Modern cpp

Les évolution du langage dans les grandes lignes...
On fera la lib std dans une autre formation

Auto

```cpp
int & return_reference();
auto return_int() { return 1; }

int main()
{
    auto i = 1; // work as int i;
    auto copy = return_reference();
    // int copy
    auto& ref = return_reference();
    //int& ref
    const auto& ref = return_reference();
    std::vector<int> vec;
    auto it = vec.begin();
    auto fn_ptr = &std::vector<int>::cbegin;
}
```

- Déduction automatique de type
- Nécessite d'être assigné à l'initialization
- Fait une copie par défaut
- N'est pas const par défaut
- https://en.cppreference.com/w/cpp/language/auto

Default/Delete constructor & operator=

```
struct Toto
{
    int value;
    Toto(int val): value(val) {}

    Toto() = default;
    virtual ~Toto() = default;
    Toto(const Toto&) = delete;
    Toto& operator=(const Toto&) = delete;
    Toto(Toto&&) = default;
    Toto& operator=(Toto&&) = default;
};
```

- Ajouter un constructor a une classe désactive la génération automatique des constructor et des operator= d'une classe
- Il est possible maintenant de explicitement demander au compilateur de les générer.

Final & override

```
struct Toto
{
    int value;
    virtual int function() const;
    virtual int cannot_be_overidden() const
final;
};

struct Child : Toto
{
    int function() const override;
    //int function() override; error!
    //int cannot_be_overidden() const; error!
};
```

- Permet d'ajouter un contrôle du compilateur pour vérifier qu'il y a pas de situation à risque sur les fonction virtuelle.

Move operation (1/4)

- Permet de déplacer des ressources pour éviter des duplicatas coûteux ou impossible (mémoire, lock sur mutex, socket, etc..).
- Ouvre la possibilité au conteneur générique de manipuler des types qui ne peuvent pas être copié.
- Est implicitement fait au retour d'une fonction
- Est explicitement fait avec std::move

Move operation (2/4)

- std::string retourné par `create_title`n'est pas copié, sa mémoire qu'il gère est déplacé dans la variable `title` qui va la prendre en charge
- std::unique_lock ne peux pas être copié, mais peut être déplacé explicitement dans le scope d'une nouvelle fonction
- **Une fois le move effectué, la variable initiale ne contient plus rien, ici le `std::unique_lock l` ne lock plus rien.**

```
bool need_work();
std::string create_title();
void do_work(const std::string& title,
             std::unique_lock<std::mutex> lock);

int main()
{
    std::mutex m;

    while (need_work())
    {
        auto title = create_title();
        std::unique_lock l(m);
        do_work(title, std::move(l));
        //do_work(title, l); // error !
        //do_work(title, std::unique_lock(m));
        // /!\ l now empty /!\
    }
}
```

Move : rvalue reference (3/4)

- Utilisé pour dénoter les variables qui sont temporaires.
- Utilise la notation **T&&**
- **A ne pas utiliser en type d'une variable dans le corps d'une fonction** (sauf si vous savez vraiment ce que vous faites).
- Exemple des rvalue les plus communes: les valeurs en dures dans votre code, les retours de fonction,

```
void do_work(std::string& value) //(1)
{
    value = "toto";
}

void do_work(std::string&& value) //(2)
{
    do_work(value);
}

void crappy_code()
{
    std::string s = "titi";
    do_work(s); //(1)
    do_work("tutu"); //(2)
}
```

Move constructor (4/4)

- 95% des cas d'utilisation des rvalue references pour le commun des mortels.
- Un exemple d'une implémentation canonique.
- Toto n'est pas copiable car il contient un std::thread, par contre il est movable.
- Il peut être renvoyé d'une fonction ou inséré dans un conteneur de la std.

```
struct Toto
{
    std::string str;
    std::thread thr;
    int* i = new int(1);

    Toto(Toto&& orig) {
        operator=(std::move(orig));
    }

    Toto& operator=(Toto&& orig) {
        if (this != &orig) {
            str = std::move(orig.str);
            thr = std::move(orig.thr);
            delete i; i = orig.i; orig.i = nullptr;
        }
        return *this;
    }
};
std::vector<Toto> vec;
Toto make_toto();
```

Brace initialization (1/3)

- List initialization: A utiliser pour rendre l'init des container plus facile et analogue au C.

```
std::vector<int> v1 { 1, 2, 3, 4 };
static std::vector<int> v2 = { 1, 2, 3, 4 };
std::initializer_list<int> v3 = { 1, 2, 3, 4 };
```

- std::initalizer_list< T > est créé automatiquement.
- Il n'y aucune garantie de la durée de vie des objets, autre qu'il doivent être valident localement.
- Le compilateur tente de déduire le T a partir du types des objet contenu entre les { ... }, il a parfois besoin d'aide:

```
struct Object { Object(int) {} };
std::initializer_list<Object> deduced = { 1, 2 };
auto deduced = { Object(1), Object(2) };
```

Brace initialization (2/3)

- Construction des objets uniformisé avec le { ... }.

```
struct Object_with_long_name { Object_with_long_name(int) {} };
Object_with_long_name b{1}; // call Object(int)
```

- { ... } Peuvent être utilisé seule dans un context où le type peut être déduit:

```
Object_with_long_name function_big_question_answer() {
   return {42}; // long name deduced
}
auto deduced = { Object(1), {2} };
```

Brace initialization (3/3)

- (C++14/20) Initialization directe des paramètre pour les agrégats:

```
struct Object {
    int i;
    std::string str;
 };
Object b{1, "toto"}; // C++14 b.i=1, b.str="toto"
Object b{.i = 1, .str = "toto"}; // C++20
```

- **/!\ Priorité au constructor qui prend la std::initializer_list:**

```
std::vector<int> v1{1, 2}; // create { 1, 2 };
//^ vector( std::initializer_list<int> init );
std::vector<int> v2(1, 2); // create { 2 };
//^ vector( size_type count, const T& value = T(), ...);
```

Delegating/Inherited Constructor

```
struct A {
    A(int i) {};
    A(std::string v) : A(std::stoi(v)) {} // Call A::A(int)
};

struct B : public A {
    using A::A; // inherit A constructor
};

B b1{1}; // ok
B b2{"toto"}; // ok
```

Lambda

- Voir la formation sur les lambda
- https://github.com/Surrog/Formations/blob/master/FR/Cpp%20Lambda.pdf

Range based for

- Itération des conteneurs simplifié et sans macro:

```
std::vector<int> i{1, 2, 3, 4};
for (auto& value : i)
{
    // do something
}
```

- Fonctionne sur tous les types qui implémente begin() et end(), même ceux hors std: le pattern ne fait que cacher la manipulation des itérateurs.
- Offre au compilateur la possibilité d'optimisation du code (vectorisation, etc).

Thread_local storage

- Permet de créer des variables qui ont la durée des vie du thread qui l'a créé.

```
std::vector<int>& only_one_instance_per_thread()
{
    static thread_local std::vector<int> v = {1, 2 , 3};
    // ^ created locally (only once) when a thread execute for the first time
    return v;
}
```

Raw string literal

- Permet d'empêcher le compilateur d'interpréter le contenu d'une chaîne de caractère entre quote
- Format: **R"**delimiter**(** raw_characters **)**delimiter**"**

```
std::string interpreted = "path\to\name";
std::string not_interpreted = R"_(path\to\name)_";
// R" delimiter = `_(` and `)_`
std::cout << interpreted << '\n';
// print: "path o
//         ame"
std::cout << not_interpreted << '\n';
// print "path\to\name"
```

Class template argument deduction

- C++17
- Le compilateur peut maintenant déduire les type complet d'une classe template avec son constructeur

```
std::mutex mtx;

std::pair p{1, 2.3}; //std::pair<int, double>
std::vector v{1, 2, 3}; //std::vector<int>
std::unique_lock l{mtx}; //std::unique_lock<std::mutex>
```

Nodiscard attribute

- C++17
- Pouvez préfixer une fonction ou un constructor du mot clé [[nodiscard]][pour encourager l'utilisateur à en récupérer le retour.

```
[[nodiscard]]
error_code do_something() {
    return 42;
}

int main()
{
    do_something(); //warning: ignoring return value of function declared with
'nodiscard' attribute
}
```

Structured binding declaration

```
int a[2] = {1,2}; // work with array
auto [x,y] = a; // creates e[2], copies a into e, then x refers to
e[0], y refers to e[1]
auto& [xr, yr] = a; // xr refers to a[0], yr refers to a[1]

std::map<int, std::string> map{...};
//iterate as a pairs of (key, value)
for (const auto& [key, value] : map)
{ std::cout << key << ' ' << value << '\n'; }

struct S {
    int x1;
    double y1;
};
S instance;
auto& [x, y] = instance; //ref to x1 & y1 from instance
```

- C++17
- Permet de décomposer facilement des tableaux, des tuples ou des agrégats.

Initializer in control block

```
int get_value();
std::vector<std::string> foo();
void bar(const std::string& value, int index);

//C++98
for (int i = 0; i < 10; i++) { /*...*/}
//C++17
if (int error = get_value(); error == 0) { /*...*/}
switch (int error = get_value(); error) { /*...*/ }
//C++20
for (std::size_t i = 0; const auto& x : foo()) {
     bar(x, i);
     ++i;
}
```

- C++17 & C++20
- Harmonise les differents control block

Nested namespace definitions

- C++17
- Réduit la verbosité

```
//Before
namespace A {
    namespace B {
        class C {};
    }
}
//After C++17
namespace A::B {
    class V {};
}
```

Abbreviated function template

- C++20
- Rendre les template plus compact et lisible

```
//template <typename T, typename Y>
void function_template( auto value, auto test)
{
    std::cout << value << test << '\n';
}

int main()
{
    function_template(1, 2); // "12"
    function_template<int, float>(1, 2);
}
```

Three-way comparison (1/2)

- C++ 20
- Operator lhs **<=>** rhs
- (a <=> b) < 0 si lhs < rhs
- (a <=> b) > 0 si lhs > rhs
- (a <=> b) == 0 si lhs et rhs sont égaux/équivalents.
- Va vous renvoyer une erreur si lhs et rhs ne sont du pas du même type ou si il y a une `narrowing` conversion.
- Permet de remplacer les opérateurs de comparaison habituel
- Peut être généré automatiquement pour des classe :

```
class A
{
    auto operator<=>(const A&)
const = default;
};
```

Three-way comparison (2/2)

```cpp
struct Test {
    int i = 42;
    std::string str = "toto";
    auto operator<=>(const Test&) const = default;
};

Test a;
Test b;
if (a != b) {
    std::map<Test, int> map = {{{}, 1}};
}
```

Chose trop grosse pour être aborder ici:

- attribute
- noexcept
- constexpr | consteval | constinit | if constexpr
- Variadic template
- Coroutines
- Modules
- Constraints and concepts