Getting TEXnical: Insights into TEX Macro Writing Techniques

Amy Hendrickson
TEXnology Inc., 57 Longwood Avenue, Brookline, MA 02146
617-738-8029 Internet: amyh@ai.mit.edu

Abstract

Most of us understand the basic form of TEX macros but that understanding alone is often inadequate when we need to solve certain problems. We need additional insight to be able to develop methods of passing information, moving text with changed catcodes, preserving blank lines, and more. Writing a large macro package brings in a new set of issues: how to avoid bumping into implementation restrictions, e.g., constraints of hash size, string size, and others; how to make a pleasant user interface; how to make your code as concise as possible.

Some of the techniques to be discussed here include making a macro with a variable number of arguments; changing catcodes in macros, defining a macro whose argument is intentionally never used; conserving hash size by using counters instead of newifs; csname techniques and non-outer dynamic allocation; and table making techniques. Finally, some suggestions are included on methods to use when developing new macros.

A Quick Review of Some Important TEX Primitives

Expandafter. `\expandafter`, a TEX primitive often used in this article, affects the timing of macro expansion. Macro expansion is that step in TEX's processing which changes a control sequence to whatever that control sequence is defined to represent. `\expandafter` is usually followed by calls to two macros. It expands the *first* macro following it only after it has expanded the *second*. Thus, `\expandafter` makes it possible for the first macro to process the pieces of the second macro as if the second macro were written out, not represented by a control sequence.

Here is an example: If we define `\letters` and `\lookatletters`,

```
\def\letters{xyz}
\def\lookatletters#1#2#3{First arg=#1,
  Second arg=#2, Third arg=#3 }
```

and follow `\lookatletters` with `\letters ? !`, `\lookatletters` takes the whole definition of `\letters` as the first argument, `?` as the second argument, and `!` as the third. Thus

```
\lookatletters\letters ? !
```

produces

First arg=xyz, Second arg=?, Third arg=!

But if we use `\expandafter`, `\lookatletters` will be able to process the contents of `\letters` for each argument:

```
\expandafter\lookatletters\letters ? !
```

produces

First arg=x, Second arg=y, Third arg=z ? !

String. `\string` is a TEX primitive which causes the control sequence following it to be broken into a list of character tokens in order to print the control sequence or to process it with another macro. `\tt\string\TeX` will produce `\TeX`. (What is the `\tt` doing in there? It makes the backslash print as backslash (\) when it would otherwise print as a quote mark (“). If you are curious about this, look up `\escapechar` in *The TEXbook*.)

Csname. `\csname ...\endcsname` is an alternative way to define and invoke a TEX command. Its function is the inverse of that of `\string`. `\string` takes a control sequence and turns it into tokens; `\csname...\endcsname` takes tokens and turns them into a control sequence.

Commands called for with `\csname` produce the same results as the backslash form (e.g., `\csname TeX\endcsname` and `\TeX` are equivalent) but the `\csname` construction combined with `\expandafter` allows you to build and invoke a control sequence dynamically at the time the file is processed, as opposed to knowing its name at the time the macros are written. This technique has many interesting and useful applications, as we will soon see.

If and ifx. Since both `\if` and `\ifx` are conditionals used to compare tokens, the TEX user may well wonder when to use `\if` and when to use `\ifx`. When we understand how each of these conditionals works, we may conclude that the answer is to use `\if` *only when comparing single tokens* and to use it with care.

How 'if' works. `\if` expands whatever immediately follows it until it arrives at two unexpandable tokens. It then compares them to see if their charcodes match. This test is useful to see if a given letter is upper- or lower-case and in some other instances where we need to test a single token.

The two tokens that are compared are the first that appear after `\if`, *even if they are both* found inside the same macro following it. Understanding that principle makes sense of these samples which would otherwise be mystifying.

```
\def\aa{ab}
\def\bb{ab}
\if\aa\bb
```

tests false, because `\if` expands `\aa` and compares 'a' with 'b'. Whereas

```
\def\aa{aa}
\def\bb{bb}
\if\aa\bb
```

tests true, because TEX compares 'a' with 'a' in the macro `\aa`. `\if` doesn't process `\bb` since it has already found two unexpandable tokens and in this case will cause the letters 'bb' to print since the conditional is set to true and `\bb` is found in the true part of the conditional.

There is another problem to consider. Since `\if` expands a control sequence to its bottom level, meaning every control sequence that is found in the definition of a command being expanded will itself also be expanded, it may generate an error message if a control sequence is expanded that contains an `@` in its name.

This problem arises because Plain TEX commonly includes `@` as part of macro names, with the catcode of `@` set to that of a letter. The catcode of `@` is set to 'other' in normal text so that when a Plain TEX command of this sort is expanded in text the `@` is no longer understood as a letter, and TEX will give the user an error message about an undefined control sequence. For example,

```
\if\footnote X  Yes\else No\fi
```

produces this error message:

```
! Undefined control sequence.
\footnote #1->\let \@sf
                        \empty \ifhmode...
```

How 'ifx' works. `\ifx`, on the other hand, will not have this problem since it only expands to the first level of macro expansion. If `\dog` is defined by `\def\dog{\cat}`, for instance, `\ifx` will expand `\dog` as far as `\cat` but will not expand `\cat` to use its definition.

This means that when we want to compare control sequences, and to supply one control sequence as an argument to a macro, we can use the `\ifx` conditional to look at the name of the macro supplied without having to worry about macros that may be contained in its definition.

