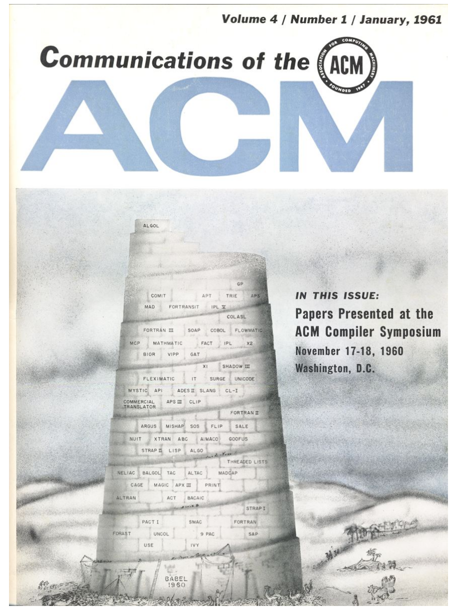
Programming Languages



CSE 3302, Summer 2019
Module 03
Names, Scopes, and Bindings

M03

Names, Scopes, and Bindings

Introduction

- The initial high-level programming languages were significantly more *abstract* than the assembler languages then in use.
  - Fortran, Algol, Lisp, and COBOL were *machine independent*.
- Further, they were *easier* for humans to use than assembler.
  - This *ease of use* is still a primary goal for language design.

- Why easier?
  - Fundamentally because of *names* — that is, a way to refer to program elements *symbolically* rather than with low-level machine descriptions such as addresses.

Names …

- Names are a key part of any *abstraction* scheme.
  - Using a name to refer to something helps to reduce its *conceptual complexity* by *hiding irrelevant details*.
  - Using subroutines is an example of *control abstraction*.
    - We get the result of executing the subroutine without having to know all of the grubby details of how it ran.
  - Using classes is an example of *data abstraction*.
    - We get the behavior of an object without having to know how it is represented (or how its methods are implemented).

Names …

- What's a *name?* "A mnemonic character string used to represent something else."
  - Usually *alphanumeric*, though strings such as ':=' or '+' can be names (of operators, for example).
- To be useful, a name has to be associated with the thing it is to represent.
  - We call this action *binding* the name to the thing.
  - *Unbound* names do not refer to anything.
  - A name might be bound at one time, then unbound, …
  - A binding's *scope* is the part of the program where it is active.

Names and Binding

- Names can be bound to many kinds of things …
  - Execution points (labels), mutable variables, constant values, functions, types, type constructors, classes, modules, packages, execution points with environments (continuations), …
  - It varies language to language.
- *Binding* in general means the resolving of *any* design decision in a language, its implementation, and the programs that are written in the language.

Names and Binding

- Binding associates *attributes* with names.
  - For example, declarations, assignments, prototypes, function definitions.
  - May be *explicit* or *implicit*.
- For example, in C,
  - `int x;` *explicitly* binds the *type* of `x` to `int`.
  - The *allocation method* of `x` (*static* or *dynamic*) is *implicit* since it depends on where this declaration is placed.
  - The *initialization method* of `x` (zero or garbage) is *implicit* since it depends on where this declaration is placed.

```
// static allocation
// initialized to 0
int x;

void f()
{
    // dynamic allocation
    // initialized to garbage
    int x;
}
```

Binding Time

- Names can become bound at many different times …
- ***Language design time***
  - When the language is defined, many names are bound to the fundamental parts of the language.
    - Control-flow constructs, primitive types, constructors for more complex types, …
  - In many languages, the name "`if`" is bound to the if-statement control-flow construct.
  - In C, "`int`" is bound to the primitive integer type and "`struct`" to the constructor for record types.

Binding Time

- ***Language implementation time***
  - When a language is implemented in a particular environment, many local decisions are made.
  - Names can be bound to represent these decisions so the information is accessible to the users.
    - Exact sizes of the primitive types (e.g., how wide is an `int`?)
    - I/O and other OS interaction channels.
    - Organization and limits for the stack, heap, etc.

Binding Time

- ***Program writing time***
  - The user chooses names to represents the various elements of the program being written.

- ***Compile time***
  - Compilers choose at least …
    - The mapping of high-level constructs to the target code.
    - The structure and layout of statically defined data in memory.

Binding Time

- ***Link time***
  - Once compiled, a program may have to be *linked* to *separately compiled* entities.
    - May be from a library, may be supplied by the user.
  - The linker resolves inter-entity references (that is, a name in one entity referring to an object in another).
  - (Virtual) addresses are selected for all objects.

Binding Time

- ***Load time***
  - *Loading* is the insertion of the program into memory for execution.
    - Originally at a fixed physical memory location, now into *virtual memory*. (Small scale processors still use physical addresses.)
  - The (virtual) addresses used by the program are translated to physical addresses by the processor's *memory management unit* (*MMU*).

Binding Time

- ***Runtime***
  - Very broad term!
    - Includes all time from the start of execution to when the program finally exits.
    - Subtimes here include program start-up time, module entry time, *elaboration* time, subroutine call time, block entry time, expression evaluation time, statement execution time, …
    - Languages differ.
  - Variable values are bound at runtime as well as many other bindings which vary language to language.

Binding Time

- *Static* bindings are bindings made at any time *before* runtime.
- *Dynamic* bindings are bindings made during the program's execution, that is, at runtime.

- In general,
  - *Earlier* binding times are associated with greater *efficiency*.
    - Compiled languages tend to have early binding times.
  - *Later* binding times are associated with greater *flexibility*.
    - Interpreted languages tend to have later binding times.

Binding Time

- Generally it's considered better to bind as much as possible *statically* rather than *dynamically*.
  - Q: Why?
  - A: *Efficiency*! The sooner a binding is known, the more efficient its processing can be made.
  - A2: *Correctness*! More and more types of errors can be detected the earlier a binding is made.

Binding Time Example

- Q: When is the meaning of the "+" operator bound for the expression "`x + 10`"?
  - Hint: it will depend on the language.
- A: Pretty much *any* time (depending on the language)!
  - Could be language design time, language implementation time, program writing time, …
  - Might even be at *runtime*! (Suppose it's a dynamically typed language and the type of `x` isn't known until the expression is actually evaluated?)

Managing Binding Time

- Bindings need to be handled during both compilation *and* execution.
- During compilation, binding information is kept in the compiler's *symbol table* (① *names* and their *attributes*).
- During execution, the runtime environment keeps track of

  bindings (② *names* and the *objects* to which they refer)

  and the *state* (③ *objects* and their *values*)
- [An interpreter keeps track of all three kinds ①②③ of bindings.]

Scope Rules

- A binding's *scope* is that part of the program where the binding is *active*.
- Fundamental to all programming languages is the ability to *name* data, that is, refer to data using *symbolic* names rather than addresses.
- Not all data is named!
  - In C and Pascal, for example, *dynamic* storage is referenced by *pointers*, not *names*.
  - Yes, the pointer itself can have a name, but the name is of the pointer, not what is being pointed at.

Scope Rules

- Where can declarations happen?
- Depending on the language, in a number of places …
- ***External***
    - Came from some separately compiled module.
- ***Global***
    - Outside all other scopes in the file.
- ***Blocks***
    - `{ … }`, `begin … end`, etc. in Algol-descended languages.
- ***Structured data type***
- ***Class***

Binding Lifetime

- The time between a binding's *creation* and *destruction* is the *binding's lifetime*.
- The time between an object's *creation* and *destruction* is the *object's lifetime*.
- These do *not* have to be the same!
    - A binding can be created for an existing object and then destroyed. For example, a reference parameter for a function.
- A name that outlives its object is a *dangling reference*.
- An object that outlives all of its references is *garbage*.

Key Lifetime Events

- Creation of objects
- Creation of bindings
- References to variables (that use bindings)
- (Temporary) deactivation of bindings
- Reactivation of bindings
- Destruction of bindings
- Destruction of object

Object Lifetime and Allocation Methods

- Generally three lifetime spans …
- ***Static objects***
  - Lifetime is the entire execution period of the program.
  - Example: global, external variables in C.
- ***Stack objects (dynamic)***
  - Lifetime is from function or block entry to its exit.
  - Example: a function's local variables.
- ***Heap objects (dynamic)***
  - Lifetime is arbitrary; not tied to any particular function or block, but not necessarily the program's entire execution.
  - Example: dynamic data objects; C++ `new`, C `malloc`.

Static and Dynamic Objects

- *Static* objects can be allocated at compile time.
- *Dynamic* objects can be allocated only at runtime.
- Q: Why would an object have to be dynamic?
- A: Several reasons, but generally because the object depends on information that is not known until runtime.
  - Recursion
  - Explicitly allocated objects (`new`, `malloc`)
  - Higher-order functions, etc.

Static and Dynamic Objects

- Generally, a program's memory usage is a *mixture* of static and dynamic objects and the limits of a program's memory usage cannot be determined at compile time.
  - Early Fortran is an exception!
    - It had no recursion and no dynamic memory objects.
    - The total amount of user memory was known at compile time.

Object Lifetime and Storage Management

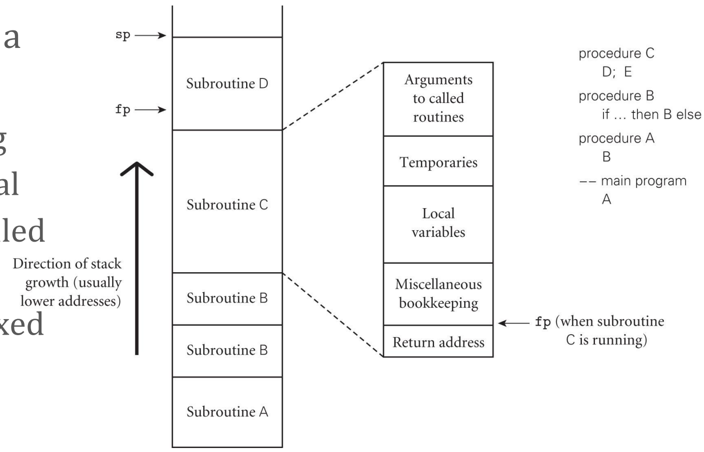
- *Static Objects*
  - Global variables, literals (string, numeric), explicit constants, the actual machine code for the program, (internal) tables used to support execution and/or for debugging purposes.
  - Since statically allocated, may even be in ROM, if it's only read and not written.
- Q: Could a function's local variables be static objects? Why or why not?
- A: Yes! If the function is not called recursively, there's need for only one set of local variables. (Early Fortran was like this.)

Object Lifetime and Storage Management

- *Stack-based Objects*
- A central stack can be used for
  - Parameters to function calls, function or block local variables, temporary values during execution, etc.
- Q: Why use a stack?
- A: Obvious answer is to support *recursion*. Each time a function or block is entered, a new set of variables is allocated.
- A2: More subtle answer is *efficient use of space*. Even if variables could be allocated statically, putting them on the stack reuses space.

Stack Frame

- The stack objects for a function are organized into a *stack frame*.
- Holds return address to calling routine, local variables, internal data, and arguments to any called functions.
- Local variables are assigned fixed *offsets* showing their locations relative to the current *frame pointer* (`fp`).



Stack Maintenance

- Ensuring that the stack is always in the proper structure and format is handled by the *calling sequence* and the function's *prologue* and *epilogue*.
  - The *calling sequence* is what the calling function has to do before transferring control to the called function (and what it might do after getting control back).
  - The *prologue* is what the called function does just as it is entered.
  - The *epilogue* is what the called function does just as it is about to return control to the function that called it.

Stack Maintenance

- *Space* is saved by having as much in the prologue and epilogue and as little in the calling sequence as possible.
  - There is only one prologue and one epilogue per function, but the calling sequence occurs every place the function is called.
- *Time may* be saved by putting more in the body of the called function.
  - Some compilers do what's called *interprocedural optimization* in an attempt to speed up execution.
  - This could involve moving code into *or* out of the body of the called function.

Function Call Example …

```
public static void main( String[] args ) {
  int i = 5;
  int j = 2;
  int k = max( i, j );

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k );
}
```

```
public static int max( int num1, int num2 ) {
  int result;

  if ( num1 > num2 )
    result = num1;
  else
    result = num2;

  return result;
}
```

Space required for the main method.
k:
j: 2
i: 5

① The main method is invoked.

Space required for the max method.

num2:
num1:

Space required for the main method.
k:
j: 2
i: 5

② The max method is invoked.

```java
public static void main( String[] args ) {
  int i = 5;
  int j = 2;
  int k = max( i, j );

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k );
}
```

```java
public static int max( int num1, int num2 ) {
  int result;

  if ( num1 > num2 )
    result = num1;
  else
    result = num2;

  return result;
}
```

Space required for the main method.
k:
j: 2
i: 5

① The main method is invoked.

Space required for the max method.

num2: 2
num1: 5

Space required for the main method.
k:
j: 2
i: 5

② The max method is invoked.

Function Call Example …

```java
public static void main( String[] args ) {
  int i = 5;
  int j = 2;
  int k = max( i, j );

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k );
}
```

```java
public static int max( int num1, int num2 ) {
  int result;

  if ( num1 > num2 )
    result = num1;
  else
    result = num2;

  return result;
}
```

Space required for the main method.
k:
j: 2
i: 5

① The main method is invoked.

Space required for the max method.

num2: 2
num1: 5

Space required for the main method.
k:
j: 2
i: 5

② The max method is invoked.

Space required for the max method.
result: 5
num2: 2
num1: 5

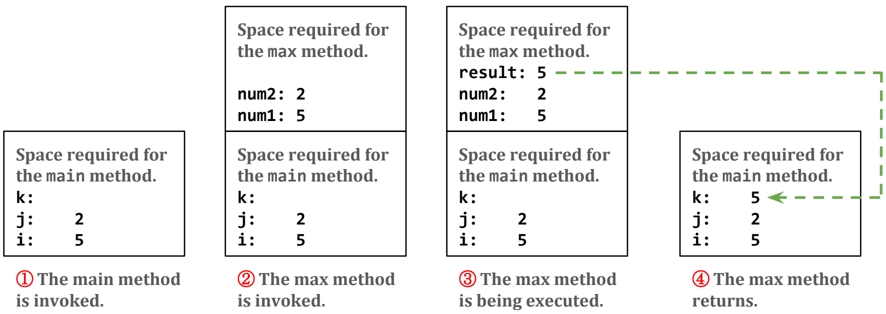
Space required for the main method.
k:
j: 2
i: 5

③ The max method is being executed.

Function Call Example …

```java
public static void main( String[] args ) {
  int i = 5;
  int j = 2;
  int k = max( i, j );

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k );
}
```

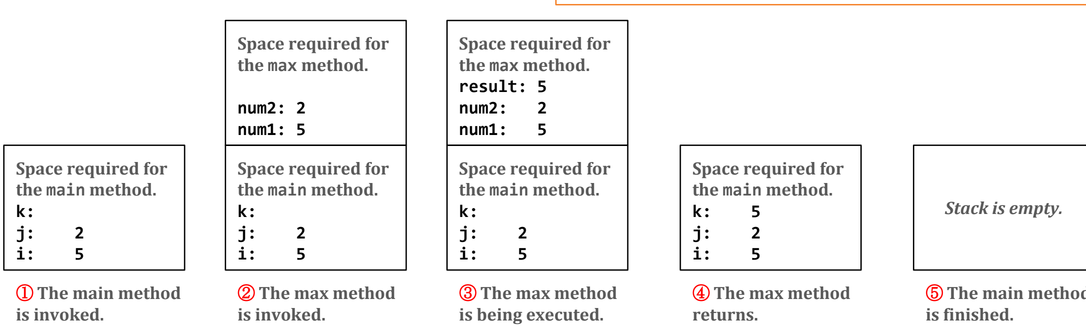
```java
public static int max( int num1, int num2 ) {
  int result;

  if ( num1 > num2 )
    result = num1;
  else
    result = num2;

  return result;
}
```



Function Call Example …

```java
public static void main( String[] args ) {
  int i = 5;
  int j = 2;
  int k = max( i, j );

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k );
}
```

```java
public static int max( int num1, int num2 ) {
  int result;

  if ( num1 > num2 )
    result = num1;
  else
    result = num2;

  return result;
}
```



Object Lifetime and Storage Management

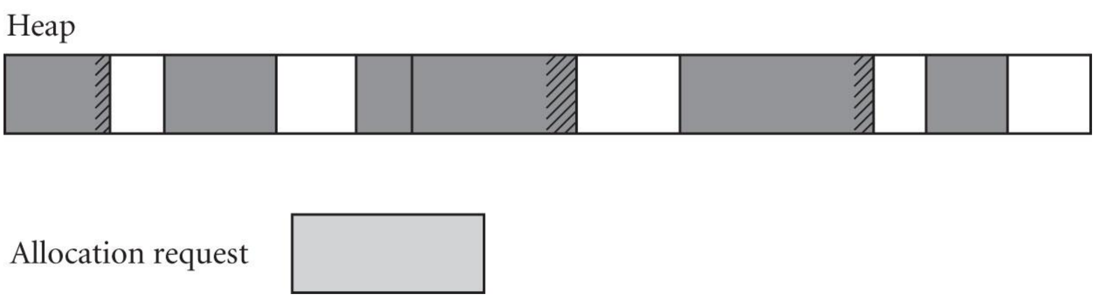
- *Heap-based Allocation*
- A *heap* is a region of storage from which blocks of memory can be allocated and deallocated at arbitrary times.
    - Used for dynamically allocated data structures whose size may change during execution.
- Q: Why not use the stack for this?
- A: The stack is not convenient for an object whose size may change. The space on the stack is fixed size.
- A2: A heap object may outlive the function that created it. The function's stack frame is gone when it exits.

Heap Management

- Often the free parts of the heap are kept in a (singly) linked list known as the *free list*.
    - The *first fit* algorithm uses the first block that's big enough.
    - The *best fit* algorithm looks for a block that's just big enough.
- Instead of a single free list, some heaps have multiple lists, based on the size of the blocks.
- The *buddy system* uses blocks that are a power of 2 in size.
- The *Fibonacci system* uses sizes based on the Fibonacci sequence instead of powers of 2.

Heap Fragmentation

- A heap may *fragment* as allocation and deallocation occurs.
- *Internal* fragmentation is when an allocated block is larger than the requested size.
- *External* fragmentation is when there is enough space to satisfy an allocation request, but it's too spread out.
- When allocation fails, *compaction* may be tried (depends on the language).



Object Lifetime and Storage Management

- *Garbage Collection* (*GC*)
  - Generally required in all languages where deallocation is not *explicit*.
  - Manual deallocation errors are the among the *most common* and *costly* of bugs in real-world programs.
- With GC, objects are deallocated *implicitly* when they are no longer accessible.
- Many, many methodologies to do this …
  - Reference counting, Mark / Sweep, Copying, Generational, …
  - Stop-the-world, Incremental, Concurrent, …

Garbage Collection

- The aim of *Garbage Collection* (*GC*) is to deallocate any object that will never again be used.
- How to find such objects?
- Recognize that for an object to be used, it must be *reachable*.
  - It's a *global*.
  - It's in an *active stack frame*.
  - It's *pointed to* by a reachable object.
- This is a graph-traversal problem.

Garbage Collection

- One graph-traversal method is *Mark* and *Sweep*.
  - Every object has an extra bit, called the *mark bit*, which starts off clear.
- *Mark* phase
  - Set the mark bit of every global object, every object in a stack frame, and every temporary object.
  - Every time we set an object's mark bit, we also set the mark bits of every object pointed to by that object.
    - This is a recursive procedure.
- *Sweep* phase
  - For every object, if its mark bit is clear, it's *garbage*. Free it.
  - Otherwise clear its mark bit (so it's ready for the next GC).

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.
- Scoping rule example: *Declaration before Use*.
  - Q: In Java can a name be used before it is declared?
  - A: For a local variable, no. For a class property or method, *yes*!

```
public class example {
 public static void method_1()
 {
   ...
   method_2();
   ...
 }

 public static void method_2()
 {
   ...
 }
}
```

Scope Rules

- In most languages with subroutines, we *open* a new scope on subroutine entry.
  - Create bindings for the (new) local variables.
  - Deactivate bindings for *outer* variables that are re-declared; these are said to have a *hole* in their scope.
- Algol and later Ada used the term *elaboration* for the process of creating bindings when entering a scope.
  - Storage may be allocated, tasks started, even exceptions propagated as a result of declaration elaboration.

