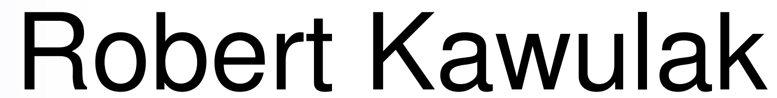

Robert Kawulak

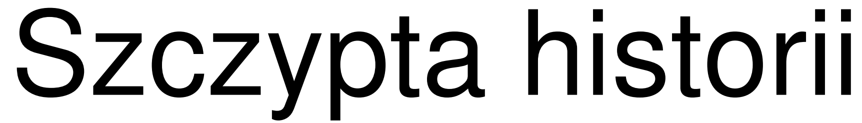
C++09

Co nowego w nadchodzącym standardzie języka?

Wprowadzenie

Zawsze marzyłem żeby mój komputer był tak samo łatwy w obsłudze jak mój telefon. Moje marzenie się spełniło. Już nie umiem obsłużyć telefonu.

Bjarne Stroustrup



Szczypta historii

- 1979: Bjarne S. rozpoczyna pracę nad *C with classes*
- 1983: Bjarne S. zmienia nazwę na *C++*, dodając: funkcje wirtualne, przeładowanie funkcji i operatorów, referencje i in.
- 1989: *C++ Release 2.0* – wielokrotne dziedziczenie, klasy abstrakcyjne, statyczne funkcje składowe, `protected`, ANSI rozpoczyna standaryzację
- 1990: *The Annotated C++ Reference Manual* – szablony, wyjątki, przestrzenie nazw, `bool`
- 1998: ukończenie standaryzacji języka przez ISO (ANSI jako doradca) – C++98
- 2003: wydanie standardu z poprawkami – C++03
- 2005: TR1 - `shared_ptr`, `bind`, `type_traits`, `regex`, `array` i in.
- 2009?: C++0x, czyli C++, jakiego jeszcze nie widziałeś!

Jak tworzony jest standard C++?

- standard otwarty, nie związany z żadną firmą
- duży wpływ społeczności użytkowników (każdy może coś zaproponować)
- kilkudziesięcioosobowy komitet ekspertów - ochotników
- regularne spotkania, głosowania i sugestie
- po standaryzacji 5-letni okres zamrożenia (jedynie poprawki błędów)

Główne kryteria rozwoju C++

- bezpieczeństwo (świadomość zagrożeń)
- łatwość uczenia („The problem with using C++ is that there's already a strong tendency in the language to require you to know everything before you can do anything” - Larry Wall)
- przenaszalność
- wydajność („nie płać za to czego nie używasz” – „zero-overhead rule”)
- brak ukierunkowania na konkretny styl programowania (programowanie proceduralne, obiektowe, generyczne...)
- dawanie programiście wyboru (nawet jeśli może wybrać źle)
- wsteczna kompatybilność z C (na ile to możliwe)
- unikanie elementów specyficznych dla danej platformy lub nie będących ogólnie użytecznymi
- jeśli nowy element może być zrealizowany bibliotecznie, to nie powinien być dodawany do samego języka, ale do biblioteki standardowej

Więcej w książce Bjarne’a *The Design And Evolution of C++*

Opis nowości

Wątki

- zadanie trudniejsze niż się z pozoru wydaje (asynchroniczne wyjątki, zmiana modelu pamięci itd.)
- większość kompilatorów i tak je implementuje, ale „na dziko”
- trzeba zastosować podejście mieszane (język + biblioteka)
  - język: nowy model pamięci, elementarna synchronizacja, nowe znaczenie `volatile`
  - biblioteka: tworzenie i zarządzanie wątkami, elementy synchronizacji wyższego poziomu

Garbage Collector

- `smart_ptr` niewystarczający
- GC tak, ale:
  - opcjonalny
  - niewidzialny dla programisty
- pozostaje problem finalizacji

Koncepty

- pozwalają łatwiej tworzyć bezpieczniejsze szablony
- upraszczają użycie szablonów
- przykład:

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

co się stanie, gdy napiszemy:

```
std::complex<float> a, b;
min(a, b);
```

rozwiązanie:

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

Automatyczna dedukcja typów

- `auto` – automatycznie nadaje typ nowej zmiennej
  - problem:
    ```
    std::map< std::basic_string<char>, std::complex<double> > cont;
    for(std::map< std::basic_string<char>, std::complex<double>
      >::const_iterator i = c.begin(), end = c.end(); i != end; ++i)
    {...}
    ```
  - rozwiązanie:
    ```
    for(auto i = c.begin(), end = c.end(); i != end; ++i)
    {...}
    ```
- `decltype` – pozwala otrzymać typ wyrażenia
  - problem:
    ```
    template<class A, class B>
    ??? add(const A& a, const B& b) { return a + b; }
    ```
  - rozwiązanie:
    ```
    template<class A, class B>
    decltype(a + b) add(const A& a, const B& b) { return a + b; }
    ```

Konstrukcja obiektów

- konstruktor sekwencyjny
  - problem – pojemników nie można inicjalizować tak jak tablic:

```
std::vector<int> v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

  - rozwiązanie:

```
template<typename T>
class vector
{
    ...
    vector(initializer_list<int> seq) {...}
};
```

- delegacja konstruktorów

```
struct S {
        S(int i, string s) {...}
        S(int i) : S(i, ””) { }
        S(string s) : S(0, s) { }
        S() : S(0, ””) { }
};
```

- dziedziczenie konstruktorów

```
struct X : public S {
        using S::S;
        X(int i) : S(i, ”X”) { }
};
X a(4, ”4”), b(8), c(”c”), d;
```

Moduły

- podejście C (nagłówki i jednostki translacji):
  - długa kompilacja
  - słabe odseparowanie interfejsu od implementacji
  - trudne ukrycie implementacji
- przykład nowego podejścia:

```
// Plik1.cpp:
export Lib {
    import std;
public:
    struct S {
      S() { std::cout << "S()\n"; }
    };
}

// Plik2.cpp:
import Lib;
int main() {
    S s;
}
```

- korzyści:
  - szybsza kompilacja
  - uniknięcie problemów z interferencją makr
  - ukrycie danych prywatnych
  - większa pewność odpowiedniej kolejności inicjalizacji danych statycznych
  - uniknięcie niewykrytych problemów ODR („one definition rule” – „zasada jednej definicji”)
  - umożliwia tworzenie bibliotek dynamicznych
  - upraszcza drogę do implementacji szablonów `export`

Wyrażenia lambda

- umożliwiają łatwe tworzenie obiektów funkcyjnych
- podejście biblioteczne ma pewne ograniczenia
- przykład – szukanie w pojemniku `v` liczb mniejszych od 4:
  - tradycyjnie:

```
struct lessthan {
  int val;
  lessthan(int i) : val(i) { }
  bool operator()