Olena – Tutorial

LRDE

Copyright

Contents

Chapter 1

Welcome

Welcome to Milena's tutorial.

1.1 How to learn Milena

Milena is only a subpart of Olena but tends to be a large system too. Therefore it is not possible to present all the functionalities in a tutorial.

Milena targets several audiences: *end users*, *designers* and *providers*. *End users* want to apply and assemble algorithms to solve image processing, pattern recognition or computer vision problems, *designers* build new algorithms and *providers* are interested in developing their own data structures and extend an existing library.

Whatever the kind of user you are, the key to learning how to use Milena is to become familiar with its palette of objects and the way of combining them.

As an *end user*, you may start with this simple tutorial and the Quick tour . They describe and illustrate the key features of the library. *End users* getting familiar with Milena and *designers*, should take a look at the Quick Reference Guide . It is a more detailed explanations of the library's features.

end users and *designers* may be also interested by all the examples provided with the documentation and the tutorial. The source code is available in *milena/doc/examples* and is usually pointed out and commented by the documentation.

Taking a look at the test suite is also a good idea. The tests usually focus on a single functionality and handle several use cases which may overlap your needs. The test suite is located in *milena/tests* .

Still not enough information? More information about all the routines is available in the User HTML documentation . It mainly targets *designers* and *providers*. The latter may also be interested by the Developer HTML documentation (not available yet).

1.2 Obtaining the library

There are two ways of getting Milena on the web:

- Download a tarball/package from the website,
- Checkout the SVN repository.

Downloading a package or a tarball is the best choice for a new user. Except for nightly builds which are packages generated every night from the SVN repository, packages and tarballs contain only a released version of Milena. It guaranties a certain quality: no building issues, no bugs (ok, maybe some...), ...

This tutorial is based on the latest released version of Milena. Therefore, if you decide to use the SVN version, you may notice different behaviors or results compared to what it is described in this document.

Using the SVN version implies some drawbacks: the code might crash, not compile or produce incorrect results. Besides, The SVN version is always up to date and you may find new functionalities, bug fixes and new syntax improvements. This version targets users familiar with build systems and compilation issues. We strongly advise you to not use it for production use.

1.3 Downloading the library

1.3.1 Downloading from SVN

First, be sure that SVN is already installed on your system. Open a terminal and type:

```
$ svn --version --quiet
1.4.6
```

You should see your version of SVN installed. If you read 'Command not found' then you need to install SVN.

Usually, systems providing packages reference SVN's package as 'subversion'.

To install SVN on Debian or Ubuntu, run:

```
$ sudo apt-get install subversion
```

For other distributions, please refer to the user documentation of your system.

Once you have SVN installed, go to the directory where you would like to download Olena and create a new directory.

```
$ cd $HOME
$ mkdir olena
$ cd olena
```

Then 'checkout' (download) the repository with the following command.

```
$ svn co https://svn.lrde.epita.fr/svn/oln/trunk
```

Enter the 'trunk' directory.

```
$ cd trunk
```

You are now ready to configure the directory and install Milena as described in section 2. We invite you to take a look at the description of the directory structure (section 1.5). If you encounter any issues in the installation process or if you have any question, do not forget to join the mailing lists (section 1.4) and/or use the other documentations resources (section reftuto1documentation).

1.3.2 Downloading packaged releases

Milena's packages can be downloaded from:

http://www.lrde.epita.fr/Olena/Download

On this page you will find the latest and past releases. Currently, we provide only '.tar.gz' and 'tar.bz2' archives.

Once downloaded, you just need to uncompress the archive.

For the '.tar.gz' archive:

```
$ tar zxvf olena-1.0.tar.gz
```

For the '.tar.bz2' archive:

```
$ tar jxvf olena-1.0.tar.bz2
```

Then, enter the new created directory:

```
$ cd olena-1.0
```

You are now ready to configure the directory and install Milena as described in section 2. We invite you to take a look at the description of the directory structure (section 1.5). If you encounter any issues in the installation process or if you have any question, do not forget to join the mailing lists (section 1.4) and/or use the other documentations resources (section 1.6)).

1.4 Join the mailing lists

Regardless your use of Olena, we strongly advise you to join our mailing lists. This is the best way to keep up to date about new releases, bug notifications/-fixes and future updates. This is also a good opportunity to tell us what you would like to find in Milena and what could be improved.

Currently four mailing-lists are available:

Olena	Discussion about the project Olena
Olena-bugs	Bugs from Olena projects
Olena-core	Internal list for the Olena project
Olena-patches	patches for the Olena project

You can subscribe to these mailing lists at the following address:

`https://www.lrde.epita.fr/mailman/listinfo/`

Just click on the name of the mailing list you want to subscribe to and fill out the form.

1.5 Directory structure

Milena's directory is composed of several subdirectories. In order to help you finding what you need, you will find a description of all these subdirectories.

List of *milena*'s subdirectories:

- ***apps*** — A full example of a 3D mesh visualisation tool. It uses milena.
- ***doc*** — THE directory you must know. Contains all the documentation material.
- ***img*** — A set of common test images. They are used in the test suite. Feel free to use it in your programs.
- ***mesh*** — A set of 3D meshes. They can be used with the full example located in *milena/apps*.
- ***mln*** — The core of the library. Contains all the library headers.
- ***tests*** — The test suite. Is it subdivided in sub directories. The directory hierarchy respects *milena/mln*'s.
- ***tools*** — Small tools written with milena. They can be used as examples.

