive?
  - Are the special words: reserved words or keywords?

Names

- Length
  - If too short, they will not be connotative
  - Language examples:
    - FORTRAN I: maximum 6
    - COBOL: maximum 30
    - FORTRAN 90 and ANSI C (1989): maximum 31
      - ANSI C (1989): no length limitation, but only first 31 chars significant
      - Ada and Java: no limit, and all are significant
      - C++: no limit, but implementors often impose one
  
- Connector characters
  - C, C++, and Perl allow "_" character in identifier names
  - Fortran 77 allows spaces in identifier names:
    - Sum Of Salaries and SumOfSalaries refer to the same identifier

Names

- Case sensitivity
  - C, C++, and Java names are case sensitive
  - Disadvantages:
    - Readability (names that look alike are different)
    - Writability (must remember exact spelling)
    - Java: predefined names are mixed case (e.g. IndexOutOfBoundsException)
  
- Earlier versions of Fortran use only uppercase letters for names (because the card punches had only uppercase letters!)
**Names**

- Special words
  - Make program more readable by naming actions to be performed and to separate syntactic entities of programs
  - A keyword is a word that is special only in certain contexts
    - Disadvantage: poor readability
      - e.g., Fortran: Real Integer ? Integer is a Real variable
      - Integer Real ? Real is an Integer variable
  - A reserved word is a special word that cannot be used as a user-defined name

**Variables**

- A variable is an abstraction of a memory cell
- Variables can be characterized by several attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

- Address
  - The memory address with which it is associated
  - A variable may have different addresses at different times during execution – e.g., local variables in subprograms
  - A variable may have different addresses at different places in a program – e.g., variable allocated from the runtime stack

- Aliases
  - If two variable names can be used to access the same memory location
  - Harmful to readability (program readers must remember all of them)
  - How aliases can be created:
    - Pointers, reference variables, Pascal variant records, C and C++ unions, and FORTRAN EQUIVALENCE
Variables

Type
- determines the range of values of variables and the set of operations that are defined for values of that type
- int type in Java specifies a value range of –2,147,483,648 to 2,147,483,647 and arithmetic operations for addition, subtraction, division, etc
- in the case of floating point, type also determines the precision (single or double)

Value
- the contents of the memory cells with which the variable is associated
- Abstract memory cell - the physical cell or collection of cells associated with a variable
- The l-value of a variable is its address
- The r-value of a variable is its value

Binding
- A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Binding time is the time at which a binding takes place
Binding Times

Possible binding times:
1. Language design time
   - e.g., bind operator symbols to operations
2. Language implementation time
   - e.g., bind floating point type to a representation
3. Compile time
   - e.g., bind a variable to a type in C or Java
4. Load time
   - e.g., bind a FORTRAN 77 variable to a memory cell
     (or a C static variable)
5. Runtime
   - e.g., bind a nonstatic local variable to a memory cell

Static vs Dynamic Binding

A binding is static if it first occurs before run time and remains unchanged throughout program execution.

A binding is dynamic if it first occurs during execution or can change during execution of the program.

Type Binding

Type Bindings
- How is a type specified?
- When does the binding take place?
Static Type Binding

- May be specified through explicit or an implicit declaration
- **Explicit declaration** is a program statement used for declaring the types of variables
  
  - Ex: `int a`
- **Implicit declaration** is a default mechanism for specifying types of variables (first appearance of variable in program)
  
  - FORTRAN, PL/I, and BASIC provide implicit declarations
  
  - Ex: Fortran: vars starting with I-N are integers; others are reals
- **Advantage:** writability (fewer lines of code to write)
- **Disadvantage:** reliability
  
  - Implicit declaration prevents compilation process from detecting typographical and programmer errors
  
  - In FORTRAN, variables that are accidentally left undeclared are given default types and unexpected attributes
  
  - Less trouble with Perl: uses names beginning with special char ($ for scalar; @ for arrays, % for hash structure)

Dynamic Type Binding

- Dynamic Type Binding (APL, JavaScript, SNOBOL)
  
  - Type is not specified by a declaration statement, nor can it be determined by the spelling of its name
  
  - Type is specified through an assignment statement e.g. in JavaScript:
    
    - `list = [2, 4.33, 6, 8];` (1-dim array)
    - `list = 17.3;` (scalar) % using same var name
  
  - **Advantage:** flexibility (generic program units)
  
  - Ex: program to sort data; at run time, can sort integers, reals, characters, etc.
  
