Theory of Programming Languages

Week 4

VARIABLES

Names, Bindings, Type Checking, Scopes

Overview (I) – Topics

We will discuss the fundamental semantic issues of **variables** – including:

I. NAMES: The nature of names and special words in programming languages
II. BINDINGS: The attributes of variables: type, address and value Binding and binding times
III. TYPE CHECKING: strong typing and type compatibility rules
IV. SCOPE: static and dynamic

Overview (II) – The basic architecture

- The von Neumann architecture has two basic components: **memory** (containing both data and code), and the processor.
  - A variable in a programming language is an abstraction of the memory-cells in a von Neumann architecture
  - The abstraction may be slight (e.g. for an int or a char) or it maybe significant (e.g. a 3d array)
  - Variables are described by a collection of properties – or attributes, including name, type, address and scope.

(I) Nature of names and special words in a Programming Language

- Names (or identifiers) are associated with labels, subprograms, formal parameters and other program constructs.
- **Design issues with names**
  - What is the max length
  - Can connecter characters be used
  - Are names case sensitive
  - Are the special words *reserved words* or *keywords*

Length, connecters and case-sensitivity

- Name lengths range from single characters, to six characters (Fortran), to 31 characters (Fortran 90 and C), unlimited characters (Java, C#) – What is the design issue in allowing long names?
- Most contemporary languages allow connectors, like underscores.
- A case sensitivity language violates a fundamental design principle: “Language constructs that look the same should have the same meaning.”
  - Is case sensitivity good or bad for readability?

Special words make programs more readable by specifying the actions to be performed, and, separating syntactic entities.

Two kinds of special words are used in languages:

- **Keyword:** is a word of a programming language that is special only in certain contexts.
  e.g. Fortran keyword **REAL**
  REAL identifier
  REAL = 3.42
- **Reserved word:** is a special word of a language that cannot be used as a name.
  - Better than keywords because the ability to redefine keywords can lead to readability issues.
    - Example, from Fortran (keywords), following two statement are valid but create readability hazards :
      REAL INTEGER
      INTEGER REAL
  - On the other hand, a great number of reserved words create writability (and learnability) issue. For example, COBOL has 400 reserved words – all of which are difficult to remember.

Same program in C, and 2 version in COBOL

```
// Calculation in C:

if (salaried)
        amount = 40 * payrate;
else
        amount = hours * payrate;
```

```
* Calculation in COBOL:

IF Salaried THEN
        MULTIPLY Payrate BY 40 GIVING Amount
ELSE
        MULTIPLY Payrate BY Hours GIVING Amount
END-IF.
```

```
* Other example of calculation in COBOL:

IF Salaried
        COMPUTE Amount = Payrate * 40
ELSE
        COMPUTE Amount = hours * payrate
END-IF.
```

(II) Attributes of variables

A variable is generally represented as a sixtuple of attributes:

(name, address, value, type, lifetime, scope)

Other concepts related to these attributes include:

- Aliases
- Binding
- Binding times
- Declarations
- Type checking
- Strong typing
- Scoping rules
- Referencing environments

Address

- Two variables with the same name can have different addresses at different places, or at different times in a program
  - e.g. local variables of the same name in various functions, have different addresses because of different places.
  - e.g. local variables within a recursive routine will get different addressed at different times, during execution.

This issue will be discussed later in detail.

Aliases allow multiple identifiers (names) to refer to a variable at the same address.

- Aliases make it difficult to read a program (e.g. A and B refer to the same variable, hence, the programmer must remember that changing A will change B)
- Two pointer variables are aliases when they point to the same memory location.
- A pointer, when dereferenced, and its corresponding data variable are also aliases.

Why may a programming language allow aliases?

The **type** of a variable determines the range of values that the variable can have and the set of operations that are defined for the variable of this type.

The extent to which a programming language discourages or prevents **type errors**, is the type-safety of that language: as we will see in detail later.

The **value** (r-value)of a variable is the content of the memory cell (or cells) associated with the variable.

