Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Concepts of Programming Languages

Names and Variables

Dr. Amin Allam

[For more details, refer to “Concepts of Programming Languages” by *Robert Sebesta*]

1 Special words

Special words in programming languages (such as *int*, *struct*, *while*, *private*, *static*, and *else*, in C++) are used to make programs more readable by:

- Naming actions to be performed.
- Separate the syntactic parts of statements and programs.

Special words of a programming language are classified as one of the following:

- *Keywords*: that can be redefined to be used in other purposes, which reduces *readability*.
- *Reserved words*: that cannot be used for any other purposes, such as variable names.

In most languages, such as C++ and Java, *special words* are *reserved words*. In few languages, such as Fortran, *special words* are only keywords.

2 Names

A *name* is a string of characters used to identify some entity in a program. *Names* (or *identifiers*) are used to identify variables, subprograms, formal parameters, classes, and other program constructs. A *reserved word* cannot be used as a *name*.

Names do not have a limit on length in most languages to increase *readability*. In most languages such as C++, *names* consist of a letter followed by a string consisting of letters, digits, and underscore _ characters. Two common *naming styles* are `my_small_stack` and the *camel notation* `mySmallStack`. Such *naming styles* are only programming styles which are not forced by the programming language, although built-in packages and libraries are affected by the *naming style* choice of the language designers.

Some programming languages force naming rules to improve *readability*. For example, all variable names in PHP must begin with a dollar sign `$`. In Perl, a variable name starts with a special character (`$`, `@`, or `%`) which specifies its type.

In most languages, such as C++, *names* are *case sensitive*, meaning that uppercase and lowercase letters in names are considered distinct. For example, the following names are distinct in C++: `rose`, `ROSE`, `Rose`, and `rOse`. If *case sensitivity* is not used carefully by the programmer, it may reduce *readability*.

3 Variables

A program *variable* is an abstraction of a computer *memory cell*. By a *memory cell*, we mean an abstract *memory cell* which have enough *size* to hold the *variable*. The initial purpose for introducing *variables* was to use them as names for memory locations to improve *readability* by replacing absolute numeric memory addresses by their names. As programming languages developed, a *variable* gained more important meanings. Specifically, a *variable* is characterized by *six attributes*: • Name • Type • Address • Value • Scope • Lifetime •

• *Name:* A *name* identifies a *variable*. Naming a variable follows the rules specified previously. A *variable* may not have a *name* and be accessed by knowing its *address*.

• *Type:* The *type* of a *variable* determines its *size*, the range of values it can store, and the set of associated operations. For example, the *int* type in Java specifies a value range of -2^{31} to $2^{31}-1$ and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

• *Address:* The *address* of a *variable* is the machine memory address with which it is associated. The *address* of a variable is called its *l-value*, because the *address* is required when the *name* of a *variable* appears in the *left* side of an assignment statement. The term *l-value* also means a *variable* which have an *address* (to differentiate it between a *constant value* which have no *address*).

• *Value:* The *value* of a *variable* is the contents of the memory cell associated with the *variable*. A *variable*'s *value* is called *r-value* because it is required when the name of a *variable* appears in the *right* side of an assignment statement. The term *r-value* also means a *variable* or a *constant value*.

• *Scope:* The range of statements *where* the variable is *visible*. A *variable* is *visible* in a statement if it can be referenced in that statement. *Scope* is *locations* in source program (*spatial*).

• *Lifetime:* The time during which the variable physically exists in memory (time during which it is associated with or *bound* to its *storage*). *Lifetime* is *time durations* at execution (*temporal*).

4 Aliases

Variables having the same *type* and *address* are called *aliases*.
Aliases can be created with reference variables in C++:

```
int x=10;
int& y=x;    // both x and y now contain 10
x=5;         // both x and y now contain 5
y=9;         // both x and y now contain 9
```

Pointer variables holding the same address are called *aliases* as well, as in C++:

```
int* x=new int;
int* y=x;
*x=5;        // both *x and *y now contain 5
*y=9;        // both *x and *y now contain 9
delete x;
```

5 Named constants

A *named constant* is a *variable* that is assigned a constant value only once before *run time* and remains unchanged during every execution of the program, as in the following C++/C# statement:

```
const int hash_size=100003;
```

Named constants improve *readability* and also parametrize programs, so that it is possible to change the constant value at only one place inside the program instead of changing it from every place it is used.

