Standard C++0x (C++1x?)

Marcin Świderski
sfider@students.mimuw.edu.pl

O czym będzie mowa?

- Wytyczne komitetu standaryzacyjnego
- Rozszerzenia języka
- Rozszerzenia języka – szablony
- Rozszerzenia biblioteki standardowej

Wytyczne komitetu standaryzacyjnego

- Zachować stabilność oraz zgodność wstecz
- Dodawać nowe funkcjonalności poprzez bibliotekę standardową, a nie rozbudowę języka
- Dodawać funkcjonalności rozwijające technikę programowania
- Skupić się na usprawnieniu projektowania systemów oraz bibliotek

Wytyczne komitetu standaryzacyjnego c.d.

- Zwiększyć bezpieczeństwo typologiczne
- Poprawić wydajność i możliwość bezpośredniej pracy ze sprzętem
- Zapewnić rozwiązania realnych problemów
- Zasada “zerowego narzutu”
- Prosty dla początkujących, użyteczny dla ekspertów

Rozszerzenia języka

- `nullptr` (N1488)
- Listy inicjalizujące (N1509)
- Silnie typowane enumeracje (N1513)
- Rozszerzone wyrażenia stałe (N1521)
- Delegowanie konstruktorów (N1581)
- `explicit` w zastosowaniu do operatorów rzutowania (N1592)
- Referencja rvalue (semantyka przenoszenia) (N1690)

Rozszerzenia języka c.d.

- Rozszerzenie semantyki przenoszenia na **`*this`** (N1784)
- Lambda wyrażenia i domknięcia (N1968)
- Automatyczne wyznaczanie typu oraz nowa deklaracja funkcji (N1978)
- Pętla `for` po zakresie (N2049)
- Wielowątkowy model pamięci (N2052)

Rozszerzenia języka - szablony

- Szablony ze zmienną liczbą parametrów (N1483)
- Aliasy szablonów (N1489)
- Nawiasy ostrokątne (N1649)
- `extern` w zastosowaniu do szablonów (N1960)
- Koncepcje (N2042)

Rozszerzenia biblioteki standardowej

- Wsparcie dla wielowątkowości (N2320)
- ???

Listy inicjalizujące

- Konstruktor parametryzowany sekwencją

```
class SequenceClass {
public:
  SequenceClass(std::initializer_list<int> list);
};

SequenceClass someVar = {1, 4, 5, 6};
```

- Typy użytkownika (kontenery w szczególności) są traktowane na równi z typami wbudowanymi (założenie języka C++)

Silnie typowane enumeracje

```
enum class Enum : unsigned int { Val1, Val2 };
```

- Brak ukrytej konwersji do użytego typu
- Ograniczony zakres widoczności nazw

```
Val1 // invalid
Enum::Val1 // valid
```

Rozszerzone wyrażenia stałe

```
constexpr int GetFive() { return 5; }

int some_value[GetFive() + 5];
```

- Funkcja nie-void poprzedzone słowem kluczowym **`constexpr`**
- Zawiera tylko instrukcję **`return`** przekazującą wartość będącą wyrażeniem stałym

Delegowanie konstruktorów

```
class SomeType {
  int number;
public:
  SomeType(int newNumber) : number(newNumber) {}
  SomeType() : SomeType(42) {}
};

class DerivedType : public SomeType {
public:
  using default SomeType;
};

class SomeOtherType {
  int number = 42;
};
```

Referencja rvalue (semantyka przenoszenia)

- Aktulanie funkcja nie jest w stanie rozpoznać obiektu tymczasowego (przekazanego przez **`const`**`&`)
- Nowy typ referencji (`&&`)
- Semantyka przenoszenia
- Konstruktor przenoszący (przyjmujący referencję rvalue) dla `std::vector` pozwala na ominięcie kopiowania tablicy zawartej w kontenerze

Rozszerzenie semantyki przenoszenia na `*this`

- Wyznaczanie metody wywoływanej w przypadku lvalue i rvalue

```
class X {
    std::vector<char> data_;
public:
    // ...
    std::vector<char> const & data() const & {
      return data_;
    }
    std::vector<char> && data() && {
      return data_;
    }
};
```

Lambda wyrażenia i domknięcia

```
<>(int x, int y) (x + y) // wyrażenie

<>(int x, int y) -> int { // funkcja
  int z = x + y;
  return z + x;
}

int total;
<>(int x) : [&total] (total += x) // referencja
<>(int x) : [total] (total += x) // kopia
<&>(int x) (total += x) // referencja to stosu
<=>(int x) (total += x) // kopia stosu

<>() : [this] (this->SomePrivateMemberFunction());
```

Automatyczne wyznaczanie typu

- słowo kluczowe **auto**

```
for (vector<int>::const_iterator i = myvec.begin();
i != myvec.end(); ++i)

for (auto i = myvec.begin(); i != myvec.end(); ++i)
```

- słowo kluczowe **decltype**

```
int a = 1;
float b = 1.5f;
decltype(a); // int
decltype(b); // float
decltype(a + b); // float
```

Automatyczne wyznaczanie typu c.d.