For example, we can define `\def\aster{*}` so that we can use it to compare with another macro. Inside the macro where we want to make the comparison, we can write

```
\def\sample#1{\def\one{#1}\ifx\one\aster...
```

making both the argument to `\sample` and * be defined as macros.

When `\sample` is used, `\ifx` causes only one level of expansion. If the argument given to `\sample` is `\footnote`, as in the `\if` example above, `\one` will be defined as `\def\one{\footnote}`. `\ifx` will expand `\one` to find '`\footnote`' but will not expand it any further, and will not give an error message.

Another reason to use `\ifx` to compare control sequences is that `\ifx` will pick up both control sequences following it and compare them. When we try the same samples with `\ifx` that we did with `\if`, we will find that we get results opposite to those we got with `\if` — and we will get the results we would want when comparing control sequences.

```
\def\aa{ab}
\def\bb{ab}
\ifx\aa\bb
```

tests true, because `\aa` and `\bb` match each other in their first level of expansion, whereas

```
\def\aa{aa}
\def\bb{bb}
\ifx\aa\bb
```

tests false because the first level of expansion of `\aa` and `\bb` do not match.

Picking Up Information

Defining a macro that will pick up and process a variable number of arguments. There are many instances where you might want to allow a variable number of arguments. Table macros are one such case, in which the user might supply the width of each column as an argument, and the number of columns may well vary from table to table. A table alignment macro that determines whether each column in the table should be aligned to the right, left, or center, is another case where processing a variable number of arguments is necessary.

There is a general method for constructing a macro that will accomodate a variable number of arguments. This method is to pick up all the arguments as one unit and then take that unit apart as a second step. For example, `\table 1in 2.3in 4in*` can be the command to start a table using dimensions to specify the width of each column. When `\table` is defined as `\def\table#1*{....}` we can pick up all the dimensions as the first argument, since the first argument ends with `*`, then use a second macro to process each dimension as its argument. The second macro will call itself again after each dimension is processed until all the dimensions have been used.

Here, in a sample macro, we define `\pickup` as `\def\pickup#1*{...}` to use it to pick up everything between `\pickup` and the `*` as its first argument. Then we use `\expandafter` to allow `\lookatarg` to process the contents of the first argument.

```
\def\pickup#1*{\expandafter\lookatarg#1*}
```

The definition of `\lookatarg` contains a looping mechanism: It is a conditional that tests to see if its argument is equal to '`*`'. It will keep calling itself (recursing) until its argument is `*`. It calls itself by redefining the command `\go` within the conditional, and calling for `\go` outside the conditional. (`\go` must be placed outside the conditional. If it were to be used inside the conditional it would take the `\else` or the `\fi` as its argument and massive confusion would result.) When `\lookatarg` sees '`*`' as the argument, it will define `\go` as `\relax` and thus will not call itself again.

First we define `\aster` so that we have a command to use with `\ifx` to compare with the argument of `\lookatarg`:

```
\def\aster{*}
```

Now we can compare the argument of `\lookatarg` with `\aster`. Thus, with

```
\def\lookatarg#1{\def\one{#1}
  \ifx\one\aster\let\go\relax
  \else Do Something \let\go\lookatarg
  \fi\go}
```

if we use the `\pickup` macro as follows

```
\pickup abc def*
```

the results would be:

Do Something Do Something Do Something Do Something Do Something Do Something

`\lookatarg` has been invoked 6 times since it picked up 6 tokens before it found the `*`. We can substitute some other command for 'Do Something' and build a more useful macro.

Here are two applications of the technique demonstrated in `\lookatarg`; a macro to underline every word in a given section of text, and a macro to process a given section of text to imitate the small caps font.

First, we define `\underlinewords`, which picks up the whole body of text to be underlined:

```
\long\def\underlinewords #1*{%
  \def\wstuff{#1 }\leavevmode
  \expandafter\ulword\wstuff * }
```

Here `\leavevmode` asks TEX to go into horizontal mode. Since each word will be placed in a box, we need this command to prevent the boxes from stacking vertically, as they would in vertical mode.

Now we define `\ulword` which will unpack the text picked up, word by word, put each word in a box, and provide a horizontal rule under each:

```
\long\def\ulword#1 {\def\one{#1}%
  \ifx\one\aster\let\go\relax
  \else\vtop{\hbox{\strut#1}\hrule \relax}
    \let\go\ulword
  \fi\go}
```

The space given after the argument number in the parameter field will allow us to pick up one word at a time, since the collection of the argument will be completed only when `\ulword` sees a space. Here we use `\underlinewords`:

```
\underlinewords
non-outer dynamic allocation*
```

which results in:

<u>non-outer</u> <u>dynamic</u> <u>allocation</u>

The macro `\fakesc` is another construction using this technique. It lets you set text in large and small caps, imitating the 'small caps' font. Its arguments are, in order, the font for the larger letters, the font for the smaller letters, and the text that is to be set in small caps.

