Chapter 5

Variables

Topics

- Imperative Paradigm
- Variables
- Names
- Address
- Types
- Assignment
- Binding
- Lifetime
- Scope
- Constants

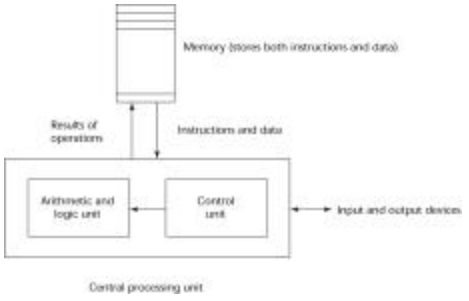
Imperative Paradigm

- The most widely used and well-developed programming paradigm.
- Emerged alongside the first computers and computer programs in the 1940s.
- Its elements directly mirror the architectural characteristics of most modern computers
- This chapter discusses the key programming language features that support the imperative paradigm.

Von Neumann Architecture

- The architecture of the von Neumann machine has a memory, which contains both program instructions and data values, and a processor, which provides operations for modifying the contents of the memory.

Von Neumann Architecture



Programming Language: Turing Complete

- *Turing complete*: contains integer variables, values, and operations, assignment statements and the control, constructs of statement sequencing, conditionals, and branching statements.
  - Other statement forms (while and for loops, case selections, procedure declarations and calls, etc) and data types (strings, floating point values, etc) are provided in modern languages only to enhance the ease of programming various complex applications.

Imperative Programming Language

- Turing complete
- Also supports a number of additional fundamental features:
  - Data types for real numbers, characters, strings, Booleans and their operands.
  - Control structures, for and while loops, case (switch) statements.
  - Arrays and element assignment.
  - Record structures and element assignment.
  - Input and output commands.
  - Pointers.
  - Procedure and functions.

Variables

- A *variable* is an abstraction of a memory cell or collection of cells.
  - Integer variables are very close to the characteristics of the memory cells: represented as an individual hardware memory word.
  - A 3-D array is less related to the organization of the hardware memory: a software mapping is needed.

Variables: attributes

- A variable can be thought of as being completely specified by its 6 basic attributes (6-tuple of attributes).
  1. Name: identifier
  2. Address: memory location(s)
  3. Value: particular value at a moment
  4. Type: range of possible values
  5. Lifetime: when the variable is accessible
  6. Scope: where in the program it can be accessed

Names

- Names have broader use than simple for variables.
- *Names* or *identifiers* are used to denote language entities or constructs.
  - In most languages, variables, procedures and constants can have names assigned by the programmer.
- Not all variables have names:
  - Can have a nameless (anonymous) memory cells.

Names

- We discuss all user-defined names here.
- There are some clear design issues to consider:
  - Maximum length?
  - Notation?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names: length

- If too short, they may not convey the meaning of the variable.
- It too long, the *symbol table* of the compiler might become too large.
- Language examples:
  - FORTAN I: maximum 6
  - COBOL: maximum 30
  - FORTAN 90 and ANSI C: maximum 31
  - Ada and Java: no limit and all are significant
  - C++: no limit, but implementers often impose one

Names: notation

- Variables can consist of one or more letters, numbers (as long as a number is not the first character), and an underscore character (the underline key.)

  <ident> ::- <letter> { <letter> | <digit> | '_' }

- Some old languages allowed embedded spaces which were ignored
  - FORTRAN 90:

    Sum Of Salaries vs. SumOfSalaries

Names: "standard" notation

- Some standards can be applied to how variables are named when one word is used to describe a variable.
- ***Camel*** notation
  - Uses capital letters to indicate the break between words.
  - Camel is named such because the capital letters separating the words look like little camel humps
  - Example: CostOfItemAtSale

Names: "standard" notation

- ***Underscore*** notation
  - Uses an underscore to separate words that make up a variable.
  - Example: Cost_of_item_at_sale
- Some other standards are used to identify the data type stored in the variable

Names: "standard" notation

- ***Hungarian*** notation
  - Uses two letters, both lower-case
    - First letter indicates the scope of the variable
    - Second letter indicates the type of the variable
  - Example: l_fCostOfItemAtSale
- ***Prefix*** notation
  - Uses a prefix (usually three letters) to indicate the type of variable.
  - Example: floCostOtItemAtSale

Variable name	Explanation
i	This is a really bad variable to use. You can't tell what it contains and if anyone wants to fix it later, a simple search and replace will be very tedious since single letters are used in words as well.
lastname	This is much better but uses no form of notation.
LastName	This is camel notation
strLastName	This is prefix – camel notation. Note that the prefix is in all lower case.
last_name	This is underscore notation. As with camel notation, you can easily identify the two words that make up the variable name
str_last_name	This is prefix underscore notation. Again, the prefix is in lower case.
lcLastName	This is Hungarian camel notation. The first two letters tell us what type of variable is used. In this case, this variable contains a last name, is local to the function/procedure, and is a character string.
lc_last_name	This is Hungarian underscore notation.