Scope Rules

- We *close* the scope on subroutine exit.
  - Destroy the bindings for local variables.
  - Reactivate the bindings for those outer variables that were deactivated on entry.

Scope Rules

- *Static scoping* (also called *Lexical scoping*)
  - The scope is defined in terms of the *physical* (*lexical, textual*) structure of the program.
  - Scope determination can be made by the *compiler*.
  - All bindings can be resolved by simply *examining the text* of the program.
  - Typically, the *most-recent active binding* made at compile time is the correct one.
  - Most compiled languages use lexical scoping rules.

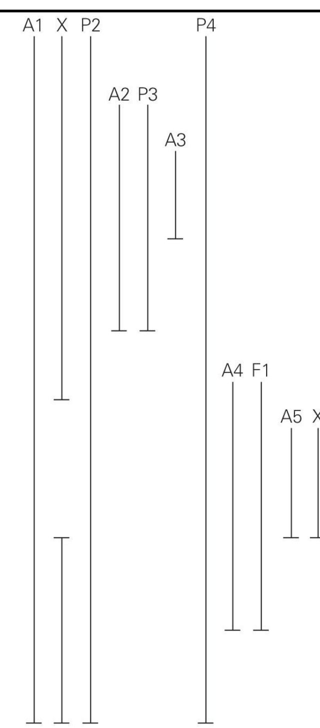
Scope Rules

- *Static scoping* examples
  - One big scope (Basic)
  - Scope of a function (locals live only during function execution)
  - Block scope (any { … }, `begin` … `end` unit)
  - Nested subroutines
- If a name is active in one or more scopes, the *closest nested scope* rule applies.

Scope Rules

- Typical static scoping example.
- Most recent enclosing bindings are visible.
- *Overriden* bindings have one or more *holes* in their scope.
  - `P1`'s `X` is *overriden* by `F1`'s `X` inside `F1`'s body.
  - `P1`'s `X` thus has a *hole* in its scope.

```
procedure P1(A1)
    var X               -- local to P1
    ...
    procedure P2(A2)
        ...
        procedure P3(A3)
            ...
        begin
            ...         -- body of P3
        end
        ...
    begin
        ...             -- body of P2
    end
    ...
    procedure P4(A4)
        ...
        function F1(A5)
            var X       -- local to F1
            ...
        begin
            ...         -- body of F1
        end
        ...
    begin
        ...             -- body of P4
    end
    ...
begin
    ...                 -- body of P1
end
```

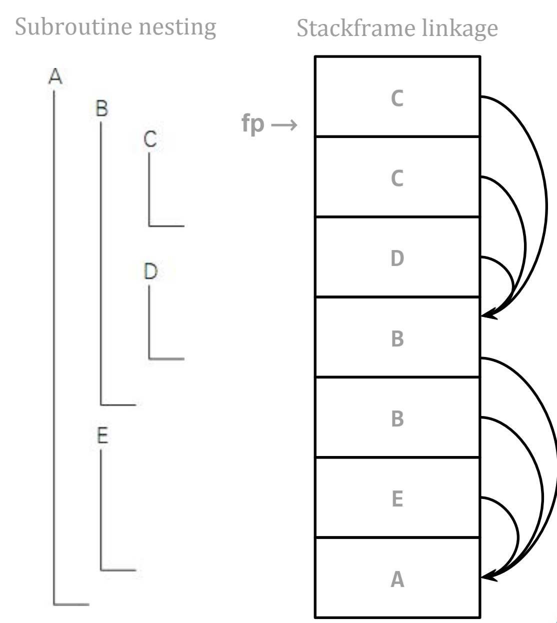


Scope Rules

- Static scoping rules go back to Algol.
- All of Algol's derivatives follow the same block structure, closest nested scope rule.
  - A name is known in its own scope and each enclosed scope unless it is re-declared in an enclosed scope.
  - To resolve a reference to a name, examine the local scope and statically enclosing scopes until a binding is found.

Static Chains

- Nested functions have access to the local variables of their enclosing functions.
- At runtime, there needs to be a way for these bindings to be found.
- One mechanism is *static chains*.
- Each nested routine has a linkage to its enclosing function's "most recent" stack frame.
- Even if an enclosing function has been called recursively, it's the "most recent" that has to be found.



Scope Rules — Declaration Order and Nesting

- The earliest languages required all declarations to be at the beginning of their scope.
- Even so, their order matters because declarations can refer to each other.
- A subtle point is the exact range of the scope of a declaration in its “block”.
  - The *entire* block?
  - The portion of the block *after* the declaration?

Scope Rules — Declaration Order and Nesting

- Pascal has the properties that a declaration’s scope is its entire block, *and* that an object must be declared before it can be used.
- The result can be some mystifying errors.
- ① The second declaration of `N` hides the first, but since it follows `M`, it cannot be used in the declaration of `M`.
- ② Same kind of problem. The error messages are
  - “`N used before declaration`”
  - “`N is not a constant`”
- Very confusing, especially in a long program, where there might be some distance between the two declarations of `N`.

①
```
const N = 10;
…
procedure foo;
const
    M = N;   (* static semantic error! *)
    …
    N = 20;  (* hiding declaration *)
```

②
```
const N = 10;
…
procedure foo;
const
    M = N;   (* static semantic error! *)
var
    A : array [1..M] of integer;
    N : real;  (* hiding declaration *)
```

Scope Rules — Declaration Order and Nesting

- A similar issue can occur with C#, though the error message is a bit more useful.
    - Though it doesn't say which line has the 'local variable' that is doing the hiding.

```
scope.cs(9,19): error CS0844: A local
variable 'N' cannot be used before it is
declared. Consider renaming the local
variable when it hides the member 'A.N'
```

```
1. class A {
2.   // Outer declaration of N.
3.   const int N = 10;
4.
5.   void foo() {
6.     // The N in the initializer is
7.     // the one on line 14 below, not
8.     // line 3 above.
9.     const int M = N;
10.
11.    // Inner declaration of N.
12.    // This one hides the outer
13.    // declaration on line 3.
14.    const int N = 20;
15.  }
16. }
```

Scope Rules — Declaration Order and Nesting

- Following the rules of Pascal, the compiler must examine *all* declarations for a scope before it can decide if any enclosing declarations are hidden.
- To simplify this, some versions of Pascal (and its successors) state that the scope *starts* at the point of declaration instead of being the entire block.
    - Ada, C, C++, Java are like this.
- C++ and Java even relax the Declare-Before-Use rule.
    - Members of a class are visible to all methods no matter which order they are declared in.

Scope Rules — Declaration Order and Nesting

- Python goes even further!
  - No declarations are required but a scope has a variable *if it is written to inside the scope*.
    - *Written to ≡ Appears on the LHS of an assignment*.
  - If not written to, an outer scope will be searched for the variable.
    - In ①, both prints say x is 1.
    - In ②, inner print says x is 2, outer *still* says x is 1.

①
```
def outerFunction() :
  x = 1

  def innerFunction() :
    print( "inner: x is", x )

  innerFunction()
  print( "outer: x is", x )
```

②
```
def outerFunction() :
  x = 1

  def innerFunction() :
    x = 2
    print( "inner: x is", x )

  innerFunction()
  print( "outer: x is", x )
```

Declarations and Definitions

- *Recursive* and *mutually-referential* types cause problems in Declare-Before-Use languages.
  - How can something be declared if *declaring* it requires *using* it?
  - How can two declarations each occur *before* the other?
- C and C++ solve the problem by distinguishing *declaration* from *definition*.
  - The *declaration* introduces a name and indicates its scope.
    - Likely omits many details required for implementation.
  - The *definition* provides all of the implementation details.

Declarations and Definitions

- C recursive and mutually referential types.
- The `employee` struct refers to both itself and the `manager` struct.
- The `manager` struct refers to the `employee` struct.
- `manager` has to be *declared* before the `employee` struct.
- `manager` is later *defined*.

```
struct manager;          // declaration only

struct employee {
 struct manager *boss;
 struct employee *next_employee;
 …
};

struct manager {         // definition
 struct employee *first_employee;
 …
};
```

Declarations and Definitions

- C mutually referential functions.
- `list()` and `list_tail()` each need to call to other, so *neither* can be first.
- A *declaration* is given for `list_tail()` so `list()` can call it.
- `list_tail()` is later *defined*.

```
// declaration only
void list_tail( follow_set fs );

void list( follow_set fs ) {
  switch ( input_token ) {
    case id : match( id ); list_tail( fs );
  …
}

// definition
void list_tail( follow_set fs ) {
  switch ( input_token ) {
    case comma : match( id ); list( fs );
  …
}
```

Referencing Environment

- At each point in a program's execution, the set of *active* bindings is called the current *referencing environment*.
- This set is determined by the *scope rules* of the language (either *static*, *dynamic*, or some combination).
- The referencing environment generally corresponds to a *sequence* of scopes that can be examined *in a specific order* to find the current binding for a given name.