When we use `\fakesc` we need to declare the two fonts to be used:

```
\font\big=cmr10
\font\med=cmr8
```

and then

```
\fakesc\big\med  Here are Some Words to be
Small Capped. NASA, Numbers, 1990*
```

will result in:

HERE ARE SOME WORDS TO BE SMALL CAPPED. NASA, NUMBERS, 1990

The macro starts by defining the two fonts and the text to be processed; there is a space after `#3` in `\def\stuff{#3 }` because `\pickupnewword` needs a space to complete its argument when the last word is found as `\stuff` is expanded:

```
\def\fakesc#1#2#3*{\def\bigscfont{#1}%
  \def\smscfont{#2}\def\stuff{#3 }%
  \expandafter\pickupnewword\stuff *}
```

`\pickupnewword` picks up one word at a time, in order to preserve the space between words. If we just asked `\pickupnewlett` to process the entire third argument of `\fakesc`, the space between words would be thrown away as irrelevant space appearing before the next character being looked for as the argument of `\pickupnewlett`. Here, then, is the definition of `\pickupnewword`:

```
\long\def\pickupnewword#1 {%
  \expandafter\pickupnewlett#1\relax}
```

Once `\pickupnewword` has picked up the word, `\pickupnewlett` is used to test each letter to determine whether it should be capitalized. If so, it uses the larger size font; otherwise, the smaller. `\pickupnewlett` tests to see if the argument is uppercase by using the first argument to define `\letter`, `\def\letter{#1}`, and then defines `\ucletter` in an uppercase environment.

```
\uppercase{\def\ucletter{#1}....}
```

Now it uses the `\if` conditional to compare `\letter` and `\ucletter` If they match `\pickupnewlett` makes the current letter or number be printed in the larger font; otherwise the smaller font is used. Note that we can use the `\if` conditional here since we are only comparing single letters. So, finally, the definition of `\pickupnewlett`:

```
\def\pickupnewlett#1{\def\letter{#1}%
  \if\letter*\unskip\let\go\relax
  \else%
    \if\letter\relax{\bigscfont\ }%
      \let\go\pickupnewword
    \else\uppercase{\def\ucletter{#1}%
      \if\letter\ucletter%
        {\bigscfont#1}\else{\smscfont#1}
      \fi}%
      \let\go\pickupnewlett
    \fi\fi\go}
```

When to pick up text as an argument, and when to to pick up text in a box. The correct timing of catcode changes is an issue of concern to the macro writer. Picking up text as an argument will usually be the right way to provide information for the macro, but will fail if you need to change catcodes, since catcodes are irrevocably assigned at the time TEX reads each character. Thus, by the time TEX has picked up an argument, the catcode of all the tokens in the argument are set, and no amount of fiddling with the argument within a macro will change this.

Even a catcode change asked for in the body of an argument will not effect a catcode change because the catcodes of the tokens will already be set by the time TEX expands the request for the catcode change.

There are two ways to solve this problem. In simple cases, one can build a macro containing the desired catcode changes and then invoke a second macro within the first, i.e.,

```
\def\changecat{\bgroup
  \obeylines\pickupchanged}
\def\pickupchanged#1\endchange{%
  \setbox0=\vtop{\hsize=1in#1}%
  \centerline{xx\vtop{\unvbox0}yy}%
  \egroup}
```

In `\changecat` a catcode change is produced by `\obeylines` which changes the catcode of the end-of-line character, `^^M`, to 13 ('active') so that it can be defined as `\par`. Once that catcode change is made, `\pickupchanged` is invoked. Its argument has the end-of-line character set to category 13 at the time the argument is picked up. Notice the `\bgroup` command in `\changecat` is matched with the `\egroup` command in `\pickupchanged` to confine the catcode change.

Used:

```
\changecat
What
Happens
Here?
\endchange
```

xx What yy
Happens
Here?

But, though this example will work for a catcode change set within the `\changecat` macro it will not allow `\changecat ...\endchange` to pick up an argument that contains a catcode change, for instance, an argument containing macros to produce verbatim text, as we could do with the following technique.

The two-part macro defined below will build a box, starting it in `\pickupcat` with `\setbox0\vtop \bgroup`. Any material found between it and `\endpickup` will be expanded, and finally the box will be completed with the `\egroup` found in `\endpickup`, a construction that allows a catcode change between the first and second part of the macro.

```
\def\pickupcat{\global\setbox0
  \vtop\bgroup\hsize=1in\obeylines}
\def\endpickup{\egroup%
  \centerline{XXX\vtop{\unvbox0}YYY}}
```

`\pickupcat...\endpickup` is shown using a previously defined verbatim environment, `\beginverb...\endverb`, to pick up and move verbatim text in a box. Once we see this principle, we can see that a revision of this macro would allow us to place verbatim text in a figure environment, or in a table, or in another environment where it would normally be difficult to introduce material with changed catcodes. For one example:

```
\pickupcat
\beginverb
   Test of
%$_^#@\
    Verbatim

  text.
\endverb
\endpickup
```

produces:

```
XXX    Test of     YYY
    %$_^#@\
         Verbatim

       text.
```

Looking ahead at end of line to preserve blank lines. Since TeX normally ignores blank lines between paragraphs and in some cases we might want to maintain blank lines, we need to develop a way to test for blank lines and provide vertical space when one is present. In this case, we are not interested in picking up text but in picking up information. What comes after each end-of-line character?

As previously mentioned, `\obeylines` changes the end-of-line character, `^^M` to `\par`. You can define `^^M` to do other things as well. For instance, you can define it to be a macro that will supply a baselineskip when the next line is blank or a lineskip when the next line is not.

In this example, `^^M` will be defined as `\lineending`, a macro that includes `\futurelet` to look ahead in the text. If the character that it sees is *itself* (`\lineending`), the next line is blank, since there is nothing from one end-of-line character to the next one. The macro `\looker` will then supply a baselineskip. If it does not see itself, indicating that the next line is not blank, `\looker` will supply a lineskip:

```
\def\looker{%
  \ifx\next\lineending%
    \vskip\baselineskip\obeyspaces\noindent%
  \else%
    \ifx\next\endgroup\else%
      \vskip\lineskip\obeyspaces\noindent%
  \fi\fi}

\def\lineending{\futurelet\next\looker}

{\obeylines
\gdef\saveblanklines{\bgroup\obeylines%
  \let^^M=\lineending}}

\def\endsavelines{\egroup}
```

Example:

```
\saveblanklines
Here is

a blank line,
and a non-blank line.
\endsavelines
```

which produces

Here is

a blank line,
and a non-blank line.

