Programming Languages

Lecturer: William W.Y. Hsu
Chapter 3 - Names, Scopes, and Bindings

The Goal of this lecture is to discuss object binding and memory management.
High-Level Programming Languages

- High-Level Programming languages are defined by two characteristics:
  - Machine “independence”.
  - Ease of programming.
Machine “Independence”

• With few exceptions, the code of a high-level language can be compiled on any system.
  ▫ For example `cout<< "hello world"<< endl;`
  ▫ Means the same thing on any machine.

• However, few languages are completely machine independent.

• The more machine dependent a language is the more “efficient” it is.
Names
Names

- High level languages provide abstraction relative to the assembly languages
  - Abstraction: high level language features separate from details of computer architecture (machine independence).

- A name is a mnemonic character string used to represent something else.
  - Names are essential in high-level languages for supporting abstraction.
  - *Names* enable programmers to refer to variables, constants, operations, and types instead of low level concepts such as memory address.
Ease of Programming

- Names
- Control Flow
- Types
- Subroutines
- Object Orientation
- Concurrency
- Declarative Programming
Naming

• High level languages provide abstraction relative to the assembly languages
  ▫ Abstraction: high level language features separate from details of computer architecture (machine independence).
  ▫ By naming an object we make an abstraction.
Naming

- High level languages provide abstraction relative to the assembly languages.
  - Abstraction: high level language features separate from details of computer architecture (machine independence).
- A name is a mnemonic character string used to represent something else.
  - Names are essential in high-level languages for supporting abstraction.
  - *Names* enable programmers to refer to variables, constants, operations, and types instead of low level concepts such as memory address.
Name and Abstraction

- **Control abstractions** allows programs to “hide” complex code behind simple interface
  - Subroutines (procedures and functions) allow programmers to focus on manageable subset of program text, subroutine interface hides implementation details.
  - **Control flow constructs** (if-then, while, for, return) hide low-level machine ops.

- **Data abstraction** allow the programmer to hide data representation details behind abstract operations
  - Abstract Data Types (ADTs).
  - Classes. Object-oriented classes hide data representation details behind a set of operations.

- Enhances a level of machine-independence.
Name Related Issues

- Binding time: A *binding* is an association between a *name* and an *entity*.
  - Name of an object and the object.
  - When is a name bound to the object it represents?
    - A Dook student to a loosing basketball team.
  - Some binding is done at language design time (design decision).
- The lifetime of object and lifetime of binding determine the storage mechanisms for the objectives.
- Scoping rules: the region that a binding a used.
- Alias: multiple names are bound to the same object
Binding Time

- **Binding time** is the time at which the binding between a name and an object is created.
- Depending on the names, binding may happen in many different times.
- **Potential binding time includes:**
  - **Language design time**: the design of specific language constructs.
    - Example: The primitive data type in C++ include `int`, `char`, `float`, `double`.
  - **Language implementation time**: fixation of implementation constants.
    - Example: C++ float point type uses IEEE 754 standard.
  - **Program writing time**: the programmer's choice of algorithms and data structures
    - Example: a routine is called `foo()`.
Binding Time

- **Compile time**: the time of translation of high-level constructs to machine code and choice of memory layout for data objects.
  - Example: translate “for (i=0; i<100; i++) a[i] = 1.0;”?

- **Link time**: the time at which multiple object codes (machine code files) and libraries are combined into one executable
  - Which cout routine to use? /usr/lib/libc.a or /usr/lib/libc.so?

- **Load time**: when the operating system loads the executable in memory
  - In an older OS, the binding between a global variable and the physical memory location is determined at load time.

- **Run time**: when a program executes
  - Binding between the value of a variable to the variable.
More Binding Time Examples

- **Language design:**
  - Syntax (names ↔ grammar)
    - if (a>0) b:=a; (C syntax style)
    - if a>0 then b:=a end if (Ada syntax style)
  - Keywords (names ↔ builtins)
    - class (C++ and Java), endif or end if (Fortran, space insignificant)
  - Reserved words (names ↔ special constructs)
    - main (C), writeln (Pascal)
  - Meaning of operators (operator ↔ operation)
    - + (add), % (mod), ** (power)
  - Built-in primitive types (type name ↔ type)
    - float, short, int, long, string

- **Language implementation**
  - Internal representation of types and literals (type ↔ byte encoding)
    - 3.1 (IEEE 754) and "foo bar" (\0 terminated or embedded string length)
  - Storage allocation method for variables (static/stack/heap)
The Effect of Binding Time

- *Early binding times* (before run time) are associated with greater efficiency and clarity of program code
  - Compilers make implementation decisions at compile time (avoiding to generate code that makes the decision at run time).
  - Syntax and static semantics checking is performed only once at compile time and does not impose any run-time overheads.

