What’s In a Name?

Name: way to refer to something else
variables, functions, namespaces, objects, types

```c
if ( a < 3 ) {
    int bar = baz(a + 2);
    int a = 10;
}
```
Names, Objects, and Bindings

Object 1 → Object 2
binding

Object 3 → Object 4
binding

Name 1 → Object 4
binding

Name 2 → Object 1
binding

Name 3 → Object 2
binding

Name 4 → Object 2
binding

When are objects created and destroyed?

When are names created and destroyed?

When are bindings created and destroyed?
Names, Objects, and Bindings

When are objects created and destroyed?
When are names created and destroyed?
When are bindings created and destroyed?
Part I

Object Lifetimes
Object Lifetimes

The objects considered here are regions in memory.

Three principal storage allocation mechanisms:

1. Static
   Objects created when program is compiled, persists throughout run

2. Stack
   Objects created/destroyed in last-in, first-out order. Usually associated with function calls.

3. Heap
   Objects created/deleted in any order, possibly with automatic garbage collection.
class Example {
    public static final int a = 3;
    
    public void hello() {
        System.out.println("Hello");
    }
}

Static class variable
Code for hello method
String constant “Hello”
Information about the Example class
Static Objects

Advantages:

Zero-cost memory management
Often faster access (address a constant)
No out-of-memory danger

Disadvantages:

Size and number must be known beforehand
Wasteful if sharing is possible
Stack-Allocated Objects

Natural for supporting recursion.

Idea: some objects persist from when a procedure is called to when it returns.

Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Each invocation of a procedure gets its own *frame* (*activation record*) where it stores its own local variables and bookkeeping information.
Reverse Polish Notation derived from the (prefix) Polish notation invented by Jan Łukasiewicz in the 1920s.

$1 + 2 \times 3$ vs. $1 \ 2 \ 3 \ \ast \ \div$
Stack-Based Languages

The FORTH language is stack-based. Very easy to implement cheaply on small processors.

The PostScript language is also stack-based.

Programs are written in Reverse Polish Notation:

```
2 3 * 4 5 * + . ( . is print top-of-stack)
26  OK
```
: CHANGE 0 ;
: QUARTERS 25 * + ;
: DIMES 10 * + ;
: NICKELS 5 * + ;
: PENNIES + ;
: INTO 25 /MOD CR ." QUARTERS"
    10 /MOD CR ." DIMES"
    5 /MOD CR ." NICKELS"
    CR ." PENNIES" ;
CHANGE 3 QUARTERS 6 DIMES 10 NICKELS 112 PENNIES INTO
11 QUARTERS
2 DIMES
0 NICKELS
2 PENNIES
Definitions are stored on a stack. \texttt{FORGET} discards the given definition and all that came after.

":[FOO ."Stephen" ;
:BAR ."Nina" ;
:FOO ."Edwards" ;

FOO Edwards
BAR Nina
FORGET FOO (Forgets most-recent FOO)

FOO Stephen
BAR Nina
FORGET FOO (Forgets FOO and BAR)

FOO FOO ?
BAR BAR ?
What do you need to save across a recursive call?

```c
int fib(int n) {
    if (n<2) return 1;
    else return fib(n-1) + fib(n-2);
}
```
What to save?

(Real C)

```c
int fib(int n) {
    if (n<2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

(Assembly-like C)

```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1:  tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2:  tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3:  tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

Need to be able to resume from L2 and L3. *What do we need there?*
Typical Stack Layout

↑ higher addresses

| argument 2 |
| argument 1 |
| return address |
| old frame pointer |
| saved registers |
| local variables |
| temporaries/arguments |
| ← stack pointer |

← frame pointer

↓ growth of stack
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
Executing fib(3)

```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
    L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
    L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
    L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

<table>
<thead>
<tr>
<th>FP</th>
<th>n = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>return address last frame pointer</td>
</tr>
<tr>
<td></td>
<td>tmp1 = 2</td>
</tr>
<tr>
<td></td>
<td>tmp2 =</td>
</tr>
<tr>
<td></td>
<td>tmp3 =</td>
</tr>
<tr>
<td></td>
<td>n = 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
<th>n = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Executing fib(3)