We defined an **abstract memory-cell** to have the size required by the variable to which it is associated. e.g. a floating point variable occupies four physical memory cells, but a single abstract memory cell.

The Concept of Binding

Binding is an association, such as, between an attribute and an entity or between an operation and a symbol.

- The time at which the association takes place is called the **binding time**.
- Binding may take place at: language design time, language implementation time, compile time, link time, load time, or run time.

Examples of binding at different times:

- The '*' symbol is bound to the multiplication operator at language design time.
- A data type, such as integer in FORTRAN, is bound to a range of possible values at language implementation time.
- At compile time, a variable in a Java program is bound to a particular data type.
- A call to a library subprogram is bound to the subprogram code at link time.
- A variable may be bound to a storage cell when the program is loaded into memory.
- The type of a template variable in C++ is bound to the variable at run time (this is late binding, and does reduce program speed).

The bindings involved in a small piece of code

Following are the various bindings involved in the code:

```
int count ;
 ...
count = count+ 5
```

- **Set of possible types for count:** bound at language design time.
- **Type of count:** bound at compile time.
- **Set of possible values of count:** bound at compiler design time.
- **Value of count:** bound at execution time with this statement.
- **Set of possible meanings for the operator symbol +:** bound at language definition time.
- **Meaning of the operator symbol + in this statement:** compile time.
- **Internal representation of the literal 5:** bound at compiler design time.

Why is it important to understand binding?

- A complete understanding of the binding times for the attributes of program entities is a prerequisite for understanding the semantics of a programming language.
  - For example, to understand what a subprogram does, one must understand how the actual parameters in a call are bound to the formal parameters in its definition. To determine the current value of a variable, you may need to know when the variable was bound to storage.

Binding of attributes to variables

Static, Dynamic and Hardware bindings

- A binding is **static** if it first occurs before run-time and remains unchanged throughout program execution.
- A binding is **dynamic** it first occurs during run-time and can change during the execution of a program.
- A third kind of binding is managed by the hardware (and/or the operating system), such as, the binding of a variable to an actual physical address in the main memory. Such **system controlled bindings** are not relevant to our discussion of programming languages.

Static type bindings

- Before a variable can be referenced in a program it must be bound to a data type.
- Important to note :
  1) how the type is specified
  2) when the binding takes place
- Static binding of types may take place through **explicit or implicit declarations**.

- An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type.
  - Most programming languages designed since the mid-1960s require explicit declarations of all variables.
  - This is the type of declaration you have been using most frequently, in all the modern imperative programming languages, such as, C++, JAVA and C#
- An **implicit declaration** is a means of associating variables with types through default conventions instead of declaration statements.
  - Several widely used languages whose initial designs were done before the late 1960s, have implicit declarations.
    - e.g. In Fortran, if an undeclared identifier begins with one of the letters I, J, K, L, M, or N, it is implicitly declared to be integer type; otherwise, it is implicitly declared to be real type.
    - This creates a trade-off between convenience and reliability. An accidentally undeclared identifier may be given unexpected types and hence values.
    - Sometimes, to avoid problems with implicit declarations, the variable names are required to state with special characters, for example, $apple, or @apple, or %apple declare a single variable, an array and a hash map, respectively.

HW) Given an example of modern language that uses implicit declaration. Explain, with examples, how the unreliability problem is addressed in that language.

Dynamic type binding

- In dynamic type binding the type of a variable is not specified by a declaration statement.
- The type is detected from the 'value' that is 'assigned' to a variable at run time.
- Languages in which dynamic binding is allowed are dramatically different from those in which types are only statically bound.
- The advantage of dynamic bounding is that it provides programming flexibility:
  - e.g. writing a **generic function** that processes a list, of any kind of data-ype.

Disadvantages of dynamic type-binding

- Reduced error-detection capability of the compiler: because any two types can appear on opposite sides of the assignment operator.
  - e.g. a and b are integers, and c is a float – a program mistakenly types: a=c, in place of a=b; and the compiler does not report error.
- Dynamic binding requires high execution-time cost, as type-checking is done at run time.
  - Every variable must have a descriptor associated with it to store the current type of the variable.
  - The storage used for the value of a variable may also vary as the type changes, and reallocation of space costs time.
  - Usually, dynamically bound languages are run on pure interpreters as it is difficult to generate efficient machine code by a compiler, for a language where types change at run time.

