TUG Lines 50

Communications of the Programmer Resource Organization **August/September 1992**

Perspective

Have you noticed what's been happening to technical books? TUG/Pro is on the reviewer's list of several of the technical book publishers, so I have the privilege of getting an early look at some of the new books being published. From first hand experience, I can tell you the number of books is increasing at an accelerated pace. There's seldom a week goes by that we don't receive at least one or two books - and that's just the tip of the iceberg.

I suppose one could chock up this acceleration to the information revolution in general. I recently heard that the amount of information in the world doubles *every four years*. Assuming it's true, that's a staggering statistic! Unfortunately, just because information is doubling doesn't imply that the amount of *quality* information is changing. I've seen an increasing number of "me too" books on programming - books that attempt to rewrite the user manual for a product.

But it isn't just the quantity of books that are appearing, or the marginal quality of many of them - the price of all books has been steadily increasing. Over the past couple of years PC-oriented technical books have increased from the area of $15-20 to $20-$25. Book publishers used to print a card inside the book, enabling you to order routines disks directly from the author. In the last year, we've seen a barrage of books published with disks glued right inside. It would seem some publishers have discovered that if they can include disks with their books, they can move the retail price of their books into the $30-$40 range, and sometimes higher.

Some publishers have gone a little overboard. IDG Books publishes a wonderful book titled Windows 3.1 Secrets, written by Brian Livingston. For $39.95, you get nearly 1,000 pages that contain truly useful information, plus two disks crammed with software. The software is Windows shareware, which costs the publisher virtually nothing to distribute. IDG prints the DOC files from the shareware as part of the book, which accounts for 150 of those 1,000 pages. When I recently got the book and software for the Win 3.1 edition of the book, I had the opportunity to contact one of the shareware authors. It turns out the version of his software being included with the book was well over a year out of date! I picked the book up at a local warehouse store for under $25; at that price it's a pretty good buy.

In the last Perspective I talked about the "real" retail price of software. Since that writing, I've received two more steep discount offers. One is offering PackRat 4.1 (retail price $395) for $99. The other is for Kedwell's DataBoss (retail $599), also for $99. Last week John Milner was able to purchase a collection of graphics manipulation programs (Dr. Halo), with a combined retail price in the several hundred dollar range, for less than 50 bucks. The trend continues.

Okay, let's summarize. The retail price of software and its manuals is approaching that of books which include software. And books which include software are approaching the price of software. Hmm. Could there be a trend here?

Perhaps *more* than one. There was an unconfirmed rumor making the rounds at the time Borland sold its Turbo toolbox products: that Bantam Books bought those rights, with the idea they would sell them not as software, but as books - including the software as a bonus with the book purchase. That was nearly two years ago, and (true or not) the strategy just might work today.

Also, it would seem the average consumer no longer considers computer software to be "magic" - something for which one is willing to pay large sums of money. More

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, including a special discounted rate for GeTUGether, our annual programmer's conference. For more information, request a copy of our current membership prospectus.

Renewals. The renewal date shown on your TUG Lines label tells you the month and the year by which you must renew if you don't want your membership to lapse. For example, a renewal date of "07/94" means you must renew your membership by June 30, 1994, or your membership will come to an end. You will automatically receive a renewal reminder shortly before your membership lapses. (By the way - the label also tells you the issue number of the last TUG Lines you're scheduled to receive.)

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

and more, software is being purchased to accomplish a specific task.

Is that good or bad? Well, all I can say is it's real. Hang around me for an hour, and you'll probably hear me say "People don't want products, they want solutions" at least once. Books and software have classically offered two different types of solutions.

A book is usually a learning-oriented tool. Its purpose is to *inform*, to impart knowledge to its user. A typical application might be a programming book; once the user has gained the knowledge, he or she can write software - perhaps a word processor.

Software, on the other hand, is frequently a hands-on tool. Its purpose is to *perform*, to enable its user to complete mundane tasks with greater speed and accuracy. A typical application might be a word processor; once its user has gained the appropriate skill, he or she might write a book.

Is either of these solutions inherently more valuable than the other? I would guess the answer to that question will depend on the person of whom it is asked. Perhaps the more important question is "which of these two solutions will work best to solve a particular problem?". As time marches on, I expect the purposes of books and software will begin to overlap, and will perhaps reach the point where they become almost indistinguishable, like powerful word processors such as Word for Windows and page design programs like PageMaker.

Where will the prices of books and software meet? Perhaps near the upper end of the gap between them. But what is the true retail price of any solution? You've got it - whatever the customer is willing to pay for it.

Don Taylor

Global

Product News

Warner Special Products has made available the master recordings for more than 50 prestige recording artists' musical performances for use in CD-ROM applications. In the past, high licensing rates have hampered software developers from creating applications that make use of musical performances by recording artists. Developers can now enjoy special arrangements which can net them up to 30 seconds of a performance for only $300.

All recordings available under the new licensing agreement are by major artists represented by the Time Warner Music Group. Artists include Kenny Rogers, Linda Ronstadt, Randy Newman, Travis Tritt, The Doobie Brothers, Joni Mitchell, James Taylor and many more. For more information on licensing for your multimedia project, contact Tami Vartanian at Warner Special Products, 111 North Hollywood Way, Burbank CA 91505, 818/569-0500.

Hackensack has released **INTRCPT**, a memory resident interrupt trapper and debugger. The product operates with both DOS and Windows, and includes an integrated memory map, vector map, and user-maintained function database. Interrupts may be monitored, trapped, called, and logged at the interrupt, function, or subfunction level. This provides more than 16 million possible breakpoints, which may be set individually or in combination. Logged interrupts are written to a disk file with all register and buffer values shown at the call and return.

The debugger load and execute option provides the ability to run a disassembler for real-time disassembly of the currently executing target application. INTRCPT includes special provisions for NetWare, Btrieve, and NetBIOS calls and also logs Win 3.1 enhanced mode interrupts. INTRCPT costs $99.00, and is available from Hackensack, 6905 Silber Road, Suite 114, Arlington, TX 76017, 800/325-4225.

Help Wanted

Programmer/Analyst with applied experience in both Turbo Pascal for Windows and Turbo Vision, to lead conversion effort (TP/Turbo Vision or TPW). Must have working knowledge of Bible study/research tools. This is a full-time, permanent position. Contact Kirt Willis at 206/870-1561, or write to: BIBLESOFT, 22014 - 7th Avenue South, Seattle, WA 98198.

Library Notes

Newsletter Disk *50* is the companion disk for this issue of TUG Lines. It contains the routines described in this issue. If you have joined TUG/Pro or renewed your membership since January 1, 1992, you should find this disk packed in the envelope with your newsletter. It can also be ordered separately. *Stock number NL-MIS-DOS-050.*

The DeskPop Set is a combination of a handy utility and a patch for it. **DeskPop** is a Sidekick Plus - like TSR that offers a number of useful tools for the programmer, including an excellent calculator that has most of the bit manipulation functions like SHL and SHR, plus full calculator functions in decimal, hex and binary. **DP4Prt** is a group of source files that have been modified to give DeskPop the use of Com ports 1 through 4. DeskPop is furnished in executable form only; to recompile it will require Object Professional version 1.01f or later. *Stock number UT-MIS-DOS-016.*

Custom Controls Disk *1* contains **CTLPCT**, a DLL written in OWL that is used to display a 256-color bitmap in a custom control, complete with some nice examples. **CTLDTL** is a custom control that implements the stereotypical "digital display" look inside a rectangular region. CTLBAR is a bar chart custom control, and DLLMIN, an absolute minimum DLL example. *Stock number OW-WIN-CPP-002. [Note: For a longer description, see "Library Notes" in the CScape section.]*

LVS C++ is a powerful DOS-mode VGA-text windowing package, complete with its own VGA font. It features mouse support; 28, 43, and 50 line display support; push-buttons, radio-buttons, and check-boxes; data entry support; and special support for the Paradox engine. *Stock number UT-CPP-DOS-015. [For a more complete description see "Library Notes" in the CScape section.)*

A Blast from the Past - From CP/M and Almost DOS to DOS. **22Disk** enables you to convert, format, and manipulate diskettes in over 150 CP/M 2.2 formats (340 in the registered version) to and from DOS files. **22Nice** is a companion product to 22Disk that emulates the CP/M 2.2 operating system and permits most CP/M programs to run under DOS. Once installed, the DOS-CP/M integration is seamless. **RainDOS** is a DEC Rainbow driver that lets you read your diskettes, transfer files, and even format new diskettes. Need to have HP 150 files on a PC-compatible? **HP150** is set of drivers that will do it for you. *Stock number UT-MIS-DOS-019.*

Lecture (version 1.0) by Barn Owl Software is text-based shareware that will enable anyone to create a professional-looking presentation under Windows in minutes. Simply write your presentation with any ASCII

text editor (Notepad, for instance). Add the Lecture commands to your file, and you will have a complete presentation, controlling fonts, colors and the size of text. You can change "slides" with the click of a mouse, so you can use the mouse like a remote control on a slide projector. With this program, a notebook PC and an LCD display panel, you can create great-looking presentations anywhere - even in the back seat of a cab, on the way to a meeting. *Stock number UT-MIS-WIN-018.*

