Plan

- Introduction, rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générique
- Compléments

Classes dupliquées

```
class PileD {
public:
  void empile( double v );
  // ...
};
class PileL {
public:
  void empile( long val );
  // ...
};
class PileS {
public:
  void empile( std::string val );
  // ...
};
```

Modèle de classe

```
template< class T >
class Pile {
public:
   void empile( T val );
   // ...
};

template< typename T >
void Pile< T >::empile( T val ) {
   // ...
}
```

Dans le fichier d'entête

En général, dans le fichier d'entête

Utilisation d'un modèle de classe

```
void foo() {
   Pile< int > p1{ 20 };
   Pile< Nombre > p2{ 30 };
   Pile< Pile< double >> p3{ 10 };

   // p3.empile( p2 );
   p3.empile( Pile< double >( 8 ));
   p3.sommet().empile( 3.0 );
   // ...
}
```

Démo

GitLab

cours/generique/Pile_v6.cpp

Paramètres de généricité

```
template< typename T, unsigned TAILLE = 100 >
class Pile {
  // ...

private:
  T tab_[TAILLE];
  unsigned index_ = 0;
};
```

Démo

GitLab

cours/generique/Pile_v7.cpp

Modèle de fonction

```
// version C
#define MAX(A, B) ((A)>(B)?(A):(B))

template< typename T >
inline T max( T val1, T val2 ) {
  return val1 > val2 ? val1 : val2;
}

void foo() {
  int i{ 5 }, j{ 3 }, k{ max( i, j )};
  Nombre n1{ 12 }, n2{ 16 };
  Nombre n3{ max( n1, n2 )};
}
```

Spécialisation d'un modèle

```
template<>
inline char const * max(
        char const * val1, char const * val2 ) {
  return std::strcmp( val1,  val2 ) > 0
                    ? val1 : val2;
}

void foo() {
  char const * s{ max( "hello", "bonjour" )};
}
```

Modèle de méthode

```
template< typename T, unsigned TAILLE >
class Pile {
  // ...
  template< unsigned SIZE >
  Pile( const Pile< T, SIZE > & p );

  template< typename U, unsigned SIZE >
  Pile & operator=( const Pile< U, SIZE > & p );
  // ...

};
```

Démo

GitLab

cours/generique/Pile_v8.cpp

La généricité comme langage fonctionnel

```
template< unsigned N >
struct Factorial {
  static constexpr unsigned value{
          N * Factorial< N - 1 >::value };
};

template<>
struct Factorial< 0 > {
  static constexpr unsigned value{ 1 };
};
```

Démo

GitLab

cours/generique/Factorial.cpp

Polymorphisme

- Universel
  - Héritage
  - Paramétrique

- Ad Hoc
  - Surcharge
  - Conversion

Standard Template Library

- La STL désigne la partie de la bibliothèque standard C++ qui fournit des conteneurs et les services associés.
- Son principal auteur est Alexander Stepanov, un ingénieur chez HP.
- Elle fut proposée en 1994 lors de la normalisation du langage C++.
- Elle utilise massivement la généricité, elle est à l'origine de la puissance expressive de cette partie du langage.

Conteneurs et itérateurs en C (1)

CentraleSupélec

```
void foo()
{
  int tab[10];

  for( int i = 0; i < 10; ++i ) {
    tab[i] = 1;
  }
}
```

Conteneurs et itérateurs en C (2)

```
void foo()
{
  int tab[10];

  for( int * p = tab; p != tab + 10; ++p ) {
    *p = 1;
  }
}
```

Conteneurs et itérateurs en C++ (1)

```
void foo()
{
   std::vector< int > tab( 10 );

   for( std::vector< int >::iterator
                  p{   tab.begin() };
                  p != tab.end();
                ++p ) {
      *p = 1;
   }
}
```

Parenthèses ici !

