Variables, Bindings and Scopes

• Variables   
• Initialization   
• Constants   
• Binding   
• Type Checking   
• Scope

This concepts are common among many paradigms. Mainly it is a concept for imperative programming (bunch of data manipulated through a set of imperative instructions). You need to have some semantic to manipulate your memory. These are gonna be done through variables.

**VARIABLES**

Imperative languages are abstractions of von Neumann architecture

* Memory / Storage
* Processor / CPU

Variable: an abstraction of a computer memory cell or collection of cells

Variables are characterized by attributes

Their design must consider

* Location
* Referencing
* Scope
* Lifetime
* Type checking
* Initialization
* Type compatibility

64 bit machine 🡪 64 bit is the base unit for the memory as well as the instructions. As a programmar, I don’t care about these things. Abstraction should help me.

Variables are characterized by some attributes.

Reminder for Von Neumann Architecture

Diagram

Description automatically generated

This architecture is instantiation of a turing machine (Although simple, the machine can simulate any computer algorithm, no matter how complicated it is).

Variable Attributes

Variable is an abstraction of memory cell(s)

Characterized by six attributes:

* Name (usually)
* Address (in memory)
* Value (if initialized)
* Type (range of values, interpretation, operations)
* Lifetime
* Scope

Name - not all variables have them!

Characters

Character Sets

* 40 character BCD
* 48 Fortran II
* 128 ASCII characters
* Unicode
* Some Character Categories
  + <digit> 🡪 0|1|2|3|4|5|6|7|8|9
  + <lcletter> 🡪 a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
  + <ucletter> 🡪 A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
  + <underscore> 🡪 \_

Names

Set of characters

Names, or identifiers, are used for variables, subprograms (or methods), types (user defined), classes, etc.

Not names

* Comments, line boundaries, white spaces

Tokens

* Punctuations, operators, numbers and other literals
* Names (identifiers)
  + In the tokenization process, tokens will include the identifiers as well
* Keywords, reserved words

Variable Names

User-defined names

Design issues for names:

* Maximum length?
  + Long names take more space in my program code. Also lexer can go forever then you have some inefficiency (in terms of memory to store that code, parsing that code) issues.
* Are connector characters allowed?
  + Making the readability easy.
* Are names case sensitive?
* Are some words reserved words or keywords?
* ONLY UPPERCASE LETTERS ALLOWED?

Name Length

If too short, they can’t explain the meaning

* Readability, writability

If too long?

Language examples:

* FORTRAN I: maximum 6
* COBOL: maximum 30
* FORTRAN 90 and ANSI C: maximum 31
* Ada and Java: no limit, all are significant
* C++: no limit, but implementers often impose one

What is a good maximum length?

Connectors / Non-Letter Characters

Pascal \_

* underscore only

Modula-2, and FORTRAN 77

* none allowed

Java

* $,\_ (but $ is by the system used for inner classes)

Common Lisp

* many, including \*,+,-,\_, …
* doesn’t use infix, it uses prefix so you can use x\*y as variable name but it is not very good

Advantages of having connectors?

Disadvantages of having connectors?

Case Sensitivity in Variable Names

Many languages are not case sensitive

C, C++, and Java names are case sensitive

Common Lisp: case sensitive (default)

Prolog: case of first character determines whether it's constant or variable!

Disadvantage: readability

* names that look alike are different

C++ and Java: predefined names are mixed case

* e.g. IndexOutOfBoundsException)

Effects on writability?

Named Constants

Named constant

* variable bound to a value only when it is bound to storage

Advantages:

* readability and modifiability

Used to parameterize programs

The binding of values to constants can be either static (called manifest constants) or dynamic

* Pascal: literals only
* FORTRAN 90: constant-valued expressions
* Ada, C++, and Java: expressions of any kind

Variable Addresses

*Memory address associated with variable*

* *also called l-value*

Location may change during execution

* i.e. may have different addresses at different times
* you have x variable in function f, every time you call f function x’s l-value may change

Aliases

* variable names that refer to the same memory location
* C variables doesn’t have aliases. Indirect access to pointer could mean the same thing but it’s not alias (in C).

Aliases often reduce readability

* program readers must remember that changing value of one variable means a change of all others

Variable Aliases

Created by

* pointers
* reference variables
* C and C++ unions
* Pascal variant-records
* and by parameters!