Six-Pack for Windows by Beacon Hill Software is a combination of useful shareware utilities for anyone running Microsoft Windows. **Red Button** lets you exit Windows quickly. It runs as an icon that looks like a red button, and when you double-click on it, you will leave Windows. **Launcher** is an application launcher that allows you to easily traverse directories to find an application and run it. **Runner** mimics the Program Manager's File/Run dialog, but since it isn't a dialog, you can have Runner available at any time. **Walker** is a minimalist version of Runner - an edit box that accepts an application and its path; when you hit Return, you run the app. **Payoff** is (sigh) *yet another mortgage calculator.* But **SnapShot** is a program that will capture an entire Windows screen and save it to disk. Captured images are available to other WinApps via the clipboard. *Stock number UT-MIS-WIN-017.*

PC Techniques Disk *15* contains the source code for routines published in Volume 3, Number 3 (the August/September 1992 issue). *Stock number RS-MIS-MIS-814.*

The F-Prot Virus Protection Utilities Disk has once again been updated. **F-Prot** version 2.06 has proven itself to be one of the most powerful and effective virus detectors and exterminators available at any price. But F-Prot is provided as freeware! It's on all our machines here at TUG/Pro. *Stock number UT-MIS-DOS-008.*

The Multi Edit Demo Disk has been updated for version 6.0 of the product. The MultiEdit demo from American Cybernetics highlights a richly-featured DOS programmer's editor for use with all types of ASCII files. Originally created by a professional programmer for his own use, MultiEdit provides all the power and functionality, while maintaining a clean, intuitive interface that makes it extremely easy to learn and use. This is a fully functional editor - *any size* file can be loaded, edited and saved! Works on up to 100 files simultaneously in tiled, overlapping or linked windows. Edits files with line lengths up to 2,048 characters, and up to 2 billion lines long (which will probably satisfy most requirements). Undo/Redo up to 65,535 operations, language support, mouse support and more. Stock number *DM-MIS-DOS-002.*

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 Insights

Windows Programming Editors Part 2 - Customizing CodeWright

Dave Chowning
Vancouver, BC Canada

There has been a new release of CodeWright since the last issue. After looking at last issue's review, you might wonder what more can you do with a programming editor? In this case Version 2.0 has significant additions (not changes) to the user interface, among them a toolbar (called a ribbon) and toolbox.

The toolbar and toolbox make mouse editing very easy and powerful. Now you can click on an icon to: load, save or print files; cut, copy, paste, and undo editing; search and search again; get language help, change to upper or lower case; and check or highlight matching braces. The editing toolbar could be more practical by moving the Make, Compile and Next Error icons to the toolbox, and moving the Upper and Lower case icons to the toolbar. Icons could be added for Macro record and playback, as well as a special icon to display or remove the toolbox!

Other new features which cannot be described in this limited space include the addition of language highlighting and templates for dBASE and Clipper languages, text drag and drop to copy or move text with a mouse, use of either the Windows Clipboard or multiple scrap buffers (up to 65,535) for cut/paste operations, DOS character line drawing, DDE support, grep or search across both files and buffers, and a powerful, first class spelling checker.

There are still some additions that could be made to the general features and interface of CodeWright. First, the icons could have some colors other than grey (or gray)! I believe the next release or update of CodeWright will see this change. Second, many of the additional icons in the file CWICONS.ZIP could be included in the CodeWright engine, so that they can be displayed and assigned to functions without recompiling the DLL's in C! The matching braces {} and () still reveal a C bias. The concept of braces is for language control structures. Languages already supported by CodeWright in its templates (C, Pascal, PAL, dBASE) use control structures identified by words like IF, WHILE, DO, CASE FOR ... NEXT. It would be helpful if all commonly used braces could be matched as well as control structures which aids in debugging.

Customization

CodeWright 2.0 can be easily customized and extended through a variety of built in features. If you set the UPDATE INI option in the Systems Options Dialog, then all your changes for keys, macros, and compilers etc. will be saved automatically to the INI or the State file. It is my purpose in this article to walk you through various customization features to give you an idea of what you can do and how to do it. The customization of software includes several concepts:

- creating macros for frequently used keystrokes,
- modifying the interface with colors,
- assigning links to external compilers and programs,
- reassigning frequently used functions to keys,
- editing INI files that contain customization features, and under Windows ...
- creating DLLs in another language to access or extend features in your software.

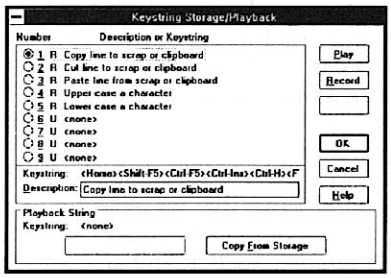
It is very important to note how a software is configured: whether you assign features and functions to keys or whether you assign keys to features and functions. The difference between the two approaches is so great as to cripple an otherwise solid software. Let me explain. I want to assign to a single key a series of functions to mark a line of text and put it in the scrap buffer or the clipboard. This can be done or added to the editor engine if the approach is binding functions to keys. If you are only set up to bind keys to functions, it is very difficult to do my line selection.

Here it is also obvious that no matter how powerful and extensible an editor, it does need an easy to learn, high-level macro language. Such a language should have basic condition and branching controls, with a <pause> command that allows you to move the cursor before playing the next keystroke or function. Although it does not have a macro language at this time (one is in the works), CodeWright does have some powerful and easy to use configuration features.

Creating Macros

Just as dBASE programmers have always loved the DOT Prompt where they could interactively test a line of code, CodeWright users can also test API functions interactively with the F9 key Command Dialog before using them in macros or in assignments to keys.

You can easily record keyboard macros by pressing the F7 key, and by using the F8 key to play back the macro. The PlayBack Strings Dialog on the KEY MENU can save and name up to 9 different macros which can be accessed by this name or description. The 9 macros displayed in



The CodeWright PlayBack Strings Dialog for recording, naming, and saving keystroke macros.

this dialog can be accessed from the F8 or Ctrl-Shift 1-9 keys.

Macros are also automatically saved in the State File: CWRIGHT.PST. Below is a sample from the State File of one of my macros to copy a line into the buffer. The command word `_StateKeyString` has three parameters: the macro number (1-9), the keystrokes as a string, and the description as a string.

```
_StateKeyString=1,
  '<Home><Shift-F5><Ctrl-F5><Ctrl-Ins><Ctrl-H>
<F7>',
  'Copy line to scrap or clipboard'
```

Colors

You can set a number of color options to cover three scopes: current window, model or default window, or for all windows in the editor. From the COLOR Dialog on the OPTIONS MENU, you can set the scope of the color changes as well as the colors. You can interactively see the effects in a display panel. The colors can be saved automatically into the INI file, under the [Colors] section.

Language Support

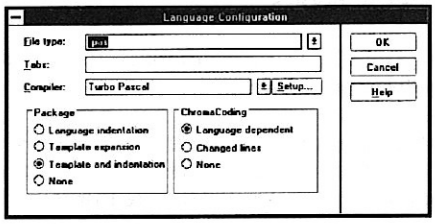
CodeWright provides language support for C, Pascal, PAL, and dBASE. This support is based on the extension of the file being edited, which you can modify for each language you work with in the Language Configuration Dialog, accessed from the UTILITIES OPTIONS MENU. For each set of file extensions, you can assign external compilers etc., code indentation and/or template expansion, and ChromaCoding (color highlight of comments, reserved words, and general code).

Several functions can be used in the INI (Editor section) or a DLL to extend or configure the language support. `ExtAssignTemplate()` can extend the language template editing. `ExtAlias()` can be used to set other file extension like a PAS or PRG or PAL file. `ExtReadKeywordFile()` reads in a list of key words (for Chromacoding) from disk. `ExtAddKeyword()` adds new keywords to the language tables for Chromacoding. You can add the help file from your own compiler so that pressing the F1 or Ctrl-F1 keys with the cursor on a word will bring up language help, which includes the cut and paste feature available from the Turbo IDE. Use the function `CWHelpSDKName` to set the language help to TPW.HLP and use the `CWHelpDefaultName` to make it work. Some sample language configuration lines from my INI file are:

```
ExtAlias=FMT,PRG
ExtReadKeywordFile=PRG, DBASE4.LST
ExtAddKeyword=.PAS,"USES"

CWHelpSDKName="C:\TPW\TPW.HLP"
CWHelpDefaultName="C:\TPW\TPW.HLP"
```

From the Language Configuration Dialog you can go to the Compiler Configuration Dialog to make entries for the compiler, make, build, debug, and other utility programs for your language. You can use '% macros' to set up your command lines for these compiler and utility programs. You can also specify the working directory for your compiler, the error parser, error file name, type of window for execution, and foreground/background execution. These settings are saved in the INI file under the [Compiler] section.



The CodeWright Language Configuration Dialog for setting language files, templates, colors, and compilers.

Reassigning Keys

You can easily reassign keys from the Key Bindings Dialog accessed from the KEY MENU. The dialog allows you to query which key invokes a function, query which function is assigned to a keystroke, and reassign key in the current keymap. You can save your key assignments to the KmapAssign section of the INI with the Save to File Button. You can use the Update Keymap Button to save during this editing session.

