Von Neumann Architecture

- Central Processing Unit (Control, Arithmetic, Logic)
- Communicate with memory, input and output devices
- Memory stores both instructions and data
- The concept of variables allows one to refer to specific memory locations.

Variable Properties

- Data type
- Scope
- Lifetime

Names

- A string of characters to identify some entity in a program
- Some issues:
  - maximum length? (e.g. FORTRAN 95: 31, C++: no limits in standard)
  - case sensitivity, camel notation
  - legal forms? Letters, digits, underscore. Starting letter?
  - language design vs. style
  - special words: reserved or can be redefined? Too many reserved words?

Variables

- Abstraction of some memory location.
- Described by the attributes:
  - name (can be anonymous)
  - address
  - data type
  - value
  - lifeime
  - scope

Variable Address

- Each time a variable is created (e.g. local variable) it may be at a different location
- Sometimes called l-value
- Aliasing: multiple variable names referring to the same address (e.g. pointers, references, unions)
- Bad for readability and maintainability

Data Type

- Determines size of the memory location, and how to interpret the binary bit pattern in memory
- Determines valid range (for numeric values) and also valid operations
- More on this later

Value

- The content of the memory locations referred to by the variable
- Sometimes called r-value
- To access the value, we need the address of the variable (l-value)
- The value is simply a sequence of bits. Its interpretation depends on the data type.

Binding

- Binding is the association between an attribute and an entity
- e.g. associating the name of a variable to its type or value
- e.g. associating the operator symbol with its meaning
- Binding time is when the association is determined.
- Binding time can be at:
  - language design (e.g. `int`)
  - static/compile time (e.g. `int x;`)
  - dynamic/run time (e.g. `int *x = new int;`)

Binding

- Static binding occurs before run time (usually at compilation time) and remains unchanged during program execution
- Dynamic binding occurs during run time and may change during program execution

Type Binding

- A variable needs to be bound to a data type
- In static binding:
  - explicit declarations: e.g. C-like languages
  - implicit declarations: first use determines its type through some convention
- Implicit declarations are more convenient but not good for reliability because compilers cannot always perform type checking
- Some languages like Perl uses special characters to indicate data type (e.g. `$` for scalars, `@` for arrays)

Type Binding

In dynamic binding:

- type is assigned when a value is assigned to a variable
- the type is based on the RHS of the assignment
- the type of a variable can change during program execution
- more flexible (e.g. no need to worry about which numeric type to use)
- common in scripting languages (e.g. Python, JavaScript, PHP)
- Hard for compiler/interpreter to detect errors
- Higher cost to implement dynamic type binding: type checking is done at run time, and more storage is needed to describe current type

Storage Bindings and Lifetime

- Allocation: process to associate a variable to a memory location from the pool of available memory
- Deallocation: process of returning the memory location to the pool
- The lifetime of a variable is the time during which the variable is bound to a memory location
- Four types:
  - static
  - stack-dynamic
  - explicit heap-dynamic
  - implicit heap-dynamic

Static Variables

- Bound before program execution and does not change until program terminates
- Global variables are often static
- In C/C++: static local variables
- efficient: no overhead for allocation/deallocation, compiler can generate direct addresses
- reduced flexibility: cannot support recursion, cannot share storage

Stack-Dynamic Variables

- Binding is done when program execution reaches the declaration
- Allocated from the run-time stack
- Typical for local variables
- Needed for recursion
- Memory can be reused
- Small overhead for allocation and deallocation, indirect addressing

Explicit Heap-Dynamic Variables

- Anonymous memory cells allocated and deallocated by explicit run-time instructions (e.g. `new` and `delete`)
- Must be referenced through pointer or reference variables
- Allows memory to be allocated and deallocated as needed
- Can be inefficient and error-prone

Implicit Heap-Dynamic Variables

- Bound to heap storage only when they are assigned values
- e.g. In Python you can create a list `L = [1,2,3,4]`, even if `L` was used for some other variable before
- Very flexible
- Can be inefficient with many allocations/deallocations
- Can be hard for compilers/interpreters to detect errors

Scope

- Scope is the set of statements in which a variable is visible
- Scope and lifetime are not necessarily the same (e.g. static local variable)
- Static scoping: scope can be determined before run time
- Many of the modern languages are block-structured and allows for new local scopes to be created
- Global scope: in the same source file or visible across source files (e.g. `extern` in C/C++)
- Variables with the same name can be hidden

Dynamic Scoping

- Scope of variables depend on the calling sequence of functions
- Poor readability, not reliable
- Not possible for static type checking

Lifetime and Scope

- Lifetime and scope are often correlated, but not always
- e.g. static local variables in C++
- In nested function calls (e.g. f calls g), the local variables in the caller function are alive but not in the scope of the called function
- Sometimes, scope and lifetime mismatch can lead to problems (e.g. memory leak)

Referencing Environment

- The referencing environment of a statement is a collection of all variables that are visible in that statement
- Static-scoped language: variables declared in local scope and all variables in surrounding scopes
- Compilers maintain referencing environment to generate code
- Dynamic-scoped language: a list of active subprograms is maintained at runtime
- Active subprograms: a subprogram has begun but not yet finished
- Referencing environment are local variables and all visible variables in the sequence of active subprograms

Named Constants

- Name constants are variables that is bound to a value once
- Improve readability and reliability
- Easier to change
- Some languages allow only static binding. Others allow dynamic binding (e.g. C++)