Names: case sensitivity

- FOO = Foo = foo ?
- Disadvantage:
  - Poor readability, since names that look alike to a human are different
  - Worse in some languages such as Modula-2, C++ and Java because predefined names are mixed case
    - IndexOutOfBoundsException

Names: case sensitivity

- Advantages:
  - Larger namespace
  - Ability to use case to signify classes of variables (e.g. make constants be in upper-case)
- C, C++, Java, and Modula-2 names are case sensitive but the names in many other languages are not.
- Variable in Prolog have to begin with an upper case letter.

Names: special words

- Used to make programs more readable.
- Used to name actions to be performed.
- Used to separate the syntactic entities of programs.
- *Keyword*
  - A word that is special only in certain contexts.
  - Example: in FORTRAN the special word `Real` can be used to declare a variable, but also as a variable itself

Names: special words

- `Real TotalSale` (variable `TotalSale` is of type Real)
- `Real = 3.1416` (`Real` is a variable)
- `Integer Real` (variable `Real` is of type Integer)
- `Real Integer` (variable `Integer` is of type Real)

- Disadvantage: poor readability
  - Distinguish between names and special words by context.
- Advantage: flexibility

Names: special words

- *Reserved Word*
  - A special word that cannot be used as a user-defined name.
  - Example: C's `float` can be used to declare a variable, but not as a variable itself.

Variables: Address

- The memory address with which a variable is associated.
  - Also called *l-value* because that is what is required when a variable appears in the LHS of an assignment.
- A variable (identified by its name) may have different addresses at different places in a program
  - Example: variable `X` is declared in two different subprograms (functions)

Address

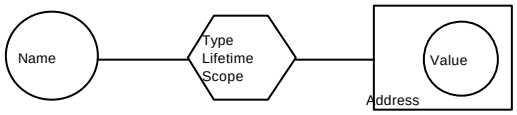
- A variable may have different addresses at different times during execution
  - Example: variable `X` of a subprogram is allocated from the runtime stack with a different address each time the subprogram is called (e.g. recursion).

```
#include <stdio.h>

// --------- Prototype ---------
void foo();
void bar();
// ---------- Definition ---------
void foo()
   {
        int x;
        printf("The address of x in foo() is: %d\n", &x);
   }
void bar()
   {
        printf("Called from bar(),");
        foo();
   }

// ---------- main ----------
int main()
   {
        int i = 0;
        foo();
        bar();
        sleep(30000);
        return 0;
   }
```

```
The address of x in foo() is: 1244964
Called from bar(). The address of x in foo() is: 1244880
```

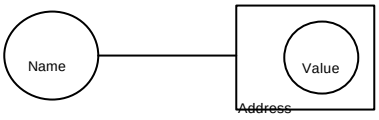

Variables: address

- A schematic representation of a variable can be drawn as:



Variables: address

- Concentrate on name, address and value attributes
  - Simplified representation:



Variables: address (aliases)

- If two variable names can be used to access the same memory location, they are called aliases.
- Aliases are harmful to readability
  - Program readers must remember all of them.
  - They are useful in certain circumstances.
- Example:

```
int i, *iptr, *jptr;
iptr = &i;
jptr = &i;
```

  - A pointer, when de-referenced (*iptr) and the variable's name (i) are aliases

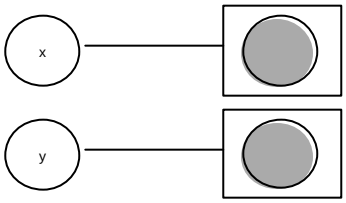
Aliases

- Aliases can occur in several ways:
  - Pointers
  - Reference variables
  - Pascal variant record
  - C and C++ unions
  - FORTRAN equivalence
  - Parameters
- Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN
  - Replace them with dynamic allocations.