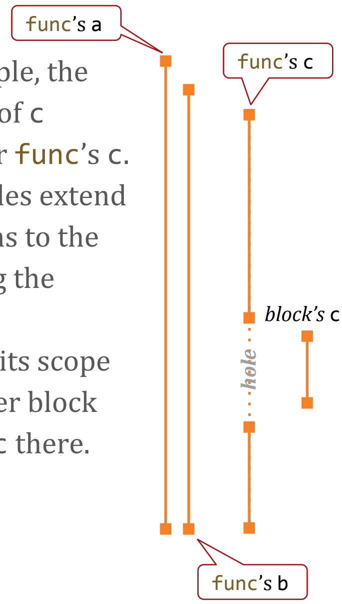
Nested Blocks

- Many languages, including Java, C#, and later versions of Algol and C allow declarations wherever a statement may appear.
  - Other languages such as Ada and early versions of Algol and C permit declarations only at the beginning of a subroutine.
- Variables declared in nested blocks can be very useful.
  - Their scope is the block itself, so they do not interfere with surrounding code.

```
void func() {
  int a;
  int b;
  int c;

  ...
  // The outer 'c' is in force here.

  {
    // This block introduces another
    // variable named 'c', which overrides
    // the outer variable named 'c'.
    // The scope of the inner 'c' is only
    // the block itself.
    int c = a;
    a = b;
    b = c;
  }

  // The outer 'c' is in force here.
  // Its value was not disturbed by the
  // inner block assignment to 'c'.
  ...
}
```

Nested Blocks can cause Scope Holes

- In our previous example, the block's redeclaration of `c` causes a *scope hole* for `func`'s `c`.
- `func`'s `a` and `b` variables extend from their declarations to the end of `func`, including the inner block.
- `func`'s `c` variable has its scope overridden in the inner block by the declaration of `c` there.




```
void func() {
  int a;
  int b;
  int c;

  ...
  // The outer 'c' is in force here.

  {
    // This block introduces another
    // variable named 'c', which overrides
    // the outer variable named 'c'.
    // The scope of the inner 'c' is only
    // the block itself.
    int c = a;
    a = b;
    b = c;
  }

  // The outer 'c' is in force here.
  // Its value was not disturbed by the
  // inner block assignment to 'c'.
  ...
}
```

'Scope Resolution' Operator

- Many languages, including C++, PHP, and Ruby, provide a 'scope resolution' operator.
  - Other languages such as Java, C#, Python, Go, etc. rely on the *module member access* concept to name objects otherwise invisible.

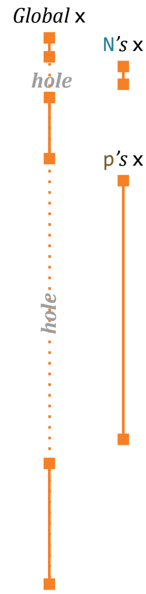
- In this C++ example, the '`::`' operator is used to make the global `x` object accessible inside the function `p()`, where it would otherwise be hidden due to the local redeclaration of `x`.
- By using the specific name `N`, the `x` in that namespace may also be made accessible.

C++

```
int x;

namespace N { float x; }

void p() {
  // This declaration of x overrides
  // the global declaration.
  char x;

  // This assignment is to the local x.
  x = 'a';

  // Using the 'scope resolution' operator
  // gives access to the global x.
  ::x = 42;

  // Using the specific name N gives access
  // to the x in that namespace.
  N::x = 3.14159;
}

void q() {
  // This assignment is to the global x
  // because there is no overriding
  // declaration in q().
  x = 24;
}
```

More on Scope Holes …

- This C++ example also gives a good example of *scope holes*.
- The global x has two scope holes because of the redeclarations inside the namespace N and the function p().
- (The scope of the x declared in the namespace N is a single line.)
- Using the scope resolution operator, both other x objects are accessible inside p().



C++

```
int x;

namespace N { float x; }

void p() {
  // This declaration of x overrides
  // the global declaration.
  char x;

  // This assignment is to the local x.
  x = 'a';

  // Using the 'scope resolution' operator
  // gives access to the global x.
  ::x = 42;

  // Using the specific name N gives access
  // to the x in that namespace.
  N::x = 3.14159;
}

void q() {
  // This assignment is to the global x
  // because there is no overriding
  // declaration in q().
  x = 24;
}
```

Modules

- *Information hiding* is one way to control *conceptual complexity* (or *cognitive load*).
  - As software systems get bigger and bigger, it's harder and harder to keep track of all the details.
- Through information hiding, we make objects and algorithms *invisible*, whenever possible, to portions of the system that do not *need* them.
  - Think of subroutines, classes, etc. as mechanisms for information hiding.

Modules

- One way to hide information is through *Modular Programming* (also known as *Modularization*).
  - Wikipedia: "***Modular programming*** is a *software design technique* that emphasizes *separating the functionality of a program* into *independent, interchangeable* ***modules***, such that *each contains everything necessary to execute* ***only one aspect*** *of the desired functionality*."
- *Modularization* helps in dividing the work among multiple programmers or programming teams.
  - Work can be focused on smaller, easier-to-understand pieces.
  - Work can proceed in parallel on multiple fronts.

`from https://en.wikipedia.org/wiki/Modular_programming but emphasis added.`

Modules

- *Modularization* improves the *quality* of the system.
  - Less chance of *name collision*.
  - Improved chance of *data integrity*.
  - Improved *compartmentalization* of run-time errors.
- *Modularization* is crucial for *maintenance*.
  - Long after the original developers may have moved on, bug fixes and enhancements will still have to be made.
    - The system must be as easy to understand as possible.
  - Maintenance costs usually *far outweigh* implementation costs.
    - Ask anyone working in the real world …

Modules

- Hiding information inside subroutines is useful, but the lifetime of such information is that of the execution of the subroutine.
  - The hidden information is created on subroutine *entry* and destroyed on subroutine *exit*.
  - The `static` construct in C, C++, `save` in Fortran, etc. can be used to retain values across executions of the subroutine.
- The result is a *single-subroutine abstraction*.
  - Useful, but not all *that* useful.
  - The information can't be used by more than one subroutine.

Modules

- We can kind of fake a module in C++ by using *namespaces*.
- In this Pseudo Random Number Generator (PRNG) example, `seed`, `a`, and `m` are all accessible to `setSeed()` and `randInt()`, but none is visible outside the namespace.
- To use this 'module', we can do this:

```
randMod::setSeed( 0xDeadBeef );
unsigned int i = randMod::randInt();
```

C++

```
#include <time.h>

namespace randMod {
  unsigned int seed = time(0);
  const unsigned int a = 48271;
  const unsigned int m = 0x7FFFFFFF;

  void setSeed( unsigned int s ) {
    seed = s;
  }

  unsigned int randInt() {
    return seed = ( a * seed ) % m;
  }
}
```

Modules

- Using a ‘module’ that way is kind of cumbersome.
  - Every time we want to use a routine, we have to explicitly name it using the scope resolution operator.

- We can instead employ C++’s `using` statement to make the functions directly accessible, like so:

```
using randMod::setSeed, randMod::randInt;
// Now setSeed() and randInt() can be used directly.
```

Modules

- Listing each function is kind of tedious, so we can also just use the namespace itself to get them all.

```
using randMod;
// Now setSeed() and randInt() are both accessible.
```

- Oops! By doing that, we *also* made `seed`, `a`, and `m` directly accessible.
- C++’s `using` statement is kind of a blunt ~~weapon~~ tool.
  - It seems to make *too little* or *too much* accessible.
  - There’s no way to explicitly *export* a *subset* of the objects.

Modules

- What we really want is a *true module* with these properties:
  - It's a *collection of objects*, subroutines, variables, types, etc.
  - … that are all *visible to each other* …
  - … but are *not visible outside the module* …
  - … unless *exported*.
  - Objects on the *outside* …
  - … are *not visible inside the module* …
  - … unless *imported*.

Modules

- That's a pretty general definition.
- Import and export conventions *differ significantly* among languages that provide the module concept.
  - What's at stake is the *visibility* of the objects.
  - Their *lifetimes* are not affected.
- Pretty much every language has at least *some* concept of a module.
  - C++, C#, PHP call them *namespaces*. Ada, Java, and Perl call them *packages*. C has no construct, but *separate compilation* helps mimic modules.

Modules

- Some languages allow *exported names* to be used only in certain ways. For example,
  - Variables may be exported as *read only*.
  - Types may be exported as *opaque*.
    - Variables of that type may be declared, passed as arguments to subroutines of the module, perhaps compared or assigned, but not otherwise manipulated.

Modules

- Modules into which names must be *imported explicitly* are said to be *closed scopes*.
- Modules that do not require explicit imports are *open scopes*.
- The imports are a valuable source of information about how the module fits into its environment.
  - Dependencies on other parts of the system are *limited* and clearly *documented*.
  - The chances of a *name collision* are greatly reduced.
  - Modules are *closed scope* in, e.g., Modula (1, 2, and 3) and Haskell.
  - Most languages have *open scope* modules.

Modules

- C++ is an example of a common style, in which names are *automatically exported*, but require *qualification* with the module's name.
  - `unsigned int i = randMod::randInt();`
- As we saw before, another scope can explicitly 'import' names by employing the `using` statement.
  - `using randMod::setSeed, randMod::randInt;`
- This could be called *selectively open* modules and appears in Ada, Java, C#, Python, etc.

Modules

- Modules facilitate *abstraction* by allowing data to be made *private* to those subroutines that use them.
- The PRNG example can be extended so that more than one generator can exist.
- Here it acts as a *manager* for instances of a generator *type* that can be used thusly,

```
using randMgr::generator;
generator *g1 = randMgr::create();
generator *g2 = randMgr::create();
...
using randMgr::randInt;
int r1 = randInt(g1);
int r2 = randInt(g2);
```

C++

```
#include <time.h>

namespace randMgr {
  const unsigned int a = 48271;
  const unsigned int m = 0x7FFFFFFF;

  typedef struct {
    unsigned int seed;
  } generator;

  generator *create() {
    generator *g = new generator;
    g->seed = time( 0 );
    return g;
  }

  void setSeed( generator *g, unsigned int s ) {
    g->seed = s;
  }

  unsigned int randInt( generator *g ) {
    return g->seed = ( a * g->seed ) % m;
  }
}
```

Modules

- This is approaching the concept of *classes*.
- Classes emerged in the language Euclid, which treated a module as a *type* rather than simply a way to *encapsulate* objects.
- Reworking the PRNG example as a C++ class is straightforward.
- The class version may be used thusly,

```
randGen *g1 = new randGen();
randGen *g2 = new randGen();
...
int r1 = g1->randInt();
int r2 = g2->randInt();
```

C++

```
#include <time.h>

class randGen {
  unsigned int seed = time(0);
  const unsigned int a = 48271;
  const unsigned int m = 0x7fffffff;

public:
  void setSeed( unsigned int s ) {
    seed = s;
  }

  unsigned int randInt() {
    return seed = ( a * seed ) % m;
  }
};
```

Modules and Classes

- What's the difference between *Modules* and *Classes*?
- Primarily, *Classes* have *inheritance* and *dynamic method dispatch* and *Modules* don't.
  - *Inheritance*: New classes may be defined as *extensions* or *refinements* of existing classes. Modules are all separate from each other.
  - *Dynamic Method Dispatch*: A *child* class may *override* the definition of a *method* in a *parent* class, and the selection of the method to execute is made at run time.

Modules and Classes

- Classes are rooted in Simula 67 and were further developed in Smalltalk (and finally emerged in Euclid).
- Many, many modern languages have classes, including Eiffel, OCaml, C++, Java, C#, Python, Ruby, etc.
- Inheritance concepts also exist in languages that aren't normally thought of as object-oriented.
  - For example, Modula 3, Ada 95, Oberon, …

Modules and Classes

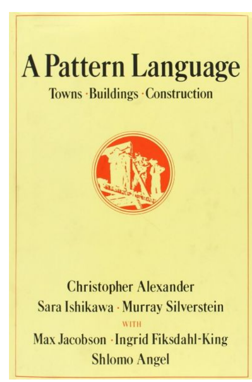
- However, classes are *not* a complete replacement for modules.
  - Classes are best when one needs *multiple instances*.
  - Modules are best when one needs *single instance functional subdivision*.
    - Some would say the *Singleton Pattern* covers this.
- Many languages recognize a place for both concepts and provide mechanisms for each.
  - C++, Java, C#, Python, Ruby, etc. have both.

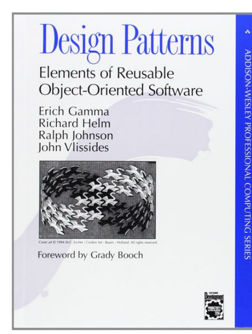
[*Software Design Patterns*]

- "… a *Software Design Pattern* ***is*** a *general, reusable* solution to a *commonly occurring problem* …"
- "It ***is not*** a *finished design* that can be transformed directly into source or machine code."
- "It ***is*** a *description* or *template* for *how to solve a problem* that can be *used in many different situations*."
- "Design patterns are *formalized best practices* … to solve *common problems* …"

from https://en.wikipedia.org/wiki/Software_design_pattern but emphasis added.

[*Software Design Patterns*]





- The idea of *design patterns* originated with Christopher Alexander (1977), an architect, and was intended as an aid in designing cities, buildings, etc.
  - The idea did ***not*** catch on in the world of architects.
  - Alexander himself has come to believe that patterns are *not enough*.
- The general concept *did* catch on in the software world and eventually resulted in the publication of *Design Patterns* in 1994.

https://commons.wikimedia.org/wiki/File:ChristopherAlexander2012.jpg
https://www.slideshare.net/VicTarchenko/adapter-designpattern2015

[*The Singleton Pattern*]

- "… the *singleton pattern* is a *software design pattern* that *restricts* the instantiation of a class *to **one** object*."
- "This is useful when *exactly one object is needed* to coordinate actions across the system."
- "The concept is *sometimes generalized* to systems that *operate more efficiently* when only one object exists, or that restrict the instantiation to a *certain number of objects*."

from https://en.wikipedia.org/wiki/Singleton_pattern but emphasis added.

[*Criticism of Software Design Patterns*]

- "The design patterns may just be a sign of some ***missing features*** of a given programming language (Java or C++ for instance)."
- "Peter Norvig demonstrates that ***16*** out of the ***23*** patterns in the *Design Patterns* book … are simplified or eliminated … in Lisp or Dylan."

Director of Research at Google; previously Director of Search Quality. Author of *Artificial Intelligence: A Modern Approach*, the most widely-used textbook for AI.

from https://en.wikipedia.org/wiki/Software_design_pattern but emphasis added.

Static (Lexical) vs. Dynamic Scoping

- With *Static* scoping rules, bindings are defined by the *Physical* (*Lexical*) structure of the program.
- With *Dynamic* scoping rules, bindings depend on the *current state of program execution*.
  - Dynamic bindings are dependent on *calling sequences* (the order in which subroutines are called), so *cannot* be definitively resolved at compile time.
  - At run time, we use the *most recent **active** binding*, not the *most recent **enclosing** binding*.

Dynamic Scoping

- Many semantic rules become *dynamic* when a language has *dynamic scoping*.
  - For example, *type checking* in expressions and argument checking in subroutine calls *usually must be deferred to run time*.
- Dynamic scoping is usually encountered in *interpreted languages*, e.g., in early Lisp ***all*** variables had *dynamic scope*.
  - Modern Lisps and Lisp dialects have *lexical scoping* rules *but* some retain *dynamic scope* for so-called *special* variables.

Scoping Example

- The program `Main` declares `A`.
- Procedures `First` and `Second` are nested inside program `Main`, so `Main`'s `A` is visible to each.
- Procedure `Second`, however, declares its own `A`, which overrides `Main`'s `A`.
- `Main` sees its own `A` and `Second` sees its own `A`.

```
program Main( input, output );
var
  A : integer;    // Main's A

  procedure First;
  begin
    A := 1;        // Which A is assigned?
  end;

  procedure Second;
  var
    A : integer;  // Second's A
  begin
    A := 2;        // Assigns to Second's A
    First;
  end;

begin
  A := 3;          // Assigns to Main's A
  Second;
  write( A );      // What value is written?
end.
```

Scoping Example

- The program `Main` declares `A`.
- Procedures `First` and `Second` are nested inside program `Main`, so `Main`'s `A` is visible to each.
- Procedure `Second`, however, declares its own `A`, which overrides `Main`'s `A`.
- `Main` sees its own `A` and `Second` sees its own `A`.
- To which `A` does `First` assign?

?

```
program Main( input, output );
var
  A : integer;    // Main's A

  procedure First;
  begin
    A := 1;        // Which A is assigned?
  end;

  procedure Second;
  var
    A : integer;  // Second's A
  begin
    A := 2;        // Assigns to Second's A
    First;
  end;

begin
  A := 3;          // Assigns to Main's A
  Second;
  write( A );      // What value is written?
end.
```

Scoping Example

- The program Main declares A.
- Procedures First and Second are nested inside program Main, so Main's A is visible to each.
- Procedure Second, however, declares its own A, which overrides Main's A.
- Main sees its own A and Second sees its own A.
- To which A does First assign?
  - Static (Lexical) scoping says Main's A.

```
program Main( input, output );
var
  A : integer;     // Main's A

  procedure First;
  begin
    A := 1;        // Which A is assigned?
  end;

  procedure Second;
  var
    A : integer;   // Second's A
  begin
    A := 2;        // Assigns to Second's A
    First;
  end;

begin
  A := 3;          // Assigns to Main's A
  Second;
  write( A );      // What value is written?
end.
```

Scoping Example

- The program Main declares A.
- Procedures First and Second are nested inside program Main, so Main's A is visible to each.
- Procedure Second, however, declares its own A, which overrides Main's A.
- Main sees its own A and Second sees its own A.
- To which A does First assign?
  - Static (Lexical) scoping says Main's A.
  - Dynamic scoping says Second's A.

```
program Main( input, output );
var
  A : integer;     // Main's A

  procedure First;
  begin
    A := 1;        // Which A is assigned?
  end;

  procedure Second;
  var
    A : integer;   // Second's A
  begin
    A := 2;        // Assigns to Second's A
    First;
  end;

begin
  A := 3;          // Assigns to Main's A
  Second;
  write( A );      // What value is written?
end.
```

Scoping Example

- The program `Main` declares `A`.
- Procedures `First` and `Second` are nested inside program `Main`, so `Main`'s `A` is visible to each.
- Procedure `Second`, however, declares its own `A`, which overrides `Main`'s `A`.
- `Main` sees its own `A` and `Second` sees its own `A`.
- To which `A` does `First` assign?
  - Static (Lexical) scoping says `Main`'s `A`.
  - Dynamic scoping says `Second`'s `A`.
- Lexical scoping results in a `1` being written.

```
program Main( input, output );
var
  A : integer;    // Main's A

  procedure First;
  begin
    A := 1;       // Which A is assigned?
  end;

  procedure Second;
  var
    A : integer;  // Second's A
  begin
    A := 2;       // Assigns to Second's A
    First;
  end;

begin
  A := 3;         // Assigns to Main's A
  Second;
  write( A );     // What value is written?
end.
```

Scoping Example

- The program `Main` declares `A`.
- Procedures `First` and `Second` are nested inside program `Main`, so `Main`'s `A` is visible to each.
- Procedure `Second`, however, declares its own `A`, which overrides `Main`'s `A`.
- `Main` sees its own `A` and `Second` sees its own `A`.
- To which `A` does `First` assign?
  - Static (Lexical) scoping says `Main`'s `A`.
  - Dynamic scoping says `Second`'s `A`.
- Lexical scoping results in a `1` being written.
- Dynamic scoping results in a `3` being written.
  - The assignment to `A` in `First` affects `Second`'s `A`, not `Main`'s `A`.

```
program Main( input, output );
var
  A : integer;    // Main's A

  procedure First;
  begin
    A := 1;       // Which A is assigned?
  end;

  procedure Second;
  var
    A : integer;  // Second's A
  begin
    A := 2;       // Assigns to Second's A
    First;
  end;

begin
  A := 3;         // Assigns to Main's A
  Second;
  write( A );     // What value is written?
end.
```

Another Scoping Example …

- Two global variables, x and y.
- Two functions, p and q.
- What happens when this runs?
- Static scoping …

  q says 1

  p says a
- Dynamic scoping …

  q says 98

  p says *

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

Another Example … Static Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

x and y are defined in the global scope.

Symbol Table

x	int	*global*
y	char	*global*

Another Example … Static Scoping

```c
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

Inside of p*, the local definition of* x *overrides the global one.*

Symbol Table

x	double	*local to* p
	int	*global*

y		
	char	*global*

Another Example … Static Scoping

```c
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

Inside of q*, the local definition of* y *overrides the global one.*

Symbol Table

x		
	int	*global*

y	int	*local to* q
	char	*global*

Another Example … Static Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

Inside of `main`*, the local definition of* `x` *overrides the global one.*

Symbol Table

x	char	*local to* `main`
	int	*global*

y		
	char	*global*

Another Example … Static Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

At run time, the binding of `x` *in* `q` *is the global one, so* `q says 1` *is printed.*

Symbol Table

x		
	int	*global*

y	int	*local to* `q`
	char	*global*

Another Example … Static Scoping

```c
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

At run time, the binding of y *in* p *is the global one, so* `p says a` *is printed.*

Symbol Table

x	double	*local to* p
	int	*global*

y		
	char	*global*

Another Example … Static Scoping

- The symbol table entries shown in the previous slides are built during *compile time*.
- These entries are used in the generation of the appropriate target code.
- For example, inside q, the global binding for x is used because that's the *most recent* ***enclosing scope*** *binding*.

```c
int  x = 1;
char y = 'a';
...
void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}
...
```

Symbol Table

x		
	int	*global*

y	int	*local to* q
	char	*global*

Another Example … Dynamic Scoping

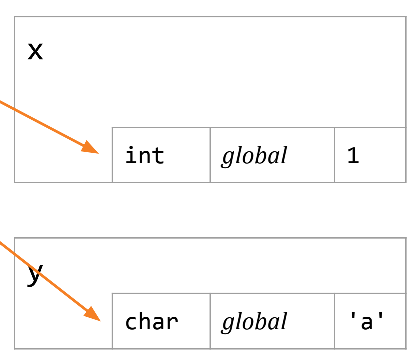
- *Static Scoping* went as expected.
  - Every reference was to the *most recent* ***enclosing scope*** *binding*.
  - All decisions about bindings could be made at *compile time*.
  - This information drove the generation of the target code.
- With *Dynamic Scoping*, binding decisions must be delayed until *run time*.
  - A symbol table must be maintained at *run time*.
  - This symbol table includes not only the name of the symbol, the type, and the scope, but also the object's *current value*.

Another Example … Dynamic Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

At run time, global bindings for x *and* y *are established.*

Symbol Table



Another Example … Dynamic Scoping

At run time, the binding of x in main overrides the global one.

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

Symbol Table

x	char	*local to* main	'b'
	int	*global*	1
y	char	*global*	'a'

Another Example … Dynamic Scoping

At run time, the binding of y in q overrides the global one.

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

Symbol Table

x	char	*local to* main	'b'
	int	*global*	1
y	int	*local to* q	42
	char	*global*	'a'

Another Example … Dynamic Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

At run time, the binding of `x` *visible in* `q` *is from* `main` *so* `q says 98` *is printed.*

Symbol Table

x			
	char	*local to* main	'b'
	int	*global*	1

y	int	*local to* q	42
	char	*global*	'a'

Q: Why is 98 printed instead of `'b'`*?*
A: The `printf` *uses the* `%d` *format effector.*

Another Example … Dynamic Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

At run time, the binding of `x` *in* `p` *overrides the one from* `main`*.*

Symbol Table

x	double	*local to* p	2.5
	char	*local to* main	'b'
	int	*global*	1

y	int	*local to* q	42
	char	*global*	'a'

Another Example … Dynamic Scoping

```
int  x = 1;
char y = 'a';

void p() {
 double x = 2.5;
 printf( "p says %c\n", y );
}

void q() {
 int y = 42;
 printf( "q says %d\n", x );
 p();
}

void main() {
 char x = 'b';
 q();
}
```

At run time, the binding of `y` *visible in* `p` *is from* `q` *so* `p says *` *is printed.*

Q: Why is `'*'` *printed instead of* `42`*?*
A: The `printf` *uses the* `%c` *format effector.*

Symbol Table

x	double	*local to* p	2.5
	char	*local to* main	'b'
	int	*global*	1
y	int	*local to* q	42
	char	*global*	'a'

Another Example … Dynamic Scoping

- Dynamic scoping laid out like this is fairly obvious.
  - Every reference was to the *most recent **active** binding*.
  - All decisions about bindings had to be made at *run time*.
- The symbol table kept not only the name of the symbol, the type, and the scope, but also the object's *current value*.
- Dynamic scoping can lead to subtle and tricky bugs, so it's not commonly used.
  - Most obvious use is supplying *implicit* parameters to functions.
  - Other, better mechanisms exist for this.
    - Default or optional parameters, for example.

Dynamic Scoping — Common Error …

- In Dynamic Scoping languages, *every* routine inherits the bindings of *every* routine in the calling sequence.
- Here, `scaledScore` intends to use the global scope `maxScore`, but because `getHighestScore` has a local variable of the same name, that binding overrides. ***Oops!***
- Just say ***No!*** to Dynamic Scoping for variables unless you're using, e.g., Common Lisp *and* you're ~~smart~~ experienced enough to handle it.

```
int  maxScore;          // Maximum possible score.

void scaledScore( int rawScore) {
  return 100.0 * rawScore / maxScore;
}

...

void getHighestScore() {
  double maxScore = 0.0;  // Highest score so far.

  foreach student in class {
    student.score = scaledScore( student.points )
    if ( student.score > maxScore ) {
      maxScore = student.score;
    }
  }

  return maxScore;
}
```

Implementing Scope

- At compile time, a *Symbol Table* is used to keep track of bindings in a Static Scoping language.
  - Every time a declaration is encountered, an entry is made for the *name* with the appropriate accompanying information.
  - Multiple entries have to be possible for a name since a binding may be hidden and then revealed. Also, a name may be used *simultaneously* in *different contexts*.
- Aside from simple variables, the symbol table must handle information about records / structures, classes, etc. (each of which is its own context).
- Symbol table information is often made available at run time to *symbolic debuggers*.
  - The symbol table must therefore maintain information about *every* name, scope, binding, etc.

Implementing Scope

- In a Dynamic Scoping language, two common methods exist for keeping track of bindings.
  - *Association List* (*A-List*): Simple and elegant, but does not scale up very well. Lookups in complex programs can be slow.
  - *Central Reference Table*: Resembles a compiler's symbol table. Requires more work on scope entry / exit than A-Lists, but makes lookup operations fast.

Association Lists for Dynamic Scoping

- An Association List (A-List) is a *linked list* of *key*, *value* pairs.
  - The key is the *name* of the binding.
  - The value includes all required information for the binding.
- When a scope is entered, the new bindings are *pushed onto* the front of the list.
- When a scope is exited, those bindings are *popped off* the list.
- When a binding is required, the system looks from the front of the list until it finds the first entry with the required name.
  - If not found, an 'unbound name' error is declared.
  - Can be *inefficient* as programs get larger and more complex.

Association List Example …

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

predefined names

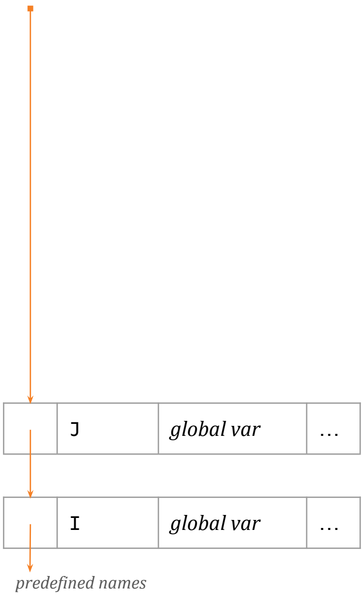
Association List Example …

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

	I	*global var*	…

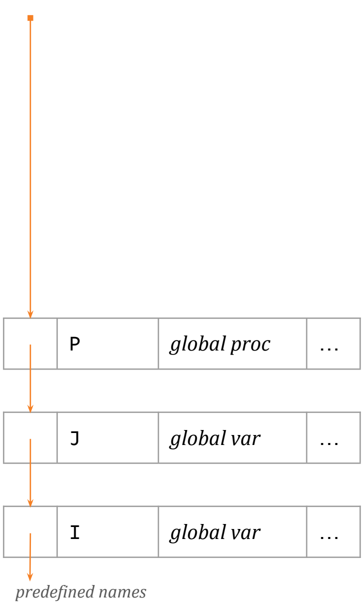
predefined names

Association List Example …



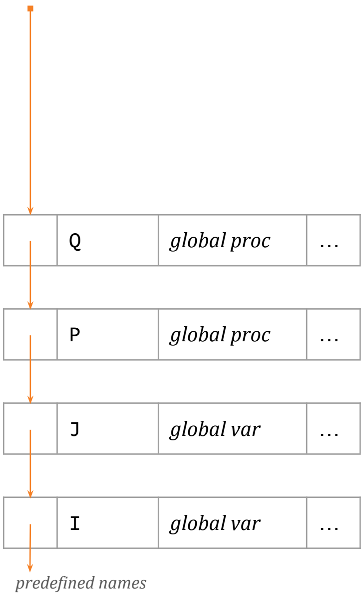

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …



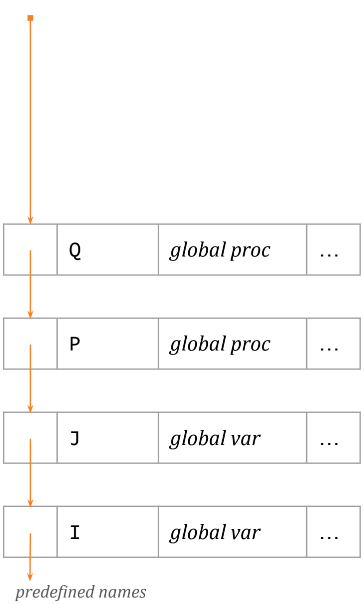

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …



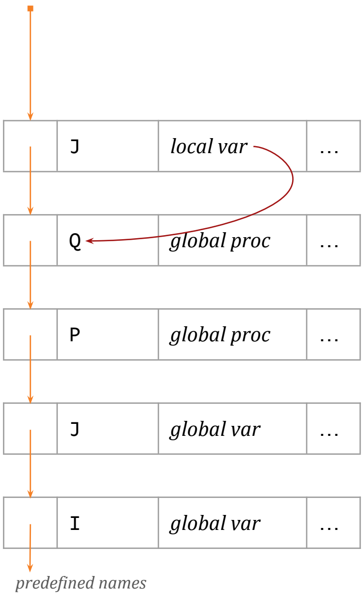

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …



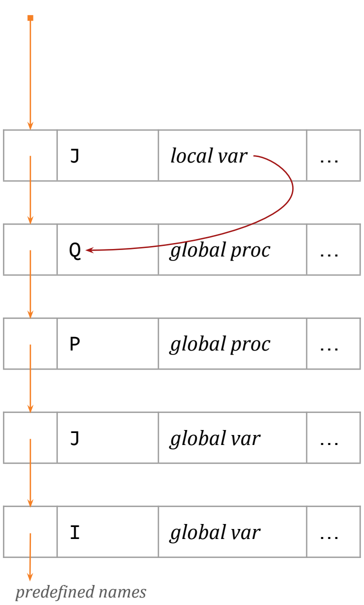

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …



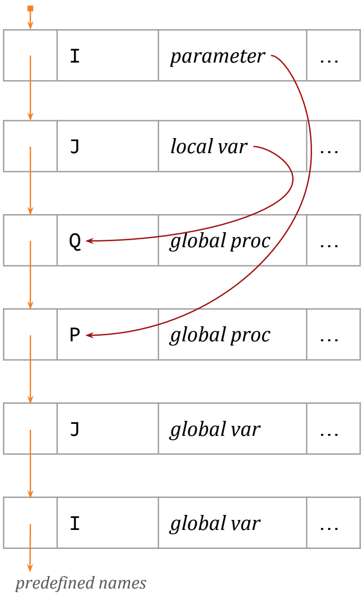

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …



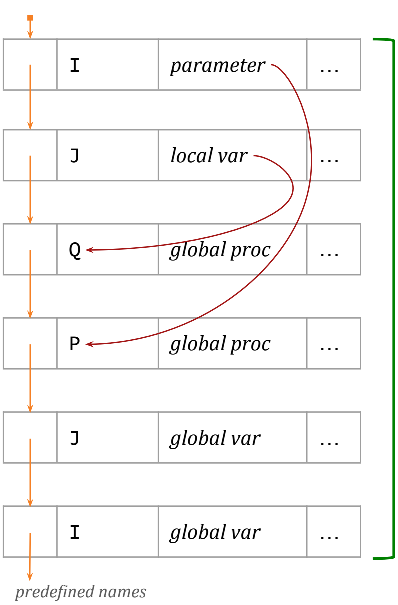

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …

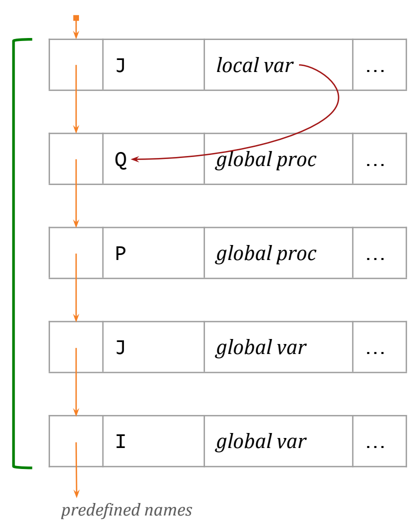