Passing Information: When Counters Can be More Advantageous than Newif's

Hash size, the size of that part of TeX's memory in which it stores control sequence names, is usually not something about which the macro writer has to be concerned. When building a large macro package, however, hash size can be exceeded, making the number of control sequences defined an important issue. One way to economize on the number of definitions in a package is to use counters to pass information rather than using `\newif`s.

When the number of control sequences is not important, `\newif` can be used to create a conditional. This conditional can then be set to true or false in one macro, and tested to see if it is true or false in another as a way of passing information from one macro to another.

However, every time a `\newif` declaration is used, three new definitions are generated. If saving hash size is an issue, we can use `\newcount` instead, and only one new definition is generated.

We can use `\newcount` to allocate a counter and assign it a name, e.g., `\newcount\testcounter`. Then, instead of setting a conditional to true or false, i.e., `\global\titletrue`, and testing for it, i.e., `\iftitle ...\else ...\fi` we can test for the value of the counter. For example,

```
\global\testcounter=1
```

could be the equivalent of `\global\titletrue`. We can then use this test:

```
\ifnum\testcounter=1 ...\else ...\fi
```

Counters have the additional advantage of allowing you to test for a range of numbers, i.e.,

```
\ifnum\testcounter>1 ...\else ...\fi
```

so you can write more compact code when testing for a number of options.

For instance, if we were to write a macro that allowed the user to choose to fill a box by pushing the text to the left, center, or right, we could assign a numerical value to each of the options. If we assigned a value to `\testnum` according to the plan, `left=0, center=1, right=2`, we could test for a range of numbers when another macro was determining which way to fill the box. The test could look like this:

```
\hbox to\hsize{\ifnum\testnum<1
    %% if text is to be pushed to the left
\else
    %% if text is to be either centered or
    %% pushed to the right, do \hfill
  \hfill\fi
  ⟨text⟩
\ifnum\testnum>1
    %% if text is to be pushed to the
    %% right, don't do \hfill
\else
    %% or text is either centered
    %% or pushed to the left
  \hfill\fi}
```

This same principle can be used in more complicated cases as a way of reducing great masses of nested conditionals to a test of the range of the value of a particular counter.

Methods of Conserving Hash Space

As mentioned earlier, using counters to pass information rather than `\newif\thinspace` s is one way to help prevent the hash size from being exceeded. Here are some others.

Input separate macro files on demand. To reduce the number of macros in a macro file, break up the complete macro package into a general macro file and a number of secondary macro files. Within the general macro file definitions can be made that read in the secondary files only when the user calls for a macro for a particular function. For instance, a file containing all the table macros will only be read in if the user uses the general table macro. This principle can be used for listing macros, indexing macros, and any other sort of macro that will not necessarily be used for every document.

Using non-outer dynamic allocation. Dynamic allocation is the way macro writers are able to access the next available number of a dimension, box, or counter at TeX processing time and assign a symbolic name to it. `\newdimen`, `\newbox` and `\newcount` are the commands that allocate these numbers dynamically. It is safer to use dynamic allocation in a macro than to use a particular numbered box, counter, or dimension, since it prevents accidental reallocation.

Unfortunately, all of the commands in this useful set are `\outer`, which means that they cannot be declared *within* a macro. By making these dynamic allocation macros non-outer, we can then include them inside macros and only declare new counters or new boxes or new dimensions when they are needed.

Here is how to make these commands non-outer. Simply copy the original definition, supply a new control sequence name and define them without the `\outer` that originally preceded the definition. For example, the definition of `\newbox` was originally

```
\outer\def\newbox{%
  \alloc@4\box\chardef\insc@unt}
```

Here are the new versions, `\nonouternewbox`, etc.:

```
{\catcode`\@=11
\gdef\nonouternewbox{%
  \alloc@4\box\chardef\insc@unt}
\gdef\nonouternewdimen{%
  \alloc@1\dimen\dimendef\insc@unt}
\gdef\nonouternewcount{%
  \alloc@0\count \countdef \insc@unt}
\catcode`\@=12}
```

In the next section we will see these non-outer commands being used in a table macro, only making named boxes or dimension when needed. Macro writers may find other uses for this technique as well.

Fun with Csname

One of the really useful features of `\csname` is that control sequences can be expanded within the body of the `\csname...\endcsname` construction:

```
\expandafter
\def\csname\testmacro\endcsname{%
  ⟨definition⟩...}
```

Counters can be used:

```
\expandafter
\def\csname\testcounter\endcsname{%
  ⟨definition⟩...}
```

Counters with roman numerals can be used:

```
\expandafter
\def\csname\romannumeral\testcounter%
  \endcsname{⟨definition⟩...}
```

You can even make a definition out of numbers or other symbols that ordinarily are not allowed for a control sequence name:

```
\expandafter\def\csname 123&#\endcsname{%
  ⟨definition⟩...}
```

The only thing to remember here, is that any control sequence made with `\csname` that contains anything other than letters must be invoked as well as defined with `\csname`: `\csname 123&#\endcsname` is the way to use this macro.

Using `\csname` with `\expandafter` makes it possible to do all sorts of things that would not otherwise be possible. Some examples will be found in the following text.

Macros that define new macros using data supplied. One example involves having one macro define another macro where the name of the second macro depends on data supplied in the text.