List of *mln*'s subdirectories:

- ***accu*** — Set of Accumulators.
- ***algebra*** — Algebraic structures like vectors or matrices.
- ***arith*** — Arithmetical operators.
- ***binarization*** — Routines to binarize an image.
- ***border*** — Image border related routines.
- ***canvas*** — Generic canvas. They define generic ways of browsing an image, compute data, ...
- ***convert*** — Automatic conversion mechanism.
- ***core*** — Core of the library. Here you can find the image types, the site set types and basic concepts.

- ***data*** — Routines that modify image data.
- ***debug*** — Debug related routines.
- ***display*** — Display images on the screen.
- ***draw*** — Draw geometric objects in an image.
- ***essential*** — Set of essential headers for 1,2,3-D manipulations.
- ***estim*** — Compute data on image values.
- ***extension*** — Image extension manipulation.
- ***fun*** — Set of functions applying on sites, values, ...
- ***geom*** — Functions related to image geometry.
- ***graph*** — Graph related routines.
- ***histo*** — Histogram related functions.
- ***io*** — I/O related routines.
- ***labeling*** — Labeling related routines.
- ***linear*** — Linear operators.
- ***literal*** — Generic image values such as zero, black, white ...
- ***logical*** — Logical operators.
- ***make*** — Small routines to construct images, windows, ...
- ***math*** — Mathematical functions.
- ***metal*** — Metallic macros/structures. Static library helping developing doing static tests.
- ***morpho*** — Mathematical morphology.
- ***norm*** — Norm computation.
- ***opt*** — Optional routines. Routines which may work on a specific image type only.
- ***pw*** — Point-wise image related routines.
- ***registration*** – Registration related routine.
- ***set*** — Set related routines.
- ***subsampling*** — Sub-sampling related algorithms.
- ***tag*** — Tag traits.

- ***test*** — Definition of predicates.
- ***topo*** — Complex related structures.
- ***trace*** — Debug trace mechanism.
- ***trait*** — Internal traits mechanism.
- ***transform*** — Algorithms based on the data::transform.
- ***util*** — Various utilitarian classes.
- ***value*** — Set of value types which can be used in an image.
- ***win*** — Set of various window kinds.

The source code and the material of the documentation is available in *milena/doc*. List of *doc*'s subdirectories:

- ***examples*** — All the source code of the documentation examples.
- ***benchmark*** — Some benchmarks.
- ***tools*** — Small tools used for generating documentation / building examples.
- ***tutorial*** — Tutorial sources.
- ***white_paper*** — White paper sources.
- ***ref_guide*** — Reference guide sources.
- ***figures*** — Reference figures for documentation generation.
- ***outputs*** — Reference outputs for documentation examples.

1.6 Documentation

This tutorial is not the only documentation of Milena. Other documents are available:

- ***White paper*** — A small document of few pages presenting the key features of the library. It intents to give a big picture of the library.
- ***Quick tour*** — It aims at giving an overview of Milena's possibilities. It does not only give the concepts but illustrate them with small sample codes.
- ***Quick reference guide*** — Presents in details all the main functionalities of Milena. Hints and full examples are also provided. The sample codes are commented and each concept in the library is detailed. This is the reference document for any *end user* and *algorithm designer*.

- ***HTML user doc*** — The full documentation of the library. The full API is described in details. Each part of the library is classified by categories and the source code is directly accessible from the documentation. This is the reference document for any *algorithm designer* and/or *provider of data structures*.

- ***Header files*** — Every object or algorithm is declared in a '.hh' file. The documentation is provided as comments in these file.

1.7 Community and Support

Even though Milena is currently developed by the LRDE in EPITA, we are open for new contributors.

- If you are a user, please send us feedback about the library. Did you find what you wanted? Do you miss something?

- Please report bugs and defects in the API. Mailing lists are the best way for reporting that (section 1.4).

- Developers, if you write cool open source programs or algorithms with Milena, send them to us. We may ship your code with Olena and/or add it to our download page.

- Educators, if you use Olena for your courses and you are ready to share your materials, you can send it to us through our mailing-lists.

- We are also interested in partnership or commercial use of Milena. If you are interested, contact us directly (1.10).

1.8 Project status

If you want to stay tuned to Milena's development, the best way is probably the mailing-lists (section 1.4).

There are other ways to get to know what is the status of the project.

- Olena's trac
  https://trac.lrde.org/olena
  Here is the road-map, the current open tickets/bugs/improvements which are taken in consideration. A source browser is also available.

- Olena's Buildfarm
  https://buildfarm.lrde.org/buildfarm/oln/
  The official build-farm. Every night and after each commit, tests are compiled and run. The build-farm can show you whether it is safe to update your svn copy of Milena or not...

- Test failures
  http://www.lrde.epita.fr/dload/olena/test-failures-daily.html
  Through this page, you can see exactly which tests do not compile or pass. This page is updated every night.

1.9 A brief history of Milena

The Olena project aims at building a scientific computation platform oriented towards image processing, image recognition, and artificial vision. This environment is composed of a high performance generic library (Milena), a set of tools for shell scripts, together with, in the more distant future, an interpreter (a la Octave, MatLab etc.) and a visual programming environment.