Type inference in ML (Metalanguage)

- **ML** is a general-purpose functional programming language developed in the early 1970s
- It is a purely **type inferring language,** as it automatically infers the types of most expressions without requiring explicit type annotations.
- **Examples of valid ML programs**

```
fun circumf(r) = 3.14159 * r * r;
fun times10(x) = 10 * x;
```

- **An invalid ML program, and corrected versions**

```
fun square(x) = x * x;

fun square(x : int) = x * x;
fun square(x) = (x : int) * x;
fun square(x) = x * (x : int);
```

Storage Bindings and Lifetime

The fundamental character of a programming language is in large part determined by the design of the storage bindings for its variables.

- The processes of **allocation** and **de-allocation**, bind and unbind, respectively, abstract memory-cells corresponding to a variable.
- The **lifetime** of a variable is the time during which the variable is bound to a specific memory location.
- According to their lifetimes, variables can be classified into four categories, which a programming language must separately facilitate:
  - Static
  - Stack Dynamic
  - Explicit Heap Dynamic
  - Implicit Heap Dynamic

Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

- Global variables, which are referenced throughout the execution of a program are ideally static.
- History sensitive variables within functions retain the same storage and hence are static.
- Static variables add efficiency to a programming language, as repeated allocation and de-allocation is not required
- A language with only static variables cannot support recursive subroutines. Why?
- Therefore, static variables result in inflexibility.
- Examples: Fortran (all variables are static), C++/Java (only the variables of choice are static), Pascal (no static variables)

What kind of a variable is **x**, in the following C++ code?

```
unsigned fact(unsigned n){
        unsigned x=n-1;
        if(n==0) return 1;
        else return x*n;
}

// How many times will x be allocated when fact(20) executes?
// Will x get a new address each time?
// Why can x not be static?
```

Stack-dynamic variables are those whose storage bindings are created when their declaration statements are **elaborated**, but whose types are statically bound. The variable **x** in the previous example is a Stack-dynamic variable.

- In JAVA and C++,local variables are by default stack dynamic.
- All attributes, other than storage, are statically bound to the stack-dynamic variables.
- We will do a detailed study of how a programming language can manage the allocation and de-allocation of stack-dynamic variables.

Explicit heap-dynamic variables are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions specified by the programmer.

Example:

```
int * x = new int; //c++
```

- Some languages, like C++, provide an explicit command to deallocate explicit heap-dynamic variables.
- Java objects are explicit heap-dynamic and are accessed through reference variables. Java has no way of explicitly destroying a heap-dynamic variable; rather, implicit garbage collection is used.
- Advantages: great flexibility
- Disadvantage: run-time costs, plus, writability problems.

Implicit heap-dynamic variables are bound to heap storage only when they are assigned values.

- In a sense, they are just names that adapt to whatever use they are asked to serve.
- Advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written.
- The disadvantage is the run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others.
- Another disadvantage is the reduction of error-checking by the compiler.

In the current standard of C++, the following lines are valid:

```
int x=5;
int array1[x];          int array2[5];
```

Question) At what times in the program-lifecycle are the variables array1 and array3 bound to their storage?

Question) Does array1 get space of the execution stack? Or the heap?

Memory Management

Memory management is the act of managing computer memory.

- The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed.
- Different languages use different "models" for providing memory management.
  - Examples, C++ provides explicit allocation and de-allocation statements.
  - Java provides a garbage collector.
  - Objective C: reference counting: a garbage collection algorithm that uses these reference counts to deallocate objects which are no longer referenced.
  - Memory management could be the biggest hurdle (or the reason to choose) to learn a new programming languages.
  - Memory management could be crucial to the running time and flexibility of a program written in a certain programming language.
  - Class presentation of Memory Management Techniques