TUG Lines 52

Communications of the Programmer Resource Organization **January/February 1993**

Perspective

The world is governed more by appearance than by realities, so that it is fully as necessary to seem to know something as it is to know it.

- Daniel Webster (1782-1852)

There was a time when it was possible to keep up with everything happening in the world. I fondly remember the early 70's, when I was easily able to keep up with everything happening in the field of electronics (including computers and programming) by investing 8 hours a week, reading 4 publications. These days I can put in 60 hours a week, and not even keep up with the latest products for my personal computer!

Technology is being used to advance technology, and the compound effect is staggering. The total amount of information produced in the world doubles every four years, and the rate is accelerating. So how can one person possibly keep up with what's going on?

They can't. That's why every one of us is forced to rely on experts. Experts on technology. Experts on finance. Experts on religion. Experts on world affairs. People in whom we place our trust to wade through the mountains of information, to pick what is relevant, and to digest it and provide it to us in a form we can quickly and easily assimilate.

But therein lies a dangerous trap. If we're not careful, we can unknowingly allow our experts to dictate our world view.

It is easy to believe there is some sort of checks-and-balances system applied to the information we receive from the government and the news media. In the last Perspective, I showed that nothing could be further from the truth. Government officials have legislated that the law of physics is incorrect - that freezing doesn't occur until the temperature drops below 28°F - not 32°F. Politicians spend their campaign funds not on gathering information about how to solve problems, but on research to find out what people want to hear; once in office, the product has little to do with the promises.

Nations have collapsed because their people were unwilling or unable to ferret out the truth about what is happening around them. As programmers, many of us have a natural tendency to lock in on what we're doing and to be oblivious to the world. But as professionals, we have an obligation to make our society a better one. Since we need information to do that, and since we can't gather all the information we need by ourselves, we have to pick our own experts. And if you've elected any of the anchorpersons on the Evening News, let me suggest that you call for a recount. In the remainder of this column, I'd like to offer some tips that may help you pick the experts who can be critically important to you, your career, and your future.

What are the qualifications of a true expert? Let me offer 5 thoughts.

1. Experts must have intelligence and a never-ending curiosity. Taken together, these two traits will be the driving force that keeps your expert working for you, digging up new, insightful information related to your interests. Should you discover your expert trotting out the same issues again and again, it's time to consider looking for someone else.

2. Experts should have access to a wide range of information that adequately covers the subject. You are, in effect, hiring your expert to keep tabs on all the information sources you don't have time for, plus some you're not even aware of. If you don't know which sources they're using, ask them. You might be surprised at the answer. It's amazing how often circular information queues get established. You can even subscribe to several different newsletters on a given subject, and you see the same opinion appearing in each newsletter and assume it's coming from several sources. Then you find out that

Continued on page 2

In This Issue

About TUG Lines . . .

TUG Lines is published bi-monthly by TUG/Pro, as a benefit of TUG/Pro membership. TUG Lines is not available on newsstands or by subscription independent of membership.

Address/Phone

TUG/Pro
PO Box 1510
Poulsbo, WA 98370
Voice: 206/779-9508
FAX: 206/779-8311

Editorial Staff

Technical Staff

Member Services

Contributors

Membership. Membership in TUG/Pro is $75.00 per year in the United States; $85.00 per year in Canada and Mexico; and $99.00 per year elsewhere. All monies must be in US dollars. Each year's membership includes a one-year subscription (six issues) to TUG Lines, six newsletter disks, and discounts on TUG/Pro products. For more information, request a copy of our current membership prospectus.

Renewals. The renewal information shown on your TUG Lines label tells you the number of the last issue of TUG Lines you are scheduled to receive. If you don't want to let your membership lapse, be sure to renew right after you receive that issue. Of course, we will send you a reminder.

Advertising. We accept a limited amount of advertising, and our rates are very reasonable. For more information, or for our latest advertising rate sheet, contact Carol Taylor at 206/779-9508.

Articles and Submissions. You are encouraged to share your knowledge and experience by contributing articles and reviews for possible publication in TUG Lines. Write for a free copy of our author's guide.

Mailing List. Our membership list is intended for TUG/Pro business only, and it will not be rented, sold or made available to another party or used for any other purpose.

Continued from Page 1

each of the editors is feeding you material from the other newsletters - and it may well not even be accurate.

3. Experts should have a knowledge of the subject at least as great as yours. After all, you're relying on them to provide you with accurate information. If they don't know as much as you do, how can they be trusted to digest it for you, so the information you get is still correct? I subscribe to a newsletter for writers who market their own work, which is written and edited by a man who has written and published more than 50 books. He's definitely an expert on what he does. But in this day and age, you can't talk about writing without the subject of computers creeping in. And in the approximately 12 years I've received his newsletter, I can't remember one occasion when he wrote more than one paragraph about computers where he didn't say something that was incorrect. I rely on his advice for writing, not computer-related topics.

A problem arises when you're dealing with a topic that's highly subjective - art, for instance. It has long been recognized that art is a matter of personal preference. But there is no lack of experts on the subject, and a couple of months ago, an incident arose that makes an interesting point. One of the most prestigious art museums in the U.S. was about to hold its annual art competition, which was open to the country's foremost artists. Of the 1,500 paintings submitted, only 150 were selected to hang in the gallery, each one methodically chosen by an august panel of experts representing knowledge in all facets of the medium. Everything was fine, until it was revealed that one of the pieces accepted came from a housewife - who had submitted a painting done by her 4-year-old daughter. She did it as a joke, to prove that when it comes to art, the term "expert" is relative.

There is a danger, too, in choosing an expert who knows (or seems to know) more than you'll ever know about a subject. This kind of situation can leave you prey to the classic con man, and many a fool and his money have been parted when an expert asks you to take what they have to say on faith. As I write this, a group of nearly 100 people are holed up and armed to the teeth in Waco, Texas with someone they firmly believe is Jesus Christ returned for the Second Coming. They know he is because he told them so. Unfortunately, it appears that many of them may die in a meaningless shoot-out with federal agents.

4. Experts should be familiar with and cover all sides of an issue. I'll be the first to admit this is a controversial suggestion. But you owe it to yourself to be educated on a topic, and that means knowing everything that's relevant to it. Admittedly, some topics - like computer programming and science - seem to be based only in fact. But when the facts must be integrated into a world view, controversy begins. Would you work for a company who writes software for family planning clinics that provide abortion services? Would you consult with a company that prides itself in polluting the planet?

When it comes to controversial issues, most people tend to pull information only from sources with which they agree, and that's a mistake. The royal families of Europe intermarried to the point they became victims of hemophilia and thus had to be on guard at all times, lest

Continued on Page 3

Global

Continued from Page 2

they would receive a simple injury and bleed to death. In much the same way, looking only to sources with which you're in tune to further yourself can be detrimental. It can cause you to become callused toward other people, and you can become brittle.

Don't get me wrong. Everyone has a bias, and that's not going to change. It's even healthy. But when you listen to only one side of an issue, you're missing part of the truth - even if you're "right". So study both sides of an issue. If you want to know about environmental issues, I recommend two books: Earth in the Balance by Al Gore (Plume, 1993) and Trashing the Planet by Dixie Lee Ray (Regnery Gateway, 1990). You'll be in for some interesting reading from two experts with two very different perspectives.

5. Experts should possess wisdom. In today's world, information is plentiful. Knowledge is relatively easy to find. But wisdom is something else again. It has that ability to cut through all the rubble, separate the wheat from the chaff, and get to the best solution, even when all the information is not available - and frequently flying in the face of knowledge. It's a timeless ability to focus clearly on the framework of principles that underlie a problem. As someone once told me: "I call 'common' sense 'practical' sense, because it isn't all that common."

I hope these 5 thoughts will be helpful to you as you pick the experts you need to advance your career and your life goals. Choose carefully. It can make all the difference.

The next Perspective will be on the subject of Shareware Marketing. In the meantime, I thought you might enjoy a few quotes from some of the finest minds in history ...

There is no likelihood that man can ever tap the power of the atom.

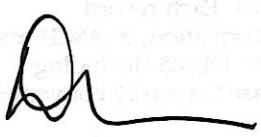
- Robert Milliken, Nobel Prize in Physics, 1923

There has been a great deal said about a 3,000 mile rocket. In my opinion, such a thing is impossible ... we can leave it out of our thinking.

- Cannevar Bush, physicist, MIT, 1945 Nobel Laureate

While theoretically and technically television may be possible, commercially and financially I consider it an impossibility, a development of which we need waste little time dreaming.

- Lee de Forest, American radio pioneer and inventor of the vacuum tube, 1926



Don Taylor

Library Notes

Starting with this issue, we are (by popular demand) bringing back the Product Drawing. For more details (including the product we're giving away this time), see the notice elsewhere in this issue.

Updates

Revision Control System (RCS) is now at version 5.6. The new version removes restrictions from file names, and is completely compatible with earlier versions. The system stores only the "deltas" - the changes - between different versions of your source files, making it easy to jump back to any previous version of your code (which makes it handy to get back from a programming tangent, to something that was working before!). Revisions can be compared and listed, and multiple versions can even be merged together. This disk also includes **GNU Diff**, a powerful file comparison tool. The source code for RCS and Diff is included on the disk. Although both of these utilities are written in C, the system works superbly with any programming language. *Stock number UT-MIS-DOS-011.*

Pretty Good Privacy (PGP) has been updated to version 2.2. The revision includes the fixing of some minor bugs, and the addition of some new features. The PGP system is a collection of high-powered utilities that form a public key encryption system. It offers you a way of encrypting your messages so only those you choose can read them. It also provides a way to "stamp" a message in such a way that you can prove only you could have written it, and it will immediately detect if someone has tried to alter it in any way! The extensive tutorial documentation (207,265 bytes of it) included on the disk is an education in itself. *Stock number UT-MIS-DOS-009.*

F-Prot is now at version 2.07. This is the program we recommend for virus prevention and eradication. It's among the easiest to install, and it is one of the most powerful anti-virus programs available today, regardless of price. We have F-Prot on every one of our machines, and every floppy disk we put in our machines - and every executable file we download from anyone - gets checked with this utility. It's interesting to note the "big" magazines never seem to review F-Prot; that may well be because this incredibly affordable utility outperforms the expensive products that are advertised in the magazines. (Please note: we receive frequent updates of F-Prot, and we immediately put the updates in the TUG/Pro Library. That means whenever you order a copy of F-Prot from us, you will be guaranteed to get the latest possible version.) *Stock number UT-MIS-DOS-008.*

New Disks

The Turbo Debugger Disk contains a variety of helpful files for anyone who is using Borland's stand alone debugger with either DOS or Windows. This disk contains the 3.0 versions of TD386.EXE, TF386.EXE (for use with the Profiler) and the TDH386.SYS device driver. For debugging under Windows, there is TDWIN.DLL. This library should be used in place of WINDEBUG.DLL if you are running TDW 3.0 or TDW 2.5 under Windows 3.1. (It will not work with the Windows 3.1 debugging kernel, however.) Although TDW handles most 2-, 4-, 16- and 256-color high-resolution Super VGA modes, it doesn't support every SVGA card. The solution is to use a special Super VGA DLL. We have included on this disk the complete set of SVGA DLLs from Borland, for use with Turbo Debugger. Along with the DLLs, you'll find a 25K tutorial on the disk that explains which DLL to use with your particular card, and how to use the DLL. Plus, there's nearly 50K of pure questions and answers about Turbo Debugger. *Stock number UT-MIS-DOS-029.*

Programming Tools Disk *11* contains one tool and one utility. The tool is **VSA256**, a C/C++ shareware library which provides access to the VESA 2.0 BIOS extensions for 256-color graphics. The utility is **Event**, a C++ class that lets you gather any event from an overloaded function. (For more details on these two packages, see "Library Notes" in the C Scape section of this issue). *Stock number PT-CPP-DOS-011.*

Utilities Disk *28* is for Windows programmers using C++. It contains **BitMaps**, an image viewer that can read and write Windows BMP files (in several different formats), and can read and display PCX, GIF and icon files. Full C++ source is included. Also on this disk, you'll find **LZSS**, a shareware DLL that offers a powerful file compression library. *Stock number UT-CPP-WIN-028.*

