Wybrane Konstrukcje C++11

Piotr Wygocki
paal.mimuw.edu.pl

November 6, 2013

Plan:

- auto
- move semantic
- lambda

auto - podstawy

```
std::vector<int> v;
std::vector<int>::iterator i = v.begin();

for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
    //...
}

std::map<int, std::pair<double, float>> map = f();

std::iterator_traits<std::vector<int>::iterator>::difference_type d
    = i - v.begin();
```

auto - podstawy

```
std::vector<int> v;
//std::vector<int>::iterator i = v.begin();
auto i = v.begin();

for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
    //...
}

std::map<int, std::pair<double, float>> map = f();

std::iterator_traits<std::vector<int>::iterator>::difference_type d
    = i - v.begin();
```

auto - podstawy

```
std::vector<int> v;
auto i = v.begin();

for(auto i = v.begin(); i != v.end(); ++i) {
    //...
}

std::map<int, std::pair<double, float>> map = f();

std::iterator_traits<std::vector<int>::iterator>::difference_type d
    = i - v.begin();
```

auto - podstawy

```
std::vector<int> v;
auto i = v.begin();

for(auto i = v.begin(); i != v.end(); ++i) {
    //...
}

//std::map<int, std::pair<double, float>> map = f();
auto map = f();

std::iterator_traits<std::vector<int>::iterator>::difference_type d
    = i - v.begin();
```

auto - podstawy

```
std::vector<int> v;
auto i = v.begin();

for(auto i = v.begin(); i != v.end(); ++i) {
    //...
}

auto map = f();

auto d = i - v.begin();
```

nowy lepszy for

```
std::vector<int> v;
auto i = v.begin();

for(auto e : v) {
    //...
}

auto map = f();

auto d = i - v.begin();
```

c++ nadal jest silnie typowany

//auto x; //compile error!!!

Nie mamy typu... Ale można go odzyskać

//wektor elementow o typie zwracacanym przez f()

Nie mamy typu... Ale można go odzyskać

```
//wektor elementow o typie zwracacanym przez f()
typedef decltype(f()) Element;
```

Nie mamy typu... Ale można go odzyskać

```
//wektor elementow o typie zwracacanym przez f()
typedef decltype(f()) Element;

std::vector<Element> elements;
```

Nie mamy typu... Ale można go odzyskać(trudniejszy przypadek)

```
//wektor elementow o typie zwracacanym przez f(T)
typedef decltype(f(T())) Element;

std::vector<Element> elements;
```

Nie mamy typu... Ale można go odzyskać(trudniejszy przypadek)

```
//wektor elementow o typie zwracacanym przez f(T)
typedef decltype(f(T())) Element; //T moze nie miec
    defaultowego konstruktora

std::vector<Element> elements;
```

Nie mamy typu... Ale można go odzyskać(trudniejszy przypadek)

```
//wektor elementow o typie zwracacanym przez f(T)
typedef decltype(f(std::declval<T>())) Element;

std::vector<Element> elements;
```

Implementujemy declvala

```
//mozliwa implementacja
template <class T> T declval();
```

Implementujemy declvala

```
//prawdziwa implementacja
template <class T>
  typename add_rvalue_reference<T>::type declval() noexcept;
```

Nie mamy typu... Ale można go odzyskać(wynik funkcji)

```
template <typename T>
??? g(T t) {
  return f(t);
}
```

Nie mamy typu... Ale można go odzyskać(wynik funkcji)

```
template <typename T>
decltype(f(std::declval<T>())) g(T t) {
  return f(t);
}
```

Nie mamy typu... Ale można go odzyskać(wynik funkcji)

```
template <typename T>
auto g(T t) -> decltype(f(t)) {
  return f(t);
}
```

auto przez kopie, referencje i const referencje

```
auto x1 = f1();
auto & x2 = f2();
auto const & x3 = f3();
```

auto przez kopie, referencje i const referencje

```
auto x = f();

decltype(f()) x2 = f();
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

```
struct Matrix {

private:
    int * data;
};
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

```
struct Matrix {
    Matrix(int n) : data(new int[n]), size(n) {}

    Matrix(const Matrix & other) :
data(new int[other.size]), size(other.size) {
        memcopy(data, other.data, sizeof(int) * size );
    }

private:
    int * data;
    int size;
};
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

```
??? add(const Matrix &, const Matrix &)
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

```
Matrix * add(const Matrix &, const Matrix &)
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

Matrix * add(const Matrix &, const Matrix &) *// kto ma zniszczyc ten obiekt.*

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

Matrix $*$ add(const Matrix &, const Matrix &) *// kto ma zniszczyc ten obiekt. I czemu mamy w ogole martwic sie o niszczenie obiektu.*

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

```
Matrix add(const Matrix &, const Matrix &) // Chcialo by sie napisac po prostu tak.
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

```
Matrix add(const Matrix &, const Matrix &) // Chcialo by sie napisac po prostu tak. Niestety nie mozemy pozwolic sobie na kopie.
```

move semantic - podstawy

Implementujemy biblioteke dzialajaca na macierzach.

Matrix add(const Matrix &, const Matrix &) *// Chcialo by sie napisac po prostu tak. Niestety nie mozemy pozwolic sobie na kopie (W zasadzie jest RVO).*

move semantic - przerywnik na szybkie wprowadzenie do l/r - values

- l-value to wyrazenie które MOZE stac po lewej stronie przypisania
- r-value to wyrazenie które MUSI stac po prawej stronie przypisania

move semantic - przerywnik na szybkie wprowadzenie do l/r - values

- l-value

```
int x;
x = 7;

// int & f(int);
f(2) = 6;
```

- r-value

```
7 + x;
// int  g(int);
g(2);
A();
```

move semantic - przerywnik na szybkie wprowadzenie do l/r - values

- l-value
- r-value
- l-value reference

move semantic - przerywnik na szybkie wprowadzenie do l/r - values

- l-value
- r-value
- l-value reference
- **r-value reference**

move semantic - podstawy

Move constructor.

```
struct Matrix {
    Matrix(const Matrix & other) : data(new int[other.size]),
     size(other.size) {
       memcpy(data, other.data, sizeof(int) * size );
    }

    Matrix(Matrix && other) : size(other.size), data(other.data) {
      other.size = 0;
      other.data = nullptr;
    }
 private:
    int * data;
    int size;
 };
```

move semantic - podstawy

Implementujemy add.

```
Matrix add(const Matrix & left, const Matrix & right) {
    Matrix sum;
    //liczymy sume ...
    return sum; // mozliwa kopia
}
```

move semantic - podstawy

Implementujemy add.

```
Matrix add(const Matrix & left, const Matrix & right) {
    Matrix sum;
    //liczymy sume ...
    return std::move(sum);
}
```

move semantic - podstawy

Move constructor.

```
struct Matrix {
    Matrix(int n) : data(new int[n]), size(n) {}

    Matrix(const Matrix & other) : data(new int[other.size]),
     size(other.size) {
       memcpy(data, other.data, sizeof(int) * size );
    }

    Matrix(Matrix && other) = default;

 private:
    int * data;
    int size;
 };
```

move semantic - podstawy

Defaultowy move constructor.

```
struct Matrix {
    Matrix(int n) : data(new int[n]), size(n) {}

    Matrix(const Matrix & other) :
      data(new int[other.size]), size(other.size) {
       memcpy(data, other.data, sizeof(int) * size );
    }

    Matrix(Matrix && other) = default;

    Matrix & operator=(Matrix && other) = default;

  private:
    int * data;
    int size;
};
```

move semantic - podstawy

Defaultowy move constructor.

```
struct Matrix {
    Matrix(int n) : data(new int[n]), size(n) {}

    Matrix(const Matrix & other) :
      data(new int[other.size]), size(other.size) {
       memcpy(data, other.data, sizeof(int) * size );
    }

    Matrix(Matrix && other) = default; // Jest tez delete

    Matrix & operator=(Matrix && other) = default;

  private:
    int * data;
    int size;
};
```