The INI File

There are several things you can do with CWRIGHT.INI: assign a default keymap (CUA or Brief), assign new keys to editor functions or dialog boxes (use strings with " marks), change the cursor shape and size (I wanted a larger cursor full across the column), or load a bitmap file into the CodeWright desktop such as my customized DC.BMP. I could have also used any Windows Wallpaper bitmap as well. Remember, since the INI file is an ASCII file, you can edit it once you're more comfortable with the CodeWright commands.

```
DefaultKeymap]
DefaultKeymap=CUA

[KmapAssign]
KmapAssign='<Ctrl-U>', 'Undo'
KmapAssign='<Ctrl-F11>', 'DlgSpellCheck'
```

```
[Editor]
SysCaretHeight=150, CARET_INSERT
SysCaretHeight=150, CARET_INSERT_VIRTUAL
SysCaretHeight=-25, CARET_OVERTYPE
SysCaretHeight=-25, CARET_OVERTYPE_VIRTUAL
SysCaretWidth= 200

SysLoadDesktopBitmap='C:\CWRIGHT\DC.BMP'
```

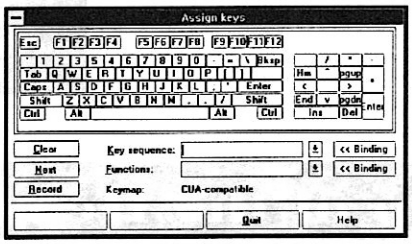
Writing Your Own DLL

You can write your own DLL to change keymap assignments, or modify an existing CUA or Brief DLL. Any language that compiles a DLL can be used. I used Turbo Pascal. The thick Programmer's Reference Manual for CodeWright contains a fantastic wealth of CodeWright API calls that can be accessed from the INI file, a user written DLL, or the F9 command key. The only drawback to this otherwise fine manual is lack of sample code or examples using the functions. I loaded this DLL from the [Editor] section in my INI file:

```
LibPreload=DCKEYS.DLL
```

On this issue's Newsletter Disk look for the full source code of DCKEYS.PAS, my INI and State PST files, and my DC.BMP bitmap file.

CodeWright 2.0 is not only a powerful editor, but an easy to use and easy to modify software. I have not even touched on the fact that you can add your own icons to the editor and attach functions to them, or change the menus, or extend the editor's functions through your own DLL. The developers at Premia Corporation are currently creating a high level macro language based on C, as well as other features being requested by users.



The CodeWright Keybinding Dialog for querying keys, querying functions, and reassigning keys.

[For more information on CodeWright, contact Premia Corporation at Suite 268 - 1075 NW Murray Blvd, Portland, OR 97229, 503/647-9902.]

Dave Chowning is the TUG/Pro Database Editor and gives college lectures to programming classes on programming editors.

Pascal

Top of the Heap

Don Taylor

The article titled "What's All the Fuss About Pointer Arithmetic" in Issue 49 evoked some comments from Al Christians in Lake Oswego, Oregon. Here's what Al had to say:

"Your article on pointer arithmetic seems to have an error. It assumed that a string[10] occupies 10 bytes of memory, which is (of course) off by one, the length byte prefix. It takes 11 bytes.

"Secondly, even though we all do this stuff, it's probably not anything to brag about. Under some operating systems, including several for exploiting the large address space of advanced hardware, pointers don't really point to data, they point to more pointers in tables that the OS maintains. So if you get clever doing pointer math, you're really creating fancy non-portable code.

"There is another hazard also lurking in this area, the fact that the 8086 architecture (which TP has not yet abandoned) gives us about 4,000 different ways to address the same byte. This creates a certain ambiguity about when pointers are equal or unequal. The compiler may insert code to "normalize" pointers for us, to select a specific representation for the pointer to each address as "normal", so that pointers can be compared for equality more rapidly. However, this code is likely to be implementation dependent, so if your program does any comparisons of pointers, it may work or not depending on things beyond your control. See any advanced C text for an idea of the grief this can cause.

"In summary, a large dose of discretion is required when doing pointer math."

Here's Jerry's response:

"It's not clear from Al's letter exactly where in the article (which continued in several files on the Newsletter Disk) he's finding the problem with `string[10]`, but I will assume it is in the code snippet in the second column on page 15. A lot of people have been left with the same erroneous impression due to this undocumented feature of TP6. If would first appear that the statement

```
inc(p,10);
```

would increment the pointer from its previous position (at the length byte prefix of the first string in the array) by 10 bytes, which would place it at the last possible character in that same string - not at the length byte of the next string in the array.

"But this is not what I was demonstrating! The point is that, under TP6, the compiler knows the size of whatever the pointer is pointing to, much like it knows how large a data file record is when asked to do a `Seek`. Here, it knows that `p` is pointing to a `Str10`, which is actually a `string[10]`, which has a length of 11 bytes. The effect of the statement `inc(p,10)` is that it advances the pointer `10 * sizeof(Str10)`, or 110 bytes, which places it at the length byte of the last string in the array. The value of `p^` is then "ten". This may at first seem a bit obtuse, but it does work, and this concept has been a part of the C language since its inception."

"As to Al's other comments, I think they are very well put."

Sliding Around in Turbo Vision [Part 3]

Danny Thorpe
Borland International

In the second part of this series we improved on the behavior of the original TSlider object by overriding its HandleEvent method. We refined our object so as to prevent it being selected when the user clicked on its background view. We then set up a one-way link between the scrollbar and the input line, so changing the position of the slider would automatically update the value in the input line.

In this final installment we'll smooth out the process of getting data entered via the input line, by defining another reusable object.

Choosing Our Own Ancestors

Scan back through the code we have written so far. It strikes me there is an awful lot of string-to-number conversions going on. It sure would be nice if the input line we chose as an ancestor was geared to handle numeric (integer) data instead of string data. That functionality shouldn't be hoarded inside this special `TSliderInputLine`, so we'll make a separate integer input line and "re-descend" our slider input line from it.

A final area that was mentioned in the goals but we have not yet addressed is *dialog data transfer*. Dialog data transfer occurs when you call `SetData` to pass a

record of initializing data to a dialog and the dialog doles out portions of the data to its subviews. (The opposite effect happens with GetData, which is used to get information *from* the dialog subviews.)

The slider has two different sets of data: the slider value (i.e., the scrollbar *value* which equals the input line value) and the slider *range bounds* (i.e., the scrollbar min and max values). The slider value obviously must be transferred by SetData or GetData, but what about the min and max values?

Being able to set the range of the slider via dialog data transfer could be a highly flexible feature, allowing the min and max endpoints to be changed between invocations of a dialog (by modifying the data between dialog invocations) or even while the dialog is active (by calling GetData, modifying the range data, and then calling SetData). While all this is interesting, it's not important that we deal with it right now. Besides, it's generally easier to add to a small object than to try to remove functionality from a large object. It should be a simple task to add slider range data transfer to a descendent of our TSlider, so we'll put range data transfer aside for now. Make a note in your idea journal and move on.

What kind of data should the TSlider exchange in the transfer? String data or integer data? The answer: the slider will be used to get an integer value in a certain range from the user, so the transfer should be made using *integer* data. The fact that the input line uses a string internally for its editing and storage has little to do with how we want the slider to look from the outside.

That having been said, we should be able to kill two birds with one stone here. We'd like to eliminate the frequent string-to-integer conversions in our existing TSliderInputline code, and we'd like to make it transfer integer data instead of string data in dialog data transfers. If we were fortunate enough to have a TLongintInputLine (i.e., an input line dedicated to working with Longints) which took care of these mundane chores for us, we could change the ancestor of TSliderInputLine to this new object, and we would get these changes without complicating the TSliderInputLine object. We would also get a new reusable object for free. We'd have to make some code changes in TSliderInputLine, but most of those would be simplifications, removing the unsightly string conversions and the local variables required for them.

Detour: TLongintInputLine

Let's turn our attention to this TLongintInputLine. Why a long integer? Why not an integer or just a byte? Well, because a longint is the largest integer type, and a TLongIntInputLine can be used to get a byte value or integer value - it's a big container. If we made a TByteInputLine, it would only be good for byte values and we'd have to make another object to hold numbers bigger than the byte range. By choosing longint as the standard internal storage type, we cover all the bases.

The stock TInputLine uses a string Data field to accept and edit user input. We want TLongIntInputLine to maintain a numeric value. Should we inherit from TInputLine, or should we start a new object from scratch? TInputLine contains a lot of editing logic that I do not want to reproduce in a completely new object. So let's figure out how much of the original TInputLine we can keep, and determine what needs to be modified or added to reach our numeric goal.

If we're keeping the editing logic of TInputLine, we also have to keep the Data string field. If we're descending from TInputLine, we can't ignore Data either - if we want the rest of the TInputLine to work. If we add a longint field to TLongintInputLine, our task becomes simply keeping the new numeric field in sync with the inherited string field. The TScrollbar's integer field is called Value, which will make a nice distinction between the internal *string* Data field and the *numeric* data of the new input line. We'll create a longint field called Value in our TLongintInputLine.