The POV Disk contains **Persistence of Vision**, an outstanding ray-tracing program. If you've ever wanted to take a graphic and turn it into a truly superb 3-dimensional rendering with accurate shading and shadows, this is the most powerful, most affordable tool you'll find to do the job. The Targa file graphics examples provided by Bob Crawford for his article in this issue (you'll find them on the Newsletter Disk) were created with POV. After seeing them, we immediately brow-beat Bob into providing a copy of POV for the Library. *Stock number UT-MIS-DOS-027.*

WinBatch (version 3.1C) from Wilson WindowWare is a powerful shareware tool for programmers writing applications for MS Windows. WinBatch provides more than 100 functions that can control nearly every aspect of the Windows environment, by simply writing batch code commands. You can easily create dialog boxes with push buttons, check boxes and radio buttons that enable you to seamlessly integrate (and automate) Windows programs, and that can smooth the interface for the user when running DOS programs under Windows. The package includes nearly 250 pages of documentation on disk. *Stock number PT-MIS-WIN-008.*

Ed for Windows from Soft As It Gets is a potent programmer's editor for the MS Windows environment. This disk contains a working demo version of Ed for Windows (the only limitations are the maximum size file you can save, the lack of ability to create custom keyboards, and the lack of the extension language compiler). As editors go, Ed is extremely flexible, and it works smoothly with Pascal, C, C++, and dBase, Clipper, DataFlex and COBOL. You don't have to leave the editor's environment to compile your code - and it includes powerful features like brace matching. (For a detailed description of Ed for Windows, see Dave Chowning's article starting on page 7 of Issue 51.) *Stock number DM-MIS-WIN-005.*

Windows Write is a great low-end word processor that has most of the features you need to write letters, manuals or whatever. It's main problem is the lack of a spelling checker. Enter **WinSpell**. This shareware utility checks your spelling in *any* window. It supervises by intercepting each word you type and comparing those words with an internal dictionary of words known to be correctly spelled. When WinSpell is active, it will notify you of any misspelled words as they are typed by either beeping the computer's speaker, flashing the title bar of the window in which the offending word was typed, or both at the same time. One particularly interesting feature of WinSpell is its ability to spell check the clipboard. This allows WinSpell to spell check any Windows application, including spreadsheets, on which many presentations are dependent. *Stock number UT-MIS-WIN-025.*

The DPMI Disk contains the full text of Version 0.9 of the DPMI specification. If you are considering writing programs using Borland Pascal or Borland C++ that use the DOS Protected Mode Interface to break the 640K barrier, you need this reference document. *Stock number PT-MIS-DOS-009.*

Paint Shop Pro (version 1.02) is a Windows shareware program that will display, convert, alter and print images. In addition, it is a screen capture utility. It supports the file formats BMP, DIB, GIF, IMG, JAS, MAC, MSP, PIC, PCX, RAS, RLE, TGA, TIFF and WPG. Paint Shop Pro displays images in many ways, including zooming in and out. Altering the image includes rotating, resizing, resampling, trimming, filter application, color adjustments, brightness and contrast adjusting, increasing and decreasing the color depth, gamma correcting, and gray scaling. Paint Shop Pro comes with a standard set of filters. It also supports applying, creating, editing, and deleting user-defined filters. Decreasing color depth can be done using a standard palette or an optimized palette. All in all, it is a sophisticated graphics tool that can be used to create illustrations for your documentation or teach art students (young or old). *Stock number GR-GRA-WIN-001.*

The Zip Code Data Disk contains a database covering 41,898 Zip codes in the United States. Each record contains Zip code, city and state information, in ASCII text format that can be imported into any DBMS (including dBASE, Paradox, Access and FoxBase) or easily converted

for your customized data handling routines. *Stock number PT-MIS-MIS-010.*

If you've ever used TapCIS to access the forums on CompuServe, you've probably thought there must be a better way - and you were right. **OzCIS** (version 1.2a) by Steve Sneed is billed as "the friendly navigator" for CIS, and for good reason. It is fully menu-driven, and it has full mouse support, making it about as easy to use as it's ever going to get. OzCIS runs under DOS, but runs beautifully in a DOS session under Windows, too. It is exceptional for working with CIS E-mail; you can compose messages, and easily edit them at any time before you send them. You can view and scroll an incoming message as you're composing your reply - and you can even copy text from the message directly into your reply! You can easily specify a file for upload as a message, and a whole lot more. Written in Turbo Pascal with TurboPower's Object Professional (no source is available - just the executable). Includes an extensive on-disk manual. *Stock number UT-MIS-DOS-026.*

***PC Techniques Disk** 17* contains the source published in Volume 3, Number 5 (the December 1992/January 1993 issue). *Stock number RS-MIS-MIS-816.*

***PC Techniques Disk** 18* contains the source published in Volume 3, Number 6 (the February/March 1993 issue). *Stock number RS-MIS-MIS-817.*

***PC Techniques Disk** 19* contains the source published in Volume 4, Number 1 (the April/May 1993 issue). *Stock number RS-MIS-MIS-818.*

When Ordering Disks

Use the order form that accompanies this issue. Please note that stock numbers ending with "I5" are in 5.25", 360K format. Stock numbers ending with "I3" are 3.5", 1.44M format.

Product Insight

The Truth About DOS 6.0 Compression Ratios

- Can You Trust DIR/C?

Doug Lowe
Mike Murach & Associates
Fresno, California

As its name implies, DoubleSpace can effectively double the capacity of your disk drive by compressing your files. For DOS 6.0, Microsoft added the /C switch to the DIR command to report the compression ratio for files stored on compressed drives. But the first time you use the DIR command's new /C switch, you'll probably be surprised to see that it reports 16:1 compression ratios for many files. After seeing so many files compressed 16:1, you might wonder why Microsoft didn't call DoubleSpace *HextupleSpace* instead.

Are these 16:1 compression ratios exaggerated, or does DoubleSpace really squeeze these files down to one-sixteenth of the space required by the uncompressed files?

A simple experiment will shed a little light on this question. If you've installed DoubleSpace, create a one byte file named ONEBYTE.TXT on a compressed volume. To do that, enter this command:

```
COPY CON ONEBYTE.TXT
```

Then, press the space bar once to add a space to the file and press Ctrl-Z to signal the end of the file. If you then type DIR ONEBYTE.TXT, you'll see that the file contains just one byte.

It's impossible to compress one byte of data, right? Wrong. Type DIR ONEBYTE.TXT /C and you'll see that DoubleSpace says it compressed this file 16:1. What gives?

When you use the DIR/C command, your first assumption will naturally be that the reported compression ratio represents the degree to which DoubleSpace was able to compress each file's data. But the compression ratio displayed by DIR /C isn't based on the degree to which the file's data can be compressed. Instead, it reflects the amount of total disk space saved by compressing the file and storing it on a DoubleSpace volume rather than on an uncompressed volume.

The extraordinary compression ratios reported for some files are not due to compression at all, but rather to the fact that DoubleSpace uses a more efficient method to allocate space to files. Simply put, DoubleSpace volumes are not tied to the notion of "clusters," or "allocation units," the way uncompressed DOS volumes are.

When DOS stores a file on an uncompressed volume, it allocates disk space to the file in increments of whole clusters. The size of each cluster depends on the size of the disk drive. For most disk drives, the cluster size is 2KB or 4KB. For large disk drives (up to 512MB), the cluster size is 8KB. Suppose, for the sake of argument, that you have a 512MB disk with 8KB clusters. The smallest file you can create on this drive will use 8KB of disk space, even if the file contains only one byte of data. That's because DOS allocates space to your files one cluster at a time.

DoubleSpace uses a different allocation strategy. DoubleSpace allocates space to your files one sector (512 bytes) at a time. Actually, DoubleSpace allocates space in clusters, too. But instead of using the same cluster size for the entire volume, the clusters on a DoubleSpace volume are variable-length. Each cluster can be from 1 to 16 sectors in length.

The same one-byte file that would have required a full 8KB cluster on an uncompressed volume now requires just a 512-byte cluster. As a result, DoubleSpace "compressed" this file from 8KB to 512 bytes - that's 16:1 - even though it obviously wasn't able to compress the data at all.

If you'll look closer at the files that report 16:1 compression, you'll see that they're all small files. The 16:1 compression ratio reported for these files is due to DoubleSpace's improved space allocation rather than to actual data compression.

The DIR command has a lesser-known switch, /CH, that reports the compression ratio based on the cluster size of the host volume rather than the cluster size of the DoubleSpace volume. If the host volume has a cluster size of 4KB, small files will show an 8:1 compression ratio instead of 16:1.

Does DIR /CH report a more accurate and fair compression ratio than DIR /C? That depends on your point of view. The ratio reported by DIR /CH lets you see how much space you're saving by using DoubleSpace. For example, your 1-byte file would have required one 4KB cluster on a 200MB uncompressed drive. When you DoubleSpace that drive, it requires only 512 bytes. So the compression ratio of 8:1 is fair. On the other hand, since DoubleSpace lets you store 400MB or more of data on a 200MB drive, it's not unreasonable to base the compression ratio on 8K clusters, since 8KB clusters would be required to support a 400MB uncompressed drive.

The point is that neither DIR /C or DIR /CH report the compression ratio the way you would expect. Unfortunately, there is currently no way to find out the actual degree of compression achieved on your data. We can only hope that Central Point, Symantec, and others are working overtime on it!

***Doug Lowe** is the author of numerous computer books, including The Only DOS Book You'll Ever Need (Second Edition), published by Mike Murach & Associates, Inc. (1-800-221-5528).*

Adventures in Consulting

Filling Your Toolbox

This column is intended to be a "semi-regular" feature that will give you information you need to establish or expand your consulting business. It will spotlight some useful resources, give you some insights into the business, and hopefully encourage you to get started, if you haven't already.

The format will be very flexible - from interviews with experienced consultants to mini-product reviews. If you are consulting and have experiences or recommendations you would like to share with others, please drop us a line at TUG/Pro, PO Box 1510, Poulsbo, WA 98370 or on CompuServe at 70441,1340.

This time we'll cover two products that can quickly make you more productive in two different areas.

WinBatch

If it hasn't already happened, sooner or later you will face a situation where you would like to automate a Windows app, or you would like to interface a DOS-only program with Windows. WinBatch from Wilson WindowWare can help immensely.

Like the DOS batch file language, WinBatch gives you access to many of the features of Windows. And unlike the macro recorder included with Windows, WinBatch lets you write, edit and troubleshoot batch file scripts written with an ASCII editor such as NotePad.

Its more than a hundred available functions permit you to control just about any reasonable feature of Windows, from running a program, right down to changing the wallpaper! Functions are provided for data entry, information display, file management, directory management, disk drive management, window management, program management, string handling, arithmetic, clipboard handling and system control.

One of its powerful capabilities is the ease with which you can create a dialog box - complete with data entry fields, check boxes and radio buttons. A client approached us with a need to integrate a DOS-based disk copying program into a Windows environment. The user wanted to be able to use the program (CopyQM) without having to deal with DOS or the plethora of command line options it offered. Plus, the client wanted to use two different modes of the program, to duplicate 3.5" disks with one drive, and to effectively convert 360K floppies to 3.5" counterparts.

The solution consisted of creating two files. The first was a definition of the dialog box:

File COPYDISK.WBD:

```
[choice^1Copy disks in Drive B:]
[choice^2Convert 360K disks to 1.44M]
```

The second was the actual batch file, which "executed" the dialog box, and used the value returned from it:

File COPYDISK.WBT:

```
DialogBox("Copy 1.44M Disks",
"C:\WINBATCH\CopyDisk.wbd")
if choice == 1 then Run("C:\COPYDISK\CopyQM.COM",
"B: PROTECT VERIFY=DATA RETRIES=3 SILENT")
if choice == 2 then Run("C:\COPYDISK\CopyQM.COM",
"SOURCE=A: B: PROTECT CONVERT=1.44M COMPARE
   SILENT")
```

The final step was to create a Windows icon and tell it when selected to run WinBatch and to use COPYDISK.WBT. When the user clicked the icon, she saw the dialog in Figure 1.

