5 Names, Bindings, Type Checking, and Scopes

- Introduces fundamental semantic issues of variables including
- Names and their attributes
- Attributes include
- Type
- Address
- Values
- Aliases
- Binding times
- Scope
- Lifetime

5.1 Introduction

- Imperative programming languages are abstractions of the underlying von Neumann computer architecture
- A variable can be characterized by a collection of properties, or attributes, the most important of which is type, a fundamental concept in programming languages.

5.2 Names

- *name* and *identifier* are equivalent

5.2.1 Name Forms

- Maximum length: 6 characters? 10 characters? 30 characters? 1000 characters?
- Connector characters: Underscore _ , Hyphen -, Space ,
- Case sensitive, as in C, C++, and Java: Is *HashIndex* is not equal to *hashindex*
- First character,
  - Perl, the first character must be one of $, @, or %, and these specify the variable's type
  - In Ruby, @ or @@, indicate that the variable is an instance or a class variable, respectively
  - In versions of Fortran prior to Fortran 90, names could have embedded spaces, which were ignored. For example, the following two names were equivalent

```
Sum Of Salaries
SumOfSalaries
```

5.2.2 Special Words

- **keyword**: special only in certain contexts. In FORTRAN, the declaration, **REAL** *APPLE* , **REAL** is special in that it defines the type of the variable declared. However, in the statement *REAL* = 3.4, *REAL* is the name of a variable.
- **reserved word**: A reserved word is always reserved and cannot have two meanings. For example the identifier **program** is always a reserve word in Pascal.
- predefined (standard): Standard types and functions are predefined in Pascal, Ada, and Modula-2. It is possible to redefine the standard type *integer*.

  type integer = 'a' .. 'z';

5.3 Variables

- Variables are names for memory locations.
- A variable can have up to six attributes
  i name
  ii address
  iii value
  iv type
  v lifetime
  vi scope

5.3.1 Name

- A name can identify a variable
- A name can identify a type
- A name can identify a class
- A name can identify a label
- A name can identify a constant
- A name can identify a subprogram
- Variables, types, classes, labels, constants, and subprograms are collectively called entities.

5.3.2 Address

- The address of a variable marks the first memory location assigned to that variable. A character occupies one byte. An integer occupies two or more bytes. A real number occupies four or more bytes.
- On the left side of an assignment statement a variable is represented by its address. The value stored in the variable is replaced by the expression on the right side of the assignment operator. **l-value** refers to the address of a variable. **r-value** refers to the actual value of the variable.

Alias

- An alias is another name for a variable. References in C++ are one mechanism for creating an alias.

```
int a=1;
int& b=a;
```

- Aliases are considered dangerous by our author. $\{i \in Integer \mid -2^{w-1} \le i \le 2^{w-1} - 1\}$

5.3.3 Type

- Type determines the set of values that can be stored in a variable.
  Examples:
  Integer $I = \{z \in Z \mid -2^{w-1} \le z \le 2^{w-1} - 1\}$, w is the number of bits in a word.
  Example: for a 16-bit word, $I = \{z \in Z \mid -2^{16-1} \le z \le 2^{16-1} - 1\}$

$$I = \{z \in Z \mid -2^{15} \le z \le 2^{15} - 1\}$$
$$I = \{z \in Z \mid -32{,}768 \le z \le 32{,}767\}$$

Example: for a 32-bit word

$$I = \{z \in Z | -2^{31} \leq z \leq 2^{31} - 1\}$$
$$I = \{z \in Z | -2{,}147{,}483{,}648 \leq z \leq 2{,}147{,}483{,}647\}$$

Boolean $B = \{0,1\}$
Real

$$R = \{-1^s \times 2^{c-b} \times 1.F, s \in \{0,1\}, 1 \leq c \leq 254, b = 127, F = \sum_{k=1}^{23} f_k \times 2^{-k}, f_k \in \{0,1\}$$

5.3.4 Value

- The value of a variable is the contents of the memory cell or cells associated with the variable. The capacity or size of the cell depends on the type and the number of elements defined by the type. Naturally, types having more elements require more memory to represent a single value.
- A variable's value is sometimes called its **r-value** because it is what is required when the variable is used on the ***r***ight side of an assignment statement.
- The value of a variable is a member of the set defined by its type.
- A variable may occupy one or more memory *cells*.
- A memory cell is the smallest group of bits (binary-digits) that can be addressed by the underlying computer.
- Usually a cell is a byte.
- A byte in times past has been defined by the size required to represent a character. Characters have been represented at various times by 5-bit Baudot, 6-bit fieldata, 7-bit ASCII, 8-bit EBCDIC, 8-bit ASCII.
- A cell on machines optimized for computation may be as large as 128 bits (Cray).

5.4 The Concept of Binding

- **binding** is an association between an *attribute* and an *entity*
  - An example of an attribute is the type assigned to a variable
  - An example of an entity is a variable
- **binding time** defines the time at which the binding is made
  - Example, in the C programming language, the type of a variable is bound at compilation time
- Consider the C++ declaration and statement.

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

- The type assigned to variable *count* is defined at compile-time. Some types are not defined at compile-time in C++. Pointers to virtual functions are not bound until execution time.

- The set of possible values for the type **int** is bound at compiler-design time, when the target architecture is known.
- The meaning of the operator + is bound at design time when + is defined to be the addition operator and further defined at compile time when the types of operands are known.
  - For example if both operands are integer, then the + operator is defined to be integer addition.
  - If one or both of the operands is a floating-point value then the + operator is defined to be floating-point addition.
  - If one or both of the operands is a string, then the + operator is defined to be concatenation.
  - In C++, the + operator can be further defined by operator overloading.
- The internal representation of the integer literal 5 is bound at compiler design time when the target computer is selected.
- The value of variable *count* is bound at execution time.

5.4.1 Binding of Attributes to Variables

- **static** bindings occur before execution-time
- **dynamic** bindings occur at execution-time

5.4.2 Type Bindings

- A variable must have a type before it can be assigned a value. A variable must have a type before it can be referenced in a program.

5.4.2.1 Static Type Binding

- An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type.
- An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements.
- Early languages – Fortran, BASIC – have implicit declarations
  - Fortran – An identifier than begins with one of the letters I, J, K, L, M, or N is implicitly declared to be an integer; otherwise the variable has type real.
- Languages that depend on implicit declarations by means of the first letter are not perfectly general in their rules for naming identifiers.
  - Perl, for example, requires that any name that begins with $ is a scalar, where a scalar is defined to be either a string or a numeric value.

    If a name begins with @, it is an array

    If it begins with a % it is a hash structure.

    It is worth noting that @apple and %apple are different, @apple is an array whereas %apple is a hash structure.

5.4.2.2 Dynamic Type Binding

- Variable type is not defined by declarations.
- Variable type is not defined by the spelling of its name.
- Variable type is defined, **at execution time**, when the variable is assigned.

Example: JavaScript, PHP

list=**[10.2,3.5];**

Regardless of the previous type of the variable named *list*, this assignment causes it to become a single-dimensioned array of length 2.

list=**47;**

Variable list becomes a scalar variable if the foregoing assignment is executed directly after the first example above.

Disadvantages:

1. Reliability: Programs are less reliable because errors related to type are diminished compared to a language where types are defined statically.
2. Cost: The cost of implementing dynamic attribute binding is considerable, particularly at execution time. Type-checking must be performed at execution time every time a statement is executed. When type-checking is performed during compilation, it is performed only once.

5.4.2.3 Type Inference

ML is a programming language that supports both functional and imperative programming. Consider how an ML function is defined.

fun *circumf*(*r*) **= 3.14159 *** *r* ***r*;**

Function *circumf* accepts a floating-point argument and produces a floating-point result. Because the constant **3.14159** is floating-point value the type of both the operator * and parameter r are inferred to be floating-point. Since, the expression **3.14159 *** *r* ***r** is floating-point the function *circumf* is inferred to have type floating-point.

fun *times10*(*x*) **= 10 *** *x*;

Because **10** is an integer constant its type is assigned to parameter *x* by inference and, subsequently the expression type **10 *** *x* is assigned to the function.

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

1. Parameter type, expression type, and function type are inferred from the operator ***.
2. Operator ***** accepts operands of type numeric.
3. By default, a numeric type is type **int**.