What's required to keep Value in sync with the inherited Data field? At the very least, when the object is created, loaded from a stream, and when the GetData method is called to retrieve the object's data, the Data string should be converted into an integer and copied into the Value field. It's probably a good idea also to update the Value when the input line loses focus, and we can use the same SetState technique we used in TSliderInputLine.

Though we stated some informal goals for this TLongintInputLine, let's keep the "featuritis" under control. We could expend a lot of time and effort to create a marvelous TLongintInputLine, but at the moment we don't need a marvelous input line, just one that does what we need it to. Our TLongintInputLine will act as an input line which handles numeric data instead of strings. There are no requirements for range checking, input filtering, formatting control, or any number of other nice features a numeric input line could have. If our TSliderInputLine is careful about what assumptions it makes about its ancestor, we can get by with a minimal TLongintInputLine now and perhaps later drop in a more elaborate variety (by changing TSliderInputLine's ancestor).

Since we're planning to update the Value field in the TLongintInputLine.SetState method, we might as well take the opportunity to open the doors for descendent objects to perform data validation when focus loss occurs. If TLongintInputLine calls its Valid method (which is virtual), all a descendent has to do to validate itself when it loses focus is override or extend its own Valid method. (The default Valid method just returns True, so it will have no effect on how TLongintInputLine behaves.) With TLongintInputLine calling Valid for us, we can remove the modified SetState method from TSliderInputLine. TSliderInputLine will inherit TLongintInputLine.SetState, and when that code

calls Valid, it will actually call TSliderInputLine's Valid method, which contains our special validation code.

Here's another consideration: should TLongintInputLine.SetState update the Value field before or after calling the virtual Valid method? We've assigned to TLongintInputLine the responsibility of keeping Value up to date, so the Value field should be updated before calling Valid. This might dip down into descendants' Valid methods. If Valid was called first, descendants would be given control when the Value field does not match the Data string, and that is a bad thing - the object data would be internally inconsistent, and TLongintInputLine would be blamed for not keeping its data straight!

TInputLine' s Load and Store methods write the Data string out to the stream file, along with some other info (such as the length of the input line window) plus whatever TInputLine's ancestor writes to the stream. Do we need or even want to change the "stream image" of TLongintInputLine to numeric info? To make TLongintInputLine store a longint instead of a string, we'd have to replace all the work that TInputLine.Store does. We can't just call TInputLine.Store and then write out our Value field - that would write a string *and* a longint to the stream! Only one is necessary to save the object's state; the other can be reconstructed at load time from the field that is written to the stream.

What we're really after is to make TSlider participate in the dialog data transfer.

We could create a TLongintInputLine.Store method that calls TInputLine's ancestor's Store method (skipping over TInputLine.Store entirely). It could write the other info that TInputLine.Store writes (but not the string), and write out the Value field as well. A similar arrangement would have to be made in TLongintInputLine.Load. But is it worth all this work?

I don't think so. I'll just let TLongintInputLine inherit the string-writing TInputLine.Store method as-is, and extend the TLongintInputLine.Load constructor to take the newly read Data string and convert it to the Value field. Plain, simple and easy.

Here we are again, with a set of well-defined plans ready to be coded. You'll find the finished code for the TLongintInputLine object in its own unit in the file INTEDIT.PAS on the newsletter disk.

Notice that TLongintInputLine.SetData takes a longint value from the Rec buffer, converts it into a string, and then passes that string to the inherited TInputLine.SetData. This allows TInputLine to update itself, without any second guessing or redundant code in our object.

The SetValue method is provided merely for convenience and type safety. If you pass the wrong type to SetData, the compiler cannot detect an error because var Rec is typeless. But if you pass anything besides one of the integer types to SetValue, the compiler will immediately tell you that you have a type mismatch error.

Note that SetValue and UpdateValue are not virtual methods. They are utility functions that help manipulate the data in this object, and thus are not intended to be overridden or extended in descendent types. Making them static (non-virtual) will save a little time when these methods are called (virtual methods are slower) and will use no memory at runtime (virtual methods use 4 bytes of memory per method per object type).

These changes make a TInputLine look like it handles numeric data. Internally, it doesn't, but an overhead of just under 400 bytes of machine code is enough to enable a convenient illusion.

Putting All This Stuff Together

Let's now apply the new TLongintInputLine to the TSlider object and make the required changes to TSliderInputLine.

First, add the IntEdit unit to the uses clause in the implementation section of the TVSlider unit. (There's no point in putting IntEdit in the interface section of the TVSlider unit, since the TSlider type declaration doesn't need anything from the IntEdit unit.) The file TVSLIDER.PAS on the newsletter disk has been updated version of the TVSlider unit.

TLongintInputLine.SetState now calls Valid for us, so we can remove that method from TSliderInputLine. We can change all the

```
Str(...,S);
SetData(S);
```

statements in TSliderInputLine's Init, HandleEvent, and Valid methods with simply SetData(X), or more appropriately SetValue(X) (so we get type checking on the parameter). Delete the local string variables the deleted statements required and you've cleaned up the code quite a bit.

What we're really after is to make TSlider participate in the dialog data transfer process. TSlider is a group, and a group's SetData and GetData methods simply call the SetData and GetData methods of "participating" subviews. If a view's DataSize method returns a number greater than zero, it means that view wants to participate in GetData and SetData operations. In our slider group, neither TStaticText, TLabels, nor TScrollbar participate in data transfers. A TInputLine will do data transfers, and so will our new TLongintInputLine.

Our TSliderInputLine inherits all that behavior, and it doesn't have any new data to transfer in and out.

But there is one little thing left to do: when data is given to our `TSliderInputLine` (through `SetData`), we should update the scrollbar to the new data value. So we'll need a small extension to the `SetData` method.

This new `SetData` method and the `Valid` method want to update the scrollbar's value. It's mostly a style decision on my part, but I'll make a static method called `UpdateStatusLine` to consolidate the scrollbar updating into one method. It makes the surrounding code easier to read and understand, and it gives you one place to go to change how the slider input line modifies the scrollbar value. And while we're making changes to the `Valid` method, we can remove a lot of complexity the old-style `SetData` required. The new type declaration and the final version of the code for the `TSliderInputLine` object is contained in TVSLIDER.PAS.

Compile and run the test program contained in DEMOSLID.PAS and see what we've got. Quite honestly, you can't see any difference in the slider group - the changes made in this third round of revisions were all internal data-handling operations. But looking at the code, we have a set of objects that are highly modular and modestly efficient in code size, data size, and execution speed.

Sliding into Home

We can look back at this `TSlider` object now and examine how usable and extensible it is. From the usability standpoint, we seem to have scored well with our `TSlider` object. It appears to be well behaved with respect to Turbo Vision norms, it participates in dialog data transfer, and it interacts with the user in an intuitive manner.

We haven't done much direct planning for extensibility, but along the way we made notes for future projects that could stem from the objects created here. We mentioned the relationship between virtual methods and extensibility. Since there are only two `TSlider` methods (`Init` and `HandleEvent`), there is hardly an issue there. `TSlider` has no instance data, but it relies upon the smarts of the components it assembles and orchestrates. This leaves ample opportunity for `TSlider` descendants to construct modified versions of the components `TSlider` uses to achieve different or more complicated behavior.

Ah - but for `TSlider` descendants to be able to make descendants of `TSliderInputLine`, the `TSliderInputLine` type declaration needs to be moved to the interface section of the `TVSlider` unit. It's not mandatory that `TSlider` descendants use `TSliderInputLine` descendants, but `TSliderInputLine` has all the basic functionality needed for the slider to work. Why throw that starting platform away and force future `TSliders` to duplicate our work? *Move the type declaration into the interface of the unit for posterity.*

`TSliderInputLine` has plenty of room for growth, too. Filtering keystrokes to allow only numbers is one extension. Hexadecimal input is another possibility; it would require a slightly different filter and data formatting and conversion routines, but I don't see anything in `TSliderInputLine` that would block such extensions.

Other avenues of growth include creating a `TSlider` variation which has a vertical orientation; a `TSlider` that can configure itself for either vertical or horizontal layout (based on the major axis of the bounding rectangle); a read-only `TSlider`, where perhaps a static text view is used instead of an editable input line; and a version that allows the `min` and `max` range endpoints to be dynamically configured with `SetData` and `GetData`.

In closing, I'd like to point out another aspect of our deliberate programming approach: we never modified the test program, in spite of all the developmental changes that were made to the `TSlider`. When we began work on the `TSliderInputLine`, `TSlider` took some changes to effect the responsibility shift, but after that, `TSlider` didn't change while we made significant revisions to `TSliderInputLine`. We could now continue working on the `TLongintInputLine`, and quite likely very few changes (and perhaps no changes at all) would be required in `TSliderInputLine`. Certainly, no changes would be required in `TSlider` or the test program.

This isn't just a phenomenon of demo programs and academic papers. It's a trait of forward-thinking, object oriented programming that grows in importance and impact with the size of the project.

Object-oriented programming can be used poorly, like any other tool or technique. But when OOP is used well - or even only *moderately well* - the additional benefits so frequently hyped actually *do* seep into your code, your design ideas, and your approaches to problem solving.