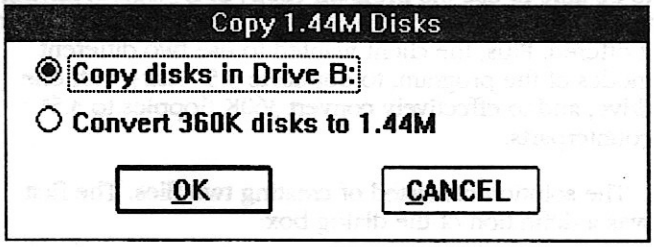


Figure 1. Example WinBatch dialog box.

Of course the program, when executed, looked just like it did when running under DOS. But as when using DOS batch files, we were able to eliminate the user from having to deal with the details. And WinBatch enabled us to make the process look and feel as much like standard Windows fare as possible.

This is a simple example, and it doesn't begin to demonstrate the range and depth of control WinBatch offers. In another situation, we wanted to run LHA and then automatically delete the LZH file it unpacked. Since LHA doesn't offer this option, we used a WinBatch file that first called LHA to unpack the file, and then called a file management function to delete it. It was then we rediscovered the multi-tasking nature of Windows: it launched LHA, then immediately went to the next statement on the list. By the time LHA was loaded and ready to process the file, it had already been deleted!

Once again, the rich set of WinBatch commands came to the rescue. We were easily able to locate the window that LHA was running in, and then halt any further processing of the commands until that window completed its task. Processing was then released, and the file was deleted. Problem solved.

If you're looking for a tool that will provide a nearly seamless interface between Windows and DOS programs - or if you want to create extremely powerful macros for Windows programs - we highly recommend WinBatch. It is a shareware product, complete with nearly 250 pages of documentation on disk, and available from the TUG/Pro Disk Library as stock number PT-MIS-WIN-008.

PackRat

We've tried numerous programs that claim to help get your act together, and PackRat is the best we've found to date for Windows users. It provides features that let you manage phone numbers, names, addresses, phone logs, to-do lists, agendas, finances, projects, time and resources.

PackRat has an incredible phone and address directory. For each entry you can specify name, company, assistant, position, nickname, four key fields of your choice, two complete addresses, any number of phone numbers, and a memo for notes. The phone facility lets you set up rates for clients, and even bill your charges to their account while you're on the phone.

The agenda manager schedules appointments, with all the features you would expect - recurring appointments, reminders, and all of that. There are several ways to set an appointment; one is to click on a time, drag the mouse to establish the length of the appointment, and then type in a summary description. To reschedule an appointment, highlight it and drag it to another time. Each scheduled appointment (or event) can have a note and/or reminder attached to it. PackRat will immediately let you know if you have a schedule conflict.

The Things to Do and Contact features let you assign dates and times to accomplish tasks or to contact people. If it's the day you set up to contact a person, their name will show up on the Contact list. If you want to call them, just click on their name and then click the phone call button - PackRat will dial the number for you, and then monitor the length of the call and compute long distance charges or bill your client for the time if you wish. To Do items can be prioritized at 7 levels. Both To Do and Contact items are color-coded to indicate their status.

Most information-manager products provide separate facilities for each of their functions. PackRat does an excellent job of integrating each of the functions by creating links between data items.

Let's look at an example. You define a project by creating a list of tasks. If you specify the relationship of one task to another, PackRat can create a Gantt chart, showing you which tasks will drive the schedules of others. Now you add resources you have defined to your project, which may include rooms, equipment, people from your phone book - whatever. Now let's say you need to postpone a meeting to a later date. You reschedule the meeting, and PackRat will show you which tasks are effected, and whether you can still meet your completion date. You can easily identify the resources effected, and with a few clicks of the mouse, be dialing the phone numbers of the people involved.

You can upload and download schedules to a Sharp Wizard, so you can carry your schedule in your pocket. Or if you'd rather be more traditional, you can print just about any custom report you can imagine, including perfect knockoffs of DayTimer schedule books.

It is impossible to describe everything PackRat can do in this small space (in fact, it would be difficult to describe what PackRat can do, given an *infinite* amount of space). Suffice it to say that PackRat is one of the most powerful, most flexible tools we've ever seen for managing the mountain of details each one of us faces in today's busy world.

PackRat is a commercial product, currently at Version 4.1. At a retail price of $395 ($695 for the network version), it's not inexpensive. The street price is around $250, which equates to a handful of billable hours. For all the power and convenience it offers, at that price, it's a bargain.

For more information on PackRat, contact Polaris Software, 17150 Via Del Campo, Suite 307, San Diego, CA 92127, 619/674-6500.

Product News

Have you been meaning to learn C++, but just can't get past the hurdles? Que Publishing may just have the answer for you in their latest book, Crash Course in Turbo C++, by Namir Shammas..

The 250-page book takes a fast-track, no-frills approach to teaching the most important aspects of C++ programming. The book covers all the necessities of writing C++ programs, how to debug and fix program problems, and how to work best with object-oriented techniques. Beginning with a clear look at how a C++ program is constructed, the book moves through variables and operators, preprocessor directives, pointers and OOP techniques. Tutorials focus on speed - getting the reader to a basic level of proficiency in the minimum time.

The book retails for $16.95. For more information, contact Que, 11711 N. College Ave., Suite 140, Carmel, Indiana 46032. 800/428-5331.

Pascal

Top of the Heap

Don Taylor

After spending the past couple of months with the Borland Pascal compilers, I like them even better. BP7 is built to work the way I do, and I appreciate the convenience.

The documentation is definitely an improvement over 6.0, but it's still Borland's documentation, and they always seem to leave gaps when explaining anything. Typically, you'll find topics explained in bits and pieces throughout the manual, which means to get the most out of the manual, you must read it so many times you almost commit it to memory. To their credit, they are getting better - and I suppose any shortcoming of the manuals is what makes a living for a lot of aftermarket book authors.

After spending a short time working with BPW, I fell back to a DOS application I've been putting off for several years. When TP6 came out, I got a burn to write the app in Turbo Vision. But after experiencing the TV learning curve, I decided to shelve the project once again.

With the introduction of BP7 and TV 2.0, I've dusted off the project and started it in earnest. After looking at several books that purport to cover Turbo Vision, I've been very disappointed. Most of them use a variation of the examples in the Borland manuals, and in doing so they stay on very safe ground. You see, it's when you try to do anything that strays from using the "canned" objects and examples that you suddenly find yourself up to your armpits in alligators. Most authors avoid those situations, which leads me to a conclusion: either they are unable to explain what they know, or they really don't understand what they're writing about.

During the last few weeks I've gained some insights into Turbo Vision, and through these pages I'm going to share them with you. Plus, you'll be reading articles on TV topics from other authors as well.

If you have completely written TV off as something too complex to deal with, please reconsider - if you want to write a DOS application in Pascal, Turbo Vision really is a viable choice. And if you have plans for any Windows programming, using TV can teach you some of the concepts you will need to know there, without having to take on the entire complexity of Windows all at once.

In this issue, I'll show you how to separate one of Turbo Vision's most powerful features and use it in your programs without using TV.

But first, you're going to be treated to 3 really great articles. Paul Gallagher continues with demonstrating how to "morph" objects in Pascal, Bob Crawford leads off a series on image processing, and Breck Carter will reveal a virtually unknown technique that can, in certain instances, speed up disk I/O significantly.

The TUG/Pro Product Drawing Returns!!

You asked for it - and you've got it. Now each time you order anything from TUG/Pro, you have a chance to win a great game - thanks to Access Software.

Links 386 Pro is the first golf game specifically designed for the graphic capabilities and power of the 386 and 486 - with Super VGA, 256-color graphics. It's got 9 different viewing windows (with 345 combinations!), wide, panoramic views, realistic putting - and it's SoundBlaster compatible. And thanks to Access Software, who publishes Links 386 Pro, it could be yours - free.

Simply order any product from the current TUG/Pro catalog (including the disks introduced in this issue), and use the current order form (with the "52" in the upper corner). We'll put a copy of every order we receive through June 1, 1993 in our contest box, and we'll draw out one form on June 1, and send that lucky person a copy of Links 386 Pro.

To qualify, you must use the "52" order form - or phone your order to us at 206/779-9508, or fax it to us at 206/779-8311 - and we *must* receive it by June 1.

Good Luck!

Object Morphing: State Programming with Turbo Pascal Part 2

Paul Gallagher
Mt. Waverley, Victoria, Australia

In the first part of this series, we examined how the OOP paradigm limits the ability to model real-world problems, and we formulated two ideals for extending the flexibility of objects. In this second of three parts, we will explore several alternatives for turning those ideals into reality.

Pursuing the Utopian Dream

Given an object hierarchy that defines a set of objects that do essentially the same thing (i.e., which have the same basic set of methods), we want a mechanism that allows an instance of one of the objects in the hierarchy to permanently change itself into any one of the remaining object types. It's worth re-iterating that whereas we conventionally talk of inheritance to add functionality, here we are considering inheritance *only* to change functionality.

That's a bit long winded! To clarify the point, refer to Figure 1. If we start off with an instance of `Object2` called "Fred", it should be able to change itself to an object of type `Object1`, `Object3` or even `BaseObject`. Impossible? Let's "step up to bat" and give ourselves three swings at the problem.

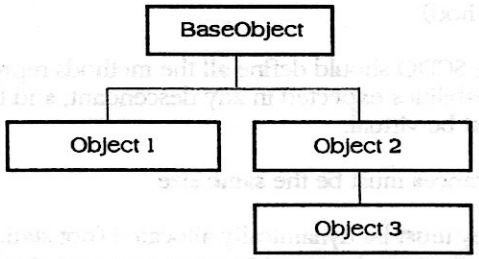


Figure 1. An inheritance example.

Proposal #1 : Typecasting

Turbo Pascal with Objects has no syntactical support for permanently retyping an object instance. Typecasts allow us to access an alternate view of an object's data (fields) and method calls, but re-assignment using typecasts does not change at all the object being manipulated. Typecasts only help us to the extent of overcoming type compatibility problems, while the type selection still needs to be hard coded into the application. Apart from anything else, a proposal based on typecasts throws type-security out the window. Strike one!

Proposal #2 : Instance Swapping

We could consider a "create/swap/dispose" algorithm. A method for it might go something like this:

- Create an instance of the desired new type;
- Copy existing data space into new instance;
- Save a pointer to the current instance;
- Make the "self" pointer point to the new instance; and
- Dispose of the old instance.

The list of problems with this approach is quite long, and quite prohibitive:

1. The absolute address of the supposedly persistent object actually changes. References from other parts of the program to the object instance would become invalid without some fancy "hack" code.

2. The new object type has to be hard coded into the method that swaps types.

3. This is a horrible waste of time - all that memory allocation, copying, deallocation (and you tend to fragment your heap).

4. Dynamic memory requirements peak during the retyping operation. You need to plan an error recovery strategy for when you don't have enough memory.

I'd say that counts as strike two!

Proposal #3 : Metaclasses/Metaobjects

A more serious contender is a proposal to use *metaclasses*. I came across an interesting discussion of this technique by Stephen R. Davis in the *Journal of Object Oriented Programming* ("C++ Objects that Change Their Types", July/August 1992 issue).

Figure 2 illustrates the concept of metaclasses. `UserObject` is an object instance that owns a pointer of type `MetaClassType`. `State1`, `State2` and `State3` are *instances* of objects derived from `MetaClassType`. By re-assigning `MetaClassPtr`, methods of the `StateX` objects can change their apparent type. Basically this is a method of implementing a re-direction layer. The implementation shown here suffers for a number of reasons:

1. You must instantiate all expected object types.

2. Implementing multiple metaclasses of the same type is a convoluted process.

3. Instances of the metaclass must know who called them (i.e., who's got the `MetaClassPtr`), and also must know how to address the other metaclass instances (so that they can change the `MetaClassPtr`).

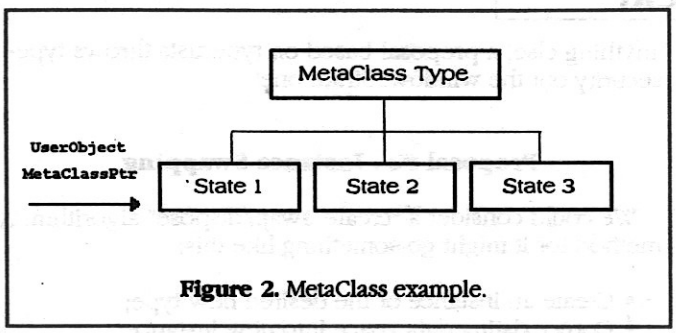


Figure 2. MetaClass example.

4. In short, metaclasses may solve a particular programming problem, but they certainly fall short of our requirements for a generic solution.

That's strike three, and we're out! Well, almost. There is one more alternative.

Object Morphing

What really defines an object at runtime? It's *methods*. If we call the right methods, then data fields follow. When we say we want to "change" an object instance's type, what we are really saying is that we want to change the set of methods that are called.

MetaClasses took us down the path of considering a re-direction layer, but Object Pascal already has a re-direction layer available to objects - virtual methods. A Virtual Method Table (VMT) exists for each object type (provided it has at least one virtual method). Object instances contain a VMT offset field which they use to find the appropriate VMT to make their method calls.

The VMT offset is the first of two attributes that are critical to object type. The second is the object instance's size, which is stored as a field in the VMT.

A natural conclusion: if we make sure the object instances are the same size, then simply modifying the VMT offset effectively lets us change types at will. Now we really *are* talking about changing object types - what I call *object morphing*.

A word of warning is in order. Playing with the object size and the VMT offset are both anti-social activities as far as Borland is concerned, and I don't blame them. The method I will describe is consistent with the internal design of Turbo Pascal v6.0 and v7.0. Any other versions may handle virtual methods differently, and thus require a revised approach.

State Objects

Mutually morphable objects, or *State Objects*, are defined as objects that share a common ancestor, equivalent object size, and transmutable functionality (i.e., they share a common set of method declarations). Together, they comprise a *State Class*. The common ancestor can be considered the State Class Definition Object (SCDO), which defines the boundaries of what the State Class is capable of. Note that the SCDO can also be a State Object in its own right - the "Base State" if you like. Programs using a State Class will limit their understanding of what functionality is available to that which is represented in the SCDO. For this reason, applications will usually only need to explicitly reference the SCDO (not descendant State Objects) when creating and using a State Class.

The first task is to make sure that all the State Objects are of the same object size. We still want to be able to add fields (and therefore *size*) at all levels of the hierarchy, but what we can do is "over-size" all the objects to some arbitrary (and equal) value, provided the objects are dynamically instantiated (i.e., made via a call to the `New` procedure).

The first word of the VMT contains the object instance size, the second word is the negative size. We just change these values as follows:

```
type
 TwoInts = record
  first,last : integer;
 end;
...
{set reserved size of this object type:
 adjust by 2 bytes to allow for the hidden
 VMT offset field that each instance has}

TwoInts(typeof(TObjectType)^).first
 := reservedSize+2;
TwoInts(typeof(TObjectType)^).last
 := -reservedSize-2;
```

This shouldn't be done whilst an object instance exists, since it affects the amount of memory freed up when the instance is disposed.

We've developed four requirements for State Objects in a State Class:

1. They must have a VMT (and hence at least one virtual method).
2. The SCDO should define all the methods representing capabilities expected in any descendant, and they must be virtual.
3. Instances must be the same size.
4. They must be dynamically allocated (not static) so that they can be recast to a common size at will.

The State Object Register (SOR)

As part of the setup code for an application using a State Class, we need to "register" all the State Objects. Registration involves test-instantiating the object, recording the VMT offset (which can be then used to control morphing), and then destroying the test instance.

A beneficial side-effect of the registration process is that it ensures that every State Object is referenced in an application at least once, and hence forces the compiler to generate an associated VMT. Remember that the SCDO may in fact be the only object in the State Class explicitly

referenced in the "guts" of an application, even if every possible state is used at one time or another.

Registration can all be done quite simply, shown here coupled with a call to a generic procedure which sets the object's reserved size:

```
var o : PData;
...
o:=new(PData,RegisterState(SOR));
if o<>nil then begin
 dispose(o,done);
 SetReservedSize(typeof(TData),
   cTDataReservedSize);
end;
```

`RegisterState` is actually a constructor defined in the SCDO. Its only task is to record the unique VMT offset for each State Object. It may also make sure that the declared size of the object fits within the reserved size, and if it doesn't, cause the registration to fail (and cause `New` to return a `nil` pointer).

Accessing the VMT field (a word-sized value that contains the VMT offset) is the trickiest proposition in the whole concept of object morphing. Basically, there's no way around manually calculating exactly where the VMT field should be ("Chapter 17 : Objects" in the TP Programmer's Guide will show you how). The simplest way to avoid any problems is to introduce virtual methods in an abstract object that has no fields (e.g., use `TObject` as the ultimate ancestor). This way the VMT field is always at offset 0 in the object record and can be accessed as `word(self)` [or `word(pointer(@self)^)` if you need to overcome the compiler's type checking].

The State Object Register is a stand alone construct which plays no active role in object morphing. It is simply a list (probably a `PCollection`) that contains the VMT offset of all the State Objects, probably along with some kind of description and a unique key to allow the lookup of an appropriate VMT offset. In the example, the SOR is passed as a parameter to the registration constructor - obviously expecting it to append all the object type information.

An interesting point to note is that the VMT offset does not change from run to run, so it is possible for the SOR to be persistent (maybe stored as a `TResource`). *Beware: if you re-compile, or write a new but derivative program, the VMT offsets are likely to change.*

In the final installment, we will explore the morphing process in detail. I'll provide a detailed example, along with code for a `Morph` unit you can use.

***Paul Gallagher** has been a Turbo Pascal fan since version 3. He works in PC/Mac Support for BHP Research - Melbourne Laboratories. Between playing with computers at work and playing with computers at home, he finds some time to make music and beer. You can reach him at PO Box 731, Mt. Waverley, Victoria 3149 Australia, or through internet mail to paulg@resmel.bhp.com.au.*

Image Processing: File and Color Fundamentals

Bob Crawford
Bowling Green, Kentucky

This time out, we do a bit of spade work in preparation for the business of image processing proper. I will begin by making a few simplifying assumptions about the images we will be working with.

Naturally, as far as the screen display of images is concerned, we will be dealing with 256-color modes exclusively. In fact, most of the time, my examples will concentrate on the 320x200 256-color mode of the VGA. I grant you this is a klunky mode with big fat ugly pixels, so that none of our pictures will look all that wonderful. It does, however, have three notable advantages: (1) It is supported by all VGA cards; (2) It minimizes both the storage requirements and the execution times for our examples; and (3) It has a simple linear memory layout scheme, allowing us to quickly blast pictures onto the screen. The processing techniques (which we will get to in an episode or two, I *promise*) easily carry over to higher resolutions.

A Little Color Background

Having decided to go with 256-color modes as our target, a quick review of how VGA colors are determined in these modes is in order. First, VGA colors are not specified directly. Suppose you ask for a VGA pixel to be displayed using color 115. The 115 does not specify an actual color - rather it is an index into a palette where the colors are actually defined. This palette is 256 entries long, with indices in the range 0..255. This rather restricted range means that one byte per pixel is sufficient for the needed index values, but it also means that only 256 colors are available at any one time.

Each palette entry consists of 3 bytes - one byte each for the red, green, and blue levels. However, only 6 bits out of each byte are actually used, so we are really talking about 18-bit color with a total of 64 x 64 x 64 = 262,144 possible colors from which the 256 active colors can be chosen. The latest version of the `Util` unit (included on this issue's Newsletter Disk) contains routines for manipulating VGA palettes. It also contains routines (independent of the BGI) for getting into and out of the 320x200x256 mode and plotting pixels in that mode.

The 256 colors which I am assuming for the screen display are all well and good, but they are certainly not sufficient for image processing. For that we want to work with 24-bit color, where each pixel is represented by three one-byte quantities, one byte each for red, green, and blue.

The primary file format which we will use for our files uses 24-bit color - the uncompressed Targa format. There are a number of different Targa formats, but the uncompressed version we are concerned with is particularly simple. It consists of an 18-byte header followed by the 24-bit color values for all the pixels (row by row). The only truly unusual thing about the Targa format is that each color value is stored in BGR order instead of the more usual RGB order. Targa is the only format I know of that uses BGR. Mercy knows why.

Taking Aim at the Targa

The `TargaU` unit you will find on the Newsletter Disk contains type definitions and routines for dealing with Targa files and their headers. The only fields of any real interest to us in a Targa header are the `ImageWidth` and `ImageHeight` fields which will usually be 320 and 200, respectively, in our examples. The scheme for reading (or writing) a Targa file is quite trivial: Use `ReadTargaHeader` (or `WriteStandardTargaHeader`) to gobble up (or spit out) the header and then read (or write) a 24-bit color value (type `TTargaBGR`) for each pixel.

For our purposes, we could easily do without the header and deal only with the 24-bit color values (at least if we stick to the 320x200 image size). However, the Targa format is not much more complicated than the bare color values, and Targa files are commonly used by many public domain, shareware, and commercial graphics programs, which makes them very convenient. Convenient they are, but they are also large - a 320x200 uncompressed 24-bit Targa file takes 192,018 bytes. (In due course, we will discuss some of the more compact file formats.)

Having a file format established doesn't do us much good if we don't have a way of viewing our files. The problem here is simple - most of us are not fortunate enough to have boards and monitors capable of displaying 24-bit graphics directly. They are still too expensive.

Now, there are a lot of different 24-bit colors - 256 x 256 x 256 = 16,777,216, to be precise. With a true 24-bit color setup, any number of them may be on the screen at once. So what we have to do is take a picture that may contain thousands of 24-bit colors and find a way to display it using only 256 18-bit colors. A rather formidable task if you want anything fancy - like making the display look anything at all like the original picture. The whole process is called *color quantization*, and no matter how carefully it is done, a substantial amount of image quality is lost.

Quality versus Quantization

Quantization is not a primary topic in this series of articles, but we have to pay enough attention to it that we can at least display our images. One very good option is to use one of the public domain or shareware display programs (some of which are available from the TUG/Pro Library). These are generally well-tuned programs, and most all of them will happily gobble Targa files and solve the quantization problem on the fly as they are displaying images. More importantly, they come with drivers tailored for all kinds of video setups, so if you have a super VGA, a Hicolor Dac chip set, or the like, they will take full

advantage of your hardware. If you want to write your own programs to do the quantization, you will have to work a bit harder. I will outline three quite simpleminded approaches here and mention two more sophisticated methods.

Finding the Best Fit

One way to get some kind of a picture on the screen is to take whatever VGA palette happens to be lying about (the default palette, for example) and then each time you are asked to plot a pixel, to search through that palette for the color that comes "closest" to the actual color the pixel should have. I have included a function called `BestFit` in the `Util` unit which does the searching using a least squares approach to measure closeness.

The `DefPal` program implements this display technique for our standard size Targa files. This program, like its buddies `PopPal` and `FixPal` (which I will discuss momentarily) takes a single command-line argument - the name of the Targa file to be displayed. If you run `DefPal` on one of the sample Targa files, you will surely note that it is rather slow and does a pretty lousy job.

One would suppose that the general approach using `BestFit` is probably not too bad (at least if you are not in a hurry) - the main fault as far as the picture quality goes probably lies with the default VGA palette.

Counting Your Colors

Things should go better if the palette is chosen so that it has something to do with the colors actually in the picture. A (rather unpopular) way to come up with such a palette is the popularity algorithm - take the 256 most frequently occurring colors in the picture as the entries in your palette and let the less common colors fend for themselves.

