CSCI: 4500/6500 Programming Languages

Names, Scopes (review) and *Binding*

Reflecting on the Concepts

Name, Binding and Scope

- A name is exactly what you think it is
  - » Most names are identifiers
  - » symbols (like '+') can also be names
- A binding is an association between two things, such as a name and the thing it names
  - » Example: the association of
    - – values with identifiers
- The scope of a binding is the part of the program (textually) in which the binding is active

Binding Time

- When the "binding" is created or, more generally, the point at which any implementation decision is made
  - » language design time
  - » language implementation time
  - » program writing time
  - » compile time
  - » link time
  - » load time
  - » run time

Bind Time: Language System View

- language design time
  - » bind operator symbols (e.g., * + …) to operations (multiplication, addition, …)
  - » Set of primitive types
- language implementation time
  - » bind data type, such as int in C to the range of possible values (determined by number of bits and affect the precision)
    - – Considerations: arithmetic overflow, precision of fundamental type, coupling of I/O to the OS' notion of files

Bind Time: User View

- program writing time
  - » Programmers choose algorithms, data structures and names.
- compile time
  - » plan for data layout (bind a variable to a data type in Java or C)
- link time
  - » layout of whole program in memory (names of separate modules (libraries) are finalized.
- load time
  - » choice of physical addresses (e.g. static variables in C are bound to memory cells at load time)

Bind Time: User View

- run time (dynamically)
  - » value/variable bindings, sizes of strings
  - » subsumes
    - – program start-up time
    - – module entry time
    - – elaboration time (point a which a declaration is first "seen")
    - – procedure entry time
    - – block entry time
    - – statement execution time

Static and Dynamic

- Typically means before or at run time?
  - » 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.

Binding Time Summary

- 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 (run time)

Lifetime and Storage Management

Key Events: Need to distinguish between names and the object to which they refer

- creation of objects
- destruction of objects

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

Storage Binding and Lifetime

- Binding Lifetime: time between creation and destruction of a name-to-object binding
- Object Lifetime: time between creation and destruction of an object

Implications:

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

Storage Allocation Mechanisms

- Static: absolute address that is retained throughout program execution
- Stack: storage bindings are created when declaration statements are elaborated (e.g., subroutine calls and returns are allocated in last in first-out order).
- Heap: created and destructed by explicit directives (e.g. new and delete in Java creates objects)

Static Allocation

- Example: Code, globals, static variables, explicit constants, scalars
- Advantages: efficiency (direct addressing), history-sensitive subprogram support (static variables retain values between calls of subroutines).
- Disadvantage: lack of flexibility (does not support recursion)

Stack Allocation

- Storage bindings are created for variables when their declaration statements are elaborated
- Central stack for parameters, local variables and temporaries
  - » Easy to allocate space for locals on stack: fixed offset from the stack pointer or frame pointer at compile time
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - » Overhead of allocation and deallocation
  - » Subprograms cannot be history sensitive
  - » Inefficient references (indirect addressing)

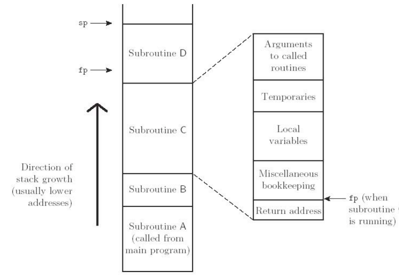


Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (sp) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (fp) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Heap Allocation

- heap-dynamic - Allocated and deallocated by explicit directives (malloc, new), specified by the programmer, which take effect during execution
- 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
- Disadvantage: inefficient (instead of static) and *unreliable*

Heap Management

- Speed and space tradeoff:
  - » Space fragmentations
    - – Internal - space left in internal blocks
    - – External - unused space is scattered through heap but not one single piece is large enough to satisfy a single request