Original reasons for aliases are no longer valid

* memory limits forced re-use of space in FORTRAN

Instead, use dynamic allocation

Variable Types and Values

Type - determines the

* range of values
* set of operations that are defined for the variable
* interpretation of bit patterns
* for floating point, determines the precision

*Value - contents of the memory location associated with the variable*

* *r–value (when appearing on right side of assignment)*

Abstract memory cell – (graphical) representation of collection of cells associated with a variable

Shape, rectangle

Description automatically generated

*Memory location: l-value  
Value itself: r-value (bit pattern hold in the cell)*

*l\_value = r\_value*

The Concept of Binding

A variable needs to have a few different bindings.

*The l-value = address*

*The r-value = value*

*Binding = association*

* *for a variable, the association of l-value to r-value*

Binding time

* time (during execution) when l-value is associated with r-value

Variable Initialization

Initialization

* the initial binding of a variable to a value

Often done with the declaration statement

* e.g., in Java
  + int sum = 0;
* e.g., in Lisp
  + (defvar sum 0)
* e.g., in Clojure
  + ?

Possible Binding Times

Language design time

* e.g., bind operator symbols to operations

Language implementation time

* e.g., bind floating point type to a representation

Compile time

* e.g., bind a variable to a type in C or Java

Load time

* e.g., bind a FORTRAN 77 variable to a memory cell, or
* bind static variable in C or Java

Runtime

* e.g., bind a non-static local variable to a memory cell

Variable cannot be done at language design time or language implementation time.

Static vs. Dynamic Binding

Static

* first occurs before run time and
* remains unchanged throughout program execution

Dynamic

* first occurs during execution, or
* can change during execution of the program.

Type Bindings

How is a variable’s type specified?

When does the binding take place?

* At declaration time (compile time) or it could be dynamic at runtime or loadtime as well

If static, type specified by either

* explicit declaration or
* implicitly

Implicit vs. Explicit Declaration

Explicit type declaration (int x; -----> we are binding the type of the variable to a certain type)

* as program statement

Implicit type declaration (x=y ------> if x isn’t defined, y’s type is gonna define x’s type)

* default mechanism - at the first appearance in the program
* e.g., in FORTRAN, BASIC, and Perl

Type Inferencing (ML, Miranda, and Haskell)

* Type is determined from the context of the reference

Unification (Prolog)

* Variables are unified with other variables or constants during the resolution process

Advantage of implicit declaration: writability

Disadvantage: reliability, readability

Dynamic Binding

JavaScript, PHP, Common Lisp

Specified through an assignment statement e.g., JavaScript

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

Advantage: flexibility (generic program units)

Disadvantages:

* Speed penalty for type checking and interpretation
* Type error detection by the compiler is more difficult

Variable Lifetime

Lifetime

* the time during which it (variable) is bound to a particular memory cell

Allocation

* getting a cell from a pool of available cells

De-allocation

* returning a cell back into the pool

Static Variables

(at load time)

Graphical user interface, text, application

Description automatically generated

Stack Dynamic Variables

Storage bindings are created for variables when their declaration statements are elaborated (storage allocation and binding)

* Allocated from the run-time stack
* De-allocate after execution (garbage collection)
* Can happen at the beginning of block or anywhere

Advantage:

* allows recursion
* conserves storage – allocate when and where needed

Disadvantages (compared to static vars):

* Overhead of allocation and de-allocation
* Subprograms cannot be history sensitive
* Indirect addressing (longer time)

Diagram

Description automatically generated with medium confidenceprogram

static

stack

heap

stack and heap are dynamic part.

Runtime Stack (Dynamic)

Graphical user interface, application, table

Description automatically generated

Stack Dynamic Variables

If scalar, all attributes except address are statically bound

* e.g. local variables in C subprograms and Java methods

Examples

* Variables defined in methods in Java, C, C#

Heap Variables (Dynamic)

A screenshot of a computer

Description automatically generated with low confidence

The heap is a collection of highly disorganized storage cells for unpredictable use…

Avoid reallocation everytime you get into block. Allocate one time, use it forever.

