
Rapport de projet : C++ avancé

# Sujet 1 : Polymorphisme statique/dynamique pour Python

Matthieu ALVERGNAT, Xavier FACQUEUR, David GHIASSI, Léo MORICE, Jérémy TRAN

Juillet 2020

# Table des matières

# 1 Structure du projet

Le projet contient un conanfile.txt qui permet d'installer Pylene (dans sa version dynamique) et pybind11. Conan va ainsi générer du code CMake pour trouver les bibliothèques et ajouter les bons flags de compilation/linker. Ce code CMake est ensuite inclus dans notre fichier CMakeLists.txt via le code suivant :

```
include(${CMAKE_BINARY_DIR}/conan_paths.cmake)
```

CMake génère ensuite un Makefile, ce dernier permettant de compiler nos fichier code (.cc) en fichier objets (.o) pour au final en faire une bibliothèque dynamique.

# 2 Fonctions de conversions

Les fonctions de conversions sont obligatoires pour avoir un projet fonctionnel. En effet, le type numpy array est utilisable en Python et en C++ grâce a pybind11, mais le type mln : :ndbuffer_info n'est utilisable qu'en C++ et la bibliothèque Pylene utilise uniquement ce type.

Nos fonctions de conversions gèrent 2 types d'images : les images codées en RGB 24 bits, et les images codées avec 8 bits par pixels en niveau de gris.
Voici la fonction de conversion numpy -> mln : :ndbuffer_image située dans src/convert.cc :

```
mln::ndbuffer_image numpy_to_ndbuffer_image(py::array array)
{
    py::buffer_info buf = array.request();

    /* Remove shape constness */
    int shape[2] = { 0 };
    for (ssize_t i = 0; i < 2; i++)
        shape[i] = buf.shape[i];

    if (buf.ndim == 3)
    {
        is_rgb = true;
        return mln::ndbuffer_image::from_buffer(
            reinterpret_cast<std::byte *>(buf.ptr),
            mln::sample_type_id::RGB8,
            2,
            shape,
            nullptr,
            false
        );
    }

    is_rgb = false;
    return mln::ndbuffer_image::from_buffer(
        reinterpret_cast<std::byte *>(buf.ptr),
        mln::sample_type_id::UINT8,
        2,
        shape,
        nullptr,
        false
    );
}
```

Nous partons du principe que si le py : :array en entrée est en 3 dimensions, il s'agit d'une image RGB, sinon s'il est en 2 dimensions, c'est une image en niveaux de gris.
Le booléen "is_rgb" est une variable globale qui permet à l'autre fonction de conversion (mln : :ndbuffer_image -> numpy) de savoir quel type d'image reconstituer.

# 3 Wrapper dynamique autour des éléments structurants

Le but étant de réussir un type erasure, nous avons donc créé une classe "fille" qui est templatée pour ainsi pouvoir accueillir les différents éléments structurants.

```
template <typename T>
class se_template
{
    public:
        se_template(const T& element);

        T get_element() const;
        enum pln::s_element get_type() const override;

    private:
        T element_;
};
```

Pour pouvoir traiter toutes les templates comme une seule et même classe, nous avons créé une classe "mère".

```
class se_t
{
    public:
        virtual enum pln::s_element get_type() const;
};
```

Finalement, notre classe "fille" va hériter de cette classe "mère" qui elle, n'est pas templatée.

```
template <typename T>
class se_template : public se_t
{
    // ...
};
```

Ainsi, nous avons résolu le type erasure.

Dans notre classe templatée, nous avons 2 méthodes qui vont nous donner la valeur qu'elle contient (la valeur du rectangle si la classe est templatée sur mln : :rect2d) ainsi que le type de la valeur qu'elle contient (ou le type sur laquelle la classe est templatée) afin de pouvoir caster celle-ci dans le futur.

Ce type est renvoyé sous la forme d'une énumération :

```
enum class s_element
{
    DISC,
    RECTANGLE,
    PERIODIC_LINE,
    MASK,
    UNDEFINED
};
```

# 4 Wrapper dynamique autour des algorithmes

Ici, notre but est d'appeler les algorithmes servant de filtres avec notre mln : :ndbuffer_image et notre élément structurant. Nous utilisons la même méthode pour les 4 algorithmes, nous allons donc en présenter un seul ici. Ainsi, nous avonsle prototype de la fonction.

```
mln::ndbuffer_image pln::morpho::dilation(mln::ndbuffer_image input,
                                          const pln::se_t& se);
```

Nous passons en argument l'élément structure en const référence afin de ne pas perdre les informations de la classe templatée qui hérite de pln : :se_t.
Notre première étape est de caster le mln : :ndbuffer_image en image2d, le problème étant de savoir si l'image2d contient des uint8_t ou des mln : :rgb8 (respectivement niveaux de gris ou couleurs).

```
auto *image_rgb = input.template cast_to<mln::rgb8, 2>();
mln::image2d<uint8_t> *image_grey = nullptr;

if (!image_rgb)
{
    image_grey = input.template cast_to<uint8_t, 2>();

    if (!image_grey)
        throw std::invalid_argument(".....");
}
```

