Programming Languages — Parsing and Data Types

Array Indexing

In C/C++ an *array* is a *pointer* to the first element of a contiguous collection of homogeneous elements.

```
int A[10]; // A is address of A[0] which is an int; sizeof(A) == 10 * sizeof(int)
char ticTacToe[3][3]; // ticTacToe is address of ticTacToe[0] which is a char[3];
                      // sizeof(ticTacToe) == 3 * 3 * sizeof(char)
float Q[2][4]; // Q is pointer at a float[4]; sizeof(Q) == 2 * 4 * sizeof(float)
```

Addressing is base (the array variable) plus the right-most index times the size of an element plus the second to right-most index times the size of an element times the size of the right-most dimension and so on. This assumes *row-major storage* of multidimensional arrays.

```
&A[5] == A + 5 * sizeof(int);
&ticTacToe[2][1] == ticTacToe + 2 * (3 * sizeof(char)) + 1 * sizeof(char);

&Q[1][0] == Q + 1 * (4 * sizeof(float)) + 0 * sizeof(float)
```

Question: How would the calculations change for *column-major* storage? For any that do not change, explain why (there are two differet reasons).
Arrays of pointers to arrays is an alternative way to store multiple dimension arrays: the last (right-most) index indexes an array of elements. The next dimension to the left indexes a (one-dimensional) array of pointers at arrays of elements and so on.
Question: What are the benefits of this layout? What are the costs?
Question: Which method does Java use for simple arrays? What about C/C++?

String

A *string* in most PL is an array of characters. Note that character is different from `char`: `char` is typically one byte is size while a modern character is whatever type can hold characters for human languages that the program can process (Unicode).
A *C-style string* is an array of `char`. It is variable length in that the `'\0'` character serves as an end of string *sentinel*. The *maximum length* of a string is constrained by the memory set aside for it but it can be marked as shorter with a `NUL` character (that is the ASCII name for the zero character).

Pointers

A *pointer* is a variable that holds the address of an element of some specified type. In C-style languages, a pointer can be used with square brackets and integer addition/subtraction to calculate the address of contiguously stored elements of the same type (actuall same size).
Something to think about: How does `delete[]` know how big the array was when `new` was called?

Memory Leaks

Dynamically allocated memory (`new`) in C/C++ and Java must be reclaimed when it is no longer in use. The two have different approaches:
Java uses *garbage collection* in that when a heap-allocated object is no longer **live** (reachable through the transitive closure of live references), it will be marked as garbage and, at some point, reclaimed.
C/C++ count on the programmer to `delete` the memory, giving it back to the run-time library's memory manager. Failure to do so creates a memory leak.

Question: Compare and contrast the two approaches.
Question: Write short C++ code that leaks. Then rewrite it to fix the leak. What is the cost of Java programmers not needing to care?

Scope, Lifetime

The *scope* of a variable is where the name of the variable refers to a particular definition of the variable. Reusing a variable name can create *scope-holes* where a given name could refer to either of two declarations; in lexcial scoping, it refers to the innermost declaration even though outer declarations live on.
Scope-holes are different from using the same variable name in two, non-overlapping scopes.
Variables in calling contexts remain *live*. So global variables typically are live until the end of program execution.
Dynamic scope resolution uses the active calling stack (rather than text of the source code) to resolve non-local variable references.

Lexical Analysis

Language processing is going from a *stream of characters* to a *stream of instructions*; there must be some intermediate form that represents the **meaning** of the program so that the instructions can preserve that meaning.
A stream of characters is typically transformed into a *stream of tokens* during the lexical analysis phase of processing. A **token** is a symbol in the alphabet of symbols that make up a program in a given language, *e.g.* integer literal, keyword, variable, +, ".
To avoid overwhelmingly large alphabets, tokens are usually grouped by *type* and a token carries with it the actual string value it is associated with, its *lexeme*. So `v` is a *variable* token with the lexeme `v`.
Consider this set of token definitions

```
<add_op>    ::= '+'|'-'
<mult_op>   ::= '*' | '/' | '%'
<assign_op> ::= ':='
<paren>     ::= '(' | ')'
<ident>     ::= Any valid Java identifier
<int_lit>   ::= [0-9]+
<EOF>       ::= End of file marker
```

What is the tokenization of the string

```
readable := 10 * comments + whitespace + goodNames <EOF>
```

Token Type	Lexeme
<ident>	readable
<assign_op>	:=
<int_lit>	10
<mult_op>	*
<ident>	comments
<add_op>	+
<ident>	whitespace
<add_op>	+
<ident>	goodNames
<EOF>	<EOF>