Explicit Heap-Dynamic Variables

Nameless abstract memory cells (heap)

* Allocated and de-allocated by explicit directives
* Specified by the programmer
* Takes effect during execution
* Referenced only through pointers or references
* e.g. dynamic objects in C++ (via new and delete)

int \* i;

i = new int;

…

delete i;

* all objects in Java

Advantage:

* provides for dynamic storage management (flexibility)

Disadvantage:

* inefficient by comparison (cost of reference)
* reliability must be shown

Implicit Heap-Dynamic Variables

Bound to heap storage automatically

* Only when they are assigned values
* All attributes are bound every time they are assigned

Allocation and de-allocation caused by assignments

* e.g. all variables in APL
* all strings and arrays in Perl an JavaScript

Example: Java statement

* “highs = [74, 84, 490, 44, 45];”
* “highs” is now an array …

Advantage:

* Flexibility
* Writability

Disadvantages:

* Inefficient, because all attributes are dynamic
* More work to do error detection

Variable Bindings

Static

* E.g., C static variables

Stack dynamic

* E.g., C method variables

Heap dynamic variables

* Explicit Heap-Dynamic Variables
  + E.g., dynamic objects in C++ (new and delete)
* Implicit Heap-Dynamic Variables
  + E.g., arrays in JavaScript

Variables

Six attributes

* Name (usually)
* Address (in memory)
* Value (if initialized)
* Type (range of values, interpretation, operations)
* Lifetime
* Scope

Type Checking

Type checking ensures that the operands and the operator are of compatible types

Generalized to include subprograms and assignments

Compatible type is either

* legal for the operator, or
* language rules allow it to be converted to a legal type

Coercion

* Automatic (implicit) conversion

Type error

* Application of an operator to an operand of incorrect type

Nearly all type checking can be static for static type bindings

Type checking must be dynamic for dynamic type bindings

Strong Typing

Strongly typed programming language

* PL where type errors are always detected

Advantages:

* Reliability
* Detection of the misuses of variables that result in type errors

Disadvantages:

* Increased size of code
* Slower development – declarations
* Reduced writability
* Exception handling may be more complicated

Strong Typing in PLs

Examples:

* FORTRAN 77 is not: parameters, EQUIVALENCE
* Pascal is not: variant records
* C and C++ are not: parameter type checking can be avoided; unions are not type checked
* Ada is, almost (UNCHECKED CONVERSION is loophole)
* Java is similar to Ada

Coercion rules affect strong typing

* can weaken it considerably (C++ versus Ada)
* C++ is less reliable than Ada (less coercion)

Java has just half the assignment coercions of C++

Java’s strong typing is still less effective than in Ada

Coercion in Java

Graphical user interface, text, application

Description automatically generated

Type Compatibility

The result of two variables being of compatible types is that either one can have its value assigned to the other

Two different type compatibility methods:

* Name compatibility
* Structure type compatibility

Most languages use combinations of the different techniques

There are some variations of these two methods

If I can replace variable x’s value with variable y’s value or y’s value with x’s, then if I don’t need to do anything else, if I can do this replacement, I call these two types compatible.

Name Type Compatibility

Variables have compatible types if they are either

* in the same declaration
* or in declarations that use the same type name
* ---
* x and y are compatible types if it is sth like this:
  + int x,y; ---------> same declaration
  + int x;
  + … in declarations that use the same type name
  + int y;
* If two types’ (Ta and Tb) structures are same, they are not compatible because their names are different

Easy to implement but highly restrictive

* Sub-ranges of integer are not compatible with integer (as in Pascal)
* Formal parameters must be the same type as their corresponding actual parameters (Pascal)

Structure Type Compatibility

Variables have compatible types if their types have identical structures

* More flexible
* But harder to implement
  + Because of structured types (comparison of whole structures instead of names)
  + Others are much simpler

Questions:

* Structurally the same record types with different field names?
* Array types with the same base type but different subscripts? E.g. [1..10] vs. [0..9]
* Enumeration types whose components are spelled differently?

With structural type compatibility, you can not differentiate between types of the same structure

* E.g. different units of speed, when both are floating point

Type Compatibility Examples

Pascal: usually structure, but in some cases name is used (formal parameters)

C: structure, except for records (it uses name compatibility)

Ada: restricted form of name

* Derived types allow types with the same structure to be different
* Anonymous types are all unique, even in:
  + A, B : array (1..10) of INTEGER

Graphical user interface, text, application

Description automatically generated