On va donc lancer une exception si le cast ne fonctionne pour aucune des situations précédentes. Nous avons maintenant notre image2d prête à recevoir un filtre.
La deuxième étape est de récupérer la valeur de l'élément structurant stockée dans la variable se de type const se_t&.

```
auto type_id = se.get_type();
```

Ici, nous récupérons le type de la valeur contenu afin du pouvoir caster la variable se et récupérer la valeur. Cette méthode nous renvoie une valeur de l'énum s_element présenté plus tôt.

Nous allons donc effectuer un switch parmi les valeurs de cette enum pour effectuer le bon cast. Comme pour tout à l'heure, nous allons présenter un seul cas, puisque les 3 autres sont identiques.

```
switch (type_id)
{
    case s_element::DISC:
    {
        if (image_rgb)
        {
            auto res = mln::morpho::dilation(*image_rgb,
                dynamic_cast<const pln::se_template<mln::se::disc> *>(&se)
                    ->get_element());
            mln::ndbuffer_image output = res;
            return output;
        }

        auto res = mln::morpho::dilation(*image_grey,
            dynamic_cast<const pln::se_template<mln::se::disc> *>(&se)
                ->get_element());
        mln::ndbuffer_image output = res;
        return output;
    }

    /* Other structuring elements */

    default:
```

```
        throw std::invalid_argument(".....");
        return input;
}
```

Nous différencions 2 cas, le cas avec image en niveaux de gris, et le cas avec image en couleur. Le principe est le même, nous avons juste des variables différentes selon le type d'image.

Nous faisons donc un cast dynamique vers la classe "fille" avec le bon type templaté et nous récupérons la valeur qui s'y trouve pour la donner au filtre. Ensuite, nous faisons un cast implicite vers mln : :ndbuffer_image pour renvoyer notre image avec l'algorithme appliqué dessus.

Bien évidemment, nous avons un cas par défaut qui est en fait un cas qui ne devrait pas arriver, donc nous renvoyons une exception.

# 5 Binding python

Nous avons dû faire en sorte que l'utilisateur mette un numpy en entrée et reçoive un numpy en sortie. Voici une des fonctions qui a ce rôle :

```
py::array pln::morpho::closing_wrapper(py::array array, const pln::se_t& se)
{
    return ndbuffer_image_to_numpy(
        pln::morpho::closing(numpy_to_ndbuffer_image(array), se)
    );
}
```

Pour pouvoir être appelée depuis python, nous avons du utiliser pybind11. Nous créons tout d'abord un module "pylene", l'utilisateur pourra alors l'importer dans son code python. Puis nous créons un sous-module "morpho" pour les filtres, et un autre sous-module "se" pour les éléments structurants. Ensuite nous associons par exemple la méthode "dilation" du sous-module morpho a sa fonction en C++, pybind11 se charge de tout le travail.

Pour les éléments structurants, le type de retour n'est pas connu de python. Il nous a donc fallu binder les classes C++ avec des classes Python.

```
PYBIND11_PLUGIN(pylene)
{
    py::module m("pylene", "pylene␣extension");
    auto morpho = m.def_submodule("morpho");

    morpho.def("dilation", &pln::morpho::dilation_wrapper);
    /* ... same for erosion, opening and closing ... */

    auto se = m.def_submodule("se");

    py::class_<pln::se_t> base(se, "se_t");

    py::class_<pln::se_template<mln::se::disc>>(se, "se_disc", base);
    /* ... same for rect2d, periodic_line2d and mask2d ... */

    se.def("disc", &pln::se::disc);
    se.def("rectangle", &pln::se::rectangle,
                        py::arg("width"), py::arg("height"));
    se.def("periodic_line", &pln::se::periodic_line);
    se.def("mask", &pln::se::mask);

    return m.ptr();
}
```

# 6 Testsuite python

Le principe de notre testsuite est simplement de lancer un par un chaque filtre avec chacun des éléments structurants, d'abord avec une image en niveaux de gris, puis avec une image en couleur. Certains tests sont la uniquement pour tester que la construction des éléments structurants fonctionne. La testsuite est lancée avec la commande "make test". Elle affiche chaque test et son résultat (OK ou KO), puis affiche un résumé des tests (nombre de OK ainsi que nombre de KO).

# 7 Avancement du projet

Nous avons rencontré un problème lors de l'initialisation des éléments structurants en python. En effet, pour le mask et la periodic_line, nous devons manipuler de std : :initializer_list en C++ mais nous n'avons pas réussi à la transcrire sous python.