VARIABLES
The VARIABLE is an abstraction of a memory cell. All variables have a sextuple of ATTRIBUTES (properties) namely: name, an address in memory, type, value, lifetime and scope

Names
NAMES refer to many objects, e.g. labels, functions, variables, etc. The equivalent term is IDENTIFIER. To understand a Programming Languages, you must understand its names.

- Length: how long are names
  - Early languages: only one letter
  - FORTRAN: 6 letters
  - C: unlimited, but only the first 63 chars are significant
- Format: the types of characters that can be used
  - Usually, names are a letter followed by an arbitrary combination of letters, numbers, and '_'
  - Early FORTRAN allowed spaces ' ' inside names
- Case sensitivity: are NUMBER, Number, and number different names?
  - C/C++/Java are case sensitive languages
  - FORTRAN, BASIC and Visual Basic are not case sensitive languages e.g. NUMBER, Number and number are the same name
- Treatment of "special words": KEYWORDS or RESERVED
  Definition: - A keyword is a word that is special only in certain contexts i.e. in Fortran: Real VarName (*Real is data type followed with a name, therefore Real is a keyword)*
  Definition: - A reserved word is a special word that cannot be used as a user-defined name
  Example: It is ok to declare and use keywords as variable names in FORTRAN but not in C:

Fortran	**C**
`Real Apple`	`int i;`
`Real Real`	`int int;`
`Apple = 3.4`	`i = 1;`
`Real = 4.5`	`int = 2;`

The gcc (C compiler) gives the error "error: parse error before '=' token"

Notice that not all variables have names (anonymous)

ADDRESS
A variable's address is a location in memory. The address is also called the 'l-value' (the address is relevant on the left-hand-side of an assignment)

- One name can refer to *different* addresses at different times.
- More than one name might refer to the *same* location in memory. When this occurs, we call the names *aliases*

In C and C++, can be created with the 'union' type, pointers and reference variables while in Python, it happens all the time.
Example of aliases in Python

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> p
[1, [2, 3], 4]
>>> q
[2, 3]
>>>
```

Generally, aliases are harmful to readability (program readers must remember all of them)

TYPE
All variables have a *type*. The type specifies the legal range of values for the variable. They also specifies the set of legal operations for the variable; in the case of floating point, type also determines the precision.

VALUE
A variable's *value* is the contents of the memory cell(s) associated with the variable.
The value is also called the 'r-value' (the value is relevant on the right-hand-side of an assignment)

The value could be a single byte (e.g. for a char) or a huge collection of bytes (e.g. for a big array)

BINDING
A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol. In programming languages it is important to know how and when *variables* are bound to their *attributes*.
The instant a binding occurs is referred as the *binding time*. The binding can occur at language design time, compile time, or run time.
Possible binding times:

- * operator is bound to the multiplication operation at language design time.
- A variable might be bound to a type at compile time, but to a memory location at run time
- Language implementation time: - bind floating point type to a representation
- Compile time: - bind a variable to a type in C or Java

Binding examples
For the following statement

```
int count;
…
count = count + 5;
```

The *type* of count is bound at compile time.
The set of *legal values* of count is bound at compiler design time.
The *meaning* of '+' is bound at compile time (when types are known)
The *value* of count is bound at execution time, when the assignment is executed.

Type Binding
There are two types of bindings:
i). STATIC binding: - The attribute in question is bound BEFORE run time and never changes during execution.
ii). DYNAMIC binding: - The attribute in question changes during execution or can change during execution of the program.

Most languages have STATIC type binding: The type of every variable can be determined at compile time
If static, the type may be specified by either an explicit or an implicit declaration. Most languages accomplish this by requiring *explicit variable declarations*
Definition: - Explicit declarations is a program statement used for declaring the types of variables i.e. give a name and its type
Definition: - Implicit declarations is a default mechanism for specifying types of variables (the first appearance of the variable in the program) i.e. use "default" conventions to assign a type to undeclared variables. E.g. In FORTRAN: Identifiers beginning with I, J, K, L, M, or N are assumed to be Integer. Others are assumed Real (Most experts think this is bad).

Declarations vs. Definitions
In C and other languages, a DECLARATION gives a type.
A DEFINITION gives a type AND causes storage allocation.
In C, only ONE definition is allowed. But there may be many (compatible) declarations.

Definitions:

```
int c;
int foo(int j) {
     return j * 10;
}
```

Declarations:

```
int c;
int foo(int);
```

Type Inference (ML, Miranda, and Haskell)
Rather than by assignment statement, types are determined from the context of the reference

E.g. ML and other languages have a funny way of doing static type binding. You can declare a function like shown below:

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

No types, but ML figures out from the * that x must be numeric. It uses the DEFAULT numeric type int for x. Then if figures out that square must be a function mapping an int to an int.

Later, calling square(2.75) is an error. To specify the real type, you need to use either of the following:

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

Dynamic Type Binding

If types are bound dynamically, no declarations are needed.
It is very convenient. You can write very flexible functions that don't care what type the arguments are, although they can cause problems:

```
int i;      mistyped:    int i;
i = c;                   i = x;
```

Suppose x is a float[]. The compiler will warn you.
Python is dynamically typed. If you say 'i = x', the variable i suddenly becomes a float array
Generally, programmers like dynamic type binding, but it comes at a cost.
– Type checking has to be done at RUN TIME.
– Every variable needs a type descriptor stored with it.
– (Usually), dynamically typed languages have to be interpreted, not compiled.
Dynamic typing is a tradeoff between execution speed and programmer convenience.

Storage Binding

When the memory cell(s) for a variable are taken from the free memory pool this is *allocation.*
Deallocation is the process of returning memory cells to the free memory pool.
The LIFETIME of a variable is the time during which it is bound to a specific memory location.
There are four types of storage binding:

i). Static: - bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
E.g. all FORTRAN 77 variables, C static variables

ii). Stack-dynamic: - bound when declaration is encountered
- In C, a local variable is bound to storage when the function is called.
- It is deallocated when the function exits
- Stack-dynamic variables are allocated on the runtime STACK.

iii). Explicit heap-dynamic: - Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution.
- Referenced only through pointers or references e.g. dynamic objects in C++ (via new and delete) and all objects in Java
- In C, use malloc() and free()
- The storage is called the HEAP because it is disorganized.

iv). Implicit heap-dynamic: - Allocation and deallocation caused by assignment statements e.g. all variables in APL; all strings and arrays in Perl and JavaScript. Also ALL Python variables are implicit heap-dynamic.

Type Checking

TYPE CHECKING is when we ensure that the operands of an expression are COMPATIBLE with the operators.
COMPATIBLE types are either the SAME type, or types that can be implicitly converted to each other.
Automatic conversion is called COERCION.
C automatically coerces ints to floats in floating-point expressions.
A TYPE ERROR is application of an operator to an illegal-type operand. E.g. treating an integer as a pointer.
If type bindings are static, type checking can be static (done at compile time)

If type bindings are dynamic, type checking has to occur at run time (DYNAMIC TYPE CHECKING).
Python does dynamic type checking.
In some statically typed languages, some type checks are still impossible.
– E.g. in C, UNION types cannot be checked.

Strong Typing
A programming language is strongly typed if type errors are always detected
This requires
- Statically bound types
- Detection of improper usage of memory locations storing values of different types.

Ada and Java are strongly typed. C/C++ are not: parameter type checking can be avoided; unions are not type checked and coercion decreases the value of strong typing.

A weakly typed language gives no guarantee that operations are performed on arguments that make sense for the operation. Many languages combine both strong and weak typing or both static and dynamic typing: Some types are checked before execution and other during execution, and some types are not checked at all. For example, C is a statically typed language (since no checks are performed during execution), but not all types are checked (e.g. unions).

Type compatibility
This is about if variables of two different types can be assigned. Two types:
i). Name Type Compatibility:
Name type compatibility means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
- Subranges of integer types are not compatible with integer types
- Formal parameters must be the same type as their corresponding actual parameters (Pascal)
- Very restrictive: prevents assignment of ints to longs.

ii). Structure Type Compatibility:
Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- 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?

Notice that most languages have a little bit of both.

Examples
Ada Type Compatibility
Ada is usually name type compatible

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

The statement "index = count" is illegal
Ada provides DERIVED types and SUBTYPES to be more flexible.

```
type celsius is new Float;
type farenheit is new Float;
```

The celsius and farenheit are DERIVED from Float, so they are all INCOMPATIBLE.

For increased type compatibility, Ada provides SUBTYPES:

```
subtype Small_type is Integer range 0..99;
```

The subtype Small_type IS compatible with the parent type, Integer.

Ada uses structure compatibility for arrays:

```
type Vector is array ( Integer range <> ) of Integer;
Vector_1 : Vector (1..10);
Vector_2 : Vector (11..20);
```

The range is unrestricted, and both arrays have 10 elements, so Vector_1 = Vector_2 is LEGAL. This shows that Ada is great language.

Scope

A variable's SCOPE is the set of statements in which it is VISIBLE. A variable is VISIBLE in a statement if it can be referred to in that statement.
LOCAL variables are variables defined in the current program unit or block while NON-LOCAL variables are visible in the current block but not defined there.
Scope can be STATIC or DYNAMIC. Most languages use static scope.

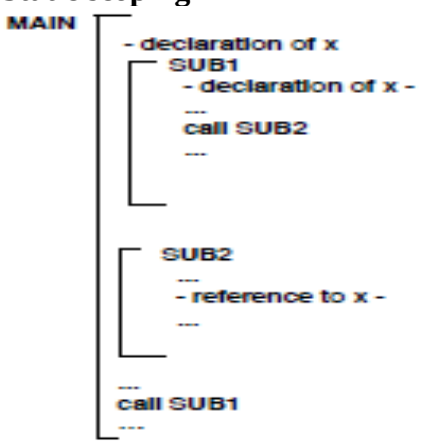
Static Scope

If the scope of all variables can be determined at compile type, then the programming language is STATICALLY SCOPED.
Some languages allow nested functions (Python), others don't (C).
E.g. Python:

```
>>> def square2( x ):
. . .     def dbl( x ):
. . .         return 2 * x
. . .     return dbl( x ) * dbl( x )
```

Nested functions can make scoping rules a bit tricky.

Static scoping



The declaration of x in SUB1 HIDES (or SHADOWS) the declaration of x in MAIN.
With static scoping, the reference to x in SUB2 is bound to the declaration in MAIN.

Static Scoping: Blocks

Many languages allow the creation of a new scope nested within executable code. This BLOCK can have its own local variables.

Ada uses DECLARE:

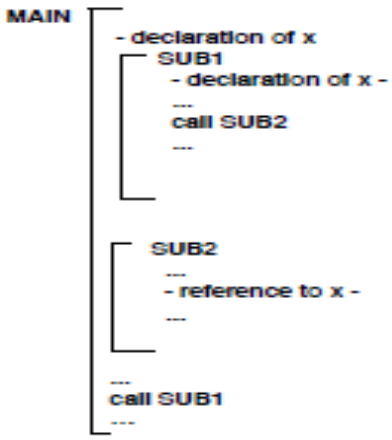
```
…
declare Temp : Integer;
begin
Temp := First;
First := Second;
Second := Temp;
end;
```

C uses { }:

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

Notice that restricting variable scope is good and global scope is bad

Dynamic scoping
In DYNAMIC scoping, the binding of a reference to variable depends on the current execution stack.
With dynamic scope, the reference to x in SUB2 would bind to the declaration of x INSIDE SUB1
Notice that most experts agree that dynamic scoping is usually bad. But it can be useful sometimes.

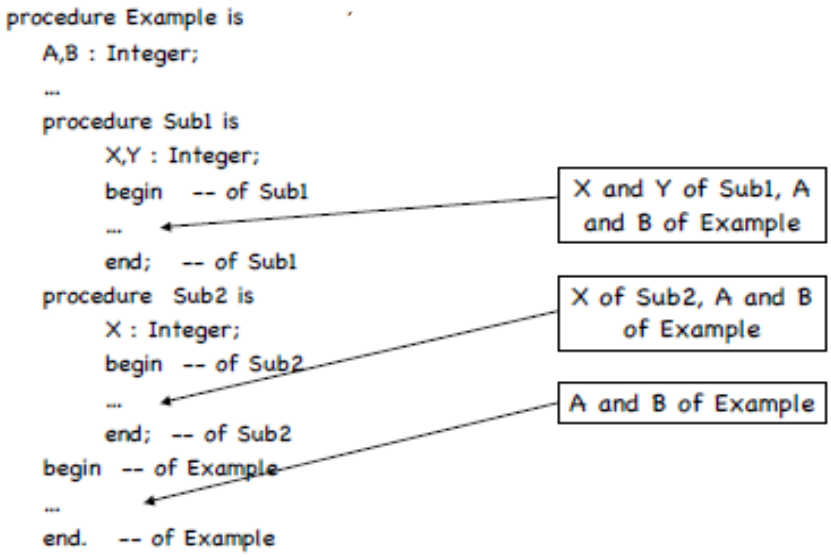


Scope vs. Lifetime
This is not the same thing. E.g. consider the code below:

```
void bar() {
      …
}
void foo() {
      int sum;
      …
      bar();
}
```

The SCOPE of sum is the function foo(). sum's LIFETIME is from the time foo() is called until it returns. sum is out of scope in bar(), but STILL ALIVE. Another example is the static locals in C.

Referencing Environments
The referencing environment for a statement is the set of all names visible to that statement.
In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes.
In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms. A subprogram is active if its execution has begun but has not yet terminated



Named Constants
A NAMED CONSTANT is a name whose value is assigned statically at compile time and can never be changed. Named constants are useful for array bounds definitions.
Java provides the "FINAL" reserved word.
C/C++/Java provides the 'const' reserved word, which provides a one-time, read-only assignment to a variable as shown below:

```
const int result = 2 * width + 1;
```

result is dynamically bound to its value, but cannot be changed once bound. You should always use named constants whenever possible in your programs

Variable Initialization
Sometimes you want a variable to have a fixed value when a program or function begins.
If a variable is statically bound to storage, the initialization can occur BEFORE runtime.
If storage binding is dynamic, the initialization is necessarily dynamic.
E.g.

```
/* at file scope */
int bad_global = 10;
void foo() {
     int good_local = 8;
     …
}
```

bad_global is initialized BEFORE the program begins execution while good_local is stack dynamically bound to storage, so it needs to be dynamically initialized on every call to foo()