In the example constructed below, `\usearg` takes the first two words as arguments `#1` and `#2`, reverses their order and uses them to make a control sequence name. This control sequence is then defined to be the complete name and address of the person whose name was used to form the control sequence name.

The order of the name is reversed so that the names of the new macros can be sent to an auxiliary file and be sorted alphabetically. The Appendix illustrates how a more elaborate form of this set of macros may be used to manipulate mailing lists.

`\obeylines` below changes the catcode of the end-of-line character (`^^M`) to 13 so it can be used as a argument delimiter in the definition of `\usearg`; `\obeylines` also defines `^^M` as `\par` so that every line ending seen on the screen is maintained when the text is printed:

```
{\obeylines
\def\usearg#1 #2^^M#3^^M^^M{%
  \expandafter\gdef\csname #2#1\endcsname
  {#1 #2\par #3}}
```

With this definition,

```
\usearg George Smith
21 Maple Street
Ogden, Utah 68709

\SmithGeorge
}
```

produces

George Smith
21 Maple Street
Ogden, Utah 68709

The Appendix contains a macro subsystem for processing and sorting address labels; it demonstrates this technique and many of the others discussed in this paper.

Macros that define new macros using a counter. Here is another use of `\csname`: In this case, it is used to define a macro that will itself define a new macro every time it is used, with the name of the new macro determined by a counter whose number is represented with roman numerals. This can be used to construct a series of macros that expand into areas of text to be reprocessed at the end of a document. For instance, this technique could be used to produce a set of slides from given portions of the text of a document.

In the following example, each time `\testthis` is called it will define a new control sequence. It makes the name of the new control sequence by advancing `\testcounter` which is operated on by `\romannumeral` to produce a new set of letters. These letters will appear in the name of the new macro.

Each control sequence is then sent to an auxiliary file, embedded in code to make the slide. (The slide formatting code is represented here as [[[ ]]]). At the end of the document the auxiliary file containing all the definitions can be input, to produce a set of slides.

```
\newcount\testcounter
\testcounter=501
    %% just to start with a large
    %% number to make into roman numerals
\newwrite\sendtoaux
\immediate\openout\sendtoaux
  \jobname.aux %% opening a file to write to
\def\testthis#1{%
  \global\advance\testcounter by1
  \expandafter\gdef\csname%
  \romannumeral\testcounter XYZ\endcsname{%
      [[[#1]]]}
\immediate\write\sendtoaux{%
\noexpand\csname\romannumeral\testcounter
XYZ\noexpand\endcsname}}
```

Here is an example of `\testthis` being used:

```
\testthis{This is the first bit of text...}
\testthis{This is the second...}
```

The code above writes the following lines into the `.aux` file:

```
\csname diiXYZ\endcsname
\csname diiiXYZ\endcsname
```

and when the `.aux` file is input, these commands produce

[[[This is the first bit of text...]]] [[[This is the second...]]]

Designing generic code with csname. Another really important use for `\csname` constructions is a way of making compact code for a section of a macro that repeats many times with a small variation each time. Table macros are often examples of this kind, since they tend to have repeating sections, one for each column.

We consider below some parts of a set of macros for table construction showing several ways that `\csname` can be used. The form adopted for the `\halign` command line, using a `&` immediately after the `\halign{`, will allow the specifications for this column to be used for each column in the table.

Using `\csname` with a counter allows this set of commands to be defined only once. Each new column entry will cause the `\colcount` counter to advance making the otherwise similar column definition use a new counter value inside the `\csname...\endcsname` constructions.

Non-outer dynamic allocation is used to name only those counters, dimensions, or boxes that are needed when the table is made up. A new set is declared for each column of the table. Since we don't need to guess ahead of time how many columns are going to be used, only those dimensions or boxes that are needed will be declared. In addition, `\ifdefined` (below) tests to see if the particular set of dimensions and boxes has been used in a previous table, and will only declare a new set if they have not been defined previously.

`\xtab` and `\dtab` involve counters only, so they can be used later in a `\csname...\endcsname` construction where it doesn't matter if the expansion will produce numbers as part of the control sequence. `\gtab` and `\vtab`, on the other hand, need to be used as ordinary control sequences which is the reason for the `\romannumeral` command that will produce letters instead of numbers when the `\gtab` and `\vtab` are expanded.

`\asizetab` and `\finishasizetab` will use these boxes and counters to actually set the table entries.

Here, finally, are some definitions for table construction:

```
\def\multipagetable{\global\firstcoltrue
  \halign\bgroup%
    &\global\advance\colcount by1\relax%
    \ifdefined{\the\colcount tab}{}{%
      \edef\xtab{\expandafter\csname
        \the\colcount tab\endcsname}%
      \edef\dtab{\expandafter\csname
        \the\colcount tabwide\endcsname}%
      \edef\gtab{\expandafter\csname
        \romannumeral\colcount
          gapped\endcsname}%
      \edef\vtab{\expandafter\csname
        \romannumeral\colcount
          vlinewd\endcsname}%
      \expandafter\nonouternewdimen\vtab%
      \expandafter\nonouternewbox\xtab%
      \expandafter\nonouternewdimen\dtab%
      \expandafter\nonouternewcount\gtab%
    }%
    \ifdefined{align\the\colcount tab}{}{%
      \edef\atab{\expandafter\csname
        align\the\colcount tab\endcsname}%
      \expandafter\nonouternewcount\atab}%
  \asizetab##\finishasizetab\cr}
```

A `\csname..\endcsname` construction defined using one counter can be invoked *using a different counter*, if that proves useful. Another part of the code for multipage tables uses a second counter to invoke macros defining boxes containing the column heads, used when the table continues over page breaks. Even though the original definition used `\colcount` as the counter to name the boxes, `\contcolcount`, another counter, can be used in another macro to invoke the same definition. When TeX expands `\csname...\endcsname` construction it produces a number as the replacement for the counter, so the name of the counter used doesn't affect the result. This might be helpful in cases where you don't want to change the value of one counter, but still wish to use a `\csname` construction that contains it.

