Memoize

Sašo Živanović

✉ saso.zivanovic@guest.arnes.si
🖥 spj.ff.uni-lj.si/zivanovic
⌂ github.com/sasozivanovic

Memoize is a package for externalization of graphics and memoization of compilation results in general, allowing the author to reuse the results of compilation-intensive code. Memoize (i) induces very little overhead, as all externalized graphics is produced in a single compilation. It features (ii) automatic recompilation upon the change of code or user-adjustable context, and (iii) automatic externalization of Ti*k*Z pictures and Forest trees, easily extensible to other commands and environments. Furthermore, Memoize (iv) supports cross-referencing, Ti*k*Z overlays and Beamer, (v) works with all major engines and formats, and (vi) is adaptable to any workflow.

The two steps of externalization of graphics in Memoize

doc.tex

```
\documentclass{article}
\usepackage{memoize}
\usepackage{tikzlings}

\begin{document}

We all love Ti\emph{k}Zlings!
\tikz\penguin; is a penguin, \tikz\koala; is a koala, and \tikz\owl; is an owl.

\end{document}
```

doc.pdf

We all love Ti*k*Zlings! 🐧 is a penguin, 🐨 is a koala, and 🦉 is an owl.

.pdf   **.pdf**   **.pdf**

Using the externalized graphics

doc.pdf

We all love Ti*k*Zlings! 🐧 is a penguin, 🐨 is a koala, and 🦉 is an owl.
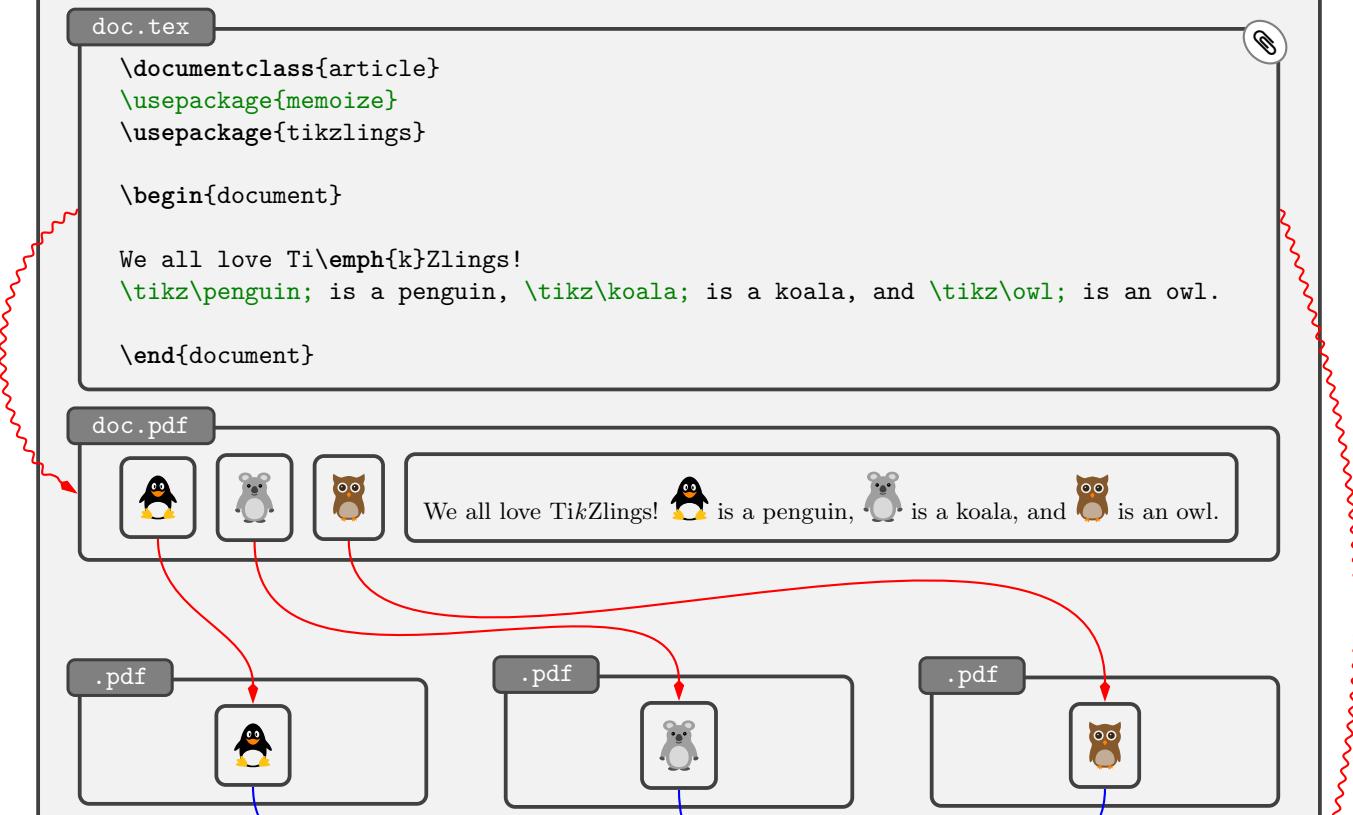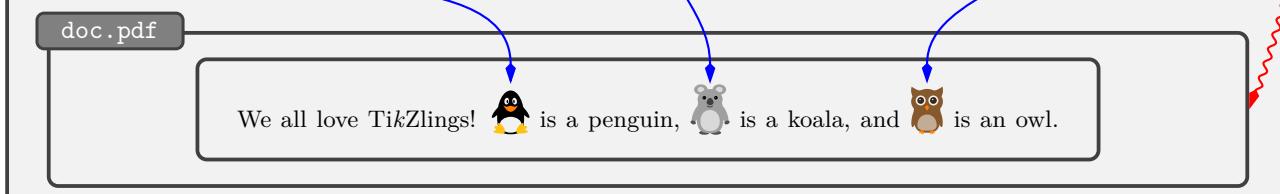
This manual also documents packages Advice (v1.1.1) and CollArgs (v1.2.0). These are auxiliary packages which were developed alongside Memoize, but are distributed as independent packages as they might be useful outside the context of Memoize, as well. See sections 4.5.1 and 5.6.1 for Advice, and sections 4.5.2 and 5.6.3 for CollArgs.

Introduction

What is externalization and why you might want it?

If you have ever worked on a long document full of TikZ pictures and maybe Forest trees, you have probably had some compilation-enforced coffee breaks — even on modern computers, compiling pictures, trees and such takes a lot of time. And you might have wondered, why do I need to compile these pictures over and over again? — after all, I'm not changing them anymore! Enter *externalization*, a mechanism designed to deal precisely with your issue, by saving the produced pictures into separate PDFs and including those PDFs in subsequent compilations — in no time at all!

Why yet another externalization package?

TikZ, the popular and all-powerful graphics language for TeX, ships including an externalization library (described in §53 of the TikZ & PGF manual). TikZ's library does an excellent job, but with one caveat. Assume you're using it for the first time (or after a clean-up) on that long document full of TikZ pictures and Forest trees. It will take ages to produce all the externalized graphics. Why? Because to get you up to speed, your document has to be compiled many many times — once for each and every externalized picture. Even with TikZ's advanced mechanism for skipping the parts of the document irrelevant for the picture at hand, the first externalization can be a daunting task.

How does Memoize save your time?

There is a reason why TikZ uses an entire compilation cycle to produce a single externalized picture: TeX itself can only produce a single PDF per compilation (at least at the moment). Memoize evades this limitation by dumping the externalized pictures right in the middle of the document (ouch!). More precisely, an externalized picture occurs in the PDF twice, first on a special page of its own and then on a regular page, where you intended it to be. The daring dump obviously necessitates a second step of the procedure, when those special pages are *extracted* into separate PDFs, called *externs*, which are then included into the document in subsequent compilations.

This two-step procedure, illustrated on the cover page of this manual, is very fast. The first step, which externalizes *all* the pictures into the document itself (the squiggly red arrow), takes virtually no more time than a regular compilation. The time needed for the second step, extraction (the normal red arrows), depends on the system setup, but it ranges from little to almost none.

When should I use Memoize?

In short, whenever you are writing a document containing lots of TikZ pictures, Forest trees, or other time-consuming constructs, and you are bored waiting for the compilation to finish.

Using Memoize on a paper containing a single picture does not make much of a difference. But with more complex documents, the speed-up can be immense. For example, the compilation time of my 400-page book containing about 160 Forest trees was reduced by more than half, and the compilation time of a 260-page Beamer presentation with a hundred complex dynamic trees went from four minutes to a mere half minute!

How much extra work does Memoize require?

In principle, none.

For one, while allowing for manual memoization of selected document chunks (§2.2), Memoize features a system which automatically triggers memoization at each invocation of selected commands and environments. Out of the box, Memoize *automemoizes* Ti*k*Z pictures and Forest trees, but the author can easily submit (almost) any command or environment to automemoization (§2.3). Memoize also does its best to automatically prevent memoization of code that cannot be externalized, like Ti*k*Z pictures with `remember picture`, and to abort memoization in case the memoized code yields any errors.

Why is Memoize not called Externalize?

Fundamentally, Memoize is about producing and utilizing *memos* — pieces of TeX code replicating the effect of the compilation of a document chunk in a computationally less intensive manner. Typically, each memo has an associated extern, which is where the effect of *typesetting* is stored, but conceptually, memos come first. For example, the extern is included back into the document by the memo, and a memo may be associated with any number of externs, including zero.

Memos solve several externalization-related problems in a generic fashion, allowing for a multitude of applications. For example, they store the information about the associated externs, so that an extern can be integrated back into the document as a box with the original orientation, width, height and depth. They solve the problem of cross-referencing from and into the memoized code by storing its *context* (§3.3) and replicating any `\label`s which occur in it. They are also crucial for externalizing pictures in Beamer frames overlay by overlay.

Incidentally, the term "memoization" is used with some programming languages to refer to the process of remembering the result of the function, along with the given arguments, so that on subsequent invocations of the function with the same arguments, the result can be returned from memory rather than recomputing it. I would say that, give or take the functions, what Memoize does fits the bill.

How can I make my command (auto)memoizable?

Any command which interacts with the rest of the document, like a command which produces a float, must receive special treatment. Some issues can be resolved from within Memoize. Other issues require a memoization-compatible (re)implementation of the command. It is in the hope that package writers will adapt their "difficult" commands to Memoize that this package offers a documented interface to the memoization process, fully described in sections 3.5, 4.4 and 4.5.

An advanced user might also want to know that Memoize ships with two auxiliary packages, which form the base of Memoize's automemoization feature. Package Advice implements a generic framework for extending the functionality of selected commands and environments, while package CollArgs provides a command which can determine the argument scope of any command whose argument structure conforms to LaTeX3 argument specification.

What else is out there?

Not long before submitting Memoize to CTAN, I became aware of another new externalization package, `robust-externalize`, and it seems that the same happened to the author of that package ☺, who found the proof-of-concept version of Memoize, which was available at GitHub for a while.

The key idea behind `robust-externalize` seems to be to extract the code submitted to externalization into separate files, and add the necessary preamble. While a compilation from scratch takes more time than with Memoize (but less than with Ti*k*Z library), the approach allows for parallel compilation of externs — nice!

Contents

1 Before you start

1.1 Installing the extern extraction software

Good news: using Memoize can be as easy as writing `\usepackage{memoize}` in the preamble.

Bad news: Memoize won't work out of the box. The culprit is the extern extraction — the process which ships the externalized graphics from the main document into separate extern files; for details, see the title page illustration and the "How" box in the Introduction. For the extraction to work, you will probably have to install some additional software, and you might also have to allow your TeX to execute the extraction script. But there's a silver lining: once Memoize is set up, it is set up for good.

> **What do I have to do?**
>
> In principle, all you have to do for Memoize to work *under the default configuration* is install Perl library `PDF::API2`; see the Perl section below for the installation guideline.

Consult section 3.2 if you want to use an extraction method other than the default `perl`-based method or adapt Memoize to your particular workflow (for example, if you're compiling via a Makefile).

If you installed Memoize through the package manager of your TeX distribution, your system should be already set up to allow the execution of Memoize's extraction scripts. If this is not the case, please contact either me or the maintainer of your distribution; until the issue is resolved, you have to either

    (a) compile documents loading Memoize with command-line option `-shell-escape` (on TeXLive) or `--enable-write18` (on MiKTeX), or

    (b) set up the restricted shell escape mode to allow for the execution of `memoize-extract.pl`.[1]

Once you have set up your system, I advise you to follow the instructions in section 2.1 to test if the setup was successful.

Perl   If you're running GNU/Linux or macOS, Perl is most likely already installed on your system. On Windows, you might have to install it. I tested Memoize with Strawberry Perl, available at `strawberryperl.com`; see `www.perl.org` for other options. On TeXLive on Windows, you should also set `TEXLIVE_WINDOWS_TRY_EXTERNAL_PERL=1`.

Once Perl is installed on your system, you will also need to install the PDF processing library `PDF::API2` (or its fork, `PDF::Builder`). On some

```
cpan PDF::API2
```

GNU/Linux distributions, this library is included as a package — just use your package manager to install it. Otherwise, install it from `CPAN` using `cpan` tool, as shown in the box.

Python   Installing (Python and) the required Python library is only necessary if you decide to use the Python-based extraction script; see section 3.2.

If you're running GNU/Linux, install Python using your package manager. Otherwise, download the installer from `www.python.org`. You can install Python even if you don't have administrator privileges: simply uncheck the "Install launcher for all users" when running the installer.

Once Python is installed on your system, you will also need to install the PDF processing library `pdfrw2` library (or its predecessor, `pdfrw`, which

```
pip install pdfrw2
```

will work just as well). Some GNU/Linux distributions offer this library as a package; if this is not the case, and on other operating systems, install it from The Python Package Index using pip tool (if you run pip as a superuser, it will install the library system-wide, otherwise locally), as shown in the box.

---

[1]On TeXLive, `tlmgr conf texmf shell_escape` will tell you which shell escape mode is currently in effect (p = partial = restricted, f = disabled and t = enabled); if necessary, use `tlmgr conf texmf shell_escape p` to set it to restricted. Then, execute `tlmgr conf texmf shell_escape_commands` to get the list of currently allowed commands ⟨*current*⟩, then add the script by executing `tlmgr conf texmf shell_escape_commands ⟨current⟩,memoize-extract.pl`.

    On MiKTeX, the shell escape mode can be reported by `initexmf --show-config-value=[Core]ShellCommandMode`, and restricted mode set by `initexmf --set-config-value=[Core]ShellCommandMode=Restricted`, and you get the ⟨*current*⟩ list by `initexmf --show-config-value=[Core]AllowedShellCommands[]`, and add to it by `initexmf --set-config-value=[Core]AllowedShellCommands[]=⟨current⟩;memoize-extract.pl`.

Memoize can be configured using command `\mmzset{⟨keylist⟩}` (and friends). The ⟨*keylist*⟩ argument is a comma-separated list of configuration settings, each setting having the form ⟨*key*⟩=⟨*value*⟩, or sometimes just ⟨*key*⟩. The ⟨*keylist*⟩ argument is processed by package `pgfkeys` (see §88 of the Ti*k*Z & PGF manual), so you can use all the bells and whistles of that fantastic PGF utility (like easily defining your own styles).

Here's some examples of `\mmzset`. For one, to whet your appetite to learn about the various keys in the `/mmz` path, but more importantly now, to show you that white-space is irrelevant in the ⟨*keylist*⟩ argument, so you can format the keylist as you wish — as long as it does not contain an empty line.[2]

```
\mmzset{memo dir,recompile,memoize=circuitikz}
```

```
\mmzset{memo dir, recompile, memoize=circuitikz}
```

```
\mmzset{
    memo dir,
    recompile,
    padding=2in}
```

```
\mmzset{
    memo dir, readonly,
    memoize=\qrcode{om},
    deactivate=\label,
    disable,
}
```

```
\mmzset{
    memo dir,
    recompile,
    % comment
    padding=2in
}
```

```
\mmzset{
    memo dir,
    recompile,

    padding=2in
}
```

Command `\mmzset` can be used any time after loading the package. It is very common in the preamble, but also useful in the document body, where its effect is local to the TEX group.[3] For example, you can use the idiom on the left below to force recompilation of a single Forest tree somewhere in the middle of the document (in the code listings below, highlighting marks the resulting scope of the `recompile` directive). However, as applying some setting to a single piece of automatically memoized code is common, Memoize provides a special command for the occasion: the keys given as the argument of `\mmznext` will be applied only at the next instance of automemoization, overriding any keys set by `\mmzset` in case of a conflict. If the command is given more than once, only the final invocation takes effect.

```
{
    \mmzset{recompile}
    \begin{forest}
      [VP[V][DP]]
    \end{forest}
    % ...
}
```

```
% ...
\mmznext{recompile}
\begin{forest}
  [VP[V][DP]]
\end{forest}
% ...
```

I like to follow `\usepackage{memoize}` by `\mmzset`, but if you prefer, you can also provide the document-wide configuration as package options. For example, the following are equivalent — both will force re-externalization of all the externs in the document.

```
\usepackage{memoize}
\mmzset{recompile}
```

```
\usepackage[recompile]{memoize}
```

---

[2]I like to add a comma after the final key as well, as shown in the bottom left example, because if I don't, I often forget to insert it when I add more keys.

[3]Any keys with a non-local effect are explicitly marked as such in the reference section.

Memoize allows you to configure settings which apply to more than just one document. It does that by attempting to load file `memoize.cfg` just before executing package options. Given how TeX searches for files, the location of this file determines whether it applies system-wide, user-wide or directory-wide. The directory-wide location is clearly the directory itself. The user-wide and system-wide location depend on the TeX distribution and which format(s) you want to use `memoize.cfg` with:[4]

⟨*the relevant texmf tree root*⟩`/tex/`⟨*format*⟩`/memoize/memoize.cfg`

You can say `generic` for ⟨*format*⟩ if you want the configuration file to be accessible by all formats, otherwise ⟨*format*⟩ should be one of the formats supported by Memoize: `plain`, `latex` or `context`. The texmf root directory depends on the distribution, here's how to figure out what it is:

The roots of TEXMF trees

TeX Live   tug.org/texlive

user-wide `tlmgr conf texmf TEXMFHOME` or `kpsewhich -var-value TEXMFHOME`
      the default (on Linux): `/home/`⟨*username*⟩`/texmf`

system-wide `tlmgr conf texmf TEXMFLOCAL` or `kpsewhich -var-value TEXMFLOCAL`
      the default (on Linux): `/usr/local/texlive/texmf-local`

Don't forget to run `texhash` or `mktexlsr` after creating `memoize.cfg`.

MiKTeX   miktex.org

Open the MiKTeX Console, select the "Settings" page and then the "Directories" tab.

If there is a folder marked with the "Generic" purpose with attribute "User" (for a user-wide `memoize.cfg`) or "Common" (for a system-wide `memoize.cfg`), that's the folder you are looking for. Otherwise, create a folder following MiKTeX's instructions and add it to the list.

Don't forget to click "Refresh file name database" (in the "Tools" menu of the MiKTeX Console) after creating `memoize.cfg`.

Note that a directory config file will override the user config, and the user config will in turn override the system-wide one. This should not concern you too much, because you will probably only want to use the user-wide config anyway,[5] which might look something like this:

```
memoize.cfg
\mmzset{
  memo dir,                    % Put the memo and extern files into a subdirectory.
  extract=python,              % Use the Python-based extern extraction method.
  memoize=circuitikz,          % Are you working on electrical circuits all the time?
}
```

I recommend including `memo dir` in your `memoize.cfg`, as shown in the first line. It reduces the clutter in the document directories; see section 2.5 for details.

The `extract` line concludes the story on "permanently" selecting the extraction method, started in chapter 1. As the final note, observe that `extract` and other extraction-related keys like `perl extraction options` only make sense as a package option or as a `\mmzset` key in `memoize.cfg`. They will have no effect as a `\mmzset` key in the document, because the extraction happens while the package is loaded.

---

[4]The `memoize` subfolder is not obligatory.

[5]If you want to have a directory-wide configuration based on (rather than overriding) the user-wide configuration, your could write down the real user-wide config in, say, `memoize.user.cfg` (located user-wide), and then `\input` this file by both the user-wide and the directory-wide `memoize.cfg`. Of course, the same logic can be used to base a user-wide config on the system-wide one.

2 Your first memoized documents

2.1 Let's see if it works!

Take example file `test.tex`,[6] or some simple document containing a Ti*k*Z picture or a Forest tree, add `\usepackage{memoize}` to the preamble,[7] and compile it twice.

- The first compilation of the example should produce a three-page PDF. The first two pages are the *extern pages* holding the externalized graphics, while the final page is the (sole page of the) real document. Note that you can see each extern twice: first on a page of its own, and then wherever it belongs to in the real document.

- At the second compilation, the extern pages should have disappeared from the PDF, meaning they were successfully extracted into extern files, which are now embedded into the main document.

```
test.tex
\documentclass{article}
\usepackage{tikz,forest}

\usepackage{memoize}

\begin{document}

Memoize will automatically externalize\\
\begin{tikzpicture}
  \node(node)
    [align=center,ellipse,draw]
    {TikZ\\pictures};
\end{tikzpicture}
\begin{forest}
  [and Forest trees [and]
    [Forest trees[Forest][trees]]]
\end{forest}

\end{document}
```
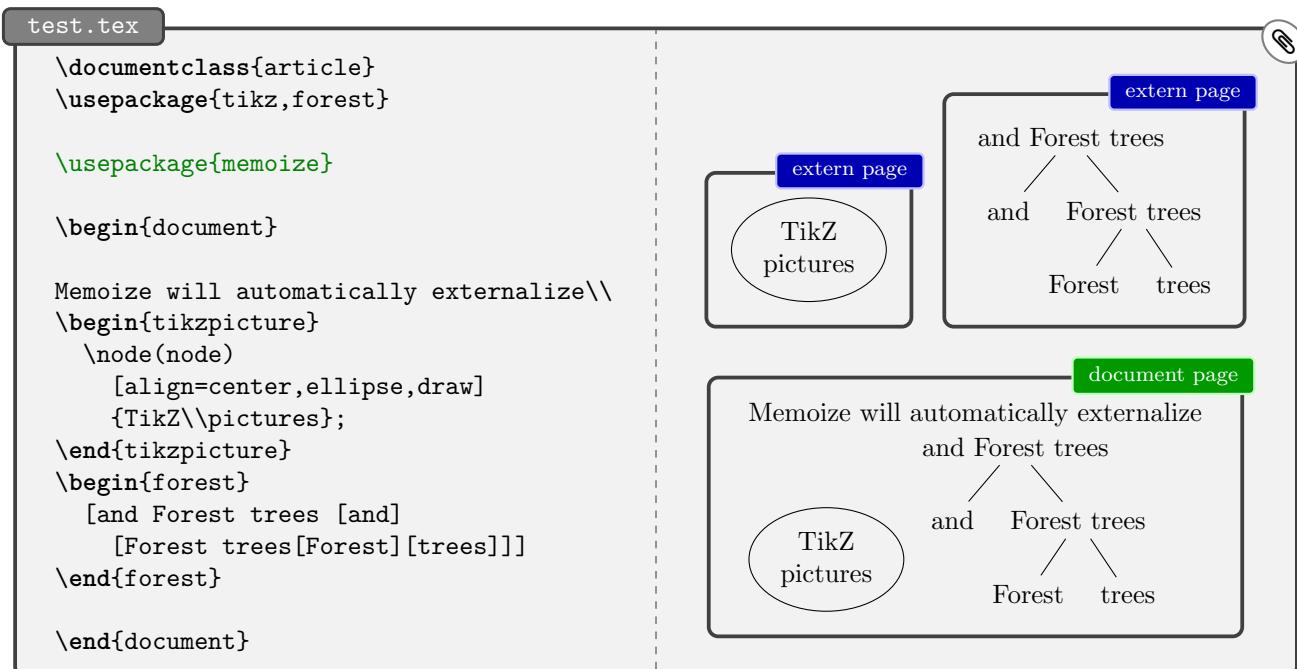
You might want to play with this example a bit now. For example, if you reverse the order of the Ti*k*Z picture and the Forest tree, you should notice that the externs don't get recompiled. You won't see any extern pages again until you change the actual code of the picture or the tree — or until you add some other picture or tree, of course.

If you *don't* want to automatically memoize Ti*k*Z pictures and/or Forest trees, you can switch off their *automemoization* using key `deactivate`. This key takes a list of command and environment names. As you can see below, the command and the environment must be deactivated separately.

```
\mmzset{
  deactivate={\tikz, tikzpicture},  % deactivate automemoization of all TikZ pictures
  deactivate=\tikz,                 % deactivate only the command
  deactivate=tikzpicture,           % deactivate only the environment
}
```

---

[6] Where can you find the example files? For one, they are integrated into this manual, so if your PDF viewer supports attachments, you can simply click on the paperclip icon on the top right of the example box (even if you're offline). Otherwise, visit the `examples` subdirectory of wherever you found this document ☺. Online, the Memoize documentation can be found at CTAN: `https://ctan.org/pkg/memoize`; and if your TeX installation includes the documentation, you should also find it in directory ⟨*the root of your TeX installation*⟩`/doc/generic/memoize`.

Incidentally, while we present a full example document in this section, many code listings will only present the parts of the file relevant for the discussion, for brevity. The example *files*, however, will remain full, compilable documents, including the document preamble etc.

[7] Memoize likes to be loaded early. If you get the warning `Cannot read file` ⟨*jobname*⟩`.pdf`, move `\usepackage{memoize}` up the preamble; see section 6.3 for details.

2.2 Memoizing by hand

In the previous section, we have compiled our very first document which used Memoize. In that document, we only had to load the package, as Memoize knows how to externalize TikZ pictures and Forest trees without any help from the author. But what if you want to externalize some other code? The manual way of doing this is by surrounding the code by a `memoize` environment, or by making it the argument of the `\mmz` command. The only difference between the two is that the environment, but not the command, ignores any spaces surrounding the given code.

```
manual.tex
\begin{memoize}
  \begin{circuitikz}
    \draw (0,0)
          to[isource] (0,3) -- (2,3)
          to[R] (2,0) -- (0,0);
  \end{circuitikz}
\end{memoize}
% ...
\mmz{\qrcode{https://ctan.org/pkg/memoize}}
```
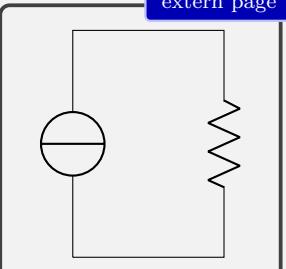
Both the `memoize` environment and the `\mmz` command take a configuration keylist as the optional argument, so their full syntax is `\begin{⟨memoize⟩}[⟨keylist⟩]⟨code to be externalized⟩\end{⟨memoize⟩}` and and `\mmz[⟨keylist⟩]{⟨code to be externalized⟩}`. The keys given in this optional argument take precedence over the keys set by `\mmzset`. Note that `\mmznext` does not apply to manual memoization.

Manual memoization is great for one-shot memoizations, but you can use it within your own macros as well. For example, assume that you don't want to externalize TikZ pictures in general (so you have `deactivate`d automemoization of the `\tikz` command, as explained at the end of section 2.1), but that you want to easily memoize selected pictures. You could define a memoized variant of the `\tikz` command, as shown below (and similary for the environment). (Note the `%` comment characters in the definition of `\mmztikz`. The definition was intentionally broken into several lines to remind you that the spaces around the argument of `\mmz` matter.)

```
mmztikz.tex
\mmzset{deactivate=\tikz}
\newcommand{\mmztikz}[2][]{\mmz{%
  \tikz[#1]{#2}%
}}
Compare \tikz{\node{this picture which \emph{won't} be externalized};} to
\mmztikz{\node{this picture which \emph{will} be externalized}}
```

2.3 Memoizing automatically

Out of the box, Memoize automatically externalizes Ti*k*Z pictures and Forest trees. Let us see how other commands and environments can be submitted to this *automemoization* process.

We start with the simpler case of environments (fortunately, externalizing environments also makes sense more often than externalizing commands). You can submit an environment to automemoization by writing `auto={`⟨*environment name*⟩`}{memoize}` (as a key in `\mmzset`). The natural (but not the only possible) location for this instruction is the preamble. Below, we automemoize environment `circuitikz` of package `circuitikz`, used for drawing electronic circuits.[8]

```
automemoize-environment.tex

  \mmzset{auto={circuitikz}{memoize}}
  % ...
  \begin{circuitikz}
    \draw (0,0) to[isource] (0,3) -- (2,3)
    to[R] (2,0) -- (0,0);
  \end{circuitikz}
```

Commands are a bit harder to automemoize, because Memoize cannot possibly know how far the arguments of a command extend (in contrast, the end of an environment is clearly marked). With commands, we must inform Memoize about their argument structure, which we achieve using key `args` in the second argument of key `auto`: `auto=`⟨*command*⟩`{memoize, args={`⟨*argument specification*⟩`}}`.

And how does the ⟨*argument specification*⟩ look like? It is a sequence of letters, each letter determining an argument type. Memoize recognizes the same argument types as `\NewDocumentCommand`, the LATEX3 command for defining new commands, available in core LATEX since 2020. For the complete story, see the reference for `args` in section 5.6.1 and the frame on page 113 in section 5.6.3; here, we focus on the two most commonly used types, `m` and `o`, and add the optional star for good measure:

letter	argument type	example
m	mandatory argument	
	— either surrounded with braces	`\foo{arg}`
	— or a single token	`\foo a`
o	optional argument, surrounded with brackets	`\foo[arg]`
s	optional star	`\foo*`

Below, we write `args=om` because command `\qrcode` (of package `qrcode`) takes two arguments: a bracketed optional argument, followed by a mandatory argument (in braces).

```
automemoize-command.tex

  \mmzset{auto=\qrcode{memoize, args=om}}
  % ...
  \qrcode{https://ctan.org/pkg/memoize}
  \qrcode[height=1cm]{https://ctan.org/pkg/memoize}
```

If the command we're submitting to automemoization was defined by `\NewDocumentCommand` or similar, `args` may be omitted, as in this case, Memoize can retrieve the argument specification on its own.

This should get you started with automemoization. We'll provide some further basic information in sections 2.4 and 2.9 of the tutorial, but only package writers will probably ever need the gory details from section 4.5. It is my sincere hope that they will support Memoize in their packages, where necessary, so that you don't ever have to write the `auto` declarations. However, there is one thing you should know if you encounter a package supporting Memoize: you should load it *after* Memoize!

---

[8]In section 2.1, we learned that automemoization can be switched off using key `deactivate`. Memoize also offers key `activate`, but you probably won't have to use it, as an `auto` declaration automatically activates the submitted command.

Memoize automatically recompiles a picture when the code producing the picture changes. However, sometimes we can modify a picture without changing its code, like when we modify the definition of a command used in the code. In the example below, a predefined style `emph` is applied to the node, producing a node with a red background. Let's say we compile the document (with memoization) and then change this style to set the yellow background. Curiously, the node will remain red.

```
recompile.tex (version 1)

\tikzset{
  emph/.style={fill=red, text=blue},
}
\begin{tikzpicture}
  \node[emph]{an emphasized node};
\end{tikzpicture}
```

document page

an emphasized node

```
recompile.tex (version 2)

\tikzset{
  emph/.style={fill=yellow, text=blue},
}
\begin{tikzpicture}
  \node[emph]{an emphasized node};
\end{tikzpicture}
```

document page

an emphasized node
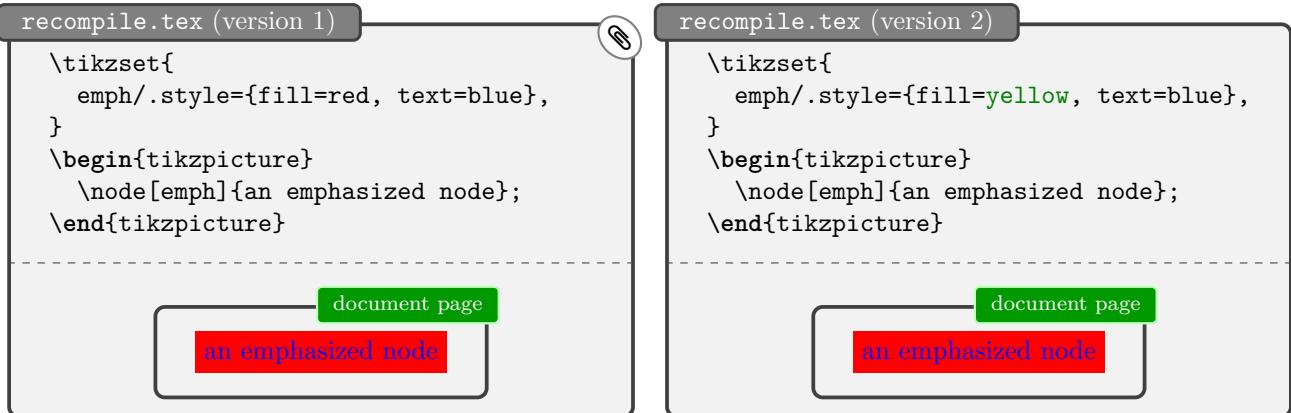
The curious thing happens (or rather, doesn't happen) because Memoize doesn't keep track of how commands and styles are defined; it just uses the extern file it created when the old style was in effect. To get a yellow node, we must ask Memoize to reexternalize the picture. The simplest way to do that is by using the `recompile` key; below, we write `\mmznext{recompile}` just before the picture and compile the document again (remember from section 1.2 that whatever keys we provide through `\mmznext` only apply to the instance of automemoization). After the compilation, we may (and should) remove the `recompile` directive (otherwise, Memoize will produce the extern page again and again).

```
recompile.tex (version 3)

\tikzset{emph/.style={fill=yellow, text=blue}}
\mmznext{recompile} % for one compilation
\begin{tikzpicture}
  \node[emph]{an emphasized node};
\end{tikzpicture}
```

document page

an emphasized node
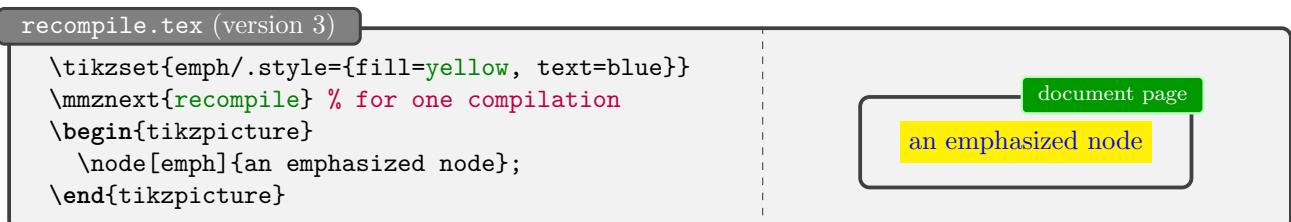
It is also common to put (again, for the space of a single compilation) `\mmzset{recompile}` in the preamble, or to use `recompile` as the package option. Either will remake all the externalized graphics in the document, so you can be sure all of them use the latest version of your macros and styles.

We'll revisit the issue of memoized code depending on macros and styles defined elsewhere in section 3.4. In this section, we will learn how the issue can be *avoided*, at least to some extent. One idea is to turn off memoization for the picture(s) we are currently working on; another idea is to let Memoize know which definitions the picture relies on. The simplest way to achieve the former is by using key `disable`; by putting it into a TeX group, we can localize its effect to the selected pictures.

```
disable.tex

\tikz\node[draw=green]{An externalized node.};
{
  \mmzset{disable}
  \tikz\node[draw=red]{
    This node is not externalized.};
  \tikz\node[draw=red]{
    And neither is this one.};
}
\tikz\node[draw=green]{
  Another externalized node};
```

extern page

An externalized node.

extern page

Another externalized node

document page

An externalized node.
This node is not externalized.
And neither is this one.
Another externalized node
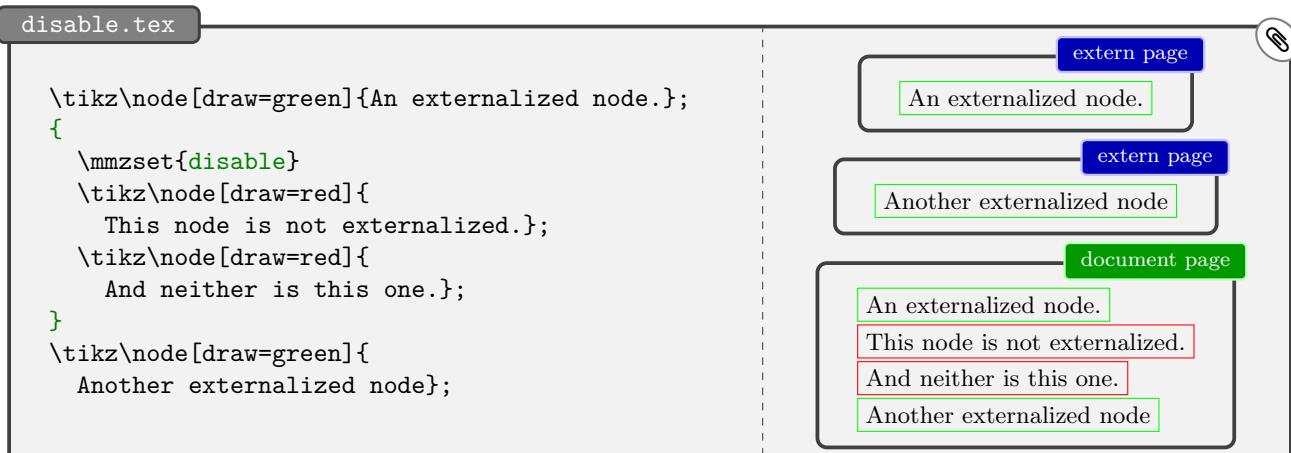
As you can imagine, key `disable` is complemented by `enable`, but it is perhaps worth mentioning a problem that can arise if you disable memoization for a part of your document by enclosing it in a pair of `\mmzset{disable}` and `\mmzset{enable}`. Yes, it might work at the moment, but say you later (e.g. when you are preparing the final version of the document) decide to disable memoization for the entire document, and say you try to do this by writing `\mmzset{disable}` in the preamble. As shown below on the left, you're in for a surprise: memoization will still be enabled in the part of the document following `\mmzset{enable}`! The solution is to always disable memoization for a part of the document by using `\mmzset{disable}` in a TeX group (i.e. the braces), as shown on the right. (In the examples below, the shaded areas mark the parts of the document where memoization is *disabled*.)

disable-bad.tex

```
\usepackage{memoize}
\mmzset{disable}
\begin{document}
% ...
\mmzset{disable}
% ...
% ...
\mmzset{enable}
% ...
\end{document}
```

The upper `\mmzset{disable}` does not have the intended effect, i.e. it doesn't apply to the whole document!

disable-good.tex

```
\usepackage{memoize}
\mmzset{disable}
\begin{document}
% ...
{
    % ...
    \mmzset{disable}
    % ...
}
% ...
\end{document}
```

The upper `\mmzset{disable}` applies to the entire document, as expected.

In fact, it might be better to disable memoization using environment `nomemoize` or macro `\nommz`. I also like these commands because it is easy to add and remove prefix `no` to switch manual memoization (triggered using environment `memoize` or macro `\mmz`) off and on.

disable-nomemoize.tex

```
\begin{nomemoize}
    % ...
\end{nomemoize}
```

Disable using the dedicated environment.

disable-nommz.tex

```
% ...
\nommz{...}
% ...
```

Disable using the dedicated command.

It is also possible to disable memoization for all occurrences of a selected command or environment. In fact, we're already familiar with the procedure from section 2.3, where we used key `memoize` inside the second argument of `auto` to automatically memoize all instances of the command or environment given as the first argument. All we have to do to auto-disable rather than auto-memoize, is substitute `nomemoize` for `memoize`. Note that this prevents memoization of not only the given command or environment, but also of any (manual or automatic) memoization which would otherwise occur during its execution; for example, if `\foo` executes `\tikz` under the hood, autodisabling `\foo` prevents memoization of the inner `\tikz`, even though that command is normally automemoized.

disable-auto-cmd.tex

```
\mmzset{auto=\foo{args=m, nomemoize}}
% ...
\foo{...}
% ...
\foo{...}
% ...
```

Autodisable within a command.

disable-auto-env.tex

```
\mmzset{auto={bar}{nomemoize}}
% ...
\begin{bar}
    % ...
\end{bar}
% ...
```

Autodisable within an environment.

All that said, Memoize actually offers a neater way to switch off the externalization for the picture I'm currently working on. The `readonly` key instructs Memoize to use whatever externs it had already

produced (thereby reducing the document compilation time), but to abstain from producing any new externs. In effect, the stuff we are currently working on does not undergo memoization and therefore does not produce the clutter which can potentially lead to trouble described in the `recompile` examples above.

What I like to do is load the package using `\usepackage[readonly]{memoize}`, work on stuff, and once I'm happy with the most recent pictures, remove `readonly` from the package options for one compilation.

```
readonly.tex (work in progress)
\usepackage[readonly]{memoize}
% ...
\tikz\node{work in progress};

        document page
      work in progress
```

```
readonly.tex (the final version)
\usepackage{memoize}
% ...
\tikz\node{the final version};

      extern page              document page
   the final version         the final version
```
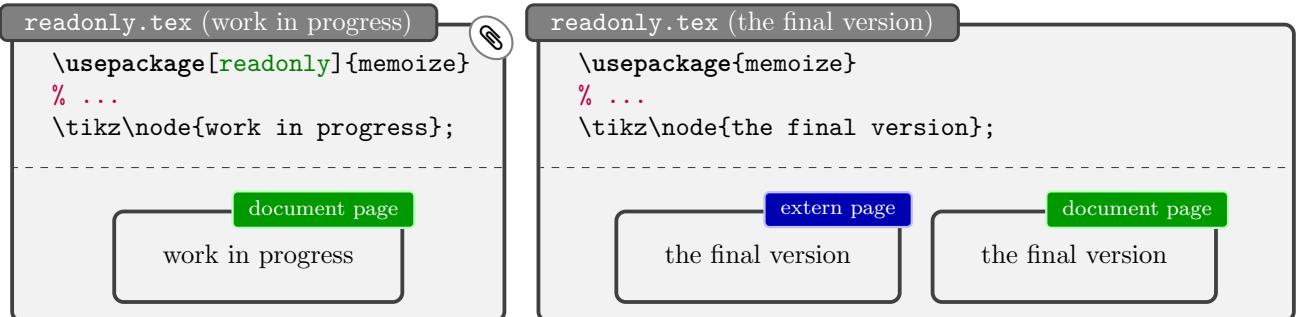
We're now ready to tackle a completely different way of avoiding the issue, by informing Memoize which definitions the externalized picture depends on. We do this by appending these definitions to *context* — when the context of a picture changes, Memoize recompiles the picture, same as if the code of the picture itself was changed. (We will talk about context in more detail in section 3.3.)

A command can be added as a dependency using key `meaning to context`. Below, we make the following externalized picture depend on the definition of macro `\answer`; changing this definition will result in the recompilation of the extern. In general, `meaning to context` accepts a comma-separated list of command and environment names, e.g. `meaning to context={\foo,bar}` (note the braces).

Memoize offers several variants of `meaning to context`, applicable to various types of commands. For example, the easiest way of making the picture depend on the definition of a `pgfkeys` style is to use handler `.meaning to context` — note the dot in the name, and observe that `emph/.meaning to context` below is executed within `\tikzset`, not `\mmzset`; see §87 of the Ti*k*Z & PGF manual to learn about key handlers.

```
meaning-to-context.tex (version 1)
\newcommand\answer{24}
\mmznext{meaning to context=\answer}
\tikzset{
  emph/.style={fill=red, text=blue},
  emph/.meaning to context,
}
\begin{tikzpicture}
  \node[emph]{The answer is \answer!};
\end{tikzpicture}

                              extern page
                          The answer is 24!
```

```
meaning-to-context.tex (version 2)
% Changing definition
\newcommand\answer{42}
% and/or
  emph/.style={fill=yellow, text=blue},
% leads to recompilation of the extern.

                              extern page
                          The answer is 42!
```
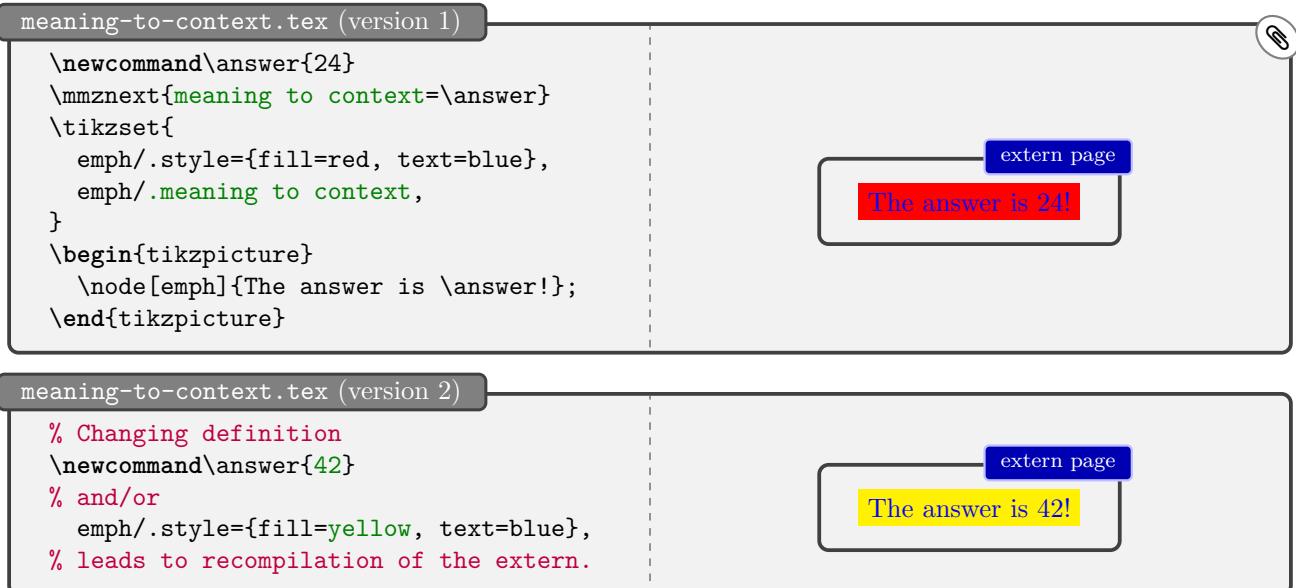
All variants of `meaning to context` (see reference section 5.3.2 for the full list) may be used within the externalized picture itself, e.g. `\node[emph, emph/.meaning to context]{...}` is perfectly valid — and also handy when you want to limit the effect of the handler to a single picture, as `.meaning to context` cannot be used within `\mmznext`.

Memoize produces lots of auxiliary files. For each piece of memoized code, it produces two *memo* files (we will learn more about these in section 4.2), which will be joined by the extern PDF upon the next compilation.[9] You can recognize these files easily: their names start with the name of your document and include one or two long hexadecimal numbers.

```
dirty-house.tex
```
                                                          the folder contents (after two compilations)

⟨the document folder⟩
— dirty-house.tex
— dirty-house.pdf   This is another auxiliary file produced
— …   by Memoize. We will mention it at the
— dirty-house.mmz ·········· end of the section.
— dirty-house.39AB85DF9887787FC044C0E10608BBBB.memo
— dirty-house.39AB85DF9887787FC044C0E10608BBBB-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.memo
— dirty-house.39AB85DF9887787FC044C0E10608BBBB-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.pdf

To top it off, changing the memoized code will produce new memo (and extern) files, with the old files staying in place. This is all by design — the first hexadecimal number in these filenames is the MD5 sum of the memoized code and that's how Memoize knows which memo belongs to which piece of code[10] — but it has the downside that the folder containing your document can get quite cluttered (imagine the directory listing as above, but for a document with a hundred externalized pictures which you have been working on for a month).

I like to keep a clean house by instructing Memoize to put memos and friends into their own directory. This can be achieved by writing `\mmzset{`memo dir`}` into the preamble (anytime after loading the package).[11] This will put the memo and the extern files into folder ⟨*document name*⟩`.memo.dir` (and it will also omit the ⟨*document name*⟩ prefix in their filenames, because it makes no sense to repeat it there).

```
clean-house.tex
    \mmzset{memo dir}
```
                                                          the folder contents (after two compilations)

⟨the document folder⟩
— clean-house.tex
— clean-house.pdf
— …
— clean-house.mmz
— clean-house.memo.dir ··········· This directory must be created, somehow.
    — 39AB85DF9887787FC044C0E10608BBBB.memo
    — 39AB85DF9887787FC044C0E10608BBBB-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.memo
    — 39AB85DF9887787FC044C0E10608BBBB-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.pdf

---

[9]If you're using the TeX-based extraction, each extern (`.pdf`) is also accompanied by a log file (`.log`) produced during the compilation that extracted the extern.