- *Late binding times* (at run time) are associated with greater flexibility (but may leave programmers sometimes guessing what’s going on)
  - Interpreters allow programs to be extended at run time.
  - Languages such as Smalltalk-80 with polymorphic types allow variable names to refer to objects of multiple types at run time.
  - Method binding in object-oriented languages must be late to support *dynamic binding*. 
Object Lifetimes

- Object lifetimes have two components:
  - Lifetime of the object.
  - Lifetime of the binding.
- These two don’t necessarily correspond.
  - For example in C++, when a variable is passed by “reference”, i.e., using “&”, then the name of the object does not exist even though the binding does.
  - For example in C++, when the value pointed to by an object is deleted the binding is gone before the object.
Binding Lifetime versus Object Lifetime

- Key events in object lifetime:
  - Object creation
  - Creation of bindings
  - The object is manipulated via its binding
  - Deactivation and reactivation of (temporarily invisible) bindings
  - Destruction of bindings
  - Destruction of objects

- **Binding lifetime**: time between creation and destruction of binding to object.
  - Example: a pointer variable is set to the address of an object.
  - Example: a formal argument is bound to an actual argument.

- **Object lifetime**: time between creation and destruction of an object.
Binding Lifetime versus Object Lifetime

- Bindings are temporarily invisible when code is executed where the binding (name ↔ object) is out of scope.
A C++ Example

```cpp
{  
    SomeClass* myobject = new SomeClass;
    ...
    {
        OtherClass myobject;
        ... // the myobject name is bound to other object
        ... // this class provides the method action().
        ... // action() allocates some memory.
    }
    ... // myobject binding is visible again
    ...
    myobject->action() // myobject in action():
                        // the name is not in scope
                        // but object is bound to `this`
    delete myobject;
    return;
}
```
Problems with Object/Binding Lifetime Mismatch

- Memory leak: the binding is destroyed while the object is not, making it noway to access/delete the object

```cpp
{
    SomeClass* myobject = new SomeClass;
    ...
    ...
    myobject->action();
    return;
}
```
Problems with Object/Binding Lifetime Mismatch

- **Dangling reference**: object destroyed before binding is destroyed.

```cpp
... myobject = new SomeClass;
foo(myobject);

Foo(SomeClass *a)
{
    ....
    delete (myobject); // myobject is a global variable
    a->action();
}
```
Problems with Object/Binding Lifetime Mismatch

```c
char *ptr;   // a global variable

foo() {
    char buff[1000];
    cin >> buff;
    ptr = strtok(buff, " ");  ...... 
}

main()  {
    ...
    foo();
    cout << ptr; ...... 
}
```
Object Lifetimes

- Object Lifetimes correspond to three principal storage allocation mechanisms:
  - **Static** objects, which have an **absolute address**.
  - **Stack** objects, which are allocated and deallocated in a **Last-In First-Out (LIFO)**.
  - **Heap** objects, which are allocated and deallocated at **arbitrary times**.
Object Lifetime and Storage

- Objects (program data and code) have to be stored in memory during their lifetime.
- In general there are three types of objects in a program.
  - The objects that are alive throughout the execution of a program.
    - E.g. global variables
  - The objects that are alive within a routine.
    - E.g. local variables
  - The objects whose lifetime can be dynamically changed.
    - The objects that are managed by the ‘new/delete’ constructs.
Object Lifetime and Storage

- The three types of objects correspond to three principal storage allocation mechanisms.
  - *Static objects* have an absolute storage address that is retained throughout the execution of the program.
    - Global variables and data.
    - Subroutine code and class method code.
  - *Stack objects* are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns.
    - Actual arguments passed by value to a subroutine.
    - Local variables of a subroutine.
  - *Heap objects* may be allocated and deallocated at arbitrary times, but require an expensive storage management algorithm.
    - Dynamically allocated data in C++.
    - Java class instances are always stored on the heap.
Typical Program and Data Layout in Memory