```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
Executing fib(3)

```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1:    tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2:    tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3:    tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
Executing fib(3)

```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
    L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
    L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
    L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```
```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
Heap-Allocated Storage

Static works when you know everything beforehand and always need it.

Stack enables, but also requires, recursive behavior.

A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)
```c
struct point {
    int x, y;
};

int play_with_points(int n)
{
    int i;
    struct point *points;

    points = malloc(n * sizeof(struct point));

    for (i = 0; i < n; i++) {
        points[i].x = random();
        points[i].y = random();
    }

    /* do something with the array */

    free(points);
}
```
Dynamic Storage Allocation
Dynamic Storage Allocation

\[ \text{free()} \downarrow \]
Dynamic Storage Allocation

```
free()
```

```
malloc()
```

```
downarrow free()
```
Dynamic Storage Allocation

↓ free([gray])

↓ malloc([gray])
Dynamic Storage Allocation

\[ \downarrow \text{free}() \]

\[ \downarrow \text{malloc}() \]
Dynamic Storage Allocation

Rules:

Each allocated block contiguous (no holes)

Blocks stay fixed once allocated

malloc()

Find an area large enough for requested block

Mark memory as allocated

free()

Mark the block as unallocated
Simple Dynamic Storage Allocation

Maintaining information about free memory
  Simplest: Linked list

The algorithm for locating a suitable block
  Simplest: First-fit

The algorithm for freeing an allocated block
  Simplest: Coalesce adjacent free blocks
Simple Dynamic Storage Allocation
Simple Dynamic Storage Allocation

malloc(

free(

malloc(

SNSSSN

SN

Simple Dynamic Storage Allocation

malloc(

malloc(

free(

free(

Simple Dynamic Storage Allocation

malloc( )

free( )
Simple Dynamic Storage Allocation

```
malloc()
```

```
free()
```
Dynamic Storage Allocation

Many, many other approaches.
Other “fit” algorithms
Segregation of objects by size
More clever data structures
Heap Variants

Memory pools: Differently-managed heap areas

Stack-based pool: only free whole pool at once
   Nice for build-once data structures

Single-size-object pool:
   Fit, allocation, etc. much faster
   Good for object-oriented programs
Fragmentation

`malloc( )` seven times give

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

`free()` four times gives

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

`malloc( )`?

Need more memory; can’t use fragmented memory.

Hockey smile
Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”

The original Macintosh did this to save memory.
Fragmentation and Handles

Standard CS solution: Add another layer of indirection.
Always reference memory through “handles.”

The original Macintosh did this to save memory.
Automatic Garbage Collection

Remove the need for explicit deallocation.

System periodically identifies reachable memory and frees unreachable memory.

Reference counting one approach.

Mark-and-sweep another: cures fragmentation.

Used in Java, O’Caml, other functional languages, etc.
Automatic Garbage Collection

Challenges:

How do you identify all reachable memory?

(Start from program variables, walk all data structures.)

Circular structures defy reference counting:

Neither is reachable, yet both have non-zero reference counts.

Garbage collectors often conservative: don’t try to collect everything, just that which is definitely garbage.
Part II

Scope

When are names created, visible, and destroyed?
Scope

The scope of a name is the textual region in the program in which the binding is active.

Static scoping: active names only a function of program text.
Dynamic scoping: active names a function of run-time behavior.
Scope: Why Bother?

Scope is not necessary. Languages such as assembly have exactly one scope: the whole program.

Reason: Information hiding and modularity.

Goal of any language is to make the programmer’s job simpler.

One way: keep things isolated.

Make each thing only affect a limited area.

Make it hard to break something far away.
A name begins life where it is declared and ends at the end of its block.

From the CLRM, “The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.”