[10]The second hexadecimal number in the memo and extern filenames is the MD5 sum of the *context*. The context is crucial for properly externalizing code containing cross-references, see section 3.3 for details.

[11]The `memo dir` key is in fact merely an abbreviation for `prefix=\jobname.memo.dir`; use this key if you need more control over the name and location of the auxiliary files. Furthermore, there is also the `no memo dir` key, which reverts the configuration back to the dirty default.

> **Why is `memo dir` not the default?**
>
> The `clean-house` example most likely compiled just fine, and you are wondering why `memo dir` is not in effect by default. Well, out of the box, TeX cannot create directories, so it is the fact that Memoize *can* create them (at least under the default settings) which requires explanation. By default, Memoize triggers extraction by executing the Perl extraction script `memoize-extract.pl`, and it is this script which actually creates the memo directory. However, not everybody will necessarily use this script … so `memo dir` should not be the default.
>
> The Python extraction script `memoize-extract.pl`, used when `extract=python`, works the same as the Perl variant. With TeX-based extraction `extract=tex`, things are different. If you are compiling the document with a full shell escape mode (`-shell-escape`), Memoize successfully creates the directory with the system command `mkdir`.[a] However, if you're using the restricted shell escape mode, the attempt to create the directory won't succeed unless you include `mkdir` among the restricted shell escape commands (see footnote 1 on page 6 for how to do this, but note that it is not recommended).
>
> If you are using external extraction, you have to create directory `clean-house.memo.dir` by hand, prior to the first compilation of the document (with Memoize). This is the case even if you are performing the extraction using one of the shipped extraction methods, and it is due to the fact that Memoize needs the memo directory to be present even before extern extraction, because it writes the `.memo` files into the same directory. (When Memoize uses the extraction script to create the memo directory, it does so completely independently of extraction, and prior to creating any `.memo` files.)
>
> ─────────────────────
>
> [a]`mkdir` is the default value of key `mkdir command`, but executing the extraction method `perl` or `python` overrides this default.

I actually suggest adding `\mmzset{memo dir}` into your user-wide `memoize.cfg` (see section 1.3 for details on this file). This will keep all your houses clean — without work! — as Memoize will automatically use the memo directory for any document your create.

It is always safe to delete memos (`.memo`) and externs (`.pdf`s residing next to memos), in the sense that you cannot lose data this way.[12]

- Many memos and externs are typically stale anyway, i.e. they reflect some previous state of your document and are not needed anymore. These files can be deleted without any repercussions whatsoever (unless you later revert to a previous version of the document, of course). In fact, you might *want* to delete them periodically, or at least once you finish writing the document. As it is hard to figure out which memos/externs are stale, Memoize ships with a clean-up script: writing `memoize-clean.pl` ⟨*document name*⟩`.mmz` (replace `.pl` with `.py` if you use Python rather than Perl) into the command line will delete all the *stale* auxiliary files belonging to the document.

- If you delete a memo or an extern currently in use, you will trigger recompilation of their code — so deleting a memo or an extern is actually a perfectly legal alternative to using the `recompile` key![13]

It is also safe to delete the `.mmz` file (or any other kind of record file, see section 4.3) residing next to your document's `.pdf`. The `.mmz` file contains the information about which externs should be extracted from the `.pdf`. Deleting it before this is done (by default, before compiling the document again) will prevent the extraction (same as if providing the package option `extract=no`) and ultimately result in the recompilation of the externs produced in the previous run. Deleting it after the extraction will have almost no effect: it will only only prevent the clean-up script from working (the `.mmz` file also lists the currently active memos and externs, and thereby indirectly informs the clean-up script which files are stale). For further information on the `.mmz` file, see section 4.3.1.

─────────────────────

[12]The same goes for the extern `.log` files produced by the TeX-based extraction.
[13]For the users of the TeX-based extraction: deleting the `.log` file does *not* trigger recompilation.

Books and other long documents are usually produced from sources which reside in more than a single file, and to speed up the editing process, authors usually use some system which allows them to compile each chapter separately. Can Memoize — designed for virtually the same task of speeding up the editing process — work sensibly in this kind of situation? More precisely, can the book and the individual chapters share the memos and the externs? Yes they can! If we instruct Memoize to use the same memo directory for both the book job and the chapter jobs, then we can externalize graphics when compiling a chapter and have the externs included when compiling the book (and vice versa).[14] All we need to do is use our old friend `memo dir` from section 2.5 — we see now that this setting is good for more than just keeping a clean house!

```
book.tex

\documentclass{book}
\usepackage{docmute}

\usepackage{memoize}
\mmzset{memo dir=chapters/book}

\usepackage{tikz}

\begin{document}

\chapter{Introduction}
This example demonstrates how to share
memos and externs between the book and
the chapters.

\chapter{A chapter}
\input{chapters/chapter1.tex}

\chapter{Conclusion}
Easy, right?

\end{document}
```

```
chapters/chapter1.tex

\documentclass{article}

\usepackage{memoize}
\mmzset{memo dir=book}

\usepackage{tikz}

\begin{document}

\begin{tikzpicture}
  \node[
    text width=5cm,
    align=flush left,
  ]
  {This picture can be externalized
    by compiling the chapter (twice,
    as we need to extract it as well).
    The extern will be picked up
    when compiling the book.};
\end{tikzpicture}

\end{document}
```

In the above example, the individual chapters reside in files stored in the `chapters` subdirectory, and that's why the `book.tex` preamble uses `memo dir=chapters/book` (rather than `memo dir=book` or just `memo dir`). However, Memoize has no trouble with a situation where the main file and the chapters reside in the same folder; the setup is even simpler, as we then say `memo dir=book` in both the book and the chapter preamble. The more complicated situation was chosen to point out the following potential problem with the setup where the chapters reside in a subdirectory.

If you're anything like me, you would first go for having a memo directory immediately contained in the project directory (so `examples/book.memo.dir` above) and set up `memo dir` as shown below. Well, this won't work, or at least it won't work with the vanilla TeX Live, because TeX will refuse to *write* into (memo) files outside the directory where it was executed,[15] and this is precisely what the chapter compilation is asked to do below.

```
the main file

\mmzset{memo dir=book}
% ...
\input{chapters/chapter1.tex}
```

```
a chapter file

\mmzset{memo dir=../book}
```

---
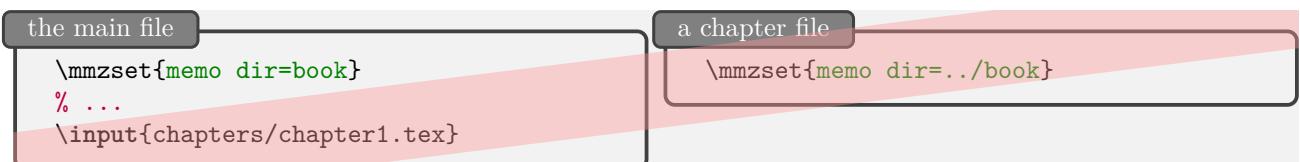
[14] Package `docmute` makes LaTeX ignore the preamble of the chapter file when including this file into the main document.

[15] In TeX Live, the `texmf.cnf` option controlling this behaviour is called `openout_any`. By default, it is set to `p` (paranoid), which "disallow[s] opening dot files [and] *going to parent directories*, and restrict[s] absolute paths to be under `TEXMFOUTPUT`" (emphasis mine).

Section 2.4 presented some ideas on how to work on a single picture. Those ideas can be all easily applied to the multi-file situation. For example, you could use `readonly` on the chapter that you're working on (and that chapter only). This way, the preview of the chapter will not be tarnished by the extern pages, and if you periodically compile it without `readonly`, or compile the book (which does not have the `readonly` set), you will have a reasonably up-to-date set of externs.

the main file
```
\mmzset{memo dir=chapters/book}
% ...
\input{chapters/chapter1.tex}
```

the current chapter file
```
\mmzset{memo dir=book, readonly}
```

For Emacs users

I often use this `readonly` trick myself, but with a twist. As an Emacs user, I don't use a TeX-based mechanism (such as the `docmute` package) to compile a chapter, but rely on the region compilation feature of Emacs' AUCTeX package. AUCTeX offers a way to compile the current buffer (if you don't know what an Emacs buffer is, read "file") or region (roughly speaking, the selected text). It does that by putting the buffer or the region into a file called `_region_.tex` while dressing it up in the preamble of the original document (when I'm working on a multi-file document, it correctly pulls the preamble from the main document). This results in a compilable region file. My trick is to detect whether I'm compiling a region (this is the job of `\ifregion`), and if so, put Memoize into the `readonly` mode (an alternative trick would be to `disable` it).

This is the trick in a nutshell, but to make it really work we have to address one further issue: the original document and the region have to share memos and externs. This happens automatically if the original document sets `memo dir` explicitly (e.g. if a document called `doc.tex` contains `memo dir=doc` in the preamble), but I'm lazy and don't want to write this in every document — if I have to do that, what's the point of `memo dir` I put into my `memoize.cfg` in section 2.5? Fortunately, the region file starts with `\message{ !name(⟨original document name⟩.tex)}` to indicate the origin. The complicated part of the code below (everything following `\mmzset{readonly}` parses this header to extract the ⟨original document name⟩, which is then fed to `memo dir`. Now, the trick works automatically for any document.[a]

memoize.cfg
```
\edef\regionfilename{\detokenize{_region_}}
\def\ifregion{%
  \edef\jobfilename{\jobname}%
  \ifx\jobfilename\regionfilename
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
}
\ifregion{%
  \mmzset{readonly}%
  \begingroup
  \openin0{\regionfilename.tex}\readline0 to \regionheader  \closein0
  \edef\temp{##1\detokenize{(}##2\detokenize{.tex)}##3}%
  \expandafter\def\expandafter\parseregionheader\temp\endregionheader{%
    \endgroup
    \mmzset{memo dir=#2}%
  }%
  \expandafter\parseregionheader\regionheader\endregionheader
}{}
```

---

[a]The assumption here is that `memo dir` is in effect for the original document. If not, the trick can be adapted to use `prefix`.

Memoize ships with built-in support for the most widespread LaTeX presentation class, Beamer, in the sense that it can externalize a picture which changes from overlay to overlay. Before we learn how to use that functionality, however, there's a peculiarity about loading Memoize in Beamer to address.

> Beamer opens the document PDF while loading the class, while Memoize requires the PDF from the previous compilation intact in order to extract the externs (when extraction is triggered internally, which is the default setting). The solution is to load Memoize (a package) before Beamer (a class), which can be done by using `\RequirePackage` instead of the usual `\usepackage`. Easy, if hacky.
>
> -
>
> ```
> \RequirePackage{memoize}
> \documentclass{beamer}
> ```

To memoize a piece of code which produces different results on different overlays — by virtue of containing `\pause`, `\only`, and/or related commands — apply key `per overlay`. Without this key, externalization of the picture will end badly, with a single extern (the final one) appearing on all overlays. The key may be invoked either from a prior `\mmznext` command,[16] or executed in the memoized code itself. The example below illustrates the latter option, and also shows that we may invoke it via its full path, `/mmz/per overlay`, when listed among options processed by `pgfkeys`.[17]

```
beamer.tex

\tikzset{only/.code 2 args={\only<#1>{\pgfkeysalso{#2}}}}
\begin{frame}{A frame with a memoized Ti\emph{k}Z picture}
  \begin{tikzpicture}[/mmz/per overlay]
    \node[ellipse, fill=yellow, only={2}{
        pin={[overlay, fill=red, pin edge={overlay, red}]60:An important remark!}
      }]{An important concept};
  \end{tikzpicture}
\end{frame}
```


> If the memoized code changes the value of Beamer's pause counter `beamerpauses`, e.g. by issuing a `\pause`, take care that (i) `per overlay` is executed prior to any changes of `beamerpauses`, and that (ii) the final value of this counter in the memoized code is the same for all overlays.

---

[16] Of course, `per overlay` may also be invoked from `\mmzset`, but I guess this won't make sense often. For example, if you set it for the entire presentation, and the presentation contains static memoized pictures as well, you will compile those pictures more times than necessary: once for each overlay, whereas once per frame would suffice. It might occasionally make sense, however, to use `per overlay` as an `auto` option — consult section 2.9 to learn what that is.

[17] Read section 4.2.4 to learn how the Beamer support is implemented. The implementation only uses Memoize's public interface, so understanding it should help if you need to support some other presentation package.

Some constructs — like plain TeX's `\llap` and `\rlap`, and, notably, Ti*k*Z overlays — fool TeX into thinking that the "size" of the typeset material is different than what it actually is. This can cause trouble for externalization: a piece of your picture might disappear! In a sentence, the solution is to manually set the *padding* of the externs, but let's slow down a bit.

The Ti*k*Z picture in the following example consists of node with a pin on the right, but let's say we want to horizontally center this picture so that only the node rather than the entire picture (including the pin) will be centered. This can be achieved by adding key `overlay` to the pin (actually, we need to add it to both the pin and its edge). Ti*k*Z normally updates the extents (called the bounding box) of the picture every time it puts something in it; when `overlay` is in effect, however, these updates are temporarily disabled. In effect, the `overlay` key on the pin below will fool TeX into thinking that the node is all there is to the picture, so centering will work as desired.

`overlay.tex` (no memoization)

```
\mmznext{disable}
\begin{figure}
  \centering
  \begin{tikzpicture}
    \node[align=center, text width=0.4\linewidth, draw=blue, thick, pin={
        [overlay, pin edge={overlay}, green!50!black] east:an \texttt{overlay}ed pin}
    ]{To horizontally center this node (rather than the entire picture),
      we make \TeX{} ignore the pin by \texttt{overlay}ing it.};
  \end{tikzpicture}
  \caption{Without memoization}
\end{figure}
\lipsum[66]
```

document page

To horizontally center this node (rather than the entire picture), we make TeX ignore the pin by `overlay`ing it. — an `overlay`ed pin

Figure 1: Without memoization

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

What happens when we try to externalize this picture? The example below shows what would happen if Memoize had no concept of *padding* — which we simulate by setting `padding`=0pt.[18] Along with the rest of TeX, Memoize would be fooled into thinking that the picture comprises of the node only, so the pin would never make it into the extern. You would end up with a document missing the pin![19]

`overlay.tex` (memoization without padding)

```
\mmznext{padding=0pt}
\begin{tikzpicture}
  % ...
\end{tikzpicture}
```

extern page

To horizontally center this node (rather than the entire picture), we make TeX ignore the pin by `overlay`ing it.

---

[18]Unlike in the rest of the manual, the extern pages in this section are shown without trimming the whitespace.

[19]On the first compilation, the document page containing the figure without padding looks fine, as it uses the result of the compilation rather than the extern file. But on the second compilation, when Memoize actually uses the extern, the pin disappears.

By default, Memoize puts an inch of space around (what it thinks is) the externalized picture, and if the overlayed parts of the picture fit into this inch of space, you will find them in the extern and therefore also in the document. In our example, however, the default padding is not enough — the pin is only partially visible.[20]

overlay.tex (memoization with default padding)

```
\mmznext{padding=1in} % this is the default
```

The solution is to set the padding manually. Below, I used `padding right` to only increase the padding on the right side (clearly, we also have `padding left`, `padding top` and `padding bottom`), but if you're not bothered by a large extern, you can just use `padding`, which sets all four sides at once. By the way, having too much padding (almost) never hurts, and as you see, you can use (simple) arithmetics in the value of these keys.

overlay.tex (memoization with extra padding)

```
\mmznext{padding right=1in+4em}
```

Incidentally, the `padding` keys only change how the externalized picture is *stored*. Memoize remembers the size of the extern as seen by TeX (e.g. the bounding box of the picture as reported by TikZ, with overlayed parts of the picture protruding out of it), and it uses that size when integrating the extern into the document — so everything works as it should!

---

[20]You might wonder why I didn't make the default padding much bigger, like 10 inches. TeX wouldn't be bothered (unless the resulting extern size exceeded its maximum dimension), but you might be, because with such a large default padding, all the externs would be huge, most often bigger than the document pages, and remember that the externs are first dumped into the document, where they can bother you.

2.9 The verbatim mode

Not all code will peacefully submit to memoization. In particular, this is the case for environments which process the environment body verbatim (or perform some other kind of `\catcode` magic). A simple environment of this kind is the standard LaTeX `verbatim`, but let us illustrate the issue with `tcblisting`, which typesets a code listing alongside its compiled effect. (This environment is defined by the `listings` library of package `tcolorbox` and was used extensively during the production of this manual.) To manually memoize a `tcblisting` environment, we enclose it in a `memoize` environment with a `verbatim` key in the optional argument — without this key, the example below would produce nothing but errors.[21]

```
verbatim-manual.tex

Use |\emph| to emphasize text:

\begin{memoize}[verbatim]
  \begin{tcblisting}{width=15em}
This is an example
\emph{within} an example.
  \end{tcblisting}
\end{memoize}

Don't use |\textit| for emphasis!
```

extern page

This is an example
\emph{within} an example.

This is an example *within* an example.

(The document page is the same as for the `verbatim-auto` example below.)

Using `verbatim` from `\mmzset` or `\mmznext` works just as well, and the latter can be very useful with automemoization, when some environment (say, `tcolorbox`) generally does not require the verbatim mode, but a specific occurrence does (say, because it contains some verbatim construction such as |⟨*verbatim text*⟩| of the `ltxdoc` class).

However, for an environment such as `tcblisting`, it makes the most sense to declare it verbatim in general, so that all instances of the environment will be processed in the verbatim mode. This is simple to do: add `verbatim` to the `auto` keylist.

```
verbatim-auto.tex

\mmzset{auto={tcblisting}{
    memoize, verbatim
}}
% ...
Use |\emph| to emphasize text:

\begin{tcblisting}{width=15em}
This is an example
\emph{within} an example.
\end{tcblisting}

Don't use |\textit| for emphasis!
```

(The extern page is the same as for the `verbatim-manual` example above.)

document page

Use \emph to emphasize text:

This is an example
\emph{within} an example.

This is an example *within* an example.

Don't use \textit for emphasis!

In fact, you can add any `/mmz` key to the `auto` keylist, and the key will be applied to all occurrences of the command or the environment. For example, adding `recompile` to the declaration of `tcblisting` above would recompile all and only the `tcblisting` environments; and as an `auto` declaration only updates (rather than completely replaces) a previous declaration, you can also say things like `auto=\tikz {recompile}` to recompile all TikZ pictures produced by the `\tikz` command (handy, as you don't know how automemoization for `\tikz` was declared unless you've read section 4.5 or looked at the Memoize's source code).

---

[21] Memoize also offers a *partial* verbatim mode, triggered by key `verb`; in this mode, the braces retain their usual category codes. Also note that the effect of `verbatim` can be "undone" by key `no verbatim`.

2.10 The final version of your document

Bluntly put, you might want to disable Memoize when compiling the final version of your document, at least if you intend to distribute it in electronic form, for two reasons:

- An externalized picture cannot contain hyperlinks. Any hyperlinks (or hyperlink anchors) contained in the original picture will silently disappear during the production of the extern.

- When the document contains many externs, the size of the resulting PDF can be several times the size of the PDF compiled without externalization.

Below, we list several ways of fully disabling Memoize. You're of course already familar with the first two ways, but what's this `nomemoize` package? The rationale behind this package is that if you want to be absolutely sure that there is no trace of memoization in your document (for example, see the `disable − enable` pitfall in section 2.4), the best thing to do is to not load the package at all. However, you have all those `\mmzset`s etc. in your source, so the document won't compile without `\usepackage{memoize}`, right? Right, but wrong. Enter `nomemoize`, a dummy package which accepts all the commands that Memoize does, but does nothing. In effect, your document will compile, but you can be sure that not a single memo or extern was loaded or produced.

```
\usepackage[disable]{memoize}
```

```
\usepackage{memoize}
\mmzset{disable}
```

```
\usepackage{nomemoize}
```

There is one issue you might need to resolve manually before package `nomemoize` works as intended, though. If you have used any `/mmz` keys outside `\mmzset`, you need to list them in `\nommzkeys`. For example, if you used `per overlay` in the manner illustrated in section 2.7, i.e. as `/mmz/per overlay` among the TikZ keys, you need to write `\nommzkeys{per overlay}` into the document preamble.

Another thing you might want to do once you have produced the final version of the document (in fact, just before you disable Memoize for good) is clean up. As we saw in sections 2.5 and 3.4, Memoize produces a lot of auxiliary files (memos and externs) and it keeps the old versions around! Once your document is prepared, you can reduce the clutter (and save some disk space) by deleting memos and externs belonging to the work-in-progress versions of your document, and keep only those used in the final version.

You could achieve this by deleting all the memos and externs (if you're using the `memo dir` directive, this amounts to the entire contents of the memo directory) and compiling your document for the final couple of times. However, there is an easier (but TeX-external) way: on the command line, change into the directory containing your (main) document and write `memoize-clean.pl` ⟨*document name*⟩`.mmz` (substitute `.py` for `.pl` to use Python rather than Perl). The script will inspect the contents of the `.mmz` record file to see which memos and externs were used in the final compilation, and delete all other memos and externs belonging to the given document.

> Deleting memos and externs is never an irreversible operation, as you can always recreate them, but it is still wise to be cautious when cleaning up. For one, avoid cleaning up after a compilation which produced errors; a failed compilation can lead to an incomplete `.mmz` file, which can in turn lead to over-deletion. Another bad idea is cleaning up after disabling Memoize for a part of a document, for the same reason.
>
> All that said, Memoize takes some precautions itself. It will cowardly refuse to perform the clean-up when the `.mmz` file is missing the end-of-file marker (`\endinput`), assuming that this indicates a fatal error in the previous compilation. It will do the same in case the `.mmz` file is absent or empty. The latter is assumed to be a result of a globally `disable`d memoization, but note that clean-up will be performed if memoization was disabled using package `nomemoize`: that package does not touch the `.mmz` file, so cleaning up should work as intended.

As the final note, memos and externs (cleaned-up or not) may be copied (along the document source) to another directory or machine, where they should be picked up by Memoize. There is no need to copy the `.mmz` file (assuming that the document PDF contains no extern pages waiting for extraction).

3 Digging deeper

3.1 Good to know

Line- and page-breaking   An extern can't be broken across lines or pages.

Externalization of a chunk of code produces a PDF, which is included into the document at subsequent compilations as a picture — an unbreakable object (a horizontal box) with fixed width and height. Therefore, the original code should produce an unbreakable object as well. For example, this means that you cannot externalize some text and expect TeX to break it across lines or pages on subsequent compilations. If you try, the compilation will succeed — without an error! — but your externalized text will end up in a single line, as shown below.

```
no-linebreaking.tex

Here's some externalized text: \mmz{\emph{\lipsum[66]}}
As you can see, the text is not integrated into the paragraph.
```

> extern page
>
> *Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis*

> document page
>
> Here's some externalized text: *Nunc sed pede. Praesent vitae lectus. Praesent n*
> As you can see, the text is not integrated into the paragraph.

> the expected document page
>
> Here's some externalized text: *Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.* As you can see, the text is not integrated into the paragraph.

That said, you *can* externalize a paragraph or some other vertical mode material using `capture`=`vbox`, but beware that the vertical spacing between the memoized material and its surroundings might change.

remember picture   TikZ pictures using this key cannot be externalized.

Memoize will silently refuse to externalize any TikZ picture using `remember picture` (see §17.13 of the TikZ & PGF manual). Such pictures interact with the outside world — they either reference or are referenced by other pictures — and are as such unsuitable for externalization. For example, while the colored boxes in this manual are generally externalized — out of principle ☺ — the title page illustration is not, and it cannot be, because of the arrows connecting the various TikZ pictures composing that illustration. Some packages use the `remember picture` mechanism under the hood, and are thus subject to this limitation; one example is package `todonotes`, but in general, any package dealing with absolute positions on the page will be limited in this way.

How does Memoize deal with this situation? Well, by cowardly refusing to externalize any code which uses `remember picture` or a similar mechanism for dealing with absolute positions. Luckily, any such mechanism eventually boils down to the TeX primitive \(pdf)savepos, so Memoize hacks — or as we will say in this manual, *advises* — this primitive to abort any ongoing memoization. Initializing and then aborting the memoization takes some time, to be sure, but the overhead is negligible, especially in the light of the fact that *not* aborting wreaks real havoc.

Memoize actually provides a user interface for aborting memoization. Memoization can be aborted either manually, by executing \mmzAbort, or automatically. The latter is a generalization of the automemoization idea: a command such as \(pdf)savepos can be advised to abort memoization by

`auto=\(pdf)savepos{abort}`. In Memoize, commands take your advice seriously, so memoization will be aborted whenever the advised command or environment is encountered.

Indirectly embedded environments   Such environments cannot be memoized.

Read this if you got an error message such as `Environment "tikzpicture" ended as "foo"`.

Some environments are defined so that they embed another environment using the idiom shown on the left: the begin-code of the outer environment opens the inner environment, and the end-code of the outer environment closes the inner environment. While this is a fine, and common, idiom, it messes up the memoization of the inner environment. In the example on the right, trying to *auto*memoize a `minipage` environment (not recommended at all!) causes trouble with package `sectionbox`.[22,23]

```
\newenvironment{foo}% ... args
{% the begin-code of foo
  % ...
  \begin{tikzpicture}
  % ...
}
{% the end-code of foo
  % ...
  \end{tikzpicture}
  % ...
}
```

sectionbox.tex

```
\usepackage{sectionbox}
\usepackage{memoize}
\mmzset{auto={minipage}{memoize}}
\begin{document}
\begin{sectionbox}{Boxed section}
  Some text in the box.
\end{sectionbox}
\end{document}
```

```
! Package collargs Error: Environment
"minipage" ended as "sectionbox".
```
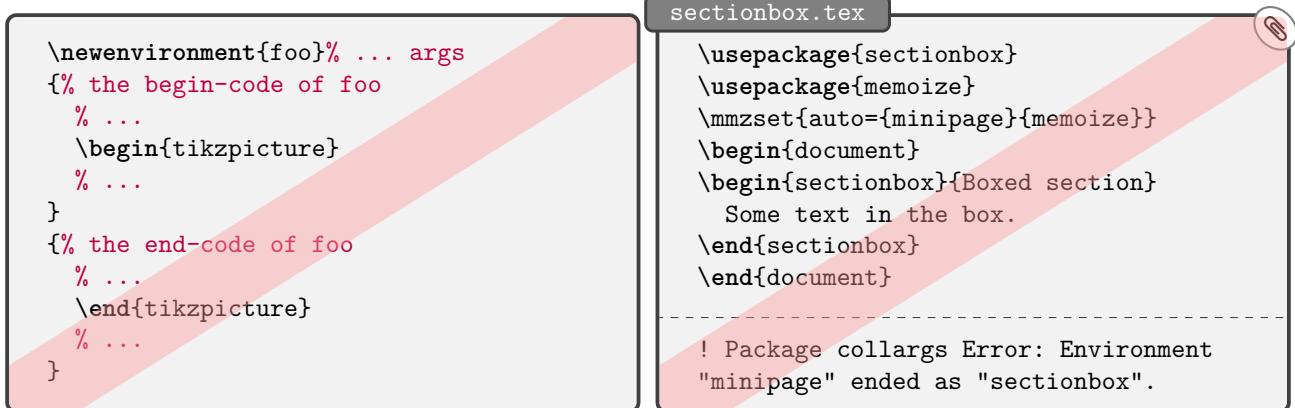
What are your options in this kind of a situation?

1. The only way to perform any memoization here is to memoize the *outer* environment — if that makes sense.[24] You can do this either on a case-by-case basis, by enclosing it in the `memoize` environment, or automemoize it: `auto={⟨outer environment⟩}{memoize}`.

2. But what if memoizing the outer environment is out of the question? Then, the only way to avoid the error is to prevent the automemoization of the inner environment.

   (a) If you are facing a single occurrence of the problem, it is perhaps easiest to use key `disable` just before the start of the outer environment.

   (b) Otherwise, you can automatically disable memoization for the span of each occurrence of the outer environment: `auto={⟨outer environment⟩}{nomemoize}`.

   (c) To deactivate the automemoization of the inner environment for the span of the outer environment, but otherwise allow for memoization inside the outer environment: `auto={⟨outer environment⟩}{noop, deactivate=⟨inner environment⟩}`. Key `noop` does nothing but apply the given options (in this case, `deactivate=⟨inner environment⟩`) to the advised command or environment.

---

[22]This is a `Package collargs Error` because Memoize outsources the actual work of collecting the environment body to the auxiliary package CollArgs, described in section 4.5.2.

[23]Why does this happen? As mentioned in section 2.5, Memoize keeps track of memos and externs by the MD5 sum of the memoized code. But to compute that sum for an environment, Memoize has to *grab* the environment body, meaning it has to collect the body in advance. This presents no problem when `\end⟨environment name⟩` is already present in the input stream at the time `\begin⟨environment name⟩` is executed, like when you use the environment normally in your document, or when some macro expands so that it produces both `\begin⟨environment name⟩` and `\end⟨environment name⟩` simultaneously — so there would be no problem above if `\end{minipage}` occured in the beginning code of `sectionbox`. The idiom presented above is problematic for memoization because at the time TeX executes `\begin{sectionbox}`, putting `\begin{minipage}` into the input stream, `\end{sectionbox}` is not yet executed and remains as it is. The input stream therefore contains a pair of `\begin{minipage}` and `\end{sectionbox}`. In the normal, non-memoizing course of events this would not be a problem, because `\end{sectionbox}` would eventually expand to `\end{minipage}`. During memoization, however, this *is* a problem, because, as we said, Memoize needs to grab the environment body: upon encountering `\begin{minipage}`, it looks in the input stream for `\end{minipage}` — but there is no `\end{minipage}` in the input stream, there is only `\end{sectionbox}`, and this results in the `Environment "minipage" ended as "sectionbox"` error.

[24]This avoids the error because Memoize grabs and memoizes the outer environment, and while it is memoizing it, further memoization is switched off.

3.2 Extraction methods and modes

Remember that in Memoize, externalization is a two-step process. First, externs are typeset on separate pages of the main document, called extern pages. Then, these extern pages are extracted out of the main document PDF into extern files. The process is illustrated on the title page.

Memoize is flexible in terms of which piece of software is used to perform extern extraction. It ships with three *extraction methods*:

perl A Perl script, `memoize-extract.pl`. This method is the default because it is fast and because Perl is usually already installed on a system running TeX. However, you will most likely still need to install the PDF processing library `PDF::API2`, or its fork `PDF::Builder`; the installation guidelines can be found in section 1.1.

python A Python script, `memoize-extract.py`. This method is even faster than the Perl script, though not by much. Try it out if you have problems installing Perl or the required libraries, or if the Perl script chokes on your document (see section 6.3 for the list of known issues). Besides Python ($\geq$ 3.8), you will also need the Python library `pdfrw` or `pdfrw2`. For the installation guidelines, see section 1.1.

tex TeX-based extraction requires no additional software, but it is much slower than the scripts. As TeX can only produce a single PDF per compilation, an instance of TeX (loading the entire document PDF) has to be invoked for each extern, and this takes time (although the entire process is still much faster than the venerable TikZ externalization library).

Memoize is also flexible in terms of how extern extraction is triggered, providing two *extraction modes*:

internal By default, extern extraction is triggered internally, i.e. by Memoize during the compilation of the document; more precisely, any externs produced in a compilation are extracted during the next compilation. To choose an extraction method other than the default Perl script, load Memoize with the package option `extract=`⟨*extraction method*⟩.

external Loading Memoize with with package option `extract=no` instructs Memoize to *not* trigger the (internal) extraction. When instructed to use extraction "method" `no`, Memoize expects you to trigger the extraction yourself, in any way that is convenient to you: manually from the command line, or automatically through your editor, a Makefile, etc. — all Memoize cares about is that the extraction takes place before the next compilation of the document.

Summing up, the extraction mode and method are selected by providing the appropriate value to package option key `extract`; the possible values are listed in the table below. Note that this key can only be used as a package option, or in `\mmzset` within `memoize.cfg`. In particular, it is disabled in the document preamble, because Memoize performs extraction while it is loaded.

extraction method	external program	Memoize invocation
perl	memoize-extract.pl	\usepackage{memoize}[25]
python	memoize-extract.py	\usepackage[extract=python]{memoize}
tex	pdftex	\usepackage[extract=tex]{memoize}
no	none (external extraction)	\usepackage[extract=no]{memoize}

For internal extraction, TeX must be allowed to execute the external program implementing the chosen extraction method. Both `memoize-extract` scripts should be listed among restricted shell escape mode commands in your TeX distribution; their execution should therefore be allowed under the default, restricted shell escape mode. However, the `pdftex` program, executed by extraction method `tex`, is not listed there, nor should it be. If you are forced to use this fallback method, I suggest you compile documents loading Memoize under the full shell escape mode, by adding command-line option `-shell-escape` (on TeXLive) or `--enable-write18` (on MiKTeX) to the invocation of the TeX program. The answers linked from question "How can I enable shell-escape?" on TeX StackExchange will tell you how you can ask your editor to do this for you.

---

[25]Or `\usepackage[extract=perl]{memoize}`. This is useful if you have changed the default using `memoize.cfg`.

You may use any extraction method to perform external extraction. The simplest option is to use the Perl or the Python script. Supposing you are doing this manually from the command line, change into the directory containing your document, which should contain the auxiliary `.mmz` file produced by Memoize, and execute:

(a) `memoize-extract.pl` ⟨*document name*⟩`.mmz`   (for the Perl script)

(b) `memoize-extract.py` ⟨*document name*⟩`.mmz`   (for the Python script)

See sections 4.3.1 and 5.5.1 for further details on the `.mmz` file and the extraction scripts.

Things are a bit more complicated if you want to use the T_EX-based extraction method externally, because an instance of `pdftex` needs to be invoked for each extern (and these have unwieldy names and can be many in number), but Memoize can help you here by producing a shell script or a makefile, executing which will extract all the externs at once. To have Memoize produce a shell script, use package option `record=sh` (or `record=bat` on Windows); package option `record=makefile` will make a makefile. By default, these files are named `memoize-extract.`⟨*document name*⟩ plus the `.sh`, `.bat` or `.makefile` suffix. If neither a shell script nor a makefile works for you, you can also define your own kind of *record file*, to be processed by the external tool of your choice (and implementation) in order to extract the externs; see section 4.3.2 to learn how to do this.

Cross-referencing presents a challenge to externalization, because without special provisions, the "communication channel" between the `\label` and the `\ref` is broken once we start utilizing the extern.

One direction of the issue occurs when a `\label` within the memoized code is referenced by a `\ref` on the outside. Without the (built-in) workaround, the `\label` command would only be executed when the extern is being produced, but not on subsequent compilations of the document, when it is merely included. Memoize addresses this problem by generalizing externalization (which can only produce a picture, the extern) to memoization (which can additionally produce arbitrary code). When Memoize is externalizing code which contains a `\label`, it automatically replicates it into the *memo*, which is input into the document on subsequent compilations. In effect, the memo–extern team will continue to produce the label even when it is utilized rather than compiled. As far as the author is concerned, `\label`s in memoized code "just work," without any observable differences to the situation without memoization. This is why we will not discuss this direction of the issue here; a reader interested in how precisely the system works is invited to read section 4.2.

The other direction of the issue occurs when a `\ref` within the memoized code references `\label` on the outside. In this situation, the extern should be recompiled when the value of the label it refers to changes. Again, Memoize addresses this problem in full generality, by associating with each extern a *context*, and recompiling the extern whenever the value of the context changes.[26] All that needs to be done for `\ref` and friends, specifically, is to advise them to add their reference keys to the context.

As we shall see presently, for the author, the only difference between a non-memoized and a memoized `\ref` is that the latter will take one more compilation cycle to "stabilize" the resulting document. (More precisely, the memoized situation will take one more cycle if the reference is undefined on the first compilation.) Then, we will show how we can teach Memoize about cross-referencing commands other than `\ref` and `\pageref`. Finally, we will learn about key `context`, the backbone of the cross-referencing support in Memoize. (The inner workings of the context are further explained in section 4.2.2.)

When the memoized code contains a `\ref` referring to a label given in another part of the document, the code is recompiled when (and only when) the reference changes. Let us look at the following example, jumping in at the point where it was already compiled enough times that the resulting PDF had stabilized into a single (document) page with correct references. (Environment `nomemoize` disables memoization of Ti*k*Zlings, so that their externs don't disturb us, and we can focus on the `\tikz` command, which does get externalized and contains a `\ref`.)

```
Here's some Ti\emph{k}Zlings:
\begin{nomemoize}
  \tikzset{x=1.3ex, y=1.3ex, baseline=0.5ex}%
  \begin{enumerate*}
  \item\label{item:koala} \tikz\koala;
  \item\label{item:penguin} \tikz\penguin;
  \end{enumerate*}
\end{nomemoize}
Where's the penguin? In \ref{item:penguin}. Yes, in
\tikz[baseline]\node[draw=red,thick,fill=yellow,anchor=base]{\ref{item:penguin}};
```

ref.tex (with stable output after three compilations)

document page

Here's some Ti*k*Zlings: 1. 🐭 2. 🐧 Where's the penguin? In 2. Yes, in 2

Let us add an owl in front of the penguin. In the next compilation, neither the "normal" nor the memoized reference is yet updated, as expected — in this compilation, the new value of the penguin label only makes it into the `.aux` file.

---

[26]The dependency of an extern upon prior definitions and such can also be addressed in a more *ad hoc* manner, by recompiling manually; we have already touched upon this subject in section 2.4, and will revisit it in section 3.4.

```
ref.tex (after the first compilation with the added owl)
    \begin{enumerate*}
    \item\label{item:owl} \tikz\owl;
    \item\label{item:koala} \tikz\koala;
    \item\label{item:penguin} \tikz\penguin;
    \end{enumerate*}
```

> **document page**
> Here's some Ti*k*Zlings: 1. 🦉 2. 🐨 3. 🐧 Where's the penguin? In 2. Yes, in `2`

During the following compilation, the `\ref`s pick up the new value of the penguin label, and the `\ref` inside the automemoized `\tikz` command forces recompilation of the extern (*how* this is done will be explained later).

```
ref.tex (after the second compilation with the added owl)
```

> **extern page**
> `3`
>
> **document page**
> Here's some Ti*k*Zlings: 1. 🦉 2. 🐨 3. 🐧 Where's the penguin? In 3. Yes, in `3`

In the next compilation, the resulting PDF is finally stabilized, as the updated extern is (extracted and) included into the document.

```
ref.tex (after the third compilation with the added owl)
```

> **document page**
> Here's some Ti*k*Zlings: 1. 🦉 2. 🐨 3. 🐧 Where's the penguin? In 3. Yes, in `3`

The message to take home? When some memoized code contains a reference and that reference changes, it will take three compilation cycles (so, one more cycle than without memoization) for the resulting document to "stabilize."

Out of the box, Memoize supports the standard LaTeX cross-referencing commands `\ref` and `\pageref`. To automatically recompile code containing some other cross-referencing command, like `\vref` of package `varioref`, we use the advising framework implemented by package Advice. This framework is a generalization of automemoization: we use the familiar `auto`, but with advice offered by `ref` rather than `memoize`.

```
vref.tex                                                    📎
    \mmzset{auto=\vref{ref}}
```

Key `ref` only works for commands which operate on a single reference key. However, that single key (which must be enclosed in braces) may be preceded by optional argumen(s) of any kind. Extensions to `\ref`, e.g. the `hyperref`'s variant, which accepts an optional `*`, work out of the box. Furthermore, Memoize offers support for cross-referencing commands which work on multireferences and reference ranges, such as `cleveref`'s `\cref` and `\crefrange`. Those commands should be advised by `auto` keys `multiref` and `refrange`, respectively.

We have jumped into first example of this section with the assumption that it had already been compiled several times, allowing the resulting PDF to stabilize. Let us now take a look at what happens at the very first, fresh compilation of our original example (the one without the owl). (Removing the `.aux` file before compiling the example again will start afresh.) The curious thing is that we don't get the extern page containing `??`. This is so because by default, Memoize aborts a memoization containing an undefined reference.

Now sometimes you might want to produce an extern even if it contains an undefined reference — for example, you might intend to write the code containing the \label much later but enjoy the speed-up offered by Memoize until then. In that case, apply the `auto` key `force ref` to \ref.

However, note that when you use `force ref`, LaTeX will *not* complain about the undefined reference once the extern containing it is included (unless that reference also occurs in some non-memoized piece of code). Using `force ref` is therefore a tiny bit dangerous, and this is why `ref`, with the abortion mechanism, is the default handler for \ref and \pageref.

As already noted in the previous section, \ref works by appending the cross-reference to the *context*, the change of which triggers recompilation. Memoize initializes the context to contain the four `padding`s — as a result, an extern recompiles if we change the padding — but we can append stuff to the context by ourselves, as well. Below, we use key `context` to append the font size (we'll talk about the value given to this key a bit later); as a result, the picture is recompiled whenever the font size changes. Below, we change the font size using command \small; changing the default size with a class option such as `12pt` works as well.

How does this work? Key `context` appends the given tokens to the *context expression*. When creating an extern or trying to use it, Memoize (fully) expands this expression and computes the MD5 sum of the expansion. This *context MD5 sum* then serves as a part of the extern's filename (see sections 2.5 and 4.2). In effect, Memoize will only find (and utilize) the extern if *the context MD5 sum computed during (attempted) utilization matches the one computed during memoization.*

As revealed by looking at the LaTeX source code, LaTeX holds the current font size in macro \f@size, and above, we have effectively added the contents of this macro to the context. Now, why didn't we simply write `context`=\f@size? First, we used \csname ... \endcsname because we were under the normal LaTeX catcode regime, where @ cannot be a part of the command name. Of course, we could have temporarily changed the catcode of @ using \makeatletter and \makeatother, but I would advise against that, because the approach does not work in general: it fails when key `context` is used *within* memoized code (we will explain why in section 4.2). Another reason why I recommend the \csname ... \endcsname approach is that it does not result in an error when the control sequence is not defined (\csname ... \endcsname will expand to \relax then); this behaviour is handy for

undefined cross-references, for example. Second, why did I write `fsize={...}` around the control sequence? Well, because I'm being paranoid, really. Writing `context={\csname f@size\endcsname}` would work just as well, but I like to explicitly "announce" the value to prevent any possibility of a conflict with an alternative context. Imagine that we don't use the "announcements" and we decide to add some other dimension instead of the font size to the context. Now if that dimension happened to have the same value as the font size, Memoize would incorrectly pick up the "font size extern" instead of producing a new one.

It bears emphasizing that whatever you add to the context expression must be fully expandable, and also not merely declared as robust. So writing `context=\ref{`⟨*key*⟩`}`, for example, would be unwise, since it would not work as intended when package `hyperref` is loaded. (This package declares `\ref` as robust, so it won't expand to the cross-reference value.) You have to look up where the cross-references are stored internally; the cross-reference for ⟨*key*⟩ turns out to be stored in the internal control sequence `\r@`⟨*key*⟩, so it is `\csname r@`⟨*key*⟩`\endcsname` that the `ref` handler actually appends to the context.

The padding and font-size contexts are useful quite generally. However, the context can be pretty command-specific, as well. Consider the `skak` package used for typesetting chess games. The board is drawn using command `\showboard`, but this command has no arguments, because it draws the state of the board that is reached by the moves given by command `\mainline`. Memoizing `\showboard` as such will therefore yield the wrong result — all the boards will be one and the same board! The solution is to provide the correct context: we dig into the `skak` sources and realize that the current board is stored in macro `\csname chessgame.skak.mainline\endcsname`.

If you remove `context={...}` from the code above, you will end up with a document where the final board drawn takes place of all the boards. This is so because in that case, all externs are written into the same file, so the final extern overwrites the previous ones, but note that you will only observe this after the second compilation, when the externs are actually used.

3.4 More on redefinitions and stale externs

In this subsection, we elaborate on an issue touched upon at the beginning of section 2.4: what happens if the memoized code depends on some macro or style which gets redefined? The answer was "nothing," and one solution was to `recompile` the code. Let us take the example from that section a bit further. We will propose no new solution or workaround, but deepen our understanding of the issue.

Working on `redefinitions.tex`

I like red. My emphasized nodes will have red background.

1
```
\tikzset{emph/.style={fill=red, text=blue}}
\tikz\node[emph]{an emphasized node};
```
an emphasized node

Hmm, this particular node is really important, let me put the text in italics as well!

2
```
\tikz\node[emph, font=\itshape]{an emphasized node};
```
an emphasized node

You know what? Perhaps yellow background would work better — in general.

3
```
\tikzset{emph/.style={fill=yellow, text=blue}}
\tikz\node[emph, font=\itshape]{an emphasized node};
```
an emphasized node

How come my node is still red?! Oh yes, I changed the style, so I have to recompile the extern!

4
```
\mmznext{recompile} % only for one compilation!
\tikz\node[emph, font=\itshape]{an emphasized node};
```
an emphasized node

Ahh, yellow background, that's much better. But you know what, this double emphasis won't do after all, let me go back to the upright shape.

5
```
\tikz\node[emph]{an emphasized node};
```
an emphasized node

Red???!???!? Ok, I know that recompiling will help, but what happened here?

6
```
\mmznext{recompile} % only for one compilation!
\tikz\node[emph]{an emphasized node};
```
an emphasized node

What happened is that the externs from steps 1 and 5 share the very same code. In step 1, this code was compiled when the red `emph` style was in effect, and that extern lingered and was eventually picked up again in step 5, Memoize having no idea that it is including an extern produced with the obsolete definition of the style.

There are two points to this story. First (and forgetting for a moment about the context, which we started discussing in section 3.3), Memoize identifies externs (and memos) by the code that produced them — or more precisely, by the MD5 sum of the code, as each piece of code has a unique (well, unique enough) MD5 sum. Each extern is saved in a file whose name contains this MD5 sum; see section 2.5 for illustration. Generally, this is a very useful feature. You can move your picture to another location in the document, insert some other (externalized) picture in front of it, and so on, all without triggering recompilation of the extern(s). (None of this is possible with TikZ's externalization library, which identifies the externs by the order in which they appear in the document.)

The downside of the MD5 sum approach is the potential pitfall illustrated above, and the downside comes about because of the second point of the story: Memoize does not attempt to delete the "old" externs. (However, as described in section 2.5, stale memos and externs can be easily removed using the `memoize-clean.pl` script.) That would be not only dangerous (as any deletion inherently is) but also potentially wasteful: what if you have only temporarily removed some code, or compiled only a portion of the document — you surely wouldn't want your hard-won externs to disappear in such a situation!

The pitfall described above applies to any command which depends on parameters which can be set prior to the invocation of the command, like TikZ pictures, which depend on the settings given in `\tikzset`. After customizing the settings, you will have to recompile the externs: `\mmznext{recompile}` is useful when you only have to recompile a single extern; use `\mmzset{recompile}`, or the package option `recompile`, to recompile all externs in the document; and there is also the middle road: if you have changed only Forest's settings, you can write `auto={forest}{recompile}` to recompile all and only the Forest trees.