4. Thus, parameter *x*, expression *x* ***** *x*, and function *square* all have type integer.

*square***(2.75);**

causes an error because 2.75 is not an integer.

fun *square***(***x***) : real =** *x* ***** *x***;**

also causes an error because ML does not permit overloaded functions.

However,

fun *square***(***x***:real) =** *x* ***** *x***;**
fun *square***(***x***) =(***x***:real) *** *x***;**
fun *square***(***x***) =** *x* *** (***x***:real);**

are all valid function definitions in ML.

5.4.2.4 Variable Declarations

- **explicit declaration** – **var** a: integer; **int** a; A variable is assigned a type in a declaration before it is referenced.
- **implicit declaration**
  - Type is implicitly defined by the spelling of the variable name. Variable names beginning with letters I, J, K, L, and M were *implicitly* assigned type integer in FORTRAN programs.
  - Perl
    - $ - the variable is a scalar. A scalar variable is either a string or a numeric variable.
    - @ signifies the variable is an array.
    - % means the variable is a hash structures

5.4.2.5 Dynamic Type Binding

- Types are bound at execution time. Types are defined when a variable is assigned.

  list = [10.2, 3.5]

- Variable *list* is bound to the type **array**[0..1] **of** real by the foregoing assignment statement. Variable *list* could have any type prior to the assignment statement.
- Disadvantages of dynamic type binding
  - Error detection ability is diminished. Obviously the type compatibility rules for the assignment operator are discarded in a language that permits dynamic type binding. Compile time checking is deferred to run time. The designer must perform run time checking. No assistance is provided for the designer. The compiler has no rules to enforce.

- Dynamic binding is costly since all variable must include a type specifier at run time.

5.4.3 Storage Bindings and Lifetime

Term	Definition
Allocation	The process by which a variable is assigned memory.
Deallocation	The process by which memory assigned to a variable is returned to the pool of available memory.
Lifetime	The time that a variable is bound to specific memory.

There are four classes of storage binding for scalar variables including:

- static
- stack-dynamic
- explicit heap-dynamic
- implicit heap-dynamic

5.4.3.1 Static Variables

Static variables are bound to specific memory, (fixed memory locations), before program execution begins and remains bound to that memory until program execution terminates.

In C++

static int *i***;**
static *ostream o***;**

Advantages:

1. Convenience: It is convenient for a program or a subprogram to record state information.
2. Efficiency: All static variables can be addressed directly whereas other kinds of variables are addressed indirectly as an offset from a register or relative to a larger structure.

Disadvantage:

1. Reduced flexibility: A programming language, like FORTRAN, than has no other type of storage allocation does not support recursion.

5.4.3.2 Stack-Dynamic Variables

- **Elaboration**
  - The type of the variable is statically bound.
  - The address of the variable – the memory bound to the variable – is created when the variable is declared.

```
double celsius(double f){return 5*(f-32)/9;}
int main()
{   double f[]={32,68,72,100,110};
    for (int a=0;a<5;a++) cout << endl << celcius(f[a]);
    cout << endl;
    return 0;
}
```

5.4.3.3 Explicit Heap-Dynamic Variables

```
int *i;             //Create a pointer
i=new  int;         //Create an explicit heap-dynamic variable
*i=6;
delete i;           //Reclaim storage
```

5.4.3.4 Implicit Heap-Dynamic Variables

JavaScript

```
highs = [74,84,86,90,71];
```

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

5.5 Scope

Term	Definition
Scope	The *scope* of a variable is the range of statements in which the variable is visible.
Visible	A variable is *visible* in a statement if it can be referenced in that statement.
Nonlocal	The nonlocal variables of a program unit or block are those that are visible within the program unit or block but not declared there.

5.5.1 Static Scope

Term	Definition
Static scoping	Static scoping is so named because the scope of variable can be statically determined – that is, prior to execution. Static scoping permits a human reader and a compiler to determine the type of every variable in the program. Static scoping is also called *lexical scoping*.

There are two categories of static-scoped languages:

- those in which subprograms can be nested, which creates nested static scopes, for example Pascal and Ada
- and those which subprograms cannot be nested, for example C and C++

```
procedure Big is
    X: Integer;
    procedure Sub1 is
        X: Integer;
        begin – Sub1
        …
        end; -- Sub1
    procedure Sub2 is
        begin – Sub2
        … X …
        end; -- Sub2
begin – Big
…
end; -- Big
```

Static scope and subprograms

Example: Variable *X* in the program above is declared in two places: in procedure Big and in procedure Sub1. Variable *X* is referenced in only one place, in procedure *Sub2*. Since the scope of variable *X* declared in procedure *Sub1* is limited to that procedure, the variable *X* that is referenced in procedure *Sub2* is the variable *X* that is declared in procedure *Big*.

5.5.2 Blocks

- Blocks, introduced in Algol 60, allow a section of code to have its own local variables.
- Blocks introduce a new namespace.
- Variables declared in a block are typically stack dynamic.
- The term **block-structured language** is derived from the presence of blocks in a language.

```
if (list[i] < list[j]) {
    int temp;
    temp = list[i]; list[i]=list[j]; list[j]= temp;
}
```

Example block in C++

- In the example above, local variable *temp* is introduced for the sole purpose of exchanging two integer values. The array list and its indexes *i* and *j* are declared in enclosing blocks. This is an example of how blocks improve readability.

```
void sub()
{   int count=25;
    …
    for (int count=0;count<10;count++) {
        cout >> count >> endl;
    }
    cout >> count>> endl;
}
```

Example variable *count* hidden by scope

- In the example above, local variable *count*, declared in function *sub*, is hidden from local variable *count*, defined as a loop variable in the for-loop.
- The example above is illegal in Java and C#. The designers of Java and C# believed that the reuse of names in nested blocks was too error prone to be allowed.

5.5.3 Declaration Order

- Data declarations must appear at the beginning of a function except those in nested blocks in C89. Standard Pascal limited the location of data declarations to a var-clause that can only appear directly after a subprogram declaration.
- C99, C++, Java, and C# permit a data declaration anywhere that a statement can appear but still before the variable is referenced.
- In C99, C++, and Java the scope of a variable extends from its declaration to the end of the block in which it was declared.
- In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block, and as long as it is not in a nested block. Note that C# still requires that all variables be declared before they are used.

```
void fun()
{ …
    for (int count=0;count<10;count++) {
        …
    }
…
}
```

Scope of loop variables

- In early versions of C++, the scope of such a variable was from its definition to the end of the smallest enclosing block.
- In the standard version of C++, the scope is restricted to the for-construct.
- Java and C# also restrict the scope to the for-construct.

5.5.4 Global Scope

```
static int global; //A file-global variable
void fun1()
{   …
    global=1;
    …
}
int main()
{   …
    cout << endl << global << endl;
    …
    return 0;
}
```

File global variable

- static specifies the storage class static meaning only one storage location is allocated for variable global and the lifetime of the variable begins when function main is invoked and ends when main returns control to the operating system.
- Storage is allocated in this compilation unit.
- static also specifies that the name is local to the file and inhibits passing the name global to the linkage editor. Variable global cannot be referenced by any other compilation unit in program.

```
int global;                 //A global variable
void fun1()
{   …
    global=1;
    …
}
int main()
{   …
    cout << endl << global << endl;
    …
    return 0;
}
```

Global variable

- The absence of the static declaration permits the variable name to be passed on to the linkage editor where it is made available to other compilation units.
- Storage is allocated in this compilation unit.

```
static void fun1()
{   …
    global=1;
    …
}
int main()
{   …
    cout << endl << global << endl;
    …
    return 0;
}
```

A local function

- Function *fun1* is made local to this compilation unit only by the use of the **static** declaration.

```
extern int global;          //A global variable reference
void fun1()
{   …
    global=1;
    …
}
int main()
{   …
    cout << endl << global << endl;
    …
    return 0;
}
```

Referencing a global variable

- Integer variable global is declared in another compilation unit. Storage is allocated for variable global in the compilation unit where it was declared.
- A declaration
  int *global*;
  would make variable global declared in another compilation unit inaccessible.

5.5.5 Evaluation of Static Scoping

Static scoping provides a method of nonlocal access that works well in many situations. However, it is not without problems.

- Static scoping permits more access to both variables and subprograms than is necessary or desirable.
- The hierarchical nature of static scoping does not support the needs of more specifically managed scopes for software development and maintenance. An ever increasing visibility for more and more variables is not conducive to good program structure.
- Encapsulation, as enabled by object-oriented design has proven to be a more effective development tool.