The Olena project started in 2000 from a small prototype on 2-D images. From November 2001 to April 2004, this prototype evolved from version 0.1 to 0.10. More image types were supported and the level of genericity expected from the library was partially obtained. During these three years, the prototype was used to experiment with genericity and to try to meet our objectives. In February 2007, Olena 0.11 was released to conform modern C++ compilers. At that time, the code was not enough readable though and the compilation time was too long.

Since June 2007 up to now, The library of the Olena platform is called Milena and the library has been rewritten. The programming paradigm has been simplified: the code is more readable and the compilation time is acceptable. The level of genericity still meets our objectives though.

Milena is now getting ready for being considered as stable and distributable. The core of the library is getting frozen and we aim at enriching the library, its documentation and the related tools.

1.10 Contacts

If you want to reach us directly, you can contact one of the following people:

- Thierry Geraud - Project Manager - thierry.geraud@lrde.epita.fr

Chapter 2

Installation

This section describes the installation process of Milena. Do not forget that Milena is a library, not a program. Therefore, no program will be installed.

Milena's examples and tests are compiled on the following platforms:

- Apple Tiger Darwin 8, PowerPC, GCC 4.0.1
- Apple Leopard Darwin 10.5, X86-64, GCC 4.0.1, 4.2
- Linux, i486, GCC 3.3, 4.1, 4.2, 4.3
- Linux, x86-64, GCC 4.1

We guaranty that Milena compiles on these platforms, e.g. Linux and Unix platforms. It may compiles on other platforms though but we have not tested. If you did, and you succeeded, please let us know in order to update this section.

Milena is known NOT to work with GCC-2.95.

Milena is actively developed under Unix systems. As a result, the build system is based on the Autotools. Autotools make sure that every dependencies are resolved before compiling or installing a program.

Milena is different from usual libraries in a way that nothing needs to be compiled to use it. The library itself is composed of headers which must be included when you need them. Then, your application will be compiled with the parts of the library used in that program. That's all.

So, why do we have a build system? It is useful for installing the library on your system, generating the doc and compiling the test suite and the examples.

2.1 Bootstrap (SVN Sources)

If you got the sources from a package/tarball, you can skip this section. Go to section 2.2.

If you downloaded the sources from the SVN repository, you must launch a script before configuring the build directory.

Run the following:

```
$ cd /my/path/to/olena-1.0
$ ./bootstrap
```

Running 'bootstrap' can take a while. Some files are generated during this process. When it's done, you are ready to configure the build directory.

2.2 Configure

First, make sure you are at the root directory of the milena source:

```
$ cd /my/path/to/olena-1.0
```

First, create and enter a build directory:

```
$ mkdir build
$ cd build
```

We are now about to configure the build directory. This process will create the necessary files to compile documentation, examples and tools and prepare the installation.

Important Note: the installation path prefix must be chosen at this step. By default, Milena will be installed in /usr/local but you may like to install it elsewhere. To do so, pass the option *--prefix=/installation/path/prefix* to the configure script (see below). Replace '/installation/path/prefix' with the wanted installation path prefix.

now, you can run:

```
$ ../configure
```

OR

```
$ ../configure --prefix=/installation/path/prefix
```

The configure script will perform various tests. If there is no dependency issues, the last lines shown before the prompt are:

```
config.status: creating config.h
config.status: executing depfiles commands
$
```

And if you type the following command, a '0' is printed out.

```
$ echo $?
0
$
```

The build directory is now configured, the library can be installed.

2.3 Install

First, be sure to be in the build directory. If you followed the previous steps, the build directory should be in the Milena sources root directory.

```
$ cd /my/path/to/olena-1.0/build
```

If you did not change the default install path prefix, set to */usr/local*, you will need to have administrator privileges to perform the installation. Then, you may type:

```
$ sudo make install
```

You will be prompted for the administrator password.

Otherwise, if you set the install path prefix to a directory own by your user, simply type:

```
$ make install
```

When the installation is finished, you are done. Milena is installed on your system. But maybe you would like to build the examples? This is described in section 2.4.

A description of the installation content is also available in section 2.5.

2.4 Optional compilation

The library itself does not need to be compiled, therefore installing Milena does not require compilation.

Though, some examples and tools are provided with the library and must be compiled if you want to use them.

2.4.1 Examples

Examples are part of the documentation. The sources are located in *milena/doc/examples*.

To compile the examples simply run:

```
$ cd /my/path/to/olena-1.0/build/milena/doc/examples
$ make
```

These examples can produce outputs and images. May be you would like to run all the examples and take a look at the outputs? To do so, run:

```
$ cd /my/path/to/olena-1.0/build/milena/doc/examples
$ make examples
```

Text and image outputs will be respectively stored in *build/milena/doc/outputs* and *build/milena/doc/figures*.

2.4.2 Tools

Few tools are provided with Milena. They can be considered as full program examples.

Currently two tools are available:

area_flooding.cc
seed2tiling.cc

To build these tools, run:

```
$ cd /my/path/to/olena-1.0/build/milena/tools
$ make
```

2.4.3 Tests

The test suite used for Milena's development is shipped with the library. It is usually useless for simple users and tends to be used by developers extending the library.

In order to build and run it, just do the following:

```
$ cd /my/path/to/olena-1.0/build/milena/tests
$ make check
```

Running the test suite is memory and CPU consuming and will take a while.

2.5 Installation content

Once installed, Milena's files are located in the installed path prefix you passed to the configure script or in the default path /usr/local.

In the installation path prefix, Milena's files are located in:

- include/mln/ — The library. All the headers are located here.
- share/olena/images — Mesh sample files which may be used with example programs.

Chapter 3

Getting started with Milena

3.1 Getting familiar with genericity

One of Milena's main features is its genericity. In order to understand how to take benefit of it, let's see what genericity really means for us and how it is illustrated in the library.

A **Generic algorithm** is written once, without duplicates, and works on different kinds of input.

Let's have a look to a small example. In any image processing library, we may be interested in a small routine to fill an image with data. A common implementation would look like this one:

```
// Java or C-like code

void fill(image *ima, unsigned char v)
{
   for (int i = 0; i < ima->nrows; ++i)
     for (int j = 0; j < ima->ncols; ++j)
        ima->data[i][j] = v;
}
```

In this example, there are a lot of **implicit** assumptions about the input:

- The input image has to be 2D;
- Its definition domain has to be a rectangle starting at (0,0);
- Data cannot be of a different type than *unsigned char*;
- Image data need to be stored as a 2D array in RAM.

So, what would happen if we would like to use it for 3D images, use rgb8 as value or even work on a region of interest?

This implementation would require to be re-implemented and the user would have to deal with the various versions of the fill routine. For the developer, it

is error prone, redundant and hard to maintain. For the user, it is confusing and forces to always think about what he is manipulating. According to our definition, this algorithm is clearly **not** generic.

This is not acceptable and that's why Milena is developed considering genericity and user/developer friendliness.

With Milena, the previous example would be written as follow:

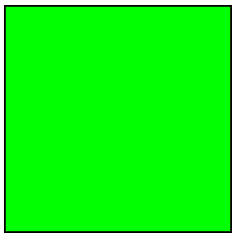
```
template <typename I>
void fill(I& ima, mln_value(I) v)
{
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

In this version, the routine can take any kind of image types as arguments. So it is for the values: the expected type depends on the value used in the given image. The *for_all* loop is also significantly generic to support any kind of images since the iterator guarantees it will pass through every sites.

This code is more generic and remains really close to the common description of the generic algorithm.

As a result, with this algorithm we can fill an image,...

```
  fill(ima, literal::green);
```



... Or fill only a region of interest (a set of points).

```
  box2d b(20,20);
  fill((ima | b).rw(), literal::green);
```

3.2 First generic algorithm

In this section, we will introduce several routines/tools which are useful while writing generic algorithms. It is more important to focus on these routines/tools than what this program actually does.

Here is the full example:

```
namespace mln
{

  template <typename I, typename N>
  mln_concrete(I)
  my_algorithm(const Image<I>& ima_,
               const Neighborhood<N>& nbh_)
  {
    trace::entering("my_algorithm");

    const I& ima = exact(ima_);
    const N& nbh = exact(nbh_);
    mln_precondition(ima.is_valid());
    mln_precondition(nbh.is_valid());

    typedef value::label_8 V;
    V nlabels;
    mln_ch_value(I,V) lbl = labeling::blobs(ima, nbh, nlabels);
    util::array<unsigned>
      count = labeling::compute(accu::meta::math::count(), lbl, nlabels);

    mln_concrete(I) output;
    initialize(output, ima);
    data::fill(output, literal::one);

    for (unsigned i = 1; i <= nlabels; ++i)
      if (count[i] < 10u)
        data::fill((output | (pw::value(lbl) == pw::cst(i))).rw(), literal::zero);

    trace::exiting("my_algorithm");
    return output;
  }

} // end of namespace mln
```

Let's see the different parts of the algorithm.

```
  template <typename I, typename N>
  mln_concrete(I)
  my_algorithm(const Image<I>& ima_,
               const Neighborhood<N>& nbh_)
```

The prototype is restrictive enough, readable and still generic. We use concepts to statically check that the generic type passed as parameter is what the routine

expects. The "exact" image type is *I*. For instance an image of type *image2d* inherits from *Image<image2d>*. So an *image2d* is an *Image<I>*. Note that the return type of this function is defined by a macro. *mln_concrete* is a macro hiding tricky mechanisms (traits) used in Milena. The important point to remember is that a generic function should not return *I* directly but *mln_concrete(I)* instead.

```
  trace::entering("my_algorithm");
```

Like any Milena's routine, note that we use *trace*. This debugging tool will be detailed in section 3.4.

```
  const I& ima = exact(ima_);
  const N& nbh = exact(nbh_);
  mln_precondition(ima.is_valid());
  mln_precondition(nbh.is_valid());
```

Since the function take some arguments as concept objects, these object cannot be used as such. Indeed, concepts are empty shells only used for dispatching and concept checking, that's the reason why they are parameterized with their exact type. The exact type let us know what is the real type of the object. To get an object with the exact type, simply call *exact()*. Of course, it is always a good idea to add few preconditions to help during debug.

```
  typedef value::label_8 V;
  V nlabels;
  mln_ch_value(I,V) lbl = labeling::blobs(ima, nbh, nlabels);
  util::array<unsigned>
    count = labeling::compute(accu::meta::math::count(), lbl, nlabels);
```

In this portion of code, the image is labeled and the number of sites per label is computed. This code does not depend on the image type at all. Again, a macro *mln_ch_value* ("mln change value") helps us. *labeling::blobs* is a routine returning an image of the same kind as the input image but with a different value. *mln_ch_value* enables the possibility of doing that, whatever the image type *I* and whatever its value type, it returns the same image type with a different value type.

```
  mln_concrete(I) output;
  initialize(output, ima);
  data::fill(output, literal::one);
```

The output image is declared here. Like any variable, it must be initialized at some point. To do so, *initialize()* is provided. It is a generic routine which can initialize the geometry of any image kind with another image of the same kind. After this call, *output* has a valid domain and is valid. It can be used in an algorithm, here *data::fill*, to have its values modified. Note that the value passed to *data::fill* is also generic. The library includes few generic common values from which any value type can convert to. *literal::one* is one of them. It is a generic one value which can convert to every value type in the library.

```
  for (unsigned i = 1; i <= nlabels; ++i)
    if (count[i] < 10u)
      data::fill((output | (pw::value(lbl) == pw::cst(i))).rw(), literal::zero);
```

In this part, every region from the labeled image, of which cardinality is lower than 10 sites, is set to *literal::zero* in *output*. Once again, a generic value is used in order to avoid constraints on the image value type.

```
  trace::exiting("my_algorithm");
  return output;
```

Don't forget to close the trace before exiting the function. Then return the result.

3.3 Compilation

3.3.1 Include path

If Milena has been installed in a custom directory, e.g. not /usr/include or /usr/local/include, the path to the library headers must be passed to the compiler.

With g++ and MinGW, the option is **-I<path>** .

```
$ g++ -Ipath/to/mln my_program.cc
```

For other compilers, please look at the documentation and search for "include path".

3.3.2 Library linking

As it is usually expected when using a library, no library linking is needed for the library itself. Milena is a "header only" library and is compiled "on demand" with your program.

If you use specific input/output you may need to link your program with the right graphic library. For more information, please refer to section **??** in the Quick Reference Guide.

3.3.3 Disable Debug

By default, Olena enables a lot of internal pre and post conditions. Usually, this is a useful feature and it should be enabled. It can heavily slow down a program though and these tests can be disabled by compiling using -DNDEBUG:

```
$ g++ -DNDEBUG -Ipath/to/mln my_program.cc
```

3.3.4 Compiler optimization flags

In this section you will find remarks about the compiler optimization flags and their impact on the compilation and execution time.

GCC

- **-O0** , combined with -DNDEBUG, it leads to the fastest compilation time. The execution is somewhat slow though since dispatch functions and one line members are not inlined by the compiler.
- **-O1** , best compromise between compilation time and execution time.
- **-O2** , **-O3** , combined with -DNDEBUG, it leads to the best execution time. However these optimizations dramatically slow down the compilation and requires much more memory at compile time.

Other compilers

Currently, we have not tested different optimization flags with other compilers. If you did, please report us your results.

3.4 Debug hints

3.4.1 Using assertions and GDB

As said above, Milena already includes a lot of post and pre conditions. Thus, if you made a mistake in your code there is a high probability that it will be detected at run time. If an assertion fails, we advice you to compile with the following options:

```
$ g++ -ggdb -Ipath/to/mln my_program.cc
```

Note that you **MUST NOT** compile with $-DNDEBUG$ since assertions will be disabled. Once compiled, restart the program with GDB.

```
$ gdb ./my_program
```

In the GDB console, run it again.

```
(gdb) run <any parameter you may want to pass to the program>
```

When an assertion fails, in the GDB console simply type:

```
(gdb) bt
```

The full backtrace will be printed out and you will be able to find from where the error come from. The filenames, the line numbers and the parameters values are printed out in the backtrace as you can see in the following example:

```
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7d00640 in raise () from /lib/i686/cmov/libc.so.6
#2  0xb7d02018 in abort () from /lib/i686/cmov/libc.so.6
#3  0xb7cf95be in __assert_fail () from /lib/i686/cmov/libc.so.6
```

```
#4  0x0804e094 in mln::image2d<bool>::has (this=0xbff32f34, p=@0xbff32f3c)
    at /lrde/stockholm/lazzara/svn/olena/git/oln/milena/mln/core/image/image2d.hh:442
#5  0x0804e6d7 in mln::image2d<bool>::operator() (this=0xbff32f34, p=@0xbff32f3c)
    at /lrde/stockholm/lazzara/svn/olena/git/oln/milena/mln/core/image/image2d.hh:460
#6  0x080490b0 in main () at test.cc:18
```

3.4.2 Traces

Sometimes, compiling for GDB without optimization flags and with debug assertions enabled could lead to execution time dramatically high. If the function parameter values are not necessary for debugging, a good alternative is the trace system provided in Milena. Each time a routine is called, a trace log is written.

This trace allows to follow the stack trace at runtime. It also provides the time passed in each function call if the call last at least 10ms.

In order to enable traces in a program, set the related global variable to true:

```
...
trace::quiet = true;
...
```

Since it's a global variable, at anytime in the source code, the trace can be enabled/disabled.

Traces are enabled:

```
// ...
trace::quiet = false;
```

labeleling::blobs is run and the debug is then disabled.

```
labeling::blobs(ima, c4(), nlabels);

trace::quiet = true;

geom::bbox(ima);
// ...
```

The previous code will produce the following trace:

```
labeling::blobs {
  core::initialize {}
  data::fill {
    data::fill_with_value {
      data::impl::fill_with_value_one_block {
        data::memset_ {
          data::impl::memset_ {}
        } data::memset_
      } data::impl::fill_with_value_one_block
    } data::fill_with_value
  } data::fill
} labeling::blobs - 0.08s
```

As you can see, *labeling::blobs* is located just after having set *trace* :: *quiet* to *true* so its trace is part of the output. *geom::bbox*'s trace is not part of the output though since traces have been disabled just before it is called.

3.4.3 Debug routines

Milena also provides a lot of debug tools. Here is a small list of the tools:

- mln::debug::println, print an image in the console.

```
image2d<int_u8> ima(5,5);
data::fill(ima, 2);
debug::println(ima);
```

```
2 2 2 2 2
2 2 2 2 2
2 2 2 2 2
2 2 2 2 2
2 2 2 2 2
```

- mln::debug::println_with_border, print an image in the console withs its border.

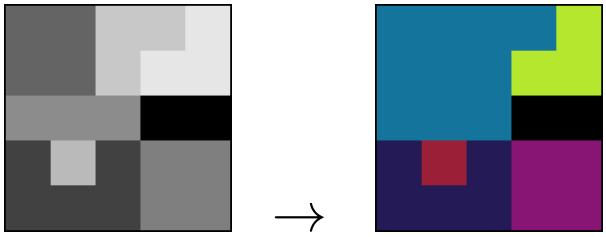
```
image2d<int_u8> ima(5,5);
data::fill(ima, 2);
border::fill(ima, 7);
debug::println_with_border(ima);
```

```
7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7
7 7 7 2 2 2 2 2 7 7 7
7 7 7 2 2 2 2 2 7 7 7
7 7 7 2 2 2 2 2 7 7 7
7 7 7 2 2 2 2 2 7 7 7
7 7 7 2 2 2 2 2 7 7 7
7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7
```

- mln::labeling::colorize, colorize a label image with random colors.

```
int_u8 vals[25] = { 100, 100, 200, 200, 230,
                    100, 100, 200, 230, 230,
                    140, 140, 140,   0,   0,
                     65, 186, 65, 127, 127,
                     65,  65, 65, 127, 127 };
```

```
image2d<int_u8> ima = make::image2d(vals);
image2d<rgb8> ima_color = labeling::colorize(rgb8(), ima, 230);
```



- mln::labeling::superpose, Superpose two images.
- mln::labeling::filename, easily format debug file names.

Chapter 4

Data representation

This chapter aims at explaining how images are stored and which objects compose an image in Milena. We will start to talk about the localization of a pixel and then the image itself which stores the values.

4.1 Sites

A pixel is an element having both information, localization and value. In Milena, we make a difference between a pixel, a pixel value and a pixel location. Thus, in order to refer to a pixel location, we have the site concept. A site can be any kind of localization element. For instance, in an image defined on a 2D regular grid, it is a 2D point with *row* and *col* coordinates.

```
point2d p(3,3);
std::cout << p << std::endl;
```

The image site type is defined by its underlying site set.

4.2 Site sets

Site sets are mainly used to define image domains. They hold all the available sites in an image, consequently they do not store any values.

Site sets can used as standalone containers.

A list of available site sets is available in section **??**.

4.2.1 Creating a site set

In this section, we will detail how to create common site sets.

This is actually simple.

4.2.2 Getting access to sites

4.3 Images

In milena, an image is seen as a composition of both a site set and a function mapping a site to a value.

4.3.1 Creating an image

In this section, we will detail how to create common images.

4.3.2 Reading an image from a file

4.3.3 Accessing data

Chapter 5

Load and save images

After this step you should know how to:

- load an image,
- save an image.

Currently, Olena supports the following input image formats:

- PBM
- PFM
- PGM
- PNM
- PPM

This support is provided through two headers for each type, *save.hh* and *load.hh*. They are located in *mln/io/<image-format>/*.

Once the right header is included, the image can be loaded:

```
image2d<bool> ima;
io::pbm::load(ima, "my_image.pbm");
```

If you wan to save an image, simply call the save routine in the proper namespace:

```
io::pbm::save(ima, "../figures/ima_save.pbm");
```

According to the image value type, the proper file format must be chosen. The supported file formats and their associated image value types are listed in section **??**.

Chapter 6

Create your first image

After this step you should know how to:

- create an image,
- display an image in console mode.

??

First, declare an array of bool which will represent the image grid. Each each cell in this grid is a site and each cell contains a value, *true* or *false*.

```
bool vals[13][21] = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
    {0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
    {0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
    {0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
    {0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0},
    {0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0},
    {0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
    {0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0},
    {0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

From that grid, simply call make::image to get an image initialized with that data.

```
image2d<bool> ima = make::image(vals);
```

This way of initializing an image is the most common one. However, there are several other ways described in section **??**.

To be sure that the data is correctly initialized, it is possible to display the image in the standard output using debug::println.

```
debug::println(ima);
```

Output:

```
- - - - - - - - - - - - - - - - - - - - - - - -
- | - | - | | | - | - - - | - - - - | - - -
- | - | - | - - - | - - - | - - - | - | -
- | | | - | | | - | - - - | - - - | - | -
- | - | - | - - - | - - - | - - - | - | -
- | - | - | | | - | | | - | | | - - | - -
- - - - - - - - - - - - - - - - - - - - - - - -
- | - | - - | - - | | - - | - - - | | - -
- | - | - | - | - | - | - | - | - - - | - | -
- | - | - | - | - | - | | - - | - - - | - | -
- | | | - | - | - | - | - | - - - | - | -
- | - | - - | - - | - | - | | | - | | | - -
- - - - - - - - - - - - - - - - - - - - - - - -
```

Finally, you may want to save the image. Since we use bool as image value, the PBM format is the best choice. Therefore, we use io::pbm::save.

```
doc::pbmsave(ima, "tuto2_first_image");
```

The output image looks like the following:

In this first step we used a boolean image. Many other value types are available though. A more detailed description can be found in section **??**.

Chapter 7

Read and write images

After this step you should know how to:

- modify/initialize image values,
- copy and paste data to an image.

??

First create an empty color image with a *box2d* of 40x40 as domain.

```
image2d<value::rgb8> ima(40, 40);
```

If you want to initialize the image with the color red, simply call data::fill as follows:

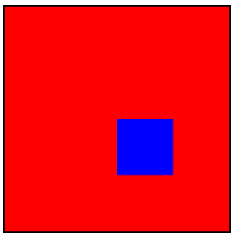
```
data::fill(ima, literal::red);
```

Updating a site value is also possible using *operator()* or the *opt::at()* routine. Here we create a blue square of 10x10 pixels from site (20, 20) to (30, 30).

```
for (def::coord row = 20; row < 30; ++row)
  for (def::coord col = 20; col < 30; ++col)
    ima(point2d(row, col)) = literal::blue;
```

```
for (def::coord row = 20; row < 30; ++row)
  for (def::coord col = 20; col < 30; ++col)
    opt::at(ima, row, col) = literal::blue;
```

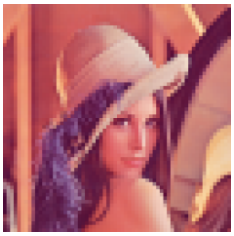
The corresponding image:



An image can also be initialized/modified thanks to another image. Let's load a new image.

```
image2d<value::rgb8> lena;
io::ppm::load(lena, MLN_IMG_DIR "/small.ppm");
```

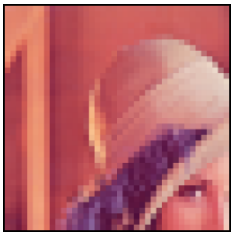
lena looks like:



If we want to initialize *ima* with *lena*, we can use *data::fill*:

```
data::fill(ima, lena);
```

Output:

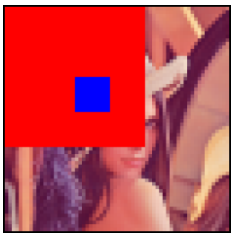


Note that to fill an image with some data, the image domain **must** be smaller or equal to the data.

Likewise, it is possible to paste data from an image to another:

```
data::paste(ima, lena);
```

Output:



More details can be found in sections **??**, **??** and **??** in the reference guide.

Chapter 8

Regions of interest

After this step you should know how to:

- take benefit of Olena's genericity,
- work only on a region of interest in an image.

??

In the previous step, we used the routine *data::fill* in order to change the values of an image. It was convenient since we did not need to write any loop by hand. The problem was that we could not specify which region to fill with data. This point leads us to talk about the genericity in Olena. All along this example we will use the routine *data::fill* to illustrate the possibilities in Olena but note that every image types passed to the routine in this example could be passed to any algorithm in the library expecting an image.

One main feature of Olena is to be able to easily work on regions of interest in images. According to the way a region of interest is defined, a specific image type is associated. Therefore, each algorithm knows exactly what it is working on and can behave differently in order to be the most efficient as possible.

All along this step, we will use the following image *lena* declared as follow:

```
image2d<value::rgb8> lena;
io::ppm::load(lena, MLN_IMG_DIR "/small.ppm");
```

data::fill has the following prototype:

```
namespace data
{

  template <typename I, typename D>
  void fill(Image<I>& ima, const D& data);

}
```

So keep in mind that the first argument we will try to construct in each example is an image. Note that this image **must** be writable, e.g. non-const.

8.1 Image domain restricted by a site set

Here, we would like to fill a small square with green in *lena*. We want this square to be of size 20x20 and to be located at (20,20). First, we just need to declare this square which is actually a site set, a *box2d*.

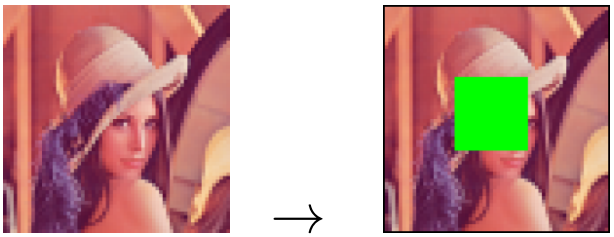
```
box2d roi = make::box2d(20, 20, 40, 40);
```

Then, we just need to tell *data::fill* that we would like to fill the image *lena* but only in this restricted part of the image domain.

```
data::fill((lena | roi).rw(), literal::green);
```

Operator '|' can be read 'restricted to'. So below, we wrote 'image *lena* restricted to the region of interest *roi*'. Actually this is not directly *lena* which is restricted but its domain.

Note the use of *rw()* which is mandatory due to C++ limitations. In C++, the image created by *lena* | *roi* is *const*, e.g. read-only, though *data::fill* expect a *non-const* image, e.g. read-write. *rw()* is a workaround to make it read-write.



Fill with blue a region of interest defined by a *box2d*.

8.2 Image domain restricted by a function

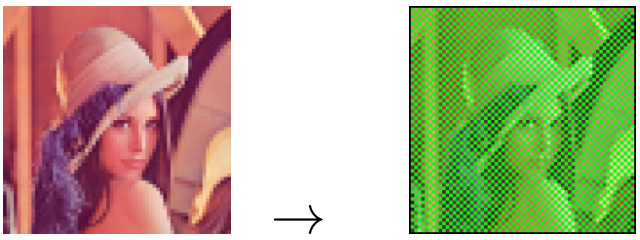
Sometimes it may not be easy to construct a site set to restrict an image. For instance, if we would like to fill with green one point out of two in the whole image, we **do not want** to write anyloop or construct any site set by hand:

```
p_array<point2d> arr;
for (def::coord row = geom::min_row(lena); row < geom::max_row(lena); ++row)
  for (def::coord col = geom::min_row(lena); col < geom::max_col(lena); ++col)
    if (((row + col) % 2) == 0)
      arr.append(point2d(row, col));
```

```
for (def::coord row = geom::min_row(lena); row < geom::max_row(lena); ++row)
  for (def::coord col = geom::min_row(lena); col < geom::max_col(lena); ++col)
    if (((row + col) % 2) == 0)
      opt::at(lena, row, col) = literal::green;
```

A shorter way to get exactly the same result, is to define that behavior by a function. In Milena, a function *fun::p2b::chess* is available and does exactly what we want. Like if it was a site set, simply restrict the image with the function.

```
data::fill((lena | fun::p2b::chess()).rw(), literal::green);
```



Fill with green a region of interest defined by a *Function*.

Note that the functions provided by default in Olena are actually functors. Thus, they must be constructed like any object which why it is written *lena | fun::p2v::chess()* and not *lena | fun::p2v::chess*.

8.3 Image domain restricted by a mask

Sometimes instead of having a site site or a function defining the regions of interest we want to work on, we may have a binary image, e.g. a mask. When a site has its value set to true, it means it will be considered as part of the masked image domain. Otherwise, it will not.

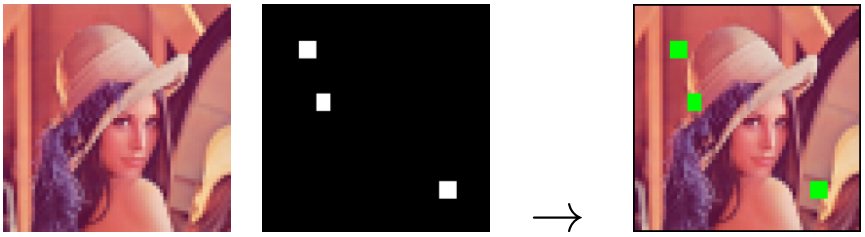
We construct a mask, *mask*. It is initialized with the same geometry properties as *lena* (domain, extension...).

```
image2d<bool> mask;
initialize(mask, lena);
data::fill(mask, false);
data::fill((mask | make::box2d(10, 10, 14, 14)).rw(), true);
data::fill((mask | make::box2d(25, 15, 29, 18)).rw(), true);
data::fill((mask | make::box2d(50, 50, 54, 54)).rw(), true);
```

Then, we cannot restrict directly *lena* with *mask*. These two images have the same domain, so *lena* | *mask.domain()* would not do anything. *mask* is a classical image, there is not specific type for mask images. Therefore, we need to express that we want that binary image to be considered as a mask.

```
data::fill((lena | pw::value(mask)).rw(), literal::green);
```

pw::value(mask) makes explicit the fact that *mask* is actually a mask. It means, that for each site of *mask*, if its value is set to *true*, then the value associated to this site in *lena* must be set to green. In this example, we use two images for two different use case: *lena* store the result and the modifications make by the algorithm and *mask* allows the algorithm to know whether it must treat a site or not.



Fill with green a region of interest defined by a mask image.

8.4 Image domain restricted by a predicate

Restricting by a predicate is exactly like restricting with a function. We want to talk about that separately in order to present the small routines available. They enable the user to write quick and efficient predicate/function.

The two routines are :

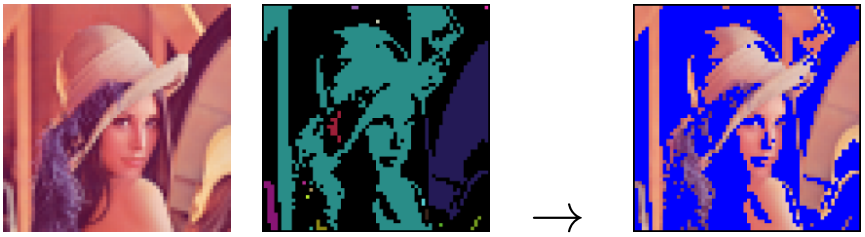
- pw::value(Image), as seen in a previous section, it is a way to express 'for each site value in Image'.
- pw::cst(Value), it is a way to specify a value to which a site value can be compared.

Let's see a common use case. First, we binarize lena according to specific criterions, only site values with specific colors are set to true in *lena_bw*. Others are set to false. This image will be used in order to label the components. Let's consider a labeled image *label*. Each component of *lena* is labeled with a unique

index. Now, we consider that that our region of interest is a component with id 16. Then we want to express 'for each site *fill* its value in *lena* if its value in *label* is equal to 16'.

```
image2d<bool> lena_bw = binarization::binarization(lena, keep_specific_colors());
value::label_8 nlabels;
image2d<value::label_8> label = labeling::blobs(lena_bw, c8(), nlabels);
```

```
data::fill((lena | (pw::value(label) == pw::cst(0u))).rw(), literal::blue);
```



Fill with green a region of interest defined by its label.