Review of Concepts from Procedural Programming Languages

Declarations/Scope/Lifetime/Binding

Static/Dynamic

- Identifiers are declared, either explicitly or implicitly (from context of first use).
- Declarations bind type and kind information to an identifier. Kind specifies the grouping of an identifier (variable, label, function, type name, etc.)
- Each identifier has a scope (or range) in a program—that part of the program in which the identifier is visible (i.e., may be used).

Data objects have a lifetime—the span of time, during program execution, during which the object exists and may be used.

Lifetimes of data objects are often tied to the scope of the identifier that denotes them. The objects are created when its identifier's scope is entered, and they may be deleted when the identifier's scope is exited. For example, memory for local variables within a function is usually allocated when the function is called (activated) and released when the call terminates. In Java, a method may be loaded into memory when the object it is a member of is first accessed.

Properties of an identifier (and the object it represents) may be set at

Compile-time

These are static properties as they do not change during execution. Examples include the type of a variable, the value of a constant, the initial value of a variable, or the body of a function.

Run-time

These are dynamic properties. Examples include the value of a variable, the lifetime of a heap object, the value of a function's parameter, the number of times a while loop iterates, etc.

Example:

In Fortran

- The scope of an identifier is the whole program or subprogram.
- Each identifier may be declared only once.
- Variable declarations may be implicit. (Using an identifier implicitly declares it as a variable.)
- The lifetime of data objects is the whole program.
**Block Structured Languages**

- Include Algol 60, Pascal, C and Java.
- Identifiers may have a non-global scope. Declarations may be *local* to a class, subprogram or block.
- Scopes may *nest*, with declarations propagating to inner (contained) scopes.
- The lexically *nearest* declaration of an identifier is bound to uses of that identifier.

**Binding of an identifier to its corresponding declaration is usually static (also called lexical), though dynamic binding is also possible.**

Static binding is done prior to execution—at compile-time.

Example (drawn from C):

```c
int x, z;
void A() {
    float x, y;
    print(x, y, z);
}

void B() {
    print(x, y, z);
}
```

**Block Structure Concepts**

- Nested Visibility
  - No access to identifiers outside their scope.
- Nearest Declaration Applies
  - Static name scoping.
- Automatic Allocation and Deallocation of Locals
  - Lifetime of data objects is bound to the scope of the identifiers that denote them.

**Variations in these rules of name scoping are possible.**

For example, in Java, the lifetime of all class objects is from the time of their creation (via `new`) to the last visible reference to them.

Thus

```java
... Object O;...
```

creates an *object reference* but does not allocate any memory space for `O`.

You need

```java
... Object O = new Object(); ...
```

to actually create memory space for `O`. 
Dynamic Scoping

An alternative to static scoping is dynamic scoping, which was used in early Lisp dialects (but not in Scheme, which is statically scoped).

Under dynamic scoping, identifiers are bound to the dynamically closest declaration of the identifier. Thus if an identifier is not locally declared, the call chain (sequence of callers) is examined to find a matching declaration.

Example:

```java
int x;
void print() {
    write(x);
}
main () {
    bool x;
    print();
}
```

Under static scoping the \( x \) written in \( \text{print} \) is the lexically closest declaration of \( x \), which is as an \( \text{int} \).

Under dynamic scoping, since \( \text{print} \) has no local declaration of \( x \), \( \text{print} \)'s caller is examined. Since \( \text{main} \) calls \( \text{print} \), and it has a declaration of \( x \) as a bool, that declaration is used.

Dynamic scoping makes type checking and variable access harder and more costly than static scoping. (Why?)

However, dynamic scoping does allow a notion of an "extended scope" in which declarations extend to subprograms called within that scope.

Though dynamic scoping may seem a bit bizarre, it is closely related to virtual functions used in C++ and Java.

Virtual Functions

A function declared in a class, \( C \), may be redeclared in a class derived from \( C \). Moreover, for uniformity of redeclaration, it is important that all calls, including those in methods within \( C \), use the new declaration.

Example:

```java
class C {
    void DoIt(){PrintIt();}
    void PrintIt()
        {println("C rules!");}
}
class D extends C {
    void PrintIt()
        {println("D rules!");}
    void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```
Scope vs. Lifetime

It is usually required that the lifetime of a run-time object at least cover the scope of the identifier. That is, whenever you can access an identifier, the run-time object it denotes better exist.

But, it is possible to have a run-time object's lifetime exceed the scope of its identifier. An example of this is static or own variables.

In C:

```c
void p() {
    static int i = 0;
    print(i++);
}
```

Each call to p prints a different value of i (0, 1, ...) Variable i retains its value across calls.

Some languages allow an explicit binding of an identifier for a fixed scope:

```plaintext
Let

<table>
<thead>
<tr>
<th>id = val</th>
<th>type id = val;</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>statements</td>
</tr>
<tr>
<td>statements</td>
<td>}</td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
</tbody>
</table>
```

A declaration may appear wherever a statement or expression is allowed. Limited scopes enhance readability.

Structs vs. Blocks

Many programming languages, including C, C++, C#, Pascal and Ada, have a notion of grouping data together into structs or records.

For example:

```c
struct complex { float re, im; }
```

There is also the notion of grouping statements and declarations into blocks:

```c
{ float re, im;
  re = 0.0; im = 1.0;
}
```

Blocks and structs look similar, but there are significant differences:

Structs are data,

- As originally designed, structs contain only data (no functions or methods).
- Structs can be dynamically created, in any number, and included in other data structures (e.g., in an array of structs).
- All fields in a struct are visible outside the struct.
Blocks are *code*,

- They can contain both code and data.
- Blocks *can’t* be dynamically created during execution; they are “built into” a program.
- Locals in a block *aren’t* visible outside the block.

By adding functions and initialization code to structs, we get *classes*—a nice blend of structs and blocks.

For example:

```cpp
class complex{
    float re, im;
    complex (float v1, float v2){
        re = v1; im = v2; }
}
```

**Classes**

- Class objects can be created as needed, in any number, and included in other data structure.
- They include both data (fields) and functions (methods).
- They include mechanisms to initialize themselves (constructors) and to finalize themselves (destructors).
- They allow controlled access to members (private and public declarations).