This method is implemented in the `PopPal` program and the `CCount` unit it uses. If you look at `CCount`, you will notice that while the approach is exactly what you would expect (count the number of times each color crops up and take the colors with the biggest counts), we do not attempt to maintain a count for each *possible* 24-bit color because there are simply too many of them. Instead, each byte (red, green, blue) is shifted three bits to the right. This has the effect of identifying neighboring colors in the RGB color cube so that there are only 32 x 32 x 32 = 32,768 different colors to try and keep track of. This simplification certainly causes some color distortion, but since we are in the business of reducing over 16 million colors down to 256, some distortion is inevitable. Even after the reduction, some tricks are necessary to keep our data objects inside the 64K barrier. Try out `PopPal` and you will find it so slow that it makes `DefPal` look like a rabbit. It, too, gives lousy pictures.

Slicing the Cube

Our third simpleminded approach yields (rather surprisingly, I think) better results. It is implemented in `FixPal`, and the idea is to construct a fixed, general-purpose palette and use it for whatever picture happens to come along. The secret to making it work fairly well is to insure that the palette you use contains a wide, representative collection of colors from all parts of the color cube.

The idea behind the particular palette we use is to take the RGB cube and slice it into four equal parts along the blue axis and into eight equal parts along both the red and green axes. We use fewer divisions along the blue axis because the human eye is much less sensitive to variations in blue than to variations in red or green. This partitions the RGB cube into 256 boxes and the centers of these boxes are taken as the palette entries. Since we are building the palette, it is an easy matter to arrange it so the index corresponding to any given 24-bit color can be easily computed. `FixPal` is much faster than either of the other two methods, and it gives better results to boot.

FixPal is much faster than either of the other two methods, and it gives better results to boot.

Developing a Sophisticated Palette

Actually, none of the methods I have mentioned should be used if you are serious about seeing good images. There are two much better approaches in common use - *octtree quantization* and *median cut quantization.*

I used octtree quantization in the `TugRay` ray-tracer that I discussed at the '92 GeTUGether. The basic idea there is to drop the requested colors into an octtree (a kind of eight-way generalization of a binary tree) as they come up, and let them filter down to become leaves. Whenever the tree starts to get too many leaves (more than 256), you coalesce the children of an appropriately chosen node, thus identifying their colors.

Median cut quantization tries to divide the RGB cube into 256 (or fewer) boxes, all of which represent about the same number of nodes in the picture. The palette colors are then weighted averages of the colors inside these boxes. We have a start on the median cut algorithm - `CCount` contains the necessary tools for collecting the color count data that is used to do the subdivisions.

Sooner or later, I suppose the weaknesses inherent in `FixPal` will begin to show up in our images and I will have to take the time to present an implementation of one of these fancier quantization techniques. But I'm going to put that off as long as possible and begin to concentrate on some standard image processing --- starting next time!

Some Parting Thoughts

I would like to close with a few notes about the files included on disk. The three sample Targa files are all ray

traces from the outstanding public-domain ray tracer POV (Persistence of Vision).

The most interesting of these is F320x200.TGA, which is a rendering of a fish jumping out of a pond. Of course, even this terrific picture doesn't look all that great in such low resolution. I would encourage you to get your hands on POV and make a rendering at the highest resolution that your hardware will support (at 800x600x32K, it is a real treat!).

Finally, there are three files with the Q extension. These are in what I call the "quick" format, designed specifically for 320x200x256 mode images. These files contain a 256 color palette at the beginning, followed by one byte (the color index) for each pixel. They can be displayed (very quickly) by the `ShowQ` program.

***Bob Crawford** is a professor in the Computer Sciences Department at Western Kentucky University, and a contributing editor of TUG Lines on a variety of subjects. He can be reached on CompuServe at 71121,777.*

From the Coding Pad

Gathering and Scattering

Breck Carter
3QC, Inc
Toronto, Ontario Canada

When I first started programming for the PC, programmers all tried to avoid disk I/O. Diskette drives were slowwww, and the first hard drives weren't exactly speed demons. If you wanted to keep up with the keyboard you had to load data into RAM.

Old lessons are hard to unlearn. Today's hard drives are blindingly fast... not as fast as RAM, but certainly faster than a typist. At the same time, programs are getting larger. If you're like me, you're still bumping up against the 640K barrier and finding it harder and harder to stay within it. It's no longer possible to load vast amounts of data into program memory; it's hard enough to get your *code* to fit.

Today you're almost *forced* to use disk I/O to read and re-read program data. The good news is that memory is cheap, and it can be used to build huge caches and RAM drives in extended memory.

The Power of Untyped Files

This article talks about extending your program's memory with Turbo Pascal's untyped file I/O. Included is a pair of utility programs (`Gather` and `Scatter`) that can be used to populate a RAM drive much faster than XCOPY.

Untyped file I/O is often overlooked because it's a little scary. A `BlockWrite` is like a `GoTo`: fast, powerful, unstructured and inherently unsafe. Unlike the `GoTo`, however, untyped I/O won't destroy the readability of your program when used properly.

Untyped I/O is fast, no question. So little Pascal overhead is involved, there's no advantage to coding it in Assembler. Unlike a text file `Write`, a `BlockWrite` does no data conversions. This is both an advantage (it's faster) and a disadvantage: If you `BlockWrite` an integer to a file and then try to print it, all you'll see is two bytes of garbage. The `BlockWrite` doesn't convert the integer to a printable string.

Untyped I/O is powerful because you can read and write anything you want. Consider the following ways you can write a string:

```
var s : String [ 5 ];

s := 'ABC';
{s contains "xABCgg" where x=length, g=garbage
 bytes}

BlockWrite ( f, s[1], Length(s)  );
{ writes "ABC"  }

BlockWrite ( f, s[1], SizeOf(s)-1 );
{ writes "ABCgg" }

BlockWrite ( f, s,  Length(s)+1 );
{ writes "xABC"  }

BlockWrite ( f, s[0], Length(s)+1 );
{ writes "xABC"  }

BlockWrite ( f, s[0], SizeOf(s)  );
{ writes "xABCgg" }

BlockWrite ( f, s[2], 2      );  { writes "BC"  }
BlockWrite ( f, s[0], 1      );  { writes "x"   }
```

Run the program BLOCKSTR.PAS (found on the Newsletter Disk) to see the results on your screen.

Some of those statements might not make much sense (especially the ones that write the trailing garbage characters), but they serve to make a point: You have absolute control over the location and length of the data to write. This is very useful when dealing with fixed-format files.

With speed and power comes responsibility. Untyped I/O is unstructured (and "unsafe") because no type or

range checking is performed. For example, this statement will clobber memory, because s is only six bytes long:

```
BlockRead ( f, s, 10 );
```

It's up to you to design your files carefully and to make sure the data is written and read in the proper order.

Faster Than a Speeding XCOPY

Opening a file is time-consuming. This is true even if you're using a cache or a RAM drive. It's especially true for small files: it takes longer to open and close them than to read or write the data (unless you're using really slow I/O operations).

When using XCOPY to copy 100 files (for example), 200 separate open and close operations must be executed. If all the files were "gathered" into a single file, and a new program was used to split that file up into its component parts as part of the copy operation, this "scatter" program would do only 101 opens and closes (1 for the single input, and 100 for the output files).

Of course, there are really 202 opens and closes: 101 for the gathering and 101 for the scattering. But if you only need to gather the files once, and scatter often, (loading a RAM drive with many read-only files at boot time, for instance) this method offers a truly great saving.

An archive utility (PKZIP, LHA, etc.) could be used to do the gather/scatter. However, the extra decompression logic makes scattering much slower. A comparison of run times (in seconds) can be seen in Table 1. You can see that `Scatter` is quite a bit faster than the alternatives, but not always: XCOPY is slightly better when copying to a RAM drive subdirectory with no cache present.

Table 1
Scatter vs. Decompression

Copy to RAM Drive Root Directory

From	Scatter	XCOPY	LHA
Local hard drive, no cache	11.8	17.3	23.6
Local hard drive, cache	6.8	12.0	23.6
Network drive	12.5	17.3	23.0

Copy to RAM Drive Subdirectory

From	Scatter	XCOPY	LHA
Local hard drive, no cache	20.4	18.4	27.3
Local hard drive, cache	8.0	13.1	27.2

Notes: *250 files, 866K total, 25 Mhz 386 workstation & server, 200 MB 15 msec. local hard drive, 640 MB server drive, Netware 3.1, 4 Mbit/sec Token Ring. The cache affected only reads, not writes (output is to RAM drive). Times are in seconds, and include program load.*

Table 2
Gather Time Examples

From	Time (sec)
Local hard drive, no cache	57.8
Local hard drive, cache	13.0*
Network drive	39.0

** Using SmartDrive with parameters 2048 512*

Some other interesting observations:

(1) The DOS 5 manual says that the RAMDRIVE.SYS SectorSize default of 512 bytes is "strongly recommended", and they're absolutely right! A SectorSize of 128 will pack the data tighter but will slow down I/O by a factor of 2 or more.

(2) The times for a Network drive are averages; actual times varied wildly (14 to 22 seconds for XCOPY). This may be due to Token Ring and server load and the state of the server cache.

(3) Storing files in the RAM drive root is better than using a subdirectory. In fact, files in a subdirectory can become fragmented even when you start with an empty drive. For years I've been saying "don't put files in the root", now I have to change my tune!

To use the root, you might have to increase the NumEntries parameter (256 here):

```
device=c:\dos\ramdrive.sys 1024 512 256 /e
```

(4) The LHA times for a local hard drive aren't affected by the presence of a cache or network. This might be because the self-extracting .EXE was highly compressed (from 866K down to 267K) and was loaded entirely into memory before copying began.

(5) The Windows 3.1 SMARTDRV.EXE is a vast improvement over earlier versions. It's faster, it runs from AUTOEXEC.BAT, it can be modified after boot, it can make use of more memory, it caches writes, it's *wonderful*! Table 2 shows the times for Gathering.

Under The Hood

In the gathered file each embedded file is represented by three fields:

- 12-character string containing the file name and extension,
- 4-byte longint containing the file length N, and
- n bytes of file data.

The first two fields are called a "header".

Rather than run the Gather program once for each file, a special "name file" is introduced. This is just a text file containing the full PathStr of each file to be gathered. Here's an example:

```
C:\EDIT\SEDIT.IDX
C:\CONTROL\MASTER.DTA
C:\DATA\MENU.DBF
```

This tells Gather to read all three files in one run and produce a gathered file with the following contents:

```
xSEDIT.IDXgggLLLL.................................
................xMASTER.DTAggLLLL.................
.................................................
................xMENU.DBFggggLLLL.................
................................
```

Where:

x	= string length byte
g	= trailing garbage beyond current length of string
LLLL	= longint file length
...	= actual file data

Notice that the gathered file contains only file names and extensions, not the full path to the file. That's because the files are usually written to a single target directory. However, you could extend these programs by requiring both source and target PathStrs be coded in the name file.

Here's the pseudo-code for the Gather and Scatter programs:

```
program gather
 for each name file entry
  write file name, extension and length to the
            gathered file
  copy blocks from named file to gathered file
 end for
end program

program scatter
 while not end of gathered file
  read file name, extension and length from the
           gathered file
  copy blocks from gathered file to named file
 end while
end program
```

Untyped file I/O makes all this possible: reading and writing fixed-format headers (the file names and lengths) and efficiently copying large but varying blocks of data. Here's are some bits of code with comments from the Gather program:

A record and array are used to hold the headers and 32K data blocks:

```
type gath_header_type = record
  file_nameextstr : string [ 12 ];
  file_size       : longint;
  end;

var gath_header   : gath_header_type;

var gath_file_buf : array [ 0 .. 32767 ] of byte;
```

The files are opened with RecSize equal to one byte. This is the most useful value; it gives you the freedom to read and write varying amounts and doesn't impose any performance penalty:

```
reset  ( named_file, 1 );
rewrite ( gath_file, 1 );
```

For each input file, first the header is written, then the data is copied in chunks of 32K (or less, for the last chunk):

```
blockwrite ( gath_file, gath_header,
  sizeof ( gath_header ) );

while not eof ( named_file ) do begin

  blockread ( named_file, gath_file_buf,
    sizeof ( gath_file_buf ), blockread_ct );

  blockwrite ( gath_file, gath_file_buf,
    blockread_ct );

  end;
```

The code for Scatter is very similar. An execution profile showed that opening and closing files took 54% of the time, while the reads and writes took only 35%. And that's why this Scatter is faster than XCOPY... it only does half the number of opens and closes. (The complete source for GATHER1.PAS and SCATTER1.PAS can be found on the Newsletter Disk).

Back to the Future

If you're under 30, you might think "Coding Pad" is some kind of 60's home for hackers. So be it... I'd love to be 30 again.

Ancient mainframers like me remember when dumb terminals were only a dream. WE had to write our code on pads of paper and send the sheets off to Keypunch. PRINT our code, actually, with a pencil: (*n.; archaic instrument for writing by hand).* The truly confident would code in pen, but I always ended the day with little pink eraser bits all over my desk, my lap, everything.

Then we waited, and waited, and finally got our punched cards only to pass them on to the Computer Room, where they kept the 256K mainframe (yes, I remember a million dollar upgrade from 256K to 512K, all the power of an early XT!). And wait, and wait again, sometimes overnight, for the dreaded Compile Error. Two turnarounds a day was a luxury granted only to the most critical projects.

In memory of those bygone days of dropped decks and countless elastic band fights I've chosen "From The Coding Pad" as the name of this column. Well, maybe it'll be a column - if you like it. I plan to talk about real-world programming problems and their solutions. I'll stick to Turbo Pascal for DOS for a while, but expect to see some Turbo Pascal for Windows as soon as I get proficient enough.

And that brings me to an introduction of sorts... "If it works for me, it'll work for you". I'm no programming genius, so you won't be getting dark analytical stuff. I've

been a journeyman programmer for twenty years at several large companies: Crown Life, Amdahl, Air Canada and Confederation Life. All through it I've tried to produce code that's clear and easy to understand, and I'll try to share some of my discoveries.

In my next column I'll spruce up the `Gather` and `Scatter` programs with some dynamic memory allocation that doesn't use the heap. If you've ever wanted to allocate huge chunks of memory, then free it up so you can `Exec` another program, you'll find this technique quite useful.

***Breck Carter** is President of 3QC Inc., makers of the HotSpotter 6 execution profiler for Turbo Pascal. He's currently working on contract at Confederation Life as Technical Architect on a large micro/mainframe Cooperative Processing project using Clipper, COBOL and DB2. You can reach him on CompuServe at 71230,3605 or call 416/763-5200.*

Taking Up a Collection

Don Taylor
Poulsbo, Washington

Using the Turbo Vision package included with Turbo Pascal 6.0 and Borland Pascal 7.0 is a little like eating a tomato: either you swallow it whole, or the process can get pretty messy.

But there is another way to conquer a tomato, by slicing off manageable chunks that can be consumed and digested more easily. That's the approach I'll take in the next two articles, where we'll take a look at two capabilities of Turbo Vision that can profitably be used apart from TV itself: collections and streams. So even if you have no earthly intention of ever using Turbo Vision - read on!

The `TCollection` is basically an update to the linked list object introduced in Turbo Pascal 5.5. It provides a framework which greatly simplifies the building and maintaining of doubly-linked lists.

Why use linked lists? Because under the right circumstances, they enable you to make the best use of your program's memory resources. In situations where you know in advance how much data will be kept by a program, that may not be an advantage - you can just create fixed-size arrays or grab a chunk of the Heap for your use. But if you truly don't know how much data the program will have to manage, there is an advantage to linked lists. In this case, a piece of the Heap is taken for each data item, plus a four-byte pointer is created to give you access to the data. By collecting the pointers in a list, you can use them to read, add or delete data.

Of course, having the overhead of the pointer (not to mention the code required to manipulate the list) means there is a trade-off in using linked lists. But when you're looking for maximum flexibility, they can't be beat. That's why frameworks like TV and OWL, and environments like MS Windows make such heavy use of them.

At one time or another, just about every one of us has written a set of linked list routines. They're always a bother, and they're no fun to test and troubleshoot, either. Each time a new situation arises, you have to dust off the last linked list code you wrote and modify it to work with the new data you're working with.

But here is where Borland has done us a real favor. Probably the real reason they chucked the original linked list routines was because the originals didn't offer the degree of flexibility Borland needed for complex projects like Turbo Vision (and thereby the Integrated Development Environment). The entire operation of TV is a "smoke and mirrors" atmosphere where nearly everything you see (and many things you don't) are members of lists. So they created a system for handling pointers that is so flexible, it can be used for virtually anything you can think of - pointers to data records, objects, other lists, cats, dogs, rainbows - well, you get the idea.

In addition, the Savants of Scotts Valley added some features that make the list-manipulation objects (called `TCollections`) take on an almost array-like capability. Instead of having to transverse a list to locate a particular item, you can specify the index of the item, in random-access fashion; the `TCollection` methods will transparently take care of the transversing for you.

Methods are provided for searching for a certain item, and for processing every item in the collection. You can easily insert or delete an item at any point. You can delete items but hold their places. You can release the memory held by individual members of the collection. And you can pack the collection to eliminate nil pointers.

The `TCollection` object's magic is due to the fact that it manipulates generalized pointers. It was created to manipulate objects (windows, frames, menus and such), and it assumes that whatever it is handling is a descendant of Turbo Vision's `TObject` class. As long as that is true, a `TCollection` doesn't ever need to know any more about what its pointers are connected to.

But it is possible to use `TCollections` with objects which are not descendants of `TObject`. In fact, you can use any data type with `TCollections`.

To do so, you need to instruct your `TCollection` in three areas:

- How to dispose of the item, so the `TCollection` can delete an item from the list and recover the Heap space it occupied;
- How to write an individual data item to a `TStream`; and
- How to get an individual item back from a stream.

We'll cover more about streams in the next article, so this time we'll limit ourselves to setting up an example TCollection descendant and telling it what it needs to know to delete a member. But first, a trip down

A Little Side Road

Have you ever had the experience of installing a Windows application, only to watch it madly copy, modify, rename and maybe even delete files in your Windows and root directories?

If so, you'll probably be able to identify with the example utility I'm about to propose. I call it Wince, because I think it's some unwritten law that every Windows program has to begin with "Win", and *wincing* is what I do every time I see a program having its way with my hard disk.

Now suppose we could run Wince before the installation, and it would make an exact "snapshot" of the files in three places - the root directory, the Windows directory, and the Windows system directory. If we then compare the status after the installation, we could easily determine if files had been added, deleted or modified in any of those critical areas.

To make the snapshot, Wince would simply read the information in those three directories, and store them out to disk. When we want to make the comparison, we could load the snapshot back in, compare it to the new status quo, and report any differences.

We would want to save groups of records that contain a file name and the timestamp. Something like

```
tStr12 = String[12];
pFStatusRec  = ^tFStatusRec;
tFStatusRec  = record
 FName : tStr12; { file name }
 FDate : Longint { date stamp }
end; { record }
```

Back on Track

Just how many records we'll have is totally dependent on the number of files in the three directories. So here we have a situation where a collection can be used to good advantage. For that reason, we've provided a pointer to the data record declaration; TCollections deal with pointers exclusively.

Let's assume that we're going to put the data records from all three directories into a single collection. We'll start by pulling in the information from the root directory, but we'll need two index variables - one to point to the collection item containing the first data record from the Windows directory, and one to indicate the first item from the Windows system directory. While we're at it, let's save the system date and time along with the other data, so we can display that date when we print any comparison reports. Our descendant collection's declaration might look something like:

```
uses Objects;
...
pSCollection = ^tSCollection;
tSCollection = object(TCollection)
 WinIdx    : Integer;
 SysIdx    : Integer;
 SavedDate : DateTime;

 constructor Init(ALimit, ADelta : Integer);
 procedure Fill(var OK : Boolean);
 procedure CompareNew(OldF : tSCollection);
 procedure CompareDeleted(NewF : tSCollection);
 procedure CompareChanged(OldF : tSCollection);
 procedure FreeItem(Item : Pointer); virtual;
 function GetItem(var S : TStream) :
           Pointer; virtual;
 procedure PutItem(var S : TStream; Item :
           Pointer); virtual;
 constructor Load(var S : TStream);
 procedure Store(var S : TStream); virtual;
end; { object }
```

Notice that we need to use Turbo Vision's Objects unit, so we have the methods available to support TCollections. The variables WinIdx, SysIdx and SavedDate are self-explanatory. We've included a few methods to give our collection the firepower it needs.

Init, of course, constructs the object. The four workhorses are Fill (which reads in the data from the three directories), and three methods that compare the snapshot against the current state of the directory contents, looking for newly added files, files that were deleted, or files whose timestamp has changed.

The Load, Store, GetItem and PutItem methods all have to do with streams, so I'll save the discussion of them for next time. For the moment, let's concentrate on Init and FreeItem. This is where we'll tell our tSCollection how to initialize itself and release the space for one of our records. Here's the code for Init:

```
constructor tSCollection.Init(ALimit, ADelta :
             Integer);
begin
 TCollection.Init(ALimit, ADelta);
 WinIdx    := 0;
 SysIdx    := 0;
 FillChar(SavedDate, SizeOf(SavedDate), 0);
end; { Init }
```

As you can see, we simply inherited the Init constructor from TCollection, and then initialized our three variables. Now here's the code for FreeItem:

```
procedure tSCollection.FreeItem(Item : Pointer);
{ - Redefine "item" so it can be properly
  disposed }
begin
 if Item <> nil then Dispose(pFStatusRec(Item))
end; { FreeItem }
```

All we needed to do was let FreeItem know the size of the item it was to dispose of, and that was accomplished by typecasting the pointer to the item. It's that simple!

As you can see, collections are *powerful* tools. All they take is a little care.

I've included the complete source code for Wince on the Newsletter Disk. Next time, we'll talk about streams, and how you can easily use TStream objects apart from Turbo Vision. You'll find it a powerful way to store information - both objects and non-objects.

The C Scape

Tim Gentry

Let's face it: C++ can get a bit complex at times, especially as one gets past the basics of class usage (notice I didn't say class *design* - that's a whole other can of worms). All of us know that the power of the language and the ability to get close to the machine doesn't come without a corresponding tradeoff. I think that the learning curve is steepest if one is coming from C rather than starting from scratch, because it is so *very* hard to "unlearn" the language usage that you picked up using standard C. Some C++ features are easy to misuse (or *over*-use, as the case may be), and even some of the seemingly everyday features can cause some "interesting" programming problems. For example, given the following (admittedly contrived) class declaration:

```
class Foo
{
  public:
    void Bar( void );
    ...
    void * operator new( size_t Size,
                         BOOL Fixed );
};
```

What happens if, somewhere else in your code, you allocate an instance of `Foo` in the normal way:

```
Foo *tempFoo = new Foo;
```

Should work, right? Well, sorry - but thanks for playing. The compiler will complain bitterly, saying "Could not find a match for 'Foo::operator new(unsigned int)'." What? That's the normal, ever-present form of new, isn't it? Well, yes it is, but defining an `operator new` (of *any* form) inside of a class hides the global definition of `operator new`. What's the solution? We could use the scoping operator to specifically ask for the global new:

```
Foo *tempFoo = ::new Foo;
```

but this seems more like a bad kludge than good usage. *Every time* somebody wants to just allocate a `Foo` object without specifying the "Fixed" parameter they have to remember to scope the operator. Yuck. A better way is to overload the new operator for this class by adding this inline definition to the class declaration:

```
class Foo
{
...
void * operator new( size_t size )
  { return ::new char[size]; }
}
```

Some of you already knew this. Okay, I can get something a bit tougher for you... Suppose you are the programmer for a city zoo. You're tasked to write a program to keep track of critters, with a side effect of keeping a running count of these critters at all times. (All right, I'm stretching things a bit here, but *bear* with me for a minute.) You decide to start things off, logically enough, with a base class `Critter`. Other animal types will be derived from this class.

```
extern class Food;

class Critter
{
  public:
    Critter( void )
      { CritterCount++; }
    ~Critter( void )
      { CritterCount--; }
    long NumberOfCritters( void )
      { return CritterCount; }
    virtual BOOL FeedCritter( Food & grub );
    ...
  private:
    static long CritterCount;
};

long Critter::CritterCount;
```

Now it becomes easy to derive new animal types from the base `Critter` class. For example:

```
class ManEatingShark : public Critter
{
  public:
    ManEatingShark( void )
      { GreatWhites++; }
    ~ManEatingShark( void )
      { GreatWhites--; }
    ...
    virtual BOOL FeedCritter( Food & grub );
    ...
  private:
    static long GreatWhites;
};

long ManEatingShark::GreatWhites;
```

Now I'm not real sure I want to know what the virtual `FeedCritter` function does in this particular class - or what it takes as an argument, for that matter - but I do know that your code is going to be shark bait when you run it. Take a look at the following code that uses these classes and tell me what's wrong.

```
...
  // a shark is born...
Critter *newExhibit = new ManEatingShark;
  // ... and immediately sold to SeaWorld
delete newExhibit;
```

Is your count of `ManEatingSharks` (in the `GreatWhites` variable) correct? No - the `ManEatingShark` destructor (sounds like a B-movie title, doesn't it?) never gets called, so even after you sell your new exhibit to SeaWorld you still have a shark in your books.