Above, we have seen the "same code, same extern" issue manifested "through time," i.e. Memoize was (incorrectly) reusing externs produced in previous compilations, but the issue can also manifest "through space." This can happen if the same code appears twice in the same document — but, crucially, with some parameters which it depends on changed from one occurrence to the next. Observe what happens in the following example, where the settings for `\progressbar` are changed by `\progressbarchange`. After the first compilation, everything looks fine. But as both extern pages were produced by the same code, they will be stored into the same file, the second one overwriting the first one. The second compilation pulls in the externs — or rather, the single extern — resulting in the document containing the second progressbar in the place of the first one as well.


```
progressbar.tex
\usepackage[auto={progressbar}{memoize}]{memoize}
% ...
\progressbar{0.85}
\progressbarchange{roundnessr=0.5, filledcolor=red!30}
\progressbar{0.85}
```
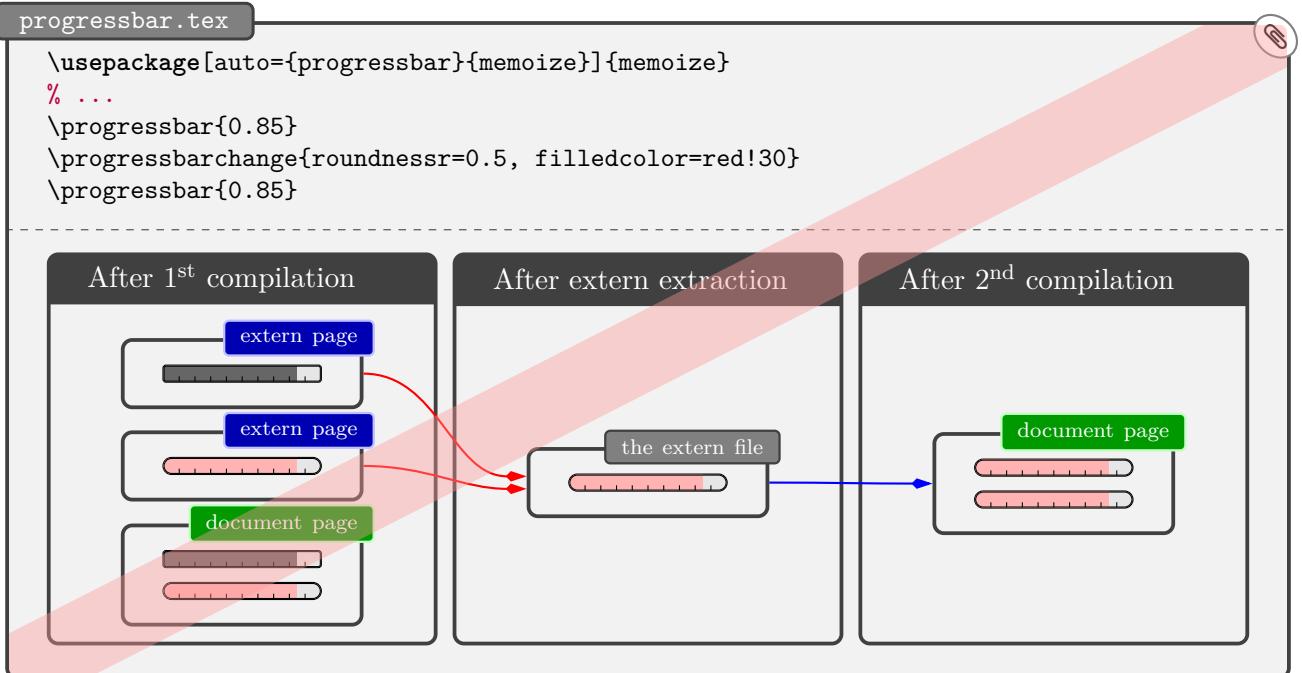
The same "extern duplication" can arise due to how a particular command is implemented. Say we deactivate automemoization of Forest trees (`deactivate=forest`), but keep on automemoizing TikZ pictures. Forest uses `tikzpicture` under the hood (a lot); in particular, the tree itself is typeset as a `tikzpicture` environment. But the code that typesets it is the same for all trees, regardless of their content (the actual content of the tree is hidden in various macros and boxes, rather than "pasted" into the `tikzpicture`). Consequently, the final tree of the document will overwrite all other trees in the document, just as the second (and thus final) progress bar overwrote the first one above.[27] Ouch!

Generally speaking, this final sort of extern duplication issue can arise whenever we have an "outer" command that we don't want to (auto)memoize which uses an "inner" command that we *do* want to automemoize. The solution is to use the `auto` key `nomemoize` on the outer command; remember that this key disables memoization for the space of the command or environment. For example, the correct way to "deactivate" automemoization of `forest` environments (but keep automemoizing TikZ pictures) is `auto={forest}{nomemoize}`.

---

[27]That is assuming that TeX doesn't simply spew a bunch of errors. This can happen as well. In the interest of full disclosure, compiling a Forest tree in the situation described above would actually also produce — but only in the first compilation — a number of small empty extern pages, one for each node of the tree. A promise: Forest will soon fully support Memoize and (among other things) avoid this pitfall. But the principle will remain.

3.5 Supporting Memoize in your package

3.5.1 Loading Memoize?

So you want to support Memoize in your package? That's great!

What form precisely this support will take of course depends on your package. For some commands, a simple `auto` declaration will suffice; for other commands, you might need to write a dedicated *memoization driver*, as explained in section 4.4. However, one thing is clear: you *don't* want to require Memoize's presence by `\RequirePackage{memoize}` in your package. That would trigger memoization, but triggering memoization should be left at the sole discretion of the author. The question is, if you're not allowed to load Memoize, how can you even issue the `auto` declaration?

Well, it's not that you really want to *memoize* anything; you want to make the commands of your package *memoizable*. So: `\RequirePackage{`memoizable`}` — and note that in 'memoizable', the final 'e' of 'memoize' is dropped, apparently this is the correct way to spell it.

Loading `memoizable` does nothing if Memoize is already loaded, and behaves like package `nomemoize` otherwise — remember from section 2.10 that `nomemoize` is a dummy package which accepts all the commands that Memoize does, but does nothing.

I have decided to require that Memoize must be loaded before any package that supports it. Allowing for an arbitrary loading order would complicate the implementation (and possibly even turn out to be problematic), and furthermore, Memoize likes to be loaded early anyway, because it needs to be loaded before the document PDF is opened if it is to perform the embedded extern extraction. I don't think the ordering requirement will cause any problems — let me know if it does! — but perhaps it is wise to inform the author about it in the documentation of your package (I did so at the end of section 2.3). Anyway, I have enforced the requirement by raising an error and refusing to load the package in case Memoize detects `memoizable` to be loaded.

Note that the loading order requirement implies that you can use `\@ifpackageloaded{memoize}` to specifically react to the presence Memoize, if necessary.

3.5.2 Memoizable design

Many commands and environments can be submitted to externalization with a single-line `auto` declaration, as illustrated in section 2.3, perhaps requiring an addition to the context (section 3.3), or some pre- or post-memoization code (section 2.7). In some situations, however, these simple approaches won't work. Most often, this will happen when the extern must be integrated into the document in some special way. For example, a command might internally create floating material, or surround the core typeset material with some stretchable space.[28] None of these behaviours can be replicated by merely including the extern; with respect to the stretchable space, remember that an extern, being a picture, has fixed size, so if our extra space ended up in the extern, it would lose the stretchability.

The key to successful memoization of problematic commands is their design. In a nutshell, the idea is to *break up the command's definition into two parts, the outer command and the inner command, and only submit the inner command to automemoization.* We will illustrate this with a simple environment — `poormansbox` — which produces a potentially framed box of the given width, and surrounds this box with some configurable material — by default, this material will be stretchable vertical space, and this will be the source of our memoization problem. (In terms of user experience, the solution in this section will leave something to be desired, but we will revisit the example in section 4.4.4 and make things right.)

Let us first take a look at a document using our to-be-developed box environment. The `poormansbox` environment takes one optional argument, a keylist of options, which can also be set with the `\pmbset` command. This being a **p**oor **m**an's **b**ox, it doesn't recognize many options. One can set the `width` of the box, or request that it occurs in a `frame`, and the surrounding material can be configured using

---

[28]Commands and environments of package `tcolorbox` exhibit both these issues (see `tcolorbox` options `float`, `before` and `after`), and were in fact the inspiration for several technical details of Memoize.

keys `before` and `after`. As we will see later in the listing of the package, the box is `\linewidth` wide by default, has no frame, and is surrounded by vertical glue `\vskip 2ex plus 1ex minus 1ex` (2ex of natural vertical space which may both stretch and shrink for `1ex`); furthermore, the default value of `before` contains `\centering` to center the box horizontally (centering is of course only observable when we change the width of the box).

```
\parskip 1ex plus 0.5ex minus 0.5ex

\begin{document}
\lipsum[3]

\begin{poormansbox}[width=.8\linewidth]
  \pmbset{width=\linewidth}
  \lipsum[101]
  \begin{poormansbox}[frame]
    \footnotesize\lipsum[66]
  \end{poormansbox}
  \lipsum[75]
\end{poormansbox}

\lipsum[4]

\begin{poormansbox}[
    width=.6\linewidth, frame,
    before=\noindent\llap{---},
    after=---
  ]
  \lipsum[65]
\end{poormansbox}Framed.

\lipsum[144]
\end{document}
```


You might want to play with the example to see that the surrounding vertical space is indeed stretchable. The example is set up so that the surrounding space is shrunk a bit to fit all the material onto one page. But if you remove the final `\lipsum[144]`, the natural amount of all vertical space can be accommodated on the page, so you should observe an increase of vertical spacing.

You might have noticed that the example contains nested `poormansbox`es: the second box (the one which contains `\lipsum[66]`) is nested within the first one (between `\lipsum[101]` and `\lipsum[75]`). This is intentional: when we will revisit the `poormansbox` example in section 4.4.4, the implementation will have to pay special attention to nesting (which presents no problem to the implementation in this section).

As you can see in the package listing below (`poormansbox.sty`), the implementation of our environment is straightforward. We first define the configuration command `\pmbset` and the option keys (we're using `pgfkeys`), and set the option defaults. Then, we move on to the environment itself: we apply the given options, execute the pre-code, typeset the box (which is a `minipage` of the given width, potentially wrapped in a `\fbox`), and execute the post-code.

Now let's make our `poormansbox` externalizable (`poormansbox-memoizable.sty`). As announced above, the idea is to split the definition of the environment into the outer part (below, the user-level environment `poormansbox`), which (applies the options and) executes the pre- and the post-code, and the inner part (below, the macro `\@poormansbox`), which typesets the actual box. If we then then submit `\@poormansbox`, rather than `poormansbox`, to automemoization, the outer part will be executed at every compilation (giving us stretchable space if we request it), while the inner command will be executed (and memoized) at the first compilation, and substituted for the extern (the fixed-size box) at subsequent compilations.[29]

---

[29]You might have wondered why our definition of the `poormansbox` environment grabs the body into an argument (`+b`,

```
poormansbox.sty

  \ProvidesPackage{poormansbox}
  \RequirePackage{pgfkeys}

  \newcommand\pmbset[1]{\pgfqkeys{/pmb}{#1}\ignorespaces}
  \newif\ifpmb@frame
  \pmbset{
    frame/.is if=pmb@frame,
    width/.store in=\pmb@width,   width=\linewidth,
    before/.store in=\pmb@before, before=\vskip 2ex plus 1ex minus 1ex \centering,
    after/.store in=\pmb@after,   after=\vskip 2ex plus 1ex minus 1ex,
  }

  \NewDocumentEnvironment{poormansbox}{
    o  % the options
    +b % the environment body
  }{%
    \pmbset{#1}% apply the options
    \pmb@before % execute the pre-code
    \ifpmb@frame\expandafter\fbox\else\expandafter\@firstofone\fi{% add the frame, maybe
      \begin{minipage}{\pmb@width}% create the minipage
        #2
      \end{minipage}%
    }%
    \pmb@after % execute the post-code
  }{}
```

Looking at the definition of the internal `\@poormansbox` command, it might strike you as weird that we have equipped this command with an optional argument (`#1`) it never uses. However, this optional argument is crucial for memoization. It will become a part of the memoized code (note `args=om` in the `auto` declaration) and thereby ensure that Memoize will produce separate externs for invocations of `\@poormansbox` with the same environment body but different options; or in other words, it will ensure that changing the options recompiles the extern.[30,31]

The downside to automemoizing an internal command is that this might be counter-intuitive for the author. For example, to deactivate automemoization of `poormansbox`, the author will have to write `\mmzset{deactivate=\@poormansbox}` (note the `\@`), but they will have no clue they have to do this unless they have carefully read `poormansbox`'s documentation. Even worse, the above

---

yielding `#2`), necessitating the use of `\NewDocumentEnvironment` over the venerable `\newenvironment`. One reason was that having the environment body as an argument simplifies wrapping the `\fbox` around the `minipage`, but there is a more important reason. If we did not grab the environment body, we would have to implement the internal part of the definition as an environment (`@poormansbox`) as well, and embed it into the user-level environment using the following idiom: `\newenvironment{poormansbox}[2][]{...\begin{@poormansbox}}{\end{@poormansbox}...}`. However, as illustrated in section 3.1, automemoizing an environment indirectly embedded in such a way produces an error, because Memoize is prevented from collecting the environment body.

   [30]Of course, this only holds for options given in the optional arguments; if the user changes an option value using a prior `\pmbset` (and that option does not occur in the optional argument), Memoize won't detect the change. But the end-user knows about this issue, as it was addressed in sections 2.4 and 3.4, and she is also aware of two workarounds: manual recompilation, or setting the context (section 3.3).

   While we're on the subject of the context, note that it is also possible to deploy context to trigger recompilation of the inner command upon change of parameters it depends on. We could simply omit the optional argument of `\@poormansbox` and add `context={width=\pmb@width,frame=\ifpmb@frame true\else false\fi}`, to the `auto` declaration. The advantage of such an approach is that Memoize reacts to the change of parameters regardless of whether they are set using the optional argument or `\pmbset`. However, the approach is unfeasible for commands depending on many parameters: can you imagine listing all the TikZ options in the context? Not to mention that a particular picture usually only depends on a small subset of these options — by and large, TikZ externs would get recompiled too often if the context contained all TikZ options.

   [31]I have toyed with the idea of splitting (using `pgfkeys` key filtering) the given options into outer options, relevant for the outer command, and inner options, relevant for the inner command, and only passing the inner options to the inner command. The thought was that would (i) avoid recompiling the extern when only outer options change, as these options don't affect the inner command, and (ii) avoid applying the inner options when utilizing the extern, as these options don't affect the outer command. However, it then hit me that the end-user might define a style which incorporated both inner and outer options — I know I do this with my `tcolorbox`es.

```
  \NewDocumentEnvironment{poormansbox}{ o +b }{% the outer part of the definition
    \pmbset{#1}%
    \pmb@before
    \@poormansbox[#1]{#2}%
    \pmb@after
  }{}

  \newcommand\@poormansbox[2][]{% the inner part of the definition
    \ifpmb@frame\expandafter\fbox\else\expandafter\@firstofone\fi{%
      \begin{minipage}{\pmb@width}%
        #2%
      \end{minipage}%
    }%
  }

  \mmzset{
    auto=\@poormansbox{% submit the *inner* command to automemoization
      args=om, memoize,
    },
  }
```

`\mmzset` command will not work unless surrounded by `\makeatletter` and `\makeatother`, as it refers to an internal control sequence containing `@`. Well, Memoize offers `auto csname`, `activate csname` and `deactivate csname`, so that `@` catagory code manipulations can be omitted by writing `\mmzset{deactivate csname=@poormansbox}`, but still.

Another downside could occur when you use the same (automemoized) internal command in service of several user interface commands. For the sake of illustration, assume we have also defined an UI-macro `\pmb` which again relies on `\@poormansbox`. How is the author to deactivate automemoization of `\pmb` but leave the `poormansbox` environment intact? This is how: `\mmzset{auto=\pmb{args=m, nomemoize}}`. Again, counter-intuitive; the author expects `\mmzset{deactivate=\pmb}` to work.

One other consequence of this approach is that the code included in the c-memo (if `include source in cmemo` is in effect) will not faithfully reflect the source: as shown in the c-memo listing below, it will contain `\@poormansbox{...}` instead of `\begin{poormansbox}...\end{poormansbox}` — even if this might actually count as an advantage, as the discrepancy will at least inform the author who refuses to read the fine material accompanying our `poormansbox` that something funky is going on.

the c-memo of the last `poormansbox` environment

```
\mmzMemo
\global \mmzContextExtra {}%
%
\mmzSource
\@poormansbox [ width=.6\linewidth , frame, before=\noindent \llap {---}, after=--- ]
            {\lipsum [65]}
```

In a nutshell, automemoizing an internal command might be counter-intuitive for the author. But the core idea — to support memoization of a resistant command by splitting its definition into the outer and the inner command — is sound, and we will elaborate on this idea in section 4.4.4, where we will revisit our `poormansbox` example and develop a variant of this environment which is both memoizable and user-friendly.

4 Under the hood

This chapter is written for three audiences: a curious user who wants to know how Memoize does what it does; a package writer who wants to support Memoize in a tricky situation; and myself, lest I forget why I made the design decisions that I made.

4.1 The entry point

From the author's perspective, the functionality of this package is entered either through the manual memoization commands (macro `\mmz` and environment `memoize`), or via automemoization. And while that is correct, those user interface entry points merely determine what code is submitted to memoization, and set any options specific to the upcoming memoization. The real fun starts with command `\Memoize`, which is eventually executed by both manual and automatic memoization.

Not every call to `\Memoize` results in memoization. Calling this macro has three possible outcomes. It can result in *memoization*, which produces the memos and externs; in *utilization* of the result of an earlier memoization (which boils down to inputting the memos); or in *regular compilation*, whereby the code is compiled (almost)[32] as if Memoize was not there. Which outcome obtains depends on several factors. The decision logic is depicted below, and note that you can `trace` the action on the terminal.

Is Memoize enabled?
(`\ifmemoize`)

yes — no

Are we currently undergoing memoization?
(`\ifmemoizing`)

regular compilation

yes — no

regular compilation

Is the
`recompile`
mode in effect?

yes — no

memoization

Do the relevant
memos and externs exist?

yes — no

utilization
(of the cc-memo)

Is the
`readonly`
mode in effect?

yes — no

regular compilation   memoization
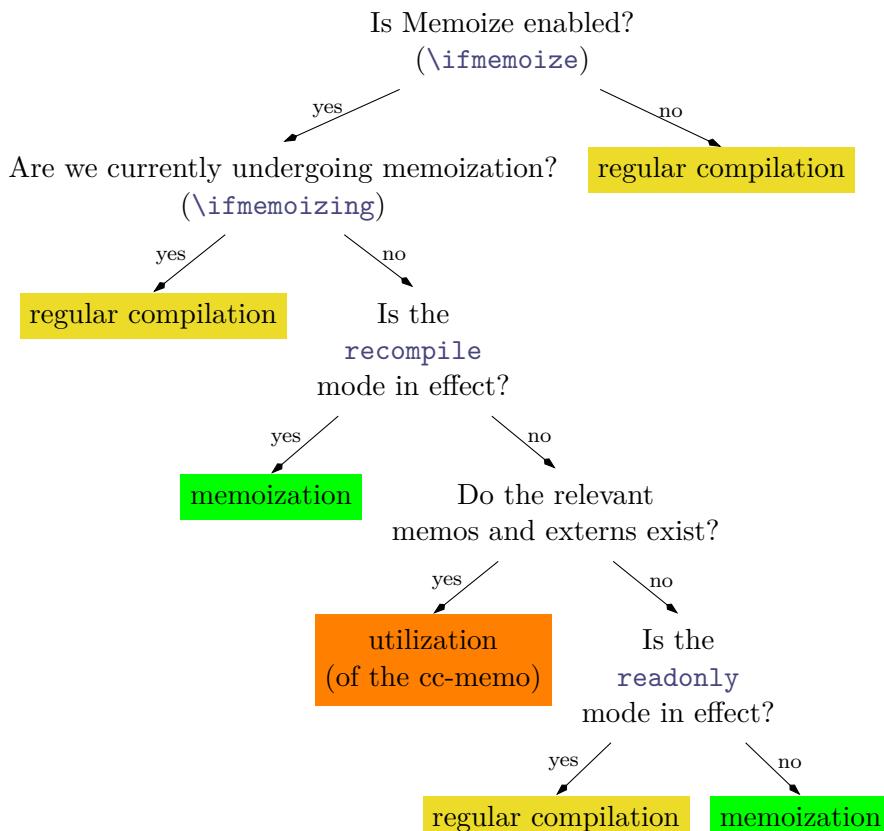
As the memoization options were already set by the user interface entry points, you might expect, quite reasonably, that `\Memoize` takes a single argument, the code submitted to memoization. After all, what more does it need? Clearly, executing this code is what produces the typeset material, and to detect whether the code has "changed" (in order to recompile the memos and externs), we compute the MD5 sum of this very code, don't we? Well, the reality is a bit more complicated. When it comes to

---

[32]This is absolutely true for memoized code which is "contained" in the sense of not peeking into the input stream following the memoized code. In general, code which fails to satisfy this containment requirement is most likely simply not memoizable; but there are borderline cases. For example, `\ignorespaces` at the end of some code will have the expected effect in the absence of Memoize, but no effect when executed either during memoization or regular compilation under Memoize, simply because it will hit some code belonging to Memoize rather than the continuation of the document. Memoize offers the `ignore spaces` provision to work around this specific problem.

automemoized commands, the code which the MD5 sum is computed off of (and which is displayed in the c-memo if `include source in cmemo` is in effect) is not exactly the same as the code we compile (during either memoization or regular compilation). We'll see what the difference is in section 4.5; what matters here is that we must provide `\Memoize` with both and that this macro therefore takes two arguments: the *identification code*, which the MD5 sum is computed off of, and the *executable code*, which, well, is the code that gets executed during memoization (or regular compilation).

Let's illustrate this with an example which is probably entirely useless (but don't worry, we'll get to a realistic example in section 4.5). We first memoize some text manually, using command `\mmz`, and then do something very stupid: we use this very text as the identification code for the following `\Memoize`, even if the executable code of that command is completely different. The second line of the typeset output should convince you that the first argument to that command was really used to produce the extern; and one further compilation should convince you that the first argument was indeed used to identify the extern: the extern produced by `\mmz` was overwritten by the extern produced by `\Memoize`, in the fashion of the `progressbar` example from section 3.4.

---

memoize-internal.tex

```
% The following line internally invokes
% \Memoize{Will this get memoized?}{Will this get memoized?}
\mmz{Will this get memoized?}

\begingroup
\Memoize{Will this get memoized?}{Something entirely different got memoized!}
% \endgroup % Don't uncomment! \Memoize already closed this group!
```

document page (after the first compilation)	document page (after the second compilation)
Will this get memoized? Something entirely different got memoized!	Something entirely different got memoized! Something entirely different got memoized!

The example above also illustrates a(nother) peculiar feature of `\Memoize`. `\Memoize` does not open a new TeX group, but it *expects a group to be opened prior to calling it*, as it will issue an `\endgroup` at some point. Specifically, the memoization group will be closed before regular compilation or utilization, but after memoization. If you want to know why, read the boxed text below.

\Memoize and grouping

One important desideratum behind the design of Memoize was that using the package should disrupt the original, Memoize-less compilation as little as possible. In particular, if the memoized code contains local assignments whose effect (in the original compilation) persists into the rest of the document (until the end of the surrounding TeX group, of course), wouldn't one want these local effects to persist when Memoize is around, as well? Fortunately, most memoized code does not have persistent local effects (at least for me, it is usually environments, like `tikzpicture`s, that I want to memoize, and environments introduce a group anyway) — fortunately, because there are design reasons for enclosing memoization in a TeX group (or two), and this enclosure will of course cancel the effect of local assignments in the memoized code.

For one, the user interface memoization commands, such as `\mmz` and automemoized commands, allow for options specific to a particular piece of memoized code (the options given as the optional argument to manual memoization, the next-options and the auto-options), and to delimit their effect, it makes most sense to apply them in a group. I have toyed with the idea of working around the introduction of a group by manually saving and restoring all the options, but I quickly gave up on this line of thought. For one, manually saving and restoring the options would be cumbersome and error-prone, and probably also slower than using the group. But even worse, all that work would not really solve the problem of the persistence of local effects, because memoization itself introduces a group, as well: during memoization, the typeset material is collected into a box, and opening a box introduces a group. In some particular situations, this could be avoided by typesetting the memoized code as-is and collecting the resulting material using `\lastbox`, but this approach cannot work in general. In general, memoization will take

---

place in a group, so the issue of local effects must be addressed in some other way. Memoize offers the following workaround: during memoization, the memoized code can (globally) add code to the `after memoization` hook, which gets executed immediately after closing the memoization group.

Does this mean it would be best if the user interface memoization commands straightforwardly surrounded `\Memoize` by `\begingroup` and `\endgroup`? For example, `\mmz` would open the memoization group, let `\Memoize` do its work, and then close the group. Not really. Remember that memoization is not the only possible outcome of calling `\Memoize`. Perhaps we can at least retain the local effects of a regular compilation, and of utilization?

We can, by finely tuning the timing of the memoization group closure within `\Memoize`. This command is designed to close the memoization group after memoization, but before regular compilation and utilization. Closing the group after memoization makes sure that the given options are in effect during this process. By closing the group prior to regular compilation, regular compilation of the memoized code (which takes place when Memoize is disabled, for example) is guaranteed to have (almost, see footnote 32) exactly the same effect as the compilation of that code in absence of Memoize; in particular, the effect of any local assignments will persist into the rest of the document. Finally, closing the group before utilization simplifies the construction of the memo in the cases where we need to replicate local effects of the memoized code — the group closed, there is no need to smuggle local assignments out of a memo.

4.2 Memos

Up until now, we have pretended that there is a single kind of a memo file. In truth, there's two kinds: *code memos*, or *c-memos* for short; and *code–context memos*, or *cc-memos* for short. In this section, we will learn what they are for, and how they look like — and also a bit on how they are produced, even if the details on that will have to wait until section 4.4.

We will see that when Memoize utilizes memos, c-memos are processed first. But conceptually, cc-memos are more important, so we will start the discussion with these.

4.2.1 Cc-memos (and extern inclusion)

When it is input, a cc-memo replicates the effect of the memoized code. This includes the reproduction of its visual output, which takes the form of inclusion of any externs produced by memoization. And yes, you got the implication right: a cc-memo can have any number of associated externs, including zero, even if the most common case is that of exactly one extern per cc-memo. The number of externs mostly depends on the memoization driver (see section 4.4); the default driver always produces exactly one extern.

A cc-memo is located in the directory determined by key `prefix` (everything up to the final `/` in the value of this key, or the current directory if this value contains no `/`). You can recognize it by its filename, which has the following form (⟨*prefix name*⟩ is everything following the final `/` in the value of `prefix`):

$$\langle prefix\ name\rangle\langle code\ md5sum\rangle\text{-}\langle context\ md5sum\rangle\texttt{.memo}$$

In fact, this is how Memoize recognizes — or rather, searches for — a cc-memo as well: Memoize will utilize a cc-memo when the code and the context MD5 sum computed during an attempted utilization match the code and the context MD5 sum computed during some previous memoization (for details on the context MD5 sum, see section 4.2.2). In detail, a cc-memo is created at the end of memoization, at which point Memoize computes the MD5 sum of the memoized code and the MD5 sum of the context, and writes the results of memoization into the cc-memo identified by (the prefix and) these two MD5 sums. And when Memoize, on a subsequent compilation, encounters a piece of memoized code, it again computes the MD5 sum of that code and the MD5 sum of the context, and tries to input the cc-memo identified by (the prefix and) these two MD5 sums. If the inputting is successful, we have utilized the cc-memo (which in the typical case amounts to including the one associated extern); if the cc-memo cannot be found, Memoize starts the memoization process, which creates the memos and the externs.

Let us take a look at the contents of a cc-memo in detail. Here's a typical cc-memo (it belongs to the titlepage penguin):

```
titlepage.memo.dir/2A2447B6AC5EBF4B454B605A60EFDCB2-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.memo

  \mmzResource{2A2447B6AC5EBF4B454B605A60EFDCB2-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.pdf}%
  \mmzMemo
  \quitvmode
  \mmzIncludeExtern {0}\hbox
    {14.76718pt}{16.24402pt}{0.65717pt}{72.26999pt}{72.26999pt}{72.26999pt}{72.26999pt}
  \global \expandafter \advance \csname pgf@picture@serial@count\endcsname 1\relax %
  \mmzEndMemo
```

A cc-memo begins by listing the externs which the memo will (actually, might) attempt to include into the document. When the cc-memo is input, each `\mmzResource` command checks if the given extern exists. If some existence check fails, Memoize enters the memoization mode, same as if the cc-memo itself did not exist. If all the resources pass the existence check, Memoize inputs the core of the cc-memo, i.e. everything following the `\mmzMemo` marker.

The core might contain arbitrary code, but most often, it will consist of only two commands. The first one is `\quitvmode` and it is included if the extern was `capture`d into a horizontal box (which is the usual situation). The second one is `\mmzIncludeExtern`, and it is this command which actually includes the extern into the document upon inputting the cc-memo. The core code is executed without

41

introducing any groups, i.e. the effect of any local assignments in the cc-memo will persist into the code following the memoized code.

Command `\mmzIncludeExtern` takes nine parameters. The first is the sequential number of the extern associated with the cc-memo, starting with 0; usually, this is simply 0 as most memos are associated with a single extern. The second one is a `\hbox` or `\vbox`, noting the type of the box the memoized code was externalized into. The next three numbers are the expected width, height and the depth of the extern. Finally, we have the four `padding` amounts (left, bottom, right and top). We should arrive at the expected size after trimming the extern PDF by the padding amounts; Memoize will complain if we don't.

Let's look at a more interesting cc-memo. Using the advising framework, described in section 4.5, Memoize hacks `\label` to support `\label`s inside memoized code — the following code "just works."

```
label.tex

The titlepage Ti\emph{k}Zlings are:

\begin{memoize}
  \begin{minipage}{5em}
    \tikzset{x=1.3ex, y=1.3ex, baseline=0.5ex}%
    \begin{enumerate}
    \item\label{item:penguin} \tikz\penguin;
    \item\label{item:koala} \tikz\koala;
    \item\label{item:owl} \tikz\owl;
    \end{enumerate}
  \end{minipage}
\end{memoize}

The penguin lives in item \ref{item:penguin}.
```

document page (compilation 1)	document page (compilation 2)	document page (compilation 3)
The titlepage Ti*k*Zlings are:	The titlepage Ti*k*Zlings are:	The titlepage Ti*k*Zlings are:
1. 🐧	1. 🐧	1. 🐧
2. 🐨	2. 🐨	2. 🐨
3. 🦉	3. 🦉	3. 🦉
The penguin lives in item **??**.	The penguin lives in item 1.	The penguin lives in item 1.

Everything seems normal — after the first compilation, we get "**??**" because the label has not made it into the `.aux` file yet, but in subsequent compilations, we learn where the penguin lives — but it is far from normal under the hood. If we de-hacked `\label` by writing `\mmzset{deactivate=\label}`, the third compilation (and subsequent compilations) would revert to "**??**". Why would that happen? The memoized code containing the `\label`s is only executed in the first compilation; in the subsequent compilations, we're simply inputting the cc-memo, so the memoized code, including any `\label`s in contains, is not compiled, and the labels don't get into the `.aux` file anymore.

The `\label` hack deploys Memoize's ability to put arbitrary code into the cc-memo. During memoization, the memoized code may add arbitrary code to register `\mmzCCMemo`, and the contents of this register at the end of the memoization form the free-form part of the cc-memo.[33] When the hacked `\label` is encountered during memoization, it appends `\mmzLabel{⟨label name⟩}{⟨current label value⟩}` to `\mmzCCMemo`, so this command winds up in the cc-memo. It is then a simple job for `\mmzLabel`, executed when the cc-memo is input at subsequent compilations, to temporarily store ⟨*current label value*⟩ (i.e. the contents of `\@currentlabel` at the time the `\label` was invoked) back into `\@currentlabel` and to execute `\label{⟨label name⟩}`. In effect, any `\label` command contained within the memoized code is executed at every compilation, even if the memoized code itself is not compiled.

---

[33]This is also how the above-described code containing `\mmzIncludeExtern` gets into the cc-memo. The code is produced by `\mmzExternalizeBox` and appended to `\mmzCCMemo` by the default memoization driver `\mmzSingleExternDriver`; see section 4.4 for details.

```
label.memo.dir/EB19BE685000E2DF39C76F321E7E2792-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.memo
   \mmzResource{EB19BE685000E2DF39C76F321E7E2792-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.pdf}%
   \mmzMemo
   \quitvmode
   \mmzLabel{item:penguin}{1} \mmzLabel{item:koala}{2} \mmzLabel{item:owl}{3}
   \mmzIncludeExtern {0}\hbox
     {50.00008pt}{28.89412pt}{23.89412pt}{72.26999pt}{72.26999pt}{72.26999pt}{72.26999pt}%
   \mmzEndMemo
```

We will continue the discussion of `\label` in section 4.2.3 using a funkier example.

4.2.2   C-memos (and context)

As explained in the previous section, a cc-memo belonging to a piece of memoized code is identified by two MD5 sums: the MD5 sum of the memoized code, and the MD5 sum of the associated context. However, when Memoize encounters some code submitted to memoization, the context expression is not yet fully known, as it may be adjusted by the memoized code itself during memoization — and this potential adjustment is crucial for `\ref` and friends to work as advertised (see section 3.3). Upon being invoked, Memoize therefore cannot immediately attempt to input the cc-memo; it needs to first learn about the context adjustments. Here's where c-memos enter the picture: *the primary job of a c-memo is to store the context adjustments made by the memoized code.* Let's see how this works in detail.

Same as cc-memos, c-memos are located in the directory determined by key `prefix`, and their filenames start by ⟨*prefix name*⟩ determined by the same key. However, a c-memo belonging to some memoized code is identified by the MD5 sum of that code alone:

⟨*prefix name*⟩⟨*code md5sum*⟩.`memo`

The c-memo is created at the end of the memoization process. At that time, the context expression is fully known, as the memoized code was already processed. Even more, Memoize keeps track of both the state of the context expression prior to memoization, stored in token register `\mmzContext`, and of the *additions* to the context expression made by the memoized code, which are stored in token register `\mmzContextExtra`. (Incidentally, key `context` automatically adapts to the situation by appending to `\mmzContext` outside memoization and to `\mmzContextExtra` during memoization.) The complete context expression is the concatenation of the contents of these two registers, but it is only the context expression additions, i.e. the contents of `\mmzContextExtra`, which Memoize stores into the c-memo, with the idea that during subsequent compilations, the initial context (`\mmzContext`) will be set up again via "normal" compilation, while inputting the c-memo will restore the additions, jointly reconstructing the complete context expression associated with a piece of memoized code to what it was at the end of memoization.

We can now complete the picture of a utilization attempt started in section 4.2.1. Memoize begins by trying to input the c-memo; this can be done as the c-memo can be identified based solely on (the MD5 sum of) the memoized code. If the c-memo does not exist, Memoize starts the memoization process, which will produce the memos and the externs. But if it does exist, inputting it reconstructs the context expression to the state at the end of memoization. Therefore, as the MD5 sum of the *expansion* of the context expression *at the end of memoization* is baked into the cc-memo filename, trying to load the cc-memo identified by (the prefix, the code MD5 sum and) the MD5 sum of the *expansion* of the context expression *at attempted utilization* will succeed precisely when the context remained unchanged from memoization to attempted utilization.

All this might have sounded very complicated, but in the end, most c-memos are quite boring, the titlepage penguin's c-memo shown below being no exception. A c-memo starts with the `\mmzMemo` marker, which is always followed by a (global) assignment to token register `\mmzContextExtra`, holding the context expression additions. As promised, the c-memo below is boring: it assigns an empty token list to this register, leaving the context expression as-is. Next comes the free-form part of the memo. Below, it is boringly empty as well (just the percent sign), but in principle, it will contain any code gathered in register `\mmzCMemo` during memoization; see 4.2.4 for an example. A c-memo is concluded by an optional part consisting of the `\mmzSource` marker, followed by the memoized code.

The source code section is not used by Memoize in any way and can be switched off by `include source in cmemo`=false; it is included by default so that an interested user can know which code produced which memo, which can be useful if one wants to trigger recompilation of an extern by deleting the corresponding memo. Incidentally, any newlines in the source code are lost in the c-memo replica (unless `verbatim` is in effect), but we will only see this once we arrive at the `beamer` example below.

```
titlepage.memo.dir/2A2447B6AC5EBF4B454B605A60EFDCB2.memo

\mmzMemo
\global \mmzContextExtra {}%
%
\mmzSource
\tikz \penguin ;
```

A c-memo of code containing a cross-reference will prove more interesting. The following c-memo was produced by the `ref` example from section 3.3. As we know from that section, when a `\ref` (as hacked by Memoize's `ref` key) occurs in some memoized code, it appends the cross-reference to the context. In the c-memo below, this is reflected by the (global) assignment of an expression containing the cross-reference macro to token register `\mmzContextExtra`, holding the context expression additions.

```
ref.memo.dir/90F2CE242AE52CAA56DEFFB44D8F8FFB.memo

\mmzMemo
\global \mmzContextExtra {\detokenize {r@item:penguin}={\expandafter \meaning \csname
  r@item:penguin\endcsname },}%
%
\mmzSource
\tikz [baseline]\node [draw=red,thick,fill=yellow,anchor=base]{\ref {item:penguin}};
```

4.2.3 More on `\label`

A `\label` inside memoized code works out of the box in the usual situation when label value is fully determined by the memoized code, as in the example in section 4.2.1, where the memoized code contained the outermost (and only) `enumerate` environment. However, the out of the box approach does not work if the label value is (fully or partially) determined outside the memoized code. To illustrate the problem, and some potential solutions, we define two very simple enumeration environments, `listi` and `listii`, which use counters `counti` and `countii`, and which are intended as the outer and the inner environment, respectively. Our interest here is in the inner environment, `listii`. While it prefixes each item by an indented `\thecountii`), the label is a composite of both counters: `\thecounti\thecountii`. The label is stored into `\@currentlabel`, so referencing works as usual. However, problems arise when we automemoize the inner environment.

```
label+.tex

\mmzset{
  auto={listii}{memoize,
    % ...
  },
}
% ...
\begin{listi}
\item pets:
  \begin{listii}
  \item\label{item:dog} dog
  % ...
\end{listi}
The dog can be found in (\ref{item:dog}).
```

document page

1. pets:
   a) dog
   b) cat
2. domestic:
   a) cow
   b) sheep
3. wild:
   a) tiger
   b) lion
The dog can be found in (1a).

While the result looks fine at first, changing the order of `listii` environments, for example by moving "pets" below "domestic", will result in a problem: the reference at the bottom will remain unchanged. This is so because the reference text is baked into the cc-memo, as shown below.

label+.memo.dir/9A04214725FF802E62550FBDCDB15249-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.memo

```
\mmzResource{9A04214725FF802E62550FBDCDB15249-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.pdf}%
\mmzMemo
\csuse {par}\mmzLabel {item:dog}{1a}\mmzIncludeExtern {0}\vbox
  {345.0pt}{20.39996pt}{3.60004pt}{72.26999pt}{72.26999pt}{72.26999pt}{72.26999pt}%
\mmzEndMemo
```

How can we remedy this? The manual option is to force the recompilation of the extern by putting an (invisible) reference to the outer item into the inner item: add `\label{item:pets}` to item "pets" and refer to it at "dog" by `\mmzNoRef`{item:pets}.⁰

An automatic variant of the recompilation solution is to add `\@currentlabel` to the context upon memoizing `listii`. This can be achieved by adding `context={@currentlabel={\csuse{@currentlabel}}}` to the `auto` declaration for `listii`. The downside of this approach is that every `listii` will get reexternalized upon movement, whether it actually contains a label or not.⁰

In fact, given that the externs produced by the inner environment do not contain the value of the outer counter, it seems wasteful to recompile any extern just to change the reference. And indeed, it is possible to avoid this, but the approach unfortunately requires adapting the inner environment code (and this is why I have not illustrated the problem using an environment of an elaborate package like `enumitem`). The idea is to "unbake" the reference to the outer item in the cc-memo. We can achieve this by changing `listii` to define `\@currentlabel` to be `\unexpanded{\thecounti}\thecountii`. Under this definition, the cc-memo will contain `\mmzLabel {item:dog}{\thecounti a}`, and rearranging `listii` environments will produce (upon two compilations, of course) the correct reference without recompiling the extern. Note again, however, that this solution can only work when the value of the outer counter does not appear in the extern, i.e. it would not work the "dog" item was prefixed by `1a)` rather than simply `a)`. In those cases, one should deploy one of the other solutions.⁰

The final solution, presented below, is an elaboration on the second one. Rather than append `\@currentlabel` to the context immediately upon beginning to memoize environment `listii`, we will at that point redefine `\label` to do that. In effect, changing the location of `listii` will only recompile it if it contains a `\label`.

As announced, we redefine `\label` once the memoization of `listii` begins, so within `at begin memoization`.[34] However, we do not redefine `\label` directly, as Memoize advises this control sequence out of the box (see section 4.5 for details). What we redefine is the command which the advising framework executes instead of `\label` — its so-called `outer handler` — and we do this by calling `\AdviceSetup`, the low-level variant of the familiar key `auto`.[35] The first argument of `\AdviceSetup` is the installation path (`/mmz`), the second one the command or environment we are submitting to the framework (`\label`), and the third one the setup *code* — here lies the biggest difference between `auto` and `\AdviceSetup`: the former expects a keylist, and the latter TEX code which directly manipulates settings macros like `\AdviceOuterHandler` (for the full list, see section 4.5 or 5.6.1).

Within `\AdviceSetup`, we prefix (using macro `\preto` of package `etoolbox`) the original outer handler `\AdviceOuterHandler` by code which causes `\outerlabeltocontext` to be executed at the end of memoization (by globally appending this macro to `\mmzAtEndMemoizationExtra`; "Extra" because we're appending during memoization). It is `\outerlabeltocontext` which then appends `\@currentlabel` to the context (`\mmzContextExtra`; again, "Extra" because we're appending during memoization),[36] and it is crucial that this happens at the end of memoization rather than when `\label` is executed. When `\label` is executed, we're inside an inner item, and `\@currentlabel` refers to that item, while at the end of memoization, the value of this macro equals the value at the beginning of memoization, namely the label of the outer item. While the outer list remains unshuffled, the value of `\@currentlabel` that

---

[34]Another generally good location for such redefinitions is among the auto-options of `listii`. We could include an `auto\label{⟨...⟩}` there, or a `/utils/exec` with `\AdviceSetup`. However, in this particular case this would be wasteful, as it would be applied regardless of whether memoization will take place or not, whereas we only need the redefined `\label` when memoizing.

[35]We could have also used `auto`, but we don't, because (a) `\AdviceSetup` is faster, (b) it is easier to prepend material to a handler using the low-level interface, and (c) I wanted to showcase `\AdviceSetup`.

[36]As a courtesy, we clear out macro `\outerlabeltocontext` once it did it's job, so that multiple `\label`s do not include multiple `\@currentlabel`s into the context. But the code would work even without this addendum, can you see why?

```
┌─ label+.tex ─────────────────────────────────────────────────── 📎 ─┐
│   \mmzset{                                                           │
│     auto={listii}{memoize,                                          │
│       capture=vbox,                                                  │
│       at begin memoization={%                                       │
│         \csuse{par}\gtoksapp\mmzCCMemo{\csuse{par}}%                 │
│         \AdviceSetup{/mmz}\label{%                                  │
│           \preto\AdviceOuterHandler{%                               │
│             \gappto\mmzAtEndMemoizationExtra{\outerlabeltocontext}  │
│           }%                                                         │
│         }%                                                           │
│       },                                                             │
│     },                                                               │
│   }                                                                  │
│   \def\outerlabeltocontext{%                                        │
│     \gtoksapp\mmzContextExtra{@currentlabel={\csuse{@currentlabel}}}% │
│     \let\outerlabeltocontext\relax                                  │
│   }                                                                  │
└─────────────────────────────────────────────────────────────────────┘
```

contributes to the context MD5 sum during the utilization attempt will therefore match the value which contributed to the context MD5 sum during memoization, resulting in matching MD5 sums and therefore in actual utilization of the extern; once the outer list is shuffled, this will cease to be the case and the extern will be recompiled.

4.2.4   The Beamer support explained

The implementation of `per overlay`, which makes memoization sensitive to Beamer overlays, provides an example of a complex interaction between various components of memoization. At the core, the Beamer support works by adjusting the context, but we will also have the occasion to observe the free-form part of the c-memo, add a bit of extra code to the cc-memo, and deploy several memoization hooks. We will show the complete Beamer support code later on; let us build our understanding of that code step by step. (Before you read on, you might want to refresh your memory about the `beamer` example from section 2.7, as we will refer to it in the present section.)

The core idea behind `per overlay` is to append the current beamer overlay number to the context:[37] `context={overlay=\csname beamer@overlaynumber\endcsname}`. This makes Memoize produce a separate extern for each overlay. However, only the first of these externs will get utilized on subsequent compilations, in general at least. Even worse, we will lose each frame whose creation is driven solely by the memoized code. We will lose the second overlay in our example (i.e. in the `beamer` example from section 2.7) as the second overlay was only created because Beamer encountered `only={2}{...}` (resolving to `\only<2>{...}` under the hood) inside the picture code; once we utilize the extern instead of compiling the picture on the first overlay, the `\only` command is not executed anymore, so Beamer thinks it is done with the frame.

As the compilation of our picture is substituted by utilization of its cc-memos, we have to somehow drive the creation of the necessary overlays from these files. An easy way to achieve this is to furnish them with a dummy `\only<`*final overlay number*`>{}`,[38,39] but there is a problem: the final overlay number is unknown when we're memoizing our picture — it is unknown even when we're memoizing the picture on final overlay itself (we simply don't know yet that this overlay will end up being the final one), let alone during the memoization on the first overlay.

The solution exploits the fact that *the c-memo is rewritten at each memoization*: at each memoization of our picture, we store the the *current* overlay number to the c-memo; after all memoizations, the

---

[37]As we saw in section 2.7, it is convenient to execute `per overlay` inside memoized code. But remember, from section 3.3, that when `context` is executed from within the memoized code, its argument winds up in the c-memo. As the c-memo is processed under the normal category code regime, where `@` is not a letter, we have to access `\beamer@overlaynumber` using the `\csname ... \endcsname` construct.

[38]'The final overlay' here should be understood as relative to our memoized picture, i.e. as the final overlay containing the memoized picture.

[39]Actually, putting this `\only` command only into the first cc-memo would suffice, but would be harder to implement.

c-memo will thus contain the number of the *final* overlay containing our memoized picture. To access this number from the cc-memo, we store it as a macro definition, and then use the defined macro, `\mmzBeamerOverlays`, in the overlay specification of the dummy `\only`. Below, you can see all this in code, as the argument to `at begin memoization`.

The implementation of `per overlay` (first attempt)

```
\mmzset{
  per overlay/.style={
    /mmz/context={overlay=\csname beamer@overlaynumber\endcsname},
    /mmz/at begin memoization={%
      \xtoksapp\mmzCMemo{%
        \gdef\noexpand\mmzBeamerOverlays{\beamer@overlaynumber}%
      }%
      \gtoksapp\mmzCCMemo{%
        \only<\mmzBeamerOverlays>{}%
      }%
    },
  }
}
```

A couple of remarks are in order here. First, the definition of `\mmzBeamerOverlays` in the c-memo is global, because it will be accessed from the cc-memo, but the cc-memo is input after closing the memoize group (which the c-memo *is* processed in). Second, using `at begin memoization` makes it possible to use `per overlay` both outside and during memoization: if `at begin memoization` is executed outside memoization, its argument is (locally) stored into hook `\mmzAtBeginMemoization`, to be called at the beginning of each memoization; if the key executed outside memoization, the argument is executed immediately. Third, the Memoize keys in the definition of `per overlay` are prefixed with `/mmz/`, so that this key can be called from `pgfkeys` option lists of other packages, for example the option list of the `tikzpicture` environment, as shown in the example in section 2.7.

I used the above version of `per overlay` for quite a while. In general, it worked as I expected, but there were glitches. Occasionally, the picture would appear on the wrong overlay, or I would get an extra overlay, or perhaps lose an overlay. Eventually, I figured out this happens when I play with the overlay structure of the frame: when I add or remove a `\pause` or similar. In hindsight, it is easy to see what was happening. Once the picture is memoized, it is fixed, forever, which extern will appear on which overlay. I cannot expect the extern–overlay correlation to change just because I added a `\pause` in front of the picture. Furthermore, the number of overlays the memos will drive to be created is fixed as well. If I memoize the picture while it follows a `\pause`, and the picture creates 10 overlays, the c-memo will define `\mmzBeamerOverlays` to 11. So what, if I then remove that `\pause`! The c-memo will still define `\mmzBeamerOverlays` to 11, and drive the creation of 11 overlays — one too many.

By now, the road ahead is probably clear — we put the `beamerpauses` counter into the context — but we will see there are still obstacles on the way. The issue is that the context is evaluated at the *end* of memoization (so that those cross-references from section 3.3 actually get into it). However, the memoized code might contain a `\pause` or similar itself, and change the value of `beamerpauses`. For one, this means that we have to write down the changed value of `beamerpauses` into the cc-memo; below, we do this using key `at end memoization` (the code given code to this key is executed after the driver but before Memoize writes down the memos and ships out the extern pages; the key itself may be executed either before or during memoization). Furthermore, if the memoized code changes the value of `beamerpauses`, the value of `beamerpauses` at the attempted utilization, which would nicely match the value from the start of memoization, will never match the changed, final value from the end of memoization, in effect preventing our hard-won externs from ever getting utilized.