Conteneurs et itérateurs en C++ (2)

```
void foo()
{
  std::vector< int > tab( 10 );

  for( auto p{   tab.begin() };
             p != tab.end();
           ++p ) {
    *p = 1;
  }
}
```

Conteneurs et itérateurs en C++ (3)

```
void foo()
{
  std::vector< int > tab( 10 );

  for( auto & p : tab ) {
    p = 1;
  }
}
```

Algorithme (1)

```
template< typename Container >
void mettre_a_1( Container & c )
{
  for( auto & p : c ) p = 1;
}

void foo()
{
  std::vector< int > tab( 10 );

  mettre_a_1( tab );
}
```

Algorithme (2)

```
template< typename Iter >
void mettre_a_1( Iter begin, Iter end )
{
  for( Iter p{ begin }; p != end; ++p ) {
    *p = 1;
  }
}

void foo()
{
  std::vector< int > tab( 10 );

  mettre_a_1( std::begin( tab ),
              std::end  ( tab ));
}
```

Généralisation de l'algorithme (1)

```
template< typename Iter, typename T >
void mettre_a_val( Iter beg, Iter end, T val )
{
  for( Iter p{ beg }; p != end; ++p ) {
    *p = val;
  }
}

void foo()
{
  std::vector< int > tab( 10 );

  mettre_a_val( std::begin( tab ),
                std::end  ( tab ), 1 );
}
```

Généralisation de l'algorithme (2)

```
template< typename Iter, typename F >
void apply( Iter beg, Iter end, const F & fun ) {
  for( Iter p{ beg }; p != end; ++p ) fun( *p );
}

void mettre_a_1( int & v ) { v = 1; }

void foo()
{
  std::vector< int > tab( 10 );

  apply( std::begin( tab ),
         std::end  ( tab ), &mettre_a_1 );
}
```

Objet fonction

```
template< typename T >
struct mettre_a_val {
  mettre_a_val( T val ) : val_{ val } {}

  void operator()( T & t ) const { t = val_; }
  const T val_;
};

void foo()
{
  std::vector< int > tab( 10 );

  apply( std::begin( tab ),
         std::end  ( tab ), mettre_a_val( 1 ));
}
```

Démo

GitLab

cours/generique/Algo.cpp

Concepts de la STL

- Nous avons vu les quatre principaux concepts de la STL :
  - les **conteneurs** permettent de stocker des objets et d'y accéder ; la complexité temporelle des opérations associées les différencient,
  - les **itérateurs** permettent de parcourir les objets d'un conteneur,
  - les **algorithmes** permettent d'effectuer des traitements sur une partie d'un conteneur identifiée avec deux itérateurs,
  - les **objets fonctions** permettent de spécifier les traitements des algorithmes.
- Un cinquième concept, les **adaptateurs**, modifient l'interface et/ou le comportement des conteneurs et itérateurs.

Les conteneurs

- Conteneurs séquentiels
  - vector< T >
  - array< T, N >
  - deque< T >
  - forward_list< T >
  - list< T >
- Adaptateurs
  - stack< T >
  - queue< T >
  - priority_queue< T >
- Conteneurs associatifs
  - map< K, V >
  - set< K >
  - multimap< K, V >
  - multiset< K >
  - unordered_map< K, V >
  - unordered_set< K >
  - unordered_multimap< K, V >
  - unordered_multiset< K >

Types et méthodes des conteneurs

```
class container {
public:
    using value_type = ...;
    using iterator = ...;
    using const_iterator = ...;
    // ...
    size_t size();
    bool empty();
    void clear();
    // ...
    iterator begin();
    iterator end();
    const_iterator cbegin();
    const_iterator cend();
    // ...
};
```

Les itérateurs

- Catégories
  - `forward_iterator` (`*`, `++i`, `i++`)
    - `input_iterator`
    - `output_iterator`
  - `bidirectional_iterator` (`--i`, `i--`)
  - `random_access_iterator` (`i + n`, `i - n`, `i[n]`)

- Adaptateurs
  - `reverse_iterator`
  - `back_inserter_iterator`
  - `front_inserter_iterator`
  - `inserter_iterator`

Les algorithmes

- Algorithmes non modifiants
  - `for_each`, `count`, `find`, `all_of`, `min_element`, accumulate…
- Algorithmes modifiants
  - `copy`, `move`, `fill`, `transform`…
- Partitionnements et tris
  - `partition`, `sort`, `stable_sort`…
- Autres algorithmes sur des conteneurs ayant certaines propriétés
  - binary_search, lower_bound, set_union…