5.5.6 Dynamic Scope

- Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Scope can be determined only at runtime.

```
procedure Big is
    X: Integer;
    procedure Sub1 is
        X: Integer;
        begin – Sub1
            …
        end; -- Sub1
    procedure Sub2 is
        begin – Sub2
            … X …
        end; -- Sub2
    begin – Big
        Sub1;
        …
    end; -- Big
```

Dynamic scope and subprograms

- Assume that Big calls Sub1 which, in turn, calls Sub2. Unlike static scoping rules, the reference to variable X in Sub2 resolves to the X declared in Sub1. A reference to a variable is resolved in nearest activation record having an allocation for a variable of the given name.

5.5.7 Evaluation of Dynamic Scoping

Dynamic scoping has several problems.

- Local variables are visible and can be accessed by all subprograms that are still active. There is no way to protect local variables from this accessibility.
- It impossible to ensure that nonlocal variables satisfy type rules. For example, a nonlocal integer variable of a certain name could be appear acceptably in an expression whereas a nonlocal character string variable would cause a run-time error because the expression was invalid for that type.
- Dynamic scope makes programs much more difficult to read because the calling sequence of sub programs must be known to determine the meaning of references to nonlocal variables. This task can be virtually impossible for a human reader.
- Accesses to nonlocal variables in dynamic-scoped languages take far longer than accesses to nonlocals when static scoping is used.

Dynamic scoping permits variables to be used as parameters because they are implicitly visible in the called subprogram.

5.6 Scope and Lifetime

```
int Count(void) { static int c=0; return c++;}
int main()
{   for (int a=0;a<5;a++) cout << endl << "Count=" << Count();
    cout << endl;
    return 0;
}
```

Scope and lifetime

- The scope of local variable *c* is limited to function *Count* but its lifetime is the same as the lifetime of function main.

```
void printheader ()
{
    …
}
void compute()
{   int sum;
    …
    printheader();
}
```

Dynamic scope and subprograms

- The scope of local variable *sum* is limited to function compute but its lifetime extends over the time during which function *printheader* executes.

5.7 Referencing Environments

- The referencing environment of a statement is the collection of all variables that are visible in the statement.

```
procedure Example is
    A,B:Integer;
    …
    procedure Sub1 is
        X,Y:Integer;
        begin – Sub1
            … <----------------1
        end; -- Sub1
    procedure Sub2 is
        X:Integer;
        …
        procedure Sub3 is
            X:Integer;
            begin – Sub3
                … <-------------2
            end; -- Sub3
        begin – Sub2
            … <------------------3
        end; -- Sub2
    begin -- Example
        … <----------------------4
    end; -- Example
```

Static referencing environment

```
void sub1()
{    int a,b;
     … <----------------------1
}
void sub2()
{ int b,c;
     … <----------------------2
     sub1();
}
int main()
{    int c,d;
     … <----------------------3
     sub2();
     return 0;
}
```

Point	Referencing Environment
1	*a* and *b* of *sub1*, *c* of *sub2*, *d* of *main*, (*c* of *main* and *b* of *sub2* are hidden)
2	*b* and *c* of *sub2*, *d* of *main*, (*c* of *main* is hidden)
3	*c* and *d* of *main*

5.8 Named Constants

- A named constant is a variable that is bound to a value only once.
- Named constants improve readability

```
void example()
{    int[] intList=new int[100];
     String[] strList=new String[100];
     double average,sum;
     …
     for (int a=0;a<100;a++){…}
     …
     for (int a=0;a<100;a++){…}
     average=sum/100;
     …
}
```

Without named constants

```
void example()
{   final int len=100;
    int[] intList=new int[len];
    String[] strList=new String[len];
    double average,sum;
    …
    for (int a=0;a<len;a++){…}
    …
    for (int a=0;a<len;a++){…}
    average=sum/len;
    …
}
```

With named constants

```
enum Season {spring,summer,autumn,winter};
```

With named constants

- Ada and C++ allow dynamic binding of values to named constants. For example, **const int** *result*=**2***width*+**1;**

- The value of the variable width must be visible when result is allocated and bound to its value.