We therefore have to invent a way to get the memoization-*initial* value of `beamerpauses` into the context.[40] Fine, we store it in a macro,[41] `\mmzBeamerPauses`, when we start the memoization (`at begin memoization`), and put `\mmzBeamerPauses` rather than `beamerpauses` into the context. Will

---

[40]This imposes a requirement on the in-code usage of `per overlay`, namely, that it should be executed prior to any changes of `beamerpauses`.

[41]We must define `\mmzBeamerPauses` globally, because `per overlay` can be arbitrarily deeply embedded in the memoized code.

this work? Not yet, because `\mmzBeamerPauses` is undefined at utilization. We need to set up the context expression so that it will expand to the value of `\mmzBeamerPauses` at (the end of) memoization, and to the value of `beamerpauses` at utilization. This leads to the `pauses=\ifmemoizing...` part of the context expression in (the final version of) the Beamer support code below.

The implementation of `per overlay`

```
\mmzset{per overlay/.style={
  /mmz/context={%
    overlay=\csname beamer@overlaynumber\endcsname,
    pauses=\ifmemoizing
               \mmzBeamerPauses
            \else
               \expandafter\the\csname c@beamerpauses\endcsname
            \fi
  },
  /mmz/at begin memoization={%
    \xdef\mmzBeamerPauses{\the\c@beamerpauses}%
    \xtoksapp\mmzCMemo{%
      \noexpand\mmzSetBeamerOverlays{\mmzBeamerPauses}{\beamer@overlaynumber}}%
    \gtoksapp\mmzCCMemo{%
      \only<\mmzBeamerOverlays>{}}%
  },
  /mmz/at end memoization={%
    \xtoksapp\mmzCCMemo{%
      \noexpand\setcounter{beamerpauses}{\the\c@beamerpauses}}%
  },
  /mmz/per overlay/.code={},
}}
\def\mmzSetBeamerOverlays#1#2{%
  \ifnum\c@beamerpauses=#1\relax
    \gdef\mmzBeamerOverlays{#2}%
    \ifnum\beamer@overlaynumber<#2\relax \mmz@temptrue \else \mmz@tempfalse \fi
  \else
    \mmz@temptrue
  \fi
  \ifmmz@temp
    \appto\mmzAtBeginMemoization{%
      \gtoksapp\mmzCMemo{\mmzSetBeamerOverlays{#1}{#2}}}%
  \fi
}%
```

Are we done? Almost. The final issue is that once we have introduced support for pauses, we have to relativize `\mmzBeamerOverlays` (the final overlay number) to `beamerpause`. So instead of a simple `\gdef\mmzBeamerOverlays` in the first version, we define `\mmzSetBeamerOverlays{⟨beamer pauses⟩}{⟨final overlay number⟩}`, which sets `\mmzBeamerOverlays` only if ⟨beamer pauses⟩ argument matches the value of `beamerpauses` (at its invocation in the c-memo). Well, the macro has some other housekeeping to do as well: it is self-replicating, so that during potential memoization, the `\mmzBeamerOverlays` values belonging to non-current `beamerpauses` values get rewritten into the c-memo.[42] (Fine, there is another fine detail, regarding anti-pollution: the macro also ensures that, relative to {⟨beamer pauses⟩}, only the instance with the greatest ⟨final overlay number⟩ is replicated.)

We are now truly done, and we can look at the final result, the c-memo and the cc-memo belonging to the extern on the first overlay of the example from section 2.7. Specifically, look at the `\mmzSetBeamerOverlays {1}{2}`, which says that the extern chain started when `beamerpauses` equals 1 should continue up to overlay 2, and at the (expanded) context included at the end of the cc-memo,

---

[42]As replication should only occur during memoization (actually, it *can* only occur then, anyway), the instruction to append to the c-memo is appended to the `\mmzAtBeginMemoization` hook (the low-level interface to `at begin memoization`). Note that the assignment to this hook must be local (once local, always local), and it *can* be local because of a little implementation detail: while the c-memo is processed in the memoize TeX group, we don't open an additional group to process it; so the local effects from c-memo will persist into memoization (but not into utilization, because remember that the memoize group is closed before inputting the cc-memo).

courtesy of `include context in ccmemo`, where you can see that the cc-memo will be used when on the first overlay (`overlay=1`) when preceded by no `\pause` command (`pauses=1`).

```
beamer.memo.dir/E2051FB7C5136FAB13436F08554C3F38.memo

\mmzMemo
\global \mmzContextExtra {overlay=\csname beamer@overlaynumber\endcsname ,
  pauses=\ifmemoizing \mmzBeamerPauses \else \expandafter \the \csname
  c@beamerpauses\endcsname \fi ,}%
\mmzSetBeamerOverlays {1}{2}%
\mmzSource
\begin {tikzpicture}[/mmz/per overlay] \node [ellipse, fill=yellow, only={2}{
  pin={[overlay, fill=red, pin edge={overlay, red}]60:An important remark!} }]{An
  important concept}; \end {tikzpicture}
```

```
beamer.memo.dir/E2051FB7C5136FAB13436F08554C3F38-1F0C25A65E527F6006CFC8FACAAB578F.memo

\mmzResource{E2051FB7C5136FAB13436F08554C3F38-1F0C25A65E527F6006CFC8FACAAB578F.pdf}%
\mmzMemo
\quitvmode
\only <\mmzBeamerOverlays >{}%
\mmzIncludeExtern {0}\hbox
  {152.6188pt}{24.08765pt}{0.0pt}{72.26999pt}{72.26999pt}{72.26999pt}{72.26999pt}
\global \expandafter \advance \csname pgf@picture@serial@count\endcsname 1\relax
  \setcounter {beamerpauses}{1}%
\mmzThisContext
padding=(1in,1in,1in,1in),overlay=1, pauses=1,
\mmzEndMemo
```

4.3 Record files

We have seen that externalization is a two-step process in Memoize: as it is impossible for TeX to create multiple PDFs during a single compilation, the externs are first dumped into the document PDF as special extern pages, and only later extracted from the main document into separate PDF files. But extraction requires a complete PDF, which is unavailable even at the very end of the compilation which produces the externs. The externs can therefore only be extracted *after* that compilation (either before or at the beginning of the next one), and this necessitates some form of communication whereby the memoization step informs the extraction step which pages should be extracted from the document PDF and into which (PDF) files they should be stored. This communication is implemented through auxiliary files called *record files*.

4.3.1 The `.mmz` file

By default, Memoize records the information needed for the extraction in a file named ⟨*document name*⟩`.mmz`,[43] henceforth a `.mmz` file. In fact, this file contains more than information on externs created during the last compilation: it records which memos and externs were either used or created during the compilation. The full information contained in the `.mmz` file is used by the clean-up script `memoize-clean.pl` to safely remove stale memos and externs. Let us take a look at the `.mmz` file produced by the titlepage illustration. In fact, we have two versions of this file, as it changes upon the second compilation.

```
titlepage.mmz (after the first compilation)

\mmzPrefix {titlepage.memo.dir/}
\mmzNewCMemo {titlepage.memo.dir/2A2447B6AC5EBF4B454B605A60EFDCB2.memo}
\mmzNewCCMemo {titlepage.memo.dir/2A2447B6AC5EBF4B454B605A60EFDCB2-E778DCCCB8AAB0BBD3%
  F6CFEEFD2421F8.memo}
\mmzNewExtern {titlepage.memo.dir/2A2447B6AC5EBF4B454B605A60EFDCB2-E778DCCCB8AAB0BBD3%
  F6CFEEFD2421F8.pdf}{1}{159.30716pt}{161.44116pt}
\mmzNewCMemo {titlepage.memo.dir/AD85DF8CABE7B570BF9EE388C750890E.memo}
\mmzNewCCMemo {titlepage.memo.dir/AD85DF8CABE7B570BF9EE388C750890E-E778DCCCB8AAB0BBD3%
  F6CFEEFD2421F8.memo}
\mmzNewExtern {titlepage.memo.dir/AD85DF8CABE7B570BF9EE388C750890E-E778DCCCB8AAB0BBD3%
  F6CFEEFD2421F8.pdf}{2}{159.56323pt}{163.74576pt}
\mmzNewCMemo {titlepage.memo.dir/BE512513CDE383A26EC0469517265018.memo}
\mmzNewCCMemo {titlepage.memo.dir/BE512513CDE383A26EC0469517265018-E778DCCCB8AAB0BBD3%
  F6CFEEFD2421F8.memo}
\mmzNewExtern {titlepage.memo.dir/BE512513CDE383A26EC0469517265018-E778DCCCB8AAB0BBD3%
  F6CFEEFD2421F8.pdf}{3}{157.49072pt}{162.97757pt}
\endinput
```

As you can see, the `.mmz` file takes the form of a TeX script (the format was chosen because it facilitated the implementation of the internally triggered TeX-based extraction). The crucial lines in this file, and the only lines used by the extraction script, occur in the first version of the file: they contain command `\mmzNewExtern`, which informs the extraction script that it should extract the document page given by the second argument into the extern file given by the first argument.[44] (The following two arguments provide the expected width and height of the extern; the extraction script may check whether the extern size conforms to these expectations, but this is not crucial, as the extern size is checked every time it is included anyway.)

A `.mmz` file also contains a record of the memos (both c-memos and cc-memos) created in the last compilation; this information is provided by the sole argument of commands `\mmzNewCMemo` and `\mmzNewCCMemo`. And once memos and externs get used in subsequent compilations, the `.mmz` file

---

[43]For TeXperts: the ⟨*document name*⟩ is of course the expansion of `\jobname`.

[44]If you look at the `.mmz` file after extracting the externs using `memoize-extract.pl` without the `--keep` option, you will find that the `\mmzNewExtern` commands are commented out; this is to prevent multiple extractions (even if they are harmless).

will reflect this with `\mmzUsedCMemo`, `\mmzUsedCCMemo` and `\mmzUsedExtern`, as shown in the second version of the file above.

Finally, both versions illustrate that a `.mmz` file always begins with command `\mmzPrefix` and ends with the `\endinput` marker. The argument of `\mmzPrefix` is the prefix to the memo and extern files, as determined by the invocation of key `prefix`. The initial `\mmzPrefix` line is written to the `.mmz` file at the beginning of the document, but an additional `\mmzPrefix` line will occur for every invocation of `prefix` in the document body. Finally, the `\endinput` marker signals that the `.mmz` file is complete.

As mentioned above, the full contingent of `.mmz` file commands is only used by the clean-up script `memoize-clean.pl`. By default, this script removes all memos and externs with the prefix given by `\mmzPrefix` (relative to the directory hosting the `.mmz` file) but those listed by any of `\mmzNewCMemo`, `\mmzNewCCMemo`, `\mmzNewExtern`, `\mmzUsedCMemo`, `\mmzUsedCCMemo` and `\mmzUsedExtern`. Furthermore, the clean-up script will cowardly refuse to delete anything if the `.mmz` file does not end with `\endinput`, as this means that the compilation ended prematurely and that the `.mmz` file might not mention all memos and externs actually used in the document. If given option `--all`, the clean-up script removes even the memos and externs mentioned in the `.mmz` file, and as this option is intended to bring Memoize to a clean slate after any "disasters," the `--all` mode also ignores the potential absence of the `\endinput` marker. Incidentally, the `--all` mode is also the raison d'être for `\mmzPrefix`: while the prefix is usually recognizable from `\mmzNewCMemo` and friends, these commands might not make it into the `.mmz` file in a fatally failed compilation, but it is precisely such compilations that could occasionally require the full clean-up.

4.3.2 Defining a new record type

The `.mmz` file is not the only kind of a record file that can be produced by Memoize. Out of the box, it can also write down the extraction instructions into a makefile or a shell script. These are useful on systems which have to employ the TeX-based extraction but cannot trigger it internally. Running the TeX-based extraction manually would be painful, as it must be done on extern-by-extern basis, so Memoize offers to automate the extraction by a makefile or a shell script; here, the record file is named `memoize-extract.`⟨*document name*⟩`.`⟨*record type*⟩ by default, where ⟨*record type*⟩ is either `makefile`, `sh` (for shell scripts on Linux), or `bat` (for shell scripts on Windows).

To turn on recording of an alternate record type, use key `record=`⟨*record type*⟩. Memoize can record any number of files simultaneously, so saying `record=sh` will produce the shell script alongside `.mmz` (Memoize internally executes `record=mmz` to start recording the `.mmz` file); this should not be a problem, but if you really want to disable the `.mmz` file production, you can say `no record`.

The predefined record types are defined through a generic system open to the user. To define an additional record type, one needs to define, using `pgfkeys`, the relevant hooks of the form

/mmz/record/⟨*record type*⟩/⟨*hook*⟩. The following ⟨*hook*⟩s can be defined (the hooks not needed for the record file type may be left undefined):

- Key `begin` will be executed at the beginning of the document; it will receive no argument. Use it to open the record file.

- Key `end` will be executed at the end of the document; it will receive no argument. Use it to close the record file.

- Key `prefix` will be executed at the end of the document and at every invocation of key `prefix` in the document body; it will receive a single argument, the prefix determined by key `prefix`.

- Keys `new cmemo`, `used cmemo`, `new ccmemo` and `used ccmemo` will be executed after creating or inputting a memo; they will receive a single argument, the full path to the memo.

- Key `used extern` will be executed after an extern was included into the document; it will receive a single argument, the full path to the extern.

- Key `new extern` will be executed after creating creating an extern, more precisely at the end of memoization, right after shipping out the extern page. It will receive a single argument, the full path to the extern, but additionally, Memoize prepares the following macros:

  - `\externbasepath` holds the full path to the extern, but (unlike `#1`) without the `.pdf` suffix;

  - `\pagenumber` holds the "physical" page number of the extern page in the document (the numbering starts by 1);

  - `\expectedwidth` and `\expectedheight` hold the width and the height (total height, i.e. the sum of TeX's height and depth) of the extern page.

Below, we present two simple examples of a record file. The first type simply records the names of all memos and externs used or created by Memoize; the resulting file could be included by `.gitignore` to have `git` automatically ignore all files produced by Memoize. The second type lists the new externs, each preceded by its page number in the `.pdf`; this file could be fed to a custom extern extraction tool.

record-files.tex

```
\newout\mmzfilesout
\mmzset{
  record/files/begin/.code={
    \immediate\openout\mmzfilesout{\jobname.files}%
  },
  record/files/new extern/.code={\immediate\write\mmzfilesout{#1}},
  record/files/new cmemo/.code={\immediate\write\mmzfilesout{#1}},
  record/files/new ccmemo/.code={\immediate\write\mmzfilesout{#1}},
  record/files/used extern/.code={\immediate\write\mmzfilesout{#1}},
  record/files/used cmemo/.code={\immediate\write\mmzfilesout{#1}},
  record/files/used ccmemo/.code={\immediate\write\mmzfilesout{#1}},
  record/files/end/.code={
    \immediate\closeout\mmzfilesout
  },
}
```

record-extern-pages.tex

```
\newout\mmzexternpagesout
\mmzset{
  record/pages/begin/.code={
    \immediate\openout\mmzexternpagesout{\jobname.extern-pages}},
  record/pages/new extern/.code={%
    \immediate\write\mmzexternpagesout{\pagenumber\space#1}},
  record/pages/end/.code={
    \immediate\closeout\mmzexternpagesout},
}
```

Finally, note that (unlike memos and externs) record files are auxiliary files and may be deleted at any time after the extraction of the externs produced in the final compilation — actually, even if these externs were not yet extracted, deleting the record file(s) will merely force their recompilation.

4.4 The memoization process

We now turn to the memoization process itself. The job of memoization is to, while compiling the given code in a regular fashion, prepare the cc-memo (which, when it is input, will replicate the effect of the given code), alongside any externs that the cc-memo will include (these hold the typeset material to be replicated). Clearly, merely compiling the code cannot have this effect (unless that code was written specifically to support memoization; more on this later), and this is why the memoized code is typically wrapped by a *memoization driver*, which can be set using key `driver`. We'll inspect the default memoization driver, `\mmzSingleExternDriver`, in the first subsection, and we will learn how to write specialized drivers in the remaining subsections. But first, let us say some words about a grouping-related TEXnical detail we need to take care about during memoization.

During memoization, we have to collect certain information, like build the contents of the cc-memo. Some of that information might be contributed by the memoized code itself. For example, a `\label` "adds itself" to the cc-memo (by appending to token register `\mmzCCMemo`); a `remember picture` aborts memoization (by issuing `\mmzAbort`); etc. The issue is that the memoized code might open any number of TEX groups; we have no idea how deeply embedded the `\label` or `remember picture` might be. Therefore, we have to collect all the information about the ongoing memoization *globally*: all assignments to `\mmzCCMemo` must be global; `\mmzAbort` sets the underlying conditional globally; etc. (Clearly, all these global variables are initialized at the start of memoization.)

This was the easy part. An additional complication arises with some options which may be set either outside memoization, or during this process. For example, you can append the font size to the context expression in the preamble (see section 3.3), so that the externs will be automatically recompiled when the font size changes, and clearly, this context adjustment should respect TEX grouping; but a `\ref` or some other cross-referencing command in the memoized code needs to append to the context as well, and as this `\ref` occurs *within* the memoized code, the assignment must be global, as explained above.

Mixing the local and global assignments to the token register `\mmzContext`, which holds the (in the actual implementation, local) context expression, will not do. For one, we *do* want to restore the pre-memoization context expression after we have memoized the code, and furthermore, mixing local and global assignments to the same variable is not recommended for save stack reasons anyway.

Memoize addresses this issue by having *two* context registers, `\mmzContext` and `\mmzContextExtra` — when computing the context MD5 sum (which happens at the end of memoization), the two registers are concatenated (the local one comes first). A package writer should know when to use which register, and how. Outside memoization, one should assign to `\mmzContext` — *locally*. During memoization, one should assign to `\mmzContextExtra` — *globally*. The user interface key `context` respects this requirement automatically: it locally appends to `\mmzContext` outside memoization, and it globally appends to `\mmzContextExtra` during memoization. (The same idea is applied to the post-memoization hooks `at end memoization` and `after memoization`.)

4.4.1 The default memoization driver

The default memoization driver, `\mmzSingleExternDriver`, produces exactly one extern, which contains whatever is typeset by the code submitted to memoization. The driver compiles the code into a horizontal or vertical box depending on the value of key `capture`. Let us look at the definition of the driver line by line:

The default memoization driver

```
1  \long\def\mmzSingleExternDriver#1{%
2    \xtoksapp\mmzCCMemo{\mmz@maybe@quitvmode}%
3    \setbox\mmz@box\mmz@capture{#1}%
4    \mmzExternalizeBox\mmz@box\mmz@temptoks
5    \xtoksapp\mmzCCMemo{\the\mmz@temptoks}%
6    \mmz@maybe@quitvmode\box\mmz@box
7  }
```

1. Macro `\mmzSingleExternDriver` (and in fact any memoization driver) takes a single argument, the code to compile. Memoize will call the driver with the code given as the second argument to `\Memoize`, but wrapped in a macro which re-reads it using `\scantokens` when `verbatim` is in effect.

2. If we're capturing into a horizontal box (`capture`=`hbox`), we put `\quitvmode` into the cc-memo — putting it to the very beginning should make sure that any replicated `\label` and `\index` commands refer to the correct page.

3. We compile the given code, storing the typeset material into a box (above, a temporary box called `\mmz@box`). `\mmz@capture` resolves into a box construction command, depending on the value `capture`.

4. Macro `\mmzExternalizeBox` instructs Memoize to externalize the box given as its first argument. However, this macro does not directly produce an extern page or write any instructions into the cc-memo; the road to this final destination is indirect. `\mmzExternalizeBox` has two effects. First, it adds the contents of the given box (above, `\mmz@box`) to an internal box dedicated to holding all the externs produced in this memoization (the contents of `\mmz@box` remain as they are) — it is only at the end of memoization that the contents of this internal box are shipped off to extern pages. Second, `\mmzExternalizeBox` produces the code which will include the extern into the document on subsequent compilations (this will be a call to `\mmzIncludeExtern`, potentially prefixed by `\quitvmode`; see section 4.2.1 for details). This code is stored into the token register gives as the second argument (above, `\mmz@temptoks`), and it is the responsibility of the driver to include it into the cc-memo. (In the interest of full disclosure, `\mmzExternalizeBox` also updates the list of externs produced in this memoization. At the end of memoization, this list is written to the beginning of the cc-memo, resulting in the `\mmzResource` lines preceding the `\mmzMemo` marker.)

5. The construction of the cc-memo is indirect as well. In the third line of the definition, we globally append the extern-inclusion code residing in `\mmz@temptoks` to token register `\mmzCCMemo`. At the end of memoization, the contents of `\mmzCCMemo` are written into the cc-memo, preceded by the `\mmzMemo` marker.

6. We put the typeset material into the document, again preceded by `\quitvmode` when capturing in a horizontal box.

You might wonder why the construction of the extern pages and the cc-memo (and actually, of the c-memo as well) is indirect, as described above.

- For one, the indirect construction facilitates potential abortion of memoization (see section 3.1). With the indirect route, aborting is easy — as nothing was permanently written anywhere yet, Memoize simply skips the final part of the process, where extern boxes are shipped into extern pages and the memo registers written into memo files — and also clean: if `\mmzExternalizeBox` immediately shipped out the extern pages, these pages would remain in the document even in the case of abortion.

- Even more importantly, the cc-memo filename contains the ⟨*context md5sum*⟩ (see section 4.2.1), but the context expression is not yet fully known when memoization starts — remember (from section 3.3) that a `\ref` in the memoized code will update the context! The cc-memo can therefore only be opened at the end of memoization, which necessitates a buffer (i.e. the `\mmzCCMemo` register) for storing its contents during memoization.

4.4.2   Pure memoization

The default memoization driver discussed above is really an externalization driver: it produces a single extern. We now move to examples of drivers with other functions, starting with a pure memoization driver, which does not externalize any typeset output — simply because it does not call `\mmzExternalizeBox` at any point — but rather remembers the result of a (pgfmath) computation (let's pretend that the computation is time-consuming).

```
\def\mmzPgfmathDriver#1{%
  #1%
  \xtoksapp\mmzCCMemo{\def\noexpand\pgfmathresult{\pgfmathresult}}%
  \xappto\mmzAfterMemoizationExtra{\def\noexpand\pgfmathresult{\pgfmathresult}}%
}
\mmz[driver=\mmzPgfmathDriver]{\pgfmathparse{6*7}}%
$6*7=\pgfmathresult$
```

the cc-memo
```
\mmzMemo
\def \pgfmathresult {42.0}%
\mmzEndMemo
```

document page

$6 * 7 = 42.0$

Command `\mmz` above memoizes its mandatory argument with the memoization `driver` set to the previously defined macro `\mmzPgfmathDriver`. Just as the default driver above, `\mmzPgfmathDriver` first executes the given code. However, there is no need to do this in the context of a `\setbox`, as the memoized code, which is obviously expected to consist of a single `\pgfmathparse` call, does not typeset anything: `\pgfmathparse` evaluates the given expression and stores the result into macro `\pgfmathresult`. The driver has two jobs: first, it must store this result into the cc-memo, to be utilized in subsequent compilations; second, because the assignment to `\pgfmathresult` (within `\pgfmathparse`) is local, the driver also needs to somehow smuggle the result out of the `\endgroup` issued by `\Memoize` and thereby make it into the following code (the final line of the example, which typesets the equation). Both jobs are easy enough: the expansion of `\def\noexpand\pgfmathresult{\pgfmathresult}` (in this case, `\def\pgfmathresult{42.0}`) is (globally) appended both to the token register `\mmzCCMemo`, which Memoize later writes into the cc-memo, and to the macro underlying the `after memoization` hook, whose contents are executed *after* closing the memoization group.

Let us consider an alternative implementation of the same goal of memoizing the result of a pgfmath computation, showcasing a couple of useful tricks.

```
\def\mmzSmuggleOneDriver#1#2{% #1 = the macro to smuggle, #2 = the memoized code
  #2%
  \xtoksapp\mmzCCMemo{\def\noexpand#1{#1}}%
  \xappto\mmzAfterMemoizationExtra{\the\mmzCCMemo}%
}
\mmzset{
  auto=\pgfmathparse{
    args=m, memoize,
    clear context,
    driver=\mmzSmuggleOneDriver\pgfmathresult,
  },
}
\pgfmathparse{6*7}%
$6*7=\pgfmathresult$
```

For one, this "embellished" example reminds us that we can list the `driver` key among the auto-options (even if I don't really recommend automemoizing `\pgfmathparse`). But even more importantly, the example shows that the driver consist of more than a single control sequence; the only requirement is that the given driver code will consume the memoized code. In this example, we have developed a generic smuggling driver and applied it to `\pgfmathresult` in particular — `\pgfmathresult` will become the first argument of `\mmzSmuggleOneDriver`, and the memoized code will become its second argument.

In the first version of the example, we have appended the same code to macro `\mmzCCMemo` and to macro `\mmzAfterMemoizationExtra` — no surprise here, as we want the effect of memoization and utilization to be the same. In the embellished version, we advertise another way to achieve the same effect, a way which might be useful for complicated drivers: we simply smuggle out the entire cc-memo.

The idea works even when memoization procudes externs; in that case, however, the driver also has to say `\mmzkeepexternstrue` — conditional `\ifmmzkeepexterns` decides whether Memoize keeps the externs around, in memory, even after shipping them out (but they are always gone at the start of the next memoization).

Finally, remember that the default context expression contains the `padding` values. However, these really have no place in the context expression of some purely memoized code. We have therefore emptied out the context expression using `clear context`.

4.4.3 Multiple externs per memo

In the next example, we show how to produce multiple externs for a single piece of memoized code. The usage case I find most appealing is breaking the typeset material, like a table, across pages — but of course, table-breaking is too complicated an example, so we illustrate the idea by defining command `\countdown`, which counts down from the given number, typesetting each number into its own line. Clearly, if we were to externalize a call to this command using the default memoization driver, page breaking would stop working, as the entire countdown would be seen as a single, unbreakable box. To externalize it properly, the chunks of the countdown that should appear on separate pages must be externalized into separate externs, as shown below.
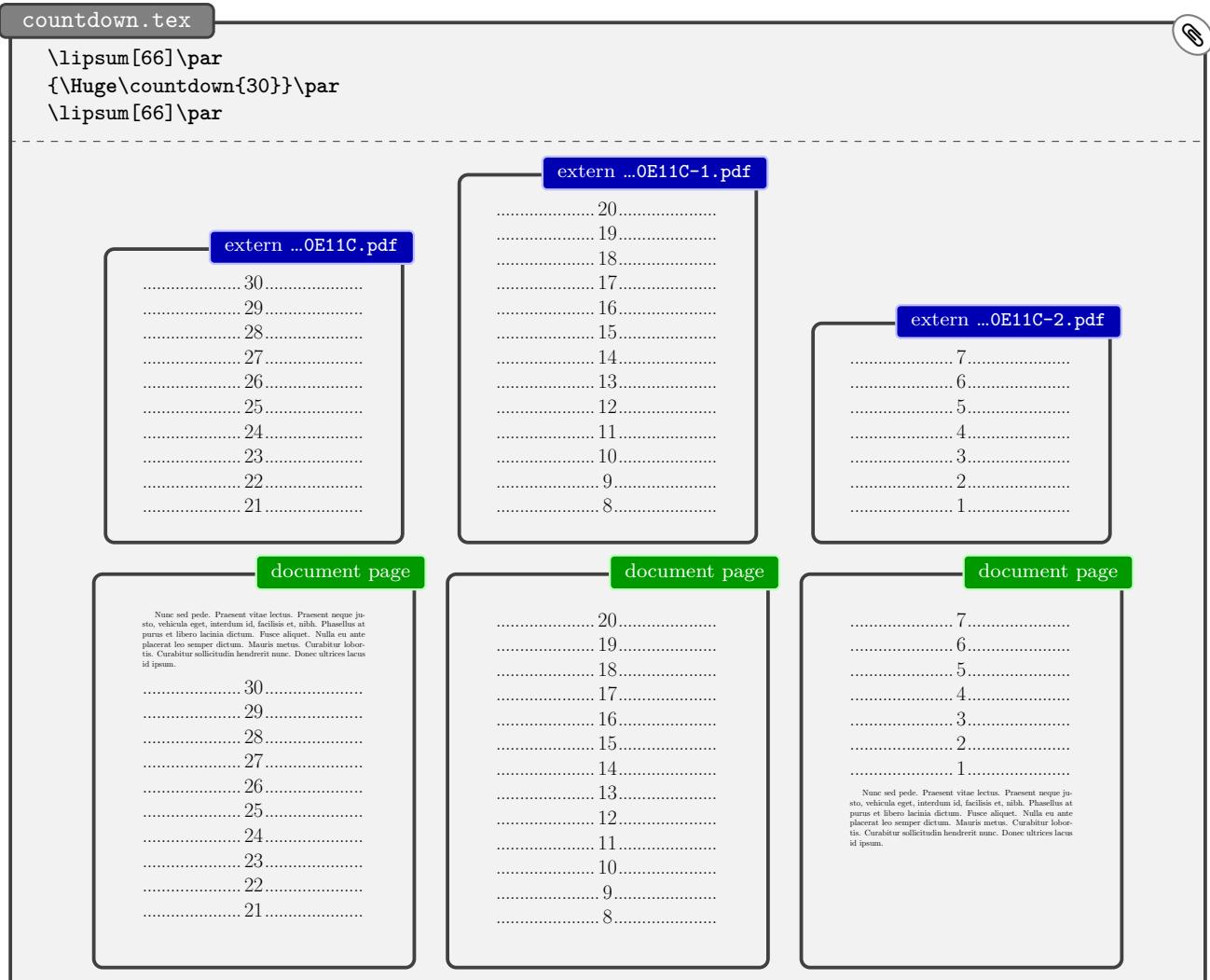
To achieve this, we will have to integrate the memoization driver into the very code of `\countdown`. This approach contrasts sharply with the standard memoization driver, which is simply wrapped around the memoized code. Let's say we have implemented a non-memoization-aware variant of `\countdown` as a loop which gathers the countdown lines into a vertical box, and periodically, when this box holds all the material that will fit onto the page, places it into the main vertical list (i.e. on the page).[45] To

---

[45]Of course, implementing `\countdown` this way would be idiotic; a sane implementation would simply spit out the

have this command support memoization, we have to externalize our box every time we're placing it into the main vertical list. This is precisely what we do in the definition of `\countdowntypeset` below:[46] the final line of this macro adds the material to the main vertical list, and the preceding lines externalize it (the two lines inside `\ifmemoizingcountdown` should be familiar from the definition of the standard memoization driver; we'll explain about the conditional below); note that Memoize automatically deals with the fact that our box is vertical. As a result of having our memoization driver integrated into the loop of the core command, we can create as many externs as necessary, complete with the code in the cc-memo for including each and every one of them on subsequent compilations. Each extern eventually makes it into its own extern file, and note that the filenames of the non-first externs have their sequential number (we start counting at 0) appended to the basename, as shown in the example.[47]

```
countdown.sty (version 1)

\ProvidesPackage{countdown}
\RequirePackage{memoizable}

% ...

\newif\ifmemoizingcountdown
\def\countdowntypeset{%
  \ifmemoizingcountdown
    \mmzExternalizeBox\countdownbox{\toks0}%
    \xtoksapp\mmzCCMemo{\the\toks0}%
  \fi
  \noindent\box\countdownbox\par
}

\def\countdowndriver#1{%
  \memoizingcountdowntrue
  #1%
}

\mmzset{
  auto=\countdown{
    args=m, memoize,
    driver=\countdowndriver,
    context={fsize=\csname f@size\endcsname,textheight=\the\textheight},
    options={context/.expanded={pagetotal=\the\pagetotal}},
  },
}
```

Of course, the chunks of the countdown should only be externalized when the code is actually being memoized, and not, say, when Memoize is disabled or performing regular compilation. (Note that this is a problem that only affects integrated drivers and not wrapped drivers such as the default driver.) The first thought is to detect whether we're undergoing memoization using conditional `\ifmemoizing`, which Memoize sets to true at the start of every memoization. This conditional is used in the run conditions of advice for `\ref` and `\label`, the idea being that they should add stuff to the context (`\ref`) and the cc-memo (`\label`) only when undergoing memoization. However, deploying `\ifmemoizing` in the current example would not be exactly right. It would work well with the main document as it is, but it would fail if `\countdown` was called from a piece of code that was independently submitted to memoization,

---

countdown lines one by one, and let TeX deal with page-breaking. However, remember that we are pretending that we are typesetting (and page-breaking) some complex material, like a table; in such a case, the loop outlined in the main text would make perfect sense.

[46]We omit the definition of the core algorithm of `\countdown` in the listing, because it is mostly irrelevant for our discussion, and only show the memoization-related code.

[47]The `auto` declaration of `\countdown` adds some relevant parameters to the context (see section 3.3). The countdown will be recompiled upon change of either the font size (`f@size`), the text height (`\textheight`), or the height of the material in the main vertical list collected so far (`\pagetotal`). The `\pagetotal` parameter is especially important; including it makes sure that the countdown will be recompiled when it is pushed up or down the page. Also note that we want the context to record the value of `\pagetotal` when the (automemoized) `\countdown` is encountered (rather than at the end of memoization), so we expand it when applying the auto-options.

e.g. `\mmz{\countdown{30}}`.[48] In that case, both the `\mmz` driver and the `\countdown` integrated driver would get executed, resulting in the creation (and in subsequent compilations, utilization) of four externs: first, the `\countdown` driver would externalize each countdown chunk separately, and then, the `\mmz` driver would externalize them, all together, yet again. You can try this out by replacing `\ifmemoizingcountdown` in `\countdowntypeset` by `\ifmemoizing` (and wrapping the `\countdown` call in `\mmz`).

The solution to the `\ifmemoizing` problem deployed in the example is to declare a new, `\countdown`-specific memoization conditional, and set it to true in `\countdown`'s formal driver, i.e. the macro set as the `driver` in the `auto` declaration for `\countdown`. In fact, Memoize can do most of this for you: when we write `integrated driver`=countdown, Memoize creates the countdown-specific memoization conditional and declares the formal driver which sets this conditional to true; you only have to access this conditional in your code, and you should do this using the LaTeX-style conditional `\IfMemoizing`, as shown below.[49]

```
countdown.sty (version 2)

\def\countdowntypeset{%
  \IfMemoizing{countdown}{%
    \mmzExternalizeBox\countdownbox{\toks0}%
    \xtoksapp\mmzCCMemo{\the\toks0}%
  }{}%
  \noindent\box\countdownbox\par
}

\mmzset{
  auto=\countdown{
    args=m, memoize,
    integrated driver=countdown,
    % ...
```

4.4.4 Driver-based memoizable design

In the previous section, we used the integrated driver approach to produce memos including multiple externs, but the approach can be useful for one-extern memos as well, when the extern must be integrated into the document in some special way. We already discussed such situations in section 3.5.2, where we suggested to split a "difficult" command into the outer command and the inner command, and only submit the inner command to automemoization. However, the vanilla flavour of this approach had a negative impact on the user interface to automemoization. In this section, we will deploy the memoization driver to overcome the issue.

Let us revisit the `poormansbox` example from section 3.5.2. Remember that that environment produced a potentially framed box of a certain width, surrounded by some pre- and post-code, and that the issue was that the pre- and the post-code should not be memoized, but rather executed at every invocation of the command, as it was primarily intended to put some stretchable vertical space around the box.

The document source[⟐] and the resulting PDF of the example are the same as in section 3.5.2, so we will not repeat them here, but jump directly into a revised definition of the environment. We will retain the core idea from the original implementation: the outer command will execute the pre- and the post-code, and the inner command will typeset the box. But unlike in the original implementation, we will not automemoize the inner, internal command (this was the source of the author's discomfort) but the outer, user-level command — and we will equip it with a custom memoization driver. The major idea here is to have the driver compose a cc-memo which not only includes the extern, but also executes the outer command.

---

[48]Such embedding occurs more often than you might think. For example, `forest` calls `tikzpicture` under the hood, and both environments are automemoized.

[49]You shouldn't directly use the plain TeX countdown-specific conditional created by `integrated driver` — to prevent accidental access, Memoize doesn't actually name it `\ifmemoizingcountdown` — because this conditional is undefined when Memoize is not loaded, i.e. when only package `memoizable` is in effect. Furthermore, `\IfMemoizing` addresses a problem faced by integrated drivers of potentially recursive commands; we will talk about this in section 4.4.4.

58

```
\NewDocumentEnvironment{poormansbox}{% the environment
    o % the options
    +b % the environment body
}{%
    \poormansbox@outer{#1}{\poormansbox@inner{#1}{#2}}%
}{}

\def\poormansbox@outer#1#2{% the outer command
    \pmbset{#1}% apply the options
    \pmb@before % the pre-code
    #2% this will be either the inner command, or |\mmzIncludeExtern|
    \pmb@after % the post-code
}
\def\poormansbox@inner#1#2{% the inner command
    \setbox0=\hbox{% typeset our product into a box
        \ifpmb@frame\expandafter\fbox\else\expandafter\@firstofone\fi
        {%
            \begin{minipage}{\pmb@width}%
                #2%
            \end{minipage}%
        }%
    }%
    \IfMemoizing[1]{pmb}{% if memoizing the this instance of poormansbox
        \mmzExternalizeBox0{\toks0}% externalize the box
        \xtoksapp\mmzCCMemo{% append to cc-memo
            \noexpand\csuse{poormansbox@outer}% call the outer command
            {\unexpanded{#1}}% the options
            {\the\toks0}% the extern-inclusion code (|\mmzIncludeExtern...|)
        }%
    }{}%
    \quitvmode
    \box0 % put the extern box into the document
}
\mmzset{auto={poormansbox}{memoize, integrated driver=pmb}}
```

In detail, the implementation (partially shown in the `.sty` listing) is as follows. The outer command (`\poormansbox@outer`) first applies the options (`#1`) and then wraps the pre-code (`\pmb@before`) and the post-code (`\pmb@after`) around some arbitrary code (`#2`). During memoization or regular compilation, the outer command is invoked through the `poormansbox` environment, and you can see that in the definition of that environment, the second argument to `\poormansbox@outer` is a call to the inner command (`\poormansbox@inner`; this command takes two arguments, the options and the environment body). During utilization, the outer command is invoked from the cc-memo,[50] and as you can see in the cc-memo listing below, the second argument to `\poormansbox@outer` there is a call to `\mmzIncludeExtern`.

```
\mmzResource{4CF57AD067E58C5F29B2FE463A62E9DE-E778DCCCB8AAB0BBD3F6CFEEFD2421F8.pdf}%
\mmzMemo
\csuse {poormansbox@outer}{ width=.6\linewidth , frame, before=\noindent \llap {---},
  after=--- }{\mmzIncludeExtern {0}\hbox
  {225.31938pt}{52.34444pt}{47.34444pt}{72.26999pt}{72.26999pt}{72.26999pt}{72.26999pt}}%
\mmzEndMemo
```

And how does `\poormansbox@outer` get into the cc-memo, which normally only includes a call to `\mmzIncludeExtern`, you ask? This is the job of the memoization driver, which is in this case integrated into the inner command. The overall shape of the driver is the same as the shape of the standard

---

[50]As you can see, in the cc-memo the outer command is invoked by `\csuse{poormansbox@outer}`. A straight `\poormansbox@outer` would not work because we're in the middle of the document where `@` is not a letter, and including a `\makeatletter` in front of it (and in a group) only works if direct ccmemo input was in effect. Under the default, indirect cc-memo input regime, the core cc-memo is tokenized before `\makeatletter` can take effect.

driver, discussed in section 4.4.1: typeset the extern material into a box, externalize this box, append the extern-inclusion code to the cc-memo, and put the extern box into the document. It is the cc-memo part which interests us right now: unlike the standard driver, we don't simply append the contents of `\mmz@temptoks`, i.e. a `\mmzIncludeExtern` call; we rather append a call to `\poormansbox@outer`, which gets the `\mmzIncludeExtern` call as its second argument (and the options as its first argument).

The core part of the driver, which externalizes the box and appends to the cc-memo, is embedded inside the true branch of conditional `\IfMemoizing[1]{pmb}`. We already used this conditional in section 4.4.3, but without the optional argument. Such usage will not work here, because it is not recursion-safe. Unlike in the `\countdown` situation, one `poormansbox` environment can be embedded in another one (and, in our example, it is). If we deployed `\IfMemoizing{pmb}` in the inner command, the driver would be executed for *both* the outer and the inner instance of the environment, whereas it should really be executed only for the outer instance.

When used in a recursion-safe way, i.e. with the optional argument, `\IfMemoizing` first tests whether the auxiliary command-specific conditional from the previous section is true, and then proceeds to compare the current group level (ε-TeX's `\currentgrouplevel`) to the group level at the start of memoization (which Memoize stored in `\memoizinggrouplevel`). Only if these group levels match do we know that we're working on the outer instance of the environment, and that we should therefore execute the memoization driver. Importantly, though, the two group levels are compared modulo the offset, given as the optional parameter to `\IfMemoizing`: in our example, the offset is 1, because the driver is located inside the `poormansbox` environment, which opens a group — note that 0 zero (no offset) is not the default optional parameter; the absence of the optional parameter indicates that the non-recursion safe method should be used.[51]

4.4.5 Shipout

Memoize is a hypocrite: when it is creating extern pages, it uses `\(pdf)primitive\shipout` to bypass the regular shipout routine of the format, but it is offended if anyone else does that.

Memoize bypasses the regular shipout because the extern pages should really not be modified or discarded by a foreign package. But using the primitive `\shipout` means that extern shipouts can't be detected by another package, at all. To facilitate peaceful coexistence with a potential package which needs to know about our extern pages, we offer public counter `\mmzExternPages` holding the number of externs shipped out so far. And if anyone really needs to *do* something at every extern shipout, they can always (ab)use `/mmz/record/⟨record type⟩/new extern` as a post-extern-shipout hook.

The other side of the story is about Memoize needing to know the "physical" page numbers of its externs in the document PDF — how else are we to extract them? Memoize computes these page numbers by adding the values of several counters: `\mmzRegularPages`, which holds the number of regular shipouts; the above-mentioned `\mmzExternPages`, which holds the number of extern shipouts; and `\mmzExtraPages`, which holds the number of other shipouts. The latter counter should be advanced by a package which, like Memoize, bypasses the regular shipout routine.

LaTeX and ConTeXt kindly provide the number of regular shipouts as publicly a accessible counter, so we define `\mmzRegularPages` as synonymous with their `\ReadonlyShipoutCounter` and `\realpageno`. In plain TeX, we have to hijack the `\shipout` control sequence and count regular shipouts ourselves; as we have to hijack it while it still refers to the `\shipout` primitive, this format provides another reason for preferring Memoize to be loaded early.

---

[51] Even this approach is not completely bullet-proof. It will only work when the inner instance of the command is guaranteed to occur in an additional group, i.e. when our command opens up a group for any free-form code. I will assume that situations which require externalization of a potentially recursive command which, for some reason, cannot open the group before processing a free-form argument, are rare enough to not warrant a generic solution here.

Automemoization is a mechanism that automatically memoizes the result of the compilation of certain commands and environments. Writing Memoize, I went to great lengths to make it flexible, yet easy to use. This resulted in automemoization deploying two specifically developed auxiliary packages: package Advice, which provides a generic framework for extending the functionality of selected commands and environments, and package CollArgs, which provides a command for collection of the arguments conforming to the given (slightly extended) LaTeX3/`xparse` argument specification.

This section lists the considerations which went into designing the system, followed by short tutorials on both auxiliary packages, which include several examples of how Memoize uses the underlying advising framework.

(De)activation   Ideally, all commands and environments where memoization makes sense would support Memoize (or Memoize would support them) and nothing would ever go wrong. In this dream world, memoization would be completely transparent to the author. However, things will go wrong, so at the very least, we need to offer the author a *simple* way to *selectively* switch automemoization on and off. This is achieved by keys `activate` and `deactivate`.

Submission   Of course, there will be commands without official support by either Memoize or the package which defines them; clearly, at the moment when I write this, all commands but `\tikz`, `tikzpicture` and `forest` are such. Or, the author might want to automemoize his or her own command. Ideally, submitting a new command to automemoization would be as simple as `memoize=`⟨*command*⟩, and for environments, this is in fact achievable, although the actual interface is `auto=`{⟨*environment*⟩} {`memoize`}. But simply submitting the name cannot work for commands, because commands are where we encounter the major TeXnical problem with automemoization: we need to somehow collect the arguments of the command — without executing the command itself.

Argument collection using CollArgs   TeX being TeX, automatically determining the scope of a command in general is just plain impossible. Note that inspecting the `\meaning` is not enough in general, because the "real" and the formal arguments of a command can, and quite often do, differ wildly. The author (or the package writer) will need to tell Memoize about the argument structure of the command. And as there is already a nice and general argument specification on the market — I'm obviously referring to the argument specification of package `xparse`, which was recently integrated into the core LaTeX — why not use that? Memoize comes with an auxiliary package CollArgs, which (given the slightly extended `xparse`-style argument specification) collects the arguments of a command into a single entity. All the user needs to write to enable automemoization for a command is thus `auto=`⟨*command*⟩ {`memoize`, `args`={⟨*argument specification*⟩}}. Even simpler, when it comes to commands defined by LaTeX3's `\NewDocumentCommand` or friends, writing `auto=`⟨*command*⟩{`memoize`} will suffice, as the argument specification of these commands can be retrieved by `\GetDocumentCommandArgSpec`.

Weird commands   Not every argument structure can be described using `xparse`'s argument specification, a case in point being `\tikz` with its totally idiosyncratic syntax — and if Memoize won't support `\tikz`, why have it at all? The interface to automemoization must be flexible enough to cover even the craziest commands, and this is why Memoize allows for arbitrary argument collectors. These are defined by the advanced user or package writer and then declared to be used for parsing the argument structure of a command by writing `auto=`{⟨*command*⟩}{`...`,`collector`=⟨*argument collector*⟩}.

Over and above automemoization: handlers   The framework facilitating automemoization must cover more than just that. For one, it sometimes makes sense to automatically *prevent* memoization during the execution of certain commands (as in the `nomemoize` example in section 3.4). It follows that the action performed to an invocation of a command should not be fixed. In the advising framework, implemented by the auxiliary package Advice, we assign each advised command a handler — a command which does the real work of memoizing or whatever. Crucially, the handler and the

collector are independent of each other, allowing a single memoization handler to handle commands with both standard and non-standard argument structure, and allowing a single collector to serve either the memoization or the no-memoization handler.

Over and above automemoization: the outer and the inner handler   Second, unlike the memoization handler set by `memoize`, not all handlers work with the entire argument list of the advised command. Some handlers don't care about the arguments of the advised command at all: `abort` simply aborts memoization whenever the advised command is executed. Other handlers are only intended to advise a single command or a small family of commands, and need to inspect specific arguments of the advised command: for example, `ref` needs to append the internal cross-reference macro to the context, and it of course constructs the name of this macro from the reference key. For all such handlers, it would be plain wasteful to first collect the arguments and then tear them apart to inspect them (or not). The advising framework therefore recognizes two kinds of handlers. The abortion and the cross-reference handler are examples of an *outer* handler, which is simply placed in front of the arguments of the handled command as they are, without invoking the collector. The memoization handler, on the other hand, is an example of an *inner* handler, which receives the entire argument list from the collector (as a single argument) — or more precisely, even `memoize` sets up an outer handler, but this outer handler doesn't do much more than invoke the collector, which in turn invokes the inner handler.

Run conditions are another, if minor, lego piece of the advising framework. Using key `run conditions`, the user can set the conditions under which the (outer) handler is executed; for example, cross-reference commands are only advised when memoization is underway. And the same goes for `\label`, and for the `replicated` `\index`, and for `\(pdf)savepos`, upon which memoization must be aborted. The bottom-line is that run conditions repeat across handlers, so it makes sense to separate them out as an independent component of the framework, with the added bonus that the system can make sure that an invocation of a advised command which does not satisfy the run conditions will incur as litte overhead as possible.

Bailout handler   An automemoized command applies the next-options (set by `\mmznext`), but what happens when the run conditions are not satisfied? If nothing happened, the existing next-options might apply to the next instance of (auto)memoization, which would not be what the author intended. This is why Advice introduces the bailout handler, a piece of code executed before the original command when the run conditions are not met. Obviously, the bailout handler for memoization clears out the next-options (and does not process them).

