Macros

A Tutorial on \expandafter
Stephan v. Bechtolsheim

Introduction
I have found from my own experience teaching \TeX{} courses that \expandafter is one of the instructions that people have difficulty understanding. After starting with a little theory, we will present a number of examples showing different applications of this instruction. Later in the article we will also deal with multiple \expandafter's.

This article is condensed from a draft of my book, Another Look at \TeX{}. See the end of this article for more information about the book.

The theory behind it
\expandafter is an instruction that reverses the order of expansion. It is not a typesetting instruction, but an instruction that influences the expansion of macros. The term expansion means the replacement of the macro and its arguments, if there are any, by the replacement text of the macro. Assume you define a macro \texttt{\textbackslash xx} as follows: \texttt{\def\textbackslash xx{\texttt{ABC}}}; then the replacement text of \texttt{\textbackslash xx} is \texttt{ABC}, and the expansion of \texttt{\textbackslash xx} is \texttt{ABC}.

As a control sequence, \expandafter can be followed by a number of argument tokens. Assuming that the tokens in the following list have been defined previously:

\begin{verbatim}
\expandafter \token_n \token_1 \token_2 \\
... \token_n ...
\end{verbatim}

then the following rules describe the execution of \expandafter:

1. \token_n, the token immediately following \expandafter, is saved without expansion.
2. Now \token_1, which is the token after the saved \token_n, is analyzed. The following cases can be distinguished:
   a. \token_1 is a macro: The macro \token_1 will be expanded. In other words, the macro and its arguments, if any, will be replaced by the replacement text. After this \TeX{} will not look at the first token of this new replacement text to expand it again or to execute it. See 3. instead!
   Examples 1–6 and others below fall in this category.
   b. \token_1 is primitive: Here is what we can say about this case: Normally a primitive can not be expanded so the \expandafter has no effect; see Example 7. But there are exceptions:
      i. \token_1 is another \expandafter: See the section on “Multiple \expandafters” later in this article, and also look at Example 9.
      ii. \(\text{\texttt{\textbackslash token_1}}\) is \texttt{\textbackslash csnname}: \TeX{} will look for the matching \texttt{\textbackslash endcsname}, and replace the text between the \texttt{\textbackslash csnname} and the \texttt{\textbackslash endcsname} by the token resulting from this operation. See Example 11.
      iii. \texttt{\textbackslash token_1} is an opening curly brace which leads to the opening curly brace temporarily being suspended. This is listed as a separate case because it has some interesting applications; see Example 8.
      iv. \texttt{\textbackslash token_1} is \texttt{\textbackslash the}: the \texttt{\textbackslash the} operation is performed, which involves reading the token after \texttt{\textbackslash the}.
3. \(\texttt{\textbackslash token_n}\) is stuck back in front of the tokens generated in the previous step, and processing continues with \token_n.

Examples 1 and 2: Macros and \expandafter
In these examples, \(\texttt{\textbackslash token_1}\) is a macro. Assume the following two macro definitions (Example 1):

\begin{verbatim}
\def\textbackslash xx {\texttt{[#1]}}
\def\textbackslash yy {\texttt{[ABC]}}
\end{verbatim}

We would like to call \texttt{\textbackslash xx} with \texttt{\textbackslash yy}'s replacement text as its argument. This is not directly possible (\texttt{\textbackslash xx\textbackslash yy}), because when \TeX{} reads \texttt{\textbackslash xx} it will try to find \texttt{\textbackslash xx}'s argument without expansion. So \texttt{\textbackslash yy} will not be expanded, and because \TeX{} expects square brackets containing the argument of \texttt{\textbackslash xx} on the main token list, it will report an error stating that “the use of \texttt{\textbackslash xx} doesn’t match its definition”.

On the other hand \expandafter\texttt{\textbackslash xx\textbackslash yy} will work. Now, before \texttt{\textbackslash xx} is expanded, the expansion of \texttt{\textbackslash yy} will be performed, and so \texttt{\textbackslash xx} will find \texttt{[ABC]} on the main token list as its argument.

