ArtWorks

Advanced C++

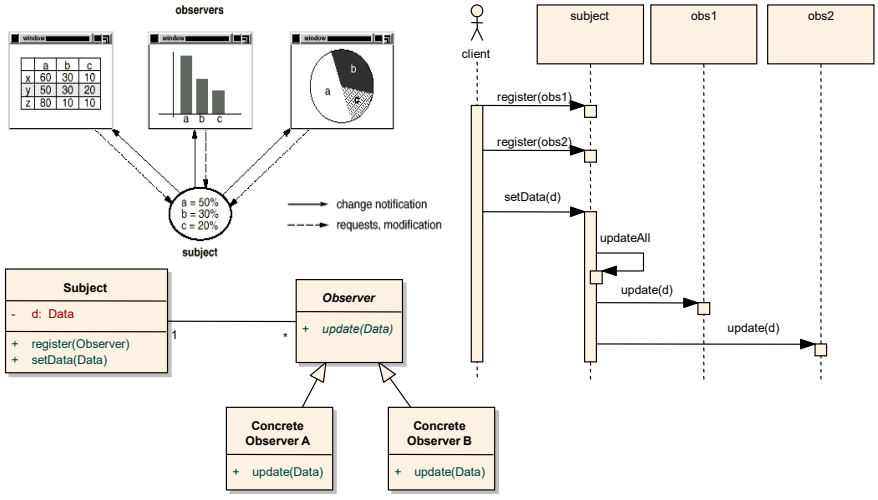
Programmation fonctionnelle

Microsoft Certified Professional
Solution Developer
Trainer

CompTIA. Certified Technical Trainer+

Michel André
michel@artworks.fr

Abonnement – Publish / Subscribe



2

Pointeur de function / méthode

```
using Func = int(*)(int);
int square(int n){ return n*n; }

struct Calculateur {
  int square(int n) const { return n*n; }
};
using Method = int(Calculateur::*)(int) const;
```

```
Func f {&square};
cout << f(3) << endl;

Method m {&Calculateur::square};
Calculateur c;
cout << (c.*m)(3) << endl;
```


3

Usage de pointeurs de méthodes

```
struct A {
  void simple(int n){cout << n << endl;}
  void carre(int n){cout << n*n << endl;}
};

int main(){
  A a;

  for (auto method : {&A::carre,&A::simple})
    (a.*method)(3);
}
```

```
9
3
9
3
```


4

La classe fonction

```
#include <functional>

int add (int a, int b){return a + b;}
function <int (int, int)> f {add};
cout << f (23, 78) << endl;
```

```
using Action = function<void()>;

void execute(const initializer_list< Action >& actions) {
  for (const auto& action : actions)
    action();
}

void func() { cout << "plain function\n"; }

struct functor {
  void operator()() const { cout << "functor\n"; }
};
```

```
execute({ func, functor{}});
```

```
plain function
functor
```


9

Binding et placeholders

```
#include <functional>
#include <iostream>

using namespace std;
using namespace std::placeholders;

int multiply(int a, int b){return a * b;}

int main(){
  function<int(int)> f {bind(multiply, 5, _1)};
  for (int i {}; i < 10; i++)
    cout << "5 * " << i << " = " << f(i) << endl;
}
```

placeholder

```
5 * 0 = 0
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
```


10

Exemples de liaisons

- l'ordre d'appel peut être modifié

```
void show(const string& a, const string& b){
  cout << a << " ** " << b << endl;
}

int main(){
  auto x {bind(show, _1, _2)};
  auto y {bind(show, _2, _1)};
  auto z {bind(show, "hello", _1)};
  x("one", "two");
  y("one", "two");
  z("one");
}
```

```
one ** two
two ** one
hello ** one
```

- Attention: les éventuels paramètres par défaut dans la signature sont ignorés

Les adaptateurs ref et cref

- std::reference_wrapper permet de manipuler une référence comme un objet (que l'on peut copier ou auquel on peut affecter une valeur).
  - std::ref et std::cref génèrent des instances de cette classe
- ref et cref permettent de manipuler une référence comme un objet
  - que l'on peut copier ou auquel on peut affecter une valeur

```
void display (int i, ostream& o) {o << i;}

vector <int> v {78, -123, 89, 12};
for_each (v.begin(), v.end(),
  bind (display, _1, ref (cout)));
```

référence

élément courant parcouru

Usage d'un objet

```
struct Player {
  void start(){ cout << "start" << endl; }
  void stop(){ cout << "stop" << endl; }
};

Player p;
function <void()> cmd1 {bind(&Player::start, ref(p))};
auto cmd2 {bind(&Player::stop, ref(p))};

// …
cmd1();
```

éviter une copie

invocation

13

Les lambda-expressions

facultatif si vide

```
[contexte](paramètres) {corps}
```

```
array<int, 5> tab {89, -45, 78, 89, 123};
size_t nbNégatifs {};
for_each (tab.begin(), tab.end(),
  [&nbNégatifs](int n){if (n < 0) ++nbNégatifs;}
);
cout << nbNégatifs << "\n";
```


14

Contexte d'une lambda-expression

- [] contexte masqué
- [=] les données accessibles en lecture seule (passage par copie)
- [&] les données accessibles en lecture / écriture (passage par référence)
- [var1, &var2] var1 est accessible en R, var2 en RW
- [this] les membres sont accessibles en RW

```
vector<int> c{ 1, 2, 3, 4, 5, 6, 7 };
int seuil{ 5 };

c.erase(
  remove_if(
    begin(c),
    end(c),
    [seuil](int n){ return n < seuil; }
  ),
  cend(c)
);

for_each(cbegin(c), cend(c),
  [](int n){cout << n << "\n"; });
```

```
5
6
7
```


Lambda-expressions et généricité

C++ 14

- La généricité s'exprime par l'usage du mot-clé auto
  - à la fois pour les paramètres et le type de retour

```
auto identity {[](auto x) {return x;}};
cout << identity (123) << identity("hello") << endl;
```

```
auto multiply {[](auto x, auto y) {return x * y;}};
cout << multiply (3, -4.2) << endl;
```

int

double

Lambdas et constexpr

C++ 17

```
constexpr auto add {[](auto a, auto b) {return a + b;}};

array arr<int, add(1, 2)>;
```