Class 2: Names, Binding, Scoping

Names

What is a *name* in a program?

An *identifier*, made up of a string of characters, used to represent something else.

What can be named?

- Execution points (labels)
- Mutable variables
- Constant values
- Subroutines (functions, procedures, methods)
- Types
- Type constructors (e.g., list or vector)
- Classes
- Modules/packages
- Execution points with environment (continuation)

Names are a key part of *abstraction*, which helps to make programming easier:

- Abstraction reduces conceptual complexity by hiding irrelevant details
- Names for subroutines: *control abstraction*
- Names for types/classes: *data abstraction*

Bindings

A *binding* is an association of two things, such as:

- Names and the things they name
- A question about how to implement a feature and the answer to that question.

Binding time is the time at which the association is made.

- *Language design time*: bind built-in features such as keywords
- *Language implementation time*: implementation dependent semantics such as bit-width of an integer
- *Program writing time*: bind names chosen by programmer
- *Compile time*: bind high-level constructs to machine code
- *Link time*: bind names to virtual addresses
- *Load time*: bind virtual to physical addresses (can change during run time)
- *Run time*: bind variables to values, includes many bindings which change during execution such as the binding of function parameters to the passed argument values.

Static binding (aka *early binding*) means before run time; *Dynamic binding* (aka *late binding*) means during run time.

What are some advantages of static binding times?

- *Efficiency*: the earlier decisions are made, the more optimizations are available to the compiler.
- *Ease of implementation*: Static binding makes compilation easier.

What are some advantages of dynamic binding times?

- *Flexibility*: Languages that allow postponing binding give more control to programmer
- *Polymorphic code*: Code that can be used on objects of different types is *polymorphic*. Examples:
    - subtype polymorphism (dynamic method dispatch)
    - parametric polymorphism (e.g. generics)

Typically, early binding times are associated with compiled languages and late binding times with interpreted languages.

Lifetimes

The period of time between the creation and destruction of a name-to-object binding is called the binding's *lifetime*.

The time between the creation of an object and its destruction is the *object's lifetime*.

Is it possible for these to be different?

- When a variable is passed by reference, the binding of the reference variable has a shorter lifetime than that of the object being referenced.
- If there are multiple pointers to an object and one of the pointers is used to delete the object, the object has a shorter lifetime than the bindings of the pointers.
- A pointer to an object that has been destroyed is called a *dangling reference*. Dereferencing a dangling pointer is usually a bug.

Static Allocation

Static objects are objects whose lifetime spans the entire program execution time. Examples:

- global variables
- the actual program instructions (in machine code)
- numeric and string literals

- tables produced by the compiler

Static objects are often allocated in *read-only* memory (the program's data segment) so that attempts to change them will be reported as an error by the operating system.

Under what conditions could local variables be allocated statically?

- The original Fortran did not support recursion, allowing local variables of functions to be allocated statically.
- Some languages (e.g. C++) allow local variables of functions to be declared `static` which causes them to be treated as static objects.

Dynamic Allocation

For most languages, the amount of memory used by a program cannot be determined at compile time (exceptions include earlier versions of Fortran).

Some features that require dynamic memory allocation:

- Recursive functions
- Pointers, explicit allocation (e.g., `new`, `malloc`)
- First-class functions

We distinguish *stack-based* and *heap-based* dynamic allocation.

In languages with recursion, the natural way to allocate space for subroutine calls is on the *stack*. This is because the lifetimes of objects belonging to subroutines follow a last-in, first-out (LIFO) discipline.

Each time a subroutine is called, space on the stack is allocated for the objects needed by the subroutine. This space is called a *stack frame* or *activation record*.

Objects in the activation record may include:

- Return address
- Pointer to the stack frame of the caller (*dynamic link*)
- Arguments and return values
- Local variables
- Temporary variables
- Miscellaneous bookkeeping information

Even in a program that does not use recursion, the number of subroutines that are active at the same time at any point during program execution is typically much smaller than the total number of subroutines in the program. So even for those programs, it is usually beneficial to use a stack for allocating memory for activation records, rather than allocating that space statically.

Some objects may not follow a LIFO discipline, e.g.

- objects allocated with `new`, `malloc`,
- the contents of local variables and parameters in functional languages.

The lifetime of these objects may be longer than the lifetime of the activation record for the call to the subroutine in which they were created. These objects are therefore allocated on the *heap*: a section of memory set aside for such objects (not to be confused with the data structure for implementing priority queues, aka. Fibonacci Heap).

The heap is finite: if we allocate too many objects, we will run out of space.

Solution: deallocate space when it is no longer needed by an object. Different languages implement different strategies for managing the deallocation of dynamic objects:

- Manual deallocation with e.g., `free`, `delete` (C, Pascal)
- Automatic deallocation via garbage collection (LISP, Java, Scala, C#, OCaml, Haskell, Python, JavaScript, ...)
- Semi-automatic deallocation using destructors, smart pointers, and reference counting (C++, Ada, Objective-C, Rust)
  - Automatic because the destructor is called at certain points automatically
  - Manual because the programmer writes the code for the destructor and/or needs to decide which smart pointer type to use.

Manual deallocation is a common source of bugs:

- Dangling references
- Memory leaks

We will discuss automatic memory management in more detail later. However, it is helpful to understand how allocation and deallocation requests by the program are implemented.

Ultimately, a program's requests for fresh heap memory are relayed to the operating system via system calls. The operating system in turn grants the program access to memory blocks that it can use to store its dynamically allocated objects. Since system calls are expensive, the program's memory management subsystem usually requests large memory blocks from the operating system at a time which it then manages itself to satisfy dynamic allocation requests for smaller chunks of memory that fit into the large block.

The heap thus starts out as a single block of memory. As objects are allocated and deallocated, the heap becomes broken into smaller subblocks, some in use and some not in use.

Most heap-management algorithms make use of a *free list*: a singly linked list containing blocks not in use.

- *Allocation*: a search is done through the free list to find a free block of adequate size. Two possible algorithms:
  - *First fit*: first available block is taken
  - *Best fit*: all blocks are searched to find the one that fits the best
- *Deallocation*: the deallocated block is put back on the free list

Fragmentation is an issue that degrades performance of heaps over time:

- *Internal fragmentation* occurs when the block allocated to an object is larger than needed by the object.
- *External fragmentation* occurs when unused blocks are scattered throughout memory so that there may not be enough memory in any one block to satisfy a request.

Some allocation algorithms such as the buddy system are designed to minimize external fragmentation while still being efficient.

Scope

The region of program text where a binding is active is called its *scope*. Notice that scope is different from lifetime. Though, the scope of a binding often determines its lifetime.

Kinds of scoping include

- *Static* or *lexical*: binding of a name is determined by rules that refer only to the program text (i.e. its syntactic structure).
  - Typically, the scope is the smallest block in which a variable is declared
  - Most languages use some variant of this
  - Scope can be determined at compile time
- *Dynamic*: binding of a name is given by the most recent alive declaration encountered during run-time
  - Used in Snobol, APL, some versions of Lisp
  - Considered a historic mistake by most language designers because it often leads to programmer confusion and bugs that are hard to track down.

To understand the difference between dynamic and static scoping, consider the following Scala code

```
1: var x = 1
```

```
 2:
 3: def f () = println(x)
 4: def g () =
 5:   var x = 10
 6:   f ()
 7:
 8: def h () =
 9:   var x = 100
10:   g ()
11:   f ()
12:
13:
14: f (); g (); h ()
```

Scala uses static scoping, so the occurrence of `x` on line 3 always refers to the variable `x` declared on line 1. So the program will print four times `1`.

If Scala were to use dynamic scoping, the occurrence of `x` on line 3 will always refer to the most recent binding of `x` before `f` is called. For the first call to `f` on line `14` this is `x` on line 1. For the call to `f` via `g`, it is `x` on line 5 and for the call to `f` via `h`, it is `x` on line 9. So in this case the program will print `1`, `10`, `10`, and `100`.

