Emacs and Programming in GNU/Linux

Ed Cashin

ALF 2.0, Sept. 18, 1999

Abstract

You are reading about GNU Emacs, the GNU incarnation of the advanced, self-documenting, customizable, extensible real-time display editor Emacs. (The 'G' in 'GNU' is not silent.)

... So says the beginning of GNU Emacs' documentation. Emacs can tie things together, making everything from the filesystem to the compiler to the network readily accessible and thereby more productive. Emacs has its origins as a text editor, a simple program for editing text, but the fact that it so customizable has led to emacs' evolution into a kind of portal to the world of UNIX resources.

In this article we focus firsst on emacs itself and from there expand to touch on some of the more interesting tools and environments for the programmer on a GNU/Linux system.

I wrote this article to get experience with a couple of LaTeX features and also to give people something to flip through at ALF—to give them a chance to flip through and see if there's anything that catches their eye.

Contents

1 Emacs Feature Highlights

1.1 annoyance

Emacs provides a rich set of annoyances to the new user. Emacs has been created by its own users, and they are often reletively hard-core computer folks. They will write nice documentation for you to read, but new users will not know where the documentation is. I lost patience with emacs several times before finding out how to use its online documentation:

C-h

Holding down the control key and hitting 'h' is the way to ask for help. Emacs' help system is very elaborate, so after *C-h* you pick what kind of help you want by hitting another key. All this is explained for you at the bottom of the screen. Just remember *C-h* and you'll be OK. Reading the docs is certainly worth it.

1.2 extensibility

Only a small part of emacs was written in C (mostly parts that are critical for performance). The bulk of the code is written is a special variant of the programming language LISP. It is called "emacs lisp," or simply "elisp."

LISP is one of the earlier programming languages, developed by artificial intelligence academics. It is a high-level programming language where the programmer does not directly handle things like memory allocation. LISP is short for "*LIS*t *P*rocessing", and programs are formed by combining lots and lots of lists. Since these lists are delimited with parentheses, LISP looks pretty funny to me:

```
(condition-case err         ; Display the time on modeline
    (display-time)
  (error
   (message "Cannot display time %s" (cdr err))))
```

The fact that most of emacs is written in emacs lisp, combined with the fact that the source code is freely available and modifiable, has attracted a lot of programmers as emacs users. What that means for we folk of the nineties is that there is a huge bunch of actively maintained emacs lisp around that we can use.

Often if you think to yourself, "I wish I could do (*whatever*) inside emacs," you'll find you *can*, you need only turn on the functionality that some programmer somewhere has already provided! Even if no one has already written code to do it, you still can, since you can add your own features to emacs by writing emacs lisp.

1.3 subprocess control

Much of what makes emacs so convenient and powerful is not really emacs itself but the way emacs interfaces with the other UNIX tools around it. You can, for example, select a region of text and pipe it through any external command, optionally replacing the original text with the external command's output.

dired, the directory editor — Emacs interacts with the *ls* command to provide a filesystem browser and editor, **dired**. **Dired** lets the emacs user zip around directory listings, reading, deleting, creating, copying, moving files ... whatever.

ange-ftp — Another example is the **ange-ftp** package, which extends dired so that it will work over an ftp session. **Ange-ftp** allows the user to browse and access files that reside on another machine.

This feature really kicks bootay because it allows you to use emacs to edit remote files without having to run emacs on the remote host. A lot of people take issue with emacs for the large amount of memory it consumes. **Ange-ftp** allows you to use the memory resources of your desktop machine to work on remote files without taxing the remote host's resources.

shell buffers — It's possible to run a shell inside a buffer in emacs. You give the command, **ansi-term**, and a prompt appears inside of an emacs window. This mode is pretty weird at first, but once you get used to it, it is handy as all get out. You can do something that takes a long time in an **ansi-term** buffer, switch to another buffer to do something else, and then return, getting the text from the **ansi-term** buffer to edit elsewhere.

compile — Of course, you can **compile** the programs that you are editing in an emacs buffer without leaving emacs. The **compile** command launches *make -k* by default, but you can tell it to run *gcc -g -Wall -O myprog.c* or whatever you like. The results of the command will be displayed in a little window. If you're using gcc, there are a lot of features, including the handy ability to jump to the lines where compilation errrors occured.

gdb in the GUD — Along those lines, one of the coolest-looking things that emacs does is to run *gdb* asynchronously such that you can see and edit the source you're debugging in one window while talking to gdb in another. This feature is called the **GUD**, or **Grand Unified Debugger**. Not many people know about it, and it's the best C debugging environment I've seen.

ssh forwarding channels — Mark Plaksin, one of the CHUGALUGgers here, has written some emacs lisp that sets up **ssh forwarding channels** as asynchronous processes. I use his these channels to encrypt the control sessions for **ange-ftp**. That way, my password doesn't have to be transmitted clear text. I also do **IMAP** in emacs over a ssh-forwarded channel.

the info viewer — **Info**, the newer hypertext-style GNU documentation system, is integrated with emacs ... I think as a subprocess, but don't quote me on that.

diff and ediff — **Ediff** is an emacs mode that handles the output of the UNIX *diff* command in an intuitive and convenient way.

1.4 text editing

Emacs is, at heart, a text editor, and it has plenty of features that make it easy to write and modify a lot of text, very quickly, once you get used to it.

I may have forgotten what the niftiest of these features are, since they are very easy to internalize and then take for granted, but I'll make a stab at listing a couple of the niftier ones.

the kill ring — One feature that changes the way we use cut and paste is the kill ring. Instead of the clipboard as the conceptual place where text goes when it has been cut or copied, there is a ring of clipboards. Not only the last thing you cut, but the last nine or so things are stored in the kill ring. By the way, emacs uses the words *kill* and *yank* instead of *cut* and *paste*. I remember reading some guy who was complaining about that, but he can sit on it.

(Just kidding. It was Robert J. Chassell, and he doesn't really have to sit on it.)

multi-undo — A lifesaver is the multiple-undo feature. Similarily to the kill-ring, emacs maintains a list of the last few things you've done. You can undo not only the last thing you did, but the last several things. That means that you can go crazy, editing like a wildcat, without fear that you will mess something up.

keyboard macros — Keyboard macros are a lightweight and quick way to do repeditive editing. You tell emacs to record what you do, you do some editing stuff, and then you tell emacs you're done. After that, you can call the new keyboard-macro as many times as you want. Of course, you can also write some elisp to automate things you do all the time.

rectangle — There is a feature called rectangle that allows you to select regions of text in a rectangular fashion. For example, you could select the second column (without selecting the first) from this table:

```
1       lunch
2       dinner
3       brunch
```

I find **rectangle** to be very helpful for working with data from databases.

2 Programming

2.1 language-specific modes

When someone writes elisp that is provides support for a specific task, the new features are grouped together in a *mode*. Many programming languages have their own special mode. A mode generally offers such features as:

Syntax highlighting —

Many people are struck by this feature the first time they notice someone using emacs. The different syntactic elements of the programming language are shown with different distinctive textual characteristics. For example, C keywords will be a white while strings are yellow.

This feature can greatly enhance readability. It also has the advantage of highlighting certain errors, e.g., runaway strings.

Indentation —

Indentation can specify programmatic meaning (e.g., in Python) or can make the code easier for humans to read (e.g., Perl, C). Language-specific modes often override your tab key to make it invoke indentation that's "correct" for the language you're using. Of course, you can customize the indentation scheme all you want.

Not only does this feature save keystrokes, it also helps to avoid simple typographical errors. If you notice that indentation is all messed up, it often means that you forgot a closing brace or some other syntactically significant element.

Support for related tools —

A programming mode often comes ready to interact with the tools that are commonly used in working with that language. C-mode, for example, has excellent support for the GNU debugger (see section 1.3).

