\expandafter

这是 Stephan v. Bechtolsheim 于 1988 年在 TUGboat 上发表的一篇关于 `\expandafter` 的文章，存档于此处。另外，本文中存在一些技术性的问题，请读者留心观察，我会在最后一节指出。

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 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 `\xx` as follows: `\def\xx{ABC}`; then the replacement text of `\xx` is `ABC`, and the *expansion of* `\xx` is `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

`\expandafter` $\langle tokene\rangle$ $\langle token1\rangle$ $\langle token2\rangle$ $\ldots$ $\langle tokenn\rangle$ $\ldots$

then the following rules describe the execution of `\expandafter`:

1. $\langle tokene\rangle$, the token immediately following `\expandafter`, is saved without expansion.
2. Now $\langle token1\rangle$, which is the token after the saved $\langle tokene\rangle$, is analyzed. The following cases can be distinguished:
    1. $\langle token1\rangle$ is a *macro*: The macro $\langle token1\rangle$ 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 execute it. See 3 instead! Examples 1 – 6 and others below fall into this category.
    2. $\langle token1\rangle$ 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:

   1. ⟨$token1$⟩ is another `\expandafter`: See the section on 「Multiple `\expandafter`s」 later in this article, and also look at Example 9.
   2. ⟨$token1$⟩ is `\csname`: TEX will look for matching `\endcsname`, and replace the text between the `\csname` and the `\endcsname` by the token resulting from this operation. See Example 11.
   3. ⟨$token1$⟩ 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.
   4. ⟨$token1$⟩ is `\the`: the `\the` operation is performed, which involves reading the token after `\the`.
3. ⟨$tokene$⟩ is stuck back in front of the tokens generated in the previous step, and processing continues with ⟨$tokene$⟩.

Example 1 and 2: Macros and `\expandafter`

In these examples, ⟨$token1$⟩ is a macro. Assuming the following two macro definitions (Example 1):

```
\def\xx [#1]{...}
\def\yy{[ABC]}
```

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

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

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

```
\def\zz {\yy}
```

Observe that `\expandafter\xx\zz` will not work, because `\zz` is replaced by its replacement text which is `\yy`. But then `\yy` is not expanded any further. Instead `\xx` will be substituted back in front of `\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`, but work with `\edef`:

```
% Equivalent to "\def\temp{\xx[ABC]}".
\edef\temp{\noexpand\xx\zz}
\temp
```

As a side remark: Example 1 can also be programmed *without* `\expandafter`, by using `\edef`:

```
% Equivalent to "\def\temp{\xx[ABC]}".
\edef\temp{\noexpand\xx\yy}
\temp
```

Example 3

In this example, $\langle token1 \rangle$ is a definition.

```
\def\xx{\yy}
\expandafter\def\xx{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{This is fun}
```

Example 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}

% \PickSecondOfTwo
% ===============
% #1 and #2 of \PickFirstOfTwo
% are reversed in their role.
% #1: drop this argument
% #2: repeat this argument
\def\PickSecondOfTwo #1#2{#2}
```

Here is an application of these macros (Example 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 \read, or a mark.
\def\a{{First part}{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 `{First part}{Second part}`The two strings inside curly braces generated this way form the arguments of `\PickFirstOfTwo`, which is re-inserted in front of `{First part}{Second part}`. Finally, the macro call to `\PickFirstOfTwo` will be executed, leaving only `First part` on the main token list.

Naturally the above `\Pick...` macros could be extended to pick out x arguments from y arguments, where $x \leqslant y$, 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\stream` to `\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 `\read\stream` to `\InLine` is equivalent to `\def\InLine{12}`, the second one to `\def\InLine{13}` and so forth. The space ending the replacement text of `\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 `\InLine`matches the space which forms the delimiter of the first parameter of `\temp` in the following. Here, the macro `\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 can not 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` can not 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 `\expandafters`

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 ii-th `\expandafter`. Then

$\backslash ex_1$ $\backslash ex_2$ $\backslash ex_3$ `\a` $\backslash ex_4$ `\b` `\c`

reverses the expansion of all three tokens. Here is what happens:

1. $\backslash ex_1$ is executed causing $\backslash ex_2$ to be saved. Then TEX looks at $\backslash ex_3$.
2. $\backslash ex_3$ is another `\expandafter`: **TEX will continue performing `\expandafters`!**
3. `\a` is saved (the list of saved tokens is now $\backslash ex_2$ - `\a`).
4. $\backslash ex_4$ is now executed: `\b` is saved (the saved token list is now $\backslash ex_2$ - `\a` - `\b`)and `\c` is expanded.
5. Now *all* saved tokens $\backslash ex_2$ - `\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 $\backslash ex_2$, 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 `\expandafter`s 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\a
% 3 \expandafters
\ex\ex\ex\b
% 1 \expandafter
\ex\c
\d
```

In general, to reverse the expansion of nn tokens $\langle token1 \rangle$ ... $\langle tokenn \rangle$, the ii-th token has to be preceded by $2^{n-i} - 1$ `\expandafter`s.

Example 8: Forcing the partial expansion of token lists of `\write`s

`\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 `\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\ex{\x\y\z}
```

Going back to our explanation of multiple `\expandafter`s: `\ws` corresponds to `\a`, `{` to `\b`, and `\x` to `\c`. In other words `\x` will be expanded (!!), and `{` 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 TTXXTT
     \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 #1\aaa#2\bbb#3\Del{#2}
% Call macro to extract substring
% from \xx.
% Prints "TTXXTT".
\expandafter\extract\xx\Del
% which is equivalent to:
\extract This is fun\aaa TTXXTT
   \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 TTXXTT.

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 #1abc#2\@Del{...}
```

Now look at the following source:

```
\def\xx{AABBCC}
% #1 of \@TestSubS is AABBCC
\expandafter\@TestSubS\xx abc\@Del
\def\xx{AABBabcDD}
% #1 of \@TestSubS is AABB
\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 decomes 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`.

```
\catcode`@ = 11
% This conditional is needed because
% otherwise we would have to call the
% following macro \IfNotSubString.
\newif\if@TestSubString
% \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\@MainString{#1}%
  \def\@TestSubS ##1#2##2\@Del{%
     \edef\@TestTemp{##1}}%
  \expandafter\@TestSubS
     \@MainString#2\@Del
  \ifx\@MainString\@TestTemp
     \@TestSubStringfalse
  \else
     \@TestSubStringtrue
  \fi
  \if@TestSubString
}
\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\endcsname{...}
```

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 applications of `\expandafter`. Some were presented to 「exercise your brain 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.

注

作者在文中提到的 Another Look At TEX 这本书应该又名为 TEX in Practice，分四卷：

- Volume I: Basics
- Volume II: Paragraphs, Math and Fonts
- Volume III: Tokens, Macros
- Volume IV: Output Routines, Tables

本文章所讲述的内容位于第三卷之中。下面指出文章中的问题：

1. 在第二节中，如果 token1 为花括号的话，TEX 会试图展开 `{`，然而任何 character token 都是不可展开的，所以在这种情况下 `\expandafter` 没有任何作用。如果 tokene 为 `{` 的话，TEX 会尝试展开花括号后面的 token (以及该 token 可能附带的参数)，一个经典的例子是 `\uppercase\expandafter{\romannumeral3}`， TEX 先将 `{` 后面的 `\romannumeral3` 展开为 `iii`，然后执行 `\uppercase{iii}` 得到 `III`。
2. 同样在第二节中，作者的叙述容易误导读者以为可以展开的 primitives 只有 `\the`、`\csname` 和 `\expandafter`，事实上，所有的 *conditionals* (包括 `\if` 系列、`\else`、`\fi`) 以及 `\number` 都是可展开的。