move semantic - podstawy

Przyklad ktory naprawde dziala.

```
Matrix m, n;
std::swap(m, n);
```

move semantic - podstawy

Przyklad ktory naprawde dziala. (Tak na prawde powinnismy napisac tak).

```
Matrix m, n;
using std::swap;
swap(mat, mat);
```

move semantic - podstawy

L value reference nie konwertuje sie do r-value reference

```
struct A {
  A() = default;
  A(A && a) = default;
  A(const A & a) = delete;
};
 A a;
 // A b(a); //compile error
 A b(std::move(a));
```

move semantic - jak pisac konstruktory

Piszemy optymalny kod!

```
struct A {
  A(std::string && x) : s(std::move(x)) {}

  A(const std::string & x) : s(x) {}

  std::string s;
};
```

move semantic - jak pisac konstruktory

Piszemy optymalny kod! (tylko robi sie troche duzy)

```
struct A {
  A(std::string && x, std::string && y) : s(std::move(x)),
     t(std::move(y)) {}

  A(const std::string & x, std::string && y) : s(x), t(std::move(y))
     {}

  A(std::string && x, const std::string & y) : s(std::move(x)), t(y)
     {}

  A(const std::string & x, const std::string & y) : s(x), t(y) {}

  std::string s;
  std::string t;
};
```

move semantic - jak pisac konstruktory

Piszemy optymalny kod!

```
struct A {
  A(std::string x, std::string y) : s(std::move(x)), t(std::move(y))
      {}

  std::string s;
  std::string t;
};
```

move semantic - perfect forwarding

Jakiego typu jest T && ?

```
tempalte <typename T>
int f(T && t) {
};
```

move semantic - perfect forwarding

Pomijajac consty mamy trzy mozliwosci: T && =

- W && (jeżeli T = W)
- W & && (jeżeli T = W &)
- W && && (jeżeli T = W &&)

```
tempalte <typename T>
int f(T && t) {
};
```

move semantic - perfect forwarding

“Lvalue references are infectious” -STL

- T & & ->T &
- T & && ->T &
- T && & ->T &
- T && && ->T &&

move semantic - perfect forwarding

```
tempalte <typename T>
int f(T && t) {
  return g(t);
};
```

move semantic - perfect forwarding

```
tempalte <typename T>
int f(T && t) {
  return g(t); //t jest l-value referencja!!!
};
```

move semantic - perfect forwarding

```
tempalte <typename T>
int f(T && t) {
  return g(std::forward<T>(t));
};
```

move semantic - implementacje

Implementacja std::forward

```
template< class T >
T&& forward( typename std::remove_reference<T>::type & t);
```

move semantic - implementacje

Implementacja std::forward

```
template< class T >
T&& forward( typename std::remove_reference<T>::type & t) {
    return static_cast<T&&>(t);
}
```

move semantic - implementacje

Implementacja std::move

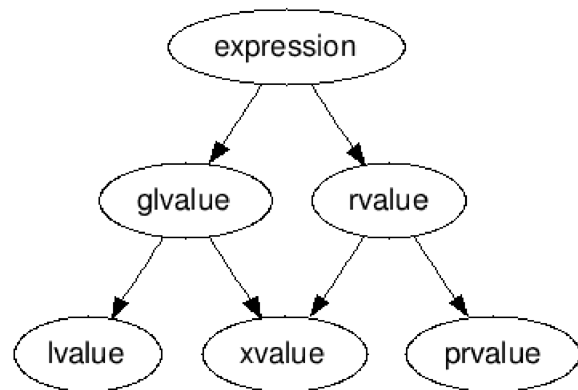
```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t );
```

move semantic - implementacje

Implementacja std::move

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t ) {
  return static_cast<typename
      std::remove_reference<T>::type&&>(t);
}
```

expresions = rodzaje



move semantic - jak pisac konstruktory

Piszemy optymalny kod!