Nested Scopes

Most languages allow nested subroutines or other kinds of nested scopes such as nested code blocks, classes, packages, modules, namespaces, etc. Typically, bindings created inside a nested scope are not available outside that scope.

On the other hand, bindings at one scope typically are available inside nested scopes. The exception is if a new binding is created for an existing name in the nested scope. In this case, we say that the original binding is *hidden*, and has a *hole* in its scope.

Many languages allow nested scopes to access hidden bindings by using a *qualifier* or *scope resolution operator*.

- In Ada, `A.x` refers to the binding of `x` created in subroutine `A`, even if there is a lexically closer scope that binds `x`.
- Similarly, Java, Scala, and OCaml use `A.x` to refer to the binding of `x` created in class (package, object, or module) `A`.
- In C++, `A::x` refers to the binding of `x` in *namespace* `A`, and `::x` refers to `x` in the *global* scope.

Scope qualifiers can be nested, e.g. `A.B.x` refers to the binding of `x` in nested scope `B` of scope `A`. Qualified names are typically interpreted relative to the scope in which they occur (respectively, the global scope). For instance a qualified name `B.C.x` that occurs in scope `A` may refer to (1) `A.B.C.x` or (2) `B.C.x` relative to the global scope. The rules used to disambiguate between these possibilities are language-specific.

Some languages allow bindings of other named scopes to be imported into the current scope so that they can be referred to without qualifiers

- In Java, `import java.util.ArrayList;` imports the class `ArrayList` from package `java.util` into the current scope, whereas `import java.util.*` imports all classes of that package.
- Scala uses similar syntax for imports as Java but allows `import` directives to occur in any scope and not just at the top-level scope of a package.
- In C++, `using std::cout;` imports the name `cout` from namespace `std` into the current scope. On the other hand, `using namespace std;` imports all bindings made in namespace `std`.
- In OCaml, the directive `open Format` imports all bindings of module `Format` into the current scope.

Often, the visibility of bindings of names whose scope is outside of the path to the current scope can be restricted using *visibility modifiers* (e.g. `public`, `protected`, and `private`).

Some languages, such as Ada, support nested subroutines, which introduce some extra complexity to identify the correct bindings. Specifically, how does a nested subroutine find the right binding for an object declared in an outer scope?

Solution:

- Maintain a *static link* to the *parent frame*
- The parent frame is the most recent invocation of the lexically surrounding subroutine
- The sequence of static links from the current stack frame to the frame corresponding to the outermost scope is called a *static chain*.

Finding the right binding:

- The level of nesting can be determined at compile time
- If the level of nesting is `j`, the compiler generates code to traverse the static chain `j` times to find the right stack frame.

Declaration Order

What is the scope of `x` in the following code snippet?

```
{
  statements1;
  var x = 5;
  statements2;
}
```

- C, C++, Ada, Java: `statements2`
- JavaScript, Modula3: entire block, but value of `x` will be undefined when read in `statements1`.
- Pascal, Scala, C#: entire block, but `x` is not allowed to be used in `statements1`! If `x` is used in `statement1`, the compiler will reject the program with a static semantic error.

C and C++ require names to be declared before they are used. This requires a special mechanism for recursive data types, which is to separate the *declaration* from the *definition* of a name:

- A *declaration* introduces a name and indicates the scope of the name.
- A *definition* describes the object to which the name is bound.

Example of a recursive data type in C++:

```
struct manager;              // Declaration
struct employee {
  struct manager* boss;
  struct employee* next_employee;
  ...
};

struct manager {             // Definition
  struct employee* first_employee;
  ...
};
```

Redeclaration

Some languages (in particular, interpreted ones) allow names to be *redeclared* within the same scope.

```
function addx(x) { return x + 1; }

function add2(x) {
  x = addx(x);
  x = addx(x);
  return x;
}

function addx(x) { return x + x; } // Redeclaration of addx
```

What happens if we call `add2(2)`?