2.2 make

The UNIX tool *make* changes everything. Make turns emacs into an IDE for anything by making it simple to do complex, interrelated tasks with one keystroke.

Make is a bit like a simple shell script, but it also has rules for dependencies. It is most often used in programming to compile only the files that need to be compiled.

For example, if I am working on a large Java project that has many class files and I change the source for a class file that my application depends on, I want that class to be recompiled before my application is compiled. Make allows me to specify the dependencies in a *makefile* so that only the files that need to be compiled get compiled.

Since emacs' *compile* command runs make by default, programming goes *much* faster after binding the compile command to your favorite keystroke! Compiling often and quickly, with all of the compiler's warnings turned on, is a great way to increase the quality and productivity of your coding efforts.

After getting acquainted with make, you may find yourself using it for all kinds of stuff — anything where one thing depends on another and automation is nice.

2.3 interesting UNIX programming tools (besides make)

version control —

CVS is a version control system that is becoming very popular in the free-software community. CVS stands for Concurrent Versioning System. Many programmers can work on the same sources at the same time — hence "concurrent".

the Bourne shell —

In UNIX, we recognize the *shell* as the software that offers a commandline prompt. We type commands and the shell uses our commands to execute programs for us, offering a few features along the way.

But sometimes the shell has no prompt—that's when the shell executes commands one after the other, in *batch* mode. The *Bourne shell*, named after the man who wrote it, is one of the earlier shells.

Because it is easy to create *scripts*, or text files containing lots of shell commands, for the Bourne shell, it has become the standard shell for batch shell programming. All of the initialization scripts in your `/etc/rc.d/init.d` directory are Bourne-shell scripts.

The Free Software Foundation wrote an improved version of the Bourne shell called, "Bash," short for "the Bourne-again Shell." It retains the shell-programming features that make it superior to the C-shell for scripting purposes, [1] but it adds many features that make the command line convenient and productive for an interactive user.

Here's an example of a shell script:

```
#! /bin/sh
echo ------------------------------------- filesystem usage
df -k
echo ------------------------------------- recent logins
```

[1] See *Unix Power Tools*, by Tim O'Reilley et al., the chapter entitled, "Shell Programming with the C-shell —NOT!!!" for a tirade against using the C-shell for scripts.

```
last | head
```

The first line tells the OS that the Bourne shell should execute this script, and the subsequent commands are executed sequentially, just as if the user had typed them in at a prompt. Things get interesting when programming logic like loops and conditionals get involved:

```
#! /bin/sh

# this stuff is a comment.  the following code says, "if
# the user isn't root, yell at them on the standard error
# and exit with an error code of 1.

if [ "$LOGNAME" != "root" ]; then
        echo You must be root to run this script 1>&2
        exit 1
fi

# this stuff is a comment.  the following code says "set
# $f to each of "/tmp" then "/var\-/tmp", and finally,
# "/home", iterating the "do" loop for each value of $f.

echo ---------------------------- report on SUID, SGID files
for f in /tmp /var/tmp /home; do
        find $f -type f \( -perm -04000 -o -perm -02000 \)
done
```

sed —

Sed is a very old-school UNIX tool. It doesn't have a lot of hype about it; it's just simple, very useful, and entirely cryptic to those who haven't read any documenation on it.

Sed stands for "stream editor" and is an offshoot of *ed*. Ed is a line editor whose influence permeates UNIX. People found that they could automate ed to do batch edits on files. Sed took that feature even farther, making it possible to do automated editing of not only files on disk but of streams too, pipes and such!

