Documentation guide

Introduction

The documentation for Sparse is written in plain text augmented with either reStructuredText (.rst) or MarkDown (.md) markup. These files can be organized hierarchically, allowing a good structuring of the documentation. Sparse uses Sphinx to format this documentation in several formats, like HTML or PDF.

All documentation related files are in the `Documentation/` directory. In this directory you can use `make html` or `make man` to generate the documentation in HTML or manpage format. The generated files can then be found in the `build/` sub-directory.

The root of the documentation is the file `index.rst` which mainly lists the names of the files with real documentation.

Minimal reST cheatsheet

Basic inline markup is:

- `*italic*` gives
- `**bold**` gives
- ``` ``mono`` ``` gives `mono`

Headings are created by underlining the title with a punctuation character; it can also be optionally overlined:

```
##############
Major heading
##############

Minor heading
-------------
```

Any punctuation character can be used and the levels are automatically determined from their nesting. However, the convention is to use:

- `#` with overline for parts
- `*` with overline for chapters
- `=` for sections
- `-` for subsections
- `^` for subsubsections

Lists can be created like this:

```
* this is a bulleted list
* with the second item
  on two lines
* nested lists are supported

        * subitem
        * another subitem

* and here is the fourth item

#. this is an auto-numbered list
#. with two items

1. this is an explicitly numbered list
2. with two items
```

Definition lists are created with a simple indentation, like:

```
term, concept, whatever
        Definition, must be indented and
        continue here.

        It can also have several paragraphs.
```

Literal blocks are introduced with `::`, either at the end of the preceding paragraph or on its own line, and indented text:

```
This is a paragraph introducing a literal block::

        This is the literal block.
        It can span several lines.

        It can also consist of several paragraphs.
```

Code examples with syntax highlighting use the *code-block* directive. For example:

```
.. code-block:: c

        int foo(int a)
        {
                return a + 1;
        }
```

will give:

```c
int foo(int a)
{
        return a + 1;
}
```

Autodoc

Sparse source files may contain documentation inside block-comments specifically formatted:

```
///
// Here is some doc
// and here is some more.
```

More precisely, a doc-block begins with a line containing only `///` and continues with lines beginning by `//` followed by either a space, a tab or nothing, the first space after `//` is ignored.

For functions, some additional syntax must be respected inside the block-comment:

```
///
// <mandatory short one-line description>
// <optional blank line>
// @<1st parameter's name>: <description>
// @<2nd parameter's name>: <long description
// <tab>which needs multiple lines>
// @return: <description> (absent for void functions)
// <optional blank line>
// <optional long multi-line description>
int somefunction(void *ptr, int count);
```

Inside the description fields, parameter's names can be referenced by using `@<parameter name>`. A function doc-block must directly precede the function it documents. This function can span multiple lines and can either be a function prototype (ending with `;`) or a function definition.

Some future versions will also allow to document structures, unions, enums, typedefs and variables.

This documentation can be extracted into a .rst document by using the *autodoc* directive:

```
.. c:autodoc:: file.c
```

For example, a doc-block like:

```
///
// increment a value
//
// @val: the value to increment
// @return: the incremented value
//
// This function is to be used to increment a
// value.
//
// It's strongly encouraged to use this
// function instead of open coding a simple
// ``++``.
int inc(int val)
```

will be displayed like this:

Intermediate Representation

To document the instructions used in the intermediate representation a new domain is defined: ‘ir’ with a directive:

```
.. op: <OP_NAME>
       <description of OP_NAME>
       ...
```

This is equivalent to using a definition list but with the name also placed in the index (with ‘IR instruction’ as descriptions).