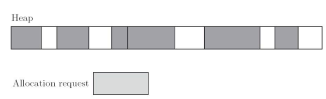


Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

  - » Speed - first fit, best fit, buddy systems

Scope Rules

- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted (see below)
- 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) – 'shadowed'
  - » make references to variables
- The scope rules (static, dynamic) of a language determine how references to names are associated with variables

Static or Lexical Scope

- Here, scope is defined in terms of the physical (lexical) structure of the program
  - » The determination of scopes can be made by the compiler (bindings are resolved by examining the program text)
  - » Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent
  - » Variables can be hidden from a unit by having a "closer" variable with the same name
    - – Ada and C++ allow access to these (e.g. class_name:: name)
  - » Most compiled languages, C and Pascal included, employ static scope rules

Creating Static Scopes

- Static scope rules:
  - » Most closest nested rule used in blocks

    C and C++:
```
for (...)
{
  int index;
 ...
}
```
  - » To resolve a reference, we examine the local scope and statically enclosing scopes until a binding is found

- Nested subroutine scope rules (later in chapter 8)
- Access to nonlocal objects

Static Scope

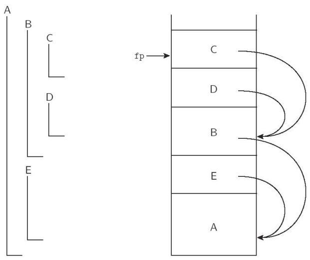


Figure 3.5: **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Dynamic and Static Scope Rules

- Key idea: in static scope rules
  - » 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 the calling sequences
  - » To resolve a reference, we use the most recent, active binding made at run time

Dynamic Scope

- Interpreted languages
  - » early LISP dialects assumed dynamic scope rules (Perl you can chose static or dynamic)
  - » Later use static – e.g., ML.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

Scope Pragmatics (Review)

```
- First
+ Second
    First
```

- Static scoping: variables always refers to its nearest enclosed binding (between name and object). Compile time
- Dynamic scoping: binding depends on the flow of control at run time and the order subroutines are called, refers to the closest active binding,

```
a: integer    // global

procedure first()
  {
  a = 1       // global or local?
  }
procedure second
  {
  a: integer // local
  first()
  }
a = 2
if read_integer() > 0
  second()
else
  first()
write_integer(a)
```

```
Static:  prints 1 : a is global scope
         of a is closest enclosed a, so
         for "first"'s a refers to global a
Dynamic: prints 1 or 2: if we go to second
         first, first's a refers to second's
         local a (closest binding).
```

Example: Static versus Dynamic Scoping

- **Static scope rules** require that the reference resolve to the most recent, compile-time binding, namely the global variable `a`
- **Dynamic scope rules,** on the other hand, require that we choose the most recent, active binding at run time
  - » Example use: 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

Referencing Environment

- The **referencing environment** of a statement is the collection of all names that are visible in the statement (e.g., env: remember scheme)
  - » In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is **active** if its execution has begun but has not yet terminated
  - » In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Dynamic Scope: Accessing Variables – Trade-Off

1. keep a stack (*association list*) of all active variables (slow access, fast calls)
   - » hunt down from top of stack to find a variable
     - – This is equivalent to searching the activation records on the dynamic chain
2. keep a central table with one slot for every variable name (fast lookup, slow calls)
   - » If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
     - – Otherwise, you'll need a hash function or something to do lookup
   - » Every subroutine changes the table entries for its locals at entry and exit.

Advantages and Disadvantages Dynamic Scope

- **Advantages:**
  - » Simple implementation for interpreted languages
  - » Lack of static structure (e.g., Unix environment variables)
- **Disadvantages:**
  - » Confusing, better to use static variables, default parameters

Some Binding Rules

- Recall that a **referencing environment** of a statement at run time is the
  - » set of active bindings.
  - » A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding.
- **Scope rules:** determine that collection and its order
- **Binding rules:** determine which instance of a scope should be used to resolve references when calling a procedure that was passed a parameter
  - » they govern the binding of referencing environments to formal procedures

Binding *within* a Scope

- Aliasing
- Overloading
  - » operator overloading
  - » function overloading
  - » polymorphism
  - » generic functions
- Modules
  - » between compilations

Aliasing

- What are aliases good for? (consider uses of FORTRAN equivalence)
  - space saving - modern data allocation methods are better
  - multiple representations - unions are better
  - linked data structures - legit
- Also, aliases arise in parameter passing as an unfortunate side effect
  - Euclid scope rules are designed to prevent this
- In general aliases tend to make programs more confusing and more difficult for compiler to perform code improvements

Overloading

- some overloading happens in almost all languages
  - » integer + v. real +
  - » read and write in Pascal
  - » function return in Pascal
- some languages get into overloading in a big way
  - » Ada (see Figure 3.18 for examples)
  - » C++ (see Figure 3.19 for examples)

Overloaded functions

- overloaded functions - two different things with the same name; in C++
  - » overload `norm`

```
int norm (int a){return a>0 ? a : -a;)
complex norm (complex c ) { // ...
```

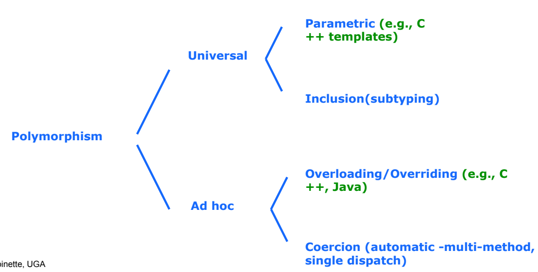
- ad hoc polymorphism

Polymorphism (having many forms)

- ad-hoc polymorphism overloading, coercion (finite types)
- subtype polymorphism – classes related by a super type.
  - » e.g., in OO languages allow parameters to have different types in the same type hierarchy by calling virtual functions appropriate to the concrete type of the actual parameter.
- parametric polymorphism :
  - » Explicit (generic)
    - Syntactic template that can be instantiated in more than one way at compile time
      - specify parameters when you declare or use generic
      - Templates in C++
      - Macro expansion
    - Generic Programming (arguably): Only one definition (e.g., templates, macro, note instantiation is laziness, "not evaluated until needed" characteristics
  - » Implicit (true)
    - Don't have to specify types for which code works, language implementation figures it out and won't let you perform operations on object that do not support them
    - Lisp (run time), ML (compiler)

Polymorphism (review)

- Universal polymorphism : Allows writing code that works with different types
- Ad-hoc polymorphism : Selecting the right implementation code to be executed



Separate Compilation

- Separately-compiled files in C provide a sort of *poor person's modules*:
  - » Rules for how variables work with separate compilation are messy
  - » Language has been jerry-rigged to match the behavior of the linker
  - » *Static* on a function or variable *outside* a function means it is usable only in the current source file
    - This *static* is a different notion from the *static* variables inside a function (retains, persistant value)

Separate Compilation (cont)

- » ***`Extern`*** on a variable or function means that it is declared in another source file
- » Functions headers without bodies are *extern* by default
- » *Extern* declarations are interpreted as forward declarations if a later declaration overrides them

Separate Compilation (cont)

- » Variables or functions (with bodies) that don't say ***static*** or ***extern*** are either ***global*** **or** ***common*** (a Fortran term)
  - – Functions and variables that are given initial values are *global*
  - – Variables that are not given initial values are *common*
- » Matching common declarations in different files refer to the same variable
  - – They also refer to the same variable as a matching *global* declaration

Summary

- **The morals of the story:**
  - » ***language features can be surprisingly subtle***
  - » **designing languages to make life 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**

Conclusion

- **A language that is easy to compile often leads to**
  - » a language that is easy to understand
  - » more good compilers on more machines (compare Pascal and Ada!)
  - » better (faster) code
  - » fewer compiler bugs
  - » smaller, cheaper, faster compilers
  - » better diagnostics

Next Week…

- Project 3 (ML prime numbers, should not take long)
- Exam 1 discussed in detail.
- Contest period – one week after turning back exams… (after that Spring Break…).