- Program code is at the bottom of the memory region (code section).
  - The code section is protected from run-time modification by the OS.
- Static data objects are stored in the static region.
- Stack grows downward.
- Heap grows upward.
Static Allocation

- Under **static allocation**, objects are given an absolute address that is retained through the program’s execution. (i.e., global variables)
Static Allocation

- Program code is statically allocated in most implementations of imperative languages.
- Statically allocated variables are history sensitive.
  - Global variables keep state during entire program lifetime.
  - Static local variables in C/C++ functions keep state across function invocations.
  - Static data members are “shared” by objects and keep state during program lifetime.
- Advantage of statically allocated object is the fast access due to absolute addressing of the object.
  - Can static allocation be used for local variables?
    - Statically allocated local variables has only one copy of each variable. Cannot deal with the cases when multiple copies of a local variable are alive!!!
Static Allocation in Fortran 77

- Fortran 77 has no recursion.
- Global and local variables are statically allocated as decided by the compiler.
- Global and local variables are referenced at absolute addresses.
- Avoids overhead of creation and destruction of local objects for every subroutine call.
- Each subroutine in the program has a subroutine frame that is statically allocated.
- This subroutine frame stores all subroutine-relevant data that is needed to execute.

Typical static subroutine frame layout:

- Temporary storage (e.g. for expression evaluation)
- Local variables
- Bookkeeping (e.g. saved CPU registers)
- Return address
- Subroutine arguments and returns
Stack Allocation

- Under **stack-based allocation**, objects are allocated in a **Last-In First-Out (LIFO)** basis called a stack. (i.e., recursive subroutine parameters)
Stack Allocation

- Each instance of a subroutine that is active has a subroutine frame (sometimes called activation record) on the run-time stack.
  - Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards.
  - Method invocation works the same way, but in addition methods are typically dynamically bound.

- Subroutine frame layouts vary between languages, implementations, and machine platforms.
Typical Stack-Allocated Subroutine Frame

- Most modern processors have two registers: fp (frame pointer) and sp (stack pointer) to support efficient execution of subroutines in high level languages.
- A frame pointer (fp) points to the frame of the currently active subroutine at run time.
- Subroutine arguments, local variables, and return values are accessed by constant address offsets from the fp.

Typical static subroutine frame layout:

- Temporary storage (e.g. for expression evaluation)
- Local variables
- Bookkeeping (e.g. saved CPU registers)
- Return address
- Subroutine arguments and returns

Lower addr: fp → Higher addr
Subroutine Frames on the Stack

Subroutine frames are pushed and popped onto/from the runtime stack.

The stack pointer (sp) points to the next available free space on the stack to push a new frame onto when a subroutine is called.

The frame pointer (fp) points to the frame of the currently active subroutine, which always the topmost frame on the stack.

The fp of the previous active frame is saved in the current frame and restored after the call.

In this example:
- M called A
- A called B
- B called A
Example Subroutine Frame

- The size of the types of local variables and arguments determines the fp offset in a frame
- Example Pascal procedure:

```pascal
procedure P(a:integer,
    var b:real)
(* a is passed by value
 b is passed by reference,
 = pointer to b's value *)
var
    foo:integer;(* 4 bytes *)
    bar:real;  (* 8 bytes *)
    p:^integer; (* 4 bytes *)
begin
    ...
end
```
Summary on Naming

- Names are essential in high level languages to support abstraction.
- Supporting names involve many issues.
- Binding time: the time when the binding between a name and an objective is created.
- Potential binding times.
- Lifetime of objects and bindings.
Calling Sequence

- On procedure call and return compilers generate code that execute to manage the runtime stack.
  - **Setup** at call to procedure `foo(a, b)`.
  - **Prologue** before `foo` code executes.
  - **Epilogue** at the end of `foo` code.
  - “**Teardown**” right after calling the code.
Setup $\text{foo}(a,b)$

1. Move $sp$ to allocate a new stack frame.
2. Copy args $a,b$ into frame.
3. Copy return address into frame.
4. Set $fp$ to point to new frame.
5. Maintain static chain or display.
6. Move PC to procedure address.
Setup $\text{foo}(a,b)$

1. **Move sp** to allocate a new stack frame.
2. **Copy args** $a, b$ into frame.
3. **Copy return** address into frame.
4. **Set fp** to point to new frame.
5. **Maintain static chain** or display.
   - If the callee is nested inside the caller, then the callee’s static link should refer to the caller’s frame.
6. **Move PC** to procedure address.
   - This changes where the code is executed.
Prologue