```c
void foo()
{
    int x;
}
```
Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, “If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.”
public void example() {
    // x, y, z not visible

    int x;
    // x visible

    for (int y = 1; y < 10; y++) {
        // x, y visible

        int z;
        // x, y, z visible
    }

    // x visible
}
Basic Static Scope in O’Caml

A name is bound after the “in” clause of a “let.” If the name is re-bound, the binding takes effect after the “in.”

```
let x = 8 in
let x = x + 1 in
```

Returns the pair (12, 8):

```
let x = 8 in
(let x = x + 2 in
 x + 2),
x
```
Let Rec in O’Caml

The “rec” keyword makes a name visible to its definition. This only makes sense for functions.

```
let rec fib i =
  if i < 1 then 1 else
  fib (i-1) + fib (i-2)
in
fib 5
```

```
(* Nonsensical *)
let rec x = x + 3 in
```
Let...and in O’Caml

Let...and lets you bind multiple names at once. Definitions are not mutually visible unless marked “rec.”

```ocaml
let x = 8
and y = 9 in

let rec fac n =
  if n < 2 then
    1
  else
    n * fac (n - 1)
and fac1 n = fac (n - 1)
in
fac 5
```
let articles words =

  let report w =

    let count = List.length (List.filter ((=) w) words)
    in w ^ ": " ^ string_of_int count

  in String.concat "", "
  (List.map report ["a"; "the"])

  in articles
  ["the"; "plt"; "class"; "is"; "a"; "pain"; "in"; "the"; "butt"]

let count words w = List.length (List.filter ((=) w) words) in

let report words w = w ^ ": " ^ string_of_int (count words w) in

let articles words =
  String.concat "", "
  (List.map (report words)
    ["a"; "the"]) in

articles
["the"; "plt"; "class"; "is"; "a"; "pain"; "in"; "the"; "butt"]

Produces “a: 1, the: 2”
Implementing Nested Functions with Static Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) (* b *)
  in
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” evaluate to?
What does “a 5 42” evaluate to?
Implementing Nested Functions with Static Links

```
let a x s =
    let b y =
        let c z = z + s in
        let d w = c (w+1) in
        d (y+1) in (* b *)
    let e q = b (q+1) in
    e (x+1) (* a *)
```

What does “a 5 42” evaluate to?
Implementing Nested Functions with Static Links

let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)

What does “a 5 42” evaluate to?
Implementing Nested Functions with Static Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  in
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” evaluate to?
Nested Subroutines in Pascal

```
procedure mergesort;
var N : integer;

procedure split;
var I : integer;
begin
  ...
end

procedure merge;
var J : integer;
begin
  ...
end

begin
  ...
end
```
Dynamic Definitions in TeX

\% \x, \y undefined
{
  \% \x, \y undefined
  \texttt{\def \x 1}
  \% \x defined, \y undefined

  \texttt{\ifnum a < 5}
  \texttt{\def \y 2}
  \texttt{\fi}

  \% \x defined, \y may be undefined
}
\% \x, \y undefined
program example;
var a : integer; (* Outer a *)

procedure seta;
begin
  a := 1 (* Which a does this change? *)
end

procedure locala;
var a : integer; (* Inner a *)
begin
  seta
end

begin
  a := 2;
  if (readln() = 'b')
    locala
  else
    seta;
  writeln(a)
end
Most languages now use static scoping.

Easier to understand, harder to break programs.

Advantage of dynamic scoping: ability to change environment.

A way to surreptitiously pass additional parameters.
program messages;
var message : string;

procedure complain;
begin
    writeln(message);
end

procedure problem1;
var message : string;
begin
    message := 'Out of memory';
    complain
end

procedure problem2;
var message : string;
begin
    message := 'Out of time';
    complain
end
Forward Declarations

Languages such as C, C++, and Pascal require *forward declarations* for mutually-recursive references.