A *named constant* differs from a *read-only variable* which is assigned a value in its constructor and does not change until its destruction, such as the following C# statement:

```
int x=get();  readonly int y=x;
```

which is equivalent to the following C++ statement:

```
int x=get();  const int y=x;
```

Note that C# uses two different *reserved words* to emphasize the two different meanings, while C++ uses the same *reserved word* for two different meanings which reduces *readability*.

6 Binding

A *binding* is an association between an entity and an *attribute*, such as between a *variable* and its *type* or *value*. The time at which *binding* takes place is called *binding time*. We are particularly interested in the following *binding times* for binding *attributes* to *variables*:

• *Compile time: Binding* occurs during compilation (for compiled languages only).

• *Load time: Binding* occurs when the program is loaded into memory and ready to run (immediately before *run time*).

• *Run time: Binding* occurs while the program is running.

A *binding* is *static* if it first occurs at *compile time* or *load time* (before *run time* begins) and remains unchanged throughout program execution. A *binding* is *dynamic* if it first occurs during *run time* or can change in the course of execution.

In the following two sections, we study when and how a variable *binds* to its:

• *Type:* A variable may *bind* to its *type* either *statically* or *dynamically*. A variable can *bind* to its *storage* only after it *binds* to its *type*, since its *size* is determined based on its *type*.

• *Storage:* A variable may *bind* to its *storage* (*memory cell*) *statically* or *dynamically*. *Binding* to *storage* directly affects the *address* and the *lifetime* of the variable.

• *Value:* A variable may *bind* to an *initial* value *statically* only if it is *statically bound* to its *storage*. Otherwise, it *binds dynamically* to values.

7 Type binding

Before a variable can be referenced in a program, it must *bind* to a *type*: *statically* or *dynamically*.

7.1 Static type binding

A variables may *bind statically* (before *run time*) to its *type* either *explicitly* or *implicitly*:

• *Explicit declaration:* A statement in a program that explicitly declares a variable and its *type*.

```
int x, y;    // x and y are integer variables
double f;    // f is a double precision floating point variable
```

• *Implicit declaration:* The *type* of a variable is implicitly deduced from the first appearance of the variable name using one of the following ways:

- The *syntactic form* of the variable. For example, In early versions of Fortran, a name beginning with certain letters such as `i`, `j`, and `k` is an integer. In Perl, a name beginning with a `$` is a scalar (string or numeric), with a `@` is an array, and with a `%` is a hash structure.

- The *context* of the variable. The following C# example deduces the variable *type* from its initial constant value:

```
var sum=0;  var total=0.0;  var name="Fred";   // int, float, string
```

7.2 Dynamic type binding

A variable *binds* to its *type* only when an assignment statement having the variable as its *LHS* is executed during *run time*. In that case, the variable *binds* to the *type* of the *RHS* expression of the assignment statement. Furthermore, a variable's *type* can change during program execution, as in the following JavaScript example:

```
list = [10.2, 3.5];     // Now, list is a single-dimensioned array of size 2
list = 47;              // Now, list is a scalar integer variable
```

Dynamic type binding usually occur in *scripting* / *interpreted* languages, since computers do not have instructions whose operand types are not known at *compile time*. Thus, *dynamic type binding* is much *costly* / *inefficient* than *static type binding*. A similar behaviour to *dynamic type binding* can be achieved to some limited extent by *compiled* languages, such as *polymorphism* in C++:

```
Shape* s; // Circle and Rectangle are inherited from Shape which has virtual Draw()
s=new Circle;    s->Draw();   // Calls Draw() of Circle class
s=new Rectangle; s->Draw();   // Calls Draw() of Rectangle class
```

Dynamic type binding improves *flexibility* / *writability* since it allows writing generic programs which can deal with different *types* of input variables. However, it reduces *reliability* since some errors may not be detected until such generic programs eventually encounters unexpected *types*, or expected *types* which were not meant by the programmer.

8 Storage binding

Before a variable can be referenced in a program, it must *bind* to its *storage* (*memory cell)*. The process of *binding* a variable to its *memory cell* is called *allocation*. The process of *unbinding* a variable from its *memory cell* is called *deallocation*. The in-between time duration is the *lifetime* of the variable. Variables can be categorized accordingly in in the following categories:

8.1 Static variables

Static variables are *bound* to *memory cells* at *load time* and remains *bound* to those *memory cells* until program execution terminates. All *Global variables* and *named constants* are *static variables*. Some subprograms or functions may require *static variables* to be history sensitive. The following C++ example demonstrates the two types:

```
int a=2;   // Global variable ( static )
void F()
{
   static int b=3;   // Static local variable
   int c=4;          // Stack−dynamic local variable (not static)
}
```

Static variables are very efficient, since they can be accessed by direct addressing (their addresses are known before *run time*) and no overhead is incorporated to allocate or deallocate them. However, *static variables* cannot support recursive subprograms.

Static class variables can be considered *static variables* with a slight difference which is they are not necessarily *bound* to *storage* at *load time*. They need to *bind* to *storage* at any time before the first class instantiation (before the first object from this class is created). A *static class variable* is created only once, while an *instance class variable* is created for each class object.

```
class Student
{
   int student_id;             // Instance class variable (not static)
   static int num_students;    // Static class variable declaration
}
int Student::num_students=0;   // Static class variable definition
```

8.2 Stack-dynamic variables

Stack-dynamic variables are *bound* to *storage* when their declaration statements are *elaborated*, but whose *types* are *statically bound*. *Elaboration* of a declaration statement refers to the *storage allocation* and *binding* process indicated by the declaration. *Elaboration* occurs when execution reaches the declaration statement or when the function including the declaration statement is called.

Stack-dynamic variables are *allocated* at *run time* from a region in memory called *the stack*, and *deallocated* when the called function terminates its execution. *Stack-dynamic variables* incur some *allocation* and *deallocation* overhead, but they support recursive programs.

8.3 Explicit heap-dynamic variables

Explicit heap-dynamic variables are nameless variables that are *allocated* by *explicit* run-time instructions called within the program. These variables *bind statically* to their *types*, but they *bind dynamically* to *storage* at the time they are created. Since they are nameless, they can be referenced only through pointer or reference variables. The pointer variables themselves do not need to be *heap-dynamic variables*; they are often *stack-dynamic variables*.

Explicit heap-dynamic variables are *allocated* and *deallocated* at *run time* from a region in memory called *the heap*. Contrary to *the stack*, *the heap* is a collection of storage cells which are highly disorganized due to the unpredictability of their usage. Therefore, *the stack* and *the heap* are separated as they need very different ways of management. Consider the following C++ function:

```
void F()
{
    Student* s;       // This pointer s is a stack-dynamic variable
    s=new Student; // Allocate a nameless heap-dynamic variable, Store its address
    delete s; // Deallocate the nameless heap-dynamic variable, but s still exists
    s=new Student[100]; // Allocate a nameless heap-dynamic array, Store address
    delete[] s; // Deallocate the nameless heap-dynamic array, but s still exists
}   // The pointer s is deallocated when the function call terminates execution
```

In C++, all *heap-dynamic variables* must be *explicitly deallocated* by run-time instructions. In Java, *heap-dynamic variables* are *implicitly deallocated* by the Java run-time *garbage collector* in unspecified time after the *garbage collector* makes sure that the *heap-dynamic variables* are no longer referenced by any existing pointer or reference. Consider the following Java function:

```
void F()
{
    Student s;        // This pointer s is a stack-dynamic variable
    s=new Student; // Allocate a nameless heap-dynamic variable, Store its address
    s=new Student[100]; // Allocate a nameless heap-dynamic array, Store address
}   // The pointer s is deallocated when the function call terminates execution
// The heap-dynamic variable and array will be deallocated by the garbage collector
```

Explicit heap-dynamic variables are often used to construct dynamic data structures, such as linked lists and trees that may need to grow and shrink during execution. They are also used to construct very large arrays, and arrays whose size is not known before *run time*.

8.4 Implicit heap-dynamic variables

Implicit heap-dynamic variables bind to *heap storage* only when they are assigned values. Actually, when they are assigned values, they *bind* to both *type* and *storage*. They exist only in interpreted languages because such variables *bind dynamically* to their *types*. The advantages and disadvantages of such variables are previously stated in the *dynamic type binding* section.

9 Value binding

Generally, a variable *binds* to a *value dynamically* at *run time*. There is exactly one case when a variable may *bind statically* to a *value*, which is when a *static* variable is *initialized*. Since a *static* variable *binds statically* to its *storage* at *load time*, it *binds* to its *initial value* also at *load time*. Consider the following C# example (note that *const* implies *static* in C#):

```
void F()
{
    static int x=10; // Static variable, binds to the value 10 only at load time
    const int y=5;   // Const static variable, binds to the value 5 only at load time
    int z=7; // Stack−dynamic variable, binds to 7 every time F() is called
    x=9;     // Not initialization, binds to 9 every time F() is called
}
```

Consider the following C++ example (note that all *global variables* are *static*):

```
int a=8;   // Global static variable , binds to the value 8 at load time
```

10 Scope

The *scope* of a variable is the range of statements where the variable is *visible*. A variable is *visible* in a statement if it can be referenced or assigned in that statement. The *scope rules* of a language determine how a particular occurrence of a name is associated with a variable. The *referencing environment* of a statement is the collection of all variables that are *visible* in that statement.

A variable is *local* in a program unit or *block* if it is declared there. The *nonlocal* variables of a program unit or *block* are those that are *visible* within the program unit or *block* but are not declared there. *Global* variables are a special category of *nonlocal* variables.

10.1 Static scoping

Most languages allow *nested blocks*. Some languages allow *nested subprograms*. Most languages apply *static scoping rules* to *nested blocks* and *nested subprograms*.

In *static scoping*, the *scope* of a variable can be statically determined prior to execution. This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.

When a variable name occurs inside a *block*, a declaration of a variable having that name is searched inside that *block*. If not found, a declaration of such variable is searched inside the *static parent block*; the *innermost block* enclosing the original *block*, and so on.

C and C++ allow a variable to be declared inside a *block* with the same name of another variable declared inside any *enclosing block*. In that case, the *outer block* variable is hidden from the *inner block* statements. Java and C# do not allow such naming to reduce programming errors.

The following example illustrates the *static scoping rules* by showing the *referencing environments* of some Ada program points:

```
procedure Example is
   A, B : Integer;
   procedure Sub1 is
      X, Y : Integer;
      begin -- of Sub1
      ... <----------------- [RE] X,Y of Sub1 - A,B of Example
      end; -- of Sub1
   procedure Sub2 is
      X : Integer;
      procedure Sub3 is
         X : Integer;
         begin -- of Sub3
         ... <-------------- [RE] X of Sub3 - A,B of Example
         end; -- of Sub3
      begin -- of Sub2
      ... <----------------- [RE] X of Sub2 - A,B of Example
      end; -- of Sub2
   begin -- of Example
   ... <-------------------- [RE] A,B of Example
   end. -- of Example
```

10.2 Dynamic scoping

Although the *scope* of variables in most languages is *static*, the *scope* of variables in some languages such as APL, SNOBOL4 is *dynamic*. Perl and Common Lisp also allow variables to be declared to have *dynamic scope*, although they have a default *static scoping* mechanism.

Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the *scope* can be determined only at *run time*.

When a variable name occurs inside a *subprogram*, a declaration of a variable having that name is searched inside that *subprogram*. If not found, a declaration of such variable is searched inside the *dynamic parent subprogram*; the *subprogram* that called the original *subprogram*, and so on.

The advantage of *dynamic scoping* is that the parameters passed from one subprogram to another are variables that are defined in the caller. None of these needs to be passed in a *dynamically scoped* language, because they are implicitly *visible* in the called subprogram.

Since references to *nonlocal* variables cannot be *statically* checked in *dynamically scoped languages* programs, they have less *readability*, *reliability*, and *efficiency* than equivalent programs in *statically scoped languages*.

The following example illustrates the *dynamic scoping rules* by showing the *referencing environments* of some program points:

```
void Sub1()
{
   int a, b;
   ... <------------ [RE] a,b of Sub1 - c of Sub2 - d of main
}
void Sub2()
{
   int b, c;
   ... <------------ [RE] b,c of Sub2 - d of main
   Sub1();
}
void main()
{
   int c, d;
   ... <------------ [RE] c,d of main
   Sub2();
}
```

10.3 Type checking

Type checking is the activity of ensuring that the operands of an operator, or parameters to a function, are of *compatible types*. A *compatible type* is one that either is *legal* for the operator / function or is allowed under language rules to be implicitly converted by the compiler or interpreter to a *legal type*. This automatic conversion is called a *coercion*. For example, if an `int` variable and a `float` variable are added in Java, the value of the `int` variable is *coerced* to `float` and a floating point addition is done.

The compiler or interpreter flags a *type error* if an operator is applied to an operand of an *incompatible type* such as `%` to `float` variables, or a parameter with *incompatible type* is passed to a function such as passing an `int*` variable to a function that needs an `int` variable.

Type checking is done in one of the following ways:

- *Statically:* Before *run time*, for most constructs in compiled languages.
- *Dynamically:* During *run time*, for most constructs in interpreted languages.
- *Not done at all:* For few constructs such as `union` in C and C++.

Static type checking is better than *dynamic type checking*, because the earlier correction is usually less costly and improves *reliability*. The penalty for *static type checking* is reduced *flexibility* / *writability* / *generality*.

10.4 Type equivalence

Two types are *equivalent* if they are considered *compatible* without *coercion*. There are two approaches to define *type equivalence*:

• *Name type equivalence:* Two variables have *equivalent types* if they are defined in declarations that use the same *type*. C++ and Java use *name type equivalence*, except for few cases such as *polymorphism*. `typedef` in C and C++ does not introduce a new *type*, so *types* defined with `typedef` fall into this category.

• *Structure type equivalence:* Two variables have *equivalent types* if their *types* have identical structures. For example, all *types* consisting of two `int` variables followed by one `float` variable are *structurally equivalent*. Fortran and COBOL use *structure type equivalence*.

10.5 Side effects

A *side effect* of a function (or operator) occurs when the function changes either one of its parameters or a global variable. *Side effects* occur only in *imperative languages* and may cause *semantic ambiguity* in some situations. For example, consider the following C++ program:

```
int Fun(int& a) {a+=10; return 10;}
```

```
int a=10;
int b=a+Fun(a);    // b may be 20 or 30
```

In C++, the *parse tree* of the expression `a+Fun(a)` determines how operands are associated to the operator, but does not determine the *evaluation order* of operands. That is, we do not know whether `a` or `Fun(a)` is evaluated first before doing the addition because *evaluation order* is not specified in the C++ standard. *Precedence* and *associativity* rules define *association order*, but there may exist several different compatible *evaluation orders*.

Similar problems occur for the expressions: `++a+a` and `Fun2(a,++a)` because the order of evaluating function parameters is not known as well. Such problems do not exist in Java since it guarantees that operands are evaluated from left to right. C++ does not have similar rule to allow the compiler to choose the order which results better optimization.

10.6 Short-circuit evaluation

A *short-circuit evaluation* of an expression is one in which the result is determined without evaluating all parts of the expression.

C++ and Java apply *short-circuit evaluation* for logical `&&` and `||` only. Expressions separated by `&&` are evaluated from left to right. When one of them is `false`, evaluation stops and the whole expression returns `false`. Expressions separated by `||` are evaluated from left to right. When one of them is `true`, evaluation stops and the whole expression returns `true`.

If *short-circuit evaluation* is supported for `&&` in a language, the expression `(i<n && a[i]!=v)` will not cause any problem assuming that array `a` contains `n` elements. Otherwise, it will cause an indexing error.