Sed has a feature called *regular expressions*. Regular expressions are extremely important in text processing, since they are a little language that allows a user to specify exactly what type of text they mean. For example, here is a user typing at the prompt, "myhost$", and using a character-class regular expression, `[oa]` to catch any of the bracketed characters (*viz.*, 'o' and 'a'):

```
myhost$ echo foobar
foobar
myhost$ echo foobar | sed 's/o/-GARGOYLE-/'
f-GARGOYLE-obar
myhost$ echo foobar | sed 's/o/-GARGOYLE-/g'
f-GARGOYLE--GARGOYLE-bar
myhost$ echo foobar | sed 's/[oa]/-GARGOYLE-/g'
f-GARGOYLE--GARGOYLE-b-GARGOYLE-r
```

Sed can do some very helpful stuff and is small and fast.

awk —

When people mention sed, they often mention *awk* in the same breath. Awk is named for its three initial developers, Aho, Weinburg, and Kernighan(!). If you know those names, you may have guessed already that awk is something else.

Larry Wall modeled much of perl after awk, so many people will tell you that there's no reason to learn awk, since perl can do everything awk can and more. I would agree that perl can do awk stuff and more, but I learned sed and awk before learning perl, and I'm very glad I did. I feel like I have a better understanding of perl, and I don't *have* to use perl when sed would do just fine, or when the awk script would be shorter and easier to read than the corresponding perl script.

Awk, by default, has an implicit loop, going over its input line by line:

```
#! /usr/bin/awk -f

# the code in braces applies only to lines matching the
# regular expression between the slashes

/[gG]argoyle/ {
        print "The following line has a gargoyle in it!"
        print $0
}

# this code applies to every line, since it is preceded
# by no condition

{
        print "column one: " $1 "\tcolumn three: " $3
}
```

Here's a common use of awk, `awk '{ print $1 }'`. Awk prints only the PID column from the output of the *ps* command:

```
#! /bin/sh
kill `ps axw | grep [b]adprocess | awk '{ print $1}'`
```

perl —

Perl is the "practical extraction and report language" or the "pathologically eclectic rubbish lister", depending on what mood Larry Wall is in. Larry Wall wrote the fine UNIX tool *patch*, and he wrote perl because awk couldn't do everything he needed.

Perl has great text-processing support, including the richest regular-expression features on the planet. It is designed for practicality, meaning that it is short on hype, sometimes short on "elegance," but it is speedy and intuitive to program, once you get the hang of it!

Larry Wall intended for perl to have syntactic similarity to C but with some of the looseness of everyday speech. There is, for example, a special variable in perl for "it."

One of the things that makes perl so popular is the wide body of reusable code that's distributed in the form of "perl modules." A lot of languages talk about code reuse; perl does it in a big way! There is a centralized system for finding perl modules, CPAN, and you can get all kinds of modules.

The most popular modules are *CGI* and *DBI*. The CGI module allows the perl programmer to write scripts that use the CGI protocol for dynamic web applications. The DBI modules gives a standard API for database programming, so that you can easily use all kinds of relational databases from your perl scripts.

Perl offers object-oriented programming support in a very practical way, allowing the programmer to take advantage of inheritance, polymorphism, and data encapsulation; however, the programmer doesn't have to abide by OOP principles, and can violate them—the data encapsulation, for example.

Here is an example CGI script (this is untested):

```
#! /usr/bin/perl -w

use CGI;
use strict;

my $cgi          = new CGI;
print $cgi->header;
print $cgi->begin_html(-title, "example CGI");

print "<p>You submitted a form with the following value "
        . "in the \"phone_number\" text field: ";
```

```
my $phone       = $cgi->param("phone_number");

print( (defined $phone) ? $phone : "(no value defined)" );

print $cgi->end_html;
```

Perl blurs the line between a tool and a programming language and so is a nice segué into the next section.

2.4 programming languages

We have already discussed several: elisp, Bourne-shell scripting, awk, and perl. Here is a look at several other languages. Note that there are way too many to list them all or to be fair in presenting them objectively. I will stick to the languages that I've used. That means I'm skipping languages that other people like a lot, like Python, FORTRAN, Smalltalk, etc.

