Chapter 5

Variables

Topics

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

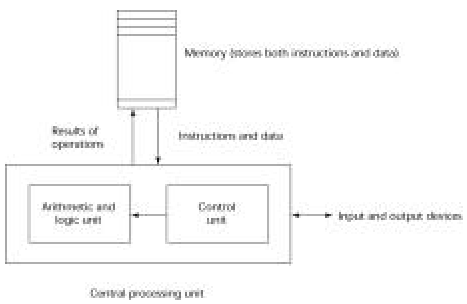
Imperative Paradigm

- 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

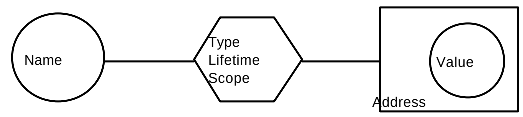
- A variable may have different addresses at different times during execution
  - Example: variable X of a subprogram is allocated from the runtime stack with a different address each time the subprogram is called (e.g. recursion).

```
#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

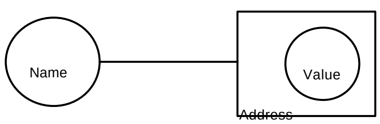
Variables: address

- 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

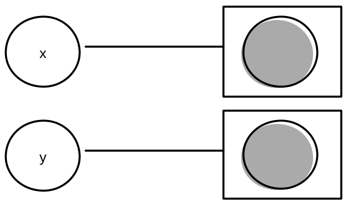
Aliases

- 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.

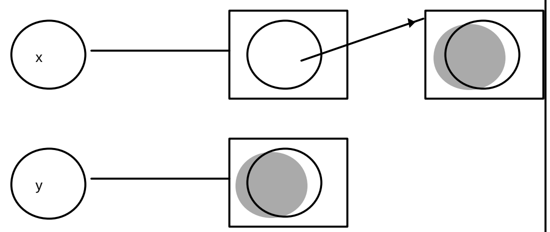

```
type intptr = ^integer;
var x, y: intptr;

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.

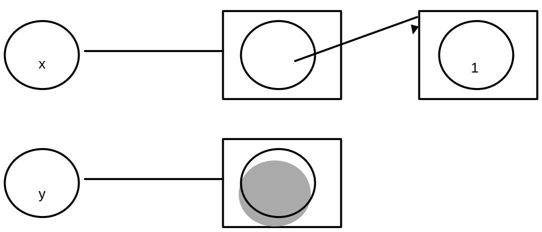
- After the declarations, both x and y have been allocated in the environment, but the values of both are undefined
  - Indicated in the diagram by shading in the circles indicating values.



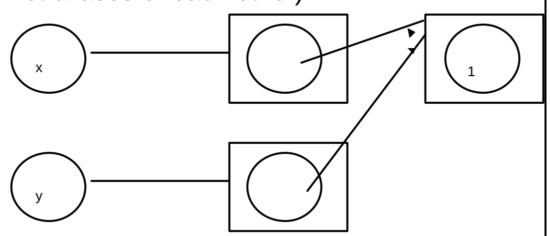
- After the call to new(x), x^ has been allocated, and x has been assigned a value equal to the location of x^, but x^ is still undefined



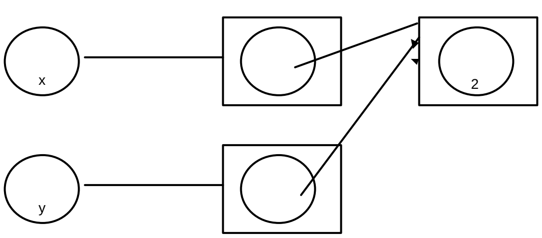
- After the assignment x^ := 1



- 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*

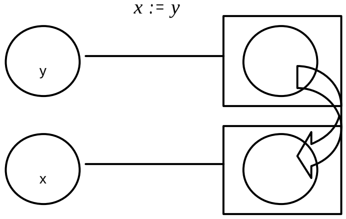
  a := a + 1;

  address value

Assignment

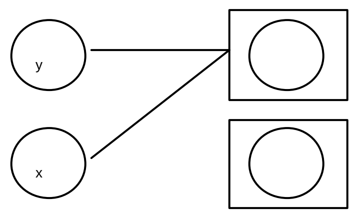
- To access an *rvalue*, a variable must be determined (dereferenced) first.

x := y



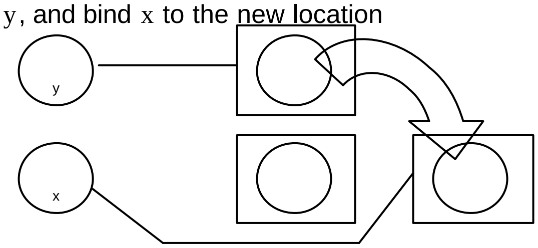
Assignment

- (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



Binding

- The assignment statement is really an instance of a more general phenomenon of attaching various kinds of values to names.
- The association of a name to an attribute is called *binding*
  - Assignment statement binds a value to a location.
  - Identifiers are bond to locations, types, and other attributes at various points in the translations of a program.

Binding

- *Binding time.* Bindings happen at different and invisible points.
- Possible binding times
  1. Language design time
     - Bind operator symbols to operations
       - Example: bind * to multiplication
  2. Language implementation time
     - Example: bind floating point type to a representation (IEEE floating-point format)
     - Example: the data type int in Java is bound to a range of values.

Binding

3. Compile time
   - Example: bind a variable to a type in C or Java
4. Link time
   - Example: bind a call to a library function to the function code.
5. Load time
   - Example: bind a C static variable to a memory cell.
6. Runtime
   - Example: bind a nonstatic local variable to a memory cell

The Concept of Binding

- Consider the following:

```
int C;
C := C + 5;
```

- Some of the bindings and their binding times are:
  - The type of C is bound at *compile* time.
  - The set of possible values of C is bound at *compiler design* time.
  - The meaning of the operator + is bound at *compile* time (when the types of its operands have been determined)
  - The internal representation of the literal 5 is bound at *compiler design* time.
  - The value of C is bound at *run* time.

Static and Dynamic Binding

- A binding is *static*
  - it occurs before run time and
  - It remains unchanged throughout program execution
- A binding is *dynamic*
  - It occurs during execution or
  - It can change during execution of the program
- As binding time gets earlier:
  - Efficiency goes up
  - Safety goes up
  - Flexibility goes down

Type Bindings

- A variable must be bound to a data type before it can be referenced.
- Two key issues in binding a type to an identifier:
  1. How is the type specified?
  2. When does the binding take place?
- How? – two kinds of declarations:
  1. Explicit declarations
  2. Implicit declarations
- When? - three kinds of type bindings:
  1. Static type binding
  2. Dynamic type binding
  3. Type inference

Variable Declarations

- An *explicit declaration* is a program statement used for declaring the types of variables.
  - Example: int x;
  - Advantage: safer, cheaper
  - Disadvantage: less flexible
- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
  - Example: in FORTRAN, variables beginning with I-N are assumed to be of type integer.

Variable Declarations

- Advantages: convenience
- Disadvantage: reliability (some typographical and programmer errors cannot be detected.
- Intermediate position: Names for specific types must begin with a given character.
  - Example: in Perl, variables of type scalar, array and hash structures begin with a $, @, or %, respectively.
  - Advantages:
    - Different namespaces for different type variables
      @apple vs. %apple vs. @apple
    - The type of a variable is know through its name.

Variable Declarations

- Implicit declarations leave more room for error
  - Example: In FORTRAN variables left undeclared will be implicitly declared as an integer.

Dynamic Type Binding

- The variable is bound to a type when it is assigned a value in an assignment statement.
  - JavaScript and PHP
  - Example: in JavaScript

```
list = { 2, 4, 6, 8 };
list = 17.3;
```

  - Dynamic binding of objects.
  - Advantage: flexibility (generic program units)

Dynamic Type Binding

- Disadvantages:
  - Compiler's type error detection is minimized.
  - If RHS is not compatible with LHS, the type of LHS is changed as opposed to generating an error.
    - This issue also appears in static type binding languages like C and C++
  - Must be implemented by a pure interpreter rather than a compiler
    - It is not possible to create machine code instructions whose operand types are not known at compile time.
  - High cost:
    - Type checking must be done at runtime
    - Every variable must know its current type
    - A variable might have varying sizes because different type values require different amounts of storage.
    - Must be interpreted.

