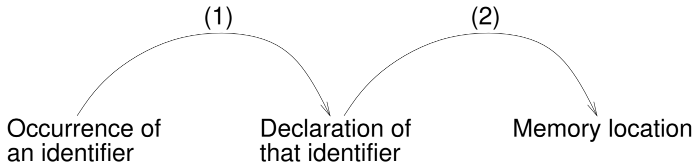
Scope and Lifetime of Variables

References:

- Sethi, sections 5.3, and 5.4
- Sebesta, sections 4.3, 4.4, 4.8, and 4.9

How Does This Work?



When a program says "`i = 47;`", which declaration of `i` applies?

- There may be many.

And in what memory location is `i` stored?

- Even if there is only a single `i` declared, there may be many different memory locations for it during execution;
  they may even be simultaneous.

In order to understand these things, we must understand two important attributes of variables: "scope" and "lifetime".

Attributes of Variables

All variables have 6 attributes: name, address, value, type, scope, and lifetime.

The **scope** of a variable is the part of the program in which the variable can be referenced.

The **scope rules** of a *language* determine the scope of every variable, and resolve the mapping from identifier occurrence to declaration.

The **lifetime** of a variable is the time during which it is bound to a particular location in memory, and so is related to the mapping from declaration to memory location.
Subprogram calls and how they work also affect this mapping. (Why?)

Scope Rules

There are two categories of scope rules.

Static scope

- Static scope (also called "lexical" scope) is based on the structure of the program.
- If a language uses static scope rules, it can thus determine the scope of every variable before run time.
- This is most familiar.

Dynamic scope

- Based on the calling sequence of subprograms during program execution.
- If a language uses dynamic scope rules, it cannot determine the scope of any variable until run time.

Static Scope

Algorithm: Look first in the block in which the reference occurs; if that fails, look in the enclosing block. Continue until successful.

C Example (based on Sebesta, p. 176):

```
int x;

void sub1 (void) {
   printf("%d\n", x);
}

void sub2 (void) {
   int x = 8;
   sub1();
}

void main (void) {
   x = 10;
   sub1();
   sub2();
}
```

Output:

Hidden variables

Notice that the declaration of `x` in `sub2` **hides** the `x` that was declared globally — that is, inside of `sub1`, that other `x` is not accessible.

In a language that allows

- variables to be declared inside any blocks (*e.g.*, C, Java), or
- subprograms to be declared inside other subprograms (*e.g.*, Ada)

there can be many nested levels of scope.

Some languages provide a way to still access hidden variables.

- Ada: `main.x`
- C++: Doesn't allow nested subprograms, but you can still hide a global inside a subprogram. To access the global: `::x`

Dynamic Scope

Algorithm: Look first in the block in which the reference occurs; if that fails, look in the calling subprogram. Continue until successful.

```
int x;

void sub1 (void) {
   printf("%d\n", x);
}

void sub2 (void) {
   int x = 8;
   sub1();
}

void main (void) {
   x = 10;
   sub1();
   sub2();
}
```

Output if scope were dynamic:

A Harder Example

```
int w = 1;
int x = 2;
String y = "achoo";
String z = "ohm";

function p () {
   int w = 13;
   int x = -1;
   String z = "hello";
   print w, x, y, z;
   q();
}
function q () {
   int x = 99;
   String y = "goodbye";
   print w, x, y, z;
   r();
   s();
}
```

```
function r () {
   int x = 101;
   String z = "googoo";
   print w, x, y, z;
}
function s () {
   int x = -555;
   print w, x, y, z;
   t();
}
function t () {
   print w, x, y, z;
}
main() {
   p();
}
```

Problems with Dynamic Scope

- It is hard to understand code with dynamic scope.
- You can change what the program does just be renaming variables!
  Example: Rename `sub2`'s `x` to anything else.
- Any subprogram you call, no matter where it is, can access your local variables.
- It's also slower to execute.

Possible advantage

- Reduces the need for parameters.

What Various Languages Do

Dynamic scope was used in early versions of Lisp, and some other interpreted languages: APL, Snobol.

(It makes more sense in an interpreted language which is slow anyway.)

But most modern languages use static scope because of the problems with dynamic scope.

Some terminology

A declaration of a variable is **visible** at a given point in the program text if that point is within the scope of the declaration.

A variable is **local** to a piece of code if it is visible only within that piece of code.

Macros Can Yield Dynamic Scope

A **macro** has a name, and a piece of code which is its body.

Examples in C:

```
/* Simple macros */
#define MAX_STRING_LENGTH 80
#define PI 3.14159

/* Parameterized macros */
#define IS_EVEN(n) ((n)%2==0)
#define MAX(x, y)  ((x)>(y)?(x):(y))

void main (void) {
    int a = 9; int b = 4; int c = 8;
    int i = MAX(a, b+c);
    if (IS_EVEN(i))
         i = i/2;
}
```

Whenever a macro is "called",

1. the actual parameters (if any) are textually substituted for formal parameters, then
2. the resulting macro body is textually substituted for the call.

Macros vs subprogram calls

A macro is like a subprogram call, but it is processed *before* run-time, not at run-time.

Key differences:

- a macro call is faster than a subprogram call.
- a macro call makes the compiled code larger than a procedure call.

Example with dynamic scope

```
#define SUB1 printf("%d\n", x);
int x;

void sub2 (void) {
   int x = 8;
   SUB1;
}

void main (void) {
   x = 10;
   SUB1
   sub2();
}
```

Lifetime

Recap: The **lifetime** of a variable is the time during which it is bound to a particular location in memory.

Variable lifetime $\neq$ variable scope.

Example 1:

```
void dosomethingelse(void) {
    ...
}
void dosomething(void) {
    int n;
    ...
    dosomethingelse(void);
    ...
}
```

Example 2:

```
void gofigure(void) {
    static int blah;
    ...
}
```


Three Categories

Static variables

- Bound to a memory location before execution, and do not move.
- Lifetime = entire execution.
- Called `static` in Java, C, and C++.

Pros and cons:

- No overhead at runtime for allocation and deallocation.
- Can't have recursion if only have static variables.
  E.g., early Fortran.
- When finished using a variable, no other variable can use its space.

Stack-dynamic variables

- Bound to a memory location when the variable's declaration is executed.
- Lifetime = from execution of declaration until end of the "block" in which it occurs.
- Called "stack-dynamic" because stored in an activation record, which is pushed/popped on the run-time stack when entering/leaving a block.
- This is the default for local variables in Java, C, and C++.

Pros and cons:

- Overhead at runtime for allocation and deallocation.
- Allows recursion to work properly.
- Efficient use of space because deallocate when pop.

Heap-dynamic variables

- Bound to a memory location when explicit instructions are executed at runtime.
- Nameless; just referred to by a pointer.
- Lifetime = from allocation instruction to deallocation instruction.
- Called "heap-dynamic" because the memory is taken from the heap.
C:	malloc	free
C++:	new	delete
Java:	new	

Pros and cons:

- Good for building dynamic structures.
- Hard to write the code correctly.
- A little slower to execute.