Tips on Table Macros

`\everycr` is a TeX primitive for a token list. It functions similarly to `\everypar` or `\everymath` in that its definition will be used every time the named environment is present, in this case after every `\cr`. By setting `\everycr` equal to some definition we can insert a set of commands after every line in a table, since every line will end with a `\cr`. A simple example is this:

```
\everycr={\noalign{\hrule}}
```

which will insert a horizontal rule automatically after every `\cr` in the table. Once this possibility is discovered, the macro writer may realize that there are many other things that can be done with `\everycr`, such as including a set of conditionals that will call for horizontal lines with breaks in them, double horizontal lines, thicker horizontal lines, thicker lines under some columns but not under others, and so on.

You can include a counter which is advanced every time `\everycr` is called, and use that counter to determine how many lines have been used in the table, in order to stop and restart the table, making it possible to have a table that will continue for hundreds of pages without TEX running out of memory.

Moreover, you can call for a small vertical skip in the `\everycr` definition which will allow the table to break over pages. If you use the following construction, your table will break over pages and a horizontal line will appear both at the bottom of the previous page and at the top of the new page, without the user having to know ahead of time where the table will break.

```
\everycr={\noalign{\hrule\vskip-1sp\hrule}}
```

When Doing Nothing is Helpful

The usual form of a macro with an argument is (in its most basic form) `\def\example#1{#1}`. There are cases in which **not using** the argument can be helpful when you want to get rid of something: `\def\example#1{}`.

You can use this principle to prevent large sections of text from being processed by TEX.

```
\long\def\ignorethis#1\endignore{}
```

Thus

```
\ignorethis
Here is some text that will be ignored...
\endignore
This is where \TeX\ starts printing text...
```

will produce

This is where TEX starts printing text...

You might want to use this macro in the process of debugging a document you are working on. All text between `\ignorethis` and `\endignore` will be ignored, making it possible for TEX to print only the part of the document in which you are interested. TEX will run out of memory after about 6 pages of text is picked up by the `\ignorethis` macro, depending on the implementation of TEX being used, but if you want to ignore more than 6 pages of text you can end the first `\ignorethis` with `\endignore` and enter a second `\ignorethis ...\endignore`.

A slight improvement, however, is needed to prevent TEX from complaining if an `\outer` command is found in the argument of `\ignorethis`. This is the error message which we would like to avoid:

```
! Forbidden control sequence found
while scanning use of \ignorethis.
```

We can avoid it by changing the catcode of the backslash to be that of a letter. Now there will be no commands processed until `\ignorethis` encounters `\endignore` and the catcode changes are turned off.

```
\long\gdef\ignorethis{\bgroup
  \catcode`\\=12 \catcode`\^=12 \finish}

{\catcode`\|=0 |catcode`|\=12
|long|gdef|finish#1\endignore{|egroup}%
}
```

Note that here, too, `#1` is never used in the replacement part of the macro.

Getting rid of backslashes. Here is another example of an argument that is thrown away:

```
\def\stripbackslash#1#2*{\def\one{#2}}
```

which only uses the second argument, throwing away the first argument, in this case stripping away a backslash from a control sequence supplied by the user. `\stripbackslash` can then be used in another macro which needs a control sequence without its backslash to work correctly, for instance:

```
\def\newdef#1{\expandafter
  \stripbackslash\string#1* \one}
```

When this is used,

```
\newdef\testmacro
```

produces

```
testmacro
```

Instead of simply printing the control sequence without the backslash, `\newdef` can be rewritten to test to see if a given macro has already been defined. In this example, `\newdef` tests to see if the expansion of the control sequence `\csname\one\endcsname`, (where `\one`, was defined by `\stripbackslash` to be the control sequence supplied by the user minus its backslash) is equal to `\relax`. This takes advantage of the TEX convention that a previously undefined control sequence invoked in a `\csname...\endcsname` environment will be understood to be equal to `\relax`, whereas an already defined control sequence will not:

```
\def\newdef#1{%
  \expandafter\stripbackslash\string#1*
      %% \stripbackslash defines \one
  \expandafter
    \ifx\csname\one\endcsname\relax
      %% \one is expanded to be the
      %% control sequence the user supplied
      %% minus the backslash.
      %% If csname construction equals
      %% \relax, do nothing
    \else   %% Else, give error message:
      {\tt Sorry, \string#1 has already been
      defined. Please supply a new name.}
    \fi}
```

In the test below, notice that we do not get an error message for `\cactus` which hasn't been previously defined, but we do get a message for TEX, which is defined:

```
\newdef\TeX
\newdef\cactus
```

produces

```
Sorry, \TeX has already been defined.
Please supply a new name.
```

Not using boxes. Similar to not using arguments, there are times when setting a box and then **not** using it can be useful.

When writing a macro to make text to wrap around a given figure, we might want to use a test box to put a given amount of text in, say, a paragraph, which has been picked up as an argument to a macro. We can then measure the box to see if it will exceed the depth of the figure. If it does not, the box can be used as it is, but if it does, the box can be ignored and the argument re-used, with changed `\hangindent` and `\hangafter`, to allow the text to fit around the figure neatly. This works because text picked up as an argument to a macro does not yet have its glue set, so it can accomodate different line widths.