```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Association List Example …




```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```



Central Reference Table for Dynamic Scoping

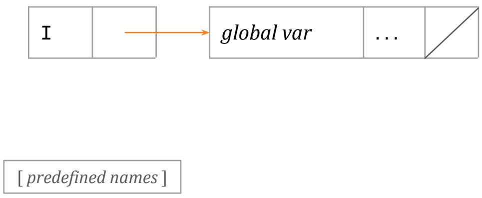
- To avoid the inefficiency of A-Lists, *Central Reference Tables* have a specific slot for each distinct name.
- Each slot is a *linked list* with the most recent binding in front.
  - Lookup is now fast because there's exactly one place to look.
- When a scope is entered, each new binding is *pushed onto* the front of the list for the name.
- When a scope is exited, those bindings are *popped off*.
  - This is a bit trickier than the A-List case since the bindings are all over the place, but not excessively so.

Central Reference Table Example …

[*predefined names*]

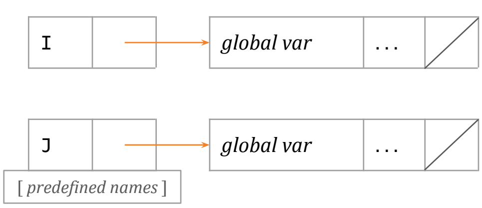
```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



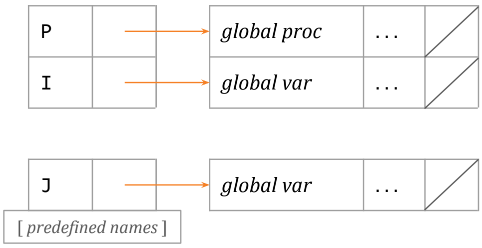

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



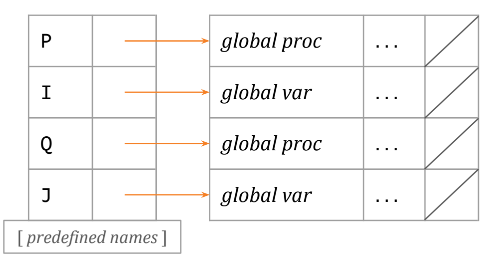

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



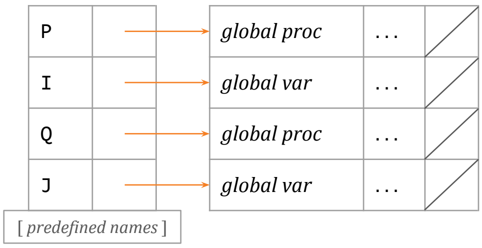

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



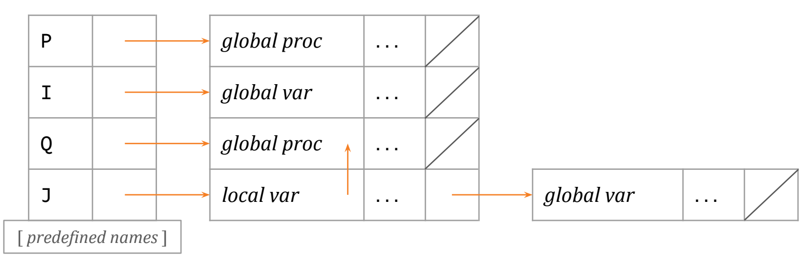

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



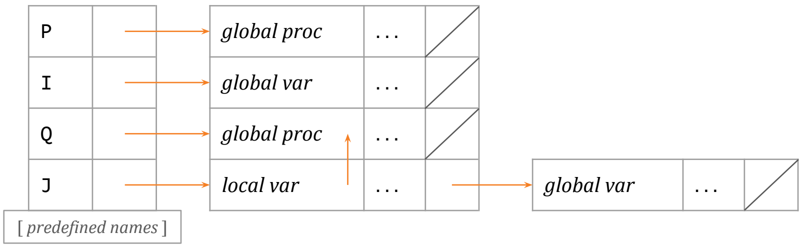

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



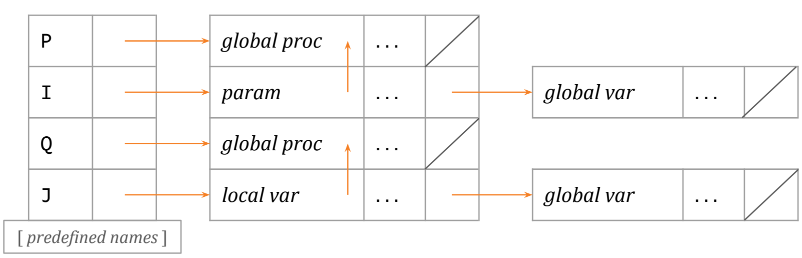

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …



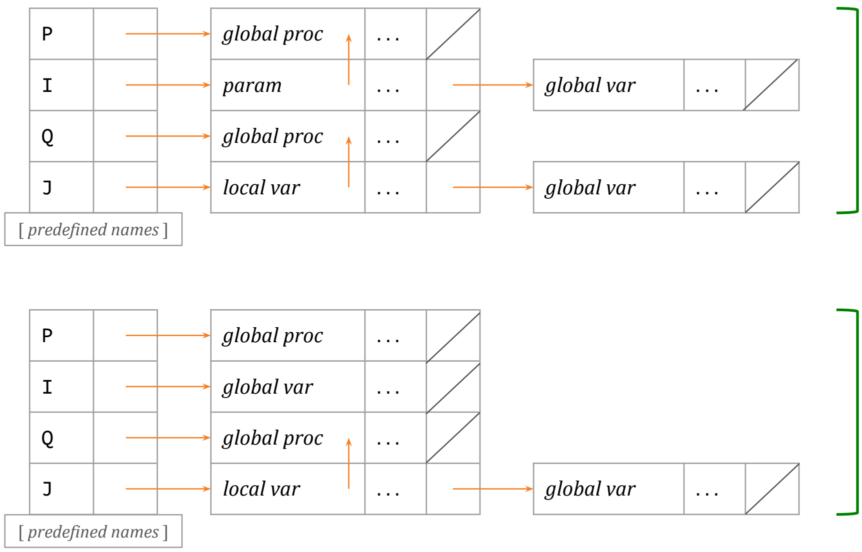

```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …




```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

Central Reference Table Example …




```
int i;
int j;

void p( int i ) {
  ...
}

void q() {
  int j;
  ...
  p( j );
}

q();
```

The Meaning of Names within a Scope

- There need *not* be a one-to-one correspondence between *names* and *objects* at any given point in a program.
  - An *object* that that has *two or more names* is said to be *aliased*.
  - A *name* that can refer to *two or more objects* is said to be *overloaded*.
- While *overloading* is a generally *useful* concept, *aliasing* can cause *inefficient code generation* or *subtle and tricky bugs* if not carefully managed.

Aliases

- Aliases are not all bad.
  - Variant record / union types exist in many languages specifically to provide a way to construct aliases.
- *Unintended* aliases are usually a source of woe.
  - Two names normally means two distinct objects.
  - Changing an object through *one name* causing a change in what is known by a *different name* is *very unexpected* (and possibly a bug).
- Compilers have to be careful when aliasing is possible.
  - Keeping a values in registers depends on the corresponding memory locations not changing.
  - If this is not guaranteed, each access has to go to / from memory.

