Modular Casebook Management: modbook.sty

Charles Duan
cduan@wcl.american.edu

Version v1.1.0, 2025/12/11

Contents

1 Introduction

This is a package for managing the compilation of a textbook made up of several interdependent modules. The purpose of this package is:

- To manage cross-dependencies between parts of the textbook. For example, one part may reference a case that should have been already read in the book, so it should be possible to raise a warning if that case hasn't already been included.
- To provide formatting for standard parts of a casebook.
- To permit local alterations to casebook files. This requires devising a directory structure for local files, such that including a file searches first for the local copy and then for default version.
- To provide ancillary packages for deferred content and multiple choice question formatting, described below, which may be used independently of this package.

The general workflow model assumed by this package is as follows. Textbooks are to be compiled out of cases, articles, and other materials described in this documentation as *readings*. One or more *editors* compile, edit, and annotate these readings, and perhaps write editorial material of their own. The editors arrange their work into *modules*, each of which contains an outline and content files including readings. A *compiler* then receives modules and arranges them into a book. The compiler may also wish to make changes to the editors' work.

2 Installation and Dependencies

The package uses hyperref for internal cross-references, and the strings package from hicite. It also requires the graphicx and etoolbox packages.

3 Repositories

This package uses a rigorous file hierarchy to manage the work of editors and compilers. At the base of the hierarchy are *repositories* of content. Generally an editor (or team of editors) would be responsible for a single repository. Within each repository are *modules* contained in subfolders.

`\RepositoryPath` `{`⟨*path,…*⟩`}`

Provides a list of repositories. The package will sequentially search through each repository given in the argument, which should be a comma-separated list, until it finds the module file required. The default path is `local,base`.

`\repodir` `{`⟨*repository*⟩`}{`⟨*directory*⟩`}`

Allows for aliasing of repository directories. By default, a repository is found in a directory with the same name. This macro can be used to change the directory for a repository.

`\mbk@repo` `{`⟨*repository*⟩`}`

Expands to the directory for the given repository, either given by `\repodir` or the repository name itself.

4 Modules

Modules contain content files consistent with the following rules:

- ⟨*module*⟩`/`⟨*module*⟩`.tex`: The default outline of the module. This file should contain only section headings, `question` environments, and `\import` commands.
- ⟨*module*⟩`/intro-`⟨*module*⟩`.tex`: The introductory text for a module should by convention have this name, making it convenient to determine whether a module has been imported into a book.[1]
- ⟨*module*⟩`/intro-`⟨*filename*⟩`.tex`: A file containing editorial or introductory text (that is, text not part of a case or other reading).
- ⟨*module*⟩`/narrative-`⟨*filename*⟩`.tex`: The same as an `intro-` file. (Starting a filename with `narrative-` can be used to indicate that the text is intended as a full standalone section, rather than as an introduction to another reading.)
- ⟨*module*⟩`/`⟨*filename*⟩`-qs.tex`: A list of questions and notes that may follow a reading. By convention, ⟨*filename*⟩ corresponds to the file to which the questions apply. A question file will be included inside a list-like environment, so items should begin with `\item`.
- ⟨*module*⟩`/`⟨*filename*⟩`.tex`: Any other filename is assumed to be a reading from an external source (which should start with `\reading`).

4.1 Importing Modules

`\module` ⟨*name*⟩

Retrieves content for a module. This command should be the primary one that compilers use when assembling a book. It is functionally equivalent to `\import` ⟨*name*⟩`/`⟨*name*⟩.

[1] It is generally inadvisable to check if the ⟨*module*⟩`/`⟨*module*⟩`.tex` file itself has been imported, because compilers will often not use the default outline when selecting parts of modules.

`\import` ⟨*name*⟩

The `\import` command retrieves content from a file within a module. The macro takes one argument, which need not be surrounded by braces, similar to `\input`. The argument may be:

1. A content file without a module name. The file is assumed to be within the last-`\import`ed module.
2. A content file with a module name (⟨*module*⟩`/`⟨*filename*⟩).

`\ModuleEndHook` When importing a complete module, the contents of this macro will be executed at the end of reading the module. By default, this is used to execute `\showquestions`, but other features may be implemented.

4.2 Editing Modules

It may be desirable to edit a file in a module from this repository or another repository, rather than making a copy of the file. The following commands enable this.

Note that these commands do not `\import` the selected file, so the processes of registration and tracking will not take place. This is appropriate because, as a general matter, these editing commands will take place in the context of another file being `\import`ed.

`\editrepofile` `{`⟨*repo*⟩`}{`⟨*mod*⟩`}{`⟨*file*⟩`}`

Edits a specific file located in a given repository and module. This offers precise control over the file to be edited.

`\editfromrepo` `{`⟨*repo*⟩`}` ⟨*instructions*⟩ `\endedit`

Edits a file of the same name and module as the current file, from the given repository. The ⟨*instructions*⟩ are the same as those for `\editfile` in the `editfile` package.

`\editmodfile` `{`⟨*filename*⟩`}` ⟨*instructions*⟩ `\endedit`

Edits a file within the current repository and module. (This is useful, for example, for editing a case while retaining the original case's full text.)

4.3 Cross-Reference Expectations

Content in modules will often cross-reference material in other modules. But if the compiler can select and reorder the modules, these cross-references will become

unanchored. The package thus provides several macros to manage cross-references. Editors should insert these macros into their module files as they write, enabling their modules and files to be rearranged without creating contextual problems.

Expectations are defined based on filenames, and are met if a corresponding file has been `\import`ed into the book at the correct time. Filenames may be given with the module name or without. (The best practice, then, is to ensure that filenames are unique even across different modules.)

Failures of any of the expectation assertions below will result in a warning and an undefined-reference warning at the end of document compilation.

See section 7.4 for the implementation of cross-reference checking, in particular the actions taken for recording information as modules are imported.

`\having` `{`⟨*filename*⟩`}` `{`⟨*before*⟩`}` `{`⟨*after*⟩`}` `{`⟨*none*⟩`}`

Chooses a text depending on the inclusion status of ⟨*filename*⟩. If the file has already been `\import`ed, then ⟨*before*⟩ is used. If the file is imported later, then ⟨*after*⟩ is used. If the file is never imported, then ⟨*none*⟩ is used.

This macro best enables flexibility for compilers, and should be used in preference to the other expectation assertion macros to the extent possible.

`\expected` `{`⟨*filename*⟩`}`

Tests whether a file has been included already, and produces a warning if not.

`\expecting` `{`⟨*filename*⟩`}`

Tests whether a file will later be included. The test fails if the file is never included, or if the file was included before this command was called. Because it relies on the `.aux` file, this command may produce spurious warnings that go away on subsequent compilations.

`\expectnext` `{`⟨*filename*⟩`}`

Indicates that the next imported file should match this filename. This is used, for example, at the end of an introductory text intended to precede a reading.

5 Key Terms

Content in a book often introduces new terms and concepts. Keeping track of those terms is useful for several reasons. Key terms in the book ought to be formatted consistently. They also are useful as the basis of an index.

Additionally, key terms serve as a mechanism for checking the order of modules. Presumably, if a key term is used in a chapter, it ought to have been defined and explained earlier in the book. Thus, commands for tracking the places where a key

term is defined and used serve as an alternative to the cross-reference management features described above.

`\term` `[`⟨*name*⟩`]{`⟨*text*⟩`}`

Indicates that a key term is being defined at or near this location. The ⟨*text*⟩ is emitted, formatted in accordance with `\FormatTerm`. The term is also indexed in accordance with `\IndexTerm`, using ⟨*name*⟩ if given, or ⟨*text*⟩ if not.

If a term is to be indexed but not displayed, make ⟨*text*⟩ empty and fill only the ⟨*name*⟩ argument. (There is no need for the converse; to format a term without indexing it, just use `\FormatTerm` directly.)

`\FormatTerm` `{`⟨*term*⟩`}` Formats the term for text. By default, it is put in bold.

`\IndexTerm` `{`⟨*term*⟩`}`

Performs any custom indexing of the term. Initially, this macro does nothing.

`\withterm` `{`⟨*term*⟩`}{`⟨*do-with*⟩`}{`⟨*do-without*⟩`}`

Tests whether ⟨*term*⟩ has been defined in the book (using the `\term` command). If so, then executes the contents of ⟨*do-with*⟩. Otherwise, executes the contents of ⟨*do-without*⟩.

`\checkterm` `{`⟨*term*⟩`}` Produces a warning if the term has not been defined so far. (TODO: This should perhaps also index the term.)

`\useterm` `{`⟨*term*⟩`}` Runs `\checkterm` on the term, and also emits it (as plain text).

6 Formatting Content

Casebooks generally use only a few types of materials for readings, and also include common types of editorial content. The macros here help with formatting these elements consistently.

6.1 Readings

These commands are useful for formatting a reading from a case or other materials. Typical usage is as follows:

```
\readingnote{Decided on the same day as Bolling v. Sharpe, 347
U.S. 497 (1954).}
\reading{Brown v. Board of Education}
\readingcite{347 U.S. 483 (1954)}
```

```
\opinion \textsc{Mr. Chief Justice Warren} delivered the opinion
of the Court.

These cases come to us from the States of Kansas, South Carolina,
Virginia, and Delaware…
```

\readingnote {⟨*note-text*⟩}

Adds a footnote to the reading's heading. This command *must come before* the \reading command.

\reading [⟨*short-name*⟩] {⟨*name*⟩}

Creates a section heading starting a reading. The title of the reading is given as ⟨*name*⟩, and a short Table of Contents version may be given as ⟨*short-name*⟩.

As a convenience, if ⟨*name*⟩ starts with *In re* or contains *v.*, the name (and short name) will automatically be italicized for being a case name.

Readings are treated as a form of LaTeX section,using \@startsection. They have table of contents level 4.

\readingcite {⟨*citation*⟩}

Produces a second heading line below a \reading entry, that gives the citation for the reading text.

\opinion {⟨*text*⟩} \par

Formats the line where the opinion author is given. The argument need not be in braces; it is terminated at the end of the paragraph.

\readinghead [⟨*level*⟩] {⟨*text*⟩}

Creates a heading inside a reading. The ⟨*level*⟩ is a heading level, by default 1.

\ReadingHeadFont {⟨*level*⟩} {⟨*commands*⟩}

Provides font and formatting ⟨*commands*⟩ associated with a heading ⟨*level*⟩ for \readinghead. The commands will be placed inside a group, and the text to be formatted will be given as an argument to the commands.

By default, reading heading level 1 is bold, and level 2 is bold italic.

6.2 Statute and Question Environments

`statute` Formats text for an indented statute's subsections. Statutes are typically formatted as indented paragraphs, with higher levels of indentation pushing the right margin but retaining the indentation structure. (Statutes are typically not formatted with hanging indentation.)

This environment provides for such indentation, for the second and higher levels. (The first level is simply normal paragraph indentation and thus requires no environment.) Each paragraph should be preceded by an `\item` command.

(This environment is currently not very well tested and ought to be improved.)

`questions` `[`⟨*title*⟩`]`

Creates an environment for notes and questions. The title of the environment is by default "Notes and Questions," and may be changed with the optional argument. If the optional argument is empty, no heading is produced.

The contents of the environment should be a list with `\item` commands.

6.3 Fonts

Two fonts are used throughout the casebook, one for editorial materials and one for readings. The following rules are used to distinguish the two:

- Files starting with `intro-` or `narrative-`, or files ending with `-qs`, are editorial material; anything else is a reading.
- Block quotes are always assumed to be readings.
- Footnotes follow their own rules, described below.

`\edfont` `\readingfont` The fonts may also be manually selected, with the commands `\edfont` (for editorial material) and `\readingfont` (for readings). Note that the `\readings` command *does not apply the reading font*. This is because some editors like to include Notes or other materials with a reading-like heading. Such material should be included in an editorially-named file (`narrative-`⟨*file*⟩`.tex`), and it will be set in the editorial font.

`\EditorialFont` `{`⟨*font-commands*⟩`}`

Executes ⟨*font-commands*⟩ for any editorial material. By default, editorial material is set in a sans serif font.

`\ReadingFont` `{`⟨*font-commands*⟩`}`

Executes ⟨*font-commands*⟩ for any reading material. By default, reading material is set in a serif font.

`\ifedmaterial` A conditional for determining whether the current text is a reading or editorial material.

`\HeaderFonts` `{`⟨*pagenum*⟩`}` `{`⟨*left*⟩`}` `{`⟨*right*⟩`}`

Sets the fonts for the running header. As described below in the Implementation section, the running header consists of a chapter name on the left and the current reading on the right. By default, sans serif fonts are used, with the chapter set in small caps.

`\SectionFont` `{`⟨*font*⟩`}`

Sets the fonts for section headings. By default, the roman font is used regardless of editorial or reading context. The ⟨*font*⟩ should require no arguments.

6.4 Footnotes

In editorial material, footnotes are assumed to also be editorial material. Small changes to the `\footnote` command must be made to accommodate this.

Footnotes in readings are more complex. Sometimes the footnote is from the original reading, and should retain the original footnote number. In other cases, the reading's editor adds an explanatory footnote. Editorial footnotes are identified with a different footnote symbol, the editorial font, and a notation. Given these two types of footnotes, the usual `\footnote` command is disallowed in reading text, in favor of two separate commands described below.

As a convenience, all footnote commands add anbefore them, so spaces before the footnote are ignored.

`\readingfootnote` `{`⟨*number*⟩`}` `{`⟨*note-text*⟩`}`

Creates a footnote from the original reading in the text. The footnote must be given a number corresponding to the original footnote number in the source document.

`\edfootnote` `{`⟨*note-text*⟩`}`

Creates a footnote by the editors. Symbolic footnote marks are used, and a separate counter `edfnct` is created to track the marks.

`\EditorMark` `{`⟨*note-text*⟩`}`

Transforms ⟨*note-text*⟩ with an indication that the text originated from an editor. By default, this just appends "—Eds.", and the macro can be redefined as desired.

`\footnote` `{`⟨*note-text*⟩`}`

Regular footnotes can only be used in editorial material, and are editorial material themselves.

6.5 Graphics

The following commands are provided for inclusion of graphics. Graphics files may be located either in a module directory or in a separate `images` directory in a repository. The extension `.png`, `.jpg`, or `.pdf` may be omitted from the filename.

`\usegraphic` `[`⟨*options*⟩`]` `{`⟨*filename*⟩`}`

Include a graphic in the current position inline with text. The ⟨*options*⟩ are those options available for the `\includegraphics` command of the graphicx package.

By default, graphics take up a maximum of 30% of the text height and 80% of the text width. The optional argument to any of the graphics inclusion macros can change that.

`\heregraphic` `[`⟨*options*⟩`]` `{`⟨*filename*⟩`}`

Centers the graphic at the current position in the text.

`\captionedgraphic` `[`⟨*options*⟩`]` `{`⟨*filename*⟩`}` `{`⟨*caption*⟩`}`

Places the graphic in a floating figure with a caption. A cross-reference label of `f:`⟨*filename*⟩ is automatically attached to the figure number.

Because captions are always editorial material (unless specified otherwise), they are displayed in the editorial font.

`\GraphicsDirectory` `{`⟨*directory*⟩`}`

Specifies the directory within repositories where images may be found. By default, it is `images`.

6.6 Multiple Choice Questions

The `multichoice` question is included in this package, with question deferral turned on. This enables adding multiple choice questions throughout the text, with all the questions for a chapter being displayed at the end for example.

Questions will be displayed in a manner consistent with the rest of the book.

7 Implementation

The remaining text describes internal operations of the package, and need not be read unless the package is doing something unexpected and you want to fix it.

7.1 Finding Module Files

`\mbk@import` `{`⟨*module*⟩`}` `{`⟨*filename*⟩`}`

Imports a file in a module, by trying all the repos in the path. If the file is not found, then issue an error message.

If the file is found, the following actions are be taken to cause the file importation:

- The file is registered for cross-reference checking (section 7.4)
- The module is added to the stack of modules (section 7.2)
- The appropriate font is selected (section 7.5)
- The file is read into input
- Any `\ModuleEndHook` is executed
- The module is removed from the stack

`\mbk@import@frompath` ⟨*repo*⟩,⟨*rest-of-path*⟩…`\@nil` `{`⟨*module*⟩`}` `{`⟨*filename*⟩`}`

Tries to find a file ⟨*repo*⟩/⟨*module*⟩/⟨*filename*⟩. If this fails, continue trying with ⟨*rest-of-path*⟩ if present; otherwise fall through to the default condition previously set. This uses the `\mbk@try@file` mechanism.

7.2 Tracking the Current Module and File

Module imports are tracked via a stack, so it is always possible to know which module is in current use. (The normal TeX grouping mechanism cannot be used, because otherwise content would be included inside groups.)

`\mbk@current@mod` The current module.

`\mbk@current@file` The current file.

`\mbk@module@stack` The stack of module inclusions. The list is composed of triples of groups, the first being the repo, the second the module, and the third being the file. An empty repo/module/file triple is included at the end.

`\mbk@push` `{`⟨*repo*⟩`}` `{`⟨*module*⟩`}` `{`⟨*file*⟩`}`

Push a module onto the stack, and sets the current module and file.

`\mbk@module@pop` Delete a module from the stack. If one tries to pop the last module from the stack, this macro will generate an argument error (there won't be enough commas).

7.3 File Inclusion

For modules and images, there are often a couple of different possible filenames to be tried, and the first one that exists is to be used. The file inclusion command pattern below allows for doing this. The usage is:

```
\mbk@try@file{⟨filename⟩}{⟨content if file exists⟩}
\mbk@try@file{⟨filename⟩}{⟨content if file exists⟩}
...
\mbk@try@file@default{⟨content if no file exists⟩}
\mbk@try@file@end
```

Once a matching file is found, then all subsequent `\mbk@try@file` and other content up to `\mbk@try@file@end` is discarded. Note that this system does not actually include the found file; it is up to the ⟨*content*⟩ arguments to do that work.

`\mbk@try@file` `{`⟨*filename*⟩`}` `{`⟨*content*⟩`}`

Tries including a file among several. Several `\mbk@try@file` commands may be included in sequence, terminated with `\mbk@try@file@end`. If the file exists, then `{`⟨*content*⟩`}` is inserted and any other material up to `\mbk@try@file@end` will be discarded.

`\mbk@try@file@default` `{`⟨*content*⟩`}` `\mbk@try@file@end`

What to do if no `\mbk@try@file` commands succeed. All material up to `\mbk@try@file@end` is used.

7.4 Cross-Reference Recordation

The following macros manage recordation of information that is used for cross-reference checking. See section 4.3 for the user interface of assertions and commands used in modules to check cross-references.

`\mbk@register@file` `{`⟨*module*⟩`}` `{`⟨*filename*⟩`}`

To implement cross-reference checking, every file is "registered" at the time it is imported. The registration confirms any assertions that can be determined upon

registration, and records information for further checking. It also creates a LaTeX cross-reference (using `\ref`) named `mod:`⟨*filename*⟩.

Specifically, registration performs the following actions:

- Any `\expectnext` assertion is checked, and then reset.
- A check is performed to see if a file of the same name was already registered. If so, a warning is issued, since each file should be uniquely named across all modules.
- The `\ref` cross-reference is inserted.
- A macro is defined recording that the file was imported.
- Commands are writen to the aux file recording that the file was imported (to support `\expecting` commands).

`\mbk@register@pre` `{`⟨*filename*⟩`}`

Marks that a file will be included at a later time.

`\mbk@register@reading` `{`⟨*short-name*⟩`}`

Associates the reading's name with a LaTeX label, for cross-referencing purposes. The label will be named `modr:`⟨*current-file*⟩.

7.5 Fonts

`\@defaultfamilyhook` This code hooks into the LaTeX command that resets the default font, forcing editorial or reading font selection every time the font is reset.

`\mbk@formatting@for` `{`⟨*filename*⟩`}`

Selects the font based on the filename.

`quotation` `quote` Redefine the `quote` and `quotation` environments to use the reading font.

7.6 Document-Level Structure

The introduction of readings as a document section type requires some modifications to the usual LaTeX document structure.

First, the package creates a new running head format, where the chapter name is placed on the left and the current reading is placed on the right.

Page numbers on plain-styled pages (e.g., beginnings of chapters) are also modified accordingly.

When a chapter ends, a blank page may be inserted on the left side. Ensure that this page has no header.

Readings are section level 4, and paragraphs/subparagraphs are placed below that level. Numbering continues up through level 3 (i.e., readings are not numbered), and the Table of Contents includes readings.

8 Options

With the option `readingnotenums`, numbering will not contain chapter numbers and will be consecutive for each block of notes (as is traditional in casebooks).

With the option `chapternotenums`, notes will have the form ⟨*chapter*⟩.⟨*note*⟩ and will run consecutively through the chapter. So if the first reading of chapter 7 has four notes, then the first note of the second reading will be 7.5. This is probably better for cross-referencing notes throughout the book, since the notes are uniquely identified.

By default, `readingnotenums` is used.

9 Supporting Packages

9.1 Deferrals

Provides functions for deferring certain text to a later point in a document, with specified template formatting.

With this package, one defines one or more "deferral classes" by name. The user adds items to a deferral class, and then may use the deferral class by printing it out. Upon printing, the items of the deferral class can be altered by a per-item macro, and the deferral class can surround all the items with text such as environment `\begin` and `\end` commands.

`\NewDeferral` `{`⟨*class-name*⟩`}`

Creates a new deferred text class. #1 is the name of the class.

`\DeferralSurround` `{`⟨*class*⟩`}{`⟨*pre-text*⟩`}{`⟨*post-text*⟩`}`

Sets the text to be placed before and after the deferred text class when it is displayed.

`\DeferralMacro` `{`⟨*class*⟩`}{`⟨*macro-def*⟩`}`

Defines a macro for processing the deferred matter. The ⟨*macro-def*⟩ is the macro definition, which should use "#1" to reference each deferred item.

`\deferral` `{`⟨*class*⟩`}{`⟨*text*⟩`}`

Adds an item ⟨*text*⟩ to the deferral class.

`\UseDeferral` `{`⟨*class*⟩`}`

Uses the deferred matter. This will (1) print the class pre-text, (2) run the class macro on each item, and (3) print the class post-text. The deferred matter will then be cleared. If there are no deferred items for this class, then nothing is produced at all (not even the pre- and post-text).

9.2 Multiple Choice Questions

This package provides support for multiple choice question formatting. It depends on the deferral package. It may be used independently of `modbook`.

A multiple choice question consists of a question text, a list of choices, an answer, and an explanation. The answer and explanation are stored in deferrals, while the question is formatted as a list item, with the choices formatted within a list environment.

The package should be bundled with a script called `multichoice.rb`, which can be used to aid in writing multiple choice questions. To use the script, prepare a file of the form:

> `Question:` ⟨*question-text*⟩
> `A:` ⟨*choice-text*⟩
> `B:` ⟨*choice-text*⟩
> …
> `Answer:` ⟨*correct-choice*⟩`.` ⟨*explanation*⟩
> [*More questions may follow*]

The file should be TEX-formatted (and may use LATEX commands), with paragraphs separated by a blank line and starting with a word followed by a colon. Answer choices must be a single letter; the other permissible words before a comma are "Question" and "Answer". For example:

```
Question: What is 1 + 1?

A: 2

B: 15

C: 10 in binary

D: (A) and (C)

E: All of the above

Answer: D. The sum of one and one is typically represented with
the digit 2 in the decimal system, but is also 10 in binary as
specified in (C). Fifteen (B) is plainly incorrect.
```

The answer choices and the explanation may cross-reference other answer choices using letters in parentheses. Phrases like "all of the above" or "none of the above" will also be identified as cross-referencing other choices. If an answer choice cross-references other choices, it *must* follow other non-cross-referencing choices and *must only* cross-reference choices presented before it.

Running the `multichoice.rb` script on this file will:

- Randomize the order of the choices. The cross-referencing choices will not be rearranged, but the others will.
- Update the correct answer, and all the cross-references, to reflect the randomized order of the choices. Parenthesized letters in the choices and explanation are replaced, and phrases of the form "(A) and (B)" and "(C), (D), or (E)" are automatically sorted to reflect the new choice order.
- Format the question using the `\multichoiceq` commmand below.

The resulting output file can then be used as a LaTeX input file after the `multichoice` package is loaded.

`\multichoiceq` `{`⟨*question*⟩`}{`⟨*choices*⟩`}{`⟨*answer*⟩`} {`⟨*explanation*⟩`}`

Formats a multiple choice question. The assumption is that this macro is called within some sort of list, such that each question can be preceded with an `\item` command.

If the `deferqs` option was selected (see below), then questions will be placed in a deferral.

Saves the current question number, the correct answer, and the question to deferrals. Two deferrals are stored: an answer key that just contains the question number and correct answer, and an explanation key that also includes the explanation. #1 is the question number, #2 the correct answer, #3 the explanation.

`\showquestions` If questions are placed in a deferral, then this command will display the questions. By default, they will be placed in a `multichoice` environment. To change this, run `\DeferralSurround{mch@questions}`.

`\multichoiceitem` `{`⟨*label*⟩`}{`⟨*text*⟩`}`

Displays one of the choices to a multiple choice question. The assumption is that this occurs within a list environment (a choices environment in view of the definition of `\multichoiceq`).

The ⟨*label*⟩ should be the letter of the answer choice. It will be parenthesized by the macro.

`choices` An environment for multiple choice question choices. This environment will be invoked automatically by `\multichoiceq`.

`\multichoicea` `{`⟨*number*⟩`}{`⟨*answer*⟩`}{`⟨*explanation*⟩`}`

Display an answer. The assumption is that this is inside a list environment so `\item` can be used; the question number will be given as the item.

This macro will automatically be invoked by `\multichoiceq`, but it may be redefined to provide for alternate formatting of answers.

`\answerkey` Produce the answer key.

`\explanations` Produce the explanations.

`multichoice` An environment for questions and answers. This environment is not required for use; `\multichoiceq` will work with a standard enumerate environment. However, it provides formatting that generally looks preferable for text-heavy multiple choice questions.

The option `deferqs` will place all questions into a deferral for later printing. The option `immediateqs` will display questions immediately.

By default, questions are displayed immediately.

9.3 Editing Files

Consider a situation where a a file exists in a repository, and you want to make small edits to it. Typically, what you would do is make a copy of the file in a new repository and make changes to the copy. There are two drawbacks to the

copying approach, however. First, it makes the changes made opaque to to other readers (or you yourself, some time in the future): To find out what changes were made, the reader must run a comparison of the two files. Second, if the original repository's file is updated, then your copied and edited file becomes out of date.

As a result, it would be preferable if you could specify instructions for editing the file, which could be automatically applied to the original. This package provides that capability.

9.3.1 The Main Command

`\editfile` `{`⟨*filename*⟩`}` ⟨*instructions*⟩ `\endedit`

The `\editfile` command introduces editing for a file. The file is read into memory, and ⟨*instructions*⟩ are sequentially applied to it. The edited contents of the file are then placed on the stack for further expansion and execution.

Note that there is no "recursive" editing of files. However, you can edit other files' `\editfile` commands. Say file A runs `\editfile` on file B, and B itself contains an `\editfile` command to edit file C. It would be incorrect in A to include instructions to modify C, because C is not read into memory until A's instructions have been processed and executed. But A can edit B to add new instructions to B's `\editfile` command, and those added instructions will be processed on C.

`\edf@read` Reads the entire content of a file (previously opened) into a macro `\edf@content`.

9.3.2 Editing Instructions

These commands are only available within the context of `\editfile`.

`\atstart` `{`⟨*text*⟩`}` Adds text to the start of the file.

`\atend` `{`⟨*text*⟩`}` Adds text to the end of the file.

`\replacestart` `{`⟨*search*⟩`}{`⟨*replace*⟩`}` Searches for the given ⟨*search*⟩ text, and replaces everything from the start up to and including that text with ⟨*replace*⟩.

`\replaceend` `{`⟨*search*⟩`}{`⟨*replace*⟩`}` Searches for the given ⟨*search*⟩ text, and replaces everything from and including that text through the end of the file with ⟨*replace*⟩.

`\replacerange` `{`⟨*start*⟩`}{`⟨*end*⟩`}{`⟨*replace*⟩`}` Searches for a range of text from ⟨*start*⟩ to ⟨*end*⟩, deletes it (including the search strings), and replaces it with ⟨*replace*⟩.

`\replace` `{`⟨*search*⟩`}{`⟨*replace*⟩`}` Searches for text and replaces it. The search/replace is done only once.

`\insert` `{`⟨*search*⟩`}{`⟨*text*⟩`}` Inserts ⟨*text*⟩ before `{`⟨*search*⟩`}`.

`\append` `{`⟨*search*⟩`}{`⟨*text*⟩`}` After `{`⟨*search*⟩`}`, appends ⟨*text*⟩.

`\delete` `{`⟨*search*⟩`}` Deletes the given text. This is equivalent to `\replace` with an empty replacement argument.

`\deleterange` `{`⟨*start*⟩`}{`⟨*end*⟩`}` Deletes text between ⟨*start*⟩ and ⟨*end*⟩. This is equivalent to `\replacerange` with an empty replacement argument.

`\edf@findtextin` `{`⟨*haystack*⟩`}{`⟨*needle*⟩`}{`⟨*callback*⟩`}`

Searches text ⟨*haystack*⟩ for text ⟨*needle*⟩. Upon finding it, executes ⟨*callback*⟩ with two arguments, the text before ⟨*needle*⟩ and the text after. If the text is not found, an error is raised.

`\edf@findtext@in@content` `{`⟨*find*⟩`}{`⟨*callback*⟩`}`

Finds text in `\edf@content`.

`\edf@findtext@range` `{`⟨*haystack*⟩`}{`⟨*start*⟩`}{`⟨*end*⟩`}` `{`⟨*callback*⟩`}`

Searches ⟨*haystack*⟩ for the range of text between ⟨*start*⟩ and ⟨*end*⟩, and then runs ⟨*callback*⟩ with three arguments: the text before the range, the text between the range (excluding the start and end texts), and the text after the range.

`\edf@findtext@range@content` `{`⟨*start*⟩`}{`⟨*end*⟩`}{`⟨*callback*⟩`}`

Searches `\edf@content` for the given range.