

# La langage C++

M4105C

Sylvain Jubertie  
[sylvain.jubertie@univ-orleans.fr](mailto:sylvain.jubertie@univ-orleans.fr)

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

# Organisation du module

## 8 semaines

- Semaines 1-3 : Cours/TD/TP
- Semaine 4 : Cours/TD/Examen en séance de TP
- Semaines 5-7 : Cours/TD/TP
- Semaine 8 : Cours/TD/Examen en séance de TP

## Évaluation surprise

Suivant attitude des étudiants...

## Historique

Langage développé par Bjarne Stroustrup au début des années 1980 comme extension du C : *C with classes*, langage standardisé

## Caractéristiques

### Langage multi-paradigmes

- programmation impérative
- programmation objet
- programmation générique

## Domaines

- embarqué : Arduino, micro-contrôleurs
- systèmes d'exploitation : Windows ! (noyau en C)
- jeux vidéo : OpenGL, Vulkan, moteurs physiques
- calcul : banque, finance, simulations scientifiques, ...
- ...

## Ressources en ligne

- [isocpp.org](http://isocpp.org) : standard C++
- [cppreference.com](http://cppreference.com) : API
- Coliru : compilateur C++ en ligne
- Godbolt : compilateur C++ en ligne + asm

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards



hello.cpp

```
#include <iostream>

int main()
{
    std::cout << " Hello C++" << std::endl;
    return 0;
}
```

Compilation avec GCC

```
g++ -o hello hello.cpp
```

Exécution

```
./hello
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Types primitifs

- types entiers signés : `char`, `short`, `int`, `long`, ...
- types entiers non signés : préfixe `unsigned`
- booléen : `bool`
- types flottants : `float`, `double`
- pointeurs
- constantes : modificateur `const`

## Initialisation des variables de types primitifs

### 1 Initialisation par affectation :

```
int a = 5;  
float f = 1.5 f;
```

### 2 Initialisation par constructeur :

```
int a( 5 );  
float f( 1.5 f );
```

### 3 Initialisation uniforme (C++11) :

```
int a{ 5 };  
float f{ 1.5 f };
```

## Pointeurs

Comme en C, mais C++11 a introduit un pointeur nul typé pour éviter certains écueils du C :

```
int a = 5;  
int * p_i = &a;  
int * p_j = nullptr;
```

## Références

Les références permettent de contenir les adresses de variables, comme les pointeurs, mais obligatoirement initialisées à partir d'une instance ou d'une fonction.

Une fois initialisée, une référence ne peut être changée pour référencer un autre objet donc elle est toujours constante et le modificateur `const` n'est donc pas utilisé.

La syntaxe pour accéder à l'instance référée est la même que pour accéder à l'instance.

```
int a = 5;
int & r_a = a;
r_a += 3;
std::cout << r_a << std::endl;
```

## Modificateur `static`

Le mot clé `static` devant une variable permet de modifier :

- la portée d'une variable globale ou d'une fonction à l'unité de traduction ( `translation unit` ) i.e. au fichier `.cpp` : le symbole n'est pas exporté dans la table des symboles.
- le stockage d'une variable locale d'une fonction : la variable locale est stockée globalement mais sa portée est toujours locale.

## Exemple

```
static int a = 123; // Stockage global mais portée limitée au fichier.  
static void fct0 () { ... } // Portée limitée au fichier.  
void fct1 ()  
{  
    static int i = 0; // Stockage global, initialisée à 0, portée locale.
```

## Tableaux statiques (pile)

Déclaration :

```
elementtype tabname[ tabsize ];
```

## Initialisation des tableaux

```
int t0[ 10 ]; // 10 entiers, valeurs non définies  
int t1[ 4 ] = { }; // 0, 0, 0, 0  
float t2[ ] = { 1.0f, 2.0f, 3.0f, 4.0f }; // float[4]  
float t3[ 4 ] = { 1.0f }; // 1.0f, 0.0f, ...
```



## Tableaux et bibliothèque standard

- Privilégier l'utilisation de `std::array` (cf. section Bibliothèque standard) à la place des tableaux statiques C.
- Les tableaux dynamiques sont présentés dans la sous-section *Gestion mémoire dynamique*.

## Chaînes de caractères

Type complexe `std::string` de la bibliothèque standard :

### Initialisation

```
std::string s0 = "Hello_C++";  
std::string s1( "Hello" );  
std::string s2{ "Hello" };
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Fonctions

- comme en C.
- passage par référence en plus : la variable passée par référence est modifiable dans la fonction. Equivalent à un passage par pointeur mais avec une syntaxe plus simple. Une référence ne peut être nulle.

## Exemple

```
void fct( int a ) { ... } // Passage par copie.  
void fct( int * p_a ) { ... } // Passage par pointeur.  
void fct( int & a ) { a = 3; ... } // Passage par référence.
```

## const

Le mot clé `const` peut être appliqué aux arguments d'une fonction.