1. **Copy registers** into local slots
2. **Object initialization.**
   - Objects that are used are initialized.
Epilogue

1. **Place return value** into slot in frame.
2. **Restore registers.**
   - Registers stored from “foo”'s subroutine are registered.
3. **Restore PC** to return address.
   - The program resumes from where it began.
“Teardown”

1. **Move sp & fp** (deallocate frame)
2. **Move return** values (if in registers)
   - If the return value was placed in a register, put it in the stack.
Heap Allocation

- Heap is used to store objects who lifetime is dynamic
  - Implicit heap allocation:
    - Done automatically
    - Java class instances are placed on the heap
    - Scripting languages and functional languages make extensive use of the heap for storing objects
    - Some procedural languages allow array declarations with run-time dependent array size
    - Resizable character strings
  - Explicit heap allocation:
    - Statements and/or functions for allocation and deallocation
    - Malloc/free, new/delete
Heap-based Allocation and Management

- In heap-based allocation, objects may be allocated and deallocated at arbitrary times.
  - For example, objects created with C++ `new` and `delete`.

- In general, the heap is allocated sequentially.
- This creates fragmentation...
Internal Fragmentation

- Internal fragmentation is caused when extra space within a single block is unused.
  - Caused by **fixed block size**.
External Fragmentation

- External fragmentation occurs when there is sufficient available space for a new object, but there is no single block of free space large enough.
Heap Allocation Problem

- Heap is a large block of memory (say $N$ bytes)
- Requests for memory of various sizes may arrive randomly: program runs ‘new’
  - Each request may ask for 1 to $N$ bytes
- If a request of $X$ bytes is granted, a continuous $X$ bytes in the heap is allocated for the request.
  - The memory will be used for a while and then return to the system (when the program runs ‘delete’).
- The problem: how to allocate memory so that as many request can be satisfied as possible.
Heap Allocation Problem

• Example

10KB memory to be managed
r1 = req(1K);
r2 = req(2K);
r3 = req(4k);
free(r2);
free(r1);
r4 = req(4k);

• How to do it makes a difference!!

• Internal fragment: unused memory within a block
  ▫ Asking for 100 bytes and get a 512 bytes block

• External fragment: unused memory between blocks
  ▫ Even when the total available memory is more than a request, the request cannot be satisfied as in the example.
Heap Allocation Algorithms

- Heap allocation is performed by searching the heap for available free space.
- For example, suppose we want to allocate a new object E of 20 bytes, where would it fit?

<table>
<thead>
<tr>
<th>Object A</th>
<th>Free</th>
<th>Object B</th>
<th>Object C</th>
<th>Free</th>
<th>Object D</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 bytes</td>
<td>8 bytes</td>
<td>10 bytes</td>
<td>24 bytes</td>
<td>24 bytes</td>
<td>8 bytes</td>
<td>20 bytes</td>
</tr>
</tbody>
</table>

- Deletion of objects leaves free blocks in the heap that can be reused.
- Two questions:
  - How to keep track of free blocks?
  - How to find the right free block for each request?
Heap Allocation Algorithms

- How to keep track of free blocks?
  - Maintain a linked list of free heap blocks
  - To satisfy a request (new), search the list to find one block whose size is equal to or larger than the requested size
    - If the size is equal, remove the block from the free list
    - If the size is larger, modify the size to (size – requested size).
  - When an object is deleted (freed), the block of memory is returned to the heap.
    - Insert a new block to the free list. If new block can be merged with its neighboring blocks to be a larger block, merge the blocks.
Heap Allocation Algorithms

- How to pick which block to be used for each request?
  - There can be many choices.
  - **First-fit**: select the first block in the list that is large enough.
  - **Best-fit**: search the entire list for the smallest free block that is large enough to hold the object.

- $O(N)$ for both ‘new’ and ‘delete’ operations
Other Heap Allocation Algorithms

- **Buddy system**: use heap pools of standard sized blocks of size $2^k$
  - If no free block is available for object of size between $2^{k-1}+1$ and $2^k$ then find block of size $2^{k+1}$ and split it in half, adding the halves to the pool of free $2^k$ blocks, etc.