The structure of advice   Together, the components mentioned above form a piece of *advice*:

---
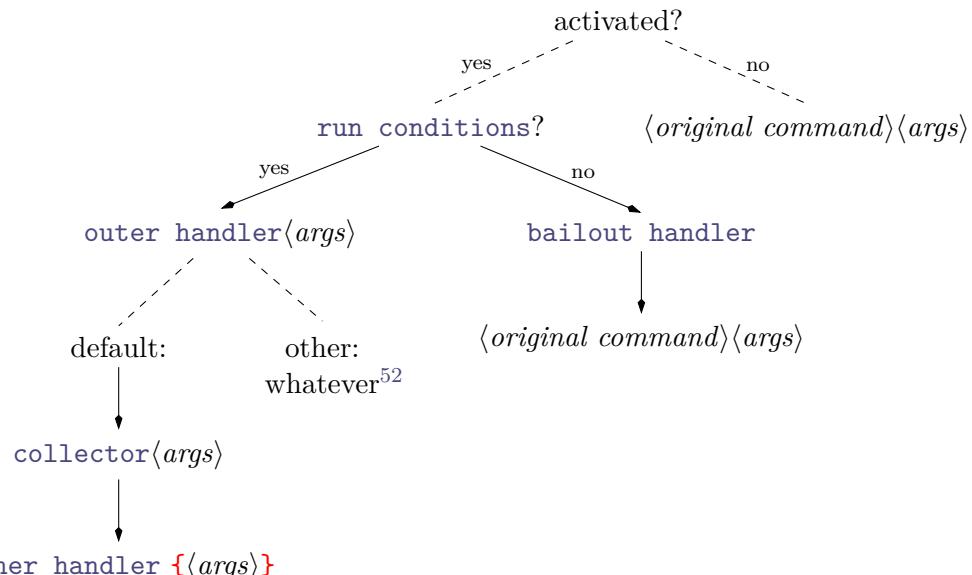
[52]The handler may do whatever as long as it consumes all and only the arguments of the original command.

Deferred activation   Memoize needs to be loaded early, but activation should take place late, so that it can surely override the submitted commands; a case in point, `hyperref` redefines `\ref` very late. To address the issue, the advising framework implements the deferred `activation` regime, under which (de)activation commands are not executed but collected in a special style, `activate deferred`. Memoize deploys the deferred activation regime throughout the preamble, and executes `activate deferred` at the latest possible `begindocument` hook; as a bonus, it also offers the author a way to avoid automatic activation completely by invoking key `manual`.

Install anywhere   Once all this machinery is developed, why not offer it to others as well?

Once I decided to offer Advice (at the time, still called Auto) as a standalone package (and I freely admit that the framework got much cleaner once I separated its code out of Memoize) it became immediately clear that if it is to serve as a generic framework, it should be possible for multiple packages to use it without interfering with each other. The package thus allows any number of installations into different namespaces, each namespace a `pgfkeys` keypath. The installation is a breeze: `\pgfkeys` {⟨*namespace*⟩`/.install advice`}.

4.5.1   Using package Advice

In this section, we will provide some examples of handler declarations, mainly based on how Memoize deploys the advising framework.

`/mmz/auto/memoize`

In section 2.3, the author was instructed to submit a command to automemoization by writing `auto=`⟨*command*⟩`{memoize,...}`. The auto-key `memoize` is a style (defined by Memoize rather than Advice) which sets the appropriate components of the automemoization advice. Residing in keypath `/mmz/auto`, it is effectively defined as follows:

```
\mmzset{
  auto/memoize/.style={
    run if memoization is possible,
    bailout handler=\mmz@auto@bailout,
    outer handler=\mmz@auto@outer,
    inner handler=\mmz@auto@memoize
  }
}
```

The heart of this advice is its inner handler, which actually triggers memoization by executing `\Memoize`. Remember that the first argument of `\Memoize` is the code which the md5sum is computed off of. This argument must therefore be identical to the author's invocation of the automemoized command or environment. Given what Advice offers, this is easy to construct: `\AdviceReplaced` holds the code replaced by the advice, and the ⟨*arguments*⟩ of the automemoized command are waiting for us in `#1`. The second argument of `\Memoize` will be similar, but as this is the code which will get compiled, we have to execute the original definition of the command, followed by the (unbraced!) ⟨*arguments*⟩ as `#1`; this is a job for `\AdviceOriginal`. (Note that *executing* `\AdviceReplaced` would run the auto-handler again, resulting in an infinite loop! Or at least a pile of errors.)

However, the overly simplistic approach shown below won't necessarily work. The issue is that the arguments of `\Memoize` contain `\AdviceReplaced` and `\AdviceOriginal` themselves, instead of their contents, i.e. (first) expansions.

```
\long\def\mmz@auto@memoize#1{%
  \Memoize{\AdviceReplaced#1}{\AdviceOriginal#1}%
}
```

Regarding the first argument, the problem is that the code md5sum will be computed off of the token

list `\AdviceReplaced`⟨*arguments*⟩ — exactly as you see it.[53] This implies that two commands sharing exactly the same ⟨*arguments*⟩ will receive the same (c-)memo. For example, if you automemoized first `\textit{foo}` and then `\textbf{foo}`, both would come out as a bold "foo" upon utilization.

The second argument illustrates a general issue about the lifespan of `\AdviceOriginal` and other `\Advice...` commands.[54] By executing `\Memoize`, we leave the advice and thereby cannot be sure that when expanded, `\AdviceOriginal` will mean what it means at the moment. In general, another piece of advice might be triggered until its expansion, or the group might be closed, etc. For example, the author may have issued `\mmznext{at begin memoization=\label{⟨key⟩}}`, and as the pre-memoization code is executed before the memoization driver, the `\label`, which is submitted to Advice so that any `\label`s inside the memoized code "just work", would execute another piece of advice, redefining `\AdviceOriginal` and friends. Effectively, you'd end up memoizing an invocation of the `\label` command.

The bottom line is that while the code following the above template *might* sometimes work, Advice offers no guarantees that it will, so I advise against using it. The actual definition of the memoization inner handler is shown below. In this definition, we expand `\AdviceReplaced` and `\AdviceOriginal` — exactly once! — into the respective arguments of `\Memoize`; of course, as the entire invocation of `\Memoize` is expanded, we have to guard against expanding the collected arguments (`#1`) and `\Memoize` itself.[55] The result of the expansion is shown under the code: the first and the second line assume we are automemoizing command `\foo` and environment `bar`, respectively. Note that `\AdviceOriginal` expands into an invocation of `\AdviceGetOriginal`, a command which may be safely used outside the advice; the first argument of this command is the auto-namespace (in our case, `/mmz`), and the second argument is the advised command. For a LaTeX environment, the advised command is actually `\begin`, and this is why the call of `\AdviceGetOriginal` is of course followed by the environment name.

The implementation of the `inner handler` for automemoization

```
\long\def\mmz@auto@memoize#1{%
  \expanded{%
    \noexpand\Memoize
      {\expandonce\AdviceReplaced\unexpanded{#1}}%
      {\expandonce\AdviceOriginal\unexpanded{#1}}%
    \ifmmz@ignorespaces\ignorespaces\fi
  }%
}
```

→ `\Memoize{\foo#1}{\AdviceGetOriginal{/mmz}{\foo}#1}`
→ `\Memoize{\begin{bar}#1}{\AdviceGetOriginal{/mmz}{\begin}{bar}#1}`

Let us now move backwards in time and look at the outer handler installed by `memoize`. It is very simple, but performs an important function of applying the auto- and the next-options (in this order), which also necessitates opening a group (closed by `\Memoize`). The final line invokes the argument collector, which then calls the inner handler; remember that the invocation of `\AdviceCollector` is the sole function of the default outer handler.

The implementation of the `outer handler` for automemoization

```
\def\mmz@auto@outer{%
  \begingroup
  \mmzAutoInit
  \AdviceCollector
}
```

Moving even further back in time, we arrive at the run conditions. The `memoize` style invokes `run if memoization is possible`, defined as `run conditions=\mmz@auto@rc@if@memoization@possible`,

---

[53]If you inspected a c-memo, you would find `\AdviceReplaced`⟨*arguments*⟩ in the `\mmzSource` section.

[54]The list of all commands available only within the handler can be found in the documentation of key `outer handler` in section 5.6.1.

[55]As the final touch, the handler also contains `\ignorespaces` after the invocation of `\Memoize`, if this was requested using `ignore spaces`. Note that this could not be done without pre-expanding the `\ifmmz@ignorespaces` conditional, as `\Memoize` closes the group in which the auto- and the next-options are applied.

with the installed macro as shown below. Indeed, memoization only makes when Memoize is `enabled` (which we test using `\ifmemoize`), but we're not already "inside `\Memoize`" (which we test using `\ifinmemoize`). The latter condition is true when we're either memoizing or regularly compiling some code submitted to memoization (see the diagram in section 4.1). Note that it is not necessary to invoke `\AdviceRunfalse` in branches where the run conditions are not satisfied.

The implementation of `run if memoization is possible`

```
\def\mmz@auto@rc@if@memoization@possible{%
  \ifmemoize
    \ifinmemoize
    \else
      \AdviceRuntrue
    \fi
  \fi
}
```

While it is clear that double memoization is a no-no, why should we avoid memoizing inside a regular compilation? Imagine that Memoize decides not to memoize a Forest tree, perhaps because `readonly` is in effect. Under the hood, Forest creates many `tikzpicture`s. Should all of them be (auto)memoized now? Certainly not.

Finally, what happens when the run conditions are not met? Not much, but something important nevertheless: by consuming the next-options, the `bailout handler` makes sure they will not erroneously apply to the next instance of (auto)memoization.

The implementation of the `bailout handler` for automemoization

```
\def\mmz@auto@bailout{%
  \mmznext{}%
}
```

The only component of the automemoization advice not determined by style `memoize` is the argument collector, which allows the user to submit a command with a weird argument structure to automemoization simply by setting key `collector` in addition to executing `memoize`. For example, Memoize submits `\tikz` to automemoization by loading `advice-tikz.code.tex`, which contains Advice's definition of the `\tikz` collector `\AdviceCollectTikZArguments`, and issuing the following declaration.

Declaring automemoization of command `\tikz`

```
    auto=\tikz{memoize, collector=\AdviceCollectTikZArguments},
```

`/mmz/auto/ref`

The cross-reference advice presents an example of an outer handler radically different from the default outer handler. This outer handler does not invoke the collector at all. As shown below, it grabs the argument of `\ref` (or whichever cross-referencing command) on its own — remember that the outer handler receives the arguments of the handled command "as they are," i.e. uncollected. It then asks `\mmzNoRef` to do the real job of getting the reference key into the context, and finally executes the original `\ref`.

A simplified[56] definition of `ref`

```
\mmzset{auto/ref.style={outer handler=\mmz@auto@ref, run if memoizing}}
\def\mmz@auto@ref#1{%
  \mmzNoRef{#1}%
  \AdviceOriginal{#1}%
}
```

The run conditions of this style are agonizingly simple: `run if memoizing` sets `run conditions` to a macro defined as `\ifmemoizing\AdviceRuntrue\fi`.

---

[56]The real outer handler allows for arbitrary optional arguments of the cross-referencing command, and shares code with `force ref`.

The advice for aborting memoization is very simple — it merely executes `\mmzAbort` — but also very sneaky. Here, the `run conditions` do the real work of aborting memoization, while the "real," i.e. outer handler, never even gets executed; note the absence of `\AdviceRuntrue`, which implies `\AdviceRunfalse`, which triggers the execution of the original command after the run conditions are "checked."

The definition of `abort`
```
\mmzset{
  auto/abort/.style={run conditions=\mmzAbort},
}
```

The point here is that executing `\mmzAbort` (itself a single-liner setting an internal conditional) is cheaper than testing for the real run conditions (`run if memoizing`) and aborting only if they are satisfied. Of course, the trick only works because (i) the advice doesn't need to inspect any arguments of advised command, and because (ii) setting the internal abortion conditional outside memoization does no damage.

Advice in chains

A command may be submitted to several instances of the advising framework, i.e. instances installed under different keypaths. In the example below, we submit `\foo` both to the instance of Advice installed in keypath `/one` and to the one installed in keypath `/two`. Under `/one`, the result of `\foo{...}` will be boxed (`\fboxWrap`); under `/two`, in will be parenthesized (`\parenWrap`). The order in which this happens depends on the order in which `\foo` was activated under different keypaths. If we first activate it under `/one` with the boxing effect and then under two with the parenthesizing effect, the box will appear within parenthesis; if we reverse the activation order, the parenthesis will appear inside the box.

chained-advice.tex
```
\usepackage{advice}

\def\foo#1{``#1''}
\def\fboxWrap#1{\fbox{\AdviceOriginal{#1}}}
\def\parenWrap#1{(\AdviceOriginal{#1})}

\pgfqkeys{/one}{.install advice, advice'=\foo{args=m, outer handler=\fboxWrap}}
\pgfqkeys{/two}{.install advice, advice'=\foo{args=m, outer handler=\parenWrap}}

\begin{document}
{\pgfkeys{/one/activate=\foo, /two/activate=\foo}\foo{bar}}
{\pgfkeys{/two/activate=\foo, /one/activate=\foo}\foo{bar}}
\end{document}
```
("bar") ("bar")

First of all, looking at the code above, you have probably noticed the absence of key `auto`. This is because by default, `.install advice` defines the *setup key* to be `advice` — Memoize overrides this default by installing the framework with `.install advice={setup key=auto, ...}`.

Next, `advice'` is a variant of `advice` which prevents automatic activation upon setup (and the same holds for `auto'` vs. `auto` in Memoize). We have used the bar variant above to make it clear that it is the order of activation, rather than declaration by `advice`/`auto`, which matters in determining which handler is applied first.

Finally, note that the deactivation order must be the reverse of the activation order. So if we activate `\foo` first in `/one` and then in `/two`, we should deactivate it in `/two` first and in `/one` next, otherwise Advice will complain.

A simple collector

Let us implement a collector for a command which accepts one (standard LaTeX) optional argument and one mandatory argument; in LaTeX3 terms, a command with argument specification `om`.

Using `\NewDocumentCommand`, such a collector is very easy to implement. We simply define a command with signature `om` and distinguish two possibilities regarding the presence of the optional argument, which we test using `\IfValueTF`. If the true branch, we pass a *braced* `[#1]{#2}` to the inner handler, which we invoke by `\AdviceInnerHandler`; in the false branch, we omit the optional argument, passing it an (additionally) braced `{#2}`.

```
om-collector-NewDocumentCommand.tex

    \NewDocumentCommand\omCollector{om}{% the collector
      \IfValueTF{#1}{\AdviceInnerHandler{[#1]{#2}}}{\AdviceInnerHandler{{#2}}}}
```

Defining a functionally equivalent collector using `\newcommand` would be a bit more involved, as LaTeX 2_ε does not offer a standardized way to test for the *presence* of the optional argument. Consider the following collector, whose optional argument has the same default value as the advised command. Is it functionally equivalent to the one above?

```
om-collector-newcommand.tex

    \newcommand\omCollector[2][green]{% the collector
      \AdviceInnerHandler{[#1]{#2}}}

    \newcommand\foo[2][green]{\textcolor{#1}{``#2''}\\}% the advised command
    \mmzset{auto=\foo{memoize, collector=\omCollector}}% the advice

    \begin{document}
    \foo[red]{red memoized text}
    \foo[green]{memoized text of the default color}
    \foo{memoized text of the default color}
    \end{document}
```

While there will be no visual difference, there is a difference under the hood. If you compile both documents, you will see that the first one creates three memos/externs, while the second one only creates two: `\foo{...}` does not have its own memo anymore, but creates and uses the same memo as `\foo[green]{...}`.

While the second version might sometimes be preferred, perhaps even in the context of memoization, the initial collector, which deploys command `\CollectArguments`, behaves like the `\NewDocumentCommand`-defined collector above,[57] as it attempts to perfectly replicate the command invocation. Furthermore, this behaviour makes it unnecessary for the author to provide the default values of optional arguments (and even allows them to replace `O{default}` in the argument specification by `o`).

We now turn to the package CollArgs, which implements the actual argument collection; we'll revisit the initial collector of Advice at the end of the following subsection.

4.5.2 Using package CollArgs

Automemoization is implemented on top of the framework offered by package Advice, and that package in turn couldn't really work as intended without package CollArgs. A regular user of Memoize shouldn't need to know anything about CollArgs, but a package writer wanting to support Memoize might have to.

The package provides two public commands, `\CollectArguments` and `\CollectArgumentsRaw`; we'll focus on the former first. `\CollectArguments` takes three arguments: optional ⟨*options*⟩ in the form of a `pgfkeys` keylist; a mandatory ⟨*argument specification*⟩ in a (slightly extended) LaTeX3/`xparse` format; and the ⟨*next-code*⟩:

---

[57]In fact, there is a slight difference after all. While the above-defined collector won't distinguish between the single-token mandatory argument given with or without braces, `\CollectArguments` will again faithfully replicate the original argument tokens.

$$\texttt{\textbackslash CollectArguments[}\langle \textit{options}\rangle\texttt{]\{}\langle \textit{argument specification}\rangle\texttt{\}\{}\langle \textit{next-code}\rangle\texttt{\}}\langle \textit{tokens}\rangle$$

Following the three formal arguments of `\CollectArguments` are some ⟨*tokens*⟩ — the rest of the document, really — and the job of `\CollectArguments` is to figure out the extent to which these ⟨*tokens*⟩ conform to the given ⟨*argument specification*⟩. In other words, `\CollectArguments` will consume as many of the ⟨*tokens*⟩ as a ⟨*command*⟩ defined by `\NewDocumentCommand`⟨*command*⟩`{`⟨*argument specification*⟩`}{...}` would. Once these ⟨*argument tokens*⟩ are collected, `\CollectArguments` executes the ⟨*next-code*⟩ with the ⟨*argument tokens*⟩ given as a single, braced argument (clearly, the ⟨*rest*⟩ of the ⟨*tokens*⟩, i.e. the non-consumed tokens, will follow):

$$\langle \textit{next-code}\rangle\texttt{\{}\langle \textit{argument tokens}\rangle\texttt{\}}\langle \textit{rest}\rangle$$

In the example below, we define macro `\PrintAndDo`, which takes two arguments, a command and the collected arguments of that command, prints out which command we're about to execute and with what arguments, and then executes that command with those arguments — `#1#2` at the end of the definition. Note that `#2` immediately following `#1` is not braced, so `\PrintAndDo\makebox{[5em][r]{text}}` executes `\makebox[5em][r]{text}`.

Executing `\PrintAndDo\makebox{[5em][r]{text}}` directly would thus yield the first line of the result below — and in fact, this is precisely what gets executed to yield that line, but in a roundabout fashion. Given the argument specification `oom` (two optional arguments followed by a mandatory argument), `\CollectArguments` figures out how many tokens following its formal arguments conform to this argument specification — below, these would be `[5em][r]{text}` following `\CollectArguments{oom}{\PrintAndDo\makebox}`— and puts them, braced, behind its ⟨*next-code*⟩ argument, `\PrintAndDo\makebox`, yielding `\PrintAndDo\makebox{[5em][r]{text}}`.

```
collargs-makebox.tex

  \newcommand\PrintAndDo[2]{%
    Executing \texttt{\string#1} with arguments ``\texttt{\detokenize{#2}}''
    yields ``#1#2''.\par
  }
  \CollectArguments{oom}{\PrintAndDo\makebox}[5em][r]{text}
  \CollectArguments{oom}{\PrintAndDo\makebox}[5em]{text}
  \CollectArguments{oom}{\PrintAndDo\makebox}{text}
```

Executing `\makebox` with arguments "`[5em][r]{text}`" yields "   text".
Executing `\makebox` with arguments "`[5em]{text}`" yields "   text   ".
Executing `\makebox` with arguments "`{text}`" yields "text".

Seeing the arguments of `\makebox` without the immediately preceding `\makebox` might seem strange, but remember that `\CollectArguments` is about the arguments of a command, not about the command's control sequence. It doesn't know or care which command the argument tokens "belong" to, as long as they conform to the given specification. In the example above, it is only in `#1#2` of `\PrintAndDo` that `\makebox` is "reunited" with its arguments, but note that the reunion is far from obligatory.

CollArgs supports all the argument types (and modifiers) that LaTeX3 does, complete with the uncommon/historical types only available upon loading package `xparse`. Below, we provide an example of the environment type `b`. Again, the code might seem strange, as it features an `\end{minipage}` without the matching `\begin{minipage}`, but the logic is similar as for commands: just as `\CollectArguments` occurs in front of the command arguments, without the command itself, so it occurs in front of the environment body, without the opening of that body. However, while `\CollectArguments` never needs to know the command name, we need to inform it of the environment name, so that it can find the end of the environment. This can be achieved as shown below, using key `environment` in the optional argument of the command, or by our extension to the LaTeX3/`xparse` argument specification, where the environment argument type `b` may be followed by a braced environment name. In the example below, we could therefore also invoke argument collection by `\CollectArguments{+b{minipage}}` (we have preceded `b` with a `+` to allow for an environment body containing paragraph tokens).[58]

---

[58]CollArgs automatically adapts to the format, i.e. it knows that environments are tagged by `\`⟨*name*⟩ and `\end`⟨*name*⟩ in plain TeX and by `\start`⟨*name*⟩ and `\stop`⟨*name*⟩ in ConTeXt.

```
collargs-minipage.tex
    \newcommand\PrintAndDoEnv[2]{%
      Executing environment ``#1'' with body ``\texttt{\detokenize{#2}}''
      yields \fbox{\begin{#1}#2\end{#1}}.\par
    }
    \CollectArguments[environment=minipage]{+b}{\PrintAndDoEnv{minipage}}[t]{6cm}
      This forms the body of a minipage environment, even if it is not preceded by
      \texttt{\string\begin\{minipage\}}.%
    \end{minipage}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Executing environment "minipage" with body "[t]{6cm} This forms the body of a minipage
    environment, even if it is not preceded by \texttt {\string \begin \{minipage\}}."
    yields ⸢This forms the body of a minipage⸣.
            ⸤environment, even if it is not pre-⸥
            ⸤ceded by \begin{minipage}.⸥
```

You might wonder why didn't we provide \CollectArguments in the previous example with argument specification omb — after all, the minipage environment takes an optional and a mandatory argument. While that would work, and produce the same result,[59] note that \CollectArguments is only interested in finding the scope of the arguments, and grabbing everything until \end{minipage} is the same as first grabbing the optional argument, maybe, then the mandatory argument, and finally the argument body.

\CollectArguments not only supports the verbatim argument type v, it can grab an argument of *any* type in the verbatim mode, triggered by option verbatim.[60] We illustrate this key below, where we also use option tags, which makes CollArgs automatically surround the grabbed environment body with the begin tag \begin{⟨environment name⟩} and the end tag \end{⟨environment name⟩}, and use \scantokens to execute the grabbed environment. Consult section 5.6.3 for the full reference on the verbatim mode and its limitations.

```
collargs-verbatim.tex
    \newcommand\PrintAndDoEnv[1]{%
      Executing \texttt{#1} yields this: \scantokens{#1}
    }
    \CollectArguments[environment=verbatim, verbatim, tags]{+b}{\PrintAndDoEnv}
      Here is some \LaTeX{} code.
    \end{verbatim}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Executing \begin{verbatim}  Here is some \LaTeX{} code. \end{verbatim} yields this:
      Here is some \LaTeX{} code.
```

Finally, CollArgs extends the LaTeX3/xparse specification by modifier &, which allows the user to specify options which apply only to the following argument, as opposed to the options given as the optional argument of \CollectArguments, which apply to all the arguments. A third way to invoke the environment body collection in the above example is thus \CollectArguments{& {environment=minipage}+b}.

Both the single-argument and the common options can be given not only as pgfkeys keys, but also in the raw, "programmer's interface" format. Every option key has a corresponding macro; for example, key environment is matched by macro \collargsEnvironment. The macros are listed alongside their corresponding keys in the reference section 5.6.3; here, we merely learn how to use them.

To use raw options for a single argument, double the ampersand in the argument specification. Therefore, the fourth way to specify the environment name is &&{\collargsEnvironment{minipage}}+b.

To set the raw options for all arguments, use \CollectArgumentsRaw, the second public command of the package. This command is exactly like \CollectArguments, excepts that it expects the options in

---

[59]Unless argument processing was in effect; see section 5.6.3 for details.

[60]We refer to the verbatim mode triggered by verbatim as the full verbatim mode, where all characters are of category "other". There is also the partial verbatim mode, triggered by verb, where braces retain their normal category codes.

the raw format and as a *mandatory* argument:

$$\texttt{\textbackslash CollectArgumentsRaw}\{\langle \textit{raw options}\rangle\}\{\langle \textit{argument specification}\rangle\}\{\langle \textit{next-code}\rangle\}\langle \textit{tokens}\rangle$$

This leads us to the fifth way to set the environment name (an overkill, I know): `\CollectArgumentsRaw` `{\collargsEnvironment{minipage}}{+b}{⟨next-code⟩}`. Furthermore, you can use a mixture of raw and key–value options: the raw option commands include `\collargsSet`, which applies the given option keylist. The idea here (incarnated by both Auto and Memoize) is that the package will provide CollArgs with the raw options, for speed, while the author can supplement them in the friendly keylist format — and this leads us to the sixth, and thankfully final way to set the environment name: `\CollectArgumentsRaw{\collargsSet{environment=minipage}}{+b}{⟨next-code⟩}`.

The initial collector

As the final example, let us study Advice's initial collector; this is a macro which is used as the collector when key `collector` is not given. This macro is not really `\CollectArguments`, as we sometimes state to simplify matters, but a macro which acts as the "bridge" between Advice and CollArgs, by compiling an invocation of `\CollectArgumentsRaw` from the given advice setup, and executing it.

The bridge macro is shown below in its full glory, but it is really less complicated than it might appear at first sight. In line 2, we use `\AdviceIfArgs` to see whether the argument structure of the handled command was given by the user. If it wasn't, we assume that the handled command was defined using `\NewDocumentCommand` or similar, and use `\GetDocumentCommandArgSpec` to retrieve it (line 3; note that `\AdviceName` holds the handled control sequence) and store it into `\AdviceArgs` (line 4), which also receives the argument specification when given by the user via key `args`.

The definition of the initial collector

```
 1 \def\advice@CollectArgumentsRaw{%
 2   \AdviceIfArgs{}{%
 3     \expandafter\GetDocumentCommandArgSpec\expandafter{\AdviceName}%
 4     \let\AdviceArgs\ArgumentSpecification
 5   }%
 6   \expanded{%
 7     \noexpand\CollectArgumentsRaw{%
 8       \noexpand\collargsCaller{\expandonce\AdviceName}%
 9       \expandonce\AdviceRawCollectorOptions
10       \ifdefempty\AdviceCollectorOptions{}{%
11         \noexpand\collargsSet{\expandonce\AdviceCollectorOptions}%
12       }%
13     }%
14     {\expandonce\AdviceArgs}%
15     {\expandonce\AdviceInnerHandler}%
16   }%
17 }
```

Lines 6–17 are somewhat of an expansion mess, because we have to construct the invocation of the CollArgs' collector from the advice setup stored in various macros. But once we think away all the (non-)expansion commands, we're left with `\CollectArgumentsRaw` plus the following three arguments:

1. The raw options (lines 8–11):

   (a) In line 8, the advised command's control sequence is designated as the `caller`. The effect is that if the given arguments don't conform to the specification, the error thrown seems to come from the advised command rather than some internal CollArgs macro. The author will be grateful for this little detail.

   (b) In line 9, we add any `raw collector options` set by Advice (plus the package deploying Advice, like Memoize); user-given options are of course possible, but not really expected here, because:

   (c) In lines 10–11, we add the user-given `collector options`, if there are any, embedded under `\collargsSet`.

2. The argument specification (line 14).

3. The `inner handler` (line 15).

5 Reference

In this section, the extra information about keys, offered at the right of a key in parenthesis, may contain the initial value of a key, and also a default value of a key. In this context, the terms "initial" and "default" have the meaning employed by the `pgfkeys` utility (§87 of the Ti*k*Z & PGF manual). The term "initial value" applies to the setting underlying a key (when there is no such setting, the key is marked as a "style"), and refers to the value of this setting that is set by the package. In other contexts, we would call this the default or the package-default value, but in the `pgfkeys` parlance, the term "default value" applies to a key taking an argument, and refers to the value that is passed to the key in the absence of that argument. (Honestly, I only keep to this convention in the reference section; elsewhere, I often say "default" or "package-default" and mean the initial value.)

Another convention I keep to in this section is the color-coding of the keys and commands. Green background indicates a basic key or command, which any user might want to know about. Red background indicates other, more or less advanced keys and commands.

5.1 Loading and initialization

LaTeX

Load Memoize by `\usepackage{memoize}` or `\usepackage[`⟨*options*⟩`]{memoize}`. The latter form functions almost identically to the former followed by `\mmzset{`⟨*options*⟩`}`, with two exceptions.

First, when used as package options, ⟨*options*⟩ may not contain the slash character (`/`), which is necessary to invoke `pgfkeys` handlers, because is it misinterpreted by LaTeX.[61] In the rare situations requiring key handling in the package options, use option `options`.

Second, key `extract` and other extraction-related keys such as `perl extraction options` should normally be used as a package option, or within `memoize.cfg`. Because Memoize extracts the externs while it is being loaded, executing these keys after the package is loaded will have a different effect; see the documentation of `extract` for details.

Memoize extracts the externs while it is being loaded (and not, say, at the beginning of the document) because extern extraction can only be performed before the output PDF is opened, and some packages cause it to be opened when they are loaded. This also implies that Memoize must be loaded before any such package. In particular, it must be loaded before PGF library `fadings` (see section 6.3) and before the `beamer` class (see section 2.7).

If you're familiar with the Ti*k*Z externalization library, you might wonder whether Memoize has an equivalent of the `\tikzexternalize` command. It doesn't. Memoize assumes that if you loaded it, you want to use it — but you can always disable it using key `disable`, or even by loading package `nomemoize` in its stead.

In LaTeX, initialization and finalization are completely automatic. Memoize defines several initialization and finalization styles — **begindocument/before**, **begindocument**, **begindocument/end** and **enddocument/afterlastpage** —and executes them at the cognominal LaTeX hooks.

plain TeX

Load Memoize by `\input memoize`. As package options cannot be provided in plain TeX, the author must trigger extraction from `\mmzset` using key `extract`; I recommend doing this immediately after loading the package. This key may be invoked with or without a value. In the latter case, Memoize will extract using the package default method `perl`, unless its has been overriden from `memoize.cfg`.

Furthermore, as plain TeX has no concept of a document body, the text must be manually enclosed in `\mmzset{`**begin document**`}` and `\mmzset{`**end document**`}`; this is where the initialization and finalization hooks described above will be executed. Note that `extract`, when used, must precede this enclosure.

---

[61]The historic LaTeX constraint prohibiting spaces in package options does not apply anymore.

Finally, plain TeX has another reason for preferring the early loading of the package. In this format, Memoize must redefine `\shipout`, at a time the meaning of this control sequence is still primitive. In particular, this means that Memoize must be loaded before `atbegshi`.

A minimal plain TeX example

```
\input memoize
\mmzset{extract}

% ...

\mmzset{at begin document}

Hello, \mmz{world}!

\mmzset{at end document}

\bye
```

A minimal ConTeXt example

```
\usemodule[memoize]

% ...

\mmzset{at begin document}
\starttext

Hello, \mmz{world}!

\stoptext
\mmzset{at end document}
```

ConTeXt

Load Memoize by `\usemodule[memoize]` or `\usemodule[memoize][⟨options⟩]`. Unlike in LaTeX, there are no restrictions on characters allowed within ⟨options⟩; the remarks on key `extract` and loading order are the same as for LaTeX.

In LaTeX, Memoize automatically executes its initialization and finalization code when at the beginning and the end of the document body. Due to my very limited experience with ConTeXt, and its project structure in particular, I don't know what the appropriate place for initialization and finalization would be in ConTeXt. I therefore provisionally leave it to the author to execute `\mmzset{begin document}` and `\mmzset{end document}` manually, and hope for advice on how to handle this properly.

Auxiliary packages

nomemoize (package)

Loading this package instead of Memoize completely disables memoization, but does not require removing any Memoize commands from the document (they all become no-ops). This package accepts any package options (and ignores them).

memoizable (package)

This package is a programmer's stub: if Memoize is loaded, it does nothing; otherwise, it provides the no-op variants Memoize commands. This package accepts no package options. Generic packages, e.g. TikZ libraries, can also load this package via `\input memoizable.code.tex`. See section 3.5.1 for details.

5.2 Configuration

\mmzset{⟨*options*⟩}

Update the Memoize configuration.

The ⟨*options*⟩ are a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs. They are processed using the `pgfkeys` utility of PGF/TikZ (see §87 of the TikZ & PGF manual), with the default path set to `/mmz`.

The changes are local to the current TeX group, except for keys where explicitly noted otherwise.

\mmznext{⟨*options*⟩}

This command accepts the same ⟨*options*⟩ as `\mmzset`, but interprets them as *next-options* — options which will be applied to the next, and only the next, *auto*memoized command or environment. (Remember that a command or environment is submitted to automemoization by `auto={`⟨*command or environment*⟩`}{memoize,...}`; see sections 2.3, 2.4 and 5.6 for details.)

Remarks for the author:

- Key `enable` has no effect inside `\mmznext`.
- If `\mmznext` is used more than once preceding an automemoized command, only the final invocation takes effect.
- The next-options also apply to commands and environments for which memoization is autodisabled via `auto={`⟨*command or environment*⟩`}{nomemoize,...}`.
- It is safe to set the next-options in front of a command submitted to automemoization which does not actually undergo memoization in this particular instance. In other words, the absence of memoization will not cause the next-options to "leak" to the next automemoized command.
- Only the linear (execution) order of `\mmznext` and the automemoized command matters. Key `\mmznext` will correctly apply to a single following automemoized command even if it occurs outside the group which that command is executed from; and it will apply to the following automemoized command even if it is called within a group closed before that command is executed.

Remarks for the programmer:

- The next-options are set globally.
- The effect of `\mmznext` is *not* cumulative. Consequently, `\mmznext{}` clears the next-options.
- The next-options are applied by executing `\mmzAutoInit` within the advice. Any piece of advice applying the next options should also clear them when the `run conditions` are not met. This is streamlined by style `apply options`, intended for use within `auto` declarations. Out of the box, this style is deployed by `memoize`, `nomemoize` and `noop`, but it may be used by any piece of advice. Note that the `outer handler` declared by this style opens a group (to apply the options in) but leaves it to the (undeclared) `inner handler` to close that group.
- Key `enable` has no effect inside `\mmznext` because when Memoize is disabled upon encountering an automemoized command, the advice bails out without ever applying the next-options. More generally, this applies to any advised command whose run conditions require `\ifmemoize` to be true. Key `disable`, on the other hand, takes effect, because `\ifmemoize` is checked within `\Memoize` as well.

memoize.cfg   (file)

The configuration file, loaded just before processing the package options. It will typically contain a `\mmzset` command, but it may contain any TeX code.

As for any other file loaded by TeX, the location of the file determines whether it applies system-wide, user-wide or directory-wide.

This file is also loaded by package `nomemoize`, on the off chance it defines some commands other than `/mmz` keys. (It is not loaded by package `memoizable`, though).

/mmz/options={⟨*options*⟩}

Execute ⟨*options*⟩ as if they were given an an argument to `\mmzset`.

This option primarily exists to allow any key–value pair accepted by `\mmzset` to be used as a package option. In particular, this applies key handlers (see §87 of the Ti*k*Z & PGF manual), because their invocation includes a slash (`/`). For example, one cannot directly use

`\usepackage[`perl extraction options`/.prefix={{`--quiet`}} ]{memoize}`

to add (prepend) option `--quiet` to the invocation of the perl extraction script `memoize-extract.pl`, but this will work:

`\usepackage[`options={perl extraction options`/.prefix={{`--quiet`}} ]{memoize}}`

\nommzkeys{⟨*key*⟩}

In package `memoize`, this command is a no-op; in packages `nomemoize` and `memoizable`, it defines key **/mmz/**⟨*key*⟩ as a no-op.

As explained in section 2.10, use this command to declare any **/mmz** keys you have used outside `\mmzset` when switching to package `nomemoize`.

5.3 Memoization

5.3.1 Manual memoization commands

\mmz[⟨*options*⟩]{⟨*code*⟩}
\begin{**memoize**}[⟨*options*⟩]
  ⟨*environment body*⟩
\end{**memoize**}

    Submit ⟨*code*⟩ or ⟨*environment body*⟩ to memoization.

    Prior to memoization, the configuration is locally updated by executing ⟨*options*⟩ given as the optional argument to this command, i.e. the given options take precedence to options previously set via \mmzset. Note that next-options, set by \mmznext, are not applied.

    The effect of the macro and of the environment version of this command is the same, except that the command version memoizes ⟨*code*⟩ exactly as-is, while the environment version trims away any spaces at the beginning and the end of the code.[62] The space-trimming feature of the environment ensures that you can write \begin{memoize} and \end{memoize} in separate lines (as shown above), but no extra space will creep into the extern.

    The space-trimming feature of the environment, which trims spaces at the beginning and at the end of the ⟨*environment body*⟩, should not be confused with the effect of `ignore spaces`, which ignores spaces following the environment end-tag (in LaTeX, \end{...}) — and which does not apply to manual memoization at all!

    The argument of \mmz *must* be enclosed in braces.

\nommz[⟨*options*⟩]{⟨*code*⟩}
\begin{**nomemoize**}[⟨*options*⟩]
  ⟨*environment body*⟩
\end{**nomemoize**}

    Disable Memoize for the span of the compilation of ⟨*code*⟩ or ⟨*environment body*⟩.

    This command consumes the ⟨*options*⟩ in the same way as \mmz/memoize described above. The macro and the environment version of the command exhibit the same space-trimming behaviour as their \mmz/memoize counterparts, and the argument of \nommz *must* be enclosed in braces.

5.3.2 Basic configuration

/mmz/**enable**
/mmz/**disable**

    Enable or disable the functionality of the package.

    What happens when Memoize is enabled depends on the memoization mode (`normal`, `readonly`, `recompile`) and many other factors. When the package is disabled, it neither creates new memos and externs, nor uses the existing ones; this applies to both manual and automatic memoization. The effect is close to not having Memoize loaded, or to loading NoMemoize, but it is not completely the same; for example, the record file (`.mmz`) *is* updated while Memoize is disabled, reflecting the fact that nothing was memoized (or utilized) in the disabled state.[63]

    If these keys are used in the preamble, their effect is delayed until the beginning of the document, to ensure that Memoize is never enabled in the preamble. Other than that, all these keys do is set the TeX conditional \ifmemoize, which you may use in your code to test whether Memoize is enabled. You may also use \memoizetrue and \memoizefalse, as long as you never enable the package in the preamble.

---

[62]What actually happens is that at the beginning of the environment body, all space tokens will be discarded. At the end of the body, no spaces are actually discarded; Memoize simply issues an \unskip. This should not matter to a regular user who simply writes down the environment.

[63]Cleaning the folder (§5.5.3) after disabling the package for the entire document is thus a bad idea.

Key `enable` cannot be applied to automemoized commands via `\mmznext`. It will take effect for manual memoization, though, and key `disable` will work for both, as expected.

`/mmz/`**`normal`** (the initial mode)
`/mmz/`**`readonly`**
`/mmz/`**`recompile`**

Select the memoization mode.

Each piece of code submitted to (either manual or automatic) memoization is associated to several files: one c-memo, one cc-memo, and some externs (zero or more, typically one). When Memoize encounters a piece of code submitted to memoization, it takes one of the following actions:

memoization The code is compiled in a special way which produces the associated memos and externs.

utilization The code is not compiled. Its effect is replicated by processing the cc-memo; typically, this includes the single extern into the document.

regular compilation The code is compiled as if Memoize was absent or disabled (the memos and externs are neither utilized nor produced).

The action taken depends on the memoization mode and on whether all the memos and externs associated with the code exist, as shown in the table on the right. Note that a single missing memo or extern implies the "no" column of the table, and that memoization will create *all* the associated memos and externs, even those which already exist.

	Do the memos and externs exist?	
mode	yes	no
normal	utilization	memoization
readonly	utilization	regular compilation
recompile	memoization	memoization

The memoization mode only has effect when Memoize is `enable`d. Mode selection is orthogonal to enabling/disabling the package; for example, if you switch to a new mode while the package is disabled, the new mode will be in effect once the package is enabled.

`/mmz/`**`verbatim`** (style)
`/mmz/`**`verb`** (style)
`/mmz/`**`no verbatim`** (style, the initial mode)

When `verbatim` or `verb` is in effect, the code submitted to memoization is read verbatim; `no verbatim` reverts to the normal, non-verbatim collection of the code. This applies to both manual and automatic memoization.

The long version, `verbatim`, switches to the full verbatim mode, where all characters are assigned category code 12 (other). With the short version, `verb`, the braces, { and }, retain category codes 1 and 2, which can be useful for verbatim collection of optional arguments. For details, see the documentation of CollArgs' `verbatim` in section 5.6.3.

Under the hood, these keys have two effects. First, they are passed on to the argument collector (typically, `\CollectArguments` of the auxiliary package CollArgs; for details, see section 5.6), instructing it to collect the code in the specified fashion, as described above. Second, if the collected verbatim code is eventually compiled (either regularly, or memoized), Memoize first rescans it using `\scantokens`.

/mmz/**padding left**=⟨*dimension*⟩   (no default, initially `1 in`)
/mmz/**padding right**=⟨*dimension*⟩   (no default, initially `1 in`)
/mmz/**padding top**=⟨*dimension*⟩   (no default, initially `1 in`)
/mmz/**padding bottom**=⟨*dimension*⟩   (no default, initially `1 in`)

Set the left/right/top/bottom padding of the extern in the extern PDF.

Without padding, the (PDF) page holding the extern would tightly fit the bounding box of the extern. These keys enlarge the extern page by the given amounts, so that any parts of the extern lying outside the bounding box will be correctly included when using the extern. See section 2.8 for details.

⟨*dimension*⟩ is evaluated with ε-TEX's `\dimexpr`, and may contain control sequences **\width**, **\height** and **\depth**, which will refer to the dimensions of the extern. \width and friends behave like dimension registers, so it is ok to write e.g. `padding right`=0.5\width.

The default padding is what pdfTEX puts around the page anyway, 1 inch, but we use `1 in` rather than `1 true in`, which is the true default value of PDF registers `horigin` and `vorigin`, as we want the padding to adjust with \magnification.

/mmz/**padding**=⟨*dimension*⟩   (style, no default)

Set all the above `padding` keys to the given value.

/mmz/**context**=⟨*tokens*⟩   (cumulative, no default, initially set by `padding to context`)
/mmz/**clear context**

These keys append the given ⟨*tokens*⟩ to the *context expression*, or clear this expression. Memoized code gets recompiled whenever the expansion of the context expression changes.

The ⟨*tokens*⟩ must be fully expandable (modulo protection); in LATEX, they will be expanded by \protected@edef when calculating the md5sum of the context.

The context expression is evaluated at the end of the memoization, and at utilization attempts after inputting the c-memo (note that the c-memo contains any additions to the context expression accumulated during memoization). At evaluation, the given context expression is fully expanded, yielding the *context value*, whose md5 sum forms the ⟨*context md5 sum*⟩ part of the filename of the cc-memo and the extern.

These keys may be used both prior to the memoization process, or during memoization. In the former case, their effect is local to the current group; in the latter case, the effect is global, so that the changes surely survive until the end of memoization, when the c-memo, where the context expression is stored, is written into a file.

Under the hood, these keys manipulate token registers **\mmzContext** and **\mmzContextExtra**, changing the contents of the former while not memoizing, and the contents of the latter during memoization. These token registers may also be manipulated directly by the user, as long as one keeps to the convention of adjusting \mmzContext locally and only while not memoizing, and adjusting \mmzContextExtra globally and only during memoization.

/mmz/**meaning to context**={⟨*comma-separated list of commands and environment names*⟩}
/mmz/**csname meaning to context**={⟨*control sequence name*⟩}
/mmz/**key meaning to context**={⟨*full path to a pgfkeys command key*⟩}
/mmz/**key value to context**={⟨*full path to a pgfkeys value key*⟩}

These keys append the definition of the given construct to the context.

Essentially, the "meaning" keys append \meaning⟨*control sequence*⟩ to the context, for the appropriate ⟨*control sequence*⟩. For example, `meaning to context` appends \meaning\foo when given \foo as in item in the list, and it appends the internal environment macros appropriate to the format when given an environment name. Similarly, `key meaning to context` resolves ⟨*control sequence*⟩ to the internal macro holding the key command.

Key `key value to context` should be used on keys which store values, e.g. keys initialized by `pgfkeys` handler `.initial`; see §87.3.4 and §87.4.5 of the TikZ & PGF manual.

All the keys prefix the meaning/value by the name of the command/environment/key, in order to prevent ambiguous contexts, see section 3.3 for details. Furthermore, they all operate through `\csname...\endcsname` construct, allowing one to safely add internal commands to the context.

`/handlers/`**`.meaning to context`** (handler)
`/handlers/`**`.value to context`** (handler)

These are the handler variants of `key meaning to context` and `key value to context`.

`/mmz/`**`padding to context`** (style)

This key appends the values of the `padding` keys to the context, causing the memoized code to be recompiled whenever the padding values are changed. This key is used to initialize `context`, so the author normally shouldn't have to use this key.

`\mmzNoRef``{⟨reference key⟩}` (LATEX only)
`\mmzForceNoRef``{⟨reference key⟩}` (LATEX only)

These macros append the current value of the ⟨reference key⟩ to the context,[64] causing the memoized code to be recompiled when the reference changes.

If the reference key is undefined, `\mmzNoRef` aborts memoization, while `\mmzForceNoRef` uses `\relax` as the reference string.

These commands are deployed in the implementation of `/mmz/auto/ref`, `force ref` and friends. If the cross-referencing commands you are using are advised by these keys, you most likely have no need of these macros.

`/mmz/`**`capture`**`=`**`hbox`**`|`**`vbox`** (no default, initially `hbox`)

Select the capture mode. This setting only applies to the default memoization `driver`.

By default, it is assumed that the memoized code should be executed in the horizontal mode, so the default memoization driver captures the output of the memoized code in a `\hbox` — and also issues a `\quitvmode` (both in the document and in the cc-memo), just in case the memoized code occurs at the start of the paragraph.

Use `capture`=`vbox` to execute the memoized code in the vertical mode: Memoize will capture the output of the memoized code in a `\vbox`, and avoid issuing `\quitvmode`. For example, this capture mode is necessary to memoize a `verbatim` environment.

---

[64]More precisely, it is ⟨reference key⟩=`{`⟨current value⟩`}` which is appended.

5.3.3 Inside the memoization process

\mmzAbort

This command aborts the ongoing memoization.

The memoization will proceed as usual (i.e. the extern boxes and the cc-memo code will be produced), but at the end of this process, no memos will be produced, no externs shipped out to extern pages, and no record files updated.

\mmzUnmemoizable

This command aborts the ongoing memoization, and marks the submitted code as unmemoizable.

The ongoing memoization produces a c-memo setting conditional **\ifmmzUnmemoizable** to true. Upon utilizing this c-memo, the system switches to regular compilation.

For example, if you are automemoizing `tcolorbox`es of package `tcolorbox`, you will want to refrain from memoizing boxes marked as `breakable` or `float`ing. Simply aborting the memoization cannot do the trick here, as memoization compiles the submitted code in a TeX box. Marking a `breakable` or `float`ing `tcolorbox` as unmemoizable (either manually using this macro, or automatically using `auto key`) makes sure that after the first compilation when memoization is attempted, the box will be compiled regularly, and will have the intended ability to break across pages, or float.

\Memoize{⟨*key*⟩}{⟨*code*⟩}

Depending on various factors, this command either memoizes ⟨*code*⟩ under key ⟨*key*⟩, utilizes the results of a previous memoization, or performs a regular compilation of ⟨*code*⟩.

The outcome of executing this command — memoization, utilization or regular compilation — depends upon the Memoize's state (**\ifmemoize**, **\ifmemoizing**) and mode (`normal`, `readonly`, `recompile`), and the existence of the relevant memos and externs. The decision process is depicted in section 4.1.

This command expects to be executed in a dedicated group, which it will close itself.

Invoking memoization through **\Memoize** might be useful for packages which want to save the results of intensive computations, regardless of whether the author loads (and enables) memoization or not. However, this usage is not yet officially allowed, because there is currently no way to load the core memoization routines without loading the entire package, thereby forcing the author to use Memoize.