Danny Thorpe is a Quality Assurance Engineer at Borland International, responsible for testing the Turbo Pascal compilers, libraries, and object architectures. When he's not writing his own software (or testing someone else's), Danny enjoys soaking up rays on the beach and driving his car. He can be reached on CompuServe at 76646,1035.

On the Make

Jeff Schafer
Silverdale, Washington

Pretty quick
Pretty quick
Programmer Man,
Make me a program
As fast as you can!

Make is a wondrous utility that C programmers have sworn by as long as can be remembered (or at least as long as I can remember). It came before today's wondrous IDEs which know how all the various parts of your source code relate to each other and their environment (and for that matter what the phase of the moon was at compile time). Make was created to solve the simple problem of how to keep all your stuff up to date. And just because Make has classically been used by C programmers doesn't mean you should rule it out if you're programming in any other language. It is a tool for *all* serious Borland language programmers.. It is available for just about every operating system on any computer. *No one should be without it or at least the basic skills of how to use it.*

Why Aren't You Using Make?

One reason you may not already be using Make is because of the simple to use, easy to learn, extremely powerful integrated development environments (IDEs) Borland provides with their languages. Under many circumstances the IDE makes life simple. Another reason may be that you can't always find it, even though it is supplied with all of the language products. With Borland C++ and Turbo C++, you will find out how to use Make in the reference manuals, and you will find the executables in the BIN directory where you would expect them.

Easy enough. But for Turbo Pascal 6.0, instructions for using Make are included in the file UTILS.DOC in the DOC subdirectory. The executable is found in the UTILS subdirectory.

Turbo Pascal for Windows muddies the waters further. You can still find the executable in the UTILS directory, but you will have a hard time finding out how to use it. It is not listed in any of the reference manuals or on-line documentation. To top it all off, they use Make in the DOCDEMOS/HELPEX subdirectory, but they never tell you anywhere else that you need it! Fear not, for there is an unending supply of information about Make. There are numerous magazine articles describing Make's uses on every conceivable computer. In fact, entire books have been written on the subject. And if nothing else, you can get the information from another Borland language.

Why Should I Use this Thing?

There are circumstances where it makes sense to use Make. If you are using the command line compiler, you definitely should use Make. It will help you keep the complexity of the project you are working on at a manageable level (and isn't that one of the most important features of any tool we use?). Make also allows us to switch between different compile configurations with only a single switch that we interpret without complex environment changes (such as the difference between a test version of a program and a production version). Another case for using Make is in mixed language programming. Mixing C, Pascal and Assembly can be simplified if a Makefile helps us keep our source and object relationships in line. Finally, with the advent of multitasking for the masses, it makes sense to execute Makefiles in the background while other work is in progress - boosting your productivity.

Of course, Make isn't always necessary. There are occasions where simple programs are written with a one time purpose. You may choose to use the compiler's IDE. You may even want to use the built in Make capabilities of the command language compiler. Don't forget that the command line compiler has both /B (for build) and /M (for make) flags. In simple cases you may not be able to justify the extra effort of creating a Makefile. However, if you plan to maintain this source - especially if the project is a large one - seriously consider using Make.

How Make Does Its Work

All files are tagged by the operating system with a date and a time. Based on that date/time stamp it should be possible to figure out if a file is obsolete (i.e., out of date). In most cases this is obvious. If the TPU files used in an executable are newer than the executable, the executable is out of date. With logic like that, it didn't take long for programmers to create a utility that would automate this type of date/time dependency checking.

Make works by reading a file (called a Makefile) at execution time and processing its contents. Makefiles can have any name you like, but the default name is MAKEFILE. In the case where the Makefile name is not specified, Make automatically assumes you have named it MAKEFILE (We will discuss the invocation syntax later). Prior to reading in the Makefile, Make will look for a file called BUILTINS.MAK, and (if it exists) process its rules. BUILTINS.MAK is a special Makefile which normally contains those definitions which are common across multiple projects. As you become more familiar with Make you will use BUILTINS.MAK more frequently.

Make is, in a sense, a programming language. It has syntax and semantics. It can give you errors related to both. It is very much like a batch programming language with some innate knowledge of temporal relationships. It can be as simple or as complex as you desire. Make works by comparing the time/date stamps of target files (the files you wish to create) with the source files needed to create the target. If one or more of the source files is newer than the target, the commands to recreate the target

are executed. (Target files which do not exist are considered to be out of date by default.)

To determine how to build the target(s), Makefiles use five basic syntactic components - comments, explicit rules, implicit rules, macros and directives. (These categories may change if you use an implementation of Make from someone other than Borland, but all of the basic pieces are still there). I will describe each of these components in a little more detail. The example Makefile on the Newsletter Disk (TPCALC.MAK) provides examples of all components of a Makefile.

Comments

Comments in a Makefile are just as important as in any program. They help you identify the flow and logic of the Makefile. Comments are denoted by a pound sign (#). All characters following the pound sign on the line are ignored. To make a comment extend across multiple lines, each line must have a pound sign as its first character

Explicit Rules

There are rules and there are rules. As applied to Makefiles, explicit rules are the simplest rules to understand. In each case, a target is listed with the source files necessary to make it. If the target is out of date, the commands listed in the explicit rule will be performed to create the target. It is the first explicit rule in the Makefile which is checked for an up to date target unless a specific target is requested. The syntax of an explicit rule is:

```
TARGET : SOURCE FILES
 < Commands to bring the target
          up to date >.
```

An example of an explicit rule is

```
tcalc.exe : TCalc.pas TCRun.tpu
     tpc TCalc.pas
```

In this example, the target TCALC.EXE depends on TCALC.PAS and TCRUN.TPU. If either of these files has a newer date than TCALC.EXE, the command

TPC TCALC.PAS

will be executed. (Notice that the command is indented.) All target declarations must begin in column 1, and all commands must be indented by at least 1 tab or 1 space. This is how Make determines the starting and ending positions of both targets and commands.

Implicit Rules

Implicit rules are generalizations of explicit rules that we commonly use. As an example, consider how to make a TPU file from its Pascal source file; we need only compile the Pascal file. Implicit rules are based on filename extensions. The basic syntax of an implicit rule is

```
.SourceExtension.DestinationExtension :
 < Commands to Make destination file >
```

An example of an implicit rule is

```
.pas.tpu:
 tpc $<
```

Notice that indentation is again used to distinguish between the dependency and the commands executed to make the target. Notice also the strange combination $<. This is a predefined Macro (more on that in a moment) which expands to the full name of the source file the implicit rule will be used against. (There are several special Macros which will be described later.) Implicit rules will be used in the case where a dependency exists with no explicit rule to make the target, or no commands are given for an explicit rule. An example of this is

```
TCUtil.tpu:     TCUtil.pas TCCOMPAR.obj
```

Here no commands are given for the rule, so that if TCUTIL.PAS is newer than TCUTIL.TPU, the implicit rule shown previously will be invoked. Implicit rules are very powerful and are commonly included in the BUILTINS.MAK file.

Macros

Macros are text expansions performed in the Makefile, not unlike Macros in your favorite word processor. They substitute one string for a symbol within the Makefile. The macro declaration is an assignment statement with the name of the symbol on the left hand side of the equal sign and the string to be substituted on the right. For example

```
TPAS = tpc
```

This declaration states that when the symbol TPAS is used, it should be replaced with the string tpc. The question now arises, how do I tell a *string* "TPAS" from the *symbol* "TPAS"? Simple: to reference the symbol, precede it by a dollar sign and surround it by parentheses. To use the symbol TPAS, for example, you would write

```
$(TPAS) TCalc.pas
```

Here the symbol TPAS would be replaced by its string equivalent during the Make.

Along with Macros you can define, special pre-defined Macros exist for your use in Makefiles. These include (but are not limited to:

1) Base File Name Macro ($*) which is the filename including path, without the extension;

2) Full File Name Macro ($<) which is the full filename including path and extension (This is the target file for explicit rules and the source file for implicit rules);

3) File Name Path Macro ($:) which expands to the path without the filename;

4) File Name and Extension Macro ($.) which is the filename and extension with no path information; and

5) File Name Only Macro ($&) which is the filename with no path or extension.

There is also a test macro to perform conditional tests within your Makefile, and all DOS environment variables are defined as Macros for use inside your Makefile.

One major advantage of Macros is that they can be defined on the command line at Make invocation. This command line definition overrides the internal definition, so you could, for instance, use a different named compiler, set different compiler flags, etc. This is a *very* strong feature of Make.

Directives and Other Stuff

Directives are a special feature of the Borland implementation of Make. They allow conditional execution of sections of commands in your Makefile. I will not discuss this feature, except to say it exists. It can add some strength to your Makefiles, and you should consider it as you become the Make expert on your block.

Want to Make Somethin' of it, Buddy?

To use Make in its simplest form, just type

```
Make
```

and the Make utility will look for a file called MAKEFILE (or MAKEFILE.MAK). It then will start at the first explicit dependency listed in the Makefile and determine if any commands need to be executed. If Make finds there are no out of date files in the first dependency, it will tell you that the target is up to date. Otherwise, it will start checking the chain of dependencies necessary to build the target. (Note that if a target that needs to be built is out of date, Make will search through all dependencies that comprise the target, until it reaches the end of the chain. It will then start executing the commands at this lowest level and build its way back up to the target.)