In most languages that support redeclaration, the new definition replaces the old one in all contexts, so we would get `8`.

In languages of the ML family like OCaml, the new binding only applies to later uses of the name, not previous uses. For example, the corresponding OCaml code of the above example looks like this

```
let addx x = x + 1

let add2 x =
  let x1 = addx x in
  let x2 = addx x1 in
  x2

let addx x = x + x
```

Calling `add2` with value `2` now yields `4` instead of `8`.

Determining Static Scopes

To understand how to determine the static scopes of variables in a given program it is helpful to remember that programs can be understood as abstract syntax trees. The view of programs as trees is invaluable in understanding scopes. Let us explain this through an example.

Consider the following program:

```
 0: var x = 2;
 1:
 2: { var x = 3;
 3:
 4:   { x = x + 1;
 5:


 6:      var x = 2;
 6:     var x = 2;
 7:
 8:     x = x + 1;
 9:   }
10: }
11:
12: { var y;
13:
14:   y = x + 1;
15: }
```

Let us analyze this program using C/Java's declaration order rules (i.e., the scope of a variable declaration extends from the declaration to the end of the block). We can make this explicit in the program above by introducing an extra code block that encompasses the declaration on line 6 up to the end of line 8:

```
var x = 2;

{ var x = 3;

  { x = x + 1;

    { var x = 2;

      x = x + 1;
    }
  }
}

{ var y;

  y = x + 1;
}
```

Now we have essentially one block per scope. So let's name all the blocks so that we can talk about them more easily:

```
var x = 2;

1: { var x = 3;

     1.1: { x = x + 1;

            1.1.1: { var x = 2;
```

```
                     x = x + 1;
                   }
          }
   }

2: { var y;

     y = x + 1;
   }
```

There is also the implicit top-level block in which we have the top-level declaration on line 0 of the original program as well as blocks 1 and 2. Let's call this block 0. So we can now arrange all these blocks in a tree according to their nesting structure. Let's also compute for each block the set of variable names that are declared directly in that block. This gives us:

```
     0
    / \
   1   2
  /
 1.1
  |
1.1.1
```

Variables declared in each scope

```
0: x
1: x
1.1: -none-
1.1.1: x
2: y
```

Now, suppose we want to determine which declaration the name `x` appearing on line 4 of the original program refers to. This line belongs to block `1.1`. To determine the right declaration we now walk up the tree starting from node `1.1` and stop at the first node that has a declaration for `x`. Since `1.1` has no declaration for `x`, we go to its parent block `1`. This block has a declaration `x`: the one on line 2. So that's the variable `x` that the occurrence of `x` on line 4 refers to.

Let us do the same for the occurrence of `x` on line 14. This line

Let us do the same for the occurrence of `x` on line 14. This line belongs to block `2`. So we start our search there. Block `2` has no declaration for `x`, so we go to its parent which is block `0`. Block `0` has a declaration for `x`: the one on line 0. So that is the declaration that the occurrence of `x` on line 14 refers to.

Now let us repeat this game for the same program but with the declaration order of C#/Scala, i.e., the scope of a declaration extends the entire block where the declaration occurs. That is, we have no implicit block `1.1.1` in this case:

```
var x = 2;

1: { var x = 3;

     1.1: { x = x + 1;

            var x = 2;

            x = x + 1;
          }
   }

2: { var y;

     y = x + 1;

   }
```

We draw the tree and compute variable declarations per block:

```
      0
     / \
    1   2
   /
  1.1
```

Variables declared in each scope

```
0: x
1: x
1.1: x
2: y
```

Now, again we determine which declaration of `x` the occurrence on line

Now, again we determine which declaration of `x` the occurrence on line 4 of the original program refers to. We start from block `1.1` to which line 4 belongs. This block has a declaration: the one on line 6. So this is the declaration this occurrence of `x` refers to. Since line 4 comes before line 6, this is a case of "use variable before it's declared". So the compiler would reject this program with a static semantic error.

For the occurrence of `x` on line 14, the analysis would proceed as before.