Another use for a box that is never printed is to use it as a container in which to expand a macro having symbols in the parameter field. For example, if the macro `\splittocentry` is defined by

```
\def\splittocentry#1-#2-#3{\gdef\one{#1}
  \gdef\two{#2}\gdef\three{#3}}
```

we can use it in a another macro to process an argument which may or may not include the hyphens, i.e.,

```
\setbox0=\hbox{\expandafter
  \splittocentry#2-{}-{}}
```

The hyphens that are necessary to complete the use of `\splittocentry` are supplied in the box but they will not print if the replacement for `#2` turns out to supply the hyphens already. Since `\one`, `\two`, and `\three` are globally defined (`\gdef`), their definition will be understood outside the box.

Some General Macro Writing Tips

There are several commands that can make the process of macro writing easier.

`\show` is a TEX primitive that will cause the definition of the macro it precedes to appear on your screen when you run TEX on a file that contains it. `\show\samplemacro` will cause the definition of `\samplemacro` to be appear on your screen, for example. `\show` can be temporarily included inside a macro to let you see what is being picked up as arguments. For instance, if

```
\def\test#1,#2{\def\one{#1}\def\two{#2}
\show\one\show\two....}
```

then

```
\test some, stuff
```

will help you see what is being picked up as argument `#1` and `#2`. In this example the results are obvious, but there are more complicated situations. For example, when one macro is looking at the contents of another macro, a test like this can quickly help you understand what TEX sees when it picks up an argument, a helpful debugging tool. It also has the advantage of giving you information at the time you TEX your file, saving you the steps of either previewing or printing the `.dvi` file.

`\show` will also send the definition of the macro that it precedes to the `.log` file, a feature which you can take advantage of when you are interested in redefining a Plain TEX macro. If you write `\show\raggedright`, for example, in a test file and run TEX on that file, the definition of `\raggedright` will appear in the `.log` file. You can then move those lines of code from your `.log` file to your macro file and you will have saved yourself the trouble of looking up the command in *The TEXbook* and copying it into your file. Now you are ready to make changes to the original macro.

A related command, `\showthe`, will give you the current value of a token list, like `\everypar`. Including `\showthe\everypar` in a test file can tell you what TEX sees as the current value of `\everypar` at that point in the file. You can also use `\showthe` to get the current value of a counter or dimension. You may want to include a `\showthe` temporarily in a macro you are developing, similarly to `\show`, as a debugging tool.

Finally, using `\message` in a conditional while working on a macro can give you helpful information. You could put this code in a headline, for instance, to be able to see the state of a particular conditional in the headline,

```
\headline={....%
  \iftitle
    \message{SEES TITLE, WIN}
  \else
    \message{NO TITLE, LOSE}
  \fi...}
```

or include a similar construction in the body of a macro while you are testing it. When you TEX the file you can quickly see if you are getting the results you were expecting.

Appendix

Code to Alphabetize an Address List

These macros demonstrate many of the techniques discussed in this paper. The macros process an existing an address list by taking the first line of each address, re-ordering the name with last name first, then turning the name into a control sequence which is sent to an auxiliary file. The user must alphabetically sort the auxiliary file. The resulting sorted file is then input back into the originating file and the whole address list will be transposed and printed in alphabetical order.

The user enters `\alphalist` at the beginning of an address list, and a blank line and `\endalphalist` at the end. `\alphalist` picks up the name, then makes a macro using the name (last name first) as the control sequence. This control sequence is sent to auxiliary file with the same name as the originating file and with an `.alf` extension. The file `filename.alf` must be sorted to produce `filename.srt`, using a sort routine on the user's system. If DOS, write

```
sort < filename.alf > filename.srt
```

`\endalphalist` checks to see if `filename.srt` exists, and if so, will `\input filename.srt`. The sorted list of control sequences will produce an alphabetized address list.

First we name dimensions and counters and set them to arbitrary sizes.

```
\newdimen\heightofentry \heightofentry=.75in
\newdimen\widthofentry \widthofentry=.3\hsize
\newcount\namenum
```

`\alphalist` makes every new paragraph start with the command `\look`. `\obeylines` will maintain the same line endings as seen on the screen.

```
\def\alphalist{\bgroup\obeylines
  \global\everypar={\look}}
```

First we discuss the definition of `\look`, then we will consider the macros used in its definition.

`\look` picks up the entire name. It then defines it as `\test`. `\test` is placed in `\box0` and expanded after `\throwawayjr` which defines `\fullname`. Then `\fullname` is expanded after `\takeapart` to define `\nameinrev`. `\nameinrev` is the name in reverse order; it is used as the name of a control sequence that defines the entire name and address. `\nameinrev` in a csname environment is also sent to an auxiliary file so that it can be sorted alphabetically. Here is the definition of `\look`:

```
{\obeylines
\gdef\look#1^^M#2^^M^^M{\def\test{#1}
  \setbox0=\hbox{%
    \expandafter\throwawayjr\test, {}}
  \global\namenum=0
  \expandafter\takeapart\fullname
  \obeylines
  \everypar={}
  \expandafter%
  \gdef\csname\nameinrev\endcsname{%
    \vtop to\heightofentry{\parindent=0pt
      \vfill\hsize=\widthofentry
      #1
      #2
      \vfill}}%
  \immediate\write\alphafile%
    {\noexpand\csname\nameinrev%
    \noexpand\endcsname}%
  \global\everypar={\look}}
}
```

Now we consider the commands used in the definition of `\look`.

