

The Name Game

- At the heart of the *raison d'être* of programming languages: people say “tomato,” the computer says “0aff8e7391”
- Beyond syntax
 - Names are a single token, e.g. < id >, at scanner time
 - Proper analysis of names at parser time is outside the main parsing algorithm; requires additional constructs such as a *symbol table*
- Binding and scope
 - When does a name have meaning? How long does this meaning last?
- Abstraction
 - Names are key to well-implemented control and data abstractions

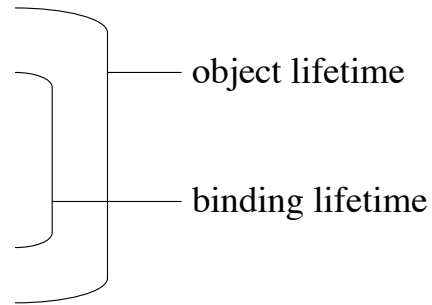
Binding Time

- When do I know what “tomato” means?
- *Language design*: literals, keywords
- *Language implementation*: arithmetic precision, I/O
- *Program creation*: developer decisions and designations
- *Compile time*: variable, function, data type resolution
- *Link time*: references to pre-existing code/libraries
- *Load time*: virtual-to-physical memory mapping; currently available code/libraries
- *Run time*: values depending on user input, external factors
- “Static” vs. “dynamic” binding: not strictly or formally defined, but in general “static” is anything prior to link time, and “dynamic” is anything after; link time is a gray area for static vs. dynamic binding

Object Lifetime

- We mean objects in general, not specific to object-oriented programming or languages

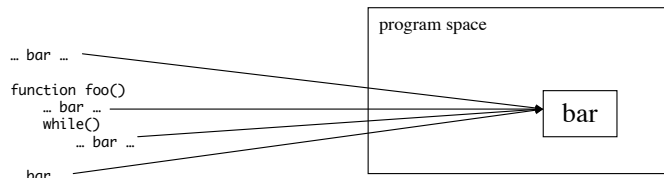
- Object creation
- Binding creation
- References
- Activation/reactivation of bindings
- Binding destruction
- Object destruction



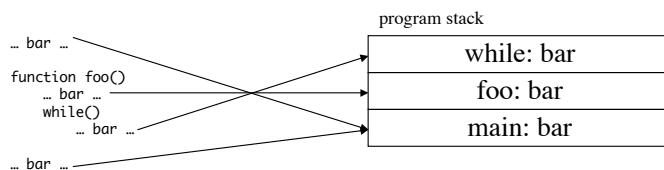
- Binding outlives object = *dangling reference*

Allocation Approaches

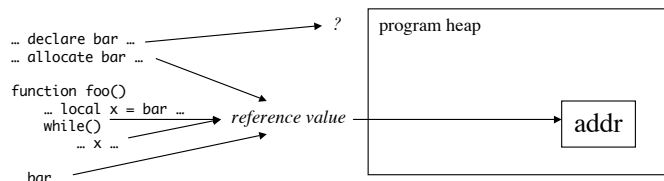
- Static
 - always available



- Stack
 - push/pop



- Heap
 - garbage collection



Scope

- Where and when can I *see* my “tomato”?

- Static/lexical scope
 - Determined from the source code
 - Nested subroutines
 - Nested blocks
 - Modules
 - Classes

```
static int total = 5;
int addToStatic(int addend) {
    total += addend;
    return total;
}
int addToLocal(int addend) {
    int total = 20;
    total += addend;
    return total;
}
```

- Dynamic scope
 - Dependent on what is known at runtime

```
our $x = 1;
sub f {
    print "$x\n";
}
sub g {
    local $x = 2;
    f();
}
g();
print "$x\n";
```

- Implementing scope
 - Lexical scope: symbol tables
 - Dynamic scope: association lists (“a-lists”), central reference table

Modules

- Key element of practical and large-scale programming
 - Code reuse
 - Separate compilation
 - Information encapsulation
 - Reduction of name conflicts
 - Separation of implementation and interface
- Module as manager
 - Module is a single universal entity
 - Module-as-manager creates and operates on entities within that module
- Module as type
 - Module corresponds to creatable entities (“instance” or “occurrences”)
 - The object-oriented “class” construct is a module-as-type with additional properties, most notably inheritance

Binding Reference Environments

- Arises in languages where subroutines can themselves be:
 - passed around as arguments
 - stored in variables
 - nested in other subroutines
- When invoking a subroutine by reference, do we use the referencing environment that is active:
 - when the reference is bound to the subroutine (“deep binding”)
 - when the reference is actually called (“shallow binding”)
- Even trickier when subroutines are first-class values
 - first class: can be passed as an argument, returned as a subroutine result, or assigned to a variable
 - some also require first class to be creatable at runtime
 - second class: can be passed as an argument only
 - third class: cannot even be passed as an argument

Overloading

- Saying “tomayto” *and* “tomahto” ... but spelling them the same way, and meaning them differently!
 - Decided by semantic analysis
 - Ambiguity = error
- Related ideas:
 - coercion — forcing one type of value into another so that it fits what is really a non-overloaded declaration
 - polymorphism — “overloading” the *data* instead of the *control*
 - generics — templates for similar activities that act on different types
 - sort of like coercing a subroutine instead of its arguments

Design Issues

- Scoping rules
 - How to treat values that are used before they are declared
 - How to treat names that exist at different levels in the code
- Programming in the large: separate compilation
 - Referencing names and values outside of your own code
 - Implicit (C)
 - Interface/implementation (C header files)
 - Packages (Java)
 - Namespaces (C++)

 - Nesting issues