Names, Scope, Memory, and Binding

Name, Scope, and Binding

- A name is exactly what you think it is
  - Usually think of identifiers but can be more general
  - symbols (like '+' or '_') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually) in which the binding is active

Binding

- Binding Time is the point at which a binding is created
  - language design time
    - program structure, possible type
  - language implementation time
    - I/O, arithmetic overflow, stack size, type equality (if unspecified in design)
  - program writing time
    - algorithms, names
  - compile time
    - plan for data layout
  - link time
    - layout of whole program in memory
  - load time
    - choice of physical addresses

Binding

- Implementation decisions (continued):
  - run time
    - 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

Binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively
  - “static” is a coarse term; so is "dynamic"
- *IT IS DIFFICULT TO OVERSTATE THE IMPORTANCE OF BINDING TIMES IN THE DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES*

Binding

- In general, 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
- Today we talk about the binding of identifiers to the variables they name
  - Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names

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
  - destruction of objects
- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding it's **garbage**
  - If binding outlives object it's a **dangling reference**
- The textual region of the program in which the binding is *active* is its scope

Lifetime and Storage Management

- Storage Allocation mechanisms
  - Static
  - Stack
  - Heap
- Static allocation for
  - code
  - globals
  - static or own variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions

Lifetime and Storage Management

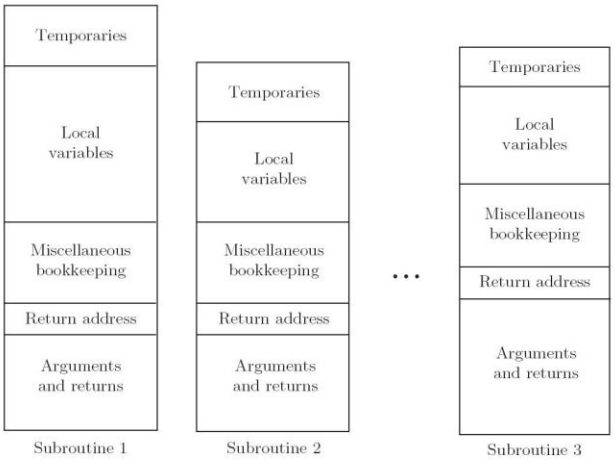

Figure 3.1: **Static allocation of space for subroutines in a language or program without recursion.**

Lifetime and Storage Management

- Central stack for
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines
    (not possible in old FORTRAN – no recursion)
  - reuse space
    (in all programming languages)

Lifetime and Storage Management

- Contents of a stack frame
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Lifetime and Storage Management

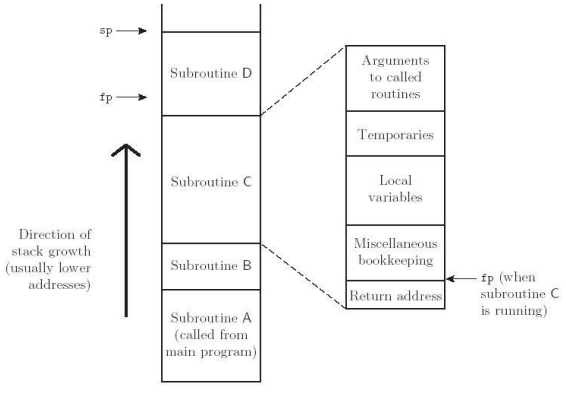

Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (sp) register points to the first 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.

Example: Examine Stack for the C Program

```
int bar(int x)
{
        int z=5;
        return z;
}

int foo(int x)
{
        int y=3;
        x = x + y;
        y = bar(x);
        return x;
}

int main(int argc, char* argv[])
{
        int a=1, b=2, c=3;
        b = foo(a);
        printf("%d %d %d\n",a,b,c);
        return 0;
}
```

Memory Management

Heap

- Region of memory where subblocks are allocated and deallocated dynamically
- More unstructured
- Allocation and deallocation may happen in arbitrary order
  - Memory may become fragmented
  - Need for garbage collection
  - We will describe garbage collection algorithms later
- Heap Management
  - Often managed with a single linked list – the free list – of blocks not in use
    - First Fit?
    - Best Fit? (smallest block large enough to handle request)

Lifetime and Storage Management

- Heap for dynamic allocation

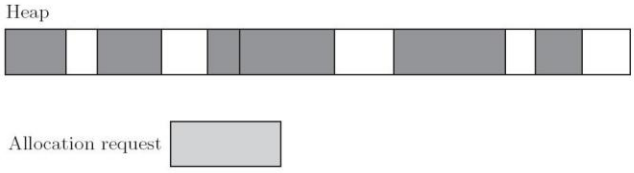

Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

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
- 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

Scope Rules

- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
- Algol 68:
  - ELABORATION = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
  - storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations

Scope Rules

- With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the physical (lexical) structure of the program
  - The determination of scopes can be made by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time
  - Most compiled languages, C++ and Java included, employ static scope rules

Scope Rules

- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Algol 60 and Pascal
  - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope
  - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found

Scope Rules

- Note that the bindings created in a subroutine are destroyed at subroutine exit
- Obvious consequence when you understand how stack frames are allocated and deallocated
- The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime
  - Bindings to variables declared in a module are inactive outside the module, not destroyed
  - The same sort of effect can be achieved in many languages with *own* (Algol term) or *static* (C term) variables

Scope Rules

- 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
  - You 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
- More details in Chapter 8

Scope Rules

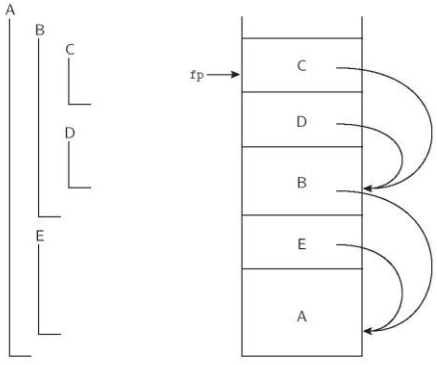

Figure 3.5: **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the 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.

Scope Rules

- The key idea in **static scope rules** is that 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 calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time

Binding of Referencing Environments

- Accessing variables with dynamic scope:
  - (1) keep a stack (*association list*) of all active variables
    - When you need to find a variable, hunt down from top of stack
    - This is equivalent to searching the activation records on the dynamic chain

Binding of Referencing Environments

- 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, you'll need a hash function or something to do lookup
    - Every subroutine changes the table entries for its locals at entry and exit (push / pop on a stack).

Binding of Referencing Environments

- (1) gives you slower access but fast calls
- (2) gives you slower calls but fast access
- In effect, variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
- Because static scope rules tend to be more complicated, however, the data structure and lookup algorithm also have to be more complicated

Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
  - early LISP dialects assumed dynamic scope rules.
- 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 Rules
Example: Static vs. Dynamic

```
int a;

void first()
{
     a = 1;
}

void second()
{
     int a = 3;
     first();
}

void main()
{
     a = 2;
     second;
     printf("%d\n",a);
}
```

Scope Rules
Example: Static vs. Dynamic

- If static scope rules are in effect (as would be the case in C), the program prints a `1`
- If dynamic scope rules are in effect, the program prints a `2`
- Why the difference? At issue is whether the assignment to the variable `a` in function *first* changes the variable `a` declared in the main program or the variable `a` declared in function *second*

Scope Rules
Example: Static vs. Dynamic

- 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
  - Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
    - Begin
      - Print_base: integer := 16 // use hex
      - Print_integer(n)
  - This is generally considered bad programming practice nowadays
    - Alternative mechanisms exist, e.g. optional parameters

```
type person = record
    . . .
    age : integer
    . . .
threshold : integer
people : database

function older_than(p : person) : boolean
    return p.age ≥ threshold

procedure print_person(p : person)
    -- Call appropriate I/O routines to print record on standard output.
    -- Make use of nonlocal variable line_length to format data in columns.
    . . .

procedure print_selected_records(db : database;
        predicate, print_routine : procedure)
    line_length : integer

    if device_type(stdout) = terminal
        line_length := 80
    else    -- Standard output is a file or printer.
        line_length := 132
    foreach record r in db
        -- Iterating over these may actually be
        -- a lot more complicated than a 'for' loop.
        if predicate(r)
            print_routine(r)

-- main program
. . .
threshold := 35
print_selected_records(people, older_than, print_person)
```

Static scoping or **Deep binding** makes sense here

Dynamic scoping or **Shallow binding** makes some sense here

Binding of Referencing Environments

- 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

- First-class status: objects that can be passed as parameters or returned and assigned
- Second-class status: objects that can be passed but not returned or assigned

- Some programming languages allow subroutines to be first-class
  - What binding rules to use for such a subroutine?

Binding within a Scope

- Aliasing
  - Two or more names that refer to a single object in a given scope are aliases
  - What are aliases good for?
    - space saving - modern data allocation methods are better
    - multiple representations
    - linked data structures - legit
  - Also, aliases arise in parameter passing
    - Sometimes desirable, sometimes not

Aliases

Java:

```
public static void foo(MyObject x)
{
        x.val = 10;
}

public static void main(String[] args)
{
        MyObject o = new MyObject(1);
        foo(o);
}
```

Aliases

C++:

```
public static void foo(int &x)
{
        MyObject o;
        x = 10;
}

public static void main(String[] args)
{
        int y = 20;
        foo(y);
}
```

Binding within a Scope

- Overloading
  - some overloading happens in almost all languages
    - integer + v. real +
  - Handled with help from the symbol table
    - Lookup a list of possible meanings for the requested name; the semantic analyzer chooses the most appropriate one based on the context
  - some languages get into overloading in a big way
    - Ada
    - C++

Ada Constant Overloading

```
declare
    type month is (jan, feb, mar, apr, may, jun,
                   jul, aug, sep, oct, nov, dec);
    type print_base is (dec, bin, oct, hex);
    mo : month;
    pb : print_base;
begin
    mo := dec;      -- the month dec
    pb := oct;      -- the print_base oct
    print(oct);     -- error!  insufficient context to decide
```


C++ Operator Overloading

```
const Money operator +(const Money& amount1, const Money& amount2)
{
   int allCents1 = amount1.cents + amount1.dollars*100;
   int allCents2 = amount2.cents + amount2.dollars*100;
   int sumAllCents = allCents1 + allCents2;
   int absAllCents = abs(sumAllCents); //Money can be negative.
   int finalDollars = absAllCents/100;
   int finalCents = absAllCents%100;

   if (sumAllCents < 0)
   {
      finalDollars = -finalDollars;
      finalCents = -finalCents;
   }

   return Money(finalDollars, finalCents);
}


bool operator ==(const Money& amount1, const Money& amount2)
{
   return ((amount1.dollars == amount2.dollars)
        && (amount1.cents == amount2.cents));
}
```

Binding within a Scope

- It's worth distinguishing between some closely related concepts
  - overloaded functions - two different things with the same name; in Ada

```
function min(a, b : integer)  return integer…
function min(x, y : real)  return real …
```

  - In Fortran, these can be automatically coerced:

```
real function min(x,y) real x,y  ...
```

    Fortran will convert int input to reals, find the min, then return the result back as a real

Binding within a Scope

- generic functions (modules, etc.) - a syntactic template that can be instantiated in more than one way *at compile time*
  - Also called **explicit parametric polymorphism**
  - via macro processors in C++
  - Templates in C++/Java

```
public class TwoTypePair<T1, T2>
{
    private T1 first;
    private T2 second;

    public TwoTypePair(T1 firstItem, T2 secondItem)
    {
        first = firstItem;
        second = secondItem;
    }

    public T1 getFirst()
    {
        return first;
    }
...
TwoTypePair<String, Integer> rating = new TwoTypePair<String, Integer>("The Car Guys", 8);
```

Separate Compilation

- Since most large programs are constructed and tested incrementally, and some programs can be very large, languages must support separate compilation
- Compilation units usually a “module”
  - Class in Java/C++
  - Namespace in C++ can link separate classes
  - More arbitrary in C
  - Java and C# were the first to break from the standard of requiring a file with header information for all methods/classes

Conclusions

- The morals of the story:
  - language features can be surprisingly subtle
  - Determining how issues like binding, naming, and memory are used have a huge impact on the design of the language
  - Static vs. dynamic scoping is interesting, but all modern languages use static scoping
  - most of the languages that are easy to understand are easy to compile, and vice versa