To make a specific target, invoke Make with the target's name. For example,

```
Make Believe
```

will invoke Make on Makefile, looking for the target Believe. When it finds Believe, it will determine if it is out of date and build it if necessary. To use a file other than one called MAKEFILE, use the -f option with Make. If your custom Makefile is called MYFILE or MYFILE.MAK, you would invoke it with

```
Make -fMyfile
```

In this case Make will search first for a Makefile named MYFILE; if that file is not found, Make will search for MYFILE.MAK. There are many other options associated with Make; you should refer to the Make reference documentation for details on how to use these options.

Making the Most of It

Make is not without its faults. Since Make interprets each of the Makefile commands at runtime, problems can arise. In creation of the Makefile for this example, I renamed the Macro "PAS" to "TPAS" without changing any of my commands. The next time I tried to run my Makefile I got a *big* surprise. My system rebooted! The Make program had attempted to execute a .PAS file, and I saw the results. The system I was running under was a 386-33, so I ran the Makefile on my old 8088 and found a more expected result. It froze up. The moral here is that Make is not without its problems. It is only as smart as we are in our construction of Makefiles.

As we make the transition into multiprocessing with products such as Windows, OS/2, and DOS switchers, Make (or something like it) should be an integral part of any programmer's toolbox. Make provides a simple method to build complex projects. It can be used for compilation, packaging, and date consistency checking. Here's hoping you Make the most of it.

Jeff Schafer is a database systems programmer for Vitro Corporation (by day), and a college instructor in Pascal and database programming (by night). In his "spare" time, Jeff and his wife enjoy hiking and sitting on the tops of mountains.

More Fuss About Pointer Arithmetic

Jerry George
Bainbridge Island, Washington

The article "So What's All The Fuss About Pointer Arithmetic" in Issue 49 could be misleading without an addendum. In that article and the associated example programs, pointers were typecast as `longints` in order to increment pointers or to assign addresses to pointers. The straightforward typecast

```
longint(p);
```

(where p is a pointer) has its use, but it should not be inferred that pointers can be compared by simply typecasting them. When comparing two pointers for a lesser | equal | greater result, a function like the following would be required:

```
function LongAddr( p : pointer) : longint;
begin
  LongAddr := ( longint(seg(p)) shl 4 ) + ofs(p);
end;
```

Assuming `p1` and `p2` are declared pointers, this line would have a reliable result:

```
if LongAddr(p1) > LongAddr(p2)
  then DoThis
  else DoThat;
```

(We are assuming that pointers are normalized-- a subject deserving of a separate article.)

The `LongAddr` function takes the high-order "word" (16-bit expression) from p and shifts it to a 20-bit expression, which happens to be the width of Intel's address bus. Then the low-order word is added. Though `LongAddr` illustrates the concept well, as it stands it is not the most useful function imaginable.

Before tweaking it into something more useful, a recurring bone of contention should be aired:

- `LongAddr` does do something useful
- but it depends on a magic number (i.e., `shl 4`)
- the version of the compiler being used depends on that same number
- therefore, that function should be part of the language.

QED

Turbo Pascal should allow this:

```
if p1 > p2 then DoThis else DoThat;
```

But it always complains: *Operand types do not match operator.*

With misgivings, we can stride forth to turn the concept in `LongAddr` into a compare function. Execution speed is likely to be a critical factor, so that must be considered as well.

A "conversion type" can be borrowed from Turbo Vision:

```
type
  PtrRec = record
    Ofs, Seg: Word;
  end;
```

to speed up the relatively slow `Seg` and `Ofs` functions.

```
function PtrDelta(p1,p2: pointer): longint;
var
  SegOfs1 : PtrRec absolute p1;
  SegOfs2 : PtrRec absolute p2;
begin
  PtrDelta := (longint(SegOfs1.Seg) shl 4
                  + SegOfs1.Ofs)
              -(longint(SegOfs2.Seg) shl 4
                  + SegOfs2.Ofs);
end;
```

With `aDelta` declared a `longint`,

```
aDelta := PtrDelta(p1,p2);
```

is up to 15% faster than

```
aDelta := LongAddr(p1) - LongAddr(p2);
```

Since procedures are inherently faster than functions, this is marginally faster than `PtrDelta`:

```
procedure GetPtrDelta(p1    : pointer;
                      p2    : pointer;
                  VAR Delta: longint);
var
  SegOfs1 : PtrRec absolute p1;
  SegOfs2 : PtrRec absolute p2;
begin
  Delta := (longint(SegOfs1.Seg) shl 4
            + SegOfs1.Ofs)
          -(longint(SegOfs2.Seg) shl 4
            + SegOfs2.Ofs);
end;
```

When dealing with pointers, the advantage of passing by reference versus passing by value becomes a moot point. Surprisingly, when p1 and p2 are VAR parameters, `GetPtrDelta` slows down a bit. It is surprising also that `GetPtrDelta` is less than 1% faster than the function `PtrDelta`.

At GeTUGether '92 Breck Carter asked (and answered) the question "Why profile?" Well, when execution speed is important, the only way to know the speed is to measure it.

`LongAddr` deserves a final tweak:

```
function LongAddr(p: pointer): longint;
var SegOfs : PtrRec absolute p;
begin
  LongAddr := longint(SegOfs.Seg) shl 4
                + SegOfs.Ofs;
end;
```

As you delve into programming with pointers and Pascal, there will inevitably be situations where you will need to compare one pointer with another. Unfortunately, there will also be ample opportunities for big-time errors. The way pointers are implemented in Pascal hasn't eliminated the programmer from having to know implicitly about the hardware a program is running on, and pointers in Turbo Pascal are not yet fully mature - as the need for functions like `LongAddr` and `GetPtrDelta` demonstrates.

Jerry George is a freelance Turbo Pascal programmer with several years' experience with various xBASE language products. When he isn't programming, you're likely to find him connected to a biofeedback machine. You can reach Jerry on CompuServe at 70214,2312.

C/C++

The C Scape

Tim Gentry

Mark this month on your calendar: December 1993. This is the target date for a committee draft release from the ANSI X3J16 C++ standards committee. Of course, the draft of the standard will probably be released later than that (just like the ANSI C standard draft), and a comment from Steve Clamage, the vice chair of the X3J16 committee, bears this out: "I estimate (unofficially) 2 years to a 'draft standard,' after which it goes into the public comment and review cycle. There could be several cycles of about 6 months each before a final standard is approved."

Why do I bring this up? Because I am sincerely looking forward to something that will bring an unruly mob of class libraries into submission. Like many of you I'm trying to avoid re-inventing the wheel, and one of C++'s best weapons in this regard is the class library. With a well-designed and thought-out set of classes programs can be constructed more quickly than before, and with greater portability than straight C code. The problems show up when I need to port my code to, for example, a UNIX system. That system might not (indeed, probably won't) have the same class library available. I would have to either port the entire class library - assuming I have the source - or wrap my application around an entire new library. Needless to say, this isn't A Good Thing.

Before C was standardized we had much the same problem: different compilers on different systems with different dialects of C and different run-time libraries available. X3J11, the people who brought you ANSI C, helped to solve this problem by defining a set of library routines that had to be present in order for a compiler to meet the standard. This immensely aided in coding portable programs. X3J16 has their hands even more full in that they are attempting to define a standard set of classes available to the programmer. This would mean that a String class, for example, would have the same behavior and interface under Borland C++ as it would under Solaris C++, thus providing a common base for use on both platforms.

My only complaint is that it might be two to three years before the standard is approved. The progress of the ANSI C++ standards committee can be followed on the Usenet News Group comp.std.C++.

Mega-Information Source

Those of you with access to the internet have a great resource at your fingertips. It's called Usenet, and can be thought of as a Bulletin Board System (BBS) on steroids. The last time I checked, my system at work carried over 300 different newsgroups (similar to forums, for you CompuServe buffs) out of over 800 newsgroups on every topic from programming languages and operating systems to cooking and childcare. Now my cooking hasn't improved as a result of reading Usenet, but I like to think that my programming *has*, mostly because of groups like comp.lang.c++ and comp.os.windows.programmer.

Usenet is organized into newsgroups, each group centered around a particular topic. Literally tens of thousands of people read these groups on a daily basis, with thousands of them posting new messages - answering or asking questions, bringing up new industry news, and (dare I say it?) gossiping. The programming groups provide a wealth of solid, current information on topics that can't be found anywhere else. The only real problem with Usenet is not letting it take over your entire day...

To gain access to Usenet you either have to have a machine connected to the internet (or a machine [connected to a machine]... connected to the internet), or you have to have access to a public-access internet system. Most of these systems are running UNIX, but I'm sure there are some running other systems. If you're interested in a list of public-access UNIX systems, please let me know either at TUGPro or on CompuServe.