The solution here is to make your base class destructor virtual. Now a virtual destructor in C++ is just like any

other virtual function, namely, it will call the destructor in the class to which the pointer (`newExhibit`, in this case) really points. But virtual destructors go one step further: They also continue to call destructors all the way up the inheritance path. This means that not only will the `ManEatingShark` destructor be called, but the `Critter` destructor will be called as well - which was precisely the behavior we wanted in the first place.

Okay - what am I trying to get across here? Simple: Some of the paths through C++ are a bit tangled, and unless your last name is Stroustrup a good road map is much needed. Some of you might think I'm about to suggest Usenet again, but I've found an elemental problem with addressing this type of concern on Usenet: *If you don't know what to ask there's no way to get an answer.* No, this topic needs someone to expose the questions and then answer them.

Until now good books on some of the pitfalls and techniques to avoid them were in very short supply, but Scott Meyers has come up with a good one. His book, Effective C++: 50 Specific Ways to Improve Your Programs and Designs is published by Addison-Wesley and is available for a suggested price of around $27. The price-to-pages ratio is a bit high, but the tips presented seem to be well worth it. If I come across any more good ones I'll let you know.

Administrivia

On a more administrative subject: I can't think of any tactful ways to put this, so I'll be blunt: With Herman Moons' article in this issue we've run out of C and C++ articles for publication here. I'll repeat from previous columns my calls for your articles, but I'll add that this is a somewhat more urgent request than before. The definition of "article" is entirely up to you: If you merely want to share a trick or hack you've discovered - without writing an entire 2-page text - that's just fine. Anything having to do with C or C++ is fine. I'll even be happy to help flesh out an outline with you, but without your assistence we'll be running a bit thin for a while. As always, I can be reached in several ways: by Internet mail as gentry@bcstec.ca.boeing.com, by CompuServe as 70441,2015, and by US Snail care of TUG/Pro at PO Box 1510, Poulso WA, 98370. 'Nuff said.

In this issue, Herman Moons' article "The Secret of a Natural Look" discusses the importance of an intuitive interface to user-defined classes in C++. For those of you who don't take note of author's names and places of residence, Herman has written most ("most," as in "all but one") of the articles that we've presented in the C Scape since our change to the new format, and several articles before that. He's consistently shown himself to be more than willing to share his knowledge, and to put in the large amounts of time and work needed to write full-size articles in the first place. I'd like to personally thank Herman here for all of his fine work, and I hope that his articles have proven useful to you. Very nice work, my friend - Thank You.

***Tim Gentry** is a professional C++ programmer for Boeing Computer Services. When he isn't programming you'll probably find him on his motorcycle, looking for the most twisted route from FOO to BAR.*

Library Notes

This month's library offerings include two disks, one for DOS and one for Windows. The DOS disk contains version 1.1 of VSA256 which, as the name might imply, is a C/C++ shareware library that provides access to the Video Electronics Standards Association (VESA) 2.0 BIOS extension standard for 256-color graphics. The demo included with the library provides an impressive 256-color display with bright, vibrant colors in 640x400, 640x480, 800x600, 1024x768, and 1280x1024 display modes. Both text and graphics calls are supported in all of these modes. Also included is EVENT, a fascinating C++ class that will allow you to gather mouse, scan code, and keyboard messages from a single overloaded function. In effect, this class assists you to create event-driven DOS programs. EVENT has been ported by its author to DOS, OS/2, UNIX, and Windows. The DOS version of EVENT is freeware. The stock number is PT-CPP-DOS-011.

The Windows disk starts off with BITMAPS, a freeware image viewer that can read and write Windows .BMP files (in several different formats), as well as read and display PCX, GIF, and icon files. It can also convert those formats into Windows .BMP files. Full source code is provided. The disk also contains LZSS, a shareware DLL that implements a powerful and easy-to-use file compression library. Note that this only works with whole files, not data chunks. The last package on the disk is PRINTER, a somewhat rough freeware package that I've included as a demonstration of printing both graphics and text under Windows, including the use of the printer control dialogs. This does not use the OWL printing classes, and could be an emerald in the rough - a little hacking and digging here could yield a gem. The stock number for this disk is UT-CPP-WIN-028.

On to something that a lot of you have been waiting for - and I have good news for you. Stu Phillips, the programmer who ported RCS 5.5 to DOS, has completed the port of RCS 5.6. The new version of the software has no restraints on filenames at all, and should be completely compatible with the previous version. Finally, there is a truly affordable system for DOS programmers to manage the changes that inevitably find their way into their code during development! The stock number of this disk is UT-MIS-DOS-011.

The Secret of a Natural Look

Herman Moons
Genk, Belgium

If we examine how people solve their problems, we find that many of them use a special-purpose programming language, tailored to the application domain. A mathematician e.g. might use APL to solve his problems. In data processing applications, one might use a database language like SQL.

People use these special-purpose languages because they offer concepts that map directly to the problem at hand. General-purpose languages on the other hand generally don't offer domain-specific programming concepts. The typical solution here is to use function libraries. A functional interface however is not nearly as intuitive as specialized syntax. When we want to add two matrices, we want to write *C = A + B* and not *MatrixSum (A, B, C)*. Fortunately, the C++ class concept provides all the power we need to expand the language with domain-specific types. In this article, I will use complex numbers to illustrate how user-defined types with intuitive interfaces can be implemented.

The Secret of My Success

A C++ class has one major advantage over a C structure: its information hiding capability! Information hiding allows us to conceal implementation details from the user. Among these are the data structures and algorithms used to implement our type. The user only sees the type's interface, i.e. the set of operations that can be performed on the type (also known as the *type protocol*).
The class description is thus divided into two parts:

- a public part, that describes the type protocol
- a private part, that defines the data structures and algorithms[1]

The biggest advantage of this separation of concerns is that we can freely change a type's implementation as long as we comply with the type protocol. Our complex numbers can now be defined as follows:

```
class complex
{
  public:
    complex( double re, double im ); // constructor
    complex add (complex c); // add a complex number
            ...
  private:
    double re;      // real part
    double im;      // imaginary part
};
```

When we later change the representation of a complex number (e.g. using polar coordinates), this change will have no effect on the users of our type.

The constructor (which always has the same name as the corresponding type) is called by C++ to initialize complex numbers when they are created. In our case, we provide the real and imaginary part. With the *add* function, users can tell a complex number object to perform a complex addition.

The *complex* class implements the concept of a complex number, but with a non-intuitive interface. For instance, to add two complex numbers we issue the method call *a.add(b)*. This is radically different from the usual addition syntax.

Back in Business with Operators

A complex number class with a functional interface is not very intuitive. This is the very reason people stick with their domain-specific languages. If we could only give our complex numbers the look and feel of normal numbers, we would be back in business. And again, C++ comes to the rescue. The language lets us redefine the meaning of operators for our own types.

```
class complex
{
  public:
    complex (double re=0.0, double im=0.0);
    complex operator+ (const complex c)
      {
      return complex( (re+c.re), (im,c.im) )
      }
    complex& operator+= (const complex c)
      {
      re += c.re;
      im += c.im;
      return *this;
      }
    ...
  private:
    double re;    // real part
    double im;    // imaginary part
};
```

With operators, operations on complex numbers can be expressed with natural syntax. We can now write constructions like:

```
    complex a = 3;      // 3 + 0j
    complex b (1,2);    // 1 + 2j

    a = a + b;          // a = 4 + 2j
```

Users of our type will be pleased that the normal syntax for addition, subtraction, multiplication, etc. will work as expected. While switching to operators, I introduced some other enhancements as well.

- the complex type constructor is extended with default arguments. Now we can construct complex numbers out of simple numbers, as illustrated in the definition of the variable a.

- we have marked the arguments that will not be changed by a member function with the keyword *const*. It is always a good idea to mark read-only arguments as constant. This allows C++ to check whether they are used correctly in the body of the function. Furthermore, compilers can take advantage of this knowledge to perform additional optimizations.

- Short functions (like **operator+**) are defined *inline*. This means that the compiler will generate inline code (macro-like) instead of making a regular function call.

When redefining operators, it is always a good idea to maintain existing conventions. In this case we make sure that e.g. **operator+** and **operator+=** behave as expected. This means that both *a=a+b* and *a+=b* should give the same result.

With the Help of My Friends...

By now we have defined a class interface with a natural look and feel. Well... almost! One remaining problem is that the defined operators are member functions, and *not* overloaded instances of the global **operator+**. The implication is that the compiler can perform no automatic type conversions when using complex numbers in mixed-type expressions. An example will clarify this:

```
complex c (1,2);

c = c + 5;    // calls complex::operator+

c = 5 + c;    // would call ::operator+
              // Type error, since there is no
              // global operator+ that can
              // take a complex argument
```

Ouch! This is not at all what we want. The illustrated asymmetry will only confuse the users of our type. Fortunately, there is an easy solution to this problem. We just need the help of some friends.

The operators don't need to be members of class complex. What we really need are operations on numbers, whatever their kind. We therefore extend the meaning of the global **operator+** through operator overloading. Continuing with our example, we define an **operator+** that takes two complex arguments.

```
inline complex operator+ (complex c1, complex c2)
{
  return complex (c1.re+c2.re, c1.im+c2.im);
}
```

The new definition of **operator+** needs access to the internals of our complex number class. We can grant it access to the internal representation of complex numbers by making it a friend.

```
class complex
{
  friend complex operator+ (complex, complex);
  ...
}
```

Does this solve our problem with mixed-type expressions? The answer (as you might have guessed) is yes. The statements *c=c+3* and *c=3+c* both invoke the newly defined **operator+**. C++ will perform an implicit type conversion to bring these expressions in correspondence with the newly defined global **operator+**. The integer number *3* will be converted to a complex number with the help of the complex class constructor. In essence this means that the following statements are identical:

```
c = 3 + c;
```

and

```
c = complex(3) + c;
```

Finally we have arrived at a complex number class that we can offer to our users. They have no reasons to complain anymore. Our implementation gives them complex numbers that look and feel like any of the built-in numbers of the language.

Conclusion

C++ offers all the tools needed to extend the language with new types with a natural look and feel. All that is required is careful use of the language's features, s.a. operator overloading and friends. We have illustrated the steps taken during class design with a simple example. The same approach can be used to define other types, s.a. vectors, matrices, sets, strings, ... Only your imagination sets the limits...

[1] In C++ the algorithms are not strictly private. Normally algorithms are available as object code, which is private enough. The exception is the use of inline functions, where the algorithm must be visible.

***Herman Moons** has used C++ since AT&T version 1.0, in 1985. He is a researcher at the Katholieke Universiteit Leuven in Belgium, working on the construction of an object-oriented network operating system. When he's not working he enjoys reading science fiction and fantasy novels.*

Since we cannot know all that is to be known of everything, we ought to know a little about everything.