- **Fibonacci heap**: use heap pools of standard size blocks according to Fibonacci numbers.
  - More complex but leads to slower internal fragmentation
Heap Management

- Some languages (C & C++) require explicit heap management.
  - In C, malloc and free
  - In C++, new and delete
- Easy to forget free/delete!
  - Called a memory leak!
Heap Management

- Some languages (Java/C#) manage the heap for you.
  - `new` object allocated on heap.
  - When done, object is reclaimed.
- Automatic de-allocation after an object has no binding/references is called garbage collection.
  - Some runtime efficiency hit.
  - No memory leaks.
Garbage Collection

- Explicit manual deallocation errors are among the most expensive and hard to detect problems in real-world applications
  - If an object is deallocated too soon, a reference to the object becomes a dangling reference
  - If an object is never deallocated, the program leaks memory
- Automatic garbage collection removes all objects from the heap that are not accessible, i.e. are not referenced
  - Used in Lisp, Scheme, Prolog, Ada, Java, Haskell
  - Disadvantage is GC overhead, but GC algorithm efficiency has been improved
  - Not always suitable for real-time processing
Garbage Collection

How does it work roughly?

- The language defines the lifetime of objects.
- The runtime keeps track of the number of references (bindings) to each object.
  - +1 when a new reference is made, -1 when the reference is destroyed
- Can delete when the reference count is 0.
- Need to determine when a variable is alive or dead based on language specification.
Sample Memory Layout
Summary on Storage Management

- **Object storage management**
  - **Static allocation**
    - What type of objects are allocated with static allocation?
    - Why not static allocation for everything?
  - **Stack allocation**
    - What type of objects are allocated with stack allocation?
  - **Heap allocation**
    - What type of objects are allocation with heap allocation?
  - **Garbage collection**
Scope
Scope

- Scope is the textual region of a program in which a name-to-object binding is active.
- Programming languages implement.
  - Static Scoping (or lexical): Active bindings are determined using the text of the program at compile time.
    - Used by almost all but a few programming languages.
    - More intuitive to user compared to dynamic scoping.
    - Most recent scan of the program from top to bottom.
    - Closes nested subroutine rule.
  - Dynamic Scoping: active bindings are determined by the flow of execution at run time
    - Used in Lisp (early versions), APL, Snobol, and Perl (selectively).
- Current active binding called Referencing environment.
Effect of Static Scoping

- The following pseudo-code program demonstrates the effect of scoping on variable bindings:

```plaintext
a: integer

procedure first
  a := 1
procedure second
  a: integer
  first()
procedure main
  a := 2
  second()
write_integer(a)
```

Program execution:

```plaintext
a: integer
main()
  a := 2
  second()
    a: integer
    first()
      a := 1
write_integer(a)
```

Program prints “1”
Effect of Dynamic Scoping

The following pseudo-code program demonstrates the effect of scoping on variable bindings:

```
a:integer
procedure first
  a:=1  Binding depends on execution
procedure second
  a:integer
  first()
procedure main
  a:=2
  second()
write_integer(a)
```

Program execution:

```
a:integer
main()
  a:=2
  second()
    a:integer
      first()
        a:=1
    write_integer(a)
```

Program prints “2”
Static Scoping

- The bindings between names and objects can be determined by examination of the program text
- *Scope rules* of a program language define the scope of variables and subroutines, which is the region of program text in which a name-to-object binding is usable
  - Early Basic: all variables are global and visible everywhere.
  - Fortran 77: the scope of a local variable is limited to a subroutine; the scope of a global variable is the whole program text unless it is hidden by a local variable declaration with the same variable name.
  - Algol 60, Pascal, and Ada: these languages allow nested subroutines definitions and adopt the *closest nested scope rule* with slight variations in implementation.
Closest Nested Scope Rule

To find the object referenced by a given name:

- Look for a declaration in the current innermost scope
- If there is none, look for a declaration in the immediately surrounding scope, etc.
Static Scope Implementation with Static Links

- With the Closest Nested Scope Rule, the program can only refer to variables that are alive: the variable must have been stored in the frame of a subroutine.
- If a variable is not in the local scope, we are sure there is a frame for the surrounding scope somewhere below on the stack:
  - The current subroutine can only be called when it was visible
  - The current subroutine is visible only when the surrounding scope is active
- Each frame on the stack contains a static link pointing to the frame of the static parent.
Nest Subroutines

- Nest subroutines are able to access parameters and local variables of the surrounding scope.

```plaintext
procedure P1(A1);
    var X : real;
    procedure P2(A2);
        procedure P3(A3);
            X = 2;
        end
    end
end
end
```
Nested Subroutines--Determining Scope

- Subroutines C and D are declared nested in B.
  - B is static parent of C and D.
- B and E are nested in A.
  - A is static parent of B and E.
- The fp points to the frame at the top of the stack to access locals.
- The static link in the frame points to the frame of the static parent.
Nested Subroutines--Determining Scope
Dynamic Scope

- Scope rule: the "current" binding for a given name is the one encountered most recently during execution.
- Typically adopted in (early) functional languages that are interpreted.
- With dynamic scope:
  - Name-to-object bindings cannot be determined by a compiler in general.
  - Easy for interpreter to look up name-to-object binding in a stack of declarations.
- Generally considered to be "a bad programming language feature".
  - Hard to keep track of active bindings when reading a program text.
  - Most languages are now compiled, or a compiler/interpreter mix.
Dynamic Scope

- Bindings between name and objects depend on the flow of control at run time.
- The current binding is the one found most recently during execution.

```plaintext
a:int;

procedure first()
a:=1

procedure second()
a:int
  first()

a:=2
if read_int()>0
  second()
else
  first()
write_int(a)
```
Dynamic Scoping Problems

• In this example, function `scaled_score` probably does not do what the programmer intended: with dynamic scoping, `max_score` in `scaled_score` is bound to `foo`'s local variable `max_score` after `foo` calls `scaled_score`, which was the most recent binding during execution:

```plaintext
max_score:integer
function scaled_score(raw_score:integer):real
    return raw_score/max_score*100

procedure foo
    max_score:real := 0

    foreach student in class
        student.percent := scaled_score(student.points)
        if student.percent > max_score
            max_score := student.percent
```
Dynamic Scope Implementation with Bindings

Stacks

- Each time a subroutine is called, its local variables are pushed on a stack with their name-to-object binding.
- When a reference to a variable is made, the stack is searched top-down for the variable's name-to-object binding.
- After the subroutine returns, the bindings of the local variables are popped.
- Different implementations of a binding stack are used in programming languages with dynamic scope, each with advantages and disadvantages.
Perl and Dynamic Scope

- Perl allows dynamic scope.
- If not declared otherwise, variables are dynamically created, global, and persistent.
  - **Dynamic** creation: Variables appear when referenced.
  - **Global**: Variables can be referenced in any and all code written
  - **Persistent**: Variables stay around until end of execution.
Per1 and Dynamic Scope

\$a = 1;

aFunc();

\$d = \$b+\$c;

#\$d = 1 + 3 = 4

sub aFunc{
    \$b=\$a;
    \$c=3;
}

sub aFunc{
    \$a = 1;
    bFunc();
}

sub bFunc{
    \$c = \$a;
    #\$c = 1 if run in or after aFunc
}

Perl

- “my $abc”
  - Makes variable statically scoped.
  - Only available to this subroutine.
  - Not available to called subroutines or originating subroutines.
  - Destroyed when execution exists the block its in.
Perl and Dynamic Scope (my)

```perl
$a = 1;
aFunc();
$d = $b+$c;
#$d = undefined + undefined = 0

sub aFunc{
    my($b, $c)
    $b=$a;
    $c=3;
}

sub bFunc{
    $c = $a;
    #$c is undefined no matter if it is run at or in bFunc
}
sub aFunc{
    my($a);
    $a = 1;
    bFunc();
}
```

Perl

- "local $var"
  - Makes variable dynamically scoped.
  - "Temporary global".
  - Available to called subroutines, but not available to originating subroutines.
  - Destroyed when execution exists current block.
Perl and Dynamic Scope (local)

```perl
$a = 1;
aFunc();
$d = $b+$c;
#$d = undefined + undefined = 0

sub aFunc{
    local($b, $c)
    $b=$a;
    $c=3;
}

sub bFunc{
    $c = $a;
    # $c is undefined if bFunc is run after aFunc, but is 1 if run in $aFunc
}
```

```perl
$aFunc();
sub bFunc{
    $c = $a;
    # $c is undefined if bFunc is run after aFunc, but is 1 if run in $aFunc
}
```

2016/10/15
Lifetime vs Scope

- Some objects exist only when scope is active.
  - However, this is not always the case.

```java
class foo{
    public static int sum = 0;
    void vooDo(){ sum ++; }
}

//Where is sum? Its not active but it exists.
g1 = new foo;
g1.vooDo();
```
Static Chain

- For finding non-local bindings at run-time.
- How do we access non-local objects?
- The static links form a static chain, which is a linked list of static parent frames.
- When a subroutine at nesting level \( j \) has a reference to an object declared in a static parent at the surrounding scope nested at level \( k \), then \( j-k \) static links forms a static chain that is traversed to get to the frame containing the object.
- The compiler generates code to make these traversals over frames to reach non-local objects.
- Each frame contains a static chain pointer (SCP), a pointer to the most recent frame on the next lexical level out.
Nested Subroutines--Determining Scope

- Subroutine A is at nesting level 1 and C at nesting level 3.
- When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object.
Nested Subroutines--Determining Scope

Static chain pointers.
Nested Subroutines--Determining Scope

- If in C we used a variable X declared in A, then we would have two hops.
- These hops are known from the symbol table.
- The problem is that at run time this can require n hops.
Display

- The **display** is a small array that replaces the static chain, where the **jth element of the display contains a pointer to the jth nesting level**.
- The display is **faster at run time than static chain**, but requires a **little more work** when entering and leaving scope levels.
Dynamic Chain

- **Dynamic Chain Pointer (DCP).**
- Shows sequence of stack frames in dynamic (call) order.
- Allows implementation of dynamic scope.

- Will be discussed later.
Static Scope: Modules

- Many modern languages are more complicated in their scope rules than PASCAL and C.
- Modules are a means to explicitly manipulate scopes and names visibility.
  - Namespaces in C++ are modules.
- They are not nested in general.
- Objects inside a module can see each other (subject to normal lexical scoping).
- Objects outside...able to see in?
namespace fooSpace{
    int bar;
}

void main(){
    bar = 3;    //WRONG!!!
    fooSpace.bar = 3;  //RIGHT!!!
}
Module as Manager & as Type

- Two ways to view a module:
  - Module-as-manager means that the module acts as a collection of objects.
    - Namespaces in C++
  - Module-as-type means that the module acts an object type that can be a collection of objects.
    - Classes in C++. 
Module as Manager & as Type

namespace fooSpace{
    int bar;
}

void main(){
    fooSpace.bar = 3;
}

class fooClass{
public:
    int bar;
}

id main(){
    fooClass qud,zod;
    qud.bar = 3;
    zod.bar = 4;
}
Import/Export

- Objects in a module are not visible outside unless **exported**.
  - In C++ classes, objects are exported via “**public**”.
- Objects outside are not visible inside the module unless **imported**.
  - C++ classes & namespaces, objects are imported via “.” as in “**namespacename.varible**”
- Bindings made in a module are **inactive outside**, but **not gone**.
Open Scope vs. Closed

- **Open scope**: Names **do not have to imported explicitly** to be visible.
  - Nested subroutines in Pascal.
  - We can see the names in outer lexical scopes without having to ask for the ability.
- **Closed scope**: Names **must be imported explicitly** to be visible.
  - Modules in C++, Perl, etc...
Open Scope vs. Closed

```plaintext
sub foo()
  a : int;

  sub bar()
    a = 2
  end
end

namespace fooSpace{
  int bar;
}

void main(){
  fooSpace.bar = 3;
}
```
Referencing in Modules

- We need a **more complicated symbol table** to generate code for non-local referencing at run-time.
- Seeing a new name during parsing makes several things happen.
  - Scopes are **counted and numbered serially**.
  - Nesting level is also counted implicitly: **scope stack**.
Scope Stack

type T = record F1:int; F2:real; end;
Var V:T;
Module M;
  export I; import V;
  var I : int;
proc P1(A1:real, A2:int):real
  END-P1
proc P2(A3:real);
  var I: int;
  with V DO... END;
END-P2;
END-M;
Scope Stack

- A **scope stack** indicates the **order and scopes** that compose the current referencing environment.
Scope Stack

A2 | para | 4 | (1) | XXXXX

F2 | field | 2 | (2) | XXXXX

X | T | type | 5 | X | Record 2

V | var | 3 | X | import

V | var | 1 | XXX

Table:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Closed?</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>rec v</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Global</td>
</tr>
</tbody>
</table>

with v
P2
M
Global
When a Name is Seen (Parsing)

• When a name is seen during parsing:
  ▫ If it’s a **declaration** -- hash name and create new entry.
  ▫ If it’s a **new scope** -- push onto scope stack.
  ▫ If it’s a **reference** -- look up, then scan down the scope stack to see if
    the scope of the name is visible.
  ▫ If it’s a **module** -- begin making new entries for the imported names.

• When a name is looked up:
  ▫ Hash the name in the table to get entry.
  ▫ Hops... stack depth - level where name’s scope is found on.
Binding Within a Scope

- **Aliasing**: two names refer to single objects.
- What are aliases good for? (Absolutely nothing? No!!)
  - Space saving.
  - Linked data structures.
- Also, aliases arise in parameter passing as an unfortunate side effect.

```c
double sum, sum_squares;
void acc(double &x){
    sum +=x;
    sum_squares +=x*x;
}
acc(sum);
```

Since x is passed by reference, this adds the value to sum, then takes the new value and squares that!
Binding Within a Scope

- Overloading
  - Some overloading happens in almost all languages.
    - integer + v. real +
    - read and write in Pascal.
    - function return in Pascal.
  - Some languages get into overloading in a big way. For example, C++.
Out of Scope

- Non-local objects can be *hidden* by local name-to-object bindings and the scope is said to have a hole in which the non-local binding is temporarily inactive but not destroyed.
- Some languages, notably Ada and C++ use qualifiers or scope resolution operators to access non-local objects that are hidden.
  - P1.X in Ada to access variable X of P1 and ::X to access global variable X in C++.
Out of Scope Example

• P2 is nested in P1.
• P1 has a local variable X.
• P2 has a local variable X that hides X in P1.
• When P2 is called, no extra code is executed to inactivate the binding of X to P1.

```plaintext
procedure P1;
var X:real;
    procedure P2;
    var X:integer
    begin
        ... (* X of P1 is hidden *)
    end;
    begin
        ...
    end
end
```
Symbols
Symbol Table

- In statically scoped languages, compilers keep track of names using a data structure called a symbol table.
- The symbol table might be retained after compiling and made available at runtime. (e.g., for debugging)
- Maps names to info about objects. Just like a hash in Perl!
Symbol Table: Simplified

- Seeing a new name during parsing makes several things happen.
  1. `addName` to the symbol table.
  2. Is the name a new scope? `addScope`  
     - New Scopes: Procedure/method names, nested blocks....
  3. Nesting Levels (Lexical level) is counted as parsing goes.
  4. Each Name is stored with its `scope number`.

- Compiler keeps track of the lexical level in force when a name is declared.

- Multiple entries are made for a name in the hash table.
  - A new inner declaration “hides” an outer declaration.
Sample Program

```
proc sum(int x){
    int k = 0;
    proc foo(){
        real sum = 0.0;
        proc inDo(int sum){
            return sum * x;
        }
    }
}
```

Sample Program

```java
proc sum(int x){
    int k = 0;
    proc foo(){
        real sum = 0.0;
        proc inDo(int sum){
            return sum * x;
        }
    }
}
```
proc sum(int x) {
    int k = 0;
    proc foo() {
        real sum = 0.0;
        proc inDo(int sum) {
            return sum * x;
        }
    }
}

Sample Program
Sample Program

proc sum(int x) {
    int k = 0;
    proc foo() {
        real sum = 0.0;
        proc inDo(int sum) {
            return sum * x;
        }
    }
}

Sample Program

```c
proc sum(int x)
    int k = 0;
    proc foo()
        real sum = 0.0;
        proc inDo(int sum)
            return sum * x;
    }
}
```
Sample Program

```plaintext
proc sum(int x) {
    int k = 0;
    proc foo () {
        real sum = 0.0;
        proc inDo(int sum) {
            return sum * x;
        }
    }
}
```
Sample Program

```
proc sum(int x){
    int k = 0;
    proc foo(){
        real sum = 0.0;
        proc inDo(int sum){
            return sum * x;
        }
    }
}
```

- The scope tells you how many static chain hops you need to make, i.e., Current scope minus your scope.
Conclusion

- A language that is easy to compile often leads to
  - A language that is easy to understand.
  - More good compilers on more machines (compare Pascal and Ada!).
  - Better (faster) code.
  - Fewer compiler bugs.
  - Smaller, cheaper, faster compilers.
  - Better diagnostics.
Supplementary – ANSI C Grammar

  - `lex (flex)` specification.
  - C language scanner.
  - `yacc` grammar.
  - Parser and semantic analyzer.