```
struct A {
  A(std::string x, std::string y) : s(std::move(x)), t(std::move(y))
     {}

  std::string s;
  std::string t;
};
```

Lambdy podstawy

```
std::vector<A> p{A("Jes","sza"), A("mo", "ci"), A("ta", "ra")};
std::max_element(p.begin(), p.end()); // compile error - nie wie
    jak posortowac
```

Lambdy podstawy

```
bool compare(const A & left, const A & right) {
  return left.s < right.s;
}

std::vector<A> p{A("Jes","sza"), A("mo", "ci"), A("ta", "ra")};
std::max_element(p.begin(), p.end(), compare);
```

Lambdy podstawy

```
struct Compare {
  Compare(F _f) : f(_f) {}

  bool operator()(const A & left, const A & right) const {
    return f(left) < f(right);
  }

private:
  F f;
};

Compare compare(jakas_funkcja);

A aa("a", "123"), bb("sda", "3213");

compare(aa, bb);
```

Lambdy podstawy

```
struct Compare {
  Compare(F _f) : f(_f) {}

  bool operator()(const A & left, const A & right) const {
    return f(left) < f(right);
  }

private:
  F f;
};

std::vector<A> p{A("Jes","sza"), A("mo", "ci"), A("ta", "ra")};
std::max_element(p.begin(), p.end(), Compare(f));
```

Lambdy podstawy

```
std::vector<A> p{A("Jes","sza"), A("mo", "ci"), A("ta", "ra")};

auto compare = [](const A & left, const A & right){ return left.s
    < right.s;};
std::max_element(p.begin(), p.end(), compare);
```

Lambdy podstawy

```
std::vector<A> p{A("Jes","sza"), A("mo", "ci"), A("ta", "ra")};

auto compare = [=](const A & left, const A & right){ return
    f(left) < f(right);};
std::max_element(p.begin(), p.end(), compare);
```

Lambdy podstawy

- Wyrazenie [ ] - Nothing to capture: an up-level reference is an error
- Wyrazenie [&] - Capture by reference: an up-level reference implicitly captures the variable by reference
- Wyrazenie [=] - Capture by copy: an up-level reference implicitly captures the variable by copy

Lambdy podstawy

- Wyrazenie [&x, y, ...]
  Capture as specified: identifiers prefixed by & are captured by reference; other identifiers are captured by copy. An up-level reference to any variable not explicitly listed is an error
- Wyrazenie [&, x, y, ...]
  Capture by reference with exceptions: listed variables are captured by value/copy (no listed variable may be prefixed by &)
- Wyrazenie [=, &x, &y, ...]
  Capture by copy with exceptions: listed variables are captured by reference only (every listed variable must be prefixed by &)

Lambdy podstawy

```
  1   2    3        4         5
[=] () mutable throw() -> int
{
    int n = x + y;

    x = y;
    y = n;

    return n;
}
```
(6 — body)

Lambdy zajawka c++14

```
auto glambda = [](auto a) { return a; };
```

Zbugowane lambdy

```
struct X {};

template <class T = X, typename U>
void f(const U& m) {
  // auto g = [] () {};
}

int main() {
    f<>(0);
}
```

Zbugowane lambdy

gcc-4.7.1:
c.cpp: In function 'void f(const U&)':
c.cpp:5:15: error: no default argument for 'U'

```
struct X {};

template <class T = X, typename U>
void f(const U& m) {
  auto g = [] () {};
}

int main() {
    f<>(0);
}
```

Jak to odpalić?

Jak to odpalić?

- g++ –std=c++0x

Jak to odpalić?

- g++ –std=c++0x
- clang++ –std=c++0x

Jak to odpalić?

- g++ –std=c++0x
- clang++ –std=c++0x
- MSVC - nic nie trzeba

Jak to odpalic?

- g++ –std=c++0x
- clang++ –std=c++0x
- MSVC - nic nie trzeba (ale sa do tylu z implementacja)

Dziekuje za uwage!