/mmz/**driver**={⟨*code*⟩}   (no default, initially **\mmzSingleExternDriver**)

This key sets ⟨*code*⟩ as the memoization driver.

Given some code submitted to memoization, the memoization driver should produce the memos and externs which will replicate the effect of that code (while retaining its regular effect). For details, see section 4.4.

Typically, the ⟨*code*⟩ argument of this key will consist of a single control sequence (the driver control sequence), but any amount of tokens is allowed. Memoize executes the driver followed by the code which it is supposed to memoize, in braces, and only cares that the driver consumes that code.

/mmz/**at begin memoization**=⟨*code*⟩	(cumulative, initially empty)
/mmz/**at end memoization**=⟨*code*⟩	(cumulative, initially empty)
/mmz/**after memoization**=⟨*code*⟩	(cumulative, initially empty)

Use these keys to set up memoization hooks.

These keys may be used both prior to the memoization process, or during memoization. In the former case, their effect is local to the current group; in the latter case, the code given to `at begin memoization` is executed immediately, while the assignment performed by the other two keys is global, so that the changes surely survive until the end of memoization.

The code given to hook `at begin memoization` is kept in macro **\mmzAtBeginMemoization**, while the content of the other two hooks resides in two macros per hook: **\mmzAtEndMemoization** and **\mmzAtEndMemoizationExtra**, and **\mmzAfterMemoization** and **\mmzAfterMemoizationExtra**. All these macros may be manipulated directly by the user,[65] as long as one keeps to the convention of adjusting the macros without "**Extra**" locally and only while not memoizing, and adjusting the macros with "**Extra**" globally and only during memoization. The "**Extra**" macros require global assignments as they might be manipulated by code residing within a TeX group of any depth.

These complications explained, let us take a look at how memoization proceeds to learn when the hooks are used:

1. Initialize various conditionals, macros and token registers. (Here is where the "**Extra**" hooks are cleared.) Remember that at this point, we're inside a group opened by \Memoize.

2. Execute `at begin memoization` hook, i.e. the contents of macro \mmzAtBeginMemoization.

3. Execute the memoization `driver`.

4. Execute `at end memoization` hook, i.e. the contents of macros \mmzAtEndMemoization and \mmzAtEndMemoizationExtra (in this order).

5. Write out the memos and ship out the externs to extern pages (unless memoization was aborted).

6. Close the memoization group.

7. Execute `after memoization` hook, i.e. the contents of macros \mmzAfterMemoization and \mmzAfterMemoizationExtra (in this order).

\mmzCMemo (token register, global, empty at the start of memoization)

This token register mediates the construction of the c-memo. During memoization (and only during memoization), arbitrary code may be added to this register; at the end of memoization, Memoize writes out its contents to the free-form part of the c-memo.

All assignments to this register should be global. Use \gtoksapp and \xtoksapp to easily append tokens to the register.

\mmzCCMemo (token register, global, empty at the start of memoization)

This token register mediates the construction of the cc-memo. During memoization, the memoization driver should append cc-memo code to \mmzCCMemo; at the end of memoization, Memoize writes out its contents to the cc-memo (preceded by the list of produced externs).

All assignments to this register should be global. Local assignments would not work, because the memoized code may contain commands, like \label and \ref, which contribute content to cc-memo as well, but these commands may appear within a TeX group of any depth.

Use \gtoksapp and \xtoksapp to easily append tokens to the register.

---

[65]Use \appto, \eappto, \gappto and \xappto of package `etoolbox` (loaded by Memoize) to easily append code to these macros.

\toksapp⟨*token register*⟩{⟨*tokens*⟩}
\gtoksapp⟨*token register*⟩{⟨*tokens*⟩}
\etoksapp⟨*token register*⟩{⟨*tokens*⟩}
\xtoksapp⟨*token register*⟩{⟨*tokens*⟩}

These commands append the given ⟨*tokens*⟩ to the ⟨*token register*⟩. \etoksapp and \xtoksapp expand the ⟨*tokens*⟩ before appending them; \gtoksapp and \xtoksapp perform a global assignment.

These commands are actually provided by CollArgs, and they are defined only if they don't already exist; in particular, note that LuaTeX provides them as primitives.

Unlike the LuaTeX primitive variant, these commands require the ⟨*token register*⟩ to be given by a (\toksdeffed) control sequence; it cannot be given as \toks⟨*number*⟩.

\ifmemoize
\memoizetrue
\memoizefalse

Use the TeX-style conditional \ifmemoize to test whether Memoize is currently enabled. Within the document body, the conditional may be set using \memoizetrue and \memoizefalse, which are then functionally equivalent to enable and disable. Do not set the conditional in the preamble of the document (unless you really know what you are doing).

\ifmemoizing   (readonly)

Use this TeX-style conditional to test whether Memoize is currently memoizing. It may be only inspected; you should *never* set this conditional yourself.

\ifinmemoize   (readonly)

Use this TeX-style conditional to test whether Memoize is currently active, in the sense of either memoizing or regularly compiling some code — so inside a call to \Memoize. The conditional may be only inspected; you should *never* set it yourself.

\mmzSingleExternDriver{⟨*code*⟩}

This is the default memoization driver, producing exactly one extern containing whatever is typeset by the submitted ⟨*code*⟩.

The ⟨*code*⟩ is compiled either within a horizontal or vertical box, depending on the value of key capture. In the case of a horizontal capture, the driver makes sure that the horizontal mode is entered prior to both typesetting the resulting box in the document, or utilizing the extern.

For the implementational details, see section 4.4.1.

\mmzExternalizeBox{⟨*box*⟩}{⟨*token register*⟩}

This macro is indended to be called by memoization drivers to produce an extern page. The given ⟨*box*⟩ is dumped into the document as a separate extern page, while the ⟨*token register*⟩ receives the cc-memo extern inclusion code.

The ⟨*box*⟩ may be given either as a control sequence (declared via `\newbox`), or as box number. The resulting extern page will contain a *copy* of the given box, padded by the `padding` values in effect at the time of invocation of `\mmzExternalizeBox`.

An implementation detail is that `\mmzExternalizeBox` does not ship out the extern page immediately. This action is delayed until the end of the memoization process; more precisely, it is carried out (in tandem with writing out the c-memo and the cc-memo) between execution of hooks `at end memoization` and `after memoization`. This delay guarantees that no extern pages are produced in the event of aborting memoization, even if the abortion is triggered after executing `\mmzExternalizeBox`.

The ⟨*token register*⟩ may be given either as a control sequence (declared via `\newtoks`) or as control sequence `\toks` followed by the register number. The register will receive the code, which, when executed from the cc-memo, includes the extern file into the main document. This code consists of a single invocation of `\mmzIncludeExtern`. It is the responsibility of the driver to include the code received by ⟨*token register*⟩ in the register `\mmzMemo`, whose contents are, unless memoization is aborted, written into the cc-memo. (See `\ifmmzkeepexterns` and `after memoization` to learn about another way to use the code received by ⟨*token register*⟩.)

The invocation of `\mmzIncludeExtern` in the produced extern-inclusion code is adapted to the type of the box (horizontal or vertical), which is detected automatically — the memoization driver does not need to inform `\mmzExternalizeBox` about this type explicitly.[66]

\ifmmzkeepexterns (initially `\iffalse`)
\global\mmzkeepexternstrue
\global\mmzkeepexternsfalse

Setting this conditional to true makes Memoize keep the extern boxes in the global temporary storage even after shipping them out as extern pages. (The temporary storage is emptied at the start of the next memoization.)

The extern inclusion code received by the `\mmzCCMemo` when executing `\mmzExternalizeBox` is primarily meant to be executed by inputting the cc-memo file; i.e. when the cc-memo is input, `\mmzIncludeExtern` is defined to include the extern file into the document. However, it sometimes makes sense to execute the cc-memo contents immediately after memoization; for example, if memoization produces several externs, intricately integrated into the surrouding environment, it might be cumbersome to replicate their typesetting both in the memoizing compilation and in the cc-memo code — easier to build up the cc-memo code and execute it right after memoization. This is why Memoize, just before executing the contents of `after memoization` hook, redefines `\mmzIncludeExtern` to include externs from the temporary storage rather than from (at that point still non-existing) extern files. However, as this mechanism requires Memoize to keep the externs around even after memoization, it is not enabled by default: it must be enabled by (globally) setting conditional `\ifmmzkeepexterns` to true.

/mmz/auto/integrated driver={⟨*name*⟩} (style)

Use this key to easily setup a memoization driver which is integrated into the command itself.

> This is an auto-key residing in keypath `/mmz/auto`.

---

[66]This does not negate the need for key `capture`, which applies to the default — and therefore generic — memoization driver. This driver cannot know whether the memoized code would prefer to be compiled in a horizontal or vertical box. It is precisely key `capture` which gives the user an opportunity to inform Memoize about this preference. Only once the memoized code is compiled into a box of the appropriate type, it is trivial to detect the type of that box.

An integrated driver must have a way of telling whether it is memoizing or regularly compiling the code. This key declares a driver-specific conditional which may be inspected, using `\IfMemoizing`, to determine this. The conditional is set to true by the formal `driver` of the command (set up by the invocation of this key), executed at the start of memoization; it should never be set elsewhere. See section 4.4.3 for details and an example.

`\IfMemoizing``[⟨offset⟩]{⟨name⟩}{⟨true code⟩}{⟨false code⟩}`

This LaTeX-style conditional is meant to be used by the `integrated driver` with the given ⟨*name*⟩. It tests whether this particular driver is currently memoizing some code.

Potentially recursive commands are supported via the optional argument ⟨*offset*⟩. If given, the conditional will only execute the ⟨*true code*⟩ when the current TeX group level matches the TeX group level at the time of the invocation of the formal driver (held in `\memoizinggrouplevel`), plus the ⟨*offset*⟩. In effect, the inner invocation of the integrated driver will perform a regular compilation. For details, see section 4.4.4.

`\memoizinggrouplevel` (readonly)

During memoization, this macro holds the TeX group level in effect at the start of the memoization.

`\mmzRegularPages` (readonly counter)

This counter holds the number of pages shipped out (so far) by the format's regular shipout routine. *Do not change its value!*

In LaTeX, this counter is synonymous with `\ReadonlyShipoutCounter`, and in ConTeXt, it is synonymous with `\realpageno`. Memoize does not touch its value.

In plain TeX, Memoize hijacks the `\shipout` control sequence to count (and only to count) regular shipouts. In order for its value to be realistic, Memoize should be loaded before other packages which hack `\shipout` — in particular, before `atbegshi`.

`\mmzExternPages` (readonly counter)

This counter holds the number of extern pages Memoize has shipped out (so far). *Do not change its value!*

A third-party tool may inspect this counter to have a realistic count of shipped-out pages.

`\mmzExtraPages` (counter)

This counter holds the number of pages shipped out (so far) in a way not tracked by either `\mmzRegularPages` or `\mmzExternPages`. It *should* be advanced by any code which performs such shipouts, or Memoize won't work correctly.

5.3.4 Tracing

/mmz/trace=true|false   (default `true`, initially `false`)

When tracing is on, Memoize shows information about its decision processes on the terminal. You can learn whether the memoized code is being memoized, utilized or regularly compiled; find out the md5sum of the code and which input line it comes from; etc.

This key has the syntax of a conditional, but there is no underlying TeX conditional. The low-level interface for switching the tracing on and off consists of macros **\mmzTracingOn** and **\mmzTracingOff**.

To learn about tracing the "auto" part of the automemoization process, also **\AdviceTracingOn**.

/mmz/include source in cmemo=true|false   (default `true`, initially `true`)

As a courtesy towards a curious user and a debugging aid, Memoize can include a copy of the memo source in the c-memo. This feature is switched on by default, but as the package itself never uses that information, it can be safely switched off at any time.

/mmz/include context in ccmemo=true|false   (default `true`, initially `false`)

When this key is in effect, the expanded context expression is appended to the cc-memo, behind the **\mmzThisContext** marker.

Memoize never uses the context information from the cc-memo; this information is only for tracing purposes.

/mmz/direct ccmemo input=true|false   (default `true`, initially `false`)

When this key is set to false, a cc-memo is processed indirectly: it is first read into a token register, and it is the contents of this register which are executed. When the key is set to true, the cc-memo is simply \inputted.

The indirect execution is implemented to facilitate inverse search. Under the direct cc-memo input, inverse search pointed at an included extern will visit the cc-memo, which is not practical; under the indirect regime, the inverse search will work as expected, and this is why the indirect cc-memo input is the default.

The overhead produced by the default indirect input method seems negligible, but there are other factors which might make the user switch to the direct input. For one, a cc-memo changing some category codes will require direct input (no such cc-memos are ever produced out of the box). Less crucially, sometimes one would like to use the inverse search to figure whether a part of the document was produced by regular compilation or utilization, and which memos/externs were utilized if the latter. Figuring this out under the indirect input regime is harder: (i) reading the tracing information shown by `trace` is the surest way to learn what's going on, although (ii) visual inspection of the externs and (iii) grepping through the `.memo.dir` folder for particular code often help, as well.

Both input methods use the same cc-memos; there is no need to `recompile` the memos when switching the cc-memo input method. Note that the default indirect input method crucially relies on cc-memos ending with **\mmzEndMemo**; this macro should not appear in the cc-memo itself.

5.3.5 Internal memo commands

The end-user should never have to use these commands. They are not formally marked as internal by a `@` in their name only because doing so would complicate `\input`ting the memos due to the category code changes it would require.

`\mmzMemo`

This macro marks the beginning of a c-memo and a cc-memo core. Without it, utilization of a memo will not work.

`\mmzSource`

This macro marks the beginning of the memoized source in the c-memo. That source is not used by Memoize in any way. It's inclusion into the c-memo may be switched off by `include source in cmemo`=`false`.

`\mmzResource`{⟨*filename*⟩}

This is an internal command, which only occurs in a cc-memo. It checks whether file ⟨*filename*⟩ exists and is non-empty, and triggers recompilation of the memoized code if the check fails.

`\mmzIncludeExtern`{⟨*seq*⟩}{\hbox|\vbox}{⟨*expected width*⟩}{⟨*expected height*⟩}{⟨*expected depth*⟩} {⟨*padding left*⟩}{⟨*padding bottom*⟩}{⟨*padding right*⟩}{⟨*padding top*⟩}

This is an internal command, which only occurs in a cc-memo. It includes the extern identified by the sequential number ⟨*seq*⟩ into the document as a box of the specified type (horizontal or vertical). The extern is trimmed by the given padding values. After trimming, the command checks whether the size of the resulting box matches the given expectations; if it doesn't, a warning is yielded.

Before this command is executed, the externs should be listed by a sequence of `\mmzResource` commands; ⟨*seq*⟩ refers to the sequential number of an extern in this sequence.

This command may also be executed by executing the entire contents of `\mmzCCMemo` after memoization.

`\mmzLabel`{⟨*label key*⟩}{⟨*label value*⟩}

This is an internal command written into the cc-memo by the auto-handler of `\label`. It temporarily stores ⟨*label value*⟩ into `\@currentlabel` and then executes `\label`{⟨*label name*⟩}.

`\mmzEndMemo`

This macro marks the end of a cc-memo. It is used to grab the cc-memo core (everything between `\mmzMemo` and `\mmzEndMemo`) under the indirect cc-memo input regime, i.e. when `direct ccmemo input` is not in effect.

5.4 Location of memos and externs

/mmz/memo dir=⟨*name*⟩   (style, default `\jobname`)

A convenient way to store memos and externs in a dedicated directory (see sections 2.5 and 2.6 for the tutorial).

Without an argument, this key places these files in subdirectory ⟨*document name*⟩`.memo.dir` of the current directory. With an argument, these files are stored in subdirectory ⟨*name*⟩`.memo.dir`. In both cases, memo and extern filenames contain no prefix, as ⟨*name*⟩ already occurs in the directory name.

The definition of this style is very simple: `memo dir/.style={prefix={#1.memo.dir/}}` (note the slash). Feel free to redefine it, or to define another style to suit your needs.

/mmz/no memo dir   (style)

A convenient way to undo the effect of `memo dir` and revert to the initial `prefix` setting which locates memos and externs in the current directory, with filenames prefixed by ⟨*document name*⟩`.` (mind the dot).

/mmz/prefix=⟨*prefix*⟩   (no default, initially `\jobname.`)

This key determines the location of memo and extern files, and the initial part of their filenames.

If ⟨*prefix*⟩ contains no `/`, memos and externs are stored in the current directory, alongside the output PDF, and their filenames begin with ⟨*prefix*⟩ (and continue with the identifier, see below). For example, this is what happens with the initial value of this key, where memo and extern filenames are prefixed by ⟨*jobname*⟩ (i.e. the name of the document) plus `.` (a dot). Such a situation is shown in the `dirty-house` example in section 2.5.

If ⟨*prefix*⟩ contains slashes, everything up to the final slash determines the directory which memos and externs will be stored into, and the part after the final slash determines the starting part of their filenames; a slash (`/`) as a directory separator should be used even on Windows, where the system directory separator is a backslash (`\`). For example, `memo dir` sets `prefix` to `\jobname.memo.dir/`. As shown in the `clean-house` example in section 2.5, this results in memos and externs stored in directory ⟨*jobname*⟩`.memo.dir`, with their filenames consisting solely of the identifier.

In detail, the paths to memos and externs are constructed as shown below, where ⟨*code md5 sum*⟩ and ⟨*context md5 sum*⟩ identify the memoized code and the `context` in effect at its memoization, and N is the sequential number of the extern with respect to that code and context (N is usually 0, as memoization normally produces a single extern).[67]

c-memo: ⟨*prefix*⟩⟨*code md5 sum*⟩`.memo`
cc-memo: ⟨*prefix*⟩⟨*code md5 sum*⟩`-`⟨*context md5 sum*⟩`.memo`
extern: ⟨*prefix*⟩⟨*code md5 sum*⟩`-`⟨*context md5 sum*⟩`-`N`.pdf`
(where "-N" is only present when $N \neq 0$, i.e. for non-first externs)

---

[67]Normally, these paths are relative to the current directory (i.e. the directory where TeX is executed from; usually, this will be the directory where the compiled `.tex` file resides). However, when TeX is invoked with option `-output-directory`, these paths are relative to the specified output directory. Furthermore, when a memo or extern cannot be written into the current/output directory, it will be stored into the temporary directory `TEXMFOUTPUT`, if specified.

> **The final slash matters**
>
> To reiterate, the presence vs. absence of a slash (`/`) determines whether memos and externs are stored in a dedicated directory or not. For example, if you want to store memos and externs in directory `memos`, you should set `prefix=memos/`, with the final slash. Without the final slash, these files would end up in the current directory.

In principle, the directory specified by this key must already exist. However, Memoize does it best to create it for you, and should succeed at least when extraction method is set to `perl` or `python`. See also `mkdir` and `mkdir command`.

When invoked from the document body, each invocation of this key records the new prefix by invoking `prefix` (this typically results in a `\mmzPrefix` entry in the `.mmz` file) and attempts to create memo directory if `mkdir` is in effect. When `prefix` is executed in the document preamble, these actions are only carried out at the beginning of the document, for the final value of the key.

/mmz/**mkdir**=true|false   (default `true`, initially `true`)

When this key is set to true and the value of `mkdir command` is non-empty, Memoize will attempt to create the memo directory set by `prefix` if it does not yet exist.

The directory is created using the system command specified by `mkdir command`. The directory creation takes place at the beginning of the document and at every subsequent invocation of key `prefix`.

/mmz/**mkdir command**=⟨*system command*⟩   (no default, initially empty)

This key sets the command used to create the memo directory specified by `prefix` when `mkdir` is in effect.

Memoize attempts to create the directory by executing ⟨*system command*⟩ followed by (a space and) ⟨*directory*⟩, where ⟨*directory*⟩ is the directory part of the ⟨*prefix*⟩ set by `prefix`. The directory creation takes place at the beginning of the document and at every subsequent invocation of key `prefix`.

This key could be set to `mkdir` (on Linux, `mkdir -p` would be an even better choice), however, this is not advisable as `mkdir` does not respect TeX's output restrictions, set by `openout_any` in TeXLive and `[Core]AllowUnsafeInputFiles` in MiKTeX. Further note that as the value of this key is a system command, an appropriate shell escape mode must be in effect to execute it successfully; again, not something to be taken lightly.

The extractions scripts shipped with Memoize accept option `--mkdir`, which makes them behave as a safe variant of `mkdir`, i.e. a `mkdir` which respects TeX's output restrictions. Therefore, extraction methods `perl` and `python` set `mkdir command` to `memoize-extract.pl --mkdir` and `memoize-extract.py --mkdir`, respectively. In effect, as the initial extraction method is `perl`, the memo directory (if set via `prefix` or `memo dir`) should be created under the initial settings without any user intervention.

5.5 Extern extraction

/mmz/extract={⟨*extraction method*⟩}   (preamble-only, initially `perl`, default: see below)

This key selects or executes the extern ⟨*extraction method*⟩, i.e. the method which Memoize will use to extract the extern pages out of the document PDF.

Out of the box, Memoize recognizes the following ⟨*extraction method*⟩ keywords: **perl**, **python**, **tex** and **no**. The first three keywords trigger extern extraction using the methods documented in sections 5.5.1 and 5.5.2. The final keyword (not available in plain TeX) instructs Memoize to *not* perform the extraction at the end of loading the package; it should be used when extraction is performed externally (for details, see section 1). Additional methods may be installed by defining key **/mmz/extract/**⟨*extraction method*⟩.

When invoked from `memoize.cfg` or used as a package option, this key *selects* the extraction method. In this case, the key has no default value, i.e. it is illegal to use it without an argument. The method selected by the package option overrides the method selected in `memoize.cfg`, which in turn overrides the package-initial value `perl`.

In LaTeX and ConTeXt, the selected method is automatically executed at the end of loading the package. This does not happen in plain TeX, because we want to allow the author to override the initial extraction method (`perl`) or the extraction method specified in `memoize.cfg`, even if plain TeX has no package options. In plain TeX, internal extraction must be triggered by an explicit invocation of `extract` in the "document preamble" — i.e. between `\input memoize` and `\mmzset{begin document}`. If given an ⟨*extraction method*⟩ argument, the key will execute the given extraction method; otherwise, it will execute either the initial method, `perl`, or the overwide specified in `memoize.cfg`.

In general, Memoize guards against triggering the extraction more than once. In formats other than plain TeX, invoking key `extract` from the document preamble is thus only allowed when `extract=no` was previously selected. In this case, `extract` in the document preamble behaves as in plain TeX, i.e. it triggers the given extraction method; if no extraction method is given, Memoize executes either the initial extraction method `perl`, or the extraction method specified in `memoize.cfg`. The possibility of invoking key `extract` from the document preamble makes it possible to bake Memoize into a user format (with `extract=no`), and trigger the internal extraction manually.

Executing extraction method `perl` or `python` has an additional effect of setting the `mkdir command` to the extraction script with option `--mkdir`. This obviates the need to include `mkdir` among the restricted shell commands if one is using the restricted shell mode.

5.5.1 Perl- and Python-based extraction

Perl- and Python-based extraction is triggered by `extract=perl` and `extract=python`, respectively.

/mmz/perl extraction command=⟨*system command*⟩   (no default, initially `memoize-extract.pl`)
/mmz/perl extraction options=⟨*options*⟩   (no default, initially: `-F` ⟨*format*⟩ `\jobname`)
/mmz/python extraction command=⟨*system command*⟩ (no default, initially `memoize-extract.py`)
/mmz/python extraction options=⟨*options*⟩   (no default, initially: `-F` ⟨*format*⟩ `\jobname`)

These keys determine the system calls used for invoking the extraction scripts `memoize-extract.pl` and `memoize-extract.py`. All the details below apply both to the Perl and the Python version.

Use `perl/python extraction command` to set the name of the extraction script. If necessary, include the full path to the script, or `perl/python` plus the path to the script. Whatever you set here must be allowed by the shell escape mode.

Use `perl/python extraction options` to set the options that the script will receive; consult the documentation of `memoize-extract.pl` for their meaning. The initial value asks the script to process ⟨*document name*⟩`.mmz`, and informs it that it is executed from within a TeX compilation of

a document in the given ⟨*format*⟩ (`latex` for LATEX, `plain` for plain TEX, `context` for ConTEXt); the latter makes the script yield any warnings or errors in a form expected by the format.

During the execution of a system call, the values of these settings are fully expanded. Furthermore, these keys were initialized using `pgfkeys` handler `.initial`, so their values may be modified by handlers `.prefix`, `.append`, etc. The initial value of the extraction options contains a space on both sides, so that these handlers are easy to use. For example, write `perl extraction options/.append=--quiet` to ask for less output.

`memoize-extract.pl` [⟨*options*⟩] ⟨*name*⟩`.mmz`
`memoize-extract.py` [⟨*options*⟩] ⟨*name*⟩`.mmz`

These scripts extract the new externs recorded in ⟨*name*⟩`.mmz` from ⟨*name*⟩`.pdf`. Memoize invokes them when loaded with package option `extract`=`perl` (the default) or `extract`=`python`.

Argument ⟨*name*⟩`.mmz` may be given either in full (e.g. `doc.mmz`), or merely as the stem (`doc`). In the latter case, `.mmz` is appended to the given argument even if it already contains a suffix (e.g. `my.doc` will result in `my.doc.mmz`); the exception is suffix `.tex`, which is replaced by `.mmz`.

The script inspects the given record file, ⟨*name*⟩`.mmz`, for lines of form `\mmzNewExtern{`⟨*extern filename*⟩`}{`⟨*extern page number*⟩`}{`⟨*expected width*⟩`pt}{`⟨*expected height*⟩`pt}`. For each such line, page number ⟨*extern page number*⟩ is extracted from ⟨*name*⟩`.pdf` into ⟨*extern filename*⟩. (The script also creates directories specified by `\mmzPrefix` lines. Other lines are ignored, and so are commented invocations of `\mmzNewExtern` and `\mmzPrefix`.)

The ⟨*extern filename*⟩ may contain a (relative or absolute) path to the new extern file. The relative paths are relative to the location of the `.mmz` file, even when the script is invoked from some other directory.

To guard against extracting a wrong page, the script checks whether the size of each extracted page matches the ⟨*expected width*⟩ and ⟨*expected height*⟩.[68] If it does not, the script refuses to extract the page, yields a warning and even removes the extern file if it exist.

The extraction script's paranoia extends further. It will refuse to extract the page, yielding a warning, if a (c)c-memo associated to the extern does not exist. And it will respect the `openin` and `openout` settings of `texmf.cnf`; under the default configuration, it will therefore refuse (yielding an error) to write to any file whose absolute path does not occur under the current directory, the temporary directory set by `TEXMFOUTPUT` (in `texmf.cnf` or as an environment variable), or the output directory (`TEXMF_OUTPUT_DIRECTORY`). Furthermore, it will refuse to write into the root directory (except on Windows, where writing into a drive root might potentially make sense).

Exit codes: 0 = success, 1 = Perl/Python error, 2 = usage error, 10 = extraction warning, 11 = extraction error.

`-P | --pdf` ⟨*pdf*⟩

Extract the externs from the given ⟨*pdf*⟩ instead of the default ⟨*name*⟩`.mmz`. Note that file ⟨*pdf*⟩, despite the different name, should be produced by the same compilation that produced ⟨*name*⟩`.mmz`, otherwise wrong pages might be extracted.

`-p | --prune`

After extraction, remove the extracted extern pages from the document PDF.

`-k | --keep`

By default, the script comments out the `\mmzNewExtern` lines in the `.mmz` file, to prevent multiple extractions. Specifying this option prevents this behaviour.

---

[68]To avoid false positives, the match need not be exact, a difference up to `0.01pt` is tolerated. Some PDF tools, notably the `PDF::API2` library deployed by the Perl version of the script, round the dimensions of a PDF page, recorded in `/MediaBox`, to two digits.

-F | --format latex|plain|context

When this option is given, the script assumes that it was called from within a TeX compilation of a document in the given format (`latex` for LaTeX, `plain` for plain TeX, `context` for ConTeXt).

For one, the script will prefix all its output by the script name, to be easily identifiable in the terminal output, but more importantly, it will create a log file (`\jobname.mmz.log`) which will receive any warnings and errors yielded by the script (in absence of this option, the warnings and errors are printed to the standard error). The log contains messages in a form recognized by the format (e.g. `\PackageError` for LaTeX), so Memoize can then `\input`s this log to reproduce the warnings and errors in the compilation.

There should be no need to use this option when executing the script form the command line.

-f | --force

Force extern extraction even if the size check or (c)c-memo existence check fails. The failure will still yield a warning.

-q | --quiet

Normally, the script prints what it is doing to the standard output; in particular, it prints out the page number and the filename of each extern it is extracting. This option disables this behaviour.

-L | --library PDF::API2|PDF::Builder   (Perl-only)

Use the specified library for PDF processing. If this option is not given, the script uses the first available library (in the order given above).

-m | --mkdir

When given this option, the extraction script transforms into a paranoid `mkdir` (`-p`). Argument ⟨*name*⟩`.mmz` is interpreted as a path to the directory to create; all other options are ignored.

The ancestors of the directory are created as needed. The script will refuse to create any directory whose absolute path does not occur under the current directory or a directory listed in `TEXMFOUTPUT` (set in `texmf.cnf` or as an environment variable).

This option exists so that the author using the restricted shell mode does not have to list `mkdir` among the restricted shell mode commands (and it is also safer than a plain `mkdir`).

Memoize automatically uses this script to create the memo directory when option `extract` is given value `perl` or `python`.

-V | --version

Show Memoize version.

-h | --help

Show help.

Functionally, the Perl (`.pl`) and the Python (`.py`) version of the script are almost equivalent. The minor differences, listed below, are mostly due to the underlying PDF-processing library: `PDF::API2` in Perl, and `pdfrw2` in Python.

- The Python script is about twice as fast as the Perl script. However, both scripts are very fast compared to the TeX-based extraction. On my computer, extracting 160 externs out of a 400-page book takes 1.4s with Python, 3.7s with Perl, and 65s with TeX. But even when using TeX-based extraction, externalization using Memoize is extremely fast compared to the TikZ's externalization library. Adding the regular compilation time of one minute to the above numbers, we arrive at the maximum externalization time of about two minues, whereas my estimate for the production of all 160 externs using TikZ's externalization would be an *hour* or more.

- The Python script cannot extract externs out of PDFs created without stream compression, i.e. with `\pdfvariable compresslevel` set to 0.

- Occasionally, the Perl script crashes during extraction externs; see section 6.3 for details.

T_EX-based extraction is triggered by `extract=tex`.

/mmz/**tex extraction command**=⟨*system command*⟩	(no default, initially `pdftex`)
/mmz/**tex extraction options**=⟨*options*⟩	(no default, initially: see below)
/mmz/**tex extraction script**=⟨*T_EX code*⟩	(no default, initially: see below)

Together, these keys determine the system call used for invoking the T_EX-based extraction script, `memoize-extract-one.tex`. (They were initialized using `pgfkeys` handler `.initial`, so their values may be modified by handlers `.prefix`, `.append`, etc.)

Memoize uses the resulting system call string at the following occasions. First, it executes it, once for each new extern in the `.mmz` file, during the internal extraction, i.e. when it is loaded with package option `extract=tex`. Second, it uses it to construct record files of types `sh`, `bat` and `makefile` when recording of these types is requested.

Key `tex extraction command` sets the T_EX binary used for T_EX-based internal extraction. By default, the pdfT_EX engine `pdftex` is used; other sensible values for this key are `luatex` and `xetex`, or the program name including the path. Note that shell escape mode must be configured appropriately, and that `memoize-extract-one.tex` must be compiled with the plain T_EX format.

The value of key `tex extraction options` is passed as options to the plain T_EX binary. As shown in the initial value in the frame below, it may use the temporary macro **\externbasepath**, which expands to the path to the extern without the `.pdf` suffix. This macro is available during internal T_EX-based extraction and during the execution of /mmz/record/⟨*record type*⟩/**new extern** key for and ⟨*record type*⟩.

The initial value of `tex extraction options`

```
-halt-on-error
-interaction=batchmode
-jobname "\externbasepath"
```

Key `tex extraction script` defines the command-line script executed to extract the extern. The default value, shown in the frame below, invokes `memoize-extract-one.tex` after setting its parameter macros. Temporary macros **\pagenumber**, **\expectedwidth** and **\expectedheight** are defined at the same occasions as **\externbasepath** above, i.e. during internal T_EX-based extraction and during the execution of a `new extern` key. The initial value requests the extern to be of the same version as the main document, if possible;[69] note that Memoize defines `\pdfmajorversion` and `\pdfminorversion` in LuaT_EX.

The initial value of `tex extraction script`

```
\def\noexpand\fromdocument{\jobname.pdf}%
\def\noexpand\pagenumber{\pagenumber}%
\def\noexpand\expectedwidth{\expectedwidth}%
\def\noexpand\expectedheight{\expectedheight}%
\def\noexpand\logfile{\jobname.mmz.log}%
\unexpanded{%
  \def\warningtemplate{%
    %<latex>\noexpand\PackageWarning{memoize}{\warningtext}%
    %<plain>\warning{memoize: \warningtext}%
    %<context>\warning{memoize: \warningtext}%
  }}%
\ifdef\XeTeXversion{}{%
  \def\noexpand\mmzpdfmajorversion{\the\pdfmajorversion}%
  \def\noexpand\mmzpdfminorversion{\the\pdfminorversion}%
}%
\noexpand\input memoize-extract-one
```

---
[69]As far as I know, it is impossible to access the version of the PDF being produced in X_ET_EX, i.e. there are no registers `\pdfmajorversion` and `\pdfminorversion`. To request production of a specific version of PDF, X_ET_EX must be invoked by with command-line option `-output-driver 'xdvipdfmx -V N'`.

As the value of `tex extraction script` is fully expanded when used, the initial value shown above must prevent the expansion of much code. Furthermore, the initial value varies with the TeX format, as indicated by the `.dtx` guards in the definition of `\warningtemplate`.

`(pdf|lua|xe)tex -jobname ⟨extern filename⟩ "⟨parameters⟩ \input{memoize-extract-one.tex}"`

Compiling `memoize-extract-one.tex` with plain TeX produces an extern file containing a single (extern) page extracted from the document PDF.

Memoize invokes this script, once for each new extern appearing in the `.mmz` file, when loaded with package option `extract=tex`.

The desired ⟨*extern filename*⟩ is given as the value of option `-jobname` of the TeX binary. To set the extraction ⟨*parameters*⟩, define the following macros before `\input`ting the file:

`\def\fromdocument{⟨document pdf filename⟩}`

> Defining this macro sets the filename of the PDF which the externs will be extracted from. The filename is relative to the working directory.

`\def\pagenumber{⟨number⟩}`

> Defining this macro sets the number of the page to extract. The first page has number 1.

`\def\expectedwidth{⟨dimension⟩}` (optional)
`\def\expectedheight{⟨dimension⟩}` (optional)

> Defining these macros sets the expected width and height of the extracted page.
>
> To guard against extracting a wrong page, the dimensions of the extracted page are compared against the expected width and height. If the size check fails,[70] the resulting extern PDF is empty (which counts as non-existent when Memoize checks for its presence when it attempts to utilize it), and a warning message (formatted via `\warningtemplate`) is printed to the log file, if logging was requested via `\logfile`.
>
> If any of these macros is undefined, the size check will be skipped.

`\def\logfile{⟨filename⟩}` (optional)

> Defining this macro sets the name of the log file. If not defined, no log file will be produced.
>
> The log file is intended to be used when the script is invoked from an outer TeX compilation. In particular, it is intended to be `\input` by that compilation to see whether the extraction was successful. Upon a failed size check, it will contain a warning (formatted by `\warningtemplate`, if that macro is defined). The log file ends with `\endinput` to signal that extraction actually took place.

`\def\warningtemplate{⟨code⟩}` (optional)

> Defining this macro determines how to log the warning message in the case of a failed size check. The macro should expand to a TeX format-specific warning message code containing the warning text given in `\warningtext`.[71]
>
> While the script formats the warning message *text* on its own ("I refuse to extract page …"), the warning message is not written into the log unadorned. The log file is intended to be `\input` by the outer TeX compilation, and the idea is that inputting it should yield a warning in that compilation (in the case of a failed size check). Therefore, the content of the log file must contain an invocation of the command used to produce warning messages in the TeX format used by the outer compilation.
>
> For example, when this script is invoked from within a LaTeX compilation, it makes sense to define something like `\def\warningtemplate{\PackageWarning{memoize}{\warningtext}}`.

---

[70]The match need not be exact, see footnote 68.

[71]Macro `\warningtemplate` is passed the warning text by a macro rather than a formal parameter to avoid category code problems with the parameter character when setting key `tex extraction script`.

`\def\force{true|false}` (optional)

> If this macro is defined to `true`, extern extraction will be carried out even if the size-check fails. The failure will still be logged.

`\def\mmzpdfmajorversion{⟨number⟩}` (optional)
`\def\mmzpdfminorversion{⟨number⟩}` (optional)

> Defining (one or both of) these macros requests that the extern PDF be produced with the given major/minor PDF version, i.e. the extraction script will set registers `\pdfmajorversion` and `\pdfminorversion`.

After extracting the extern, the script will end the compilation, i.e. intentionally, only one page documents can be produced.

5.5.3 The clean-up scripts

`memoize-clean.pl` [⟨options⟩] [⟨name⟩`.mmz` …]
`memoize-clean.py` [⟨options⟩] [⟨name⟩`.mmz` …]

This script removes memo and extern files whose filenames start with ⟨prefix⟩es mentioned in the given `.mmz` files or by the `--prefix` option. Unless option `--all` is given, the script only deletes the *stale* files, i.e. the files not mentioned in any of the given `.mmz` files.

A ⟨prefix⟩ of a memo or an extern is what was set by key `prefix`, or more commonly, one of the shortcut keys `memo dir` and `no memo dir`; see section 5.4 for details on the form of a memo/extern filename.

In detail, the script scans the given `.mmz` files for occurrences of `\mmzPrefix`, and adds their ⟨prefix⟩ arguments to the list of prefixes given on the command line by option `--prefix`; a ⟨prefix⟩ occurring in some `.mmz` file is interpreted relatively to the location of the `.mmz` file. The script removes all files whose full pathname (relative to the current directory) matches pattern ⟨prefix⟩⟨md5sum⟩(-⟨md5sum⟩)(`.memo`|(-N)`.pdf`|`.log`),[72] except those which occur as the ⟨filename⟩ argument to one of `\mmzUsedCMemo`, `\mmzUsedCCMemo`, `\mmzUsedExtern`, `\mmzNewCMemo`, `\mmzNewCCMemo` and `\mmzNewExtern` in one of the `.mmz` files.

The script is fairly paranoid. It refuses to delete anything if a `.mmz` file is malformed in any way (but not if it doesn't exist or is completely empty, which facilitates its usage in clean-up scripts), or if it would remove a file not residing under the current directory. Before removing the files, it lists the files to be removed and asks for confirmation.

Functionally, the Perl (`.pl`) and the Python (`.py`) version are completely equivalent.

`-p | --prefix ⟨prefix⟩`

> Add ⟨prefix⟩ to the list of prefixes; the given prefix is relative to the current directory. This option may be given multiple times.

`-a | --all`

> When given this option, the script removes *all* memos and externs belonging to the document, not just the stale ones, i.e. it effectively ignores the occurences of `\mmzUsedCMemo` and friends in the `.mmz` file.

`-y | --yes`

> When given this option, the script does not ask for confirmation before removing the files.

---

[72]The `.log` files are produced by the TeX-based extraction script.

-q | --quiet

Normally, the script prints what it is doing to the standard output; in particular, it prints out the filename of each file as it is deleting it. This option disables this behaviour.

-h | --help

Show help.

-V | --version

Show Memoize version.

5.5.4 Record files

/mmz/**record**={⟨*record type*⟩}   (cumulative, initially mmz, no default)
/mmz/**no record**

Memoize records which externs were produced and used in the compilation, producing a record file of every type found in the record-type list. These keys add ⟨*record type*⟩ to the record-type list, or clear this list. See section 4.3 for details.

Note that passing an undefined ⟨*record type*⟩ to this key will not yield an error.

Out of the box, the following ⟨*record type*⟩s are recognized:

mmz

This record type produces a `.mmz` file recording new/used externs/c-memos/cc-memos and changes in the `prefix` to these files; see section 4.3.1 for details.

The produced file is named ⟨*jobname*⟩`.mmz`. This name cannot be changed.

The `.mmz` file is a TEX file, but uses only a simple subset of the TEX syntax, to be easily parsable by the external scripts such as `memoize-extract.pl`. Each line of the file consists of a (possibly commented) invocation of one of the commands listed below; the final line is `\endinput`. The ⟨*prefix*⟩ below consists of the path to memos/externs and the immutable `prefix` of their filename.

\mmzUsedCMemo{⟨*filename*⟩}
\mmzUsedCCMemo{⟨*filename*⟩}
\mmzUsedExtern{⟨*filename*⟩}

Record that the (c)c-memo or extern residing in file ⟨*filename*⟩ was utilized.

\mmzNewCMemo{⟨*filename*⟩}
\mmzNewCCMemo{⟨*filename*⟩}

Record that a new (c)c-memo residing in file ⟨*filename*⟩ was produced.

\mmzNewExtern{⟨*filename*⟩}{⟨*page number*⟩}{⟨*expected width*⟩}{⟨*expected height*⟩}

Record that a new extern was produced and dumped as page ⟨*page number*⟩ into the document, that it should be extracted into file ⟨*filename*⟩, and that it should be ⟨*expected width*⟩ wide and ⟨*expected height*⟩ high (modulo tolerance of `0.01pt`, see footnote 68), where the height is the total height comprising both TEX height and depth.

\mmzPrefix{⟨*prefix*⟩}

Record that the `prefix` of memo and extern files was changed.

makefile

> This record type produces a makefile which, when processed by the `make` utility, triggers TeX-based extraction of the new externs.
>
> /mmz/**makefile**={⟨*filename*⟩} (no default, initially `memoize-extract.\jobname.makefile`)
>
> > Use this key to change the filename of the produced makefile.

sh
bat

> These record types produce a shell script which, when executed, triggers TeX-based extraction of the new externs.
>
> Use `sh` on Unix-like systems, and `bat` on Windows.
>
> /mmz/**sh**={⟨*filename*⟩}   (no default, initially `memoize-extract.\jobname.sh`)
> /mmz/**bat**={⟨*filename*⟩}   (no default, initially `memoize-extract.\jobname.bat`)
>
> > Use these keys to change the filename of the produced shell script.

/mmz/record/⟨*record type*⟩/**begin**	(definable)
/mmz/record/⟨*record type*⟩/**prefix**={⟨*prefix*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**new extern**={⟨*filename*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**new cmemo**={⟨*filename*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**new ccmemo**={⟨*filename*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**used extern**={⟨*filename*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**used cmemo**={⟨*filename*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**used ccmemo**={⟨*filename*⟩}	(definable)
/mmz/record/⟨*record type*⟩/**end**	(definable)

A new record type can be implemented by defining these keys in keypath **/mmz/record/**⟨*record type*⟩ (using the standard `pgfkeys` handlers such as `.code` and `.style`). The keys are invoked by Memoize where appropriate if recording for the defined type is activated by **record**=⟨*record type*⟩, just as for the predefined types. Only those keys which are required for implementing the desired functionality need to be defined.

The following macros are available during the execution of key `new extern`:

\pagenumber

> This macro holds the number of the extern page.

\expectedwidth
\expectedheight

> These macros hold the width and the height of the extern page.

\externbasepath

> This macro holds the filename of the extern, minus the `.pdf` suffix (but including the path leading to the extern).

5.6.1   Package Advice

Package Advice is a namesake of Emacs's Advice. As such, it implements a generic framework for extending the functionality of selected commands and environments. Each *advised* command and environment is assigned a piece of *advice* — a command which is executed instead of the advised command and environment, and which may, or may not, invoke the original command or environment during its execution. The package offers an elegant way of declaring advice, setting up the conditions upon which the advised command will actually be replaced by the advice, collecting the arguments of the advised command and invoking it, and (de)activating the advice.

Before the advising framework can be used, it must be installed into a selected `pgfkeys` keypath (multiple installations into different keypaths are allowed, even if they handle the same commands). Memoize installs the framework into keypath `/mmz` and (primarily) uses it to automatically memoize the results of compilation of selected commands and environments.

`/handlers/`**`.install advice`**`={⟨`*configuration keylist*`⟩}`   (handler)

> This key is a `pgfkeys` key handler (see §87.3.5 of the Ti*k*Z & PGF manual) which installs the advising framework into the keypath which it was invoked from — henceforth, the ⟨*namespace*⟩.
>
> For example, `\pgfkeys{/my/.install advice}` installs the framework into keypath `/my`.
>
> Argument ⟨*configuration keylist*⟩ may contain the following keys:
>
> `/advice/install/`**`setup key`**`={⟨`*name*`⟩}`   (no default, initially `advice`)
>
>> This key determines the names of the user-interface keys used to setup advice for commands and environments in ⟨*namespace*⟩.
>>
>> The keys whose names are determined by this key are the following: ⟨*name*⟩, ⟨*name*⟩ `csname`, ⟨*name*⟩ `key`, ⟨*name*⟩`'`, ⟨*name*⟩ `csname'` and ⟨*name*⟩ `key'`. Memoize sets `setup key`=`auto`, and thereby defines `auto`, `auto csname`, `auto key`, `auto'`, `auto csname'` and `auto key'`.
>
> `/advice/install/`**`activation`**`=⟨`*initial activation type*`⟩`   (no default, initially `immediate`)
>
>> This key sets the ⟨*initial activation type*⟩ for ⟨*namespace*⟩.
>>
>> At the end of the installation, the system will execute ⟨*namespace*⟩`/activation`=⟨*initial activation type*⟩; consequently, ⟨*initial activation type*⟩ must be one of `immediate` and `deferred`. In Memoize, the ⟨*initial activation type*⟩ is `deferred`.
>>
>> Setting the `activation` type during the installation only matters in LATEX, where the installation ends by advising `\begin` to implement advising of environments.

> Writing the documentation for Advice, I was faced with a dilemma. Should the documentation reflect the fact that the full names of keys defined by the package depend on the installed instance of the framework, in particular on ⟨*namespace*⟩ and ⟨*setup key*⟩? For example, should the reference headers contain things like ⟨*namespace*⟩`/activate` and ⟨*namespace*⟩`/`⟨*setup key*⟩ `csname`? In my opinion, this would make the reference hard to read, so I decided to have the reference headers refer to the Advice keys of the Memoize installation, where ⟨*namespace*⟩=`/mmz` and ⟨*setup key*⟩=`auto`, resulting in friendlier headers such as `/mmz/activate` and `/mmz/auto csname`. (Consequently, it also made sense to document Advice within the Memoize documentation.)
>
> The bottomline: if you're reading this section with a non-Memoize installation in mind, you have to mentally replace any `/mmz` and `auto` in the reference headers with the ⟨*namespace*⟩ and the ⟨*setup key*⟩ selected by that installation. (Section 5.6.2 is another matter. Keys described there are only available in Memoize.)

In more detail, key handler `.install advice` performs the following actions (as explained in the box above, we assume that the advising framework was installed into keypath `/mmz` with the setup key named `auto`):

- It defines the following keys in keypath `/mmz`: `auto`, `auto csname`, `auto key`, `auto'`, `auto csname'`, `auto key'`, `activation`, `activate deferred`, `activate`, `deactivate`, `activate csname`, `deactivate csname`. `activate key`, `deactivate key`. `force activate`, `try activate`.

- It defines the following keys in keypath `/mmz/auto`: `run conditions`, `outer handler`, `bailout handler`, `collector`, `args`, `collector options`, `clear collector options`, `raw collector options`, `clear raw collector options`, `inner handler`, `options`, `clear options`, `reset`.

- It defines the `.unknown` key handler for `/mmz/auto`. This handler appends any unknown keys (and their values) to `options`.

- It executes `/mmz/activation`=⟨*initial activation type*⟩.

- In LaTeX, it submits `\begin` to advising, thereby enabling environment support in this format. Consequently, advising of environments can switched off by writing `deactivate`=`\begin`.

The keys installed into keypath ⟨*namespace*⟩ are used to declare and (de)activate advice. In the documentation in this subsection, we assume that ⟨*namespace*⟩=`/mmz` and that ⟨*setup key*⟩=`auto`. In particular, this also applies to the reference headers.