Type Inference

- Rather than by assignment statement, types are determined from the context of the reference.
- Type inferencing is used in some programming languages including ML, Miranda, and Haskell.
- Example:
  - Legal:
    - fun circumf(r) = 3.14159 * r * r; *// infer r is real*
    - fun time10(x) = 10 * x; *// infer s is integer*

Type Inference

- Illegal:
  - fun square(x) = x * x
    // can't deduce anything (a default value could be assigned)
- Fixed
  - fun square(x : real) = x * x;
    // use explicit declaration
  - fun square(x) = (x : real) * x;
  - fun square(x) : real = x * (x : real);

Storage Bindings & Lifetime

- *Allocation* is the process of getting a cell from some pool of available cells.
- *Deallocation* is the process of putting a cell back into the pool.
- The *lifetime* of a variable is the time during which it is bound to a particular memory cell.
  - Begin: when the variable is bound to a specific cell
  - Ends: when the variable is unbound from that cell.

Variables: lifetime

- Categories of scalar variables by lifetimes:
  - Static
  - Stack-dynamic
  - Explicit heap-dynamic
  - Implicit hep-dynamic

Static Variables

- Bound to memory cells before execution and remains bound to the same memory cell throughout execution
  - Example: all FORTRAN 77 variables
  - Example: C static variables
- Advantages:
  - Efficiency (direct addressing)
  - No allocation/deallocation needed (which is run time overhead)
  - History-sensitive subprogram support (retain values between separate executions of the subprogram)

Static Variables

- Disadvantages:
  - If a language only has static variables then
    - Recursion cannot be supported (lack of flexibility).
    - Storage cannot be shared among variables (more storage required)

Stack-dynamic Variables

- Storage bindings are created for variables in the run time stack when their declaration statement are elaborated (or execution reaches the code to which declaration is attached), but types are statically bound.
  - If scalar, all attributes except address are statically bound
    - Example: local variables in C subprograms and Java methods

Stack-dynamic Variables

- Advantages:
  - Allows recursion
  - Conserves storage
- Disadvantages:
  - Run time overhead for allocation and deallocation.
  - Subprogram cannot be history sensitive
  - Inefficient references (indirect addressing)
  - Limited by stack size.

Explicit Heap-dynamic Variables

- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution.
  - Referenced only through pointers or references
    - Example: dynamic objects in C++ (via new/delete, malloc/free)
    - Example: all objects in Java (except primitives)
- Advantages:
  - Provides for dynamic storage management

Explicit Heap-dynamic Variables

- Disadvantages:
  - Unreliable (forgetting to delete)
  - Difficult of using pointer and reference variables correctly
  - Inefficient.
- Example:

```
int *intnode;          // create a pointer
…
intnode = new int      // create the heap-dynamic variable
…
delete intnode;        // deallocate the heap-dynamic variable
```


Implicit Heap-dynamic Variables

- Allocation and deallocation caused by assignment statements and types not determined until assignment.
  - Example: All arrays and strings in Perl and JavaScript
  - Example: all variables in APL
- Advantage: highest degree of flexibility
- Disadvantages:
  - Inefficient because all attributes are dynamic (a lot of overhead)
  - Loss of error detection

Summary Table

Variable Category	Storage binding time	Dynamic storage from	Type binding
Static	Before execution		Static
Stack-dynamic	When declaration is elaborated (run time)	Run-time stack	Static
Explicit heap-dynamic	By explicit instruction (run time)	Heap	Static
Implicit heap-dynamic	By assignment (run time)	Heap	Dynamic

Type Checking

- Generalizes the concept of operands and operators to include subprograms and assignments:
  - Subprogram is operator, parameters are operands.
  - Assignment is operator, LHS and RHS are operands.
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types.

Type Checking

- A *compatible type* is one that is either:
  - Legal for the operator, or
  - Allowed under language rules to be implicitly converted to a legal type by compiler-generated code.
  - This automatic conversion is called *coercion*
    - Example: adding an int to a float in Java is allowed, then int is coerced.
- A *type error* is the application of an operator to an operand of an inappropriate type.

Type Checking

- If all type bindings are
  - Static: nearly all type checking can be static
  - Dynamic: type checking must be dynamic
- Static type checking is less costly (it is better to catch errors at compile time) but it is also less flexible (fewer shortcuts and tricks).
- Static type checking is difficult when the language allows a cell to store a value of different types at different time, such as C unions, Fortran Equivalences or Ada variant records.

Strong Typing

- A programming language is *strongly typed* if
  - Type errors are always detected.
  - There is strict enforcement of type rules with no exceptions.
  - All types are known at compile time, i.e. are statically bound.
  - With variables that can store values of more than one type, incorrect type usage can be detected at run time.
- Advantages:
  - Strong typing catches more errors at compile time than weak typing, resulting in fewer run time exceptions.
  - Detects misuses of variables that result in type errors.

Which languages have strong typing?

- FORTRAN 77 is not because it does not check parameters and because of variable equivalence statements.
- Ada is almost strongly typed but UNCHECKED CONVERSIONS is a loophole.
- Haskell is strongly typed.
- Pascal is (almost) strongly typed, but variant records screw it up.
- C and C++ are sometimes described as strongly typed, but are perhaps better described as weakly typed because parameter type checking can be avoided and unions are not type checked.

Strong Typing vs. No Type

- Coercion rules strongly affect strong typing
  - They can weaken it considerably
  - Although Java has just half the assignments coercions of C++, its strong typing is still weak (less effective than Ada).
  - Languages such as Fortran, C and C++ have a great deal of coercion and are less reliable than those with little coercion, such as Ada, Java, and C#.
- In practice, languages fall on between strongly typed and untyped.

Type Compatibility

- There are 2 different types of compatibility methods for structure (nonscalar) variables:
  - Name type compatibility
  - Structure type compatibility
- *Name type compatibility* ("name equivalence") means that two variables have compatible types if
  - They are defined in the same declaration or
  - They are defined in declarations that uses the same type name.

Name Type Compatibility

- Easy to implement but highly restrictive:
  - Subranges of integer types are not compatible with integer types.
    - Example: count cannot be assigned to index

```
type IndexType is 1..100;
count: Integer;
index: Indextype;
```

  - Only two type names will be compared to determine compatibility.

Structure Type Compatibility

- *Type compatibility by structure* ("structural equivalence) means that two variables have compatible types if their types have identical structures.
- More flexible, but harder to implement.
  - The entire structures of two types must be compared.
  - May create types that are, but should not be compatible
    - Example: Celsius vs. Fahrenheit

```
type celsius = float;
Fahrenheit = float;
```


Type Compatibility

- Consider the problem of two structured types:
  - Are two record types compatible if they are structurally the same but use different field names?
  - Are two array types compatible if they are the same except that the subscripts are different (e.g. [1..10] and [0..9])?
  - Are two enumeration types compatible if their components are spelled differently?
  - With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float).

Scope

- The *scope* of a variable is the range of statements in a program over which it is visible.
  - A variable is visible if it ca be referenced in a statement.
- Typical cases:
  - Explicitly declared ⇒ local variables
  - Explicitly passed to a subprogram ⇒ parameters
  - The nonlocal variables of a program unit are those that are visible but not declared
  - Global variables ⇒ visible everywhere
- The scope rules of a language determine how references to names are associated with variables.
- The two major schemes are static scoping and dynamic scoping.

Static Scope

- Also known as "lexical scope"
- In static scoping, the scope of a variable can be determined at compile time, based on the text of a program.
- To connect a name reference to a variable, the compiler must find the declaration
  - Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name.
  - Enclosing static scopes to a specific scope are called its static ancestors; the nearest static ancestor is called a static parent.

Blocks

- A block is a section of code in which local variables are allocated/deallocated at the start/end of the block.
- Provides a method of creating static scopes inside program units.
- Introduced by ALGOL 60 and found in most PLs.

Blocks

- Variables can be hidden from a unit by having a "closer" variable with the same name.
- C++ allows access to "hidden" variables with the use of :: scope operator.
  - Example: if x is a global variable hidden in a subprogram by a local variable named x, the global could be reference as class_name::x
  - Ada: unit.x

Example of Blocks

C and C++

```
for (…) {
    int index;
    …
    }
```

Ada

```
Declare LCL:
    FLOAT;
    begin
    …
    end
```

Common Lisp

```
(let ((a 1)
    (b foo)
    (c))
  (setq a (* a a ))
  (bar a b ) )
```


Scope

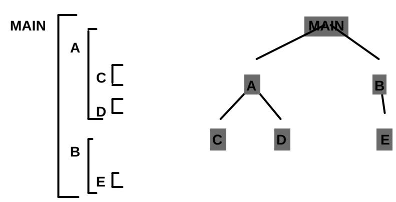
- Consider the example:

  Assume MAIN calls A and B

  A calls C and D

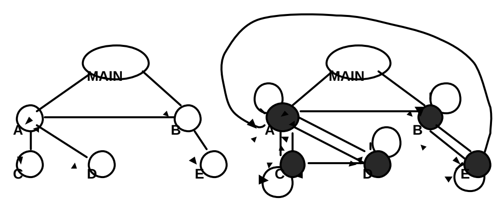
  B calls A and E

Static Scope Example



Static Scope Example

The desired call graph *The potential call graph*



Static Scope Evaluation

- Suppose now that E() needs to get access to a variable in D()
- One solution is to move E() inside the scope of D()
  - But then E can no longer access the scope of B
- Another solution is to move the variables defined in D to main
  - Suppose x was moved from D to main, and another x was declared in A, the latter will hide the former.
  - Also having variable declared very far from where they are used is not good for readability.
- Overall: static scope often encourages many global variables.

Dynamic Scope

- Based on calling sequences of program units, not their textual layout.
- Reference to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution at this point.
- Used in APL, Snobol and LISP
  - Note that these languages were all (initially) implemented as interpreters rather than compilers.
- Consensus is that PLs with dynamic scoping lead to programs which are difficult to read and maintain.

Scope Example

```
MAIN
  - declaration of x
    SUB1
      - declaration of x -
      ...
      call SUB2
      ...

    SUB2
      ...
      - reference to x -
      ...

  ...
  call SUB1
  ...
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

- *Static scoping*
  - reference to x is to MAIN's x
- *Dynamic scoping*
  - reference to x is to SUB1's x

Static vs. Dynamic Scoping

- Advantages of Static Scoping:
  - Readability
  - Locality of reasoning
  - Less run time overhead
- Disadvantages:
  - Some loss of flexibility
- Advantages of Dynamic Scoping
  - Some extra convenience
- Disadvantages
  - Loss of readability
  - Unpredictable behavior (minimal parameter passing)
  - More run-time overhead.

Scope vs. Lifetime

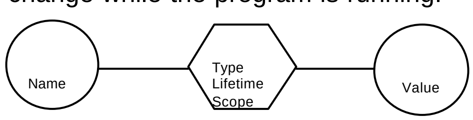
- While these two issues seem related, they can differ.
- In Pascal, the scope of a local variable and the lifetime of the local variable seem the same.
- In C/C++, a local variable in a function might be declared static but its lifetime extends over the entire execution of the program and therefore, even through it is inaccessible, it is still memory.

Referencing Environment

- The *referencing environment* of a statement is the collection of all names that are visible in the statement.
- In a static-scoped language, it is the local variables plus all of the variables in all the enclosing scopes.
- In a dynamic-scoped language, the referencing environment is the local variable plus all visible variables in all active subprograms.

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage.
- The value of a named constant can not change while the program is running.



Named Constants

- Advantages:
  - Readability
  - Maintenance
- The binding of values to named constants can be either static of dynamic
  - const int length = 5 * x;
  - final flow rate = 1.5*values;

Named Constants

- Languages
  - Pascal: literals only
  - Modula-2 and FORTRAN 90: constant-value expressions
  - Ada, C++, and Java: expressions of any kind
- Advantages
  - Increases readability without loss of effective.

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called *initialization*.
- Initialization is often done on the declaration statement
  - Example: In Java

```
int sum = 0;
```


Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors