

Programmation C++ Avancée

Session 6 – La Bibliothèque standard

Joel Falcou Guillaume Melquiond

Laboratoire de Recherche en Informatique

La sainte trinité du standard

Conteneurs

- Encapsulation des TDA classiques
- Paramétrables au niveau type et mémoire
- Parcourables via des ...

La sainte trinité du standard

Conteneurs

- Encapsulation des TDA classiques
- Paramétrables au niveau type et mémoire
- Parcourables via des ...

Itérateurs

- Abstraction du pointeur
- Utilisables dans des ...

La sainte trinité du standard

Conteneurs

- Encapsulation des TDA classiques
- Paramétrables au niveau type et mémoire
- Parcourables via des ...

Itérateurs

- Abstraction du pointeur
- Utilisables dans des ...

Algorithmes

- Parcours décorrélié du Conteneur
- Garantie de complexité et de correction
- Paramétrables via des fonctions utilisateur

Conteneurs standards

Conteneurs Séquentiels

- `vector`, `array`
- `list`, `forward_list`
- `deque`

Conteneurs Associatifs

- `set`, `map`
- `multi_set`, `multi_map`
- `unordered_set`, `unordered_map`
- `unordered_multi_set`, `unordered_multi_map`

Algorithms standards

- `all_of`, `any_of`, `none_of`
- `for_each`
- `count`, `count_if`
- `mismatch`, `equal`
- `find`, `find_if`
- `find_end`, `find_first_of`
- `search`, `search_n`
- `nth_element`
- `max_element`, `min_element`
- `transform`
- `copy`, `copy_if`
- `remove`, `remove_if`
- `replace`, `replace_if`
- `reverse`, `rotate`, `shuffle`
- `sort`, `stable_sort`
- `fill`, `iota`
- `accumulate`
- `inner_product`

Fonction anonyme

Objectifs

- Augmenter la localité du code
- Simplifier le design de fonctions utilitaires
- Notion de *closure* et de fonctions d'ordre supérieur

Principes

- Bloc de code fonctionnel sans identité
- Syntaxe :

```
auto f = [ capture ] (parametres ... ) -> retour  
{  
    corps de fonction;  
};
```

Fonction anonyme

Type de retour

- C++11 : automatique si la fonction n'est qu'un return
- C++11 : à spécifier via `->` sinon
- C++14 : déduction automatique

Fonction anonyme

Paramètres

- C++11 : types concrets, pas de variadiques

```
auto somme = [](int a, int b) { return a + b };
```

- C++14 : types génériques et variadiques

```
#include <tuple>
auto somme = [](auto a, auto b) { return a + b; };
auto as_tuple = [](auto... args)
    { return std::make_tuple(args...); };
```

Fonction anonyme

Capture de l'environnement

- [] : environnement vide
- [a] : capture a par copie
- [&a] : capture a par référence
- [=] : tout par copie
- [&] : tout par référence

```
int x, n;
```

```
auto f = [x](int a, int b) { return a*x+b; };  
auto g = [&x]() -> void { x++; };  
auto h = [&x,n]() -> void { x *= n; };  
auto s = [&]() { x = n; n = 0; return x; };
```

Algorithmes en action

```
bool match_pattern(Buffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<Buffer> const& mems)
{
    std::vector<Buffer>::const_iterator i = mems.cbegin();

    for ( ; i != mems.cend(); ++i)
    {
        if (match_pattern(*i))
            return true;
    }

    return false;
}
```

Algorithmes en action

```
bool match_pattern(Buffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<Buffer> const& mems)
{
    for (auto i = mems.cbegin(); i != mems.cend(); ++i)
    {
        if (match_pattern(*i))
            return true;
    }

    return false;
}
```

Algorithmes en action

```
bool match_pattern(Buffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<Buffer> const& mems)
{
    for (auto const& mem : mems)
    {
        if (match_pattern(mem))
            return true;
    }

    return false;
}
```

Algorithmes en action

```
bool match_pattern(Buffer const& mem)
{
    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<Buffer> const& mems)
{
    return std::find_if(std::cbegin(mems), std::cend(mems),
                        match_pattern)
           != std::cend(mems);
}
```

Algorithmes en action

```
bool process_buffer(std::vector<Buffer> const& mems)
{
    return std::find_if(std::cbegin(mems), std::cend(mems)
        , [](Buffer const& mem)
        { return mem.size() > 2
            && mem[0] == 'E'
            && mem[1] == 'Z'; }
        ) != std::cend(mems);
}
```