Administrative Notes

Don tells me that I should probably clear up a possible source of confusion regarding the on-going "Call for Articles." We're not really looking for polished pieces on how to program slick applications or yet another treatise on serial port programming. We're looking for articles on *any* topic that professional programmers (like yourself) might find useful. The spelling and grammar can be cleaned up here, so the articles don't have to be polished and in your best prose. TUG/Pro has always been about sharing - so share what you know. And if any of you think you might make an ideal product reviewer let me know about that desire, too.

Why do I keep asking you for articles? Because when we run out of submitted material there won't be anything on C or C++ in the journal. Help save the world: Send in your C and C++ articles (or article ideas) today.

Oops - there's the smoke detector. That means dinner's ready. One final point, though: If there's anybody out there with a source-level debugger for newborn babies, please get in touch with me ASAP...

Tim Gentry *is a professional programmer for Boeing Computer Services. When he isn't programming you'll probably find him tooling about on his motorcycle. He can be reached on Usenet at gentry@bcstec.ca.boeing.com, or on CompuServe at 70441,2015.*

Library Notes

The first of this issue's additions to the library is OW-CPP-WIN-002, a 1.44M disk centered around the area of custom controls in OWL. We start with CTLPCT, a custom control DLL written in OWL that is used to display a 256-color bitmap in a custom control. Besides being a well written example and tool the three bitmaps included with the package are great. The fun continues with CTLDTL, a custom control that implements the stereotypical "digital display" look inside a rectangular region. The last control package is CTLBAR, a bar chart custom control. All of these are smoothly implemented in DLL form, making them easy to use in any of your packages. Speaking of DLL's, the disk also includes DLLMIN, an absolute minimum DLL (and a program to call it) that is an illustration of the functions and calls needed to code and use a Windows DLL.

I'm also adding the shareware LVS C++ windowing package I mentioned in the last issue. This is a powerful DOS-mode VGA-text windowing package, complete with its own VGA font. Those of you who were at GetTUGether this year saw it as part of Mike Burton's session. Those of you who weren't there - well, suffice to say it was impressive. The package features mouse support; 28, 43, and 50 line display support; push-buttons, radio-buttons, and check-boxes; data entry support; and special support for the Paradox engine. The stock number is UT-CPP-DOS-015.

One last note: The GNU RCS/DIFF disk from last issue is proving to be extremely popular. We have found, however, that RCS has one annoying limitation: it generates the name for the delta file by simply appending "_V" to the filename. This means it only works with files with single-character extensions. Sigh. A modification is in the works and we'll publish a patch here shortly.

Generic Classes in C++ (Part 2)

Herman Moons
Genk, Belgium

The first part of this series showed how to implement generic classes with a version 2.0 C++ compiler, using a stack as an example. This turned out to be rather straightforward, requiring only that we write some simple macros. Unfortunately, this macro solution leaves you with the burden to generate the appropriate class declarations and implementations by hand. This is tedious at best, and certainly inelegant. In my particular case it was even worse, because the ADA boys at my department just loved to criticize C++ for this shortcoming. After all, their beloved language has always supported generic packages.

But no longer do we, C++ programmers, have to take these sneering remarks! Finally, the template facility, as described in the ARM[1], has made its way to commercial C++ compilers, like Borland C++ 3.x. With templates, support for generic classes has been fully integrated into the language. Templates open up a whole new programming dimension. At last, it is possible to write truly re-usable software, as I will demonstrate in the remainder of this article.

The Template Solution

Let's take a look again at our stack class. With the templates supported by the Borland C++ 3.x compiler, implementing this class is a piece of cake. You just write your stack as you would for, say a stack of integers, simply substituting the actual type name with a generic one, like `Type`.

```
template <class Type>
class stack
{
   public:
      stack (void)
         { index = 0; }
      virtual void push (Type elem);
      virtual Type pop (void)
         { return tab[--index]; }
   protected:
      Type tab[100];
      int index;
};
```

Apart from the `template` keyword, this looks and feels like a normal class definition. The only thing we have to remember is that the indicated `Type` serves as a place holder for the real types we will be using in our programs. As for normal classes, the member functions of a class template can be provided separately. Of course, these members are themselves generic, and are expressed by means of a function template.

```
template <class Type>
void stack<Type>::push (Type elem)
{
   tab[index++] = elem;
}
```

Once we have the stack template, we can add it to our library. The easiest way to do this is to put the template in a header file, e.g., *stack.h*, which will be included in our programs. We can now use the template whenever stacks provide a solution to our problem.

```
#include <stack.h>

   // define and use a stack of integers
stack<int> intstack;
intstack.push(20);

   // define and use a stack of EmployeeRecords
stack<EmployeeRecord> empstack;
EmployeeRecord erec = empstack.pop();
```

That's all there is to it! We no longer have to worry about instantiating particular kinds of stacks. All this is taken care of by the compiler. Ain't life wonderful!

Control your Templates

The stack template we defined above will work just fine in most situations. But we have made an implicit

assumption! We take for granted that the stack's element type supports an appropriate assignment operation. Appropriate in our case means that objects are copied in and out of the stack. In some cases, this may cause trouble! Let's say that we want to stack strings. Unfortunately, the usual C++ string type, `char*`, has wrong assignment semantics. Using a `stack<char*>` will thus result in a stack that keeps pointers to strings, not the strings themselves. This is not the behaviour we expected!

To solve this problem we must be able to take control of the template instantiation process. What we need is a way to tell the template mechanism that its idea of a stack is wrong where C++ strings are concerned. We do this by providing the correct definitions of the involved member functions ourselves. This is equivalent to a manual instantiation of these functions, except that we can now easily implement the intended semantics. In our example, we override the stack template's idea of the push operation for character strings.

```
   // stack them strings correctly !!!
template <class char*>
void stack<char*>::push (char* elem)
{
   tab[index] = new char [ strlen(elem) + 1 ];
   strcpy (tab[index++], elem);
}
```

This implementation of the push member function pushes the strings themselves on the stack, and not merely character pointers. A look at the template for the pop member function shows us that it has correct semantics. It will pop the string, and turn over responsibility for the string to the stack's user.

As you can see, the template facility is very flexible. It will automatically instantiate the appropriate classes and member functions, but this can be easily overridden by the programmer, if the need arises.

Templates and Friends

Our stack template describes a whole category of stacks. Specific stacks are instantiated from the template whenever they are needed. So far, so good. But what if we want to define a stack's friends? Well, that happens to be easy. Generic classes can have generic friends. A generic friend is simply a function template, from which specific friend functions can be instantiated.

Let's illustrate this with an example. Suppose we want to output our stacks on a stream. This is most easily accomplished through definition of an output operator. Output operators cannot be defined as class member functions though, since they must take an output stream reference as their first parameter. We therefore define the output operator as a generic friend of the stack template.

```
template <class Type>
class  stack
{
   friend ostream& operator << (ostream& os,
                        stack<Type>& st);
   ...
};

template <class Type>
ostream& operator << (ostream& os, stack<Type>& st)
{
   for (int i=0; i<st.index; i++)
      os << st.tab[i] << endl;
   return os;
}
```

For each particular kind of stack, there now exists a corresponding particular kind of output operator. Whenever a stack is instantiated, its friends are instantiated as well (provided of course that their help is needed). So, if we try to output a stack of integers ...

```
stack<int> is;
...
cout << is;
```

the correct output operator is automatically provided. The programmer has not to worry about this instantiation process. Everything is handled by the language!

Ok! But I Need a Safe Stack

By now we have added this neat little stack template to our library. Comes a colleague to you and says: *"I really like that stack thingy you wrote, but in my department we have these ADA boys that we are training on C++, and they just can't use the thing in the right way. They get stack overflows all the time (probably still accustomed to their former gigabyte memory machines ;-)."*

Well, no reason to panic. Think about what you would do with a stack of integers. Well, you would specialize it of course (that's what C++ is all about, right!). Now, if you can do it with classes, there's no reason why it can't be done with templates. So let's specialize the stack template, and turn it into a safe stack. We do this simply by using inheritance in the normal manner:

```
template <class Type>
class SafeStack : public Stack<Type>
{
   public:
      virtual void push (Type elem);
      virtual Type pop (void)
         {
         if ( index == 0 )
            {
            cerr << "Stack underflow" <<  endl;
            exit (1);
            }
         return Stack<Type>::pop();
         }
};

template  <class Type>
void SafeStack<Type>::push (Type elem)
{
   if ( index >= 100 )
      {
      cerr << "Stack overflow" << endl;
      exit (1);
      }
   Stack<Type>::push (elem);
}
```

That's all there is to it! We now have a stack that checks its own boundaries. And thanks to inheritance, we could re-use the old stack code. But can all this really work, you wonder ? After all, we are inheriting from generic types, that are only abstract entities. Well, let's have a look at what happens when we create a `SafeStack` of integers. From the template definition of `SafeStack`, we see that a `SafeStack<int>` is built by

inheriting from a `stack<int>`. Your C++ compiler is smart enough to detect this, and will thus instantiate both the `stack<int>` and the `SafeStack<int>`. So by the time you use a derived class template, all its base classes in the class hierarchy will get instantiated from their own templates too.

A Word of Caution

If you have read this far, you're probably impressed by C++'s template facility (you should be ;-). Not only does it let you capture the algorithms for whole families of classes in a general way, but it also lets you use inheritance to build relationships between such class families. From this inheritance hierarchy of related generic types, you can instantiate those actual class types that you need in your programs.

