CS 230
Programming Languages

10 / 18 / 2022

Instructor: Michael Eckmann

Today’s Topics

- Questions? / Comments?
- Chapter 5
  - Code, static data area, stack and heap
  - Different kinds of variables in terms of when/where they are allocated
  - Scope
  - Lifetime

Binding

- Binding attributes to variables
  - Static binding – first occurs *before* run time and stays same throughout program execution
    - note: load-time is considered before run-time
  - Dynamic binding
    - Either it first occurs *during* run time
    - OR it can change during execution

Binding

- Type bindings
  - Variable declaration
    - **Explicit** (what we're used to -- declare a variable by giving it a name and a type.)
    - **Implicit** – associate a type based on naming convention and declaration happens on first appearance
      - I, J, ..., N are implicitly Integer types in Fortran

Binding

- Dynamic Type Binding (e.g. JavaScript, PHP, Python)
- Type is specified through an assignment statement
- e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility
  - Disadvantages:
    - High cost at run-time (dynamic type checking and interpretation)
    - Type error detection by the compiler is impossible, why? Instead type checking, if done at all, has to wait until run-time.

Binding

- Dynamic Type Binding
- Advantage: flexibility
- Disadvantages:
  - Decreased reliability - Why?
    - Type error detection by the compiler is impossible, why?
    - compiler can't detect errors involving incorrect assignment of types.
  - High cost at run-time due to:
    - dynamic type checking
    - Pure interpretation is required for these languages b/c if types are not known at compile time, machine language instructions cannot be generated.
    - Textbook claims that pure interpreted languages take at least 10 times as long as equivalent machine code.

An aside about memory

- How is memory divided up and categorized for executing programs? A typical setup is as follows. I'll draw a diagram.
  - executable code
  - static data area
    - used for statically declared objects like globals and constants
  - stack (grows one way)
    - used for local variable allocation during “procedure / method / subroutine / function” calls.
  - heap (grows the other way)
    - used for dynamic objects
      - objects that are allocated at runtime, may change and size not known until run time
        - e.g. linked lists with unknown number of nodes, trees with unknown number of nodes, etc.

Binding

- Storage bindings and lifetime
  - Static variables (lifetime is the total time of execution)
    - e.g. globals, static variables
    - allocated in the static data area
  - Stack-dynamic variables (what's the lifetime of these?)
    - e.g. Local variables to methods
    - allocated on the stack. When are they allocated and deallocated?
  - Explicit heap-dynamic variables
    - e.g. Variables used for data structures that shrink and grow during execution, or those only referenced through pointers or references.
    - allocated / deallocated on the heap
  - Implicit heap-dynamic variables
    - All attributes (type, size, etc.) of these are bound when value is assigned. e.g. The JavaScript example of list a few slides ago.
    - allocated / deallocated on the heap

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
    - Static variables
        - What are some advantages and disadvantages?

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
  - Static variables
    - What are some advantages and disadvantages?

    + ability to have history sensitive (can retain value from after method is done) variables inside a method/function

    + efficient - addressing is direct

    + allocation is done before run-time so there is no run-time overhead of allocation and deallocation

    - reduced flexibility

    with only static variables you couldn't write recursive routines -why?

    - could waste memory

    can't share storage with other variables that do not have to overlap existence.

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
  - Stack-dynamic variables
    - What are some advantages and disadvantages?

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
  - Stack-dynamic variables
    - What are some advantages and disadvantages?

    + allows recursion

    + share memory space with other stack-dynamic variables

    - slower because bindings of variables to memory is done at runtime

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
  - Explicit heap-dynamic variables (e.g. Java references to objects)
    - What are some advantages and disadvantages?

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
  - Explicit heap-dynamic variables (e.g. Java references to objects)
    - What are some advantages and disadvantages?

    + flexibility / expressivity

    - pointers/references could be difficult to use correctly

    - slower at runtime b/c indirect addressing (what is indirect addressing?)

    - complex implementation of how these variables are stored and accessed in memory

    - complicated and costly heap management (e.g. Java's garbage collection)

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
    - Implicit heap-dynamic variables
        - What are some advantages and disadvantages?

Binding

- Evaluation of these kinds of variables. Think in terms of memory space, cost of execution, reliability, efficiency etc.
  - Implicit heap-dynamic variables
    - What are some advantages and disadvantages?

    + extremely flexible, generic code (same code works for many types)
    easy to write

    - slower

    - reduced error detection, therefore reduced safety

Type Checking / Strong Typing

- See chapter 6 sections 6.12 and 6.13
- Type checking --- checks that the operands of an operation are of compatible types
  - what does compatible type mean?
- If all bindings of variables to types are static then type checking can be done when?
- If any bindings of variables to types are dynamic then type checking must be done when?

Type Checking / Strong Typing

- Type checking --- checks that the operands of an operation are of compatible types
  - what does compatible type mean?
- If all bindings of variables to types are static then type checking can be done before run-time.
- If any bindings of variables to types are dynamic then type checking is required at run-time. This is Dynamic Type Checking.

Type Checking / Strong Typing

- Strong Typing – defined by the text as --- a language is strongly typed if type errors are always detected (whether at compile-time or run-time).
- According to this definition, what would you say about Java --- is it strongly typed?

Type Checking / Strong Typing

- Strong Typing – defined by the text as --- a language is strongly typed if type errors are always detected (whether at compile-time or run-time).
- According to this definition, what would you say about Java --- is it strongly typed?
  - Yes nearly. Only casting (which is explicitly done by the programmer) could cause an error to go undetected.

Scope vs. lifetime

- Scope of a variable is the range of statements in which the variable is visible. The lifetime is of a variable is temporal, that is, the time from when the variable comes into existence until it is released from memory.

- They are related in many cases, but two examples to show that they are clearly different concepts:

- static variables can be declared in a function in C++. These variables are used to retain values between subsequent calls.
- What's the scope of a variable like this? What's its lifetime?

Scope vs. lifetime

- Scope of a variable is the range of statements in which the variable is visible. The lifetime is of a variable is temporal, that is, the time from when the variable comes into existence until it is released from memory.

- They are related in many cases, but two examples to show that they are clearly different concepts:

- static variables can be declared in a function in C++. These variables are used to retain values between subsequent calls.
- What's the scope of a variable like this? What's its lifetime?
  - Scope: the function in which is it declared
  - Lifetime: the entire time the program is running

Scope vs. lifetime

- Many languages do not allow you to reference variables of a function that calls a particular function.
- e.g.

```
function1()
{ // do some stuff here
}
function2()
{
    int x;
    function1();
}
```

- What's the scope of x? What's its lifetime?