Now assume the following additional macro definition is given (Example 2):

\begin{verbatim}
\def\textbackslash zz {\texttt{\textbackslash yy}}
\end{verbatim}

Observe that \expandafter\texttt{\textbackslash xx\textbackslash zz} will not work, because \texttt{\textbackslash zz} is replaced by its replacement text which is \texttt{\textbackslash yy}. But then \texttt{\textbackslash yy} is not expanded any further. Instead \texttt{\textbackslash xx} will be substituted back in front of \texttt{\textbackslash yy}. In other words the expansion in an \expandafter case is not “pushed all the way”; to
accomplish a complete expansion, one should use \edef, where further expansion can be prevented only with \noexpand. This example (using \zz as an argument to \xx), which would not work with \expandafter, does work with \edef:
\edef\temp\noexpand\xx\zz
\temp
As a side remark: Example 1 can also be programmed without \expandafter, by using \edef:
\edef\temp\zz
\temp

Example 3
In this example, \(\text{\texttt{\texttt{(token1)}}}\) is a definition.
\def\xx\{\yy\}
\expandafter\def\xx\{\texttt{This is fun}\}
\expandafter will temporarily suspend \def, which causes \xx being replaced by its replacement text which is \yy. This example is therefore equivalent to
\def\yy\{\texttt{This is fun}\}

Examples 4 and 5: Using \expandafter to pick out arguments
Assume the following macro definitions \Pick..., of two macros: which both have two arguments and which print only either the first argument or the second one. These macros can be used to pick out parts of some text stored in another macro.
\% \PickFirstOfTwo
\% This macro is called with two
\% arguments, but only the first
\% argument is echoed. The second
\% one is dropped.
\% #1: repeat this argument
\% #2: drop this argument
\def\PickFirstOfTwo #1#2{#1}
 Here is an application of these macros (Examples 4 and 5) where one string is extracted from a set of two strings.
\% Define macro \a. In practice, \a
\% would most likely be defined
\% by a \read, or by a mark.
\def\a\{\texttt{First part}\}{\texttt{Second part}\}

\% Example 4: Generates "First part".
\% Pick out first part of \a.
\expandafter\PickFirstOfTwo\a

\% Example 5: Generates "Second part".
\% Pick out second part of \a.
\expandafter\PickSecondOfTwo\a

Let us analyze Example 4: \PickFirstOfTwo is saved because of the \expandafter and \a is expanded to \{\texttt{First part}\}{\texttt{Second part}\}. The two strings inside curly braces generated this way form the arguments of \PickFirstOfTwo, which is re-inserted in front of \{\texttt{First part}\}{\texttt{Second part}\}. Finally, the macro call to \PickFirstOfTwo will be executed, leaving only \texttt{First part} on the main token list.

Naturally the above \Pick... macros could be extended to pick out \(x\) arguments from \(y\) arguments, where \(x < y\), to offer a theoretical example.

Example 6: \expandafter and \read
The \expandafter can be used in connection with \read, which allows the user to read information from a file, typically line by line. Assume that a file being read in by the user contains one number per line. Then an instruction like \read\texttt{\stream} to \texttt{\InLine} defines \InLine as the next line from the input file. Assume, as an example, the following input file:
\% 12
\% 13
\% 14
Then the first execution of \texttt{\read\stream} to \texttt{\InLine} is equivalent to \def\InLine{12,}, the second one to \def\InLine{13,}, and so forth. The space ending the replacement text of \texttt{\InLine} comes from the end-of-line character in the input file.

This trailing space can be taken out by defining another macro \InLineNoSpace with otherwise the same replacement text. The space contained in the replacement text of \texttt{\InLine} matches the space which forms the delimiter of the first parameter of \temp in the following. Here, the macro \texttt{\readn} reads one line from the input file and defines the
macro \InLineNoSpace as that line without the trailing space:
\newread\stream
\def\readn\{%
\read\stream to \InLine
% \temp is a macro with one
% parameter, delimited by a space.
\def\temp \#1\%
\def\InLineNoSpace{\#1}\%
\expandafter\temp\InLine
\}

Example 7: Primitives and \expandafter
Most primitives trigger no actions by \TeX{} because in
general, primitives cannot be expanded (a few
primitives are treated differently). In this sense,
characters are primitives also. Let's look at an example:
\expandafter AB
Character A is saved. Then \TeX{} tries to expand, but
not print B, because B cannot be expanded. Finally,
A is put back in front of the B; in other words, the
two characters are printed in the given order, and
we may as well have omitted the \expandafter. So
what's the point here? \expandafter reverses the
order of expansion, not of execution.

Example 8: \expandafter to temporarily suspend an opening curly brace
\expandafter can be used to temporarily suspend an
opening curly brace. In the following case this is
done to load a token register.
% Allocate token registers \ta and \tb.
\newtoks\ta \newtoks\tb
% (1) Initialize \ta to "\a\b\c".
\ta = {\a\b\c}
% (2) \tb now contains "\a\b\c".
\tb = \expandafter{\the\ta}
% (3) \tb now contains "\the\ta".
\tb = {\the\ta}
% (4) \tb now contains "\a\b\c".
\tb = \ta

In (1) we load the token register \ta. In (2) the
\expandafter temporarily causes the opening curly
brace to be hidden, so that \the\ta is evaluated,
resulting in the generation of a copy of \ta's
content. In (3) \ta is not copied because when
a token register is loaded, the new content of
it, enclosed within curly braces, is not expanded.
Finally, (4) is equivalent to (2), without using
\expandafter.

More theory: Multiple \expandafter
Sometimes one needs to reverse the expansion,
not of two, but of three tokens. This can be
done. Assume that \a, \b and \c are macros
without parameters and \exi stands for the ith
\expandafter. Then \ex1 \ex2 \ex3 \a \ex4 \b
\c reverses the expansion of all three tokens. Here
is what happens:
1. \ex1 is executed causing \ex2 to be saved.
   Then \TeX{} looks at \ex3.
2. \ex3 is another \expandafter: \TeX{} will
   continue performing \expandafters!
3. \a is saved (the list of saved tokens is now \ex2
   \a).
4. \ex4 is now executed: \b is saved (the saved
   token list is now \ex2 \a \a \b), and \c is
   expanded.
5. Now all saved tokens \ex2 \a \a \b are
   inserted back in front of the replacement
text of \c.
6. The first token on the main token list which
   resulted from the operation of the previous step
   is \ex2, another \expandafter, which
   causes \a to be saved again, and \b to be replaced by
   its replacement text.
7. Finally \a is inserted back into the main token
   list and replaced by its replacement text. The
   execution continues with the first token of the
   replacement text of \a.

One could summarize the net effect of this
sequence of \expandafters and other tokens as
follows (this is important for Example 8): in front
of the expansion of \c the expansion of \b is
inserted and in turn the expansion of \a is inserted
in front of that.

The above example can be "expanded" to
reverse the expansion of any number of tokens.
Very rarely the reversed expansion of four tokens
\a, \b, \c and \d is needed. Assume that all four
tokens are macros without parameters. Here is how
this is done:
\let\ex = \expandafter
% 7 \expandafters
\ex\ex\ex\ex\ex\ex\ex\ex\a
% 3 \expandafters
\ex\ex\ex\b
% 1 \expandafter
\ex\c
\d

In general, to reverse the expansion of n tokens
\{token1\} \ldots \{tokenn\}, the ith token has to preceded
by \(2^{i-1} - 1\) \expandafters.
Example 8: Forcing the partial expansion of token lists of \write

\expandafter can be used to force the expansion of the first token of a delayed \write. Remember that unless \write is preceded by \immediate, the expansion of the token list of a \write is delayed until the \write operation is really executed, as side effect of the \shipout instruction in the output routine. So, when given the instruction \write\stream{\x\y\z}, \TeX will try to expand \x, \y and \z when the \shipout is performed, not when this \write instruction is given.

There are applications where we have to expand the first token (\x in our example) immediately, in other words, at the time the \write instruction is given, not when the \write instruction is later actually performed as side effect of \shipout. This can be done by:

\def\ws{\write\stream}
\let\ex=\expandafter\ex\ex\ex\ws\exC\x\y\z

Going back to our explanation of multiple \expandafters: \ws corresponds to \a, \f to \b, and \x to c. In other words \x will be expanded (!!), and \f will be inserted back in front of it (it cannot be expanded). Finally, \ws will be expanded into \write\stream. Now \write will be performed and the token list of the \write will be saved without expansion. But observe that \x was already expanded. \y and \z, on the other hand, will be expanded when the corresponding \shipout instruction is performed.

Example 9: Extracting a substring

Assume that a macro \xx (without parameters) expands to text which contains the two tokens \aaa and \bbb embedded in it somewhere. You would like to extract the tokens between \aaa and \bbb. Here is how this could be done:

% Define macro to extract substring
% from \xx.
\def\xx{This is fun\aaa TXXTT\bbb That's it}

% Define macro \extract with three
% delimited parameters.
% Delimiters are \aaa, \bbb, and \Del.
% Macro prints substring contained
% between \aaa and \bbb.
\def\extract #l\aaa#2\bbb#3\Del(#2)

% Call macro to extract substring
% from \xx.
% Prints "TXXTT".
\expandafter\extract\xx\Del
% which is equivalent to:
\extract This is fun\aaa TXXTT\bbb That's it\Del

In a "real life example" \xx would be defined through some other means like a \read. There is no reason to go to that much trouble just to print TXXTT.

Example 10: Testing on the presence of a substring

Now let us solve the following problem: We would like to test whether or not a macro's replacement text contains a specific substring. In our example, we will test for the presence of abc in \xx's replacement text. For that purpose we define a macro \@TestSubS as follows: (\&Del is a delimiter):

\def\&TestSubS #labc#2\&Del{...}

Now look at the following source:

\def\xx{AABBCC}
% #1 of \&TestSubS is AABBCC.
\expandafter\&TestSubS\xx abc\&Del
\def\xx{ABBabcDD}
% #1 of \&TestSubS is ABB.
\expandafter\&TestSubS\xx abc\&Del

Observe that

1. If \xx does not contain the substring abc we are searching for, then #1 of \&TestSubS becomes the same as \xx.
2. In case \xx does contain the substring abc, then #1 of \&TestSubS becomes that part of \xx which occurs before the abc in \xx.

We can now design \IfSubstring. It is a simple extension of the above idea, with a test added at the end to see whether or not #1 of \&TestSubS is the same as \xx.

% This conditional is needed because
% otherwise we would have to call the
% following macro \IfNotSubString.
\newif\IfTestSubString
\IfSubstring
% ============
% This macro evaluates to a conditional
% which is true iff #1's replacement
% text contains #2 as substring.
% #1: Some string
% #2: substring to test for whether it
% is in #1 or not.
\def\IfSubString #1#2{%
  \edef\QMainString{#1}
  \def\@TestSubS ##1#2##2\@DelC%
  \edef\@TestTemp{##1)}%
  \expandafter\QTestSubS
  \@MainString#2\QDel
  \ifx\@MainString\@TestTemp
    \QTestSubStringfalse
  \else
    \QTestSubStringtrue
  \fi
} \catcode'@=12

Example 11: \expandafter and \csname

A character string enclosed between \csname and
\endcsname expands to the token formed by the
character string. \csname a?a-4\endcsname, for
instance, forms the token \a?a-4. If you wanted to
use this token in a macro definition you have to do
it the following way:
\expandafter
  \def\csname a?a-4\endcsnameI.

The effect of the \expandafter is of course to
give \csname a chance to form the requested token
rather than defining a new macro called \csname.

Summary

These examples have shown some typical applica-
tions of \expandafter. Some were presented to
“exercise your brains a little bit”. I recommend
that you take the examples and try them out;
there is very little input to enter. I also encourage
you to tell Barbara Beeton or me what you think
about tutorials in TUGboat. There are many more
subjects which could be discussed and which may
be of interest to you.

This article is, as briefly mentioned in the
introduction, an adaptation of a section of my
book, Another Look At \TeX, which I am currently
finishing. The book, now about 800 pages long,
grew out of my teaching and consulting experience.
The main emphasis of the book is to give concrete
and useful examples in all areas of \TeX. It contains,
to give just one example, 100 (!) \halign tables. In
this book you should be able to find an answer
to almost any \TeX problem.

Macros for Outlining

James W. Walker
Department of Mathematics
University of South Carolina

The purpose of this note is to describe stand-
alone macros for the preparation of outlines in the
standard format. For instance, the desired output
might look like:

I. Vegetables
A. Green ones
  1. lettuce
     a. iceberg
     b. leaf
  2. Broccoli, almost universally despised
     by children. The strong flavor is only
     made palatable by quick stir-frying.
B. White ones
  1. potatoes
  2. turnips

II. Animals.

III. Minerals.

Notice that a topic is allowed to be a paragraph,
not just one line, as in topic I.A.2. I wanted
\TeX to take care of the counting and indentation
apainlessly as possible. Something like this
can be done in \LaTeX using nested enumerate
environments, but I wanted the input format to be
even simpler.

When typing an outline, it is natural to show
the structure by indenting with the tab key. This
is particularly easy if one has a text editor with an
automatic indentation feature. With that feature,
hitting the Return key produces a new line with the
same amount of indentation as the previous line.
When the input is typed this way, we can tell the
indentation level of a topic by counting tabs. We
also need to mark the beginning of a topic, since
not every line begins a new topic. I chose to mark a
new topic with a pound sign (#). Thus, the input
to produce the outline above could look something
like: