Milena: A Tutorial—Part 1

Milena Team

EPITA Research and Development Laboratory (LRDE)

November 2007

Outline

Outline

Outline

Outline

Outline

What is OLENA?

OLENA is the name for

- the project of building some modern image processing tools
- the platform, including
  - a library
  - command line executables
  - some documentation
  - etc.

MILENA

MILENA is the C++ image processing[a] library of OLENA.

[a]In the following, IP is "Image Processing" for short.

Yet Another Image Processing Library (YAIPL) ?

Yes!

- Many libraries exist that can fulfill one's needs.
- If you're happy with your favorite tool, we cannot force you to change for MILENA...
- Though, you might have a look at MILENA and be seduced!

No!

- MILENA is rather different than available libraries.
- A lot of convenient data structures that really help you in developing IP solutions.

A Short History of the OLENA Project

- 2000: Start of the project.
- From Nov. 2001 to April 2004: Evolution from version 0.1 to 0.10. The level of genericity we expected from the lib was partially obtained...
- February 2007: Update to conform modern C++ compilers = version 0.11.

During those 3 years we developed a prototype to experiment with genericity and to try to meet our objectives.

- From June 2007 up to now: Re-writing of the library with a programming paradigm that rocks.

Illustration of the Evolution

Algorithm:

$\forall p \in \mathcal{D}(f), \quad f(p) = h(f(p))$

In 2007:

```
template <typename I, typename H>
void transform_inplace(Image<I>& f_,
                       const Function_v2v<H>& h_)
{
  I& f = exact(f);
  const H& h = exact(h_);
  mln_piter(I) p(f.domain());
  for_all(p)
    f(p) = h(f(p));
}
```

The same code in 2000:

```
template< typename H,
          template< class U > class get_A = get_value,
          typename P = Pred_true >
struct transform_inplace
{
  template< typename I > static
  void on( I& f,
           P pred = P() )
  {
    H h;
    get_A< I::value_type > access;
    I::iterator_type iter( f );
    for ( iter.first(); ! iter.isDone(); iter.next() )
      if ( pred( access( iter() ) ) )
        access( iter() ) = h( access( iter() ) );
  }
};
```

Outline

What's In a Library

- algorithms:
  procedures dedicated to image processing and pattern recognition
- data types for pixel values:
  e.g., gray level types, color types
- data structures:
  image types or point set types for instance
- auxiliary tools...

MILENA as a Feature List

- Generic...
- Efficient so that one can process large images.
- Almost as easy to use as a C or Java library.
- Many tools to help writing readable algorithms in a concise way.

Genericity

MILENA is generic

- Put shortly it works on various types of images.
- Algorithms are highly reusable.

Efficiency

MILENA is efficient

- Written in C++ without the cost of function calls.
- Specialized algorithms are provided.

Easy to Use

MILENA is easy to use

- Just slightly more difficult to use than a library in C or Java.
- The user mainly write routine calls.

Many Tools

MILENA provides many tools

- A maximal amount of work is saved for the user.
- Claim: you do not think that IP people are ready to add tools to a lib.

Outline

What Is Needed

- A C++ compiler (`g++-4` is great and fast).
- A browser (e.g., `Firefox`)
- A pdf reader (e.g., `kpdf`)
- Either `unzip` or (`gzip` and `tar`)
- A directory to uncompress the MILENA archive.

Installation

1. Get a snapshot of MILENA from the web

   `http://olena.lrde.epita.fr/`

2. Uncompress the archive.
3. Have a look.

For instance:

```
tegucigalpa% cd
tegucigalpa% mkdir milena
tegucigalpa% cd milena
tegucigalpa% mv /tmp/milena-1.0-alpha.tar.gz .
tegucigalpa% tar zxvf *
tegucigalpa% ls doc
tegucigalpa% ls mln
```

The Main Directories

`doc`	some documentation materials
`img`	few tiny images to play with
`demo`	several examples of what can be done with MILENA
`mln`	the library

MILENA Brief Overview of the Library Contents

In `mln`:

`accu`	accumulator objects	`arith`	arithmetical operators
`border`	routines about virtual border	`canvas`	canvases
`convert`	conversion routines	`core`	the library core
`debug`	debugging tools	`display`	display tools
`draw`	drawing routines	`estim`	estimation operators
`fun`	functions	`geom`	geometrical routines
`histo`	histogram-related tools	`io`	input/output routines
`labeling`	labeling algorithms	`level`	point-wise operators on levels
`linear`	linear operators	`literal`	definitions of literals
`logical`	logical operators	`make`	routines to make objects
`math`	mathematical functions	`metal`	static hard-core (metallic) tools
`morpho`	mathematical morphology	`norm`	norms and related distances
`pw`	tools to point-wise expressions	`set`	mathematical set routines
`tag`	some tags	`test`	testing routines
`trace`	tracing helpers	`trait`	definitions of traits
`util`	miscellaneous utilities	`value`	types of values
`win`	windows		

Why Choosing MILENA?

- rather different...
- a strong potential
- you want to focus on what you do,
  not on implementation details about how to do it
- you have not yet found a library to easily process your particular types of data

Dev status

- alpha
- some cleanings remain to be done
- an intensive test phase is upcoming...
- rough documentation (yet in progress)

The Running Example

A class that represents:

- a discrete point of the 2D plane
- a node of a square grid
- a point of a “classical” 2D image
- basically a couple of integer coordinates
- namely `point2d`

Outline

Class v. Object

Class

A class is a type that describes at the same time both data and behavior.

- the data are described by attributes
  (equiv.) structure fields of a `struct`
- the behavior is described by methods
  (equiv.) procedures/functions attached to the class

Object v. Class

Object

An object is an instance of a class.

- the object data are a set of values: the state of the object
- the object behavior is what happens at run-time when a method is called on that particular object

Getting an Object

Constructor

A constructor is a special method to instantiate a class / to get an object.

- it allows initializing the state of this object

Example

- `point2d` is a class
- `p` is a variable that represents a point object
- this variable designates one particular 2D point at a given couple of coordinates: row and column.

```
point2d p(5,1); // construction of an object
std::cout << p << std::endl; // print this object on the std output
```

gives: `(5,1)`
meaning that `p` represents the point of the 2D grid located at row 5 and column 1.

Accessing the State of an Object

- data are usually protected / hidden from the user
- reading and modifying them is performed through method calls

Here is a call to the method `row()` defined in the `point2d` class:

```
std::cout << p.row() << std::endl; // gives: 5
```

`p` is the object targeted by the method call: we want to print the row of this particular point.

Methods are just like C functions!

The previous example is just the C++ equivalent of this C code:

```
#include <stdio.h>

struct point2d {
  int row, col;
};

int get_row(struct point2d p) {
  return p.row;
}

int main() {
  struct point2d p;
  printf("%d\n", get_row(p));
  return 0;
}
```

Modifying the State of an Object (1/2)

modifying an object is performed through method calls:

```
p.row() = 7; // method call
std::cout << p.row() << std::endl; // now gives 7
```

the C equivalent of this method would be:

```
void set_row(struct point2d* p, int r) {
  assert(p != 0);
  p->row = r;
}

int main() {
  struct point2d p;
  set_row(&p, 7);
  return 0;
}
```

note that `p.row() = 7` looks more natural than `set_row(&p, 7)`.

Modifying the State of an Object (2/2)

accessing and modifying through method calls allow for some control:

- one cannot do anything with an object
- especially putting it in an invalid state

imagine that `ima` is a 3×3 image (starting from (0,0))
trying to access the image value at point `p`, like with:

```
std::cout << ima(p) << std::endl; // remind that p is at a row 7...
```

will hopefully produce an error at run-time!

Operators

In C++ some operators can be re-defined to get a high expressiveness in client code.

- "`ima(p)`" is a call to the parenthesis operator defined as a method in every image class
  - that's great, an image looks like a function from points to values
- `2 * p` calls a multiplication operator defined as a procedure (function):
  - this way one can easily use arithmetics over points
  - the result is a 2D point which coordinates are twice those of `p`

Operators are very convenient!

Outline

Reference v. Pointer

A reference looks like a pointer, yet:

- without the pointer notations
  - no need to take the address (with &) of an object
  - no pointer arithmetics
  - no $->$ to access members
- it always designates the same object
  - one can reuse a pointer and make it points elsewhere, that's not the case for a reference
  - it is like a "constant pointer"
    "`int*const`", not "`int*`"
  - it has to be initialized

Example (1/2)

This C program:

```
void set_row(point2d* p, int r) {
  p->row = r;
}
// used with:
point2d p;
set_row(&p, 7);
```

can be re-written as:

```
void set_row(point2d& p, int r) {
  p.row = r; // no "->" here
}
// used with:
point2d p;
set_row(p, 7);
```

Example (1/2)

and this one:

```
int get_row(point2d p) { // copy of a point at procedure call
  return p.row;
}
```

is better written as:

```
int get_row(const point2d& p) {
  return p.row;
}
// used with:
point2d p;
int i = get_row(p);
```

which avoids the copy of a point at function call.

References rock!

Realize that with:

```
class point2d {
public:
  int& row() { return row_; }
  // ...
private:
  int row_, col_;
};
```

one can write:

```
point2d p;
p.row() = 5;
```

so it really performs `p.row_ = 5`

A few Remarks

- the attribute `row_` of the class `point2d` is not accessible from the user
  - thanks to the keyword `private`
  - writing `p.row_` outside this class is not allowed (does not compile)
- the method `row()` is accessible (keyword `public`)
  - in the method body we have some room to add code
  - a simple access to data can perform some clever stuff that you do not really have to know (neither want to)!

Outline

The “Is-A” Relationship

Inheritance

The inheritance between classes maps the “is-a” relationship.

For instance, since we can say that a rabbit is an animal:

- it is safe to make the `rabbit` class inherits from the `animal` one
- we also say that:
  - `rabbit` derives from `animal`
  - `animal` is a base class for `rabbit`

In C++ we write:

```
class animal { ... };
class rabbit : public animal { ... };
```

Using the "Is-A" Relationship

When one wants to have a procedure to feed an animal, one can write:

```
void feed(animal& a) {
  ...
}
```

then the following use is valid

```
int main() {
  rabbit r;
  feed(r); // works fine since a rabbit "is-an" animal
  ...
}
```

One Object but Several Variables and Types (1/2)

Considering only the `feed` routine:

```
void feed(animal& a) {
  ...
}
```

we can say that:

- the variable `a` can represent an object being of any type deriving from `animal`
- it may be a `rabbit`
  - yet we do not really know!
  - it might be a `sheep` instead...

This routine is general since it can work on objects of different types.

One Object but Several Variables and Types (2/2)

Now considering the entire program, with:

```
int main() {
  rabbit r;
  feed(r); // first call
  sheep s;
  feed(s); // another call
}
```

- the variable `r` represents an object whose type precisely `rabbit`

  we say that it is the exact type behind this variable
- for the first call to `feed`, we know that `a` represents a rabbit

  during this execution, the exact type of `a` is `rabbit`

Static type

Static type

- A variable is declared with one type.
- This type can be read in the code; it is known at compile-time.
- For instance, in “`animal& a`”: `a` is an animal.

The variable type is said to be the static type.

Exact type

Exact type

- A variable represents an object.
- Its static type can be a *base* class (like in “animal& a”)
- In that case
  - at compile-time: there are many possible types of objects represented
  - at run-time: there is one object represented so just one type.

At run-time, the type of the object behind a variable is said to be the exact type.

A Clue to Understand MILENA

About “classical” object-orientation:

- abstractions (like `animal`) lead to poor performance at run-time when involved in intensive scientific code.
- it is due to the fact that the exact type is lost (the `virtual` keyword has an effective cost)

The clue:

- genericity leads to dedicated code, thus it is efficient at run-time
- though we really want abstractions!

Outline

Methods and Attributes

```
class point2d {
public:
  point2d(int r, int c) { // constructor
    row_ = r; col_ = col;
  }
  int& operator[](unsigned i) {
    assert(i < 2);
    return i == 0 ? row_ : col_;
  }
  int row() const { return row_; }
  int& row() { return row_; }
  // ...
private:
  int row_, col_;
}
```

Sample use:

```
point2d p(5, 1);
assert(p[0] == p.row());
```

Values and Typedefs

```
class point2d
{
public:
  enum { dim = 2 }
  typedef int coord;
  // ...
}
```

Sample use:

```
std::cout << point2d::dim << std::endl; // gives 2
point2d::coord c; // c is an int
```

At first glance, that seems weird to equip this class with `dim` and `coord` (but is is <u>not</u>!)

Associated Types

Associated Type

A typedef (type alias) defined in a class (e.g., `coord` in `point2d`) is called an "associated type."

We have defined macros to access those types:

- given a type T, `mln_something(T)` gives the associated type `something` defined in `T`
- example of use: `mln_coord(point2d)`
- we can see `mln_coord` like a function that takes a type and returns a type

`mln_something(T)` is for a `template` d piece of code,
whereas `mln_something_(T)` is for a non-`template` d code

Outline

About Naming

we say	a C or C++ user says
attribute	a field (C) or member (C++)
procedure	function
method	a member function

In the following:

routine

A routine designates either a procedure (function) or a method (a member function).

A rationale for Genericity

Suppose that you want a routine that computes twice its input:

```
int twice(int i) { return 2 * i; }
```

Suppose now that you want the "`twice`" operation to work also with values of type `float`.

- you can rely on <u>overloading</u>
- that is the ability of defining several versions of a function
  - having the <u>same</u> name
  - but different signatures

Precisely, you write:

```
int twice(int i) { return 2 * i; }
float twice(float f) { return 2 * f; }
```

Overloading is Limited

This code is quite poor:

- it is redundant
  - tedious to write (copy-paste, many lines at the end)
  - thus error-prone
- it is still limited to `int` and `float`
- so it is not re-usable!
  - understand that "`twice`" should be able to work with `point2d` too

Nevertheless overloading is great; think of `operator*`...

We Want Genericity

Actually

for every type `T`, `twice` a value `t` of type `T` returns `2 * t` which is of type `T`

So this procedure definition looks like

```
// ...
T twice(T t) {
  return 2 * t;
}
```

except that we have to say first what `T` is:

```
template <typename T>
T twice(T t) {
  return 2 * t;
}
```

Syntax of Genericity

In **template** <**typename** T> T twice(T t)

- the declaration "`<typename T>`" is very similar to the one of the procedure argument "`(T t)`"
- the nature of `t` is `T`, the nature of `T` is `typename` (so it designates a type)
- the C++ keyword introducing a generic piece of code is `template`
  - it can be read as "for all" (the universal quantifier ∀) so you read here: "for all <type T>, we have..."
  - the definition (symbolized by "...") follows a classical C++ syntax
- yet the major difference is that:

  `t` is valued at run-time, whereas `T` is valued at compile-time

Use of a Generic Procedure

With

```
int main() {
  int i = 1;
  int j = twice(i); // Calls twice with T being int
  float pi = 3.14;
  float two_pis = twice(pi); // Calls another "version" of twice with T being float
}
```

Once this program is compiled

- two different versions of `twice` cohabits:
  - `int twice(int t) return 2 * t;` and
  - `float twice(float t) return 2 * t;`
- so it is not so different than overloading

except that:

this generic definition of `twice` is reusable

Generic Procedures and Reusability

Precisely

- the generic definition of a procedure is written once
- and is possibly usable in a large number of different versions

If the client wants to write this kind of use:

```
point2d p(2,3), pp;
pp = twice(p);
std::cout << pp << std::endl; // writes (4,6)
```

it also works!

Exercise

Consider this mathematical parameterized function:

$$\forall a \in \mathbb{Z},\ f_a : \left\{ \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \mapsto & \cos(ax) \end{array} \right.$$

- how can it be translated in C++?
- how can a call to such a function work?

Outline

A First Generic Class

We want to define a class to represent couples of values, whatever their respective type is; it gives:

```
template <typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
};
```

A sample use is:

```
pair<float, int> c;
c.first = 3.14, c.second = 3;
```

Another possible use is:

```
pair<bool, point2d> c;
c.first = true, c.second = p;
```

Exercise

Consider this:

```
template <typename T1, typename T2>
struct pair
{
  template <typename S>
  void operator*=(S scalar)
  {
    first *= scalar;
    second *= scalar;
  }
  T1 first;
  T2 second;
};
```

- explain what you see
- then write a program to use this class

Forewords (1/2)

We want some arithmetics over points:

- a “delta-point” is a difference between two points
- a point + (plus) a delta-point gives a point
- the addition (resp. subtraction) of a couple of delta-points gives a delta-point.

For instance

```
point2d p(4, −1);
dpoint2d dp(1, 2); // dpoint is “delta-point” for short
std::cout << (p + dp) << std::endl; // gives (5, 1)
```

Forewords (2/2)

Our objectives:

- write the `operator+` (resp. '`-`') routine corresponding to

  "a `point2d` + a `dpoint2d` $\mapsto$ a `point2d`"

- understand that we actually want:

  "any `point P` + a compatible `dpoint D` $\mapsto$ a `point P`"

  for instance with `P` and `D` being respectively `point3d` and `dpoint3d`
- make different versions of operators cohabit...

Outline

Overloading

```
point2d operator+(const point2d& p, const dpoint2d& dp)
{
  point2d q(p.row() + dp.row(), p.col() + dp.col());
  return q;
}
point3d operator+(const point3d& p, const dpoint3d& dp)
{
  point3d q;
  for (unsigned i = 0; i < 3; ++i) q[i] = p[i] + dp[i];
  return q;
}
```

What do you think of that?

A Solution with Genericity

```
template <typename P, typename D>
P operator+(const P& p, const D& dp)
{
  P q;
  for (unsigned i = 0; i < P::dim; ++i)
    q[i] = p[i] + dp[i];
  return q;
}
```

Continuing with Genericity

Now the subtraction:

```
template <typename P>
? operator−(const P& p1, const P& p2)
{
  ? dp;
  for (unsigned i = 0; i < P::dim; ++i)
    dp[i] = p1[i] − p2[i];
  return dp;
}
```

What shall we write instead of the question mark?

Solution

```
template <typename P>
mln_dpoint(P) operator-(const P& p1, const P& p2)
{
  mln_dpoint(P) dp;
  for (unsigned i = 0; i < P::dim; ++i)
    dp[i] = p1[i] - p2[i];
  return dp;
}
```

Make Things Better

In the addition:

```
template <typename P, typename D>
P operator+(const P& p, const D& dp)
{
  P q;
  // accessing the dimension is really useful:
  for (unsigned i = 0; i < P::dim; ++i)
    q[i] = p[i] + dp[i];
  return q;
}
```

How can we ensure that the delta-point type `D` really corresponds to `P`? (we really do not want `P` and `D` resp. being `point3d` and `dpoint2d`!)

Solution

```
template <typename P>
P operator+(const P& p, const mln_dpoint(P)& dp)
{
  ...
}
```

Cohabitation is Hard

Consider:

```
template <typename D>
D operator+(const D& dp1, const D& dp2) {
  ... // addition of a couple of delta-points
}

template <typename I>
I operator+(const I& ima1, const I& ima2) {
  ... // addition of a couple of images
}
```

What is the problem? (Hint: read both signatures out loud)

Outline

“Classical” Object-Orientation

In “classical” OO programming (OOP), we would write:

```
Dpoint operator+(const Dpoint& dp1, const Dpoint& dp2) {
  ... // addition of a couple of delta-points
}

Image operator+(const Image& ima1, const Image& ima2) {
  ... // addition of a couple of images
}
```

which is clearly not ambiguous (but slow at run-time...) where `Dpoint` and `Image` are abstract classes.

Between OOP and Genericity

Can we try to mix OOP and Generic Programming (GP)?

That is, getting something between:

```
Dpoint operator+(const Dpoint& dp1, const Dpoint& dp2) {
  ...
}
```

and

```
template <typename D>
D operator+(const D& dp1, const D& dp2) {
  ...
}
```

MILENA Paradigm

Yes:

```
template <typename D>
D operator+(const Dpoint<D>& dp1, const Dpoint<D>& dp2) {
  ...
}
```

here `dp1` is a delta-point (`Dpoint`) of type `D`

- it is not ambiguous at compile-time
- it is efficient at run-time

Reading a MILENA Signature of Routine

```
template <typename D>
D operator+(const Dpoint<D>& dp1, const Dpoint<D>& dp2) {
  ...
}
```

The `operator+` takes a couple of `Dpoint` of type `D` and returns the same type.

Outline

Reading Again

In MILENA:

> when we have something like “Point<P>& p”
> it means that `p` is actually a point of type `P`

For instance, if the type of `p` is `point2d`, then “another” type for `p` is Point<point2d>.

So

- in “Point<P>& p”, `P` is the exact type of `p`
- a type of point `P` derives from the abstraction Point<P>

Conclusion

More generally:

Abstractions and Exact Types

When a concrete class `T` is related to an abstraction named `Abstraction`, then `T` derives from Abstraction<T>.

Every abstraction in MILENA has exactly one parameter, which represents its exact type.

Examples

the class	derives from (“is a”)
point2d	Point< point2d >
point3d	Point< point3d >
image2d<**float**>	Image< image2d<**float**> >
image3d<**int**>	Image< image3d<**int**> >
win::rectangle	Window< win::rectangle >
box2d	Point_Set< box2d >

Excerpts from MILENA

```
namespace level
{
  template <typename I, typename J>
  void paste(const Image<I>& data, Image<J>& destination);
}

namespace morpho
{
  template <typename I, typename W>
  mln_concrete(I) erosion(const Image<I>& input, const Window<W>& win);
}

namespace convert
{
  template <typename S>
  array_p<mln_point(S)> to_array_p(const Point_Set<S>& pset);
}
```

Outline

A Hierarchy

We have `dpoint2d`

```
template <typename E>
class Dpoint
{};

class dpoint2d : public Dpoint< dpoint2d >
{
public:
  int& operator[](unsigned i);
  ...
};
```

A Troubleshooting

This routine is almost (so not) correct:

```
template <typename D>
D operator+(const Dpoint<D>& dp1, const Dpoint<D>& dp2)
{
  D dp;
  for (unsigned i = 0; i < D::dim; ++i)
    dp[i] = dp1[i] + dp2[i];
    // above: dp[i] is OK
    // but dp1[i] and dp2[i] do not compile!
  return dp;
}
```

because the `operator[]`

- is defined in concrete classes like `dpoint2d`
- but not in the abstract class Dpoint<dpoint2d>

The `exact` Routine

This updated routine works fine:

```
template <typename D>
D operator+(const Dpoint<D>& dp1_, const Dpoint<D>& dp2_)
{
  const D& dp1 = exact(dp1_); // Cast to the exact type.
  const D& dp2 = exact(dp2_);
  D dp;
  for (unsigned i = 0; i < D::dim; ++i) dp[i] = dp1[i] + dp2[i];
  return dp;
}
```

Exact

The "`exact`" routine allows getting a variable with the exact type of an object.

Remember about Genericity

When this routine is called with `dpoint2d` objects, then the compiled version is this one:

```
dpoint2d operator+(const Dpoint<dpoint2d>& dp1_, const Dpoint<dpoint2d>& dp2_)
```

and its definition is finally:

```
{
  dpoint2d dp;
  dp[0] = dp1[0] + dp2[0];
  dp[1] = dp1[1] + dp2[1];
  return dp;
}
```

Conclusion

This routine:

- is generic
  it works for any delta-point type
- is fast, you cannot get more efficient code
- is user-friendly, just write "`dp1 + dp2`" to add a couple of delta-points

Exercise 1

Explain the code below:

```
template <typename I>
void set(const Image<I>& ima_, const mln_point(I)& p, const mln_value(I)& v)
{
  const I& ima = exact(ima_);
  ima(p) = v;
}
```

What do have we in image classes?

Exercise 2

Explain the code below:

```
template <typename I, typename H>
void oper(Image<I>& f_, const Function_v2v<H>& h_)
{
  I& f = exact(f);
  const H& h = exact(h_);
  mln_piter(I) p(f.domain());
  for_all(p)
    f(p) = h(f(p));
}
```

What do have we in image classes?