But you have to remember that the template facility is not a magic wand! It's just a tool, and you, as a programmer, should still know what you're doing. The thing to watch out for is code duplication. Every C++ programmer I know is a realist (not a dreamer), and they all know what's behind fancy words. Templates are a nice mechanism, but if you look at it a little bit closer, you'll notice that it is no more than a fancy macro facility (although a very sophisticated one). When a template is instantiated, the template mechanism just fabricates a new class definition from the template.

Please reread the last sentence! Now consider what happens if we use a `stack<int>`, a `stack<float>` and a `stack<EmployeeRecord>` in our program. Right! the template facility will simply fabricate three different stack classes, each with their own specialized element type. In the case of our stack classes, this doesn't really matter. But for more substantial class templates, the duplication of code may well become intolerable.

Experienced C++ programmers always try to factor out code that is common to all class instances fabricated from a template. If there is commonality, it is wise to use a non-template base class to capture it. This way we will not duplicate the common code, but simply re-use it (thanks, once again, to inheritance!).

I'll illustrate the concept with a small example. Let's say that you have a need for pointer stacks, i.e. stacks that store pointers to objects, instead of the objects themselves. Your first thought is *"Hey, this seems to be an ideal candidate for using templates. Let's do it!"* So, you jump into your editor, and write a class template as follows:

```
template <class Type>
class PtrStack
{
   public:
      PtrStack (void) { index = 0; }
      void push (Type* elem);
      Type* pop (void);
   protected:
      Type* tab[100];
      int index;
};
```

This works of course, but the C++ compiler will fabricate a complete pointer stack for every new element type you specify. Now this is really a waste of memory, since we all know that pointers are simply addresses, and all this type stuff is simply language dressing. So we could just as well express the pointer stack algorithm in terms of void pointers, and use the template facility simply to perform the necessary type casts. In this case, the solution becomes:

```
class BasePtrStack
{
   public:
      BasePtrStack (void) { index = 0; }
      void push (void* elem);
      void* pop (void);
   private:
      void *tab[100];
      int index;
};

template <class Type>
class PtrStack : private BasePtrStack
{
   public:
      void push (Type* elem)
         { BasePtrStack::push(elem); }
      Type* pop (void)
         { return (Type*)BasePtrStack::pop(); }
};
```

When we now fabricate different pointer stacks from the template, they will all share the same implementation code (expressed in terms of void pointers). The template simply performs the necessary type conversions, but won't cause any run-time overheads, nor will it result in a duplication of code. Nevertheless, using the template approach is superior to using a stack of void pointers, since C++ can now perform its strong type checks, making it easier for us to detect errors at compile time.

Roll your own!

I hope that by now you are convinced that templates are really worth looking into. If you are serious about writing re-usable software (and that's the main reason why most of us moved from C to C++), you definitely need templates.

As I demonstrated in this article, writing templates is a lot like writing normal classes. The C++ compiler will do all the hard work for you, fabricating real classes from your templates as the need arises. Remember though, that the template mechanism simply does what you tell it to do. If you tell it to duplicate code, it will happily do so. So be careful, and factor out common code in base classes, whenever you think it is worth the effort.
So, start your computer, jump into the editor, and write some class templates you can re-use in your later projects. Soon, you'll wonder how you ever lived without them.

[1]The ARM is the Annotated C++ Reference Manual. It is the base document for the current ANSI C++ standardization efforts. If you're serious about C++, this book belongs on your shelf.

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.*

Database

Covering the Bases

Dave Chowning

The Culture Clash (Part 2)

The Culture Clash discussed in last issue's column was a manifestation of intense technological change. *A computing revolution.*

A revolution is a change in the basic concepts, methods, morals, and conditions (technology), whereas a coup is a sudden action taken to obtain or retain power. In a coup, the underlying structure does not have to be changed, nor the moral precepts, methods or technology. Just the control. The introduction of the IBM PC and other micro computers was a revolution in both technology and methodology. The introduction of DEC mini computers was a coup, intended to obtain a share of power or market without a fundamental change in methodology or technology.

Today's move to Client Server with object oriented GUI interfaces is a revolution. The resulting culture clash is between traditional mainframe based MIS and newer Client Server based information professionals and desktop programmer/analysts.

Demands of Technological Change

At first glance, it is hard to see why distributed processing with GUI front ends would cause such a change. For years organizations have built their computer systems on the basis of who owned the data, and who had the budget. The usual answer was each department owned its own data, and Finance or Accounting had the budget (or controlled yours).

Today's focus on multi-user, enterprise-wide database systems and application development for downsized Client Server applications is redefining corporate databases. The enterprise owns the data, and everyone has access to that portion which they need. Thus, a centralized budget for systems can be an effective tool instead of a barrier to developing information applications.

Chris Date has said, mainframe COBOL jobs are not going to India, they are simply disappearing. Why? LANs and WANs, Client Server, open architecture, end user computing, downsizing, rightsizing, integration, enterprise wide systems, GUI, desktop development tools. Traditional structure is under attack, because the traditional technology is being replaced. Mainframe use is dropping each year in favor of powerful smaller computers, i.e., Client Server distributed systems.

Dumb terminal technology led to dumb users and dumb programmers. Today we are replacing terminals with PCs, COBOL programmers and 4GL programmers with desktop GUI developers. The emphasis on Windows is development tools, not canned software packages. Even the spreadsheet under Windows becomes a front end for SQL servers. DLLs written in C, C++ and Pascal plug into spreadsheets, report writers, databases, query tools.

Today's IS must contribute to the company's profits. Distributed business decisions need distributed systems. Flexibility and adaptability are important during times of rapid change. Companies need cost effective information systems rather than data processing organizations. To produce information for a business, IS must have both business and analysis skills. IS must become corporate data strategists, shaping the corporate information standards and data structures.

The current recession/depression means there is less cash for new systems, but also that competition intensity has increased demand for new systems. The will to survive is important here, both for programmers and for companies. Change is not an option for survivors. The need is for a new breed of information professional with business, analysis and people skills, who can adapt quickly and use new tools for rapid prototyping and development.

New IS Professionals

Traditional MIS staff must become the new IS Professionals who empower users with tools to find solutions and develop applications at the desktop level. New developers will create productivity, profits and eliminate project backlogs. This makes a leaner IS, where a single PC programmer can perform the duties of project leader to designer to analyst to programmer. No hierarchy of staff with complicated communication. Just one person talking to usually one or two key line staff in a department and conferring with a manager. This creates a new integrated working partnership in companies.

Of course, the new IS professionals still receive direction, standards, and data strategies from IS. What is needed is not only open systems where you can plug in solutions, but open professionals who can be plugged into different responsibilities as the need arises. This requires programming skills in Windows, SQL databases, GUI and OOP, as well as skills in analysis. Obviously the traditional career path is gone in IS.

Revolution or Coup?

How is your company responding to technological change: with growth and flexibility - or is it managing change with a coup? The key is to follow the money in your IS budget. Where and how are human resources assigned, *not* what hardware and software is purchased.

There is a recognizable pattern when traditional MIS tries to maintain control of restructuring and new

distributed processing and Client Server systems, or when IS is moving with business and technological change. The pattern can be seen in layoffs and reassignment of duties.

The way to manage change without shaking up the accumulated MIS culture in an organization is to keep on staff only those whose very jobs or skills do not threaten that culture. This includes keeping the core of operations staff to unpack boxes and install hardware and software. Any programmers who are retained as well as staff with software skills are assigned to support and train users on commercial software packages. Of course supervisory staff and administrative staff need to be kept, since the organization's main job is still pushing paper, managing a budget, and creating a backlog of work to defer more staff cuts.

A coup tries to manage technological change like a recession. The staff that can be sacrificed are middle managers along with programmers and analysts, especially PC programmers with business or office skills. These skills are replaced by outsourcing to expensive contract consulting firms. Occasionally, key people can be hidden in the budget by laying them off and rehiring them on renewable term contracts. In the long run, a coup cannot cut costs without crippling a business's functionality.

On the other hand, adapting to technological change means retaining the key skills a business needs: analysts and application developers (PC programmers), while outsourcing all the support, installation and training jobs. Administrative and clerical staff salaries can then be cut without detracting from departmental efficiency or functionality. This frees up an organization's budget from the 'army' of easily replaceable lower skilled people with their huge overhead in benefits. These skills are easily found in the market place, by contracting individuals for term employment or by outsourcing.

One side of the culture clash attempts to turn PC's into dumb terminals on networks, removing disk drives and hard disks where ever possible, and by turning professionals into dumb staff, who have no business knowledge or programming skills. The remaining staff is divided into artificial levels of positions in the organization that block quick response and communication.

The other side of this culture clash retains the business skills of its department, and its in-house developers who can communicate with and empower end users on the desktop, as well as develop applications that turn data into information.

In an information age with intense technological change, can any business survive where the IS organization has little or no knowledge of the business itself?

This culture clash affects you. How it unfolds in your professional environment will determine whether you have a job. Collectively, it could affect whether your country has an economy. How you respond to technological change and the culture clash will determine whether you have a career.