2.4.1 compiled languages

C —

C and UNIX grew up together. C was originally designed to be a systems programming language, meaning that it was for writing things like OS'es (the Linux kernel is written in C) and compilers. C has a reputation for being the language to use if speed is a priority. It resembles algebra, with english-like keywords thrown in, but the sense of it is still pretty close to the way computers think.

That is, if you say in C, "take the value of x, add 1 to it, and store the result in y," then it looks like this:

```
y = x + 1;
```

One of the distinguishing features of C is its use of **pointers**. Many people hate or scorn pointers because many bugs involve pointers, but those people are anathema. Pointers are not hard once you learn to use them.

A pointer is a variable whose value is a memory address. Pointers have a bad reputation because people often try to use a pointer that does not hold the address of writable memory. Pointers allow the programmer to do just about anything. That can be good or bad, depending on the programmer.

While there are languages that are newer and more fashionable, C is stable. You can be pretty sure that after you read the (relatively short) book, *The C Programming Language*, by Kernighan and Ritchie, you have a strong basis for getting to work.[2] All the features have been implemented and thoroughly debugged, so it's easy to be productive using C.

As an example, most of the software on a typical Linux system was written in C. The Linux kernel is C, and most GNU software is C. Since so many people know C, it is a good language for free software or any project where unknown numbers of people may contribute.

C++ —

C++ is a language that was developed by Bjarne Stroustrup. His stated goal was to create a programming language that "makes programming more enjoyable for serious programmers," by making a better C that supports data abstraction and object-oriented programming. [1], p.110.

Stroustrup tended to allow C++ to grow, allowing the language to become bigger and more complex, with more features, if it seemed to serve the needs of programmers with complex problems. The idea was that even if a programmer doesn't need all the features of C++, "What you don't know won't hurt you," i.e., you can use C++ without knowing all of C++. [1], p. 113

C++ is one of many languages that offers support for newer styles of programming, especially object-oriented programming (OOP). What distinguishes C++ is that it tries to offer these features with *no* performance cost, so that an OO program should run as fast as a conventionally-programmed one.

In order to provide OOP support without sacrificing performance, C++ relies heavily on *inlining*. Inlining is the technique of expanding all the code you can by the time it gets to be runtime. Without getting too much into it, suffice to say that C++ programs are often fast because of this approach but are often big in the computer's memory, too, which can sometimes affect performance adversely, negating the benefit of inlining.

A C++ programmer who is aware of specifically how his code will be implemented by the C++ compiler can use C++ features such that his programs don't suffer from "bloat," but at that point, the benefits of OOP are dissolving, since the programming is being done in accordance with the implementation. YMMV.

Objective-C —

Objective-C (Obj-C) is much less popular than C++. It grew up about the same time as C++, but its initial developer, Brad Cox, is no where near as prolific or outspoken as Bjarne Stroustrup. Also, Obj-C has been used primarily in commercial settings, like the Stepstone Corporation, Cox's company, and NeXTSTEP, which is now owned by Apple.

Apple continues to sit on Obj-C. It owns the trademark on the name "Objective-C", yet if you ask an Apple rep about it, they will probably tell you how great Java is!

I used to think Obj-C was God's gift to programmers, doing what C++ tries to do, but with only a few simple extensions to ANSI C. Obj-C takes a Smalltalk approach to programming,[2] thinking in terms of objects and messages.

Unlike C++, Obj-C does not try to offer OOP features at no cost. Instead, Obj-C is very fast but incurs overhead from a runtime support system that is much more dynamic than C++. For instance, up until recently C++ did not have RTTI, or runtime type information, as a feature, but the idea that you can send a message to an object and get information about it is integral to the Smalltalk paradigm that Obj-C is coming from.

Unfortunately, the Obj-C community is plagued by a certain amount of mutual resentment and stagnation. The GNU folks have gone the way of NeXTSTEP, and the people who support Brad Cox's style of Obj-C don't like it. The GNU folks are trying to make a GNU version of NeXTSTEP (see 2.5), which is a very ambitious undertaking. If they succeed, Obj-C might have a resurgence of popularity.