- *Blaise Pascal (1623-1662)*

Database

Covering the Bases

Dave Chowning

In the last issue I covered the Borland World Tour session held in Vancouver, Canada, where I attended Bill French's sessions on the dBASE for Windows compiler. Bill is vice-president of Global Technologies in Denver, Colorado, but he is perhaps best known as the author of dBrief, a dBASE-oriented add-on to the Brief programmer's editor.

Bill has also written a similar add-on product for the Multi Edit programmer's editor. It's called Evolve, and it is reviewed in this issue by Richard Hansen.

While we were at the World Tour, I was able to speak with Bill French at some length.

Bill explained why he wrote Evolve for Multi Edit. "Programmers wanted a menu-driven, programmable editor for dBASE, but under $300." Now that he has written Evolve, Bill has no formal relationship to American Cybernetics, the company that owns and publishes Multi Edit. "Evolve belongs to them."

Bill started out working in finance, after graduating from Colorado State with an Accounting degree. In addition to co-authoring LapLink and CommTools, Bill also wrote a great utility that many people have loved but didn't know that he wrote: the Graphics Design Center for dGE graphics, version 3.0. The publishers of dGE went to a German programmer for the version 4.0 design center. So you will understand the difference in quality, I returned my copy of 4.0 calling it a downgrade instead of an upgrade. I still use dGE 3.0.

What does the future hold for Bill French? He's currently working on four macro projects for Brief that are not dBASE oriented, and one hypertext project. Bill wants to "find a better way to do things for text editors" and make money at it.

Sounds like a good idea to me.

***Dave Chowning** is a professional database programmer and instructor who resides in Vancouver, British Columbia, Canada. He can be reached at Box 101 Station A, Vancouver, BC Canada V6C 2I8, by phone at 604/736-3200, or on CompuServe at 71672,3501.*

Product Insight

Multi Edit with Evolve

Richard W. Hansen
Minneapolis-St. Paul, Minnesota

As our programs get more and more complex, we need ever more powerful tools to create them. If you are a dBASE programmer, or if you write programs in some dBASE derivative like Clipper, then Evolve for Multi Edit may be just the thing you need. Evolve is an add-on to the Multi Edit programmer's editor published by American Cybernetics. Since many of you will be unfamiliar with Multi Edit I will give you a quick overview.

What is Multi Edit?

Multi Edit is a very flexible and powerful programmer's text editor. Multi Edit includes all the standard features we have come to expect in a good text editor. Features like undo and redo, mouse support, multiple files, macro language, alternate keymaps, compiler and language support, etc. What really sets Multi Edit apart is support for things like the editing of UNIX and binary files, unlimited undo and redo, and multiple configurations per compiler. There are many other nice touches built in like line drawing, a programmer's calculator, an ASCII chart, and current line and changed text highlighting. In addition there are a whole array of add-on products including Evolve, network E-mail, source tagger and browser, and C++ and Turbo Pascal on-line help.

Multi Edit comes in three flavors, Lite, Standard, and Pro. The Lite version does not include the Multi Edit macro language and will not work with Evolve. The Standard and Pro versions include the full macro

language. The Pro version also includes a document module with spell checker, communications module, and all macro source.

It is the powerful macro language that makes Evolve possible. Evolve was written entirely in the Multi Edit macro language, and the full macro source for Evolve is included with the package. Unfortunately, the source is in the old style, prior to version 6.0, and not the new "C" like style.

Installation

Installation of Evolve is a snap and takes about five minutes. A full and automatic install program is included and all you need to do is specify the source and destination directories. The install process will add an EVOLVE directory to your Multi Edit directory and use about half a meg of disk space.

If you don't already have Multi Edit, you may be wondering about its installation. While not quite as easy as Evolve, it is still a lot easier than Microsoft Word for Windows! It uses the same basic install program as Evolve, but it creates a couple directories. Multi Edit creates directories for macros, macro source, help, documentation and utility programs. Depending on your setup and version it will use 1 to 2 megs of drive space.

Since Evolve is written in the older macro style, it needs the old MEMAC.EXE macro compiler. You must make sure that MEMAC is installed prior to installing Evolve. MEMAC is included with Multi Edit 6.0, but is not installed unless you specify it from the Module/Option Installation menu.

A Look at Evolve Features

Evolve includes too many features to cover in this review, so I will list many and then take a longer look at several. Evolve includes support for construct matching, template expansion, displaying database and index structures, commenting, source code browsing, source scanning, object editing, code reformatting, function prototyping, window layout, and source code printing. There is extensive on-line help to aid you in remembering what to do with all those features .

There are several ways features can be accessed: from the ASSIST menu, with hot keys, or by clicking the mouse. Evolve probably has the most complex mouse interface I have ever seen. There are at least six different operations that can be invoked with a click of the mouse, and mouse usage is wholly context sensitive. What you click on decides which Evolve feature is invoked. Click on a database filename and the database structure is displayed. Click on a function name and the function is displayed. If you don't like the mouse, just call up the ASSIST menu to get access to the same features.

Source Code Browsing

Evolve includes several features for browsing your source code. First is a feature called *object editing*. Basically this lets you click on a function name and edit that function (object). Evolve will extract just that function and insert it into a window for editing. When you are done it will reinsert that function, complete with any changes, into the original source code. This can be handy, but it requires a RETURN statement to end each function. However, it does not handle nested returns, so it often will only display a portion of a function, down to the first RETURN.

Along with object editing goes source code browsing and scanning. Browsing will create a list of functions and procedures for one or more of the files you are editing. From the list you can select and view individual routines. Scanning is a lot like browsing, but it is more powerful. You can create a list of every occurrence of a particular procedure, function, and variable, or a list of every function, or a list of every variable in just the current file, etc. The combinations are nearly endless. Once the list is created, you can use it to select and jump to one of the items in the list.

Here is where I had a problem with the design of Evolve. With the browse function you create a list, look at one item then return to the list. With scanning you create a list then jump to the place in the source with the occurrence of the selected item. With object editing, you search for a particular item then edit it in a new window. Seems like a lot of ways to get the same sort of thing done. It can be confusing which function can do what and why. While handy, these features could probably be incorporated into a more unified browse facility.

Creating Source Code

I feel Evolve's best features are those for writing code. One I liked the most was the support for commenting. We all know that good comments are needed, but we hate to put them in. So anything that can make the job easier is a blessing. Evolve can help you create program comment headers, function or procedure comment headers, same line comments, and it can even comment out a block of text. All this with one key and various combinations of block marking (stream, line, or column).

Press Ctrl-F7 at the first line in a file and you get a program header comment block. If you are on the line above a function or procedure you will get a subroutine comment block. Do so on a plain old source line, and a comment marker is placed at a preset column and the cursor moved just beyond, waiting for you to impart your wisdom to future programmers. If you mark a block of code and press Ctrl-F7, the block of code is now inside a comment. It is just as easy to mark a block of code to uncomment it. As you would expect, all the comment

types and headers are user configurable.

Probably the most unusual feature in Evolve is one called CodeWriter. CodeWriter makes it very easy to chop apart and rearrange your code. If you suddenly realize that half of that routine you wrote yesterday really should be made into another function, just mark the offending block and invoke the Code Writer function. You supply a name and parameter list and Evolve uses the marked block to create a new routine complete with a comment header. In place of the old code Evolve will place a call to the new routine. You have complete control over placement and format. As an example,

```
...
M->prodcode = SPACE(4)
M->itemcode = SPACE(2)
M->descript = SPACE(25)
M->supplier = SPACE(20)
M->suptel = SPACE(14)
M->price = 00000.00
...

RETURN
```

becomes

```
...

Init_Product()

...

RETURN


FUNCTION Init_Product()
 M->prodcode = SPACE(4)
 M->itemcode = SPACE(2)
 M->descript = SPACE(25)
 M->supplier = SPACE(20)
 M->suptel = SPACE(14)
 M->price = 00000.00
RETURN
```

If you have ever found yourself bogged down in a project with multiple database files and complex keys, you will probably appreciate the structure viewer. The structure viewer can display either the database record or index structure. The index viewer is a simple display of the index expression. The DBF structure display is where the real power lies.

The DBF structure dialog displays all the fields and their characteristics, including type, size, and decimals. You can insert a comment listing of the structure into your code, print the structure, or save it to a text file. In addition, you can insert a series of statements for loading or storing the structure to or from memory.

There are a couple code creation features left to check out. The first is brace matching and completion (these work with all types of braces such as "(" or "["). Brace matching has two modes, one to display a matching brace and the other is to complete a set of braces. If invoked on a starting brace, it will display the matching one. If invoked at the end of a statement it will complete the braces, including multiples, as needed.

Finally, there is construct completion and template expansion. Template expansion will expand `if`, `for`, etc. statements for you. Construct completion will finish out a construct such as `do while`, `for`, or `if`. Template expansion watches what you have typed and completes it for you if possible. For example, if you type "for " it will expand to :

```
for = 1 to

next
```

Evolve's template expansion is pretty good. In most editors I have to turn template expansion off, because it has a habit of expanding in places I don't want, like comments. Evolve seems to be smart enough to avoid this problem.

Construct completion is a feature I have never seen in an editor. Construct completion allows Evolve to insert closing `endif`, enddo, etc. into your source automatically. As an example, if there is a `do while` statement in your code above the current cursor position, Evolve will insert the closing enddo if requested. This

feature operates strictly by scanning backwards at the current indent level, so you have to keep your code evenly indented.

A final feature to consider is source code output. Evolve has a macro, called REPORT, that allows you to print formatted source code listings. You can include headers, footers, line numbers, page numbers, file date and time, and more.

Evolve Configuration

There are two ways to configure Evolve. The first is from the "ASSIST" menu, which brings up a configuration dialog box. Here you can do everything from setting the dBase dialect to remapping the keyboard. As an alternative, you may directly edit the macro source and compile.

For things like report headings and comment block styles you edit the Evolve configuration file, EC.CFG. You can create this file in every directory you work in, if so desired. In this way you can have different setups for different projects.

Conclusion

This is where I confess I am not really a dBase programmer, so I hope I have given Evolve a fair review. Overall, Evolve seems like a useful tool for dBase development and I have only a couple of reservations. First, it is clear that Evolve was written for Multi Edit prior to version 6.0. It would be nice if the macros were redone in the CMAC macro language. Also, under version 6.0 Evolve displays a couple of quirks, mostly related to the block marking features added. Some operations do not work if persistent blocks are not turned on. And the new feature which lets you click the left mouse button to mark the current word can get in the way of the brace matching feature that also uses the left mouse button. Second, I think that the edit, scan, and browse features could be integrated into a more harmonious whole.

[Note: If you would like to try Multi Edit, you can obtain a functional demo version from the TUG/Pro library (stock number DM-MIS-DOS-002-I5), or you can obtain a copy directly from American Cybernetics. -Ed]

***Richard Hansen** is a full time programmer/consultant in the Minneapolis-Saint Paul area and a beta tester for Multi Edit. Rick is also the author of a TP OOP library for Btrieve and a Btrieve file utility.*

New Products

Software Science has released version 4.0 of TOPAZ, its database and screen management library for Pascal. The new version supports Borland Pascal 7.0, providing a development environment for DOS, DPMI protected mode, and Windows.

TOPAZ provides 550 procedures and functions for database management, reporting, file browsing, text editing, full screen data entry, string manipulation, time and date math, pick lists, moving bar menus, screen display, and printer management. It comes with 700 pages of documentation in two volumes: a User's Guide providing tutorial information, and a Technical Reference that describes all procedures and functions..

TOPAZ 4.0 is available to new customers for $199 during an introductory period, and it comes with the company's 60-day guarantee. For more information, call Software Science at 415/697-0411 or fax them at 415/697-1916.

Calling All Authors!

If you have written an application using C or Pascal with CodeBase, Paradox engine, TOPAZ, or a SQL library; ObjectVision, with your own functions in a DLL; a third party DLL product with ObjectVision; OLE or DDE with Quik Reports and ObjectVision; SQL with dBASE or Paradox files; or code generators or CASE tools for database systems, we want to hear from you! If you're not comfortable writing for publication, we can help you by co-writing the article!

Contact TUG/Pro at the editorial address, or contact the database editor directly at Box 101 Station A, Vancouver, BC Canada V6C 2L8, 604/434-1348.