## Exemple

```
void fct( int const a ); // a: copie non modifiable.
void fct( int const * p_a ); // copie du pointeur modifiable
                             // mais valeur pointée non modifiable.
void fct( int * const p_a );
// copie du pointeur non modifiable mais valeur pointée modifiable.
void fct( int const * const p_a ); // rien n'est modifiable.
void fct( int & a ) // Passage par référence, a modifiable.
void fct( int const & a )
    // Passage par référence, a non modifiable.
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Types complexes

2 méthodes de définitions possibles :

1 struct

2 class

## Structures

Déclaration similaire au C.

Par défaut attributs accessibles publiquement.

### Définition

```
struct Student
{
    unsigned int id;
    std::string firstname;
    std::string lastname;
    // ...
}; // Attention au ; à la fin de la définition !
```



## Instantiation/Accès

```
Student s0;  
s0.id = 3;
```

```
Student s1{ 42, "Arthur", "Accroc" };  
std::cout << s1.firstname << std::endl;
```

## Classes

- Définition similaire à une structure.
- Par défaut attributs privés donc non accessibles de l'extérieur de la classe.
- Possibilité de déclarer les attributs publics : `public`.
- Sinon nécessite des constructeurs/accesseurs pour manipuler les attributs.
- Accès aux attributs publics comme pour une structure.

## Exemple : attributs privés (par défaut)

```
class Student
{
private: // attributs privés
    unsigned int id;
    std::string firstname;
    std::string lastname;
    // ...
};
```

## Exemple : attributs publics

```
class Student
{
public: // attributs accessibles de l'extérieur de la classe.
    unsigned int id;
    std::string firstname;
    std::string lastname;
    // ...
};
```

## Exemple : attributs publics et privés

```
class Student
{
private:
    unsigned int id;
public:
    std::string firstname;
    std::string lastname;
    // ...
};
```

## Encapsulation

Encapsulation = Attributs + Méthodes associés.

## Vie d'une instance

- 1 Instanciation : constructeurs, initialisation des attributs
- 2 Utilisation : méthodes
- 3 Destruction : destructeur, préparation avant libération de la mémoire

## Constructeurs

- Par défaut : sans arguments.
- Avec arguments.
- Par copie : copie des attributs d'une autre instance.
- Par déplacement (move constructor) : appropriation d'une autre instance.

## Exemple

```
class Student
{
public:
    // constructeur par défaut.
    Student() { ... }

    // constructeur avec arguments.
    Student( ... ) { ... }

    // constr. de copie, on ne modifie pas l'original donc const.
    Student( Student const & s ) { ... }

    // move constr., rvalue reference.
    Student( Student && s ) { ... }
```



## Constructeur de recopie

- Si le constructeur de recopie n'est pas définie par défaut, alors un constructeur de recopie par défaut est utilisé : tous les attributs sont copiés (comme pour les structures en C).
- L'instance à copier est passée par référence : `&`.
- Si l'instance à copier n'est pas modifiée (attributs non modifiés) par le constructeur de recopie, alors la déclarer `const`

## Appels entre constructeurs

Un constructeur peut en appeler un autre (C++11).

### Exemple

```
Student( std::string const & firstname ,  
         std::string const & lastname ) ...  
  
Student( std::string const & firstname ,  
         std::string const & lastname ,  
         unsigned short firstinscr )  
: Student( firstname , lastname ) ...
```

## Destructeur

- Appelé automatiquement lors de la sortie de la zone de portée de la variable.
- Pas d'argument.
- Utilisé pour modifier des attributs statiques (compteur), libérer des ressources (mémoire, fichier : RAII), ....

## Exemple

```
...  
// Destructeur  
~Student() { ... }  
...
```

## Exemple

```
...  
{ // Début de la zone de portée.  
  Student s( ... ); // Instanciation  
  std::cout << s.firstname << std::endl;  
  ...  
} // Fin de la zone de portée, appel au destructeur Student
```

## Problème

Lorsque la classe contient des pointeurs, descripteurs de fichiers, ... de manière générale des données dynamiques, la copie par défaut effectue uniquement la copie du pointeur, du descripteur de fichier, ...

Pour effectuer une copie complète (*deep copy*), il est obligatoire de surcharger le constructeur de recopie pour effectuer la copie des données pointées.

## Exemple

```
class A {  
    ...  
    int * p_i;  
    A() {  
        p_i = new int;  
    }  
};  
...  
A a;  
A b( a ); // b.p_i pointe vers la même donnée que a.p_i  
*(a.p_i) = 42;  
std::cout << *(b.p_i) << std::endl; // affiche 42...
```

Mais double free corruption à la sortie du programme...

## Forme de Coplien

Pour effectuer la *deep copy* d'une instance il convient de mettre la classe dans la forme canonique de Coplien qui impose que la classe possède :

- un constructeur par défaut
- un constructeur de recopie
- un destructeur
- un opérateur d'affectation

## Méthodes

Définition comme les fonctions.

Attention, l'instance est accessible par le pointeur **this**.

## Exemple

```
class Student {  
    ...  
public:  
    void setFirstname( std::string const & firstname ) {  
        this->firstname = firstname;  
    }  
};
```



## Méthodes constantes

Les méthodes ne modifiant pas l'état de l'instance doivent être déclarées constantes (pas obligatoire mais...).

### Exemple

```
class Student {  
    ...  
public:  
    unsigned int getId() const {  
        return id;  
    }  
};
```

## Modificateur `static`

En plus de pouvoir se placer devant des variables et des fonctions (C), le modificateur `static` peut être placé devant :

- les attributs d'une classe,
- les méthodes d'une classe.

## Attributs static

- Attributs de classe et non d'instance
- Déclaration dans la classe
- Attributs stockés dans segment data
- Initialisation obligatoire en dehors de la classe

## Exemple

```
class A {  
    ...  
    static int n;  
};  
int A::n = 0;
```

## Méthodes static

- Accès à des attributs statiques
- Appel par nom complet

## Exemple

```
class A {  
public:  
    static void fct() { ... }  
};  
// appel  
A::fct();
```

## Surcharge d'opérateurs

Presque tous les opérateurs peuvent être surchargés :  $+$ ,  $-$ ,  $*$ ,  $=$ ,  $[]$ ,  $()$ , ...

Certains doivent être définis dans la classe, d'autres à l'extérieur, d'autres indifféremment.

- [cppreference.com](http://cppreference.com): operator overloading
- [wikipedia.org](http://wikipedia.org): operators in C and C++

Cela permet de simplifier l'écriture mais doit être utilisé à bon escient et de manière pertinente.

Dans l'exemple suivant, une classe [Matrix](#) permet de définir des matrices.

Pour 2 instances [a](#) et [b](#) de cette classe, cela a un sens d'écrire [a](#) + [b](#) pour effectuer la somme des 2 matrices.

## operator+ dans la classe

Dans ce cas l'opérateur s'applique entre l'instance courante et celle passée en argument.

```
class Matrix {  
    ...  
    Matrix operator+( Matrix const & m ) { ... }  
    ...  
};
```

## operator+ externe

```
Matrix operator+( Matrix const & m0,  
                  Matrix const & m1 )  
{ ... }
```

## `operator=` : opérateur d'affectation

Définition obligatoire dans la classe :

```
class Matrix {  
    ...  
    Matrix & operator=( Matrix const & m ) { ... }  
};
```

L'instance courante prend le contenu de son argument et est retournée.

Cela permet d'écrire : `a = b = c = ...;`

## operator<<

Dans le cas de la surcharge pour l'affichage, l'opérateur modifie et retourne le flux (`std::ostream`) passé en paramètre :

```
std::ostream & operator<<( std::ostream & out, Matrix const & m )  
{  
    ...  
    return out;  
}
```

L'opération n'est pas censée modifier l'instance passée en argument donc `const`.

La matrice ne doit pas être passée par copie donc `&`.



## Déclaration des constructeurs/méthodes/...

- Dans la classe (.hpp).
- En dehors de la classe mais dans le fichier .hpp.
- Dans le fichier .cpp.

## Dans la classe

```
class C {  
    void fct() { ... }  
};
```

## En dehors de la classe, dans le .hpp

```
class C {  
    void fct();  
};  
  
void C::fct() { ... }
```

## Dans un .cpp

Définition dans le .hpp :

```
class C {  
    void fct();  
};
```

Déclaration dans le .cpp

```
#include <....hpp>  
void C::fct() { ... }
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## auto : inférence de type (C++11)

Le spécificateur **auto** permet d'attribuer un type en le déduisant automatiquement. Il permet d'obtenir des codes plus génériques.

### Exemple

```
int a = 5;  
int b = a;  
float x = fct();
```

Le compilateur peut déterminer le type de **b** en fonction de celui de **a**, ou le type de **x** en fonction du type de retour de la fonction **fct**.

```
auto b = a;  
auto x = fct();
```

Par contre on ne peut pas écrire : **auto x**;

## Remarques

**auto** déduit uniquement le type, pas les modificateurs :

```
auto i = 5; // i de type int.  
auto const j = 3; // j de type const int  
auto & k = &i; // k de type int &
```

## Autres utilisations

**auto** peut aussi déduire le type de retour d'une fonction :

```
auto fct() { return 5; }
```

## `decltype` : récupération de types

Le spécificateur `decltype( expr )` permet de récupérer le type d'une expression donnée en argument.

```
int fct() { ... }
```

```
decltype( fct() ) x; // int x;
```

```
int tab[ 10 ];
```

```
decltype( tab[ 0 ] ) e; // int e;
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards



## Structures conditionnelles : if

Comme en C :

```
if( cond )  
{  
    ...  
}  
else  
{  
    ...  
}
```

## if constexpr : C++17

Lorsque l'expression peut être évaluée à la compilation, seul le code du bloc `if` ou `else` est généré par le compilateur :

```
if constexpr( const_expr ) { ... }
```

### Exemple

```
if constexpr( sizeof(int) == 4 )
```

## Structures conditionnelles : **switch**

Comme en C :

```
switch( value )  
{  
    case 0: ...; break;  
    case 1: ...; break;  
    ...  
    default: ...  
}
```

## Boucles : **for**

Comme en C :

```
for( init ; stop_cond ; op )  
{  
    ...  
}
```

## Exemple

```
for( std::size_t i = 0 ; i < 10 ; ++i )  
{  
    ...  
}
```

## *Range-based loops depuis C++11*

```
int tab[ 5 ] = { ... };

// par réf : on modifie le contenu.
for( int & i: tab ) { // ou : auto & i: tab
    i = std::rand() % 10;
}

for( int i: tab ) { // ou auto i: tab
    std::cout << i << std::endl;
}
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Rappel de C

Allocation/désallocation dans le tas :

- Allocation : `malloc`
- Désallocation : `free`

Inconvénient : taille donnée en octets pour l'allocation.

## C++

- Allocation : **new**
- Désallocation : **delete**

### Exemple

```
int * p_i = new int; // allocation d'un int dans le tas.
```

```
*p_i = 4;
```

```
...
```

```
delete p_i;
```

```
int * p_tab = new int[100]; // allocation de 100 int.
```

```
p_tab[ 10 ] = 123;
```

```
...
```

```
delete [] p_tab;
```



## Problèmes

- fuites mémoire : mémoire jamais désallouée.
- erreurs de segmentation : accès ou suppression de pointeurs invalides.

## Erreurs communes

```
// Fuite mémoire.  
int * t0 = new int[ 100 ];  
t0 = new int[ 100 ];
```

```
// Erreur segmentation.  
int * t1; // Valeur indéfinie !  
t1[ 10 ] = 5;
```

## Solutions

- *smart pointers* pour remplacer les pointeurs.
- conteneurs standards pour masquer l'allocation dynamique.

## *smart pointers* en C++11

Les *smart pointers* (pointeurs intelligents) ajoutent un aspect sémantique aux pointeurs et s'occupent de la gestion des pointeurs et de la libération de la mémoire en gardant la syntaxe des pointeurs.

3 types sont disponibles :

- `unique_ptr` : pour un pointeur qui ne doit pas être partagé,
- `shared_ptr` : pour un pointeur partagé,
- `weak_ptr` : pour un pointeur temporaire sur un `shared_ptr`.

## unique\_ptr

Le pointeur ne peut être partagé ou accessible à partir de plusieurs liens.

La propriété est transmise par la fonction `std::move` qui "déplace" le pointeur.

```
unique_ptr< Student > p_s0( new Student( ... ) );
// ou : auto p_s0 = std::make_unique< Student >( ... );
std::cout << *p_s0 << std::endl;
//auto p_s1 = p_s0; // Pas de constructeur de copie.

// p_s1 prend la propriété, p_s0 devient invalide.
unique_ptr< Student > p_s1( std::move( p_s0 ) );
// std::cout << *p_s0 << std::endl; // Erreur segmentation.

p_s1->setFirstName( "Hector" );
```

## shared\_ptr

Le pointeur peut être partagé mais l'instance pointée n'est pas copié.

Un compteur contient le nombre de pointeurs sur l'instance qui est détruite lorsque le compteur est égal à 0.

```

shared_ptr< Student > p_s0( new Student( ... ) );
// ou : auto p_s0 = std::make_shared< Student >( ... );
std::cout << *p_s0 << std::endl;
auto p_s1 = p_s0;
std::cout << p_s1.use_count() << std::endl;
p_s1->setFirstName( "Hector" );
std::cout << *p_s0 << std::endl;
std::cout << *p_s1 << std::endl;
  
```

# Plan

- 1 Bases
- 2 **Généricité**
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

# Plan

- 1 Bases
- 2 **Généricité**
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Templates

La généricité consiste à développer un code ou une structure de données avec des types non explicitement définis i.e. génériques. La généricité peut s'appliquer aux :

- fonctions, par exemple `std :: max`,
- classes, par exemple `std :: vector`.

On peut parler alors de méta-fonctions ou de méta-classes qui vont être instanciées et produire respectivement des fonctions ou des classes.

Ces méta-fonctions et méta-classes peuvent être paramétrées par des types primitifs ou complexes et par des valeurs entières.



## Fonctions génériques

```
template < typename T >  
T fct( T & v )  
{  
    return v + v; // suppose que l'op. + est défini sur le type.  
}
```

## Instantiation

```
int a = 3;  
auto b = fct( a ); // ou fct<int>( a );
```

## Spécialisation

Si un type particulier nécessite un traitement spécifique, il est possible de spécialiser un template.

## Exemple

```
template <T>  
float fct<float>( float & v )  
{  
    return v * 3.0 f;  
}
```

## Classes génériques

```
template < typename T >
class Matrix
{
private:
    std::vector< T > m;
    std::size_t dimx, dimy;
public:
    ...
    T operator()( std::size_t i, std::size_t j ) const
    {
        return m[ j * dimx + i ];
    }
}
```

## Variadiques templates

- Fonctions ou classes avec nombre de paramètres variables.
- Syntaxe : **template** < **typename...** **TS** >
- Utilisation de la récursion dans de nombreux cas avec spécialisation pour arrêt.

## Classes variadiques

cf. `std::tuple`.

## Fonctions variadiques

```
// Fin de la récursion.  
template < typename T >  
void printer( T const & t ) {  
    std::cout << t << std::endl;  
}  
  
// Cas général.  
template < typename T, typename... TS >  
void printer( T const & t, TS const &... ts ) {  
    std::cout << t << ",_";  
    printer( ts... );  
}  
  
// Utilisation.  
printer( 1.0f, 3, 'a', "Hello!" );
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard**
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard**
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Standard Template Library

Concepts :

- Conteneurs
- Itérateurs
- Algorithmes



## Conteneurs

- `std :: array`
- `std :: vector`
- `std :: deque`
- `std :: forward_list`
- `std :: list`
- `std :: set`
- `std :: map`
- ...

## Itérateurs

- Généralisation des pointeurs pour le parcours/manipulation des conteneurs
- Récupération à partir des conteneurs
- Interface pointeur

## Algorithmes

- Manipulation des conteneurs
- Traitement, recherche, dénombrement, tri, ...

## Exemple

```
std::vector< float > v( 100 );  
std::generate_n( v.begin(), v.end(), std::rand );  
std::sort( v.begin(), v.end() );
```

## Foncteurs

- Type complexe + surcharge de l'opérateur ()
- Ressemble à un appel de fonction
- Possibilité de stocker un état interne

## Exemple simple

```
class A {  
    void operator() const {  
        std::cout << "Hello_functor\n";  
    }  
};  
// Utilisation  
A a;  
a();
```

## Exemple : Compteur

```
class counter {  
    std::size_t cpt;  
    counter(): cpt( 0 ) {}  
    auto operator() { return cpt++; }  
};
```

## Exemple : Générateur

```
struct generator {  
    generator() {  
        std::srand (...);  
    }  
    auto operator() const { return std::rand(); }  
};
```

## Lambdas

- Fonctions anonymes.
- Utiles pour passer en paramètre d'un algorithme.
- `[capture list] (params) code;`.

## Exemple

```
std::vector< int > v0( 10 );  
std::vector< int > v1;  
std::generate( begin( v0 ), end( v0 ),  
               []() { return std::rand()%10; } );  
std::copy_if( begin( v0 ), end( v0 ),  
              std::back_inserter( v1 ),  
              []( int i ) { return i%2; } );
```

## Lambdas : Capture de l'environnement

- `[]` : pas de capture.
- `&` : capture des variables externes à la lambda par référence.
- `=` : capture par valeur.
- `[a]` : capture de la variable `a` par valeur.
- `&a` : capture de la variable `a` par référence.

## Exemple

```
std::vector< int > v( 10 );  
int i = 0;  
std::generate( begin( v ), end( v ),  
               [&i]() { return i++; } );
```



# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard**
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Affichage/Saisie console

### Objets globaux :

- `std::cin` : entrée standard.
- `std::cout` : sortie standard.
- `std::cerr` : sortie erreur.

### Exemple

```
int a;  
std::cin >> a;  
std::cout << a << std::endl;
```

## Fichiers

- Lecture/Écriture
  - mode texte : manipulation de caractères, mots, lignes.
  - mode binaire : manipulation d'octets.
- `#include <fstream>`
- `std :: ifstream` , `std :: ofstream`

## Exemple : Mode texte

```
std::ofstream ofs( "output.txt" );  
ofs << "Hello!\n";  
  
std::ifstream ifs( "input.txt" );  
std::string s;  
ifs >> s;  
std::cout << s << std::endl;
```

## Exemple : Mode binaire

```
std::ofstream ofs( "output.bin" );  
int a = 5;  
// Cast de pointeurs : de int* vers char*.  
ofs.write( reinterpret_cast< char * >( &a ),  
          sizeof( int ) );  
  
std::ifstream ifs( "input.bin" );  
ifs.read( reinterpret_cast< char * >( &a ),  
          sizeof( int ) );  
std::cout << a << std::endl;
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard**
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Threads

- `std::thread`
- Constructeur variadique.
- Passage de la fonction à appeler en paramètre + arguments.
- Possibilité de passer une lambda.
- Attention : passage par référence avec `std::ref()`.
- Compilation avec l'option `-pthread`.
- Synchronisation avec `std::mutex`.

## Exemple : thread

```
// Fonctions à appeler.  
void fct0() { ... }  
void fct1( int n ) { ... }  
void fct2( int & n ) { ... }  
  
std::thread th0( fct );  
std::thread th1( fct , 5 );  
int a = 5;  
std::thread th2( fct , std::ref( a ) );  
std::thread th3( []() { std::cout << "Lambda!\n" ; } );  
  
th0.join();  
th1.join();  
th2.join();  
th3.join();
```



## Futures

- Exécution de tâches asynchrones.
- Plus simple que les threads.

### Exemple : tâche asynchrone

```
std::future< int > f0 = std::async( std::launch::async , fact ,  
...  
int res = f0.get(); // Récup. du résultat.
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme**
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme**
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Héritage

**class B : public/protected/private A**

- **public** : les membres publics de A restent publics :
  - accessibles par l'héritier i.e. dans la classe B,
  - accessibles à partir des instances de B.
- **protected** : les membres publics de A deviennent protected :
  - accessibles dans la classe B,
  - inaccessibles à partir des instances de B.
- **private** : Les membres publics et protected deviennent privés :
  - inaccessibles dans la classe B,
  - inaccessibles à partir des instances de B.

## Héritage multiple

- Il est possible d'hériter de plusieurs classes contrairement à Java.
- Risques d'ambiguïtés si plusieurs ancêtres possèdent des attributs ou méthodes de même prototype.

## Exemple

```
struct A {  
    int a;  
};  
struct B {  
    int a;  
};  
struct C : public A, B {  
    void display() {  
        std::cout << a << std::endl; // ???  
        std::cout << A::a << std::endl;  
        std::cout << B::a << std::endl;  
    };  
};
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme**
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards

## Polymorphisme : principe

Définir une hiérarchie de type et pouvoir manipuler ces types à partir du type ancêtre.

### Exemple

```
class Tool { ... };  
class Screwdriver: public Tool { ... };  
class Hammer: public Tool { ... };  
  
Tool * screwdriver = new Screwdriver();
```



## Problème avec l'héritage

```
struct Tool {  
    void display() {  
        std::cout << "This is a tool.\n";  
    }  
};  
  
struct Screwdriver : public Tool {  
    void display() {  
        std::cout << "This is a screwdriver.\n";  
    }  
};  
  
int main() {  
    std::unique_ptr< Tool > screwdriver  
        = std::make_unique< Screwdriver >();  
    screwdriver->display(); // affichage ?  
}
```

## Résultat

This is a tool.

## Explications

- Par défaut le type est déterminé statiquement (à la compilation) et ne dépend pas du type utilisé pour l'instanciation.
- Dans ce cas le type de `screwdriver` est `std::unique_ptr< Tool >`.
- Donc la méthode `display` de `Tool` est appelée.

## Polymorphisme : mots-clés

- **virtual** : définition d'une méthode virtuelle qui peut être surchargée par les héritiers.
- **virtual ... = 0** : méthode virtuelle pure i.e. la classe contenant cette méthode ne peut être instanciée.
- **override** : indique que la méthode doit surcharger la même méthode (même prototype) de l'ancêtre.
- **final** : indique que la méthode ne peut plus être surchargée par les héritiers.

## Exemple basé sur le polymorphisme

```
struct Tool {  
    virtual void display() {  
        std::cout << "This is a tool.\n";  
    }  
};  
  
struct Screwdriver : public Tool {  
    void display() {  
        std::cout << "This is a screwdriver.\n";  
    }  
};  
  
int main() {  
    std::unique_ptr< Tool > screwdriver  
        = std::make_unique< Screwdriver >();  
    screwdriver->display(); // This is a screwdriver.  
}
```

## Mécanisme 1/2

- **virtual** indique que la méthode n'est plus appelée directement.
- Une table virtuelle est créée à l'instanciation pour la classe de base et pour ses héritiers qui contient des pointeurs vers les méthodes virtuelles uniquement.
- Pour la classe de base, ici `Tool`, la table contient un pointeur vers la méthode `Tool::display`.
- Pour la classe héritière, ici `Screwdriver`, la table contient un pointeur vers la méthode `Screwdriver::display`.

## Mécanisme 2/2

- Lors de la compilation le type `std :: unique_ptr < Tool >` est bien attribué à l'instance.
- Lors de l'instanciation la table virtuelle est par contre remplie lors de la construction et contient donc le pointeur vers la méthode `Screwdriver :: display`.
- Lors de l'appel, la méthode `display` est appelée sur le type `Tool` mais c'est le pointeur de la table virtuelle qui est utilisé.

## Conséquences du polymorphisme

- L'appel de méthodes ne dépend plus du type de l'instance.
- Uniquement valable pour les méthodes virtuelles.
- Les méthodes standards ne sont pas affectées.
- Le polymorphisme engendre un surcoût :
  - 1 mémoire : La table virtuelle augmente la taille de l'instance : un pointeur par méthode virtuelle.
  - 2 temps d'exécution : L'appel d'une méthode virtuelle est plus coûteux que pour une méthode standard : indirection, il faut récupérer le pointeur dans la table puis faire l'appel.

???

```
Tool * screwdriver = new Screwdriver;  
...  
delete screwdriver;
```



## Destructeur

- `delete` est appelé sur le type déterminé statiquement, ici `Tool`.
- Donc le destructeur de la classe `Screwdriver` n'est pas appelé : fuite mémoire.
- Solution : rendre le destructeur virtuel.

## Destructeur virtuel

```
struct Tool {  
    ...  
    virtual ~Tool() = 0;  
};  
Tool::~~Tool() {}  
  
struct Screwdriver {  
    ...  
    ~Screwdriver() { ... }  
};
```

## Interface

- Une interface est une classe contenant uniquement des méthodes virtuelles pures.
- Il n'y a pas de mot-clé spécifique comme `interface` en Java.

## Exemple

```
class AnInterface {  
public:  
    virtual method( ... ) = 0;  
};
```

## Classe abstraite

Une classe est abstraite si au moins une de ses méthodes est virtuelle pure.

### Exemple

```
class AnAbstractClass {  
    int value;  
public:  
    AnAbstractClass() { ... }  
    virtual method( ... ) = 0; // au moins une méthode VP.  
};
```

## Exemple : Design pattern Factory

```
struct Tool { // Abstract class.
    std::string name;
    virtual void use() = 0;
    virtual void display() {
        std::cout << "This is a " << name;
    }
};

struct Screwdriver: public Tool {
    std::string type;
    Screwdriver( std::string type ) type( type ) { ... }
    void use() override { ... }
    void display() {
        Tool::display();
        std::cout << " of type " << type;
    }
}
```

## Exemple : Design pattern Factory

```
enum class ToolType { screwdriver , hammer , ... };

struct ToolFactory {
    template < typename T >
    static std::unique_ptr< Tool > create(
        ToolType tooltype ,
        T &... args )
    {
        switch( tooltype ) {
            case ToolType::screwdriver :
                return std::make_unique<Screwdriver>( args );
                break;
            case ...
        }
    }
};
```

## Exemple : Design pattern Factory

```
using ToolBox = std::vector< std::unique_ptr< Tool > >;
ToolBox toolbox;
toolbox.push_back( ToolFactory::create(
                                hammer,
                                ... ) );
toolbox.push_back( ToolFactory::create(
                                screwdriver,
                                "Phillips" ) );

for( auto & tool: tool_box ) {
    tool->use();
    tool->display();
}
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation**
- 6 Boost
- 7 Futurs standards



## Métaprogrammation

- Idée : calculer/transformer le code à la compilation pour améliorer les performances à l'exécution du programme.
- Effet de bord du mécanisme des templates.
- Techniques : utilisation des templates et de `constexpr`.

## Exemple : Fonction factorielle

```
template < unsigned int N >
struct fact {
    static unsigned int const value
        = N * fact< N - 1 >::value;
};

template <
struct fact< 0 > {
    static unsigned int const value = 1;
};

std::cout << fact< 10 >::value << std::endl;
```

## Expression templates

- Objectifs : suppression des temporaires et des recopies.
- Évaluer le code le plus tard possible.

## Exemple : addition de vecteurs

```
class vec3f {  
    vec3f & operator=( vec3f const & v ) { ... }  
    vec3f operator+( vec3f const & v ) { ... }  
};  
...  
vec3f v0 = v1 + v2 + v3 + v4;
```

- 1 addition de v3 et v4 (for) dans un `vec3f` temporaire,
- 2 addition de v2 et du temporaire (for) dans un nouveau temporaire,
- 3 addition de v1 et du nouveau temporaire (for) dans un autre temporaire,
- 4 recopie du dernier temporaire dans v0 (for).

Total : 3 `vec3f` temporaire + 4 boucles for.

## Version sans classe (C)

```
for( ... ) {  
    v0[ i ] = v1[ i ] + v2[ i ] + v3[ i ] + v4[ i ];  
}
```

Total : pas de vec3f temporaire, 1 boucle for.

## Expression templates

- Allier la programmation haut-niveau/objet avec l'efficacité des langages bas-niveau.
- Objectif : suppression des temporaires, des parcours inutiles.
- Idée : surcharger les opérateurs et retarder l'évaluation.

## Exemple

```
class vec3f {  
    ...  
    vec3f operator+( vec3f const & v ) {  
        vec3f tmp( size() );  
        for( std::size_t i = 0 ; i < size() ; ++i ) {  
            tmp[ i ] = *this[ i ] + v[ i ];  
        }  
        return tmp;  
    }  
    ...  
};
```

## Exemple

- 1) Une expression générique contient 2 opérandes et 1 opérateur. Dans notre exemple, une expression peut contenir 2 `vec3f`, ou être composée d'autres expressions.
- 2) L'opérateur `[]` applique l'opérateur `Op` sur les 2 opérandes. Si les opérandes sont des expressions, l'application s'applique récursivement.

```
template < typename L, typename Op, typename R >
struct expr {
    L const & l;
    R const & r;
    Expr( L const & l, R const & r ): l(l), r(r) {}
    auto operator [] ( std::size_t i ) {
        return Op::apply( l[ i ], r[ i ] );
    }
};
```



## Exemple

3) Surcharge de l'opérateur `+` pour les expressions : plus de calcul, construction d'une expression.

```
template < typename L, typename R >  
expr< L, Plus, R > operator+( L const & l,  
                               R const & r ) {  
    return expr< L, Plus, R >( l, r );  
}
```

## Exemple

4) Définition de l'opération à effectuer.

```
struct Plus {  
    static unsigned int apply( unsigned int a,  
                               unsigned int b ) {  
        return a + b;  
    }  
};
```

## Exemple

Composition des expressions :

```
vec3f v0 , v1 ;  
... v0 + v1 ; // expr< vec3f, Plus, vec3f >  
... v0 + v1 + v2 ; // expr< vec3f, Plus, expr< vec3f, Plus, vec3f > >  
...
```

## Exemple

5) Évaluation de l'expression par les opérateurs `[]` ou `=`.

```
template < typename L, typename Op, typename R >
struct expr {
    ...
    auto operator[]( std::size_t i ) {
        return Op::apply( l[ i ], r[ i ] );
    }
};
```

## Exemple

```
class vec3f {  
    ...  
    template < typename Expr >  
    array & operator=( Expr const & e )  
    {  
        v.resize( e.size() );  
        for( std::size_t i = 0 ; i < v.size() ; ++i )  
        {  
            v[ i ] = e[ i ];  
        }  
        return *this;  
    }  
    ...  
};
```

# Démo

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost**
- 7 Futurs standards

## Boost

- bibliothèques C++ open-source libre (licence Boost)
- contient beaucoup de fonctionnalités non intégrées au C++
- "bac à sable" avant éventuel introduction dans le standard
- Contenu :
  - Asio : réseau
  - Algorithm
  - DateTime
  - Filesystem
  - GIL : Generic Image Library
  - Interprocess
  - Log
  - Serialization
  - ...



## boost::property\_tree

Lecture d'un fichier json :

```
{  
    "login": "jubertie",  
    "server_name": "192.168.0.254",  
    "array":  
    [  
        "1", "2", "7"  
    ]  
}
```

## boost::property\_tree

```
boost::property_tree::ptree p;  
  
boost::property_tree::read_json( "config.json", p );  
  
auto login = p.get( "login", "none" );  
auto server_name = p.get( "server_name", "localhost" );  
auto missing = p.get( "missing", "missing" );  
  
std::cout << "login=" << login << std::endl;  
std::cout << "server_name=" << server_name << std::endl;  
std::cout << "missing=" << missing << std::endl;  
  
for( auto x: p.get_child( "array" ) ) {  
    std::cout << x.second.get_value<int>() << std::endl;  
}
```

# Plan

- 1 Bases
- 2 Généricité
- 3 Bibliothèque standard
- 4 Héritage/Polymorphisme
- 5 Métaprogrammation
- 6 Boost
- 7 Futurs standards**

## C++17 : langage

- Mot clé **typename** comme alternative à **class** pour les paramètres **template template**
- Nouvelles règles de déduction pour **auto**
- Imbrication simplifiée des **namespaces** : **namespace X::Y { ... }** au lieu de **namespace X { namespace Y { ... } }**
- Caractères UTF-8
- **if** statique à la compilation : **if constexpr(expr)**
- **if** avec initialisation : **if( init ; expr )**
- Déduction de type pour les constructeurs :  
**std::pair( 1.0f, 'a' )** au lieu de **std::pair<float, char>( 1.0f, 'a' )**
- ...

## C++17 : bibliothèque

- Suppression de types et fonctions dépréciées : `std::auto_ptr`, `std::random_shuffle`, ...
- Ajout de la bibliothèque `filesystem`
- Versions parallèles des algorithmes
- `std::byte` : type octet
- `std::string_view`
- `std::optional`
- `std::any`
- ...

## C++2x

- Modules : remplacement du mécanisme `#include` par `import` : amélioration du temps de compilation, plus de `#ifdef ... _HPP` `#define` pour les fichiers `.hpp`.
- Concepts
- Ranges
- Coroutines
- ...