To make the last name appear first in the command sent to the auxiliary file, we count the number of parts to the name ("Mr. R. G. Greenberg" has four parts, for example) and use `\ifcase` to select the correct order. After `\nameinrev` (for 'name in reverse order') is defined, it will then be used in the `\look` macro to create a control sequence by being expanded within a csname.

```
\def\makerightdef{\ifcase\namenum\or
  \or\gdef\nameinrev{\one}
  \or\gdef\nameinrev{\two\one}
  \or\gdef\nameinrev{\three\one\two}
  \or\gdef\nameinrev{\four\one\two\three}
  \or\gdef\nameinrev{\five\one\two\three%
    \four}
  \fi}
```

`\makedef` gives a control sequence name to the argument of `\takeapart` according to the number of times `\takeapart` is invoked:

```
\def\makedef#1{\ifcase\namenum
  \or\gdef\one{#1}
  \or\gdef\two{#1}
  \or\gdef\three{#1}
  \or\gdef\four{#1}
  \or\gdef\five{#1}
  \fi}
```

In order to make an `\ifx` comparison, we set

```
\def\aster{*}
```

`\takeapart` loops until it sees the `*`, which will be supplied in the `\throwawayjr` macro:

```
\def\takeapart#1 {%
  \global\advance\namenum by1
  \def\onex{#1}
  \makedef{#1}
  \ifx\onex\aster
    \makerightdef\let\go\relax
  \else
    \let\go\takeapart
  \fi\go}
```

We want to alphabetize according to the last name, and not mistakenly use 'Jr.' as the last name. The first argument ends when `\throwawayjr` sees a comma, which would normally occur before a Jr. or Sr. following a name. The second argument is never used, which is how Jr., or Sr., or III, are thrown away:

```
\def\throwawayjr#1, #2{%
  \gdef\fullname{#1 * }}
```

`\throwawayjr` is used inside a box that is never used, so we can supply the comma that ends argument `#1`, in case there is no comma in the name given. If a name is used that contains a comma, that comma delimits the first argument. Since the extra comma is in a box that is never invoked, the extra comma is never printed.

Here is code to open an auxiliary file whose name is the same as the file containing `\alphalist`, but with an `.alf` extension:

```
\newwrite\alphafile
\immediate\openout\alphafile=\jobname.alf
```

Now we have finished describing the commands needed to define the names and address and to send their macro names to the auxiliary file, and it is time to input the sorted list.

`\endalphalist` turns off the `\everypar` that was established with `\alphalist` and inputs the `.srt` file if it exists. Since all the definitions precede `\endalphalist`, when the `.srt` file is brought in with the csname control sequences in it, each control sequence will produce its defined name and address:

```
\def\endalphalist{\egroup
  \global\everypar={}
  \openin1 \jobname.srt
  \ifeof1 %
    \message{<<Please sort \jobname.alf
      to produce \jobname.srt >>}
  \else
    \immediate\closein1
    \input \jobname.srt
  \fi}
```

Example:

```
\alphalist
George Smith
21 Maple Street
Ogden, Utah 68709

Jacqueline Onassis
Upper East Side
NYC, NY

Mr. W. T. C. Schoenberg, Jr.
Travesty Lane
Culver City, Iowa

\endalphalist
```

This writes the following lines in the file `test.srt` after `test.alf` is sorted:

```
\csname OnassisJacqueline\endcsname
\csname SchoenbergMr.W.T.C.\endcsname
\csname SmithGeorge\endcsname
```

which will transpose the original list to print the names and addresses in alphabetic order.

The complete address list code.

```
\newcount\namenum
\newdimen\heightofentry \heightofentry=.75in
\newdimen\widthofentry \widthofentry=.3\hsize
\def\alphalist{\bgroup\obeylines
  \global\everypar={\look}}
{\obeylines
\gdef\look#1^^M#2^^M^^M{\def\test{#1}
\setbox0=\hbox{%
 \expandafter\throwawayjr\test, {}}
  \global\namenum=0
  \expandafter\takeapart\fullname
  \obeylines  \everypar={} \expandafter%
  \gdef\csname\nameinrev\endcsname{%
    \vtop to\heightofentry{\parindent=0pt
      \vfill\hsize=\widthofentry
      #1
      #2
      \vfill}}%
  \immediate\write\alphafile%
    {\noexpand\csname\nameinrev%
    \noexpand\endcsname}%
  \global\everypar={\look}}}
\def\makerightdef{\ifcase\namenum\or
  \or\gdef\nameinrev{\one}
  \or\gdef\nameinrev{\two\one}
  \or\gdef\nameinrev{\three\one\two}
  \or\gdef\nameinrev{\four\one\two\three}
  \or\gdef\nameinrev{\five\one\two\three%
    \four}\fi}
\def\makedef#1{\ifcase\namenum
  \or\gdef\one{#1}
  \or\gdef\two{#1}
  \or\gdef\three{#1}
  \or\gdef\four{#1}
  \or\gdef\five{#1}\fi}
\def\aster{*}
\def\takeapart#1 {%
  \global\advance\namenum by1
  \def\onex{#1} \makedef{#1}
  \ifx\onex\aster \makerightdef\let\go\relax
  \else \let\go\takeapart
  \fi\go}
\def\throwawayjr#1, #2{%
  \gdef\fullname{#1 * }}
\newwrite\alphafile
\immediate\openout\alphafile=\jobname.alf
\def\endalphalist{\egroup
  \global\everypar={}
  \openin1 \jobname.srt
  \ifeof1 \message{<<Please sort \jobname.alf
      to produce \jobname.srt >>}
  \else
    \immediate\closein1
    \input \jobname.srt\fi}
```