/mmz/**activation**=`immediate`|`deferred`   (no default, initially set by `.install advice`)
/mmz/**activate deferred**   (style)

Key `activation` selects the activation regime. Under the `immediate` regime, keys `activate`, `deactivate`, `force activate` and `try activate` behave as described in their documentation below. Under the `deferred` regime, however, those keys are not executed; rather, their invocations are appended to style `activate deferred`. For example, writing `activate`=`\foo` in the deferred activation regime appends `activate`=`\foo` to `activate deferred`. It is up to the user if and when to execute the keys collected in `activate deferred`; see the documentation of `manual` to learn what Memoize does with the contents of this style.

/mmz/**activate**={⟨*list of commands and/or environments*⟩}   (style)
/mmz/**deactivate**={⟨*list of commands and/or environments*⟩}   (style)

These keys activate or deactivate the advice for the given commands and environments. When the advice is activated, it replaces the advised command; when it is deactivated, the command is reverted to its original definition.

In Memoize, these keys are most commonly used to activate or deactivate automemoization for the given commands or environments. For example, write `deactivate`={`\tikz,tikzpicture`} to deactivate automemoization of TikZ pictures (which is declared and active by default). The curly braces may be omitted if the list contains a single command or environment, e.g. `deactivate`=`\tikz` or `deactivate`=`tikzpicture`.

(De)activation of a piece of advice is completely orthogonal to its declaration with `auto`. For example, there is no need to deactivate a command before redeclaring its advice, and reactivate it afterwards. A command may be activated even before declaring its advice — however, the command itself must be defined at the time of activation.

As the advice is normally automatically activated upon declaration with `auto`, explicit activation is rarely needed, but see `auto'`. The effect of these keys under the deffered activation regime is described in `activation`.

Note that I sometimes speak of (de)activating a command, and sometimes of (de)activating its advice. I mean the same thing.

/mmz/**activate csname**={⟨*control sequence name*⟩}   (style)
/mmz/**deactivate csname**={⟨*control sequence name*⟩}   (style)

These keys activate and deactivate a command given by its ⟨*control sequence name*⟩; for example, `activate csname`=foo is equivalent to `activate`=\foo. Note that unlike the regular `activate` and `deactivate`, their `csname` variants only accept a single command at a time (otherwise, including a comma in the command name would be impossible).

/mmz/**activate key**={⟨*list of full key names*⟩}   (style)
/mmz/**deactivate key**={⟨*list of full key names*⟩}   (style)

These keys activate and deactivate `pgfkeys` keys. Note that *full* key names must be given, i.e. the names must include the keypath.

Under the hood, these keys merely execute `activate` and `deactivate` on the internal macros corresponding to the given keys.

/mmz/**try activate**=true|false   (default `true`, initially `false`)

When this conditional is set to true, `activate` will not yield an error if the advice is already activated, and `deactivate` will not yield an error if the advice is not yet activated.

This key applies to the next, and only to the next, invocation of key `activate` or `deactivate`, i.e. it is reset back to `false` after invoking `activate` or `deactivate`.

/mmz/**force activate**=true|false   (default `true`, initially `false`)

When this conditional is set to true, `activate` will activate even a previously activated command, provided that, additionally, the command has been redefined since the prior activation.

In more detail, the original definition of the advised command is saved upon activation (to provide the possibility of both deactivation and the usage of the original command by the handler). Consequently, activation of an already activated command would result in the saved original definition being overwritten by the redefinition made during the first activation. However, if the handled command was meanwhile redefined by a third party, reactivation makes sense, under the assumption that the former original definition is obsolete and should be replaced by the (third party) redefinition. As a safeguard, however, `activate` requires such reactivation to be explicitly requested using conditional `force activate`.[73]

This key applies to the next, and only to the next, invocation of key `activate`, i.e. it is reset back to `false` after invoking `activate`. This key does not apply to `deactivate`.

/mmz/**auto**={⟨*command or environment*⟩}{⟨*keylist*⟩}   (style)

This key sets up the advice for the given command or environment, or updates the configuration of an existing piece of advice.

In Memoize, this key is most commonly used to submit a command or environment to automemoization. For an environment (say, `bar`), it suffices to write `auto`={bar}{memoize}; for a command, we usually need to include its argument specification: `auto`=\foo{memoize, args={...}}. Another common usage is to prevent memoization during the execution of a command or environment: `auto`={bar}{nomemoize}. For details, see sections 2.3, 2.4 and 2.9.

Note that in LaTeX, advising an environment (say, `bar`) is different than advising the internal command (\bar) containing the ⟨*begin-code*⟩ of the environment, created by \newenvironment. Advising environment `bar` effectively replaces the entire \begin{bar}...\end{bar} construction, so that LaTeX's \begin code is never executed. The advice of command \bar, on the other hand, is executed after \begin initializes the environment; in particular, it is executed within the group introduced by the environment.

---

[73]A potential problem, not (yet) addressed, is that the third party might be another incarnation of the advising framework. In this case, forced reactivation will result in the loss of the original command and a circular dependency between the two pieces of advice.

The advice is configured by the given ⟨*keylist*⟩, which is executed with the default keypath set to `/mmz/auto`. Any unknown keys in ⟨*keylist*⟩ are passed on to key `options`; for example, a plain `verbatim` or `padding=2in` have the same effect as `options=verbatim` or `options={padding=2in}`.

This key automatically activates the declared advice, unless it is already activated; under the deferred activation regime, the automatic activation is deferred as well. Use variant `auto'` when you don't want to automatically activate the advice.

When this key is used on a command or environment with an existing piece of advice, the advice is merely updated. This makes it easy to, for example, temporarily switch to verbatim collection of an environment in Memoize: `auto={tcolorbox}{verbatim}`. Use key `reset` to setup the advice from scratch: `auto={...}{reset, ...}`.

A piece of advice consists of several interlocked components, declared by keys residing in path `/mmz/auto`: `run conditions`, `bailout handler`, `outer handler`, `collector` and `inner handler`. During the execution of the advice, these components are available through the following macros: `\AdviceRunConditions`, `\AdviceBailoutHandler`, `\AdviceOuterHandler`, `\AdviceCollector` and `\AdviceInnerHandler`. These macros are also defined during setup, and it is possible to change the configuration by modifying them directly; in that case, it likely also makes sense to use the low-level variant of this key, macro `\AdviceSetup`.

Control sequences used in the advice components do not need to be defined at the time of invoking key `auto`, or activating the advice; they must only be defined at the time the advice is actually executed. It is thus perfectly fine to declare `inner handler=\myinnerhandler` before defining `\myinnerhandler`, or to redefine `\myinnerhandler` between the invocations of the advised command.

This key configures not only the components, but also the options of the advice. These options are set by keys `args`, `collector options`, `raw collector options` and `options`. Whether these options are used or not depends on the advice components. (Same as the components, the options have their corresponding low-level macros: `\AdviceArgs`, `\AdviceCollectorOptions`, `\AdviceRawCollectorOptions` and `\AdviceOptions`.)

Parameter symbols (i.e. `#`) are not allowed in advice settings.

A command or environment may be submitted to several instances of the advising framework, i.e. instances installed under different keypaths. The effect of such chained advice depends on the order of activation. If advice *A* is activated before advice *B*, it will also be applied before *B*.

The advice setup takes place in a group. Use key `after setup` to execute code outside this group.

In general, the name of this key equals whatever was submitted to `setup key` during the installation of the advising framework via `.install advice`; the initial value is `advice` (and Memoize sets the value to `auto`).

`/mmz/`**`auto csname`**`={`⟨*control sequence name*⟩`}{`⟨*keylist*⟩`}` (style)

This key is a variant of `auto`, but with the command of the first argument given as a control sequence name, i.e. `auto csname={foo}{...}` is equivalent to `auto=\foo{...}`.

`/mmz/`**`auto key`**`={`⟨*full key*⟩`}{`⟨*keylist*⟩`}` (style)

This key is a variant of `auto`, but it works with `pgfkeys` keys. The first argument should be a ⟨*full key*⟩ like `/tcb/float`, i.e. it must consist of both the keypath and the keyname.

This key sets up advice for the internal command corresponding to the given ⟨*full key*⟩, and also properly initializes the collector, so that `inner handler` will "just work."

`/mmz/`**`auto'`**`={`⟨*command or environment*⟩`}{`⟨*keylist*⟩`}` (style)
`/mmz/`**`auto csname'`**`={`⟨*control sequence name*⟩`}{`⟨*keylist*⟩`}` (style)
`/mmz/`**`auto key'`**`={`⟨*full key*⟩`}{`⟨*keylist*⟩`}` (style)

These keys are variants of `auto`, `auto csname` and `auto key` which do not attempt to activate the command after setting it up.

\AdviceSetup{⟨*namespace*⟩}{⟨*command or environment*⟩}{⟨*setup code*⟩}

This macro is the low-level variant of key `auto`. The differences between the two are the following:

- An invocation of the macro must provide the namespace (i.e. the installation keypath) as the first argument.
- There is no automatic activation at the end of the setup.
- The final argument should not be a keylist (of keys belonging to `/mmz/auto`) but TEX code adjusting the contents of the settings macros **\AdviceRunConditions**, **\AdviceBailoutHandler**, **\AdviceOuterHandler**, etc. For the full list of available macros, see the documentation of their corresponding keys below; the setting macros are mentioned at the end of each entry.

\AdviceTracingOn
\AdviceTracingOff

Advice tracing is initially off. When it is on, Advice will show (on the terminal and in the `.log` file) which advice components are executed, and what arguments and options they have received.

The keys installed into keypath ⟨*namespace*⟩/⟨*setup key*⟩ are used to configure advice. They may only occur within the second argument of the setup key. In the documentation in this subsection, we assume that ⟨*namespace*⟩=`/mmz` and that ⟨*setup key*⟩=`auto`. In particular, this also applies to the reference headers.

`/mmz/auto/`**run conditions**=⟨*TEX code*⟩   (initially and default: **\AdviceRuntrue**)

This key declares the ⟨*control sequence*⟩ as the run conditions component of the advice.

The run conditions macro is executed at the very start of the advice. Its function is to decide whether we should proceed to advise the command by executing the outer handler, or execute the original command (after invoking the bailout handler).

The run conditions macro should take no arguments. If it determines that the run conditions are satisfied, it should set the TEX conditional **\ifAdviceRun** to true by executing **\AdviceRuntrue**. There is no need to execute **\AdviceRunfalse** when the run conditions are not satisfied.

Initially, the run conditions are set to **\AdviceRuntrue**, translating to "always run." For two non-trivial examples, see `run if memoization is possible` and `run if memoizing`. Executing this key without a value restores it to the initial value.

During advising and advice setup, the run conditions of the advised command are accessible through **\AdviceRunConditions**, a parameterless macro expanding to the given ⟨*TEX code*⟩.

`/mmz/auto/`**bailout handler**=⟨*TEX code*⟩   (initially and default: **\relax**)

This key declares the ⟨*TEX code*⟩ as the bailout handler component of the advice.

The bailout handler is executed when the run conditions are not met, just prior to executing the original definition of the advised command. The bailout handler should take no arguments.

The initial bailout handler, **\relax**, does nothing. Memoize defines and uses a bailout handler which clears the next-options. Executing this key without a value restores it to the initial value.

During advising and advice setup, the bailout handler of the handled command is accessible through **\AdviceBailoutHandler**, a parameterless macro expanding to the given ⟨*TEX code*⟩.

(initially and default: `\AdviceCollector`)

This key declares the ⟨*TEX code*⟩ as the outer handler component of the advice.

The outer handler can be safely imagined as the command which replaces the handled command. This also holds for handled environments, but with a caveat: for a plain TEX or ConTEXt environment `foo`, the outer handler replaces `\foo` and `\startfoo`, respectively; in the case of a LATEX environment, it replaces `\begin{foo}`.

The outer handler is the first component which has the opportunity to inspect the arguments given to the handled command. It is invoked just in front of these arguments (which are, in case TEX hasn't seen them yet, untokenized), and while it is expected that the advice will consume the same arguments as the advised command itself would, how precisely that happens may vary from situation to situation. In particular, the argument structure of the outer handler is not prescribed.

In fact, the outer handler has complete control over the remainder of the advising process. In situations where advising requires knowledge of the advised command's arguments as a whole, the outer handler executes the collector, which in turn invokes the inner handler, which does the real work; see `memoize` for the usage case which inspired this design. Sometimes, however, it is the outer handler which does the real work (and there is thus no inner handler). This is the case in situations when the arguments of the handled command are irrelevant for the functioning of the advice, or when the advice needs to inspect some individual argument of the handled command; for examples of such situations, see `abort` and `ref`.

To reiterate the argument situation of the outer handler, it sees the arguments of the handled command as they were given. The arguments are *not* collected before invoking the outer handler — in fact, avoiding the argument collection is the raison d'être of the outer handler! (In the case of an advised environment, the environment body can be seen as an argument of LATEX3 type `+b`.)

The outer handler (and any other component of the advice it invokes) has access to the following auxiliary macros, defined by the framework:

- the macros holding the configuration of the advised command, as set up by `auto`: `\AdviceRunConditions`, `\AdviceBailoutHandler` and `\AdviceOuterHandler` are probably useless, as they refer to components already invoked, but the remaining components (`\AdviceCollector` and `\AdviceInnerHandler`) and their options (`\AdviceArgs`, `\AdviceCollectorOptions`, `\AdviceRawCollectorOptions` and `\AdviceOptions`) should be commonly used.

- the macros holding information about the namespace and the advised command or environment: `\AdviceNamespace`, `\AdviceName`, `\AdviceCsname`, `\AdviceReplaced` and `\AdviceOriginal`. (Commands `\AdviceGetOriginal` and `\AdviceCsnameGetOriginal` might also be useful, although using `\AdviceOriginal` will likely be more practical.)

This key is initially set to `\AdviceCollector`; in other words, the initial outer handler leaves all the work to the collector and the inner handler. There is no need to specifically set up the outer handler when using the inner handler. Executing this key without a value restores it to the initial value.

During advising and advice setup, the outer handler of the advised command is accessible through **`\AdviceOuterHandler`**, a parameterless macro expanding to the given ⟨*TEX code*⟩.

(initially and default: see below)

This key declares the ⟨*TEX code*⟩ as the collector component of the advice.

The collector, if used, is invoked by the outer handler. It is invoked immediately in front of the advised command's arguments (which are, in case TEX hasn't seen them yet, untokenized), and its function is to collect these arguments and pass them on, as a single argument, to the inner handler.

While this manual occasionally states that the initial argument collector is `\CollectArguments` of package CollArgs, this is, if we're precise, incorrect on two counts. For one, the initial collector is not a CollArgs command, but an internal macro which acts as the "bridge" between Advice and

CollArgs. Second, the initial collector does not really invoke `\CollectArguments`, but its cousin, `\CollectArgumentsRaw`, which allows Advice (and Memoize) to fine tune its behaviour using the fast low-level ("programmer interface") commands rather than the slower `pgfkeys` interface; clearly, the latter point also provides raison d'être for `raw collector options`. Summing up, this key is initially set to an internal control sequence which compiles the settings provided by `args`, `collector options` and `raw collector options` into an invocation of `\CollectArgumentsRaw` of package CollArgs.[74] Executing this key without a value restores it to the initial value.

The above-mentioned collector settings were clearly tailored to suit `\CollectArgumentsRaw`. In general, a collector might or might not use them, and if it does, it may interpret them in any way. For example, Advice ships with a `\tikz` collector, `\mmzCollectTikZArguments`, which ignores them completely, as it knows everything about the idiosyncrasies of that command anyway. Incidentally, `\mmzCollectTikZArguments` becomes available upon loading `advice-tikz.code.tex` (which Memoize does automatically in the presence of TikZ).

The collector has access to the same auxiliary macros as the outer handler. In particular, it will *have* to use `\AdviceInnerHandler` (followed by the braced collected arguments) to invoke the inner handler.

During advising and advice setup, the collector of the advised command is accessible through `\AdviceCollector`, a parameterless macro expanding to the given ⟨*TEX code*⟩.

`/mmz/auto/`**`args`**`=`⟨*argument specification*⟩   (initially and default: unset)

This key describes the ⟨*argument specification*⟩ of the advised command.

The value of this key is intended to be interpreted by the `collector`. Assuming this key retained its initial value, the given ⟨*argument specification*⟩ is eventually interpreted by command `\CollectArguments` of package CollArgs, which expects an argument specification in the format specified by LaTeX3, also allowing for the uncommon/historical types only defined upon loading package `xparse`, and several CollArgs-specific extensions. For further information, see the following sources:

- The LaTeX3 version of "LaTeX for authors" documents the standard argument types.
- The `xparse` manual additionally documents the uncommon/historical types only available upon loading package `xparse`.
- CollArgs extensions are documented in section 5.6.3.
- For convenience, the entire LaTeX3 argument specification format, complete with `xparse` and CollArgs extensions, is summarized in the frame on page 113 in section 5.6.3.

If ⟨*argument specification*⟩ is not provided, the initial collector assumes that the advised command was defined using LaTeX3's `\NewDocumentCommand` (or similar), and will attempt to retrieve the argument specification automatically.[75]

In general, however, an argument collector may this interpret this setting in any way it sees fit — or not at all. For example, in Memoize the value of `args` is ignored for command `\tikz`, which requires a special collector (`\mmzCollectTikZArguments`).

When setting up advice for a *command*, this key is initially "unset," i.e. it holds a special value indicating that the argument specification is not provided. Note that this special value is not an empty string; `args={}`, or simply `args=`, indicates a command which takes no arguments. During the execution of the advice, one may use the LaTeX-style conditional `\AdviceIfArgs`{⟨*true branch*⟩}{⟨*false branch*⟩} to test whether the argument specification was provided. Executing this key without a value restores it to the initial, unset value.

When setting up the advice of an *environment*, this key is initialized to `+b` (a long environment body), making it unnecessary to specify this value manually. Note that this holds even for

---

[74]The initial collector also sets the CollArgs' option `caller` to the name of the advised command or environment.
[75]The automatic retrieval deploys command `\GetDocumentCommandArgSpec` of package `xparse`, which is the sole reason that Advice loads this package in LaTeX.

environments with arguments other that the environment body "argument": those arguments will be caught as the start of the body even if not explicitly specified.

During advising and advice setup, the argument specification of the advised command is accessible through **\AdviceArgs**, a parameterless macro expanding to the given ⟨*argument specification*⟩.

/mmz/auto/**collector options**={⟨*keylist*⟩}   (cumulative, initially empty, value required)
/mmz/auto/**raw collector options**={⟨*code*⟩}   (cumulative, initially empty, value required)

These keys append the given value to the list of user-friendly and raw collector options, respectively. A comma is prefixed to the user-friendly ⟨*keylist*⟩ before appending it.

Both kinds of collector options are intended to be used by the collector, which may interpret them in any way it sees fit — or not at all. The initial collector, which invokes \CollectArgumentsRaw of package CollArgs, passes both lists to this command, which interprets collector options as a user-friendly pgfkeys keylist (which therefore requires a bit of processing) and raw collector options as plain TeX code (expecting it to contain only the allowed, "programmer's interface" macros).[76] The raw variant is used internally by both Advice and Memoize, and may be used by a package deploying the advising framework which wants to save a few processing cycles. In CollArgs, the two kinds of options are functionally equivalent; both are documented in section 5.6.3.

Initially, the list of collector options is empty, and for commands, so is the list of raw collector options. For environments, however, the latter list is initialized to set (the raw equivalent of) environment to the environment name, and end tag to true. The rationale for the latter is that the environment body containing the end tag (e.g. \end{foo}) is nicely compatible with \AdviceReplaced (which equals the begin tag, e.g. \begin{foo}) and \AdviceOriginal (which executes the original definition of e.g. \begin{foo}). For example, thanks to end tag, writing \AdviceOriginal#1 in the inner handler executes the original environment. Importantly, the original environment can be executed without explicitly referring to the environment's name, and with code that works not only for environments of any TeX format, but is actually the same as the code which invokes an original *command*. Consequently, the same inner handler works for both commands and environments, and in all TeX formats.

Furthermore, the initial collector also sets option caller to the name of the advised command or environment (however, caller never appears in any of the collector options lists; it is simply prefixed to them while constructing the invocation of \CollectArgumentsRaw). And in Memoize, using keys verbatim, verb or no verbatim triggers the addition of the cognominal verbatim, verb or no verbatim among the collector options.

Precious few CollArgs' options thus remain to be set by the author. For memoization, the most likely candidates are ignore nesting and ignore other tags, which could help deal with unusual environments. Overriding the initial end tag by begin tag, end tag and/or tags might also be useful on occasion.

During advising and advice setup, the pgfkeys and the raw collector options of the advised command are accessible through **\AdviceCollectorOptions** and **\AdviceRawCollectorOptions**, both a parameterless macro expanding to the given ⟨*keylist*⟩ and ⟨*code*⟩, respectively.

/mmz/auto/**clear collector options**
/mmz/auto/**clear raw collector options**

These keys empty the list of user-friendly and raw collector options, respectively.

---

[76]Clearly, raw collector options are why Advice deploys \CollectArgumentsRaw rather than \CollectArguments. But how does it then pass the user-friendly collector options to that command? It embeds them in \collargsSet.

(initially and default: see below)

This key declares the ⟨*T_EX code*⟩ as the inner handler component of the advice.

The inner handler is intended to be used in situations which require knowledge of the advised command's arguments as a whole. In such situations, the outer handler will normally invoke the collector, which will in turn execute the inner handler and provide it with a single (braced) argument, containing the collected arguments of the advised command. See `memoize` for the usage case which inspired this design.

The simplest example of an inner handler is a (single-parameter) macro which does nothing. Surprisingly enough, such an inner handler could be useful. Defining `\def\Gobble#1{}` and setting `auto=\foo{inner handler=\Gobble, args={...}}` with the argument structure appropriate for `\foo` could be used to eradicate all invocations of `\foo` from the document.

The inner handler has access to all the macros available to the outer handler, but given that most of them have already fulfilled their function, only the following will likely be useful in the inner handler: `\AdviceNamespace`, `\AdviceName`, `\AdviceCsname`, `\AdviceReplaced`, `\AdviceOriginal`, and `\AdviceOptions`.

Because there is clearly no reasonable default for the inner handler, this key is initially set to an internal control sequence producing an "undefined inner handler" error. Note that it is not necessary to define a dummy inner handler when handling is entirely performed by the outer handler, i.e. in cases when the inner handler is not invoked. Executing this key without a value restores it to the initial value.

During advising and advice setup, the inner handler of the advised command is accessible through `\AdviceInnerHandler`, a parameterless macro expanding to the given ⟨*T_EX code*⟩.

/mmz/auto/**options**={⟨*keylist*⟩}   (cumulative, initially empty, value required)
/mmz/auto/**clear options**

The first key appends the given ⟨*keylist*⟩ to the list of advice options (after prefixing it by a comma), and the second one empties this list. For a ⟨*key*⟩ undefined in keypath /mmz/auto, ⟨*key*⟩=⟨*value*⟩ has the same effect as `options`={⟨*key*⟩=⟨*value*⟩}.

In Memoize, the options set by this key are known as *auto-options* — options which are applied (using `\mmzset`) at every invocation of the advised command or environment. For example, the `tcolorbox` environment of package `tcolorbox` is used extensively for typesetting this manual, and I have submitted this environment to automemoization. However, the `tcolorbox`es in this manual often include code listings. To memoize such environments successfully, their bodies must be grabbed verbatim. I have therefore submitted the `tcolorbox` environment to automemoization like this: `auto={tcolorbox}{memoize, options=verbatim}`; the simpler `auto={tcolorbox}` `{memoize, verbatim}` would work as well.

In general, whether to use the options set by this key, and how, remains at the sole discretion of the advice. Note that they might be used by either the outer or the inner handler, or perhaps even the collector.

During advising and advice setup, the options of the advised command are accessible through `\AdviceOptions`, a parameterless macro expanding to the given ⟨*keylist*⟩.

/mmz/auto/**reset**   (style)

Executing this key restores all `auto` keys to their initial values.

Invoking `auto` on the same command or environment again *updates* the advice configuration. Use this key to start from scratch.

/mmz/auto/**after setup**   (initially empty, cumulative)

The code given to this key will be executed after exiting the group opened by `auto`. The same effect may be achieved by appending to macro `\AdviceAfterSetup`.

For example, `integrated driver` uses this key to declare a new conditional.

Commands available during the execution of advice

With the exception of `\AdviceGetOriginal` and `\AdviceCsnameGetOriginal`, the commands listed below only become available in the outer handler, and if that handler does nothing funky, they should be available in the collector and the inner handler, as well. However, once the advice yields control to foreign code, these macros are not guaranteed to hold the expected values anymore, because the foreign code might trigger another piece of advice. Consequently, these macros should be expanded, once, before integrating them into arbitrary (non-advice) code; in particular, this applies to `\AdviceOriginal`.

`\AdviceNamespace`

This macro holds the ⟨*namespace*⟩, i.e. the keypath which this instance of the advising framework was installed into.

`\AdviceName`

This macro holds the name of the advised command or environment, i.e. the name which was used as the first argument to `auto`. For a command, this will be a control sequence, e.g. `\foo`; for environments (in any TeX format), their name, e.g. `foo`.

`\AdviceCsname`

This macro holds the control sequence name of the advised command; it is undefined for environments. For example, when `\AdviceName` contains `\foo@bar`, this macro will hold `foo@bar`.

`\AdviceReplaced`

This macro holds the code which was replaced by the outer handler. For commands, this will be the command itself, e.g. `\foo`, so `\AdviceReplaced` will equal `\AdviceName`. For an environment `foo`, `\AdviceReplaced` is set to `\begin{foo}` in LaTeX, `\foo` in plain TeX and `\startfoo` in ConTeXt.

`\AdviceOriginal`

This macro executes the original code of the advised command.

This macro is defined as `\AdviceGetOriginal{`⟨*namespace*⟩`}{`⟨*name*⟩`}`, and therefore acts as a shortcut for an explicit invocation of `\AdviceGetOriginal`. When executing the original command directly from the advice, one may safely write `\AdviceOriginal`. However, whenever `\AdviceOriginal` is embedded in code which might contain other advised commands, it should be pre-expanded, exactly once.

`\AdviceGetOriginal`{⟨*namespace*⟩}{⟨*control sequence*⟩}

This command invokes the original definition of the ⟨*control sequence*⟩ advised by the ⟨*namespace*⟩ instantiation of Advice; more precisely, the full expansion of this macro produces the (internal) control sequence holding the definition of ⟨*control sequence*⟩ in effect when this control sequence was activated in ⟨*namespace*⟩. This macro may be safely used outside the advice, even if the advised command is not activated.

For example, upon executing key `/ns/auto=\foo{...}`, `\AdviceGetOriginal{/ns}{\foo}` will recall the original definition of `\foo` if `\foo` is activated, and simply execute `\foo` otherwise.

The second argument of this command should *not* be an environment name. To execute the original environment `foo` in plain TeX or ConTeXt, use `\AdviceGetOriginal` with the appropriate macro: `\AdviceGetOriginal{/ns}{\foo}` or `\AdviceGetOriginal{/ns}{\startfoo}`. In LaTeX, one should use `\AdviceGetOriginal{/ns}{\begin}{foo}`, which executes the original `\begin` and provides it with the environment name.

Within the advice, you will probably never have to use this command directly, but will rather rely on the (plain or pre-expanded) `\AdviceOriginal`. However, outside the advice, this command provides the only official means to access the original definition of an advised command. (Unlike the commands described above, this command is available throughout the document.)

> A typo in the invocation of this command may result in an infinite loop. Assume that the advice for `\foo`, declared in namespace `/mmz`, executes `\AdviceGetOriginal{/zzm}{\foo}`, which incorrectly refers to the non-existing namespace `/zzm`, and that command `\foo` is activated. Executing `\foo` will eventually execute `\AdviceGetOriginal{/zzm}{\foo}`, which won't find the original definition of `\foo` in the non-existing namespace `/zzm` and will thus execute macro `\foo` (again), which, being advised, will lead to another `\AdviceGetOriginal{/zzm}{\foo}`, etc. My advice is to define an abbreviation like `\def\mmzAdviceGetOriginal{\AdviceGetOriginal{/mmz}}`. And note that the namespace is a full keypath, which begins with a slash (`/`), but has no slash at the end.

`\AdviceCsnameGetOriginal`{⟨*namespace*⟩}{⟨*control sequence name*⟩}

This is a version of `\AdviceGetOriginal` which accepts a control sequence name as the second argument.

This macro is used by auto-key `to context` to include the meaning of any command (even internal commands containing `@`, or `expl3` commands) into the context.

Support for specific packages   At the moment, Advice only implements specific support for TikZ, by defining a `collector` for command `\tikz`.

`\AdviceCollectTikZArguments`

This command collects the arguments in the format expected by `\tikz`, and executes macro `\AdviceInnerHandler` with the collected arguments given as a single braced argument. The collector supports both the group and the semicolor invocation of `\tikz`, i.e. both `\tikz{...}` and `\tikz...;`.

This command is only available upon `\input`ting file `advice-tikz.code.tex`.

command/environment	handler	notes
\begin	custom	Only in LATEX; declared by Advice.
\errmessage	abort	Not available in LuaTEX, where better error-detection is implemented.
forest	memoize	
\Forest	memoize	
\index	replicate	The argument is expanded prior to replication.
\label	custom	run if memoizing; globally appends \mmzLabel{⟨*label key*⟩}{⟨*current label*⟩} to register \mmzCCMemo.
\pageref	ref	
\pdfsavepos	abort	Not available in LuaTEX.
\pgfsys@getposition	abort	Available only in Ti*k*Z is loaded, it aborts memoization of a picture which gets accidentally marked as "remembered".
\ref	ref	
\savepos	abort	Available only in LuaTEX.
\tikz	memoize	
tikzpicture	memoize	

Table 1: Commands advised by Memoize

5.6.2 Memoization-related additions to the advising framework

In section 5.6.1, we have seen that Memoize installs the advising framework into keypath /mmz, with setup key name auto. This populates keypaths /mmz and /mmz/auto with various generic advice keys. However, Memoize installs further advice/automemoization-related keys into these keypaths. It is these keys which are described in this section.[77]

Therefore, in contrast to section 5.6.1, /mmz and auto have no secret generic meaning here, i.e. they should *not* be generalized to ⟨*namespace*⟩ and ⟨*setup key*⟩ of .install advice.

Keys residing in /mmz

/mmz/**manual**=true|false   (preamble-only, default true, initially false)

When this conditional is set to true, no commands are activated at the beginning of the document. The list of commands and environments advised and activated out of the box can be found in Table 1.

The auto-framework allows the activation to be deferred (see activation and .install advice) but leaves it open to the specific instance of the framework to use the deferred activation commands as it sees fit. Normally, Memoize switches to immediate activation at the end of the preamble (hook begindocument/before) and issues activate deferred at the beginning of the document, more precisely in hook begindocument/end (afterwards, activate deferred is emptied). However, when manual is in effect, the deferred activation is suppressed (though it may be still carried out by the user by executing activate deferred).

In LATEX, manual affects the (internal) activation of \begin as well, which effectively deactivates handling of all environments.

/mmz/**ignore spaces**=true|false   (default true, initially false)

Ignore any spaces following *auto*memoized code. This key has no effect for manual memoization, i.e. command \mmz and environment memoize.

It is common practice to conclude the definition of a command by TEX primitive \ignorespaces, which consumes any following spaces, to prevent unintended blank space after the command's

---

[77]One such key, integrated driver, is actually documented in section 5.3.

invocation. Automemoizing such a command disrupts this behaviour.[78] The workaround is to use this key, normally as an option in the `auto` declaration; it will work both for automemoized macros and environments.

Keys residing in `/mmz/auto`

`/mmz/auto/`**`memoize`** (style)

This key sets up advice which triggers memoization of the command or environment whenever it is encountered; we often refer to such a command as "automemoized," or say that it was "submitted to automemoization."

An automemoized command will consume the next-options, whether memoization actually occurs or not.

Under the hood, this key declares both an `outer handler` and an `inner handler`. The outer handler opens the memoization group (so that options can be applied locally), applies the auto-options (given by `options` within `auto`) and the next-options (given by `\mmznext`) by executing `apply options`, and appends the verbatim keys to `collector options` if necessary. The inner handler invokes `\Memoize` (which closes the group opened by the outer handler): the first argument is `\AdviceReplaced`, expanded once and followed by the arguments of the handled command; the second argument is `\AdviceOriginal`, also expanded once and followed by the arguments of the handled command. The inner handler also makes sure that `ignore spaces` is respected.

`/mmz/auto/`**`nomemoize`** (style)

This key installs advice which disables memoization for the space of the command or environment; we sometimes refer to such commands as "autodisabled."

This key is merely an abbreviation for `noop, options=disable`. See the documentation of `noop` for further details.

`/mmz/auto/`**`noop`** (style)

This key sets up advice which does nothing.

Ok, not nothing at all. The installed handler applies the auto-options and the next-options by executing `apply options`, and makes sure that `verbatim` and `ignore spaces` are respected.

For commands and non-LaTeX environments, this key declares the same outer handler as `memoize`, while the inner handler merely executes the original command (respecting the potential verbatim mode), closes the group opened by the outer handler, and makes sure that `ignore spaces` is respected.

For LaTeX environments, which open the group necessary for the local application of options themselves, this key declares an outer handler which adds the relevant code into the next hook `env/`⟨*environment name*⟩`/begin`. There is no need to open a group, collect the environment body, or make special provisions for the verbatim mode.

`/mmz/auto/`**`apply options`** (style)

This style, used by `memoize`, `nomemoize` and `noop` described above, installs two handlers:

- an outer handler which opens a group, applies auto-options and next-options by executing `\mmzAutoInit`, and executes the collector; and
- a bailout handler which clears the next-options.

---

[78]It is clear that `\ignorespaces` is disrupted during utilization; in this case, the original command, including the concluding `\ignorespaces`, is never even executed. However, the disruption also occurs during memoization, and even during regular compilation. In both cases, the memoized code is embedded in some internal Memoize code. Therefore, the original `\ignorespaces` does not occur directly in front of the rest of the document.

\mmzAutoInit

This macro applies the auto-options and the next-options.

Additionally, if `verbatim`, `verb` or `no verbatim` was previously executed, this style appends the corresponding CollArgs key (`verbatim`, `verb` or `no verbatim`) to `\AdviceRawCollectorOptions`. In case several of the verbatim keys were executed, the final one takes effect.

`/mmz/auto/`**abort** (style)

This key sets up advice which aborts any ongoing memoization.

Under the hood, the advice merely executes `\mmzAbort` followed by `\AdviceOriginal`. The advised command does *not* consume the next-options. Out of the box, we submit two control sequences to this handler:

- \errmessage: this allows us to detect and abort upon at least some errors.
- \pdfsavepos (in LuaTeX, \savepos): one common effect is that memoization of any TikZ picture with `remember picture` set is aborted.

`/mmz/auto/`**unmemoizable** (style)

This key sets up advice which aborts the ongoing memoization and marks the automemoized code as unmemoizable, so that it will be henceforth compiled regularly.

Under the hood, the advice merely executes `\mmzUnmemoizable` followed by `\AdviceOriginal`. The advised command does *not* consume the next-options.

Out of the box, we submit no control sequences to this advice, but it might make sense to submit `\pdfsavepos`/`\savepos`. Keys `abort=\savepos` and `unmemoizable=\savepos` will most often have the same effect, as far as the author is concerned; the former was chosen as the default because it does not produce a c-memo; see `\mmzUnmemoizable` for a situation where `unmemoizable` is preferred.

`/mmz/auto/`**ref** (style)
`/mmz/auto/`**force ref** (style)

These keys set up advice which adds the reference key to the context expression. They are intended to be used with cross-referencing commands such as `\ref` and `\pageref`.

Indeed, `\ref` and `\pageref` are submitted to this advice by Memoize, with the effect that standard cross-referencing inside memoized code "just works." Note that the stabilization of the document after changing the reference takes three compilation cycles, i.e. one cycle more than without memoization.

The advice set up by `ref` aborts memoization if the reference key is undefined, the rationale being that the produced memo and extern would most often be useless, and could even obscure an undefined reference. The `force ref` handler produces the memo and the extern even when the reference is undefined.

The reference produced by the advised command should be fully expandable (because it will be expanded as a part of the context expression).

Typically, a `\ref` command takes a single argument, the reference key. However, some packages may define a reference command which takes optional arguments, as well; in particular, the `hyperref`'s incarnation of `\ref` takes an optional star. This advice does not care: it will accept any number of any kind of optional arguments, as long as the reference key is the first braced argument following the advised command; for example, `\ref*{key}`, `\ref[opt]{key}`, `\ref*[opt]{key}` etc. will all be handled correctly, while `\ref{mand}{key}` will not work. Effectively, it is as if we had set `args`=lm — and with the same downside, namely that an unlikely unbraced single-token reference key, like `\ref k`, will not work.

Under the hood, these two pieces of advice pass the reference key to macros `\mmzNoRef` and `\mmzForceNoRef`, and it is these commands — which may also be used in user-defined advice or the document itself — which actually add the reference key to the context expression.

/mmz/auto/**refrange** (style)
/mmz/auto/**force refrange** (style)

These keys have the same function as `ref` and `force ref`, but they operate on reference-range commands, such as `cleveref`'s `\crefrange`, which take two arguments (the starting and the ending reference key).

/mmz/auto/**multiref** (style)
/mmz/auto/**force multiref** (style)

These keys have the same function as `ref` and `force ref`, but they operate on "multireference" commands, such as `cleveref`'s `\cref`, which allow the author to list several comma-separated reference keys in a single argument.

/mmz/auto/**to context** (style)

This key sets up advice which appends the original meaning of the advised command to the context. It invokes `run if memoizing`, so that the command is only advised during memoization.

It is safe to apply this key to internal commands such as commands whose name contains `@`, or `expl3` commands. It is not necessary to provide the argument structure of the advised command (using `args`).

/mmz/auto/**replicate** (style)

This key sets up advice which replicates the invocation of the command in the cc-memo during memoization.

When using this key, it is necessary to set `args` as well. For `\index`, Memoize executes `auto=\index {args=m, replicate}`.

This key takes an auto-option, **expanded**.[79] If given, the collected arguments will be expanded before replicating them in the cc-memo; in LaTeX, this expansion is `\protect`ed.

In LaTeX, Memoize submits `\index` to this handler (with expansion). Therefore, any `\index{key}` in the memoized code gets copied into the cc-memo. Effectively, indexing from within the memoized code "just works."

Note that `\label`, despite essentially requiring replication, cannot use this advice, because it needs to replicate not only the label key but `\@currentlabel` as well.

/mmz/auto/**run if memoization is possible** (style)

Under the run conditions installed by this key, a command is only advised if Memoize is enabled but we're not already "within Memoize," i.e. memoizing or normally compiling some code submitted to memoization. In code: `\ifmemoize\ifinmemoize\else\AdviceRuntrue\fi\fi`.

Internally, this key is used by `memoize` and `noop`.

/mmz/auto/**run if memoizing** (style)

Under the run conditions installed by this key, a command is only advised during memoization. In code: `\ifmemoize\ifmemoizing\AdviceRuntrue\fi\fi`.

Internally, this key is used by `abort`, `replicate`, and `ref` and friends.

---

[79]This option is unrelated to Memoize's options, settable by `\mmzset`.

\CollectArguments[⟨*options*⟩]{⟨*argument specification*⟩}{⟨*next-code*⟩}⟨*tokens*⟩

This command determines the extent to which the ⟨*tokens*⟩ following the the three formal arguments of the command conform to the given ⟨*argument specification*⟩, effectively splitting ⟨*tokens*⟩ into ⟨*argument tokens*⟩ and the ⟨*rest*⟩ of the tokens, and then executes ⟨*next-code*⟩ with the ⟨*argument tokens*⟩ provided as a single, braced argument:

$$⟨next\text{-}code⟩\{⟨argument\ tokens⟩\}⟨rest⟩$$

If the initial part of ⟨*tokens*⟩ does not conform to ⟨*argument specification*⟩, **\CollectArguments** throws an error. (In this case, ⟨*next-code*⟩ is not executed, and the ⟨*tokens*⟩ collected until the error are thrown away.)

The optional ⟨*options*⟩ are processed using the **pgfkeys** utility of PGF/Ti*k*Z (see §87 of the Ti*k*Z & PGF manual), with the default path set to **/collargs**. The given options apply to all the arguments in ⟨*argument specification*⟩. The recognized keys are listed in the rest of the section.

\CollectArguments expects the ⟨*argument specification*⟩ to conform to the LaTeX3 argument specification format, i.e. it accepts the same kind of ⟨*argument specification*⟩ as **\NewDocumentCommand** and friends, complete with extensions available upon loading package **xparse**.[80] Furthermore, CollArgs introduces several extensions to the LaTeX3/**xparse** argument specification format:[81]

- We introduce modifier **&** taking a mandatory argument specifying the options to apply to the following argument in the specification. Options given here override the ⟨*options*⟩ given as the optional argument.

- The environment body type **b** may be followed by an optional *braced* argument providing the name of the environment to collect. The name given here overrides the name given by the **environment** option.

- The number of collected "arguments" is unlimited.

For convenience, the entire LaTeX3 argument specification format, complete with **xparse** and CollArgs extensions, is summarized in the frame on the following page. For the official documentation of the format, consult the following sources:

- The LaTeX3 version of "LaTeX for authors" documents the standard argument types.

- The **xparse** manual additionally documents the uncommon/historical types only available upon loading package **xparse**.

Note that in CollArgs, **O**{⟨*default*⟩} has the same effect as **o**, and similarly for other pairs of types with and without defaults (**R** and **r**, **D** and **d**, and **E** and **e**). CollArgs is dedicated to collecting the argument tokens precisely as they are given: if an optional argument is missing, its default value is *not* inserted among the collected arguments — consequently, **\CollectArguments** is utterly uninterested in the default value.

Collection of environments automatically adapts to the format, i.e. given environment body name **foo**, **\CollectArguments** knows to search for **\begin{foo}** ... **\end{foo}** in LaTeX, **\foo** ... **\endfoo** in plain TeX, and **\startfoo** ... **\stopfoo** in ConTeXt. For further information on environment collection, see keys **ignore nesting** and **tags**.

\CollectArgumentsRaw{⟨*option-setting code*⟩}{⟨*argument specification*⟩}{⟨*next-code*⟩}⟨*tokens*⟩

This command is the programmer's interface to CollArgs, intended to be used instead of **\CollectArguments** when compilation speed is an issue. The two commands only differ in how they deal with options.

---

[80]Note that CollArgs collects the arguments by its own means. It does *not* require either LaTeX3 or **xparse** to do its job. Specifically, it may be used in plain TeX and ConTeXt as well.

[81]Collargs internally uses a dot (.) to delimit the argument specification from the following argument tokens. Therefore, the dot really counts as an extra argument type, in the sense that CollArgs will stop working if the dot becomes an argument type or a modifier in some future release of LaTeX.

Mandatory argument types

m	standard (a single token or multiple tokens in braces)
r⟨*token₁*⟩⟨*token₂*⟩	delimited by ⟨*token₁*⟩ and ⟨*token₂*⟩
R⟨*token₁*⟩⟨*token₂*⟩{⟨*default*⟩}[a]	delimited by ⟨*token₁*⟩ and ⟨*token₂*⟩, with a default value
v	verbatim, in the style of \verb[b]
b	the body of an environment

Optional argument types

o	in square brackets
O{⟨*default*⟩}[a]	in square brackets, with a default value
d⟨*token₁*⟩⟨*token₂*⟩	delimited by ⟨*token₁*⟩ and ⟨*token₂*⟩
D⟨*token₁*⟩⟨*token₂*⟩{⟨*default*⟩}[a]	delimited by ⟨*token₁*⟩ and ⟨*token₂*⟩, with a default value
s	an optional star
t⟨*token*⟩	an optional ⟨*token*⟩
e{⟨*tokens*⟩}	a set of embellishments
E{⟨*tokens*⟩}{⟨*defaults*⟩}[a]	a set of embellishments, with default values

Weird argument types

l[c]	a mandatory argument until the first begin-group token
u{⟨*tokens*⟩}[c]	TeX's delimited argument
g[c]	an optional argument inside braces

Modifiers

+	allow the next argument to be long
!	disallow spaces before arguments of type d and t
>{⟨*processor*⟩}	process the next argument

CollArgs extensions

b{⟨*name*⟩}	the body of an environment with the given ⟨*name*⟩
&{⟨*options*⟩}	apply CollArgs options to the next argument
&&{⟨*raw options*⟩}	apply raw CollArgs options to the next argument

---

[a]As `\CollectArguments` does not use the arguments but only collects them, it does not care about the default values of optional arguments. Therefore, while CollArgs recognizes argument types with defaults (O, D, R, and E), they may be safely substituted by their -NoValue- counterparts (o, d, r and e).

[b]`\CollectArguments` can grab an argument of any type in the verbatim mode.

[c]Argument types l, u and g are not defined by the core LaTeX3; they are only available upon loading package xparse. As noted in the main text, CollArgs does not make any difference between standard LaTeX3 types and the xparse extensions; it supports them all at all times, and even in plain TeX and ConTeXt.

One difference is that for `\CollectArgumentsRaw`, the options form a mandatory rather than an optional argument. More importantly, however, they do not take the form of a keylist, but should be composed out of low-level option-setting commands. Each key documented in this section has a corresponding low-level macro; these macros are listed in footnotes alongside the keys. The name of the macro starts with `\collargs` and continues with the name of the key, without spaces, each word capitalized; if the key is boolean, this convention applies to the base of the TeX conditional. For example,

```
\CollectArguments[caller=\foo, tags, verbatim]{⟨argument specification⟩}{⟨next-code⟩}
```

is equivalent to

```
\CollectArgumentsRaw{%
  \collargsCaller{\foo}%
  \collargsBeginTagtrue\collargsEndTagtrue
  \collargsVerbatim
}{⟨argument specification⟩}{⟨next-code⟩}
```

Withing the option-setting code, the programmer may also deploy macro `\collargsSet`, which

processes the ⟨*options*⟩ in the keylist format. One idea could be to execute this macro at the end of the low-level options; this would set the "defaults" using the fast programmer's interface, but still allow for user customization.

\collargsSet⟨*options*⟩

Locally update the CollArgs configuration.

The ⟨*options*⟩ are a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs. They are processed using the `pgfkeys` utility of PGF/Ti*k*Z (see §87 of the Ti*k*Z & PGF manual), with the default path set to **/collargs**.

This key may be used either outside an invocation of \CollectArguments/\CollectArgumentsRaw, to set defaults or define additional styles, or within the ⟨*option-setting code*⟩ in the invocation of \CollectArgumentsRaw.

/collargs/**alias**={⟨*symbol*⟩}{⟨*definition*⟩}   (style)[82]

Define ⟨*symbol*⟩ as an abbreviation of {⟨*definition*⟩} within the ⟨*argument specification*⟩ passed to \CollectArguments.

For example, Memoize's `biblatex` support sets alias={*}{&&{\mmz@biblatex@mark@citation@key}}, with \mmz@biblatex@mark@citation@key an internal macro which stores the contents of \collargsArg into another internal macro.

/collargs/**caller**=⟨*control sequence (name)*⟩   (no default, initially \CollectArguments)[83]

Set the control sequence to refer to in error messages.

If ⟨*tokens*⟩ do not match the ⟨*argument specification*⟩, \CollectArguments throws an error. By default, the error message contains a reference to \CollectArguments itself, for example ! Argument of \CollectArguments has an extra }. However, this might not be very informative to the author. When caller=\cs is in effect, the error messages will refer to the given \cs instead.

If the value of this key is not a control sequence, it is assumed to be an environment name, but as the caller must be a macro, this name will be converted into a control sequence. Setting caller=foo will result in error messages referencing \foo in plain TEX, \startfoo in ConTEXt and \begin{foo} (a single control sequence!) in LATEX.

/collargs/**environment**=⟨*environment name*⟩   (applicable to type b, no default, initially empty)[84]

Set the name of the environment collected by argument type b.

---

[82]The programmer's interface: **\collargsAlias**.
[83]The programmer's interface: **\collargsCaller**.
[84]The programmer's interface: **\collargsEnvironment**.

/collargs/**begin tag**=true|false   (applicable to type b, default `true`, initially `false`)[85]
/collargs/**end tag**=true|false   (applicable to type b, default `true`, initially `false`)
/collargs/**tags**=⟨*boolean*⟩   (applicable to type b, style, default `true`)

When `begin tag`/`end tag` is in effect, the begin/end tag will be will be prepended/appended to the collected environment body. Style `tags` is a shortcut for setting `begin tag` and `end tag` simultaneously.

