ArtWorks

Advanced C++

Move semantics

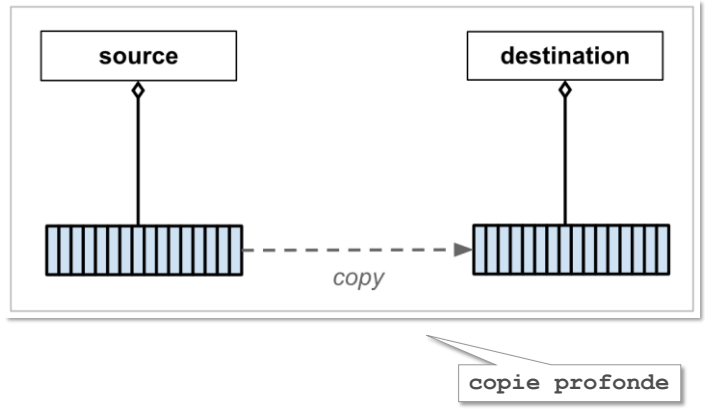
Microsoft Certified Professional Solution Developer Trainer

CompTIA. Certified Technical Trainer+

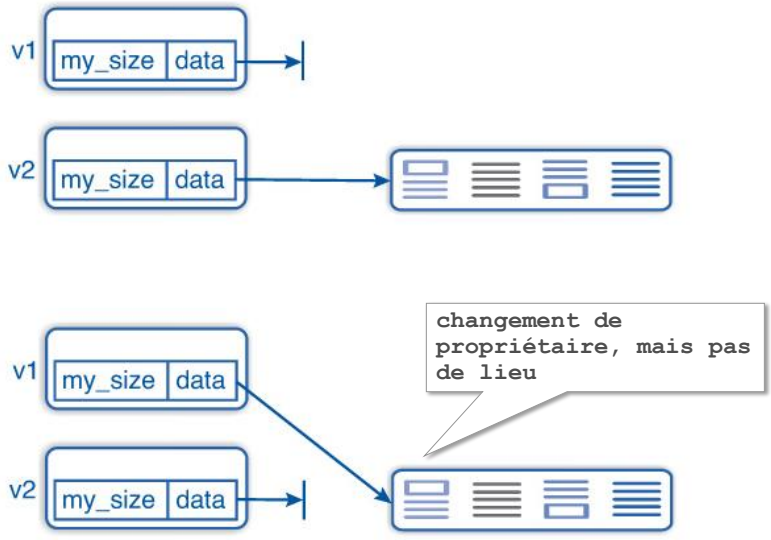
Michel André
michel@artworks.fr

Copie versus déplacement

- la construction et l'affectation par copie peuvent être couteux
- prélever l'implémentation de l'objet source est plus efficace



Données "déplacées"



3

Objet anonyme (ou temporaire)

- un nouveau terme fait son apparition avec C++11
  - right value reference
- il s'agit en fait d'une référence sur un objet anonyme

```
#include <iostream>

ofstream {"toto.txt"} << "hello";


string s;
std::getline(ifstream{"toto.txt"}, s);
cout << s << endl;


int m {123};
int&& n {m + 4};

int&& n2 {n};
```

objet anonyme

objet anonyme

right value reference

objet anonyme

échec

4

Move semantics

- un nouveau spécificateur est introduit: && (right value reference)
  - permet de référencer et "fixer" un objet temporaire anonyme
    - Une rvalue reference est donc une lvalue

```
string str {"hello"};
string& rStr {str};

string&& rvr {str + " you !"};
rvr[0] = 'Z';
cout << rvr << endl;
```

l-value

l-value reference

r-value

r-value reference

Zello you !

- move permet de "transmettre" l'implémentation d'un objet nommé
  - de façon statique, il s'agit d'un cast

```
string s1 {"bonjour"};
string s2 {move(s1)};
cout << s1 << "***" << s2 << endl;
```

aiguillage vers le move constructor

```
*** bonjour;
```

contenu transféré depuis s1 vers s2

Exemple d'usage

```
struct A {
  string s;
  A(const string& s) : s{s}{cout << "&\n";}
  A(string&& s) : s{move(s)}{cout << "&&\n";}
};
```

```
string message{ "hello" };
A a1{ message };

A a2{ "bonjour" };
A a3{ move(message) };
cout << "\"" << message << "\"" << endl;
```

usage de &

usage de &&

```
&
&&
&&
""
```

la chaîne est vide

Move constructor

- Copie superficielle (shallow copy) de l'argument
- Mise à zéro de son état
  - Pas de partage, l'objet courant possède seul l'implémentation
- L'argument peut être détruit
  - et ne doit plus être utilisé

```
MyVector:: MyVector(MyVector && a)
  : size{a.size},
    buffer{a.buffer}
{
  a.buffer = nullptr;
  a.size = 0;
}
```

"réinitialisation" de l'argument

Move assignment operator

- Pas d'allocation, donc pas de possibilité d'exception
  - libérer l'implémentation courante
  - copier l'implémentation de l'argument
  - "nettoyer" l'argument prêt à être purgé

```
MyVector& MyVector ::operator=(MyVector &&a) {
  if (&a != this) {
    delete [] buffer;
    size = a.size;
    buffer = a.buffer;
    a.size = 0;
    a.buffer = nullptr;
  }
  return *this;
}
```

copie du pointeur, pas des données pointées

"réinitialisation" de l'argument

Complétude étendue

```
class X {
 public:
  X(Sometype st);
  X();
  X(const X&);
  X(X&&);
  X& operator=(const X&);
  X& operator=(X&&);
  ~X();
  // ...
};
```


9

Héritage et move constructor

```
struct A {
  string s;
  A(const string& s) : s{s}{ cout << "A::&\n"; }
  A(string&& s) : s{move(s)}{ cout << "A::&&\n"; }
};

struct B : A {
  B(const string& s) : A{s}{ cout << "B::&\n"; }
  B(string&& s) : A{move(s)}{ cout << "B::&&\n"; }
};
```

cible

```
string s{ "deux" };
B b1{ s };

B b2{ "un" };
```

```
A::&
B::&
A::&&
B::&&
```


10

Héritage et op=

```
struct A {
  string s;
  A(const string& s) : s(s){}
  A& operator= (const A& a) {
    cout << "A::op=&\n";
    if (&a != this)
      s = a.s;
    return *this;
  }

  A& operator= (A&& a) {
    cout << "A::op=&&\n";
   if (&a != this)
     s = move(a.s);
   return *this;
  }
};
```

cible

```
struct B : A {
  B(const string& s) : A(s){}
  B& operator= (const B& b) {
    cout << "B::op=&\n";
    if (&b != this)
      A::operator= b;
    return *this;
  }
  B& operator= (B&& b) {
    cout << "B::op=&&\n";
    if (&b != this)
      A::operator= (move (b));
    return *this;
  }
```

```
B b1{"un"};
B b2{"deux"};
b1 = b2;

b1 = move (b2);
```

```
B::op=&
A::op=&
B::op=&&
A::op=&&
```


STL C++11 et swap / move

C++98 algorithm header

```
template <class T> void swap ( T& a, T& b){
  T c(a); a=b; b=c;
}
```

copie et 2 affectations

avant

C++11 utility header

```
template <class T>
void swap (T& a, T& b){
  T c{move(a)}; a=move(b); b=move(c);
}

template <class T, size_t N>
void swap (T (&a)[N], T (&b)[N]){
  for (size_t i {}; i<N; ++i)
    swap (a[i],b[i]);
}
```

swap peut être redéfini pour T

Quelques mauvaises pratiques

```
A&& getA(){
  return A{};
}
```

&& incorrect pour un retour

```
const A getA(){
  return A{};
}

void f (A&& a){
  // …
}

f (getA());
```

r-value non mutable

échec: le retour de getA() est const

13

Reference qualifiers

```
struct A {
  void doIt() & { cout << "&\n"; }
  void doIt() && { cout << "&&\n"; }
};

int main(){
  A a;
  a.doIt();
  A{}.doIt();
}
```

&
&&

14

Perfect forwarding et std::forward<T>

```
void g(const int& x){cout << "&\n";}
void g(int&& x){cout << "&&\n";}

template <class T> void f(T&& x) {
  g(x);

  g(forward<T>(x));
}
```

référence universelle

l-value dans tous les cas (pas de move)

```
int a;
f(a);
f(0);
```

génération par inférence de 2 fonctions non-membres différentes

```
&
&
&
&&
```


15