Dec@ding style files

Andres Löh

Universiteit Utrecht

andres@cs.uu.nl

November 8, 2002

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

- ➔ We will do that.

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

- We will do that.
- We will learn that there is much more to know about LaTeX than is covered by most LaTeX books.

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

- We will do that.
- We will learn that there is much more to know about LaTeX than is covered by most LaTeX books.
- We will explore some of the theory necessary to understand what is going on in style files.

Goals of this t@lk

Have you ever looked at a LaTeX style (package) file?

- We will do that.
- We will learn that there is much more to know about LaTeX than is covered by most LaTeX books.
- We will explore some of the theory necessary to understand what is going on in style files.
- We will learn that LaTeX offers an extremely powerful, but also extremely confusing programming language.

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

- We will do that.
- We will learn that there is much more to know about LaTeX than is covered by most LaTeX books.
- We will explore some of the theory necessary to understand what is going on in style files.
- We will learn that LaTeX offers an extremely powerful, but also extremely confusing programming language.
- We will **not** learn how to write own programs/styles for LaTeX.

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

- We will do that.
- We will learn that there is much more to know about LaTeX than is covered by most LaTeX books.
- We will explore some of the theory necessary to understand what is going on in style files.
- We will learn that LaTeX offers an extremely powerful, but also extremely confusing programming language.
- We will **not** learn how to write own programs/styles for LaTeX.
- We will **not** explain everything in detail.

Goals of this `t@lk`

Have you ever looked at a LaTeX style (package) file?

- We will do that.
- We will learn that there is much more to know about LaTeX than is covered by most LaTeX books.
- We will explore some of the theory necessary to understand what is going on in style files.
- We will learn that LaTeX offers an extremely powerful, but also extremely confusing programming language.
- We will **not** learn how to write own programs/styles for LaTeX.
- We will **not** explain everything in detail.

Don't panic!

Using p@ck@ges

LaTeX provides the \usepackage command:

```
\usepackage{tabularx}
\usepackage[german]{babel}
```

Using p@ck@ges

LaTeX provides the \usepackage command:

```
\usepackage{tabularx}
\usepackage[german]{babel}
```

- There is a huge amount of packages available for LaTeX – far more than are shipped with the common distributions.

Using p@ck@ges

LaTeX provides the \usepackage command:

```
\usepackage{tabularx}
\usepackage[german]{babel}
```

- There is a huge amount of packages available for LaTeX – far more than are shipped with the common distributions.
- Check www.ctan.org if you are interested.

Using p@ck@ges

LaTeX provides the \usepackage command:

```
\usepackage{tabularx}
\usepackage[german]{babel}
```

- There is a huge amount of packages available for LaTeX – far more than are shipped with the common distributions.
- Check www.ctan.org if you are interested.
- The command usepackage essentially includes the corresponding style file into your LaTeX source.
  - \usepackage{tabularx} would look for tabularx.sty.
  - \usepackage{babel} would look for babel.sty.

Using p@ck@ges

LaTeX provides the \usepackage command:

```
\usepackage{tabularx}
\usepackage[german]{babel}
```

- There is a huge amount of packages available for LaTeX – far more than are shipped with the common distributions.
- Check www.ctan.org if you are interested.
- The command usepackage essentially includes the corresponding style file into your LaTeX source.
  - \usepackage{tabularx} would look for tabularx.sty.
  - \usepackage{babel} would look for babel.sty.
- Options (in square brackets) can be passed to the packages.

What d@ style files d@?

- They can **change** the behaviour of LaTeX (for instance, by redefining existing commands).

What d@ style files d@?

- They can **change** the behaviour of LaTeX (for instance, by redefining existing commands).
- They can **provide new** commands (and environments).

What `d@` style files `d@`?

- They can **change** the behaviour of LaTeX (for instance, by redefining existing commands).
- They can **provide new** commands (and environments).
- In principle, style files are nothing more than LaTeX sources themselves.

What d@ style files d@?

- They can **change** the behaviour of LaTeX (for instance, by redefining existing commands).
- They can **provide new** commands (and environments).
- In principle, style files are nothing more than LaTeX sources themselves.

But . . .

An excerpt fr@m the tabularx style file

```
[...]
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
\def\TX@{tabularx}
[...]
```

An excerpt fr@m the tabularx style file

```
[...]
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
\def\TX@{tabularx}
[...]
```

An excerpt fr@m the tabularx style file

```
[...]
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
\def\TX@{tabularx}
[...]
```

It's a kind of `m@gic`

- The code looks generally cryptic.

It's a kind of `m@gic`

- The code looks generally cryptic.
- It seems to use all sorts of strange commands.

It's a kind of `m@gic`

- The code looks generally cryptic.
- It seems to use all sorts of strange commands.
- If one looks, for instance, into the LaTeX Companion one will not find any explanation.

It's a kind of `m@gic`

- The code looks generally cryptic.
- It seems to use all sorts of strange commands.
- If one looks, for instance, into the LaTeX Companion one will not find any explanation.
- (Nevertheless, the LaTeX companion is a very nice book and hereby recommended.)

It's a kind of `m@gic`

- The code looks generally cryptic.
- It seems to use all sorts of strange commands.
- If one looks, for instance, into the LaTeX Companion one will not find any explanation.
- (Nevertheless, the LaTeX companion is a very nice book and hereby recommended.)
- In fact, we deal here with primitive and LaTeX kernel commands ...

A bit of `hist@ry`

A bit of `hist@ry`

- TEX is a typesetting system created by Donald E. Knuth.

A bit of hist@ry

- TEX is a typesetting system created by Donald E. Knuth.
- TEX itself offers only a small number of primitive commands, but is programmable through a powerful macro language.

A bit of `hist@ry`

- **TEX** is a typesetting system created by Donald E. Knuth.
- TEX itself offers only a small number of primitive commands, but is programmable through a powerful macro language.
- For TEX to be usable, Knuth also created a set of useful macros called **plainTEX**.

A bit of `hist@ry`

- **TEX** is a typesetting system created by Donald E. Knuth.
- TEX itself offers only a small number of primitive commands, but is programmable through a powerful macro language.
- For TEX to be usable, Knuth also created a set of useful macros called **plainTEX**.
- Later, Leslie Lamport developed a far more sophisticated (and complex) macro package that he called **LATEX**.

A bit of `hist@ry`

- **TEX** is a typesetting system created by Donald E. Knuth.
- TEX itself offers only a small number of primitive commands, but is programmable through a powerful macro language.
- For TEX to be usable, Knuth also created a set of useful macros called **plainTEX**.
- Later, Leslie Lamport developed a far more sophisticated (and complex) macro package that he called **LATEX**.
- To facilitate the switch for former plainTEX users, he included most of plainTEX's macros in LATEX, but additionally created improved versions of these commands.

A bit of `hist@ry`

- **TEX** is a typesetting system created by Donald E. Knuth.
- TEX itself offers only a small number of primitive commands, but is programmable through a powerful macro language.
- For TEX to be usable, Knuth also created a set of useful macros called **plainTEX**.
- Later, Leslie Lamport developed a far more sophisticated (and complex) macro package that he called **LATEX**.
- To facilitate the switch for former plainTEX users, he included most of plainTEX's macros in LATEX, but additionally created improved versions of these commands.
- The key to LATEX's success is its class/package system which makes it easy to integrate third-party extensions into LATEX.

A bit of `hist@ry`

- **TEX** is a typesetting system created by Donald E. Knuth.
- TEX itself offers only a small number of primitive commands, but is programmable through a powerful macro language.
- For TEX to be usable, Knuth also created a set of useful macros called **plainTEX**.
- Later, Leslie Lamport developed a far more sophisticated (and complex) macro package that he called **LATEX**.
- To facilitate the switch for former plainTEX users, he included most of plainTEX's macros in LATEX, but additionally created improved versions of these commands.
- The key to LATEX's success is its class/package system which makes it easy to integrate third-party extensions into LATEX.
- The term LATEX nowadays refers to all packages available for it. The core macro package originally written by Lamport is called the **LATEX kernel** or **LATEX format**.

What do we learn `fr@m` that?

- TEX is the name of the underlying typesetting system.
- When we call `latex` on the command line, we still call TEX, but with the LATEX format preloaded.
- When we call `tex` on the command line, then TEX with the plainTEX format would be used.
- There are more formats than just plainTEX and LATEX, some of them more recent, among them the very promising ConTEXt.

What has `@ll` this `t@` do with style files?

If we encounter unknown commands (for instance in in style files), that can have multiple reasons:

What has `@ll` this `t@` do with style files?

If we encounter unknown commands (for instance in in style files), that can have multiple reasons:

- The command is defined in another style (or the class) file.

What has `@ll` this `t@` do with style files?

If we encounter unknown commands (for instance in in style files), that can have multiple reasons:

- The command is defined in another style (or the class) file.
- The command is defined by the LaTeX kernel. Maybe it is even a plainTeX command that is included in the LaTeX kernel.

What has `@ll` this `t@` do with style files?

If we encounter unknown commands (for instance in in style files), that can have multiple reasons:

- The command is defined in another style (or the class) file.
- The command is defined by the LaTeX kernel. Maybe it is even a plainTeX command that is included in the LaTeX kernel.
- The command is a primitive command.

What has `@ll` this `t@` do with style files?

If we encounter unknown commands (for instance in in style files), that can have multiple reasons:

- The command is defined in another style (or the class) file.
- The command is defined by the LaTeX kernel. Maybe it is even a plainTeX command that is included in the LaTeX kernel.
- The command is a primitive command.

If a LaTeX book does not explain a certain command, we might have a chance looking at

- the LaTeX kernel sources;
- books about plainTeX and TeX itself.

The mystery of the `@` ...

LaTeX tries to hide a great number of commands from the user.

The mystery of the `@` ...

LaTeX tries to hide a great number of commands from the user.

- ➔ If a command contains an `@`, it is not accessible in normal source files, but only in style files.

The mystery of the `@` ...

LaTeX tries to hide a great number of commands from the user.

- If a command contains an `@`, it is not accessible in normal source files, but only in style files.
- The author of a style file can in this manner distinguish between

The mystery of the `@` ...

LaTeX tries to hide a great number of commands from the user.

- If a command contains an `@`, it is not accessible in normal source files, but only in style files.
- The author of a style file can in this manner distinguish between
  - internal macros that are only used inside the package

The mystery of the `@` ...

LaTeX tries to hide a great number of commands from the user.

- If a command contains an `@`, it is not accessible in normal source files, but only in style files.
- The author of a style file can in this manner distinguish between
  - internal macros that are only used inside the package
  - external macros that provide the interface to the package for a user

The mystery of the `@` ...

LaTeX tries to hide a great number of commands from the user.

- If a command contains an `@`, it is not accessible in normal source files, but only in style files.
- The author of a style file can in this manner distinguish between
  - internal macros that are only used inside the package
  - external macros that provide the interface to the package for a user
- The LaTeX kernel itself defines a huge number of internal commands that are also used by package authors.

The mystery of the @ ...

LaTeX tries to hide a great number of commands from the user.

- If a command contains an @, it is not accessible in normal source files, but only in style files.
- The author of a style file can in this manner distinguish between
  - internal macros that are only used inside the package
  - external macros that provide the interface to the package for a user
- The LaTeX kernel itself defines a huge number of internal commands that are also used by package authors.
- Note that there is (unfortunately) no namespace management in LaTeX. Internal commands defined in one package are still visible in all other packages.

The rest of the `t@lk`

A tour of the `tabularx` package

The rest of the `t@lk`

A tour of the `tabularx` package

- The package defines an environment `tabularx`, which is based upon `tabular`. Given a total (target) width, it can compute the width of one or more columns automatically.

The rest of the `t@lk`

A tour of the `tabularx` package

- The package defines an environment `tabularx`, which is based upon `tabular`. Given a total (target) width, it can compute the width of one or more columns automatically.
- The package `tabularx` is not a typical package, because it is very well documented.

The rest of the `t@lk`

A tour of the `tabularx` package

- The package defines an environment `tabularx`, which is based upon `tabular`. Given a total (target) width, it can compute the width of one or more columns automatically.
- The package `tabularx` is not a typical package, because it is very well documented.
- We will review (a part of) the source file page by page.

The rest of the `t@lk`

A tour of the `tabularx` package

- The package defines an environment `tabularx`, which is based upon `tabular`. Given a total (target) width, it can compute the width of one or more columns automatically.
- The package `tabularx` is not a typical package, because it is very well documented.
- We will review (a part of) the source file page by page.
- We will introduce and discuss new concepts as we encounter them.

The rest of the `t@lk`

A tour of the `tabularx` package

- The package defines an environment `tabularx`, which is based upon `tabular`. Given a total (target) width, it can compute the width of one or more columns automatically.
- The package `tabularx` is not a typical package, because it is very well documented.
- We will review (a part of) the source file page by page.
- We will introduce and discuss new concepts as we encounter them.
- We will concentrate on the general ideas and skip many details.

In the @beginning

```
%%
%% This is file ‘tabularx.sty’,
%% generated with the docstrip utility.
%%
%% The original source files were:
%%
%% tabularx.dtx  (with options: ‘package’)
%%
%% This is a generated file.
%%
%% Copyright 1993 1994 1995 1996 1997 1998 1999 2000
%% The LaTeX3 Project and any individual authors listed elsewhere
%% in this file.
[...]
```

In the `@beginning`

```
%%
%% This is file ‘tabularx.sty’,
%% generated with the docstrip utility.
%%
%% The original source files were:
%%
%% tabularx.dtx  (with options: ‘package’)
%%
%% This is a generated file.
%%
%% Copyright 1993 1994 1995 1996 1997 1998 1999 2000
%% The LaTeX3 Project and any individual authors listed elsewhere
%% in this file.
[...]
```

- The file starts with commentary, explaining the nature of the file.

In the `@beginning`

```
%%
%% This is file ‘tabularx.sty’,
%% generated with the docstrip utility.
%%
%% The original source files were:
%%
%% tabularx.dtx  (with options: ‘package’)
%%
%% This is a generated file.
%%
%% Copyright 1993 1994 1995 1996 1997 1998 1999 2000
%% The LaTeX3 Project and any individual authors listed elsewhere
%% in this file.
[...]
```

- The file starts with commentary, explaining the nature of the file.
- Obviously, this file has been generated from yet another file, namely `tabularx.dtx`, with the help of a mysterious tool called `docstrip`.

The `phil@s@phy` of literate programming

The `phil@s@phy` of literate programming

- Knuth himself favored a style of programming where the program documents itself.

The `phil@s@phy` of literate programming

- Knuth himself favored a style of programming where the program documents itself.
- The program code is embedded in a file which can be typeset by TEX, yielding a nicely typeset version of the program code together with its documentation.

The `phil@s@phy` of literate programming

- Knuth himself favored a style of programming where the program documents itself.
- The program code is embedded in a file which can be typeset by TEX, yielding a nicely typeset version of the program code together with its documentation.

`docstrip` – literate programming in LATEX

- The `docstrip` utility (written in LATEX!) has been designed in this spirit, to allow LATEX package writers to use literate programming.

The `phil@s@phy` of literate programming

- Knuth himself favored a style of programming where the program documents itself.
- The program code is embedded in a file which can be typeset by TEX, yielding a nicely typeset version of the program code together with its documentation.

`docstrip` – literate programming in LATEX

- The `docstrip` utility (written in LATEX!) has been designed in this spirit, to allow LATEX package writers to use literate programming.
- A central source file (with extension `.dtx`) contains documentation as well as all the program code.

The `phil@s@phy` of literate programming

- Knuth himself favored a style of programming where the program documents itself.
- The program code is embedded in a file which can be typeset by TEX, yielding a nicely typeset version of the program code together with its documentation.

`docstrip` – literate programming in LATEX

- The `docstrip` utility (written in LATEX!) has been designed in this spirit, to allow LATEX package writers to use literate programming.
- A central source file (with extension `.dtx`) contains documentation as well as all the program code.
- LATEX can be run on the `.dtx` to generate the package documentation.

The `phil@s@phy` of literate programming

- Knuth himself favored a style of programming where the program documents itself.
- The program code is embedded in a file which can be typeset by TEX, yielding a nicely typeset version of the program code together with its documentation.

`docstrip` – literate programming in LATEX

- The `docstrip` utility (written in LATEX!) has been designed in this spirit, to allow LATEX package writers to use literate programming.
- A central source file (with extension `.dtx`) contains documentation as well as all the program code.
- LATEX can be run on the `.dtx` to generate the package documentation.
- The `docstrip` utility can be run on the `.dtx` to extract all the program code, i.e. to produce the `.sty` file.

Pr@ctical information about docstrip

- If you download a package (for instance from CTAN), you often get just two files: an installation script `.ins`, and the literate source `.dtx`.
- Run LaTeX on the `.ins` file. This will call `docstrip` on the `.dtx` to generate all the needed source files, among them the `.sty` style file.
- Run LaTeX on the `.dtx` file directly to generate the documentation.
- The `docstrip` program **is** documented in the LaTeX companion.

The `interf@ce` of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

The `interf@ce` of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

- `NeedsTeXFormat` states that the package requires the current LaTeX version LaTeX 2_ε and will not work with older versions of LaTeX.

The `interf@ce` of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

- `ProvidesPackage` is used to give the name of the package and version information.

The `interf@ce` of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

- `DeclareOption` can be used to declare options that can be passed to the package in square brackets to activate or deactivate specific functionality.

The `interf@ce` of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

- `ProcessOptions` is needed to really parse the options passed to the package and execute the appropriate commands.

The `interf@ce` of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

- `RequirePackage` is the package writer's version of `\usepackage`. It loads a package, but only if it has not yet been loaded.

The interf@ce of the package

We skip the license and the copyright. The package's author is David Carlisle, who wrote a great number of excellent packages and participates actively in the development of the LaTeX kernel.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{tabularx}
          [1999/01/07 v2.07 `tabularx' package (DPC)]
\DeclareOption{infoshow}{\AtEndOfPackage\tracingtabularx}
\DeclareOption{debugshow}{\AtEndOfPackage\tracingtabularx}
\ProcessOptions
\RequirePackage{array}[1994/02/03]
```

LaTeX provides a limited amount of package and version management:

- `AtEndOfPackage` can be used to store a command for execution at the end of the package.
  Question: Why not execute it here?

Reserving@ registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

Reserving@ registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

- `newdimen` allocates a **dimension** register. It can store a numerical value plus a unit, for example a length or a width. Valid values would be `1em` or `7.3cm`.

Reserving@ registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

- `newcount` allocates a **counter** register. It can store a (possibly negative) integer value, such as `-2` or `7`. Possible applications would be counters for the current page number, or the current chapter number.

Reserving@ registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

- `newif` allocates a **boolean**. It can only store **true** or **false**. Decisions can be made depending on the current value of the boolean.

`Reserving@` registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

- There are more: `\newtoks` allocates a **token** register. It can store a number of words from the input stream. We will hear more about those later.

Reserving@ registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

- ➔ `\newbox` allocates a **box** register. Boxes can contain portions of typeset text. They can be measured.

`Reserving@` registers

```
\newdimen\TX@col@width
\newdimen\TX@old@table
\newdimen\TX@old@col
\newdimen\TX@target
\newdimen\TX@delta
\newcount\TX@cols
\newif\ifTX@
```

TEX has so-called registers, slots in memory for values of a certain number of datatypes:

- `\newskip` allocates a new **skip** register. Skips are similar to dimensions, but can contain stretchable and/or shrinkable **glue**. We will not need them.
- There are yet more that we do not need: `\newmuskip` for mathematical skips, `\newread` and `\newwrite` for file input/output, `\newfam` for math families, and `\newlanguage` for hyphenation rules.

The re@l w@rk begins

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

The re@l w@rk begins

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

The plan

The re@l w@rk begins

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

The plan

- On encountering a `tabularx` environment, scan the input until the end of the environment.

The `re@l w@rk` begins

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

The plan

- On encountering a `tabularx` environment, scan the input until the end of the environment.
- Store the contents of the environment somewhere for later use (in a **token register**).

The re@l w@rk begins

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

The plan

- On encountering a `tabularx` environment, scan the input until the end of the environment.
- Store the contents of the environment somewhere for later use (in a **token register**).
- Do several trial runs to determine the width of the `X` columns.

The `re@l w@rk` begins

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

The plan

- On encountering a `tabularx` environment, scan the input until the end of the environment.
- Store the contents of the environment somewhere for later use (in a **token register**).
- Do several trial runs to determine the width of the `X` columns.
- Typeset the environment with the computed column widths.

Definiti@ns with \def?

- In LaTeX, new commands are defined using `\newcommand`.
- `\newcommand` is defined in the LaTeX kernel.
- `\def` is the TeX **primitive** to define new commands.
- `\newcommand` uses `\def` internally.

Definiti@ns with \def?

- In LaTeX, new commands are defined using \newcommand.
- \newcommand is defined in the LaTeX kernel.
- \def is the TeX **primitive** to define new commands.
- \newcommand uses \def internally.

The definition

```
\newcommand{\MyCommand}[n]{replacement text using #1 to #n}
```

corresponds (more or less) to

```
\def\MyCommand#1#2...#n{replacement text using #1 to #n}
```

Definiti@ns with \def?

- In LaTeX, new commands are defined using \newcommand.
- \newcommand is defined in the LaTeX kernel.
- \def is the TeX **primitive** to define new commands.
- \newcommand uses \def internally.

The definition

```
\newcommand{\MyCommand}[n]{replacement text using #1 to #n}
```

corresponds (more or less) to

```
\def\MyCommand#1#2...#n{replacement text using #1 to #n}
```

Therefore

```
\def\tabularx#1{%
```

is not much different from

```
\newcommand{\tabularx}[1]{%
```

A brief `l@@k` at the definition of `\newcommand`

```
\def\newcommand{\@star@or@long\new@command}
\def\new@command#1{%
  \@testopt{\@newcommand#1}0}
\def\@newcommand#1[#2]{%
  \@ifnextchar [{\@xargdef#1[#2]}{\@argdef#1[#2]}}
\long\def\@argdef#1[#2]#3{%
   \@ifdefinable #1{\@yargdef#1\@ne{#2}{#3}}}
\long \def \@yargdef #1#2#3{%
  \ifx#2\tw@
    \def\reserved@b##11{[####1]}%
  \else
    \let\reserved@b\@gobble
  \fi
  \expandafter
    \@yargd@f \expandafter{\number #3}#1}
\long \def \@yargd@f#1#2{%
  \def \reserved@a ##1#1##2##{%
    \expandafter\def\expandafter#2\reserved@b ##1#1}%
  \l@ngrel@x \reserved@a 0##1##2##3##4##5##6##7##8##9###1}
```

Who can find the relevant occurrence of `\def`?

A brief l@@k at the definition of \newcommand

```
\def\newcommand{\@star@or@long\new@command}
\def\new@command#1{%
  \@testopt{\@newcommand#1}0}
\def\@newcommand#1[#2]{%
  \@ifnextchar [{\@xargdef#1[#2]}{\@argdef#1[#2]}}
\long\def\@argdef#1[#2]#3{%
   \@ifdefinable #1{\@yargdef#1\@ne{#2}{#3}}}
\long \def \@yargdef #1#2#3{%
  \ifx#2\tw@
    \def\reserved@b##11{[####1]}%
  \else
    \let\reserved@b\@gobble
  \fi
  \expandafter
    \@yargd@f \expandafter{\number #3}#1}
\long \def \@yargd@f#1#2{%
  \def \reserved@a ##1#1##2##{%
    \expandafter\def\expandafter#2\reserved@b ##1#1}%
  \l@ngrel@x \reserved@a 0##1##2##3##4##5##6##7##8##9###1}
```

Who can find the relevant occurrence of \def? There it is!

Where to find `th@t`?

Where to find `th@t`?

Look in the sources

If you know approximately where the command comes from, you can check the sources directly. For instance, `\newcommand` is defined in the file `ltdefns.dtx`. One can also look in the typeset version of the kernel sources, `source2e.ps`.

Where to find th@t?

Look in the sources

If you know approximately where the command comes from, you can check the sources directly. For instance, `\newcommand` is defined in the file `ltdefns.dtx`. One can also look in the typeset version of the kernel sources, `source2e.ps`.

Use `\show`

TEX provides the primitive command `\show`. You can ask for the definition of a command with `\show` (but you won't get any documentation).

Envir@nments in LaTeX

- Similar to command definitions, LaTeX knows **environments**.
- Environments can be defined with

```
\newenvironment{name}{code at beginning}{code at end}
```

- They can be used in blocks of the form

```
\begin{name}
[...]
\end{name}
```

Envir@nments in LaTeX

- Similar to command definitions, LaTeX knows **environments**.
- Environments can be defined with

```
\newenvironment{name}{code at beginning}{code at end}
```

- They can be used in blocks of the form

```
\begin{name}
[...]
\end{name}
```

- Internally, `\newenvironment{name}` defines two commands `\name` and `\endname`. These are executed by `\begin` and `\end`.

Envir@nments in LaTeX

- Similar to command definitions, LaTeX knows **environments**.
- Environments can be defined with

```
\newenvironment{name}{code at beginning}{code at end}
```

- They can be used in blocks of the form

```
\begin{name}
[...]
\end{name}
```

- Internally, `\newenvironment{name}` defines two commands `\name` and `\endname`. These are executed by `\begin` and `\end`.
- The environment we are currently in is always available via the internal LaTeX command `\@currenvir`.

Envir@nments in LaTeX

- Similar to command definitions, LaTeX knows **environments**.
- Environments can be defined with

```
\newenvironment{name}{code at beginning}{code at end}
```

- They can be used in blocks of the form

```
\begin{name}
[...]
\end{name}
```

- Internally, `\newenvironment{name}` defines two commands `\name` and `\endname`. These are executed by `\begin` and `\end`.
- The environment we are currently in is always available via the internal LaTeX command `\@currenvir`.

Now let's have another look at the `tabularx` sources ...

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- With `\tabularx`, the beginning of the `tabularx` environment is defined!

Another l@@k

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- This occurrence of `\endtabularx` has oviously something to do with the end of the environment (although we don’t know yet what `\let` does).

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- Here, the name of the current environment is stored (although we don’t know yet what `\edef` does).

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=`}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- This is completely strange (and, in fact, a very dirty trick which is described in the TEXbook).

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- The `tabularx` environment gets as argument the desired total width of the table. This width is stored in a dimension register.

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- We print something to the log, using a command later defined in `tabularx.sty`.

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- Here, we initialise a globally predefined token register (it is defined in the LaTeX kernel) to the empty token list.

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

- ➔ This calls the command that continues the work.

Another `l@@k`

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We know a bit more now:

But first more about `\def`, `\edef`, and `\let` ...

About m@cros and expansion

An example

```
\begin{enumerate}
\item We save the current environment in a new command.
\newcommand{\envsave}{\@currenvir}

\item And now print it:
  \begin{center}
  \envsave
  \end{center}
\end{enumerate}
```

Question: What do think will be the result?

About m@cros and expansion

An example

```
\begin{enumerate}
\item We save the current environment in a new command.
\newcommand{\envsave}{\@currenvir}

\item And now print it:
  \begin{center}
  \envsave
  \end{center}
\end{enumerate}
```

Question: What do think will be the result? **Answer:**

1. We save the current environment in a new command.
2. And now print it:

center

About m@cros and expansion

An example

```
\begin{enumerate}
\item We save the current environment in a new command.
\def\envsave{\@currenvir}

\item And now print it:
  \begin{center}
  \envsave
  \end{center}
\end{enumerate}
```

Replacing \newcommand by \def does not change anything:

1. We save the current environment in a new command.
2. And now print it:

center

About `m@cros` and expansion

An example

```
\begin{enumerate}
\item We save the current environment in a new command.
\edef\envsave{\@currenvir}

\item And now print it:
  \begin{center}
  \envsave
  \end{center}
\end{enumerate}
```

But using `\edef` **does** change a lot:

1. We save the current environment in a new command.
2. And now print it:

enumerate

To expand or `n@t` to expand

- `\def` defines a **macro**. The replacement text is stored **as is**, and inserted at the position where the macro is called.
- This replacement is called **expanding** the macro.
- `\edef\`*`name`* **first** expands its argument as completely as possible (say, to *`result`* and the defines `\`*`name`* to expand to *`result`* directly.

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
\MyCommand
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: \MyCommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: \Mycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: \Mycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: \Mycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: \Mycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: \Mycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: More:  \Mycommand\xspace\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: More: \Mycommand\xspace\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: More: More:  \Mycommand\xspace\xspace\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: More: More: \Mycommand\xspace\xspace\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: More: More: More:  \Mycommand\xspace\xspace\xspace\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 1: Naïve redefinition

```
\newcommand{\MyCommand}{something}
\renewcommand{\MyCommand}{More: \MyCommand\xspace}
```

Won't work! This is what happens if \MyCommand is expanded:

```
More: More: More: More:  \Mycommand\xspace\xspace\xspace\xspace
```

TEX runs out of memory ...

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 2: Using `\edef`

Will work sometimes, but not in general. Assume

```
\newcommand{\MyCommand}{%
  \addtocounter{equation}{1} and some text}
\edef{\MyCommand}{More: \MyCommand\xspace}
```

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 2: Using `\edef`

Will work sometimes, but not in general. Assume

```
\newcommand{\MyCommand}{%
  \addtocounter{equation}{1} and some text}
\edef{\MyCommand}{More: \MyCommand\xspace}
```

The execution of `\edef` will fail because `\addtocounter` (being an assignment to a register) cannot (and should not) be completely expanded.

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using `\let`

With `\let`, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using `\let`

With `\let`, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
\MyCommand
```

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using `\let`

With `\let`, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  \OldMyCommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using \let

With \let, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  \OldMycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using `\let`

With `\let`, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  \OldMycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using \let

With \let, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  \OldMycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take `\renewcommand` (or just use `\def` again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using `\let`

With `\let`, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  \OldMycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using \let

With \let, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  \OldMycommand\xspace
```

H@w to redefine a command

- Easy, you say! Take \renewcommand (or just use \def again).
- But what if we want to use the old meaning (i.e. expansion) of the command in defining the new?

Attempt 3: Using \let

With \let, we can introduce an alias for the current expansion of a command (which is exactly what we need here).

```
\newcommand{\MyCommand}{something}
\let\OldMyCommand\MyCommand % save meaning of MyCommand
\renewcommand{\MyCommand}{%
  More: \OldMyCommand\xspace} % use saved meaning
```

Now the expansion runs as expected:

```
More:  something\xspace
```

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

- At the beginning of a `tabularx` environment, we first save the current environment name in macro `TX@`. Isn't that always `tabularx`? Not necessarily (exercise).

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=`}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

- ➔ This was the dirty trick. Let's just say that it opens a group.

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

- ➔ We store the target length in a register for further use.

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

- We print some debugging information.

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

- We initialise a token register to the empty list and continue with `\TX@get@body`.

Yet an@ther look

```
\def\tabularx#1{%
\edef\TX@{\@currenvir}%
  {\ifnum0=‘}\fi
  \setlength\TX@target{#1}%
  \TX@typeout{Target width: #1 = \the\TX@target.}%
  \toks@{}\TX@get@body}

\let\endtabularx\relax
```

We can almost completely understand the code now:

- We let the end-part of the two environment-related macros mean the same as `\relax`, which is a TEX primitive that does (almost) nothing.

Sc@nning the contents

The next thing that is accomplished in the tabularx style is that the contents of the environment are scanned and saved.

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

Again, first the plan

- We scan until the next occurrence of \end in the input.
- We add the tokens to the register so far.
- If the \end ends the tabularx, we are done and can then try to typeset the table.
- If the \end ends some other (nested) environment, we have to repeat the procedure.

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- \long modifies \def; arguments to the defined macro may then span multiple paragraphs
- \newcommand always uses \long\def, but \newcommand* uses plain \def.

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- \def has another interesting feature: You can specify delimiters for macro arguments.

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- \def has another interesting feature: You can specify delimiters for macro arguments.
- Normally, an argument is either grouped with { and } or consists just of a single token. Here, it extends until the next \end in the input stream.

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- \def has another interesting feature: You can specify delimiters for macro arguments.
- Normally, an argument is either grouped with { and } or consists just of a single token. Here, it extends until the next \end in the input stream.
- The \end itself will not be part of the argument.

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- For multiple arguments one can have multiple delimiters.

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- For multiple arguments one can have multiple delimiters.

```
\def\MyRemark#1.#2\End{%
  \noindent\textbf{#1.}\quad #2 \hfill$\bullet$\par}

\MyRemark Nota bene. \TeX's macro definition construct
  is extremely powerful.\End
```

will result in

Nota bene. TEX's macro definition construct is extremely powerful. •

M@re about \def

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- **In normal documents, use of \def is discouraged. \newcommand is safer in many ways, and using delimiters will decrease readability, especially for other readers.**

More about exp@nsi@n

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

More about `exp@nsi@n`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- `\toks@` is a token register. We already know that.

More about `exp@nsi@n`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- `\toks@` is a token register. We already know that.
- We can assign something to that token register by saying

  `\toks@{`*something*`}`

  The tokens in *something* are then saved in the register.

More about `exp@nsi@n`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- `\toks@` is a token register. We already know that.
- We can assign something to that token register by saying

  ```
  \toks@{something}
  ```

  The tokens in *something* are then saved in the register.
- The contents of a token register can be used with

  ```
  \the\toks@
  ```

  (In fact, also other register contents can be used this way, with the help of `\the`.)

More about exp@nsi@n

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- If we just look at the highlighted part, we can read off the intention.

More about `exp@nsi@n`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- If we just look at the highlighted part, we can read off the intention.
- The macro argument `#1` (i.e. all the tokens until the next `\end`) should be appended to the previous contents of `\toks@`.

More about exp@nsi@n

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
```

- If we just look at the highlighted part, we can read off the intention.
- The macro argument #1 (i.e. all the tokens until the next \end) should be appended to the previous contents of \toks@.
- But we have an expansion problem again, similar to the situation where we needed \edef. But \edef is not an option here, because there is no definition. But \expandafter helps ...

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- ➔ `\toks@` is processed by TEX.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- \toks@ is processed by TEX.
- Being recognised as a token register, TEX expects a { next to start a token list.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- ➔ `\expandafter` isn't a `{`. Therefore TEX starts expanding.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- `\expandafter` isn't a `{`. Therefore TEX starts expanding.
- `\expandafter` tells TEX to skip the next token, expand the following token **once**, then continue with the skipped token.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- The { is skipped (but not discarded!) because of \expandafter.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- The \the is expanded.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- The \the is expanded.
- The primitive command \the expects a register next, so it looks ahead.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@\expandafter{\the\toks@ brand new}
```

- The register \toks@ is found, so \the\toks@ expands to the current contents of \toks@.

Ch@nging the order of expansion

Let us investigate how the assignment to the token register is processed by TEX:

```
\toks@{terribly old brand new}
```

- The register \toks@ is found, so \the\toks@ expands to the current contents of \toks@.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- We want to check if we have already found the `\end` that ends the `tabularx` environment.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- We consume the **next** argument from the input stream. After the end we expect the name of an environment.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- We consume the **next** argument from the input stream. After the end we expect the name of an environment.
- This name is stored in a temporary command `\@tempa`.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- The primitive command `\ifx` **compares** the two commands `\@tempa` and `\TX@`.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- The primitive command `\ifx` **compares** the two commands `\@tempa` and `\TX@`.
- If they have the same expansion, everything up to the next `\else` is executed, i. e. we call yet another command `\TX@endtabularx`.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- The primitive command `\ifx` **compares** the two commands `\@tempa` and `\TX@`.
- If they are not equal in that sense, everything between the `\else` and the `\fi` is executed ...

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- The primitive command `\ifx` **compares** the two commands `\@tempa` and `\TX@`.
- If they are not equal in that sense, everything between the `\else` and the `\fi` is executed ... We first add the `\end` command for the other environment to the token register.

Finding the `end@`

```
\long\def\TX@get@body#1\end
  {\toks@\expandafter{\the\toks@#1}\TX@find@end}
\def\TX@find@end#1{%
  \def\@tempa{#1}%
  \ifx\@tempa\TX@\expandafter\TX@endtabularx
  \else\toks@\expandafter
    {\the\toks@\end{#1}}\expandafter\TX@get@body\fi}
```

- The primitive command `\ifx` **compares** the two commands `\@tempa` and `\TX@`.
- If they are not equal in that sense, everything between the `\else` and the `\fi` is executed ... We first add the `\end` command for the other environment to the token register. The we loop and scan to the next `\end` (the `\expandafter` gets rid of the `\fi`).

C@nditionals in TEX

TEX knows a family of conditional operators that all work the same way:

```
\ifsomething condition
  things to be done if true
\else
  things to be done if false
\fi
```

C@nditionals in TEX

TEX knows a family of conditional operators that all work the same way:

```
\ifsomething condition
  things to be done if true
\else
  things to be done if false
\fi
```

Some of them are:

C@nditionals in TEX

TEX knows a family of conditional operators that all work the same way:

```
\ifsomething condition
  things to be done if true
\else
  things to be done if false
\fi
```

Some of them are:

- `\ifx` compares the (one-step) expansions of two commands

C@nditionals in TEX

TEX knows a family of conditional operators that all work the same way:

```
\ifsomething condition
  things to be done if true
\else
  things to be done if false
\fi
```

Some of them are:

- `\ifx` compares the (one-step) expansions of two commands
- `\ifnum` compares two integers

C@nditionals in TEX

TEX knows a family of conditional operators that all work the same way:

```
\ifsomething condition
  things to be done if true
\else
  things to be done if false
\fi
```

Some of them are:

- `\ifx` compares the (one-step) expansions of two commands
- `\ifnum` compares two integers
- `\ifdim` compares two dimensions

C@nditionals in TEX

TEX knows a family of conditional operators that all work the same way:

```
\ifsomething condition
  things to be done if true
\else
  things to be done if false
\fi
```

Some of them are:

- `\ifx` compares the (one-step) expansions of two commands
- `\ifnum` compares two integers
- `\ifdim` compares two dimensions
- `\if` compares two characters

Flying thr@ugh the rest

```
\def\TX@endtabularx{%
  \expandafter\TX@newcol\expandafter{\tabularxcolumn{\TX@col@width}}%
  \let\verb\TX@verb
  \def\@elt##1{\global\value{##1}\the\value{##1}\relax}%
  \edef\TX@ckpt{\cl@@ckpt}%
  \let\@elt\relax
  \TX@old@table\maxdimen
  \TX@col@width\TX@target
  \global\TX@cols\@ne
  \TX@typeout@
    {\@spaces Table Width\@spaces Column Width\@spaces X Columns}%
  \TX@trial{\def\NC@rewrite@X{%
          \global\advance\TX@cols\@ne\NC@find p{\TX@col@width}}}%
[...]
```

Flying thr@ugh the rest

```
[...]
  \loop
    \TX@arith
    \ifTX@
    \TX@trial{}%
  \repeat
  {\let\@footnotetext\TX@ftntext\let\@xfootnotenext\TX@xftntext
    \csname tabular*\expandafter\endcsname\expandafter\TX@target
      \the\toks@
    \csname endtabular*\endcsname}%
  \global\TX@ftn\expandafter{\expandafter}\the\TX@ftn
  \ifnum0=‘{\fi}%
  \expandafter\end\expandafter{\TX@}}
```

Conclusions

Conclusions

If you want to, you can

Conclusions

If you want to, you can
– but you don't have to . . .