```plaintext
int foo(void);
int bar() { ... foo(); ... }
int foo() { ... bar(); ... }
```

Open vs. Closed Scopes

An *open scope* begins life including the symbols in its outer scope.

Example: blocks in Java

```java
{
   int x;
   for (;;) {
      /* x visible here */
   }
}
```

A *closed scope* begins life devoid of symbols.

Example: structures in C.

```c
struct foo {
   int x;
   float y;
}
```
Part III

Overloading

What if there is more than one object for a name?
Overloading versus Aliases

Overloading: two objects, one name

Alias: one object, two names

In C++,

```c++
int foo(int x) { ... }
int foo(float x) { ... } // foo overloaded

void bar()
{
    int x, *y;
    y = &x;  // Two names for x: x and *y
}
```
Examples of Overloading

Most languages overload arithmetic operators:

1 + 2 // Integer operation
3.1415 + 3e-4 // Floating-point operation

Resolved by checking the *type* of the operands.

Context must provide enough hints to resolve the ambiguity.
C++ and Java allow functions/methods to be overloaded.

```cpp
int foo();
int foo(int a); // OK: different # of args
float foo(); // Error: only return type
int foo(float a); // OK: different arg types

Useful when doing the same thing many different ways:

```
Function Overloading in C++

Complex rules because of *promotions*:

```cpp
int i;
long int l;
l + i
```

Integer promoted to long integer to do addition.

```cpp
3.14159 + 2
```

Integer is promoted to double; addition is done as double.
Function Overloading in C++

1. Match trying trivial conversions
   int a[] to int *a, T to const T, etc.

2. Match trying promotions
   bool to int, float to double, etc.

3. Match using standard conversions
   int to double, double to int

4. Match using user-defined conversions
   operator int() const { return v; }

5. Match using the elipsis ...

Two matches at the same (lowest) level is ambiguous.
Part IV

Binding Time

When are bindings created and destroyed?
Binding Time

When a name is connected to an object.

**Bound when**
- language designed
- language implemented
- Program written
- compiled
- linked
- loaded
- run

**Examples**
- if else
- data widths
- foo bar
- static addresses, code
- relative addresses
- shared objects
- heap-allocated objects
Earlier binding time $\Rightarrow$ more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```c
switch (statement) {
    case add:
        r = a + b;
        break;
    case sub:
        r = a - b;
        break;
    /* ... */
}
```

```
add %o1, %o2, %o3
```
Dynamic method dispatch in OO languages:

```java
class Box : Shape {
    public void draw() { ... }
}

class Circle : Shape {
    public void draw() { ... }
}

Shape s;
s.draw(); /* Bound at run time */
```
Interpreters better if language has the ability to create new programs on-the-fly.

Example: Ousterhout’s Tcl language.

Scripting language originally interpreted, later byte-compiled.

Everything’s a string.

```
set a 1
set b 2
puts "$a + $b = [expr $a + $b]"
```
Binding Time and Efficiency

Tcl’s `eval` runs its argument as a command.

Can be used to build new control structures.

```tcl
proc ifforall {list pred ifstmt} {
    foreach i $list {
        if [expr $pred] {
            eval $ifstmt
        }
    }
}

ifforall {0 1 2} {
    $i % 2 == 0
} {
    puts "$i even"
}

0 even
2 even
```
Part V

Binding Reference Environments

What happens when you take a snapshot of a subroutine?
References to Subroutines

In many languages, you can create a reference to a subroutine and call it later. E.g., in C,

```c
int foo(int x, int y) { /* ... */ }

void bar()
{
    int (*f)(int, int) = foo;
    (*f)(2, 3); /* invoke foo */
}
```

Where does its environment come from?
References to Subroutines

C is simple: no function nesting; only environment is the omnipresent global one. But what if there were?

```c
typedef int (*ifunc)();

ifunc foo() {
    int a = 1;
    int bar() { return a; } /* this is not C */
    return bar;
}

int main() {
    ifunc f = foo(); /* returns bar */
    return (*f)(); /* call bar. a? */
}
```