  - **Disadvantages:**
    
    - High cost (dynamic type checking; can only be implemented using interpreters)
    - Type error detection by the compiler is difficult

Dynamic Type Binding (2)

- Type Inferencing (ML, Miranda, and Haskell)
  
  - Types are determined from the context of the reference
  
  - E.g., ML function:
    
    ```ml
    fun square (x) = x * x;
    ```
    
    - because this is an arithmetic operator, the function is assumed to be numeric, which by default is int type
  
  - If we want real return values:
    
    ```ml
    fun square (x) : real = x * x;
    ```
Storage Binding

- Storage Bindings
  - Allocation
    - getting a cell from some pool of available memory cells
  - Deallocation
    - putting a cell back into the pool of memory cells

- Lifetime of a variable is the time during which it is bound to a particular memory cell
- 4 types of variables (based on lifetime of storage binding)
  - Static
  - Stack-dynamic
  - Explicit heap-dynamic
  - Implicit heap-dynamic

Storage Binding Lifetime

- Static
  - bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
  - e.g. all FORTRAN 77 variables, C static variables
  - Advantages: efficiency (direct addressing), support for history-sensitive subprogram
  - Disadvantage: lack of flexibility (no recursion)

Stack-dynamic binding lifetime

- Storage bindings are created for variables when their declaration statements are encountered during run time and binding takes place (i.e., elaboration).
  - but whose types are statically bound
  - If scalar, all attributes except address are statically bound
  - e.g. local variables in C subprograms and Java methods
  - Advantage: allows recursion; conserves storage
  - Disadvantages:
    - Overhead of allocation and deallocation
    - Subprograms cannot be history sensitive
    - Inefficient references (indirect addressing)
Explicit heap-dynamic binding lifetime

- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution

```c
int *intnode;

intnode = new int; /* allocates an int cell */

intnode points to int node;
```

- Variables are nameless and referenced only through pointers or references
  - e.g. dynamic objects in C++ (via new and delete), all objects in Java
- Advantage provides for dynamic storage management
  - Useful for dynamic structures, such as trees and lists that grow/shrink during execution
- Disadvantage: inefficient and unreliable

Implicit heap-dynamic binding lifetime

- Allocation and deallocation caused by assignment statements
  - e.g. all variables in APL; all strings and arrays in Perl and JavaScript
- Advantage: flexibility
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Type Checking

- For this discussion, generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either
  - legal for the operator, or
  - is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type -- coercion
- A type error is the application of an operator to an operand of an inappropriate type
  - If all type bindings are static, nearly all type checking can be static
  - If type bindings are dynamic, type checking must be dynamic
Strong Typing

- A strongly typed language is one in which each name in a program has a single type associated with it, and the type is known at compile time.
- A programming language is strongly typed if type errors are always detected.
- Advantage: allows detection of misused variables that result in type errors.
- FORTRAN 77 is not: use of EQUIVALENCE between variables of different types allows a variable to refer to a value of a different type.
- Pascal is not: variant records.
- C and C++ are not: unions are not type checked.

Coercion rules strongly affect strong typing.

- Expressions are strongly typed in Java.
  - However, an arithmetic operator with one floating point operand and an integer operand is legal.
  - Value of integer is coerced to floating point.

