Tools for Our Time

by Chuck Allison

ISTOCKPHOTO

Sometimes, when I can come up for air, I ponder how software development has changed over the years. I probably started before many of you, back in the day of paper tape and punch cards, even before the innovative eight-inch floppy disk. When I was doing classified work, a courier would take my card deck to a service bureau's mainframe an hour's drive away. I had to wait until the next morning to find out if I had any syntax errors. I quickly learned that desk-checking one's code was not really optional. Getting a program to compile was a major milestone—a highly praised achievement that called for celebration.

Recently, I perused Monster.com, finding that today's developers require skills way beyond knowing how to compile code:

> **Networking/Web:** SOAP, HTML, DHTML, CSS, XML, JAXP, JAXB, DOM, SAX, ASP/.NET, Sockets, RPC, CORBA, RMI, DNS, LDAP, TCP/IP, NDS, Active Directory, VOIP, Ajax, Struts, WebLogic, WebSphere, Servlets, Hibernate, Flash
> **Database:** SQL, Oracle, PL/SQL, ProC, SQL Server, T-SQL, MySQL, PostgreSQL, DB2, ADO.NET, ODBC, JDBC, MS Access
> **Platforms:** Windows, Linux, AS/400, J2EE, .NET, Solaris, HP/UX, J2ME, Windows CE
> **Languages/Scripting:** C, C++, C#, Java, PHP, Perl, Python, VB, VB.NET, COBOL, JavaScript
> **Design:** UML, Rational Rose, Design Patterns

That's a lot to just have heard of, let alone be familiar with. Software development isn't what it used to be.

Programming languages are not what they used to be, either. Consider how they've evolved. Here's a much abbreviated timeline, showing the major innovations selected languages have introduced:

- 1954 FORTRAN (first high-level language): arrays, loops, subprograms
- 1958 Algol: block scope, structured programming, recursion
- 1959 Lisp: interpreted language, functional paradigm, garbage collection
- 1965 PL/I: exception handling, concurrency, pointers,
- 1967 Simula-67: objects and classes, inheritance, co-routines
- 1968 Algol 68: user-defined structures, references, dynamic arrays
- 1980 Smalltalk: graphical user interfaces, object-oriented programming
- 1998 C++: type-safe generic (and multi-paradigm) programming

These are the shoulders on which today's languages stand.

Languages for Our Time

As of September 2007, the coding standards company TIOBE (see the StickyNotes for a link) rated the top twenty programming languages in order of popularity as follows: Java, C, Visual BASIC, PHP, C++, Perl, C#, Python, JavaScript, Ruby, PL/SQL, SAS, D, Delphi, ABAP, Lisp, COBOL, Lua, Ada, and FORTRAN.

Three of these—PL/SQL, ABAP, and SAS—are special-purpose languages. Visual BASIC and Delphi, while based on general-purpose languages, are specifically targeted for GUI development. What kind of languages are the other fifteen? Table 1 considers selected attributes of these languages.

One language in particular crosses some interesting attribute boundaries. It compiles to native executables and is statically typed, yet it is garbage-collected—not your everyday combination.

The D Programming Language

The D language is mostly the handiwork of Walter Bright, developer of Zortech C++, the first native-code C++ compiler on DOS in the 1980s. Like many of us, he wondered what C++ would be like if compatibility with C were not an issue. Unlike the rest of us, he did something about it. The result is an efficient, easy-to-use language with built-in support for many of

Attribute	Languages
Compiled (native code)	C, C++, D, COBOL, Ada, FORTRAN
Interpreted	Java, PHP, Perl, C#, Python, JavaScript, Ruby, Lisp, Lua
Scripting	Perl, Python, JavaScript, Ruby, Lua
Dynamically Typed	PHP, Perl, Python, JavaScript, Ruby, Lisp, Lua
Statically Typed	Java, C, C++, C#, D, COBOL, Ada, FORTRAN
Object Oriented	All but C (requires modern versions of Ada and FORTRAN)
Systems Programming	C, C++, D, Ada
Functional Programming	Lisp (100%), Perl, Python, Ruby, C++, D, Lua
Garbage Collection	Java, PHP, Perl, C#, Python, JavaScript, D, Lisp, Lua

Table 1

the features we take for granted in modern programming languages, and then some.

The obligatory "Hello, world" program is shown in listing 1. The import statement works just like Java's, except that modules have a one-to-one correspondence to files, as in Python.

```
// hello.d
import std.stdio;

void main(string[] args) {
    writefln("Hello, modern world");
}
```

Listing 1

Free-standing functions behave as in C and C++, except that `main` doesn't have to declare `int` as a return type (although it can). The `writef` function is like C's `printf` with formatting extensions. The `writefln` function appends a newline.

For Java programmers, D has pretty much everything except dynamic class loading, which doesn't apply since it is a compiled language. Other Java features are there, including interfaces and inner classes. In contrast to Java, D's generics (templates) are type-safe. Listing 2 is a simple stack template.

```
// stack.d: A stack template
struct Stack(T) {
   private T[] data;
   public void push(T t) {
      data ~= t;
   }
   public T pop() {
      T t = data[$-1]; // $ == data.length
      data.length = data.length-1;
      return t;
   }
   public T top() {
      return data[$-1];
   }
   public int length() { return data.length; }
}
```

Listing 2

Since it is declared with the empty `[]` syntax, `data` is a dynamic array. The `length` property for dynamic arrays can be updated, which resizes the array. Listing 3 is a sample driver for `Stack`.

You use the exclamation point to instantiate a template. Note how the `length` member function automatically acts as a *property* in the assert statements (i.e., no function-call syntax is required). Note also that instead of throwing an underflow exception in `pop` and `top`, I just let automatic array bounds-checking do the job. You could also use D's built-in support for contract programming to validate preconditions, if you prefer, as the implementation of `top` in listing 4 illustrates.

```
import std.stdio;
void main() {
   Stack!(int) s;          // Instantiate template
   s.push(1);
   assert(s.length == 1);  // Property access
   writefln("top: %d", s.top());
   writefln("popping %d", s.pop());
   assert(s.length == 0);
   try {
      s.top();             // Intentional underflow
   }
   catch {
      writefln("caught exception");
   }
}

/* Output:
top: 1
popping 1
caught exception
*/
```

Listing 3

```
public T top()
in {
   assert(data.length > 0);
}
body {
   return data[$-1];
}
```

Listing 4

Preconditions go in an `in` clause while postconditions go in an `out` clause. Violated assertions raise an `AssertError` exception. D can also enforce class invariants.

What does D have that Java doesn't? Here's a partial list: stand-alone functions, nested functions, delegates, inline assembler, native-code compilation, deterministic destruction of objects for automatic resource management ("scope guards"), compile-time programming (a.k.a. metaprogramming), type inference (see the program in listing 5), and built-in support for automated unit testing and contract programming.

In listing 5, the sample D program counts the number of occurrences of words in text files and lists the results alphabetically. Running this program on the previous sentence yields the output in Listing 6.

The call to `std.string.split` takes the string returned from `std.stdio.read` (which is actually a byte array, hence the cast) and splits on whitespace. The `auto` keyword declares `words` to be the same type as the output from `split`, which here is a dynamic array of strings. `Counts` is an associative array (a.k.a. a map or hash) with string keys and integer values. The `foreach` keyword is D's generic iterator construct. Associative arrays have a `keys` property that returns the keys as an array, and arrays have a `sort` property that sorts the array in place. The expression `[1..$]` is an array *slice*, in this case comprising the second slot through the end of the array (denoted by `$`). Slices are composable, mutable views into an array—no copies are made.

D's power and readability have appealed to a large audience in a short time, hence the favorable TIOBE ranking. Already in version 2.0, D held its first Developers' Conference in August 2007 in Seattle, sponsored by Amazon. If you are looking for a high-level, type-safe language with the efficiency of C++ and the convenience of Java or C# and then some, D might be the language for you. You can find information regarding all things D at *digitalmars.com*. {end}

```
// wc.d: Counts the number of occurrences of each word in a text file
import std.stdio;
import std.string;
import std.file;

// This function does all the work (Reads words into a list;
// computes counts; displays results)
void wc(string filename) {
    auto words = split(cast(string) read(filename));
    int[string] counts;
    foreach (word; words)
        ++counts[word];
    foreach (w; counts.keys.sort)
        writefln("%s: %d", w, counts[w]);
}

// A simple driver: process all file arguments
void main(string[] args) {
    foreach(f; args[1..$]) {    // Start at second arg ([1])
        writefln("\n%s:", f);
        wc(f);
    }
}
```

Listing 5

```
text.dat:
5,: 1
D: 1
In: 1
alphabetically.: 1
and: 1
counts: 1
in: 1
listing: 1
lists: 1
number: 1
occurrences: 1
of: 2
program: 1
results: 1
sample: 1
text: 1
the: 3
words: 1
```

Listing 6

Chuck Allison developed software for twenty years before becoming a professor of computer science at Utah Valley State College. He was senior editor of the C/C++ Users Journal *and is founding editor of* The C++ Source. *He is also the author of two C++ books and gives onsite training in C++, Python, and Design Patterns.*

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- TIOBE

Are you marketable? In how many of the top-twenty languages are you fluent? Which are your favorites?

Follow the link on the StickyMinds.com homepage to join the conversation.