Java —

Java is really an interpreted language, but it's interpreted by a "virtual machine" after getting compiled into bytecode by a Java compiler. That's a little like perl ...

Anyway, the virtual machine is a program that runs on your computer. It reads *bytecode*, which is just preprocessed Java code in a streamlined binary form, and then takes action based on that bytecode. The idea is that the extra layer leads to portability, since all you have to do is make a virtual machine for lots of machine architectures and then your Java code will compile and run on all those machines.

That is not a new idea, but Sun has been very successful in presenting it as such, and now Java is very popular. The language offers OOP support and has a syntax that's similar to C++, but Obj-C gurus claim that Sun modeled Java's OOP features after Obj-C. Indeed, the OOP styles of Java and Obj-C are more like each other than they are to C++.

There is one profound difference between Java and other OOP languages: the security model. In Java, you can actually run Java code that someone you don't know wrote while constraining the foreign code to only a narrowly-defined set of actions. While that may be possible with other programming platforms, Java has it right now.

If you don't need the security feature, you might not need Java. Java is very slow compared to other languages, and while it is easy to write GUI applications in Java, the clunkiness of Java apps is a deterrent for many programmers. The clunkiness results from Java's distance from the

[2]Smalltalk is one of the first and most influential OOP languages. Usually in the form of a complete GUI-based commercial programming system, Smalltalk represents *everything* in a program as an object.

native implementation; however, they're working on that. Today's Java programs are less clunky than last years. But that's another issue: the Java APIs change a lot and often.

2.4.2 document languages

Besides programming languages for developing systems and applications software, there are also languages for the communication of information. An interesting tension in these languages is between the definition of structure versus the definition of presentation. Structure has to do with what the information is; e.g., "this is an article's abstract," "this is a product description," and is sometimes called, fancily, *metadata*.

Presentation, on the other hand, has to do with the way something looks; e.g., "that paragraph should be above the picture." Some languages, like SGML (including HTML), were designed with structure definition in mind, while others, like PostScript, don't really care what is being presented.

For information on SGML and its relatives HTML and XML, see the W3 website:

`http://www.w3.org/`

SGML —

Standard Generalized Markup Language (SGML) is pretty old now. It is a non-proprietary language used to define documents, both the logical structure and the content of the documents.

SGML is very abstract and general. From what I've seen, SGML is usually converted into another less abstract form before it is used. Linux has a set of SGML tools (see the sgmltools(1) manpage) that can convert SGML into other more easily used forms such as HTML and LaTeX code.[3]

Many languages have been influenced by SGML or are subsets of SGML.

HTML —

Hypertext Markup Language (HTML) is a subset of SGML. It is the ASCII fabric of the World Wide Web. HTML is non-proprietary and standardized. It is a simpler SGML, oriented toward defining the logical structure of content.

The idea is that by putting the definition of the content's logical structure inside the HTML document, a browser can present that structure any way that makes sense. That way, a person with blindness could benefit from an HTML document just as could a person with sight.

For example, browsers for people with sight would recognize the `<em>` (emphasis) tag and display the emphasized element in boldface or italics,

[3] I had trouble with them, though. Some of the tools don't seem to work properly.

while a browser for a person without sight would change the intonation of the speech software's "voice."

The flexibility of this approach runs contrary to the tendency of authors everywhere (and the authors' bosses') tendency to say, "No, it would look better in boldface than italic," taking control from the browser and putting it into the language. So HTML has a lot of proprietary extensions, introduced by the browser vendors, that allow users to more directly influence presentation. HTML is a bit messy now, as a consequence, but it is still a very easy language to learn and use.

XML —

Extensible Markup Language (XML) is the "next big thing" in the SGML family. Many people think XML is great but still are a bit foggy as to exactly how it will be great. (I'm one.) XML uses tags like HTML does, but a tag like `<p>` might not mean what you think. It depends on the application that is interpreting the XML.