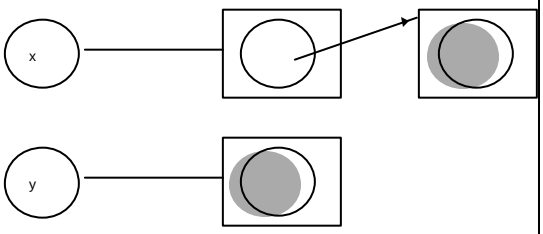

```
type intptr = ^integer;
var x, y: intptr;

begin
   new(x);
   x^ := 1;
   y := x;
   y^ := 2;
   writeln(x^);
end;
```

After the assignment of x to y, y^ and x^ both refer to the same variable, and the preceding code prints 2.

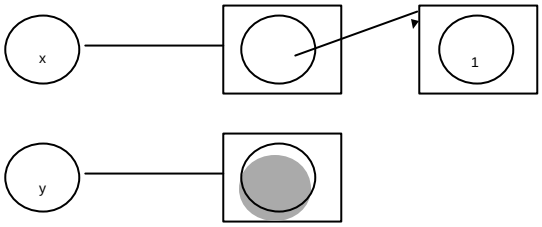
- After the declarations, both x and y have been allocated in the environment, but the values of both are undefined
  - Indicated in the diagram by shading in the circles indicating values.



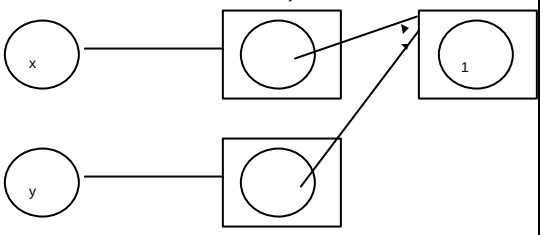
- After the call to new(x), x^ has been allocated, and x has been assigned a value equal to the location of x^, but x^ is still undefined



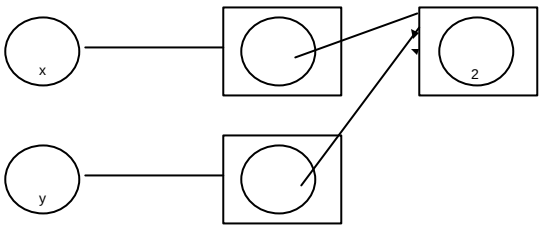
- After the assignment x^ := 1



- The assignment y := x now copies the value of x to y, and so makes y^ and x^ aliases of each other (note that x and y are not aliases of each other)



- Finally, the assignment y^ := 2 results in



Variables: type

- Determines the range of values of variables
- Set the operations that are defined for values of that type
- Example: in Java, int type:
  - Value range of –2,147,483,648 to 2,147,483,647
  - Operations: addition, subtraction, multiplication, division, and modulus.

Variables: value

- Contents of the location with which the variable is associated.
- *Abstract memory cell*
  - The physical cell or collection of cells associated with a variable
    - The smallest addressable cell is a byte.
    - But most types (system-defined or user defined) take more.
    - Abstract memory cell refers to the number of cells held by a variable.
      - Example: `float` uses 4 bytes on most machines.

lvalue and rvalue

- Are the two occurrences of *a* in this expression the same?

```
a := a + 1;
```

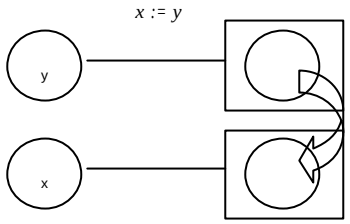
- In a sense:
  - The one on the *left* of the assignment refers to the location of the variable whose name is *a*
  - The one on the *right* of the assignment refers to the value of the variable whose name is *a*

```
a := a + 1;
```

address value

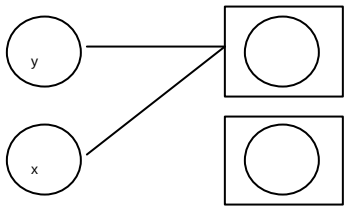
Assignment

- To access an *rvalue*, a variable must be determined (dereferenced) first.



Assignment

- (Some languages) Different meaning to assignment: locations are copied instead of values.



Assignment

- Assignment by sharing. An alternative is to allocate a new location, copy the value of y, and bind x to the new location