- Nowa deklaracja funkcji

```
auto f(int) -> int;

template <class T, class U>
auto add(T t, U u) -> decltype(t + u);
```

- Pozwala na uzależnienie typu wyniku od typów argumentów

Pętla `for` po zakresie

- Do biblioteki standardowej zostanie dołączona koncepcja zakresu (Boost)
- Dodana zostanie także konstrukcja językowa oparta na tej koncepcji

```
int my_array[5] = {1, 2, 3, 4, 5};
for (int &x : my_array) {
  x *= 2;
}
```

Szablony ze zmienną liczbą parametrów

```
template<typename... Args> class count;

template<>
struct count<> {
  enum { value = 0 };
};

template<typename T, typename... Args>
struct count<T, Args...> {
  enum { value = 1 + count<Args...>::value };
};
```

Szablony ze zmienną liczbą parametrów c.d.

```
void printf(const char* s) {
  while (*s) {
    if (*s != '%' || *++s == '%') std::cout << *s++;
    else throw std::runtime_error();
  }
}
template<typename T, typename... Args>
void printf(const char* s, T value, Args... args) {
  while (*s) {
    if (*s != '%' || *++s == '%') std::cout << *s++;
    else {
      std::cout << value;
      return printf(*++s, args...);
    }
  }
  throw std::runtime_error();
}
```

Aliasy szablonów

```
template<
   typename first,
   typename second,
   int third
> class SomeType;

template<typename second>
using TypedefName =
   SomeType<OtherType, second, 5>;
```

Koncepcje

```
auto concept LessThanComparable<typename T> {
  bool operator<(T, T);
};
template<LessThanComparable T>
const T& min(const T& x, const T& y) {
  return x < y ? x : y;
};

auto concept Regular<typename T> {
  T::T();
  T::T(const T&);
  T::~T();
  T& operator=(T&, const T&);
  bool operator==(const T&, const T&);
  bool operator!=(const T&, const T&);
  void swap(T&, T&);
};
```

Koncepcje c.d.

```
auto concept Convertible<typename T, typename U> {
  operator U(const T&);
};

template<typename U, typename T>
  where Convertible<T, U>
U convert(const T& t) {
  return t;
}
```

Składanie koncepcji

```
concept InputIterator<
  typename Iter, typename Value
> {
  where Regular<Iter>;
  Value operator*(const Iter&); // dereference
  Iter& operator++(Iter&); // pre-increment
  Iter operator++(Iter&, int); // post-increment
}

concept ForwardIterator<
  typename Iter, typename Value
> : InputIterator<Iter, Value> {
/* no syntactic differences,
 * but adds the multi-pass property
 */
}
```

Mapowanie koncepcji

```
concept_map InputIterator<int> {
  typedef int value_type;
  typedef int reference;
  typedef int* pointer;
  typedef int difference_type;

  int operator(int x) { return x; }
};

copy(1, 17,
  std::ostream_iterator<int>(std::cout, " "));
```

Mapowanie koncepcji c.d.

```
concept BackInsertionSequence<typename X> {
  typename value_type = X::value_type;
  void X::push back(const value_type&);
  void X::pop back();
  value_type& X::back();
  const value_type& X::back() const;
  bool X::empty() const;
};
template<BackInsertionSequence X>
concept_map Stack<X> {
  typedef X::value_type value_type ;
  void push(X& x, const value_type &value) {
    x.push_back(value);
  }
  void pop(X& x) { x.pop_back(); }
  T top(const X& x) { return x.back(); }
  bool empty(const X& x) { return x.empty(); }
};
```

Koncepcje c.d.

- Poprawność szablonów używających koncepcji jest sprawdzana w definicji szablonu
- Szablon może korzystać jedynie z tych funkcjonalności typu, które są zdefiniowane przez koncepcje
- Koncepcje pozwalają na przeładowanie szablonów

Źródła

- JTC1/SC22/WG21 - The C++ Standards Committee: http://www.open-std.org/JTC1/SC22/WG21
- Wikipedia: http://en.wikipedia.org/wiki/C%2B%2B0x

Dziękuję i zapraszam do dyskusji