In LaTeX, using `tags` will thus dress up the collected body in a pair or \begin{⟨*environment name*⟩} and \end{⟨*environment name*⟩}. CollArgs will automatically use the tags appropriate to the format.

In the verbatim modes, the added tags are verbatim as well, with the detail that in LaTeX, there is a slight difference between the full `verbatim` and the partial `verb` mode. In the full verbatim mode, the braces surrounding the environment name are verbatim (the characters used as braces are actually determined by key `braces`). In the partial verbatim, as well as the non-verbatim mode, the environment name is surrounded by a pair of actual braces of category 1 and 2, regardless of which characters are of these categories in the calling code.

/collargs/**ignore nesting**=true|false   (applicable to type b, default `true`, initially `false`)[86]

When this key is *not* in effect, CollArgs respects the hierarchical structure created by tag pairs such as \begin{foo} and \end{foo}. Given the situation below on the left, argument type b{foo} will collect everything up until the *second* \end{foo}. Now this is what we usually want, because LaTeX keeps track of environment embedding as well. However, all *verbatim* environments I know of, starting with the standard LaTeX `verbatim`, will ignore the nesting and simply scoop up everything up to the first \end{verbatim}. In CollArgs, we can replicate their behaviour by setting `ignore nesting`, as shown below on the right. (Of course we also need to set `verbatim` if we want to grab the environment body in the verbatim mode.)

```
ignore nesting=false

    ...
    \begin{foo}
      ...
    \end{foo}
    ...
  \end{foo}
```

```
ignore nesting=true

      ...
      \begin{verbatim}
      ...
    \end{verbatim}
    ...
  \end{verbatim}
```
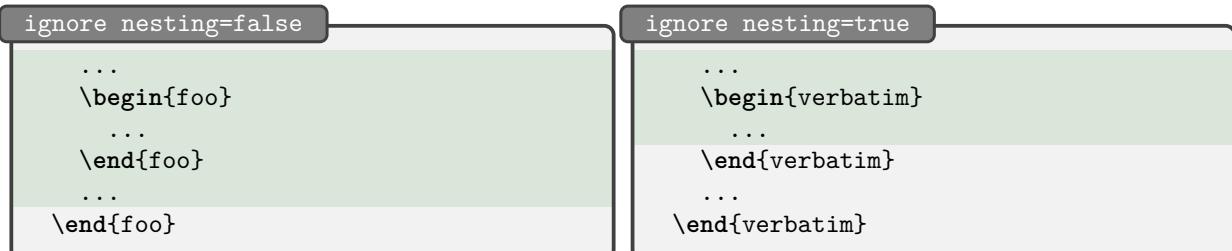
This key applies not only to argument type b (in either normal or verbatim mode), but also to the verbatim argument type v and to argument types m and g in the `verbatim` (but not normal, or `verb`) mode. With these keys, the relevant structure markers are braces, { and }.

/collargs/**ignore other tags**=true|false   (applicable to type b, default `true`, initially `false`)[87]

In LaTeX, the environment tags, \begin{⟨*name*⟩} and \end{⟨*name*⟩}, contain braces, which retain their usual category codes in the non-verbatim and in the partial verbatim mode. Consequently, CollArgs cannot easily search for the full tags to delimit the environment.

When this key is *not* in effect, CollArgs takes the easy path, and determines the end of the environment only by inspection of \begins and \ends, without reference to what ⟨*name*⟩ they begin or end. Only when this key *is* in effect does CollArgs inspect these ⟨*name*⟩s, effectively ignoring any \begins and \ends not followed by the name of environment being collected. The effect of absence vs. presence of this key is shown below, where the shaded area marks the code collected into environment `foo`.

---

[85]The programmer's interface: **\ifcollargsBeginTag**, **\ifcollargsEndTag**; `tags` has no corresponding low-level command.

[86]The programmer's interface: **\ifcollargsIgnoreNesting**.

[87]The programmer's interface: **\ifcollargsIgnoreOtherTags**.

ignore other tags=false	ignore other tags=true

```
\CollectArguments
            {\NextCommand}{b{foo}}
    ...
    \end{bar}
    ...
    \end{foo}
```

```
\CollectArguments[ignore other tags]
            {\NextCommand}{b{foo}}
    ...
    \end{bar}
    ...
    \end{foo}
```
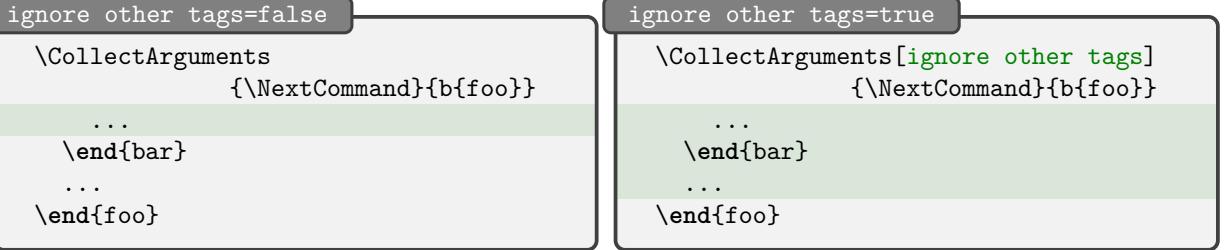
This key does not have any effect the full verbatim mode, which always behaves as if this key was set to true, because braces are of category "other" as well. Similarly, it is as if this key was always true in plain TEX and ConTEXt, simply because environment tags in these formats don't contain braces.

/collargs/**append preprocessor**=⟨*code*⟩	(style, no default)[88]
/collargs/**prepend preprocessor**=⟨*code*⟩	(style, no default)
/collargs/**append postprocessor**=⟨*code*⟩	(style, no default)
/collargs/**prepend postprocessor**=⟨*code*⟩	(style, no default)

These keys declare processors which will transform the collected argument before appending it to the argument list.

A processor should be a macro taking no arguments, (locally) transforming the contents of token register **\collargsArg** as it sees fit.

A collected argument undergoes the following transformations:

- First, the argument is processed by any *pre*processors, in the order indicated by `append` and `prepend`.

- Next, the processed argument is dressed up in the delimiters according to its type. For example, an optional argument of type `o` will be surrounded by square brackets.

- Finally, the delimited argument is processed by any *post*processors, again in the order indicated by `append` and `prepend`.

The following example illustrates how one could go about reimplementing Bruno Le Floch ingenious package `cprotect`,[89] which allows the author to use verbatim text in arguments. Below, we will end up with with \footnote{⟨*footnote text*⟩} replaced by \footnote{\input _fn.tex}, with ⟨*footnote text*⟩ waiting in file _fn.tex. To achieve this, we define processor \writetofile which dumps the argument into a file, replacing it with the \input statement. (Of course, to allow for verbatim content in the footnote, we also have to mark the argument as `verbatim`. And we use `no delimiters` to get rid of the braces around the footnote text.)

---

[88]The programmer's interface: **\collargsAppendPreprocessor**, **\collargsPrependPreprocessor**, **\collargsAppendPostprocessor**, **\collargsPrependPostprocessor**.

[89]This example is merely a proof of concept. For the bells and whistles which would make it useful in real life, see the documentation of `cprotect`.

```
    \newwrite\argfile
    \newcommand\writetofile[1]{%
      \immediate\openout\argfile{#1}%
      \newlinechar=13
      \immediate\write\argfile{\the\collargsArg}%
      \immediate\closeout\argfile
      \collargsArg={\input{#1}}%
    }
    We write the argument of \verb!\footnote! into a file,%
    \CollectArguments{
      &{verbatim, append preprocessor=\writetofile{_fn.tex}, no delimiters}
      m
    }{\footnote}{This footnote was read from a file by command \verb!\input!,
      so it may contain verbatim material!} and then read it back in.
```

We write the argument of **\footnote** into a file,[a] and then read it back in.

---

[a]This footnote was read from a file by command **\input**, so it may contain verbatim material!

/collargs/**clear preprocessors**
/collargs/**clear postprocessors** (style)

Clear the list of pre- or post-processors.

/collargs/**append expandable preprocessor**=⟨*code*⟩ (style, no default)[91]
/collargs/**prepend expandable preprocessor**=⟨*code*⟩ (style, no default)
/collargs/**append expandable postprocessor**=⟨*code*⟩ (style, no default)
/collargs/**prepend expandable postprocessor**=⟨*code*⟩ (style, no default)

These keys may be used to simplify the definition of processors which can arrive at the result by means of full expansion. A processor declared by one of these keys assigns the full expansion of ⟨*code*⟩ to \collargsArg, i.e. it is defined as \collargsArg\expandafter{\expanded{⟨*code*⟩}}.

A processor added with one of these keys will end up among the processors declared by append preprocessor et al.

⟨*code*⟩ will typically contain \the\collargsArg to refer to the input state of the argument. For example, to add quotes around the collected argument: append expandable preprocessor= ``\the\collargsArg''. Or, to trim spaces from an argument using \trim@spaces@noexp from package trimspaces: append expandable preprocessor=\expandafter\trim@spaces@noexp\expandafter {\the\collargsArg}. A pre-processor is the right choice in both these cases because we want to modify the argument before it is dressed up in the original delimiters.

Let us consider another, more involved example. As the example for append preprocessor above, it is inspired by cprotect, but instead of writing the argument into a file and then reading it back, we rescan it using \scantokens. Remarks: (a) rescanning only happens when processing \fbox; append expandable preprocessor merely inserts \ScanMultilineTokens in front of the argument. (b) \ScanMultilineTokens below temporarily changes the \newlinechar to 13 to allow for linebreaks in (the source code of) the argument; (c) \endinput is added to prevent the extra end-of-file space creeping in; (d) we have used verb instead of verbatim so that the argument is eventually dressed up in a pair of real, category code 1 and 2, braces.

---

[90]The programmer's interface: **\collargsClearPreprocessors**, **\collargsClearPostprocessors**.
[91]The programmer's interface: **\collargsAppendExpandablePreprocessor**, **\collargsPrependExpandablePreprocessor**, **\collargsAppendExpandablePostprocessor**, **\collargsPrependExpandablePostprocessor**.

```
\long\def\ScanMultilineTokens#1{%
  \expanded{%
    \newlinechar=13
    \unexpanded{\scantokens{#1\endinput}}%
    \newlinechar=\the\newlinechar
  }%
}
\newcommand\CollectAndRescan[1]{%
  \CollectArguments[
    verb,
    append expandable
  preprocessor=\noexpand\ScanMultilineTokens{\the\collargsArg},
  ]{m}#1%
}
Remember that \CollectAndRescan\fbox{command \verb!\foo! is important.} Really!
```

Remember that   command \foo is important.   Really!

/collargs/**no delimiters**=true|false   (default `true`, initially `false`)[92]

When this key is in effect, the collected argument will not be dressed up into delimiters that
it was dressed up in ⟨*argument tokens*⟩. For example, an optional argument, encountered as
[⟨*argument*⟩], will be spit out simply as ⟨*argument*⟩.

Any user-specified pre- or post-processing will still be applied (and usually is, when this key is
used).

```
\texttt{no delimiters=false}:
\CollectArguments{o m}\ShowArguments[optional]{mandatory},
\texttt{no delimiters=true}:
\CollectArguments[no delimiters]{o m}\ShowArguments[optional]{mandatory},
```

no delimiters=false: [optional]{mandatory}, no delimiters=true: optionalmandatory,

/collargs/**return**=**braced**|**plain**|**no**   (initially `braced`, no default)[93]

This key determines the form in which the collected arguments are passed on to ⟨*next-code*⟩:

- When set to `braced` (the default), ⟨*next-code*⟩ receives the collected arguments as a single,
  braced argument.

- When set to `plain`, the collected arguments are simply inserted behind ⟨*next-code*⟩. Combined
  with other keys, this keys makes it possible to transform the collected arguments into form
  expected by ⟨*next-code*⟩. For example, `&{...} b` below transforms the body of an environment
  into a mandatory argument (after trimming the surrounding spaces); the preceding optional
  argument remains unmodified. Without `return=plain`, \processbody would receive a single
  argument, containing both the optional argument and the transformed environment body;
  with `return=plain`, which makes CollArgs refrain from bracing the collected arguments, it
  receives an optional plus a mandatory argument.[94]

---

[92]The programmer's interface: **\ifcollargsNoDelimiters**.
[93]The programmer's interface: **\collargsReturn**.
[94]For further examples, see the definition of **\mmz@remove@quotes** and advising of **\blx@bbl@entry** in the Memoize
source.

```
collargs-return-plain.tex

  \collargsSet{
    brace/.style={
      no delimiters,
      append expandable postprocessor=\empty{\the\collargsArg},
    },
    trim spaces/.style={
      append expandable preprocessor=
      \expandafter\trim@spaces@noexp\expandafter{\the\collargsArg},
    },
  }
  \newcommand\processbody[2][default]{optional=`#1', body=`#2'}
  \CollectArguments
    [return=plain, environment=foo]
    {o &{trim spaces, brace} b}
    \processbody
    [optional] % may be omitted
    some text
  \end{foo}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  optional='optional', body='some text'
```

- When set to `no`, the collected arguments are not inserted into the input stream; ⟨*next-code*⟩ can access them through `\collargsArgs`, a token register dedicated to holding the collected arguments (regardless of the value of `return`).

  In conjunction with `clear args`=`false`, this setup can be useful for reiterating command `\CollectArguments`.[95] Command `\CollectOptionalArguments` defined below collects any number of optional arguments given either in square brackets or in parentheses. To achieve this, we instruct CollArgs to neither clear `\collargsArgs` at the start of argument collection, nor insert it into the input stream once the collection is finished; we thus have to clear this token register manually at the start, and use it directly at the end. (There's a number of ways of detecting whether further arguments follow. We use a pure Collargs solution below: we define a postprocessor which sets an internal conditional to true if any argument was found, and test for that conditional in the ⟨*next-code*⟩.)

---

[95]The idea illustrated with this example is deployed in the definition of `cites` in the Memoize source.

```
collargs-return-no.tex                                                    📎
  \newcommand\CollectOptionalArguments{%
    \global\collargsArgs{}%
    \CollectOptionalArgument
  }
  \newbool{CollectedOptionalArgument}
  \newcommand\CollectOptionalArgument{%
    \global\boolfalse{CollectedOptionalArgument}%
    \CollectArguments
      [clear args=false, return=no,
        append postprocessor=\global\booltrue{CollectedOptionalArgument}]
      {o d()}
      {%
        \ifbool{CollectedOptionalArgument}%
          \CollectOptionalArgument
          \FinishCollectOptionalArguments
      }%
  }
  \newcommand\FinishCollectOptionalArguments{%
    The collected arguments: \texttt{``\the\collargsArgs''}.
  }
  \CollectOptionalArguments[opt1][opt2](opt3)[opt4] Some text.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  The collected arguments: ``[opt1][opt2](opt3)[opt4]''.  Some text.
```

\collargsArgs   (token register, global)

This token register holds the collected arguments upon the exit from \CollectArguments; the token register is assigned to globally.

This register may be manipulated by the user, both within and outside an invocation of \CollectArguments; all assignments should be global.

/collargs/clear args=true|false   (default true, initially true)[96]

When this conditional is set to false, token register \collargsArgs is not cleared upon entering \CollectArguments; see return=no for an example.

/collargs/verbatim   (style)[97]
/collargs/verb   (style)
/collargs/no verbatim   (style, the initial mode)

Select the full verbatim, the partial verbatim, or the non-verbatim mode of argument collection.

In the full verbatim mode, the arguments are collected under a category code regime in which all characters are of category 12, "other". The same goes for the partial verb mode, except that in this case, the grouping characters — usually the braces { and } — retain their usual category codes 1 and 2. Key no verbatim selects the normal, non-verbatim mode.

The partial verb mode can be useful for verbatim collection of an optional argument. To pass] as an optional argument to command \foo, we normally enclose it in braces: \foo[{]}]. However, if we try to collect [{]}] with \CollectArguments[verbatim]{o}, we will get { (and most likely an error, as well), because in the verbatim mode, braces do not have their grouping function. Using the verb mode solves the problem: occuring within braces, the first] is "invisible" to \CollectArguments[verb], so the optional argument is correctly recognized as ending at the second].

The partial verb mode is also useful for collecting the bodies of LaTeX environments. The full verbatim mode will only correctly collect these bodies when the relevant \begin and/or \end

---

[96]The programmer's interface: \ifcollargsClearArgs.
[97]The programmer's interface: \collargsVerbatim, \collargsVerb, \collargsNoVerbatim. To ensure the same effect as with the keys, place these macros at the end of the option code.

control sequences are followed by the grouped environment name without any intervening spaces. The partial `verb` mode has no such restriction.

In the verbatim modes, modifier + has no effect. The arguments are always collected as if they were long.

To correctly collect arguments in the verbatim modes, CollArgs has to mimic the many details of TeX's tokenization and argument delineation. These details depend on the category code regime, and CollArgs automatically adapts to the "outside" category code regime, i.e. the regime in effect at the time of invoking `\CollectArguments`. In particular, CollArgs remembers which characters were of category codes 0, 1, 2, 5, 10 and 11, and adapts the argument collection accordingly. For example, it will correctly pick up a control sequence as a single-token m-type (TeX's undelimited) argument even when it begins with a non-standard character of category code 0. The single caveat is that only a single pair of characters can function as the grouping characters in the full verbatim mode; to compensate for the deficiency, this character pair is customizable via key `braces`.

/collargs/**fix from verbatim**   (style)[98]
/collargs/**fix from verb**   (style)
/collargs/**fix from no verbatim**   (style)

Key `fix from no verbatim` should be used when the first argument should be collected in a verbatim mode, but the outside code has already tokenized the first character of the subsequent input stream (most probably by a `\futurelet`) in the non-verbatim category code regime. Using this key will trigger a CollArgs' "mode transition" (described below) which will fix the situation. (This key is used in the implementation of `\mmz`.)

The other two keys should be used in the unlikely reverse situation, where the outside code has tokenized the following character in a verb(atim) mode, while CollArgs is requested to collect the first argument in the non-verbatim mode.

/collargs/**braces**=⟨*begin-group char*⟩⟨*end-group char*⟩   (no default, for the initial value see below)[99]

This key sets the verbatim begin-group and end-group characters. The setting affects collection of argument types m, g, v, l, e and (in LaTeX) b in the *full* verbatim mode.[100]

For example, in the non-verbatim and the partial verbatim mode, an m-type argument may be delimited by any characters of category code 1 ("begin-group") and 2 ("end-group"). In the full verbatim mode, there are of course no characters of these categories, so CollArgs internally assigns the grouping function to some pair of characters. When entering the full verbatim mode, CollArgs automatically sets the verbatim grouping characters to characters which were of categories 1 and 2 in the "outside" category code regime, i.e. the regime in effect at the time of invoking `\CollectArguments`. However, in contrast to TeX's internal argument parser, only one pair of characters may serve as the begin-group and the end-group character in CollArgs' full verbatim mode. In case multiple characters were of category 1 or 2 on the outside, CollArgs therefore has to make a choice, and it chooses the candidate with the lowest character code. This choice may be overridden by the user by invoking key `braces`; the user may even choose characters which did not belong to categories 1 and 2 in the outside regime.

When ⟨*begin-group char*⟩ and ⟨*end-group char*⟩ are of categories 1 and 2 in the outside category regime, they must be enclosed in a triple group. For example, if both () and {} have the grouping function on the outside, and the user wants to select {} as the verbatim grouping characters (CollArgs would go for (), as this pair has lower character codes), the correct way to invoke this key is `braces={{{{}}}}` or `braces=(((({})))`.[101]

---

[98]The programmer's interface: **\collargsFixFromVerbatim**, **\collargsFixFromVerb**, **\collargsFixFromNoVerbatim**.
[99]The programmer's interface: **\collargsBraces**.
[100]The choice of the verbatim grouping characters also affects the effect of `begin tag` and/or `end tag`; see the documentation of these keys for details.
[101]This complication is due to the details of `pgfkeys`' keylist processing, and does not apply to `\collargsBraces`.

/collargs/verbatim ranges={⟨*from*⟩-⟨*to*⟩(, ⟨*from*⟩-⟨*to*⟩)*}   (no default, initially `0-255`)[102]

If run under the pdfTeX or XeTeX engine, this key determines which characters will be assigned category code 12 in the verbatim mode. In pdfTeX, the range should remain at the initial `0-255`, but in XeTeX, some rare situations might require extending this range (don't attempt to set the full range of `0-1114111`, as this would be very slow and you would most likely run out of save stack).

In LuaTeX, we switch the category code regime using category code tables, so this key has another meaning: it determines the range in which CollArgs will scan for characters of category codes 1, 2 and 14, whose identity it needs to know, for internal reasons.

Mode transition limitations

`\CollectArguments` has some minor limitations regarding the transition from a verbatim into non-verbatim mode, or vice versa. The gist of the issue is best illustrated with the optional argument type `o` collected in the verbatim mode. CollArgs determines whether an argument of this type is present by peeking ahead (using TeX's `\futurelet` primitive) into the input stream. If the argument is present (i.e. if the input stream continues with an open bracket, `[`), all is well. But when the optional argument is absent, the peek-ahead will tokenize the following character, which presents a problem when no more arguments are present in the input stream, like in the example below, where the verbatim `o` is the (only and) final type in the argument specification. In this case, the peek-ahead "incorrectly" assigns category code 12 ("other") to the first `$`. This character was intended to be tokenized as the math shift character of category 3, to start the math mode after `\CollectArguments` is finished, but having been assigned category code 12, it cannot perform this function, resulting in error `! Missing $ inserted` once TeX encounters the superscript character `^`.

```
collargs-transition-ok.tex
    \CollectArguments[verbatim]{o}{\ShowArgs}$2^2=4$          Collected: ""2² = 4
```

Well — this is what *would* happen if CollArgs didn't address the transition issue described above. In fact, the above example compiles just fine, because CollArgs *does* address this issue, but unfortunately, certain transition problems simply cannot be resolved — read on to learn what can go wrong.

For example, you can typeset the name of the document author via LaTeX's internal command `\@author`, but to use this command in the document, you have to precede it by `\makeatletter`. As shown by the first line of the example below, this works rather nicely: `\makeatletter` sets the category code of `@` to 11 ("letter"), so `@` may help form the control word `\@author` — importantly, `\makeatletter` sets the category code of `@` *before* control sequence `\@author` is constructed, even if it precedes it immediately.

```
collargs-transition-cs.tex
    \makeatletter\@author\makeatother\par
    \CollectArguments[verbatim]{o}{\ShowArgs}\makeatletter\@author\makeatother
```
Sašo Živanović
Collected: ""author

In the second line of the example, our clever invocation of `\@author` is immediately preceded by a call to `\CollectArguments`, which tries to collect a verbatim argument of type `o`. It doesn't find it, which results in the wrong, verbatim tokenization of the escape character of `\makeletter`. CollArgs realizes the problem and tries to fix it. But while it is searching for the end of control sequence `\makeletter` (which it successfully constructs), it triggers the tokenization of what follows — which, as `@` is at that point of category 12 ("other"), yields the control symbol `\@` (later followed by word "author", typeset in the example).

In short, the solution has created another, delayed instance of the problem — an instance which cannot be addressed any further. But we're nevertheless better off, as this particular issue will bite only in the case when the "corrupted" control sequence immediately following the invocation `\CollectArguments` changes category codes in a way that affects the tokenization of what immediately follows it.

---

[102]The programmer's interface: `\collargsVerbatimRanges`.

This was an example of what can go wrong in the transition from the ⟨argument tokens⟩ to the ⟨rest⟩ of the tokens following an invocation of `\CollectArguments`. As the "outside world" is non-verbatim, this transition can only be problematic if the argument which corrupted the first of the ⟨rest⟩ tokens was verbatim, so if the transition was from the verbatim to the non-verbatim mode. Such a transition can also occur within the ⟨argument tokens⟩, but the good news here is that CollArgs successfully solves any problems that occur there, so you should only worry about the end-of-arguments situation.

The other direction of the transition, from the non-verbatim to the verbatim mode, however, can affect both the internal and the external transitions. Let us illustrate the problem with the internal transition. Say you want to collect an optional argument (`o`) in the non-verbatim mode, and then a mandatory argument (`m`) in (either full or partial) verbatim mode. In the first invocation of `\CollectArguments` below, the optional argument is present, and we get what we expect: the percent sign is collected, verbatim, into the mandatory argument. In the second invocation, however, the percent character retains its usual commenting function — despite the fact that we have requested verbatim mode for the mandatory argument — which results in the group in the second line being picked up as the mandatory argument. Again, this happens because CollArgs has to peek ahead in the input stream when determining whether the optional argument is present. Having requested non-verbatim mode for the optional argument, the peeking is performed in the non-verbatim mode, and as the optional argument is not present, it finds the comment character, which fulfills its regular function of disappearing along with the rest of the line. Once CollArgs sets to find the (verbatim) mandatory argument, the rest of the line is already gone, so it searches for, and finds, this argument in the next line.

```
collargs-transition-comment.tex

    \CollectArguments{o&{verbatim}m}{\ShowArgs}[opt]% comment
    {more text}

    \CollectArguments{o&{verbatim}m}{\ShowArgs}% comment
    {more text}
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Collected: "[opt]%" comment more text
    Collected: "{more text}"
```

Nothing can be done here — commenting deletes information, irrevocably — and in a similar fashion, nothing can be done to catch a verbatim end-of-line character when preceded by an absent optional argument in the non-verbatim mode (because it was already tokenized into a space token).

Finally, the transition issues are not limited to transits from argument type `o`. The full list of argument types which give rise to transition problems (when transiting *from* arguments of these types) is as follows: `o`, `d`, `s`, `t`, `g`, `e`.

5.7 Support for specific classes and packages

5.7.1 PGF

When package PGF is loaded, commands `\mmzPgfAtBeginMemoization` and `\mmzPgfAtEndMemoization` are defined and appended to hooks `at begin memoization` and `at end memoization`, respectively.

Note that environment `pgfpicture` is *not* submitted to automemoization. Doing this would probably not be a good idea, as it is used by many other commands and environments.

PGF support
```
\mmzset{%
  at begin memoization=\mmzPgfAtBeginMemoization,
  at end memoization=\mmzPgfAtEndMemoization,
}%
```

`\mmzPgfAtBeginMemoization`
`\mmzPgfAtEndMemoization`

These commands are meant to be executed inside hooks `at begin memoization` and `at end memoization`. Together, they make sure that utilizing an extern containing PGF/TikZ pictures advances the PGF picture ID (`\pgf@picture@serial@count`) as if the pictures were compiled.

This avoids potential problems for various mechanisms which rely on a stable PGF picture ID. For example, in absence of advancing the ID, placing a TikZ picture scaled by `ext.scalepicture` of `tikz-ext` *after* some memoized TikZ picture would confuse the scaled picture, as its ID would depend on whether the extern of the previous (unrelated!) picture was memoized or utilized.

Command `\mmzPgfAtBeginMemoization`, executed in hook `at begin memoization`, remembers the PGF picture ID at the beginning of memoization. Command `\mmzPgfAtEndMemoization`, executed in hook `at end memoization`, checks whether the ID was advanced during memoization, and if so, appends to the cc-memo the code advancing the ID counter accordingly.

Note that out of the box, `\mmzPgfAtBeginMemoization` and `\mmzPgfAtEndMemoization` reside in their respective hooks at all times, not only when memoizing `tikzpicture` and `\tikz`. This way, the PGF picture ID mechanism is applied to all pictures deploying PGF pictures under the hook, e.g. `forest` trees.

5.7.2 TikZ

When package TikZ is loaded, environment `tikzpicture` and command `\tikz` and are automatically submitted to automemoization; this may be prevented by `deactivate`=`tikzpicture` and/or `deactivate`=`\tikz`, or by `manual`. Note that command `\tikz` requires a dedicated argument collector, `\AdviceCollectTikZArguments`.

TikZ support
```
\mmzset{%
  auto={tikzpicture}{memoize},
  auto=\tikz{memoize, collector=\AdviceCollectTikZArguments},
}%
```

5.7.3 Forest

When package `forest` is loaded, environment `forest` and command `\Forest` are automatically submitted to automemoization; this may be prevented by `deactivate`=forest and/or `deactivate`=\Forest, or by `manual`. Note that command `\Forest` is defined by `\NewDocumentCommand`, which makes specifying option `args` unnecessary.

```
Forest support
    \mmzset{
      auto={forest}{memoize},
      auto=\Forest{memoize},
    }%
```

5.7.4 Beamer

Memoize should be loaded by writing `\RequirePackage{memoize}` (rather than `\usepackage{memoize}`) before (rather than after) `\documentclass{beamer}`.

`/mmz/`**per overlay**   (style)

> Applying this style to a piece of memoized code contained in a Beamer `frame` will produce a separate cc-memo (and extern) for each overlay of the frame. See section 2.7 for the tutorial.
>
> Note that this style is *not* intended for externalization of entire frames.
>
> The implementation, documented in section 4.2.4, is not trivial, but only deploys the public interface offered by Memoize.

5.7.5 Biblatex

`/mmz/`**biblatex**=`{`⟨*additional configuration*⟩`}`   (style)

> This key loads the `biblatex` support, but only if package `biblatex` is loaded (anywhere in the preamble); after loading the support, ⟨*additional configuration*⟩ is executed within `\mmzset`.
>
> When `biblatex` support is loaded, all citation commands defined by this package[103] behave like `\ref` in the sense that changing a bibliography entry referred to by a citation command within a piece of memoized code triggers recompilation of that code. For example, it is safe to use `\textcite{`⟨*citation key*⟩`}` inside memoized code, in the sense that the extern will be recompiled when the entry for the ⟨*citation key*⟩ is changed in the `.bib` file.
>
> Furthermore, Memoize takes care that even once an extern which contains a reference to ⟨*citation key*⟩ is utilized (i.e. included into the document as a picture), the ⟨*citation key*⟩ is marked (by `\nocite`) as used, and will appear among the items listed by `\printbibliography`.
>
> In most cases, the out-of-the-box level of `biblatex` support will suffice. However:
>
> - To support an additional citation command (most likely defined via `\DeclareCiteCommand` etc.), advise it by one of `cite` and friends below.
> - To have ibid (and similar) tracking sensitive to citation commands within memoized code, execute `biblatex ccmemo cite`.
>
> Under the hood, `biblatex` support does two things:

---

[103]In `biblatex` version 3.19: `\cite`, `\Cite`, `\parencite`, `\Parencite`, `\footcite`, `\footcitetext`, `\textcite`, `\Textcite`, `\smartcite`, `\Smartcite`, `\supercite`, `\cites`, `\Cites`, `\parencites`, `\Parencites`, `\footcites`, `\footcitetexts`, `\smartcites`, `\Smartcites`, `\textcites`, `\Textcites`, `\supercites`, `\autocite`, `\Autocite`, `\autocites`, `\Autocites`, `\citeauthor`, `\Citeauthor`, `\citetitle`, `\citeyear`, `\citedate`, `\citeurl`, `\nocite`, `\fullcite`, `\footfullcite`, `\volcite`, `\Volcite`, `\volcites`, `\Volcites`, `\pvolcite`, `\Pvolcite`, `\pvolcites`, `\Pvolcites`, `\fvolcite`, `\Fvolcite`, `\fvolcites`, `\Fvolcites`, `\ftvolcite`, `\ftvolcites`, `\Ftvolcite`, `\Ftvolcites`, `\svolcite`, `\Svolcite`, `\svolcites`, `\Svolcites`, `\tvolcite`, `\Tvolcite`, `\tvolcites`, `\Tvolcites`, `\avolcite`, `\Avolcite`, `\avolcites`, `\Avolcites`, `\notecite`, `\Notecite`, `\pnotecite`, `\Pnotecite`, `\fnotecite`, `\citename`, `\citelist`, `\citefield`.

- It declares advice for `biblatex`'s internal macro `\blx@bbl@entry`, executed while processing the `.bbl` file, which makes it possible for Memoize to detect changes in bibliography entries.

- It defines the `auto`-keys of the `cite` family, documented below, and advises all `biblatex`'s citation commands with these keys.

/mmz/auto/**cite**	(style)
/mmz/auto/**cites**	(style)
/mmz/auto/**volcite**	(style)
/mmz/auto/**volcites**	(style)

When a citation command is advised by one of these `auto`-keys, a piece of memoized code containing the citation command will be recompiled upon changing the bibliography entry of a citation key used by the command; in other words, the citation command will behave in a fashion analogous to `ref`.

Furthermore, the memo corresponding to the memoized piece of code will contain a `\nocite` command for each citation key occurring within the code; in effect, citation keys continue to be marked as used even when utilizing the memo/extern. For further details, see `biblatex ccmemo cite`.

Out of the box, Memoize advises all `biblatex`'s (v3.19) citation commands with these keys. If a new citation command is defined, it may be submitted to these keys by the author. Use

- `auto=⟨citation command⟩{cite}` for a ⟨*citation command*⟩ in the `\cite` family,

- `auto=⟨citation command⟩{cites}` for a ⟨*citation command*⟩ in the `\cites` family,

- `auto=⟨citation command⟩{volcite}` for a ⟨*citation command*⟩ in the `\volcite` family, and

- `auto=⟨citation command⟩{volcites}` for a ⟨*citation command*⟩ in the `\volcites` family.

Which key should be used with which command depends on the argument structure of the citation command, as documented in section 3.9 of the `biblatex` manual. Accordingly, the auto-keys of the `cite` family set `args` as follows:

	args
`cite`, `cites`	`l *m`
`volcite`, `volcites`	`lm l *m`

Commands like `\cite` take one mandatory argument (the citation key), which may be preceded by an optional star, prenote and postnote. Key `cite` grabs all the optional arguments by `l`, while `*m` receives the braced ⟨*citation key*⟩ argument; `*` is locally defined to mark the following argument as the citation key. Commands like `\volcite` take two mandatory arguments: the volume and the citation key, in this order; each may be preceded by an optional argument. Again, the optional arguments are collected by `ls`, and the mandatory arguments by `ms`, with the second argument being marked as the citation key by `*`. The two `cites` variants use the given `args` unconventionally: they iterate the given pattern as long as the citation arguments continue (see `return` for the gist of how they achieve this).

When advising new citation commands, the author may override the given `args` if necessary. In fact, this is what Memoize does do support `\citename`, `\citelist` and `\citefield`. With these commands, the braced ⟨*citation key*⟩ is followed by optional arguments followed by another braced mandatory argument, which implies `lm` following `cite`'s `args`. For example, `\citename` is advised by `auto={cite, args=l *m lm}`.

/mmz/**biblatex ccmemo cite**=`nocite`|`replicate`	(initially `nocite`, no default)

Select the citation command which Memoize puts into the cc-memo in order to broadcast that the citation key was used in the compilation of the extern.

By default, this citation command is `\nocite`. Even if the memoized code uses e.g. `\textcite{⟨citation key⟩}`, the cc-memo contains `\nocite{⟨citation key⟩}`. This usually suffices, as we normally only have to inform `biblatex` that the ⟨*citation key*⟩ was used in the utilized extern.

However, in the rare situations where a tracker (page, cite, ibid, opcit, loccit or idem tracker) depends on the citation used in the extern, the `\nocite` approach does not suffice. If such tracking is required, the author should issue `biblatex ccmemo cite=replicate` to have the cc-memo contain the original citation command; note that if the cc-memo was already created, recompilation is necessary.

The replication approach has the downside of yielding lots of 'Missing character' warnings. This happens because, in order to use the original citation command but not produce an actual citation, Memoize precedes the invocation of the original citation command by `\nullfont`.

6 Varia

6.1 Changelog

The changelogs of Memoize, Advice and CollArgs can be found in the documentation directory of the respective package, i.e. in *⟨texmf⟩*/`doc/generic/`*⟨package⟩*/`CHANGELOG.md`. They are written in the MarkDown format. Here are direct links to these changelogs on CTAN: Memoize, Advice, CollArgs.

6.2 Known issues

Bitmap graphics export is coming up in the next release of the package.

An error occurs, but disappears in the next compilation   If a non-fatal internal TeX error occurs during memoization, the memos and externs may be nevertheless produced and utilized in subsequent compilations. In such a case, the erroneous code won't be compiled again, and therefore won't yield any errors, giving the mistaken impression that the code is error-free.

This problem does not apply to errors which trigger `\errmessage`, because that control sequence is advised by `abort`. Note that internal TeX errors like `Undefined control sequence` are *not* reported through `\errmessage`, and will therefore cause the issue.

This problem does not affect LuaTeX, because this engine allows Memoize to detect errors and abort memoization if it encounters any.

A minimal issue with XᴇLATEX   If the very first page of a document of class `minimal`, compiled by XᴇLATEX, happens to be an extern page, we have a problem: all the regular pages of the document will be of the same size as that extern page. pdfLATEX and LuaLATEX do not exhibit this behaviour, nor do LATEX classes other than `minimal`, even when compiled with XᴇLATEX.

CollArgs

Due to an unfortunate design decision, CollArgs does not accept a dot `.` as the *⟨token⟩* argument of types, `r`, `R`, `d`, `D`, and `t`. The issue will be fixed in a subsequent release.

6.3 Troubleshooting

Extern extraction does not work

Upon an unsuccessful extern extraction, you should get one of the errors or warnings listed below. They should appear both in the terminal output and in the log file. All but the final error can also appear when extern extraction is performed externally.

Errors:

1. `Error: Python module 'pdfrw' was not found'`,
   `Error: Perl module 'PDF::API' was not found"`, or
   `Error: Perl module 'PDF::Builder' was not found"`

   You have not installed the PDF processing library, at least not successfully. The installation instructions can be found in section 1.1. Perhaps you have multiple instances of Perl/Python on your system and you have installed the library into the wrong instance?

2. `Error: File "`*⟨jobname⟩*`.pdf" seems corrupted. Perhaps you have to load Memoize earlier in the preamble?`

   While the PDF could be corrupted for various reasons, the most probable reason is the package loading order, as indicated in the error message itself. (Note that the absence of *⟨jobname⟩*`.pdf` does not yield this error. This is so that a fatal error unrelated to Memoize doesn't make Memoize throw another error in the next compilation.) Embedded extern extraction requires an intact

document PDF from the previous compilation, so Memoize must be loaded before the document PDF is opened for writing the results of the ongoing compilation. In particular, the PDF is opened by PGF library `fadings`, included by TikZ's libraries `fadings` and `shadows`, so Memoize must be loaded before any of these libraries. With `beamer`, the problem is particularly acute because the PDF is opened while loading the class. In this case, simply moving `\usepackage{memoize}` up the preamble, as suggested, won't help: you have to write `\RequirePackage{memoize}` *before* `\documentclass{beamer}`!

It is also possible that the PDF processing library deployed by the extraction script chokes on your particular document. Inspecting the error message original error message produced by Perl/Python, which can be found at the end of the long error message in the log, should help you see whether this is the case. For example, a couple of times, I got `Invalid dictionary key at /.../perl5/site_perl/PDF/API2/Basic/PDF/File.pm line` N when extracting with the Perl-based script. The issue might be related to PDF version — by default, TeX produces PDFs of version 1.5, while the PDF library `PDF::API2` officially only supports versions up to 1.4 — but I'm afraid I haven't identified the exact circumstances yet (possibly, the externalizing a picture containing an embedded PDF file might be the culprit). In general, the workaround is to use another extraction method or PDF processing library, see `extract` and `--library`.

3. `Error: I'm not allowed to write to '⟨`*filename*`⟩' (openout_any = ⟨`*mode*`⟩)"`

   Your TeX distribution does not allow this file to be written to. The relevant setting is called `openin_any` in TeXLive and `[Core]AllowUnsafeOutputFiles` in MiKTeX.

4. `Error: I'm not allowed to read from '⟨`*filename*`⟩' (openin_any = ⟨`*mode*`⟩)"`

   Your TeX distribution does not allow this file to be read from. The relevant setting is called `openout_any` in TeXLive and `[Core]AllowUnsafeInputFiles` in MiKTeX.

5. `Error: "Semi-absolute" paths are disallowed: '⟨`*filename*`⟩'`

   This error can only occur on Windows. The extraction scripts are even more paranoid than TeX and don't allow paths such as `C:foo\bar` or `\foo\bar`, neither as path to a `.mmz`, memo or extern file nor as an output or temporary directory.

6. `Error: I cannot extract page ⟨`*n*`⟩ '⟨`*filename*`⟩.pdf', as it contains only ⟨`*N*`⟩ pages`

   The document PDF is too short. This could happen if you are accidentally triggering the extern extraction twice, and the first extraction was called with `--prune`.

7. `Error: Python error: ⟨`*error message*`⟩` or `Error: Perl error: ⟨`*error message*`⟩`

   A runtime error occurred during the execution of the extraction script. The error message should be followed by the traceback information which includes the line number (in the extraction script) where the error occurred.

   One possible source of this error are insufficient filesystem permissions. For example, it will occur if the current directory is marked as read-only and the temporary output directory (`$TEXMFOUTPUT`) is not set.

8. `Error: Extraction of externs from document "⟨`*jobname*`⟩.pdf" using method "⟨`*extraction method*`⟩" was unsuccessful.`

   This is a generic error produced when the extraction script was either not executed at all, or the execution didn't finish properly.[104]

   A couple of reasons for the failure to execute the script:

   - Is the shell escape configured properly? It should be, as both Memoize's extraction scripts are listed among restricted shell escape commands in both TeXLive and MiKTeX, but it never hurts to check.

---

[104]How does Memoize know whether this happened? When invoked given option `--format`, the extraction script is supposed to write a log (actually, status) file called ⟨*jobname*⟩`.mmz.log`. Upon a clean completion, the final line of this log reads `\endinput`. If this marker is missing, Memoize produces the error under discussion. Feel free to see what's up with this file if all else fails.

The easiest way to see if shell escape is the culprit is to compile the document with command option `-shell-escape` (on TeXLive) or `--enable-write18` (on MiKTeX). If this resolves the problem, check your shell escape mode; see footnote 1 on page 6 for how to do this.

- Is environment variable PATH set correctly, so that the system can find the extraction script?

- Is Perl/Python installed on your system, and accessible to TeX?

Normally, you should get error 7 if a runtime error occurs during the execution of the extraction script. You can only get the generic error if the runtime error occurred before the script could set up the log file, perhaps due to insufficient filesystem permissions in combination with the paranoid `openout_any` setting (which is the default).

If the source of the error remains a mistery, I suggest inspecting the following sources of information, to help you with your investigation:

- Can you can run the extraction script by hand? Open the terminal, go into the directory containing your document, write `memoize-extract.pl` ⟨*document name*⟩ (or `memoize-extract.py` ⟨*document name*⟩), and see what happens.

- Inspect ⟨*jobname*⟩`.log` — search for `runsystem(memoize-extract.pl ...)`, it will tell you whether the script was executed.

- Inspect the TeX terminal output — if the script was executed, it should've announced itself by `Extracting externs from` ⟨*jobname*⟩`.pdf`; are there any further messages between this header and the error message?

- Inspect ⟨*the path to the extern*⟩`.log`, if you are using TeX-based extraction (`extract`=`tex`).

Not every failure to extract externs results in an error. When a warning is produced, the compilation will succeed, it's just that as Memoize cannot extract the externs, they will be produced, and dumped into your document, at each and every compilation. For warnings other than the missing document PDF, Memoize extracts as many externs as possible.

1. `Warning: Cannot open '`⟨*filename*⟩`.pdf'`

   This is not an error because what if you deleted the document PDF on purpose?

2. `Warning: I refuse to extract page` n `from "`⟨*jobname*⟩`.pdf", because its size (`⟨*width*⟩$\times$ ⟨*height*⟩`) is not what I expected`
   (⟨*expected width*⟩ $\times$ ⟨*expected height*⟩)

   If the compilation which produced the offending extern pages yielded any errors, you should probably disregard this warning, fix the errors, and compile again. Otherwise, you have somehow winded up with mismatched ⟨*jobname*⟩`.pdf` and ⟨*jobname*⟩`.mmz` (the latter file contains instructions on which pages to extract, complete with the expected dimensions). Are you sure that they were produced by the same compilation, and have remained untouched since? Are you perhaps trying to perform the extraction the second time, after the first extraction `--prune`d the PDF?

   If the warning stubbornly persists, but you are sure that the page the script is refusing to extract is correct, you can force the extraction by adding option `--force` to the script invocation, which can be set by `perl extraction options`. However, as such a situation probably indicates a bug in Memoize, please let me know about it.

3. `Warning: I refuse to extract page` n `into extern` ⟨*extern filename*⟩`, because the associated (c)c-memo does not exist.`

   Assuming that you haven't deleted the memos (`.memo` files), either manually or via `memoize-clean.pl`, could it be that they were never created in the first place? Check where they should be written to (the configuration commands are listed in section 5.4). Is that directory writeable, both in the sense of the system and for TeX (the `openout` setting in `texmf.cnf`)?

An extern won't be included

Did you receive a warning or error message?

1. `Package memoize Warning: Unexpected size of extern "`⟨*extern path*⟩`.pdf"; expected` ⟨*expected width*⟩ × ⟨*expected height*⟩`, got` ⟨*width*⟩ × ⟨*height*⟩

   This warning is related to warning 2 above, only that it occurs once the extern is extracted. The same investigative methods apply.

2. `!pdfTeX error: pdflatex (file` ⟨*extern path*⟩`.pdf): reading image file failed`, or something similar for engines other than pdfTeX

   This is a fatal error. The extern file got corrupted, somehow — inexistent and even empty extern files merely trigger recompilation.

3. `pdfTeX warning: pdflatex (file` ⟨*extern path*⟩`.pdf): PDF inclusion: found PDF version <m>, but at most version <n> allowed`

   When you produced the externs, a higher `\pdfmajorversion` and/or `\pdfminorversion` was in effect than now. I guess you shouldn't worry about this warning if the output looks fine.

If there was no warning or error — are you certain that Memoize is `enable`d, and that it is not in the `recompile` mode? Remember that these settings can also apply only to a part of the document; search for any stray `\mmzset` or `\mmznext` commands.

Warnings about duplicate labels, indices, etc. may be safely disregarded.

Externalization causes any (non-immediate) `\write` commands in the extern to be executed twice, once upon the shipout of the regular page, and once upon the shipout of the extern page. This results in warnings about doubly defined labels, hyperreferences, indices, etc. For example, you might get `LaTeX Warning: Label `<name>' multiply defined` or `warning (pdf backend): ignoring duplicate destination with the name '<name>'`. You can safely disregard these warnings; they will disappear once the extern is utilized.

Memoization was aborted

This warning means that either:

- you are trying to (auto)memoize a `tikzpicture` with `remember picture` set, or more generally, some code which contains `\(pdf)savepos` — this can't be done, see section 3.1; or

- an error occurred during memoization — in this case, Memoize cowardly refuses to proceed with memoization, see section 6.2 for details.

6.4 License

Copyright © 2020- Sašo Živanović.

This work may be distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3c of this license or (at your option) any later version. The latest version of this license is in `https://www.latex-project.org/lppl.txt` and version 1.3 or later is part of all distributions of LaTeX version 2008 or later.

This work comprises of the sources, generated files, accompanying scripts, auxiliary files and scripts, documentation, documentation sources, examples and example sources of packages `memoize`, `nomemoize`, `memoizable`, `auto` and `collargs`. The files belonging to this work and covered by LPPL are listed in ⟨*texmf*⟩`/doc/generic/memoize/FILES`.

This work has the LPPL maintenance status 'maintained.' The Current Maintainer of this work is Sašo Živanović. The work is available on CTAN at `https://ctan.org/pkg/memoize` and on GitHub at `https://github.com/sasozivanovic/memoize`.

6.5 Acknowledgments

First and foremost, my gratitude goes to Stefan Müller of Language Science Press for his encouragement and patience — without him, this package might've remained an idea forever!

But there were others as well. Shunsaku Hirata of the X∄TEX team promptly resolved an issue with forward references in `xdvipdfmx`, and thereby made Memoize work with X∄TEX. Joseph Wright kindly didn't object to me misappropriating `etoolbox` for other formats. Petra Rübe-Pugliese (CTAN) and Karl Berry (TEXLive) offered good advice on packaging the package; I would never get to call Advice Advice without their advice. Karl also performed a security review of the extraction scripts, providing some very useful comments along the way, and agreed to include the scripts among the shell escape commands allowed in TEXLive. Christian Schenk worked hard to support Memoize in MiKTEX. After the package was first published, amonakov, mbert, Qrrbrbirlbel, and Ulrike Fischer provided valuable feedback, suggestions and/or bug reports; cfr did all that, and also advertised the package extensively.

Thank you all!

Index