Type Compatibility

- Type compatibility by name:
  - Two variables have compatible types if they are in either the same declaration or in declarations that use the same type name.

```plaintext
type Indextype is 1..100;
count: Integer;
index: Indextype;  /* count and index are not type compatible
```

- Easy to implement but highly restrictive:
- Subranges of integer types are not compatible with integer types.
- Formal parameters must be the same type as their corresponding actual parameters (Pascal).
Type Compatibility

Type compatibility by structure
- two variables have compatible types if their types have identical structures
- More flexible, but harder to implement

STOP

Static vs Dynamic Binding

- Static Binding: bindings occurring BEFORE run time
- Dynamic Binding: bindings AFTER run time
- Early binding: more efficient
  - Compiled languages: more early binding
- Later binding: greater flexibility
  - Interpreted languages: more late binding
**Scope Rules**

- Scope rules control bindings
- Naming of data: key ability with programming languages
  - Use symbolic identifiers rather than addresses to refer to data
- Not all data is named:
  - Dynamic storage in C and Pascal referenced by pointers, not names

**Items of concern**

- creation of objects
- creation of bindings
- references to variables (which use bindings)
- (temporary) deactivation (hiding) of bindings
- reactivation of bindings
- destruction of bindings
- destruction of objects

**Note:**
- If object outlives binding it's garbage
- If binding outlives object it's a dangling reference

**Scope**

- Binding lifetime: period of time from creation to destruction
- Scope: Textual region of program in which binding is active
  - Secondary defn: program section of maximal size in which no bindings change
- Ex: Subroutines:
  - Open a new scope on subroutine entry
  - Create bindings for new local variables
  - Deactivate bindings for global variables that are redeclared
  - Make references to variables
  - Upon exit: destroy bindings for local vars
  - Reactivate bindings for global vars that were deactivated
Scope Rules

- Referencing Environment (of stmt or expr):
  - Set of active bindings
  - Corresponds to a collection of scopes that are examined (in order) to find a binding
  - Scope rules: determine the collection and order
  - Static (lexical) scope rules:
    - a scope is defined in terms of the physical (lexical) structure of the program.
    - Can be handled by compiler
    - All bindings for identifiers resolved by examining program
    - Chose most recent, active binding made at compile time
  - Ex: C and Pascal (and most compiled languages)

Evolution of data abstraction facilities

- none: Fortran, Basic
- subroutine nesting: Algol 60, Pascal, many others
- own (static) variables: Algol 68, Fortran ("save"), C, others
- module as manager: Simula (predates Modula; clearly before its time), Euclid
- module as type: Simula, Smalltalk, C++, Eiffel, Java, others
- classes, with inheritance: Simula, Smalltalk, C++, Eiffel, Java, others
- Modern OO languages:
  - Reunify encapsulation (information hiding) of module languages with
  - Abstraction (inheritance and dynamic type binding) of Smalltalk
  - Both threads have roots in Simula

Storage Management

- Static allocation: for code, globals, "own" variables, explicit constants (including strings, sets, other aggregates):
  - Scalars may be stored in the instructions themselves
- Central stack for
  - parameters
  - local variables
  - temporaries
  - bookkeeping information
- Why a stack?
  - allocate space for recursive routines (no need in Fortran)
  - reuse space (useful in any language)
- Heap for
  - dynamic allocation
Maintaining the Run-time Stack

- Contents of a stack frame
  - bookkeeping: return PC (dynamic link), saved registers, line number, static link, etc.
  - arguments and returns
  - local variables
  - Temporaries

sp: points to unused stack
fp: known locn within frame
(activation record)

Maintaining the Run-time Stack

- Maintenance of stack is responsibility of "calling sequence"
  - and subroutine "prolog" and "epilog".
  - space is saved by putting as much in the prolog and epilog as possible
  - time "may" be saved by putting stuff in the caller instead, or
  - by combining what's known in both places (interprocedural optimization)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Access to non-local variables

- Static links:
  - Each frame points to the frame of the (correct instance of) the routine inside which it was declared.
  - In the absence of formal subroutines, "correct" means closest to the top of the stack.
  - Access a variable in a scope k levels out by following k static links and then using the known offset within the frame thus found.
Dynamic Scope Rules

- Bindings depend on current state of execution.
- Cannot always be resolved by examining the program (textual, static structure)
- Dependent on calling sequences
- To resolve a reference:
  - Use most recent, active binding made at run time
- Dynamic scope rules used in interpreted languages
  - Ex: early LISP dialects
- Such languages do not typically have type checking at compile time because type determination is NOT always possible with dynamic scope rules

Static vs Dynamic Scope Rules: Example

1. program scopes (input, output);
2. var a : integer;
3. procedure first;
4. begin a := 1; end;
5. procedure second;
6. var a : integer;
7. begin first; end;
8. begin
9. a := 2; second; write(a);
10. end.

- Static scope rules:
  - Program prints "1"
- Dynamic scope rules:
  - Program prints "2"

Why difference?
- Static:
  - Reference resolved to most recent, compile-time binding,
  - Global variable "a" gets printed (last modified in procedure "first")
- Dynamic:
  - Choose most recent, active binding at run time
  - Create binding for "a" when enter main program
  - Create another binding for "a" when enter procedure "second"
  - Write global variable "a" because the "a" local to procedure second is no longer active.

Accessing Variables with Dynamic Scope

- (1) Keep a stack ("association list") of all active variables.
- When finding a variable, hunt down from top of stack.
- Equivalent to searching activation records on the dynamic chain.
- Slow access, but fast calls
- Ex: Lisp: deep binding
Accessing Variables with Dynamic Scope

(2) Keep a central table with one slot for every variable name.
- If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time.
- Otherwise, need a hash function or something to do lookup.
- Every subroutine changes the table entries for its locals at entry and exit.
- Slow calls, but fast access
  - Ex: Lisp: shallow binding

Binding Rules

- **Referencing Environment** (of a stmt):
  - Set of active bindings
  - Corresponds to a collection of scopes that are examined (in order) to find a binding
- **Scope rules**: determine collection and its order
- **Binding Rules**:
  - Determine which instance of a scope should be used to resolve references
  - When calling a procedure passed as a parameter
  - Govern the binding of reference environments to formal parameters

- **Shallow binding**: nonlocal referencing environment of a procedure instance is the referencing environment in force at the time it is invoked.
  - Ex: original LISP works this way by default
- **Deep binding**: nonlocal referencing environment of a procedure instance is the referencing environment in force at the time the procedure’s declaration is elaborated.
  - For procedures passed as parameters, environment is same as it would be extant if the procedure were actually called at the point where it was passed as an argument.
  - When the procedure is passed as an argument, this referencing environment is passed as well.
  - When the procedure is eventually invoked (by calling it using the corresponding formal parameter), this saved referencing environment is restored.
  - Ex: Procedures in Algol and Pascal work this way
Binding Rules – a few notes

Note 1: see difference between shallow and deep binding when:
- Pass procedures as parameters
- Return procedures from functions
- Store references to procedures in variables
  - Irrelevant to languages such as PL/0 – no formal subroutines

Note 2: No language with static (lexical) scope rules has shallow binding
- Some languages with dynamic scope rules – only shallow binding (e.g., SNOBOL)
- Others (e.g., early LISP) offer both, where default is shallow binding.
  - Funarg specify deep binding

Note 3: Binding rules have no relevance to (lexical) local/global references
- since all references are always bound to currently executing instance and only one instance of main program contains global variables.
- Binding irrelevant to languages that:
  - Lack nested subroutines (e.g., C)
  - Only allow outermost subroutines to be passed as parameters (e.g., Modula-2)

Binding rules -- Example

Simple example (assume dynamic scope):

```pascal
Program Simple;
    procedure A (p1 : procedure; i : integer);
    procedure B;
        begin B
            writeln(i);
        end;
    begin A
        if i = 1 then A(B,2)
        else p1;
    end A;
begin main
    A(C,1);
end main.
```

Two activations of A when B is finally called
- The deep version: is the A that is active when B was passed as a parameter
- Under deep binding: program prints a 1
- Under shallow binding, it prints a 2.

Naming: Overloading

- Overloading: using the same name for multiple things
- Some overloading happens in almost all languages:
  - Ex: integer + v.real; read/write in Pascal; function return in Pascal
- Some languages make heavy use of overloading (e.g., Ada, C++)
  - Ex:
    1. overload norm;
    2. int norm (int a) { return a > 0? A: -a; }
    3. complex norm (complex c) { // ...}
Naming: Polymorphism

- **Ad hoc polymorphism**: overloading of names
- **Subtype polymorphism** (in OO languages):
  - Allows code to do the "right" thing to parameters of different types in the same type hierarchy
  - By calling the virtual function appropriate to the concrete type of actual parameter.
  - Ex: shape hierarchy and draw function.

Naming: Parametric Polymorphism

- **Parametric Polymorphism**
  - **Explicit (generics)**: specify parameter(s) (usually type(s)) when declare or use the generic.
    - Templates in C++ are an example:
      ```
      typedef set<string>::const_iterator string_handle_t;
      set<string> string_map;
      ...
      pair<string_handle_t, bool> p = string_map.insert(ident);
      // *pair.first is the string we inserted
      // pair.second is true iff it wasn't there before
      ```
  - Implemented via macro expansion in C++:
  - Built-in in Standard C++. (BTW: be warned when using nested templates)
    - In C++: pair<foo, bar<glarch>> won't work,
      - because >> is a single token.
    - Have to say: pair<foo, bar<glarch>>. Yuck!

Naming: Implicit (True) Parametric Polymorphism

- No need to specify type(s) for which code works;
- Language implementation determines automatically – won’t allow operations on objects that don’t support them.
- Functional languages support true parametric polymorphism:
  - In run-time system (e.g., LISP and descendants)
  - In compiler (e.g., ML and its descendants)
Naming: Aliasing

- Aliasing: more than one name for the same thing.
- Purposes:
  - Space saving: modern data allocation methods are better
  - Multiple representations: unions are better
  - Linked data structures: legit
- Aliases also arise in parameter passing, as an unintended (bad?) side effect.

Gotchas in language design

- Fortran spacing and do-loop structure (use of ',')
- If-then-else nesting in Pascal
- Separately compiled files in C provide a "poor person's modules"
- Rules for how variables work with separate compilation are messy
- Language has been "rigged" to match behavior of the linker
  - 'Static' on a function or variable OUTSIDE a function means it is usable only in the current source file
  - Different from the 'static' variables inside a function!
  - 'Extern' on a variable or function means that it is declared in another source file

Gotchas in language design (2)

- Separately compiled files (in C), cont'd
  - Function headers without bodies are 'extern' by default
  - 'Extern' declarations are interpreted as forward declarations if a later declaration overrides them
  - Variables/functions (with bodies) that do not have 'static' or 'extern' are either 'global' or 'common' (a Fortran term).
    - Variables that are given initial values are 'global', otherwise considered 'common'.
    - Matching 'common' declarations in different files refer to the same variable
    - They also refer to the same variable as a matching 'global' declaration
- Above are examples of poor language design
Morals of the language design story

- Language features can be surprisingly subtle
- Designing languages to make it easier for the compiler writer CAN be a GOOD THING
- Most of the languages that are easy to understand are easy to compile and vice versa
- A language that is easy to compile often leads to easier to understand language
- More good compilers on more machines (e.g., compare Pascal and Ada)
- Better (faster) code
- Fewer compiler bugs
- Smaller, cheaper, faster compilers
- Better diagnostics

Some questionable features

- goto statements
- the original C type system and parameter-passing modes
- 'cut' in Prolog
- the lack of ranges on case statement arms in Pascal