The interpretation of the XML tags depends on the application reading it. Unfortunately, that is so general that not many people are using XML right now. However, it could lead to neat things. You could store XML in a database such that the description of what the data is was right there in the database. By storing style sheets and other stuff with the XML, you could have a very powerful system.

Perl (and Python, I've heard) have nice facilities for making your own XML interpreting applications. When I had trouble using the sgmltools I wrote an XML to plain TeX converter using perl, and it has been very handy for making documentation. If I wanted some form other than plain TeX—say HTML—it would be pretty easy to write a little perl code and have the interpreter.

TeX —

Donald Knuth,[4] a very famous programmer at Stanford, wrote TeX so that he would be able to beautifully typeset his series, *The Art of Computer Programming*. Once I told my friend Tom White that I'd like to have his old copies of *The Art of Computer Programming*, and when I started reading the first book, I was laughing out loud. Knuth's introduction for that series is very funny. A few more minutes later, though, my head hurt. Tom said, "You want Knuth? You want Knuth!? You can't handle Knuth!!" referring to a movie where somebody says that about the truth.

Knuth is an interesting guy. He's a programming genius, but he thinks a little bit differently than most people. TeX is his baby. TeX is famous for the professional quality of the documents it produces, especially the quality of typeset mathematics, like this:

[4] Like so many things in the world of computers, Knuth is not pronounced as you'd expect. It's *kuh-nooth*. Similarily, TeX rhymes with "tech", and LaTeX sounds like, "lay tech."

$$\int \sum_{n=0}^{100} x^2 1/4\sqrt{x}$$

Knuth developed it for years and then put a feature freeze on it. He actually *pays* people who can find bugs in TEX!! But no one can find any. It is a great piece of software, able to do tremendous amounts of work on dinky machines. However, it is a bit weird.

TEX is a little programming language that defines a typeset document. I can say programming language because it has things that you'd expect in a programming language: conditionals, flow control, variables, etc. Someone has written a BASIC interpreter in TEX. That is silly, but it gives you an idea of the power Knuth gives his users.

LATEX —

Many folks create packages of TEX code that handle a lot of details on behalf of the user, freeing the user to concentrate on the content of the documents. By far the most popular *macro package* is LATEX. Leslie Lamport made LATEX, along with lots of other people who added packages later.

If you want to do something normal, like write a dissertation, LATEX can be great. You can find and use a dissertation package, and you'll be good to go.

If you want to do more, you may run into the same problem I did: LATEX is a messy bunch of sometimes conflicting and endlessly proliferating packages and commands. If you want to do anything very unusual, you may have to learn not only the high-level user-oriented side of the packages involved, but also the low-level code that the package maintainers wrote.

Many LATEX packages assume that PostScript is the output format and rely on PostScript. That can cause problems today, when PDF is often the preferred medium, and pdfTEX is the TEX variant you're using.[5]

ConTEXt —

ConTEXt, like LATEX, is a typesetting system based on TEX. However, ConTEXt is more strongly rooted in the "plain" set of macros that were developed by Knuth. ConTEXt is a product of a company in the Netherlands, but although they maintain control over it, they do release the sources and work with outside developers. This centralized development, combined with a great sense of vision for what documents can be today when computers have more power and are often networked, has made ConTEXt the most coherent, mature, and powerful TEX–based system today.

Among the highlights that make ConTEXt distinctive are:

[5] pdfTEX is a new variant version of TEX. Unlike TEX, which produces *DVI*, or device independent, format output files, pdfTEX produces PDF directly

- Interactivity. ConTEXt can produce PDF documents that interact with the user through form elements, buttons, and hyperlinks. The ConTEXt folks even went so far as to make a PDF document that is a calculator!
- Integrated fancy stuff. Most TEX-based systems rely on postprocessors, especially PostScript-fiddling software, to handle tricky stuff. ConTEXt knows that many people are going to be using pdfTEX, so it lets you do things right in your document source. Inline reusable METAPOST figures, for example.
- Fewer conflicts. If, in LATEX, you'd like to have one column at the top of the page and then switch to two columns for the rest of the document, you can, but it breaks a separate feature—floats (figure inclusions and other things that have an undefined placement until the document is typeset). In ConTEXt you can have your multicolumns and your floats too.
- Fewer commands. LATEX has lots of packages, and each package has lots of commands. ConTEXt has a set of commands that follow a predictable pattern. Even if you don't remember a command, chances are you can guess what it is.

troff —

Troff is an old-school UNIX typesetting program that can produce multiple formats. If you want to write your own man pages, it's not too hard to learn a little troff (maybe from Aileen Frisch's book, *Essential System Administration*) and do it.

PostScript —

Adobe Systems, Incorporated, made PostScript. PostScript is great for describing exactly how something should look. It's a programming language, complete with variables, conditionals, loops, etc., but most people just use applications that generate PostScript so that they can print anywhere and the document will look the same.

METAPOST —

Donald Knuth, the author of TEX, made a font-description language called METAFONT. It generates raster graphics output. John Hobby of Bell Labs made a language called METAPOST that produces vector graphics in PostScript.[6]

It is possible to include METAPOST figures in documents produced with TEX. That opens up a lot of possibilities, since with METAPOST it is easy

[6]Raster graphics are defined by lots of little dots. Like your CRT display, each dot must be described individually. JPEGS are an example of raster images. Vector graphics are defined by describing shapes, lines, and filled areas mathematically, rather than by defining a set of dots. That means that vector graphics are often smaller and can scale well. Macromedia's Flash format is an example of vector graphics.

to do graphics in a portable way. You can do shapes and shading; you can typeset text within METAPOST figures; you can do color ... pretty much anything you can do with PostScript, you can do with METAPOST.

2.4.3 database languages

SQL —

The standard language for working with relational database management systems (RDBMs) like PostGres and MySQL.

2.5 programming APIs, libraries, and environments

libc —

Libc is the standard C library. Type “info libc” for a *lot* of information.

Gnome —

GNOME is the GNU Networked Object Model Environment, I think. It is supposed to make free software more accessible and friendly to users. If you are interested in making GUI applications, GNOME is worth looking at.

gnome-objc —

Last time I checked the Obj-C support of GNOME was undocumented and unfinished, but since GNUstep is elaborate and unfinished, maybe this gnome-objc will get finished, and be useful to Obj-C programmers, first.

GTK —

The Gimp Toolkit is a GNU project in C that gives the programmer lots of widgets. It seems nice, but I’ve only played with it a little.

GNUstep —

GNUstep is still developing. When I was trying it out as a programming platform, it was very difficult to get it all set up and to use it. They say that the current version is much easier to use.

It’s supposed to be a cross-platform programmer’s paradise based on NeXTSTEP (who is now owned by Apple). With Obj-C as its primary language, it will have lots of GUI objects for the programmer to use, as well as utility stuff like vectors and “dictionaries” (hash tables or associative arrays). It will make it easy to do distributed computing, with your GNUstep application running on multiple architectures in parallel.

pthreads —

The POSIX thread API is a standard that has been implemented on Linux and many other systems. That means that you can write multi-threaded code that will easily port to other platforms. Multi-threading is a way for one process to do more than one thing at once; e.g., the process could be waiting for the network while reacting to input from the user. If you like low-level C programming, this is a nice and fun API.

glib —

Glib is a library that makes C programming easy. It was developed as a portability library so that GNU software could be more easily ported to multiple architectures, but it has all kinds of features. It has memory management features, linked lists, hash tables, nice error-handling stuff. Check it out.

gmp —

GNU MP — the GNU multiple precision arithmetic library. It's a library based on the BSD MP library, but with a more consistent interface. I really liked it when I tried it: you just make a little struct and then you can do math with absurdly precise numbers by using strings with lots of zeros after the decimal point.

References

[1] Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison Wesley, 1994

[2] Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall, 1988