Modern cpp lib

2ème partie de modern cpp : les évolutions de la lib standard

C++11: unique_ptr, shared_ptr (1/2)

- Simplifie la gestion de la mémoire
- Unique_ptr n'est pas copiable mais est movable
- La copie d'un shared_ptr est thread-safe mais pas l'accès a std unique_ptr
- Il est possible de passer un `deleter` une fonction custom qui s'exécute pour delete le pointer
- Il y a une surcharge pour les tableau C
- Il y a une fonction pour dynamic_cast

```
#include <memory>

std::unique_ptr ptr = std::make_unique<int>(1);
// ^ safe pointer to a integer initialized with 1
std::shared_ptr sptr = std::make_shared<double>(3.2);
// ^ safe pointer to a double initialized with 3.2
std::unique_ptr array = std::make_unique<int[]>(10);
// ^ safe pointer to an array of 10 uninitialized integer
struct A : public B {};
std::shared_ptr<A> aptr = std::make_shared<B>(3.2);
std::shared_ptr<B> bptr =
std::dynamic_pointer_cast<B>(aptr);
std::unique_ptr ptr_deleter =
std::unique_ptr<int>(new int(), [](int* v)
{ std::cout << *v << '\n'; delete v;});
```

C++11: unique_ptr, shared_ptr (2/2)

```cpp
#include <memory>

std::unique_ptr ptr = std::make_unique<int>(1);
// ^ safe pointer to a integer initialized with 1
std::shared_ptr sptr = std::make_shared<double>(3.2);
// ^ safe pointer to a double initialized with 3.2
std::unique_ptr array = std::make_unique<int[]>(10);
// ^ safe pointer to an array of 10 uninitialized integer
struct B : public A {};
std::shared_ptr<A> aptr = std::make_shared<B>();
std::shared_ptr bptr = std::dynamic_pointer_cast<B>(aptr);
std::unique_ptr ptr_deleter = std::unique_ptr<int, std::function<void (int *)>>(
      new int(10), [](int* v)
      { std::cout << *v << '\n'; delete v;}
);
```

C++11: tuple

- #include <tuple>
- Contient 1 a X éléments
- S'utilise ensuite le plus souvent avec std::tie, std::get ou un structured binding

```
std::tuple<std::string, std::string, error_code>
failable_function()
{
    if (isOk()) {
        return std::make_tuple("toto", "titi", 0);
    }
    return std::make_tuple("", "", 1);
}

std::string husband, wife;
error_code err;

std::tie(husband, wife, err) = failable_function();
// husband="toto", wife="titi", err=0
auto [first, second, e] = failable_function();
auto tuple = failable_function();
std::string& ref_to_first = std::get<0>(tuple);
```

C++11: forward list

- #include <forward_list>
- std::list mais avec un iterator qui ne peut qu'avancer
- Peu d'intérêt en soit sauf cas très particulier ce contrainte mémoire

C++11 unordered_map, unordered_set

- Conteneur associatif qui utilise std::hash pour stocker les élément dans un tableau
- Complexité de l'accès : O(1) par rapport a O(log n) d'un std::map
- Par rapport a std::map, les élement ne sont pas trié quand vous itérez dessus.

```
std::unordered_map<std::string, std::string> u = {
    {"RED","#FF0000"},
    {"GREEN","#00FF00"},
    {"BLUE","#0000FF"}
};

// Iterate and print keys and values of unordered_map
for( const auto& n : u ) {
    std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
}
```

C++11: array

- #include <array>
- Permet de remplacer les tableaux C
- Encode la taille directement dans le type, pas de conversion vers T*

```
std::array<double, 10> func();

std::array<int, 3> a2 = {5, 2, 3};
std::array<std::string, 2> a3 =
    { std::string("a"), "b" };

// container operations are supported
std::sort(a2.begin(), a2.end()); // 2, 3, 5
auto result = func(); //ok
```

C++11: random (1/2)

- Offre des outils pour générer des nombres aléatoire correctement et potentiellement plus rapidement qu'avec les API C.
- Les outils s'articule en 3 morceaux:
- Le random_device: La source d'aléatoire généré par le hardware et l'os, maj à leur bon vouloir.
- Le generator/engine: Un outils qui contient un ensemble de formule mathématique qui ont des comportement chaotique.
- La distribution: Une fonction qui permet de redistribuer un nombre dans un ensemble statistique. (ex, distribution linéaire entre 0 et 10 fait que chaque chiffre a autant de chance de sortir qu'un autre)

C++11: random (2/2)

```
std::random_device rd; // entropy generated once at creation
std::mt19937 engine; // basic engine twister
engine.seed(rd()); // initialize the engine so that no one knows the starting point

std::uniform_int_distribution<std::mt19937::result_type> distribution(0, 10);
auto random_value = distribution(engine);
// ^ a value between 0 and 10 included
std::bernoulli_distribution d(0.25);
auto bell_value = d(engine);
// give "true" 1/4 of the time
// give "false" 3/4 of the time
```

C++11/C++20 Chrono (1/3)

- Outils pour manipuler le temps standardisé en c++
- Permet de gérer des durées et des point dans le temps.
- Offre différente horloge, chacune ayant leurs précisions et leurs comportement:
  - system_clock est l'horloge système tel que vous la voyez afficher sur le bureau, avec ses changement de tz, d'heure d'hiver & manipulation de l'utilisateur.
  - steady_clock est l'horloge hardware, qui reste maj par le hardware tout le temps et qui ne peux pas revenir en arrière.
  - high_resolution_clock est l'horloge qui offre la mesure la plus précise du temps, mais n'est pas forcément steady (peut être testé avec un getter)
- C++20 ajoute la gestion des time zones

C++11 Chrono (2/3)

```
using namespace std::chrono;
time_point p1 = steady_clock::now();
time_point p20 = p1 + seconds(20);
std::this_thread::sleep_for(1s); //here implicit creation of 1 seconds duration
duration d = steady_clock::now() - p1;
static_assert(std::is_same_v<steady_clock::duration, nanoseconds>);
std::cout << "time elapse: " << d.count() << " nanoseconds\n"; // 1003432580 nanoseconds
std::cout << "time elapse: " << duration_cast<seconds>(d).count() << " seconds\n"; // 1 seconds
std::cout << "time elapse: " << duration_cast<duration<double>>(d).count() << " seconds\n"; // 1.00343
seconds
std::cout << "time elapse: " << duration_cast<duration<double, milliseconds::period>>(d).count() << "
milliseconds\n"; // 1000.07 milliseconds
```

C++11 Chrono (3/3)

- Les clock ont leurs propre type de duration qu'elle renvois, vous pouvez la tester en regardant leurs typedef `::duration`
- Vous pouvez facilement créer des durations avec les prefix ""s, ""ms, ""h, etc...
- Les durées sont de base stocké avec un integer et une unité:
  - Permet d'exprimer des durées très petite ou très grandes
  - L'unité est dans le type de la duration: std::chrono::duration<long long, std::milli> pour ms. Par defaut l'unité est la seconde.
  - Quand vous cast vers une unité moins précise, opté pour un stockage en double ou risqué un arrondi violent.
- Vous pouvez convertir d'une unité à une autre avec std::chrono::duration_cast
- Les time point sont stocké comme des duration depuis epoch.

C++11 Thread (1/4)

- Offre les outils de base pour lancer des thread et des taches asynchrone en c++.
- Pas encore de pool ou de task queue, c'est prévu pour le C++23 avec l'inclusion de asio.
- 3 catégorie majeures :
  - std::thread & std::jthread, qui sont les primitives qui permettent de lancer un thread
  - std::mutex, std::atomic & std::lock_guards qui sont les outils pour faire de la synchronisation autour de données
  - std::future & std::async qui permettent de gérer les tâches asynchrone dans votre application.
- Utilitaire: `std::this_thread` qui permet d'accèder au thread executant votre code.

C++11 Thread (2/4)

- std::thread, std::jthread et std::mutex fonctionne telle que vous pouvez les imaginer.
- Vous devez soit `join()` soit `detach()` avant que instance de std::thread soit détruite pour éviter de crash
- Vous devez link avec pthread sous linux

```
{ std::mutex mtx;
int val; double f;

std::thread thd1 { [&]() {
    std::unique_lock l{mtx};
    val++;
    f++;
} };

std::thread thd2 { [&]() {
    std::unique_lock l{mtx};
    val++;
    f++;
} };
thd1.join();
thd2.join();
}
```

C++11 Thread (3/4)

- std::atomic permet de simplifier la synchronisation autour de simple primitive
- Ne permet pas de garantir une série d'instruction, juste la lecture et l'écriture
- Offre la possibilité de préciser en détails les contraintes pour le lire ou écrire une valeur
- **Utile pour les variable qui sont intensément utilisé par des thread**

```
std::atomic<int> value = 0;

std::thread printer { [&] ()
    {
        std::ofstream l("log");
        while (true) {
            l << value.load(std::memory_order_relaxed) <<
'\n';
            using namespace std::chrono;
            std::this_thread::sleep_for(1s);
        }
    }
};

printer.detach();

value = 10;
value += 39;
value = value * 10;
```

C++11 Thread (4/4)

- Future, promise & async permettent de distribuer facilement des taches a des threads.
- Le pool de thread est créé automatiquement.
- L'async est garantie d'être executé a un moment.
- Promise et future fonctionnent en pair pour transmettre le retour de la fonction exécutée par le thread

```
std::future<int> execute_async() {
    std::promise<int> pro;
    std::future<int> future = pro.get_future();
    std::thread thd(
        [promise = std::move(pro)]() mutable {
            // do stuff
            promise.set_value(42);
        }
    );
    thd.detach();
    return future;
}
auto future = execute_async();
auto future2 = std::async( []() {
    return 42;
});
// do stuff
std::cout << "execute_async: " << future.get() <<
'\n';
std::cout << "execute_async: " << future.get() <<
'\n';
```

C++17 Any & Variant

- std::any, contient tout les type possible, s'utilise avec std::any_cast<T>
- std::variant<T, Y, ...>, contient une variable qui a soit le type T, Y, ou etc

```
auto a = std::any(12);
std::cout << std::any_cast<int>(a) << '\n';
try {
    std::cout
     << std::any_cast<std::string>(a)
     << '\n';
}
catch(const std::bad_any_cast& e) {
    std::cout << e.what() << '\n';
}
auto var = std::variant<int, double>(2.1);
auto d = std::get<double>(var); //d = 2.1
auto err = std::get<int>(var); //throw error
```

C++17 string_view

- Permet de wrap une chaine de caractere.
- N'offre pas de garantie qu'elle finit par \0
- Pas d'allocation
- La string contenu n'est pas modifiable

```
//before
int work_on_string(const char*); //C string
int work_on_string(const std::string&); //C++
string
//after
int work_on_string(std::string_view str) // work
on both
{
    for (char c : str)
    {
        //do stuff
    }
}

Std::string str = "toto";
work_on_string( std::string_view(str.begin(),
str.begin() + 2 ) ); // view on only 2 char
```

C++17 filesystem

- Standardisation des accès au path
- Plein de fonction très utile comme:
- copy_file(path from, path to)
- create_directories(directories path)
- bool exists(path)
- path current_path()

```
using namespace std::filesystem;
create_directories("sandbox/a/b");
std::ofstream("sandbox/file1.txt");
std::ofstream("sandbox/file2.txt");
for(auto& p: directory_iterator("sandbox"))
    std::cout << p.path() << '\n';
    /* "sandbox/a"
        "sandbox/file1.txt"
        "sandbox/file2.txt" */
remove_all("sandbox");
```

C++17 optional

- Permet la généralisation des pattern ou une fonction renvois une valeure ou une erreur
- Possède un conversion vers bool et une fonction has_value pour tester si elle contient une valeur

```
std::optional<std::string> create(bool b) {
    if (b)
        return "Godzilla";
    return {};
}

// create(false).has_value() == false
// create(true).get() == "Godzilla"

auto create_ref(bool b) {
    static std::string value = "Godzilla";
    return b ?
std::optional<std::reference_wrapper<std::string>>{value}
            : std::nullopt;
}
```

C++20 format

- Remplace printf et std::cout
- Est beaucoup plus rapide que ses remplaçant
- Similaire au Qt format

```
std::format("{} {}!", "Hello", "world",
"something");
// OK, produces "Hello world!"

std::format("{1} {0}!", "Hello", "world",
"something");
// produces "world Hello!"
```

C++20 span

- std::string_view mais pour tout les type de tableau
- Peux mapper un tableau fixe en memoire ou réallouable

Trop grosse pour être dans cette formation

- std range
- std concept
- std error_code