Aliases

- Fortran introduced the `EQUIVALENCE` statement specifically to permit aliasing.
  - Originally, memory was quite limited and being able to overlay variables so that they used the same space was vital.
  - These variables were used in different phases of the program, so no problems were expected.
    - Yeah, right.
- *Reference parameters* are a common way for *unintended* aliasing to occur.
  - A routine gets as a reference parameter an object that it can *already* see in an enclosing scope. It then has *two names for the same object*.
  - Some languages have scoping rules that prevent this (e.g., Euclid).

Aliases Examples …

- In example ①, the addUp function updates two global variables.
- When main calls addUp with sum as a reference parameter, the sum and x names are bound to the same object. ***Oops!***
- In example ②, without knowing what q points at, the compiler doesn't know if what p points at changed.
  - Generally only known at run time.
  - Conservative code therefore must be generated, thus less efficient.

①
```
1. double sum = 0.0;
2. double sumOfSqr = 0.0;
3.
4. void addUp( double &x ) {
5.   sum += x;
6.   sumOfSqr += x * x;
7. }
8.
9. double aVar = 10.0;
10.
11. main () {
12.   addUp( aVar ); // OK
13.   addUp( sum );  // Not OK; aliasing
14. }
```

②
```
int a, b;
int *p, *q;

...

a = *p; // Get what p points at.
*q = 3; // Save to where q points at.
b = *p; // Did what p points at change?
```

Aliases Examples [*details*] …

- On the first call to addUp (line 12.), sum and sumOfSqr are both 0.0 and x *refers* to aVar.
- sum is increased by 10.0 since x *refers* to aVar and the value of aVar is 10.0. sum is now 10.0.
- sumOfSqr is increased by 10.0 * 10.0 since x *refers* to aVar and aVar is 10.0. sumOfSqr is now 100.0.
- On the second call to addUp (line 13.), sum is 10.0, sumOfSqr is 100.0, and x *refers* to sum.
- This time, sum is increased by 10.0 (the value of sum) and becomes 20.0 …
- … *but*, sumOfSqr is increased by 20.0 * 20.0 since x *refers* to sum and sum is now 20.0. sumOfSqr is now 500.0 instead of 200.0. ***Oops!***

```
1. double sum = 0.0;
2. double sumOfSqr = 0.0;
3.
4. void addUp( double &x ) {
5.   sum += x;
6.   sumOfSqr += x * x;
7. }
8.
9. double aVar = 10.0;
10.
11. main () {
12.   addUp( aVar ); // OK
13.   addUp( sum );  // Not OK; aliasing
14. }
```

Aliases Examples …

- In example ③, a C `union` type is used to construct aliasing on purpose.
- The same memory space is used for the `double` `dVal` as it used for the two `unsigned int`s `uiVal1` and `uiVal2`.
- By setting `dVal` to a particular value, that value's FP representation may be accessed through `uiVal1` and `uiVal2`.
- The program prints

```
(1) 0x54442D18, (2) 0x400921FB
```

- These results are *extremely* implementation dependant. ***Don't*** do this kind of thing in your own code! (Until you are *very* ~~smart~~ experienced.)

③

```c
#include <stdio.h>

int main() {
 union {
   double dVal;
   struct {
     unsigned int uiVal1;
     unsigned int uiVal2;
   }
 } var;

 var.dVal = 3.141592653589793;

 printf( "(1) 0x%08X, (2) 0x%08X\n",
   var.uiVal1, var.uiVal2 );
}
```

Overloading

- Most languages have at least *some* overloading.
  - C uses + to represent unsigned integer addition, signed integer addition, floating point addition, …
  - This seems reasonable since it's the same mathematical concept, but the underlying representations and operations are *very* different.
- Overloading is *resolved* by considering the set of *possible* bindings with respect to the *context* of the usage.
  - If the usage is *still* ambiguous, an error is declared.

Overloading Example …

- This Ada fragment declares two enumerations, `month` and `print_base`.
    - `dec` and `oct` occur in each, but that's OK with Ada.
- The context on lines `15.` and `17.` resolves the ambiguity.
- The explicit type mark on line `19.` resolves the ambiguity.
- Line `21.` lacks both suitable context and explicit type marking. ***Error!***

Ada

```
1. declare
2.   type month is
3.     ( jan, feb, mar,
4.       apr, may, jun,
5.       jul, aug, sep,
6.       oct, nov, dec );
7.
8.   type print_base is
9.     ( dec, bin, oct, hex );
10.
11.  mo : month;
12.  pb : print_base;
13.
14. begin
15.  mo := dec;  -- mo is month, so OK.
16.
17.  pb := oct;  -- pb is print_base, so OK.
18.
19.  print( month'(oct) ); -- Explicit, so OK.
20.
21.  print( oct ); -- Error!  Ambiguous.
22.
23.  ...
```

Overloading Example …

- Ada and C++ allow the overloading of subroutine names.
- The routines must differ in the number or type of their arguments.
- Much of this C++ facility carries over into C# and Java.

C++

```
1. struct complex {
2.   double real;
3.   double imaginary;
4. };
5.
6. enum base { dec, bin, oct, hex };
7.
8. void printNum( int n ) { ... }
9. void printNum( int n, base b ) { ... }
10. void printNum( complex c ) { ... }
11.
12. ...
13.
14. int     i;
15. complex c;
16.
17. printNum( i );        // Line 8's printNum
18. printNum( i, hex );   // Line 9's printNum
19. printNum( c );        // Line 10's printNum
```

Redefining Built-In Operators

- Aside from subroutine names, many language allow the overloading of built-in operators.
  - Ada, C++, C#, Fortran, …
  - Operators retain pre-defined precedence and associativity.
- Some languages allow the *creation* of operators.
  - Haskell, e.g., lets one specify the new operator's associativity and precedence.
- Some languages (e.g., Lisp) don't have operators *at all*.

C++

```cpp
1. class complex {
2.   double real;
3.   double imaginary;
4.
5. public:
6.   complex &operator+( complex &other ) { ... }
7. };
8.
9. complex &operator+( complex &a, int b ) { ... }
10.
11. int main() {
12.   complex a;
13.   complex b;
14.   complex c;
15.
16.   c = a + b;    // Uses operator+ at line 6
17.   c = a + 1;    // Uses operator+ at line 9
18. }
```

Polymorphism

- *Polymorphism* allows a single subroutine to accept arguments of multiple types.
  - For example, many languages have a print routine that can print objects of any type.
  - From the user's point of view, this print is *polymorphic* in that it accepts arguments of any type.
  - However, from the compiler's point of view, that print might be implemented with a combination of *coercion* and *overloading*.

Overloading vs. Polymorphism

- *Overloading* and *Polymorphism* are closely related topics, but they are *different*.
- In *Overloading*, we have *two or more different objects* that have the same name. For example,
  - `int norm( int a ) { return a > 0 ? a : -a; }`
  - `complex norm( complex a ) { return sqrt( a.re*a.re + a.im*a.im ) }`
- In *Polymorphism*, we have *one object* that operates in more than one way. For example,
  - `int a; complex b; print( a ); print( b );`

Coercion

- *Coercion* is the automatic conversion of a value from one type to another when required by context.
  - For example, in C, `1.23 + 1` becomes `1.23 + double( 1 )`.
  - The integer `1` has been *coerced* to floating point so the floating point addition operation may be used.
- Coercion is generally a good thing, but make sure you understand when and where it happens.
  - Coercion might be reasonable from the compiler's point of view yet unexpected by the user.

Review Questions (1)

1. What is *binding time*?
2. Explain the distinction between decisions that are bound *statically* and those that are bound *dynamically*.
3. What is the advantage of binding things as *early* as possible? What is the advantage of *delaying* bindings?
4. Explain the distinction between the *lifetime* of a name-to-object binding and its *visibility*.
5. What determines whether an object is allocated *statically*, on the *stack*, or in the *heap*?

Review Questions (2)

6. List the *objects* and *information* commonly found in a *stack frame*.
7. What is a *frame pointer*? What is it used for?
8. What is a *calling sequence*?
9. What are *internal* and *external fragmentation*?
10. What is *garbage collection*?
11. What is a *dangling reference*?
12. What do we mean by the *scope* of a name-to-object binding?
13. Describe the difference between *static* and *dynamic* scoping.

Review Questions (3)

14. What is *elaboration*?
15. What is a *referencing environment*?
16. Explain the *closest nested scope rule*.
17. What is the purpose of a *scope resolution operator*?
18. What is a *static chain*? What is it used for?
19. What are *forward references*? Why are they prohibited or restricted in many programming languages?
20. Explain the difference between a *declaration* and a *definition*. Why is the distinction important?

Review Questions (4)

21. Explain the importance of *information hiding*.
22. What is an *opaque* export?
23. Why might it be useful to distinguish between the *header* and the *body* of a module?
24. What does it mean for a scope to be *closed*?
25. Explain the distinction between *modules as managers* and *modules as types*.
26. How do *classes* differ from *modules*?

Review Questions (5)

27. Why might it be useful to have *modules* and *classes* in the same language?
28. Why does the use of *dynamic scoping* imply the need for run-time type checking?
29. Explain the purpose of a compiler's *symbol table*.
30. What are *aliases*? Why are they considered a problem in language design and implementation?
31. Explain the value of the `restrict` qualifier in C.

Review Questions (6)

32. What is *overloading*? How does it differ from *coercion* and *polymorphism*?
33. What are *type classes* in Haskell? What purpose do they serve?

[*Questions 33 to 42 are for material we are not explicitly covering in this course and are therefore omitted.*]

43. List the basic operations provided by a symbol table.

Review Questions (7)

44. Outline the implementation of a LeBlanc-Cook style symbol table.
45. Why don't compilers generally remove names from the symbol table at the ends of their scopes?
46. Describe the association list (A-list) and central reference table data structures used to implement dynamic scoping. Summarize the tradeoffs between them.
47. Explain how to implement deep binding by capturing the referencing environment A-list in a closure. Why are closures harder to build with a central reference table?