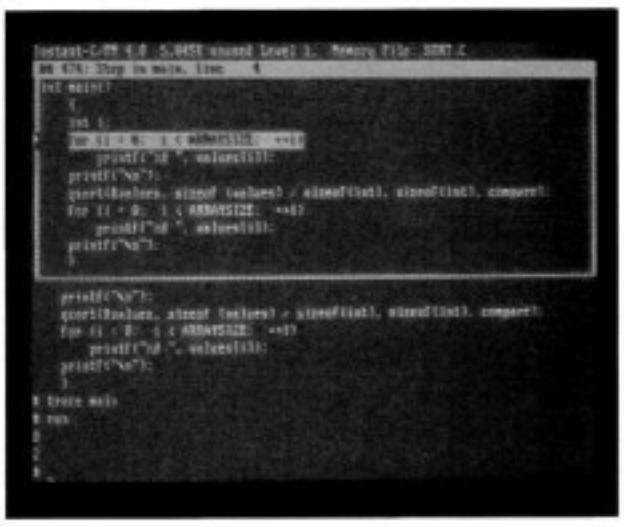
Martin Heller

REVIEW

Just Add Water



Instant-C 4.0's windowed debugging environment.

Most C programmers don't spend much time thinking about Lisp, but the technical staff at Rational Systems did. They looked at the productivity of Lisp programmers and concluded that C was a clod by comparison. Rational set out to create an environment that would give C programmers all the conveniences available at a Lisp workstation, including immediate execution, incremental compilation, automatic formatting, easy browsing, and built-in source-level debugging. When the wraps came off, Instant-C was born.

Rational Systems' first attempt to boost C programmers' productivity was stymied a bit by a popular misconception: Too many potential users thought Instant-C was a C interpreter. So, instead of touting its unique abilities, Rational found itself in unwilling competition with other interpretive versions of C.

Instant-C 4.0 has come a long way from that first "C interpreter." It's not built on an interpreter (and never was) but rather on an incremental compiler. The difference is speed: Instant-C runs your code a good 10 times faster than C interpreters but a factor of two or three times slower than most optimizing compilers.

Instant-C 4.0

Company
Rational Systems, Inc.
220 North Main St.
Natick, MA 01760
(508) 653-6006

Hardware Needed
A 286- or 386-based PC with at least 1 MB of RAM

Price
$795

Inquiry 881.

Rising Above Adversity

Given that the incremental compiler, the monitor, the debugger, your compiled program, and your source code all have share memory, the 640K bytes available under DOS was a crippling limitation for earlier versions of Instant-C. Instant-C 4.0 surmounts the DOS memory limit using Rational's DOS/16M DOS extender to run in protected mode. For programs that break the rules of protected mode, Instant-C 4.0 offers, in addition to pure protected mode, a mixed mode of operation where Instant-C itself runs in protected mode, and your compiled program runs in real mode.

Instant-C (and DOS/16M) honors the Virtual Control Program Interface convention for managing protected mode. I ran Instant-C from my normal DOS environment, which includes DOS 3.3, 386Max, Super PC-Kwik Power Pak, the Hitachi CD-ROM driver, Microsoft's CD-ROM extensions, and the Logitech mouse driver. Getting all this stuff to work at once is something of a juggling act. Some other protected-mode programs, which do not honor VCPI, forced me to reboot my computer with a simpler configuration before I could use them.

How good is Instant-C's protected mode? I had Instant-C and all of the image-display program (IMDISP), which I will discuss later in this article, loaded in high memory in protected mode. I wrote this article in Sprint running in a real-mode DOS session shelled out from Instant-C. I could even shell out again from Sprint to execute DOS functions; when I finally closed Sprint, I could still go back immediately to Instant-C. That's productivity.

Cycling Around the Learning Curve

At one time, the standard picture of how software is developed was the "waterfall model." In it, software is first specified, then designed, then implemented, and finally tested.

My own development style involves many iterations in the development process. When I write code, I quickly build up a user interface—sort of an armature—that reflects the design of the program as a whole but doesn't implement too much code that actually does anything. The users get to see the shell and comment while I start to code the most important innards. At weekly intervals, stable versions with additional features go out for testing and comment; I then fold the test results into the next week's development.

Working in this way lets me break the development process into tasks lasting one to five days each. While I work on each task, the task's module is fluid while most of the rest of the program is frozen. I spend a lot of time editing, compiling, linking, and testing. Running the code teaches me things I didn't know when I was designing, so I modify the design as I go.

Instant-C shortens the cycle time once I'm into a project, but it has its own start-up costs. With the program, I can make a change in a function and be ready to start testing (with full source-level debugging) in 10 seconds. That's an enormous improvement in cycle time, and I can get much more accomplished. By issuing calls to a new function from the Instant-C command line, I can even test out of context.

However, the learning curve for Instant-C is steep. To begin with, I had to build a version of Instant-C with the Microsoft libraries I normally use. The standard version of Instant-C uses Rational's own libraries, but tools are supplied that build versions compatible with the large- and medium-model libraries supplied with Microsoft and Lattice C compilers. The rebuild went smoothly.

Then I had to load my code into Instant-C. Understand that this is code that compiles without error in Microsoft C. Instant-C is a little fussier, however. Since it knows about all the modules of a project, it can combine the checking of a compiler, a linker, a debugger, and `lint`.

I originally tried to load a project that embodies over 20,000 lines of C code and uses several third-party libraries: Vermont Views, MDBS IV, Essential Graphics, The Heap Expander, and TurboGeometry. I have source code for all but MDBS IV. As shipped, the program and its overlays take 600K bytes of disk space but run in a little over 300K bytes of code space.

Rational was able to supply me with support files for Vermont Views, Essential Graphics, and some other libraries that I use in other programs. I could easily have disabled my Heap Expander

continued

code, but I was stuck on the problem of supporting MDBS IV.

Instant-C can load object code and object libraries, but you have to tell it what the object code is doing. To do this, you write an .IC file that looks a lot like C prototypes. I wouldn't have too much trouble converting my MDBS IV function prototypes into an Instant-C file, but I already knew that some of the MDBS IV code registers arithmetic, which is forbidden in protected mode. As I didn't have source code and the project was too large for Instant-C's mixed mode, I was forced to choose another project.

The IMDISP Project

IMDISP, written at the Jet Propulsion Laboratory under a NASA contract, is used to display planetary and satellite data. As supplied, IMDISP knows about standard VGA modes, but not about the Super VGA modes of my Video Seven video RAM card. Adding support for my VGA card seemed like a good test for Instant-C, for several reasons: IMDISP includes one assembly module and a lot of DOS- and hardware-dependent code, it's reasonably sized (some 8000 lines of C and 500 lines of assembly), and I wasn't familiar with its code.

When I tried to load IMDISP into Instant-C, I discovered bugs in both. Instant-C helped me find and fix several relatively benign errors in IMDISP, and IMDISP revealed a couple of nasty errors in Instant-C. Rational quickly supplied me with a later version of Instant-C (4.02) in which all the bugs I discovered were fixed. Unfortunately, in the course of working through IMDISP, I discovered a few more problems with Instant-C. As of this writing, the folks at Rational were able to duplicate and isolate all the errors that I found; they told me that these errors would be fixed in the next version of Instant-C. By the time you read this, even that version will be history. With any luck, Instant-C 4.1 will be stable and shipping.

Once I got IMDISP loaded into Instant-C and running, my job was quite easy: Instant-C's browsing and cross-referencing features made finding the code that I needed to modify almost trivial. I could have done the same thing using my standard tools and `grep`, but I would have spent more time doing the edit/compile/link/test cycles. Instant-C's debugger seemed to give me more control and better diagnostics than CodeView—protected mode isn't perfect, but it sure helps in finding uninitialized pointers and other common problems that can be totally baffling under DOS.

The Instant Conclusion

Was all the pain I went through bringing my code into Instant-C worthwhile? Probably. I'd be more likely to start my next project in Instant-C than to start it with my usual tools. I have learned that moving code from Instant-C to a compiler is as painless as the reverse is painful.

On the other hand, Instant-C can't handle Windows code, which sends me back to my usual tools for Windows programs. Nor does Instant-C help much for programs targeted for OS/2. But Instant-C sure helps in developing large (and small) C programs for DOS: It finds errors that seem to elude even hardware-assisted debuggers.

I can wish for more, of course. Currently, Instant-C's editor only handles one object at a time; I would like to have many different functions open at a time, as I can in other DOS programmer's editors. I would also like to see the Instant-C editor integrated with the program's symbol table and its knowledge of C syntax: There is no reason, in principle, that Instant-C could not do truly intelligent template editing and name completion.

At $795, Instant-C isn't cheap. It also won't finish a project for you satisfactorily; you still need an optimizing compiler to generate production-quality code. But as a productivity enhancement tool, even counting an initial week lost to learning time, Instant-C is likely to cover its costs in a couple of months. By my standards, Instant-C is something any full-time, professional C programmer working in DOS will want to have in his or her toolbox. ■

Martin Heller develops software and writes about technical computer applications. He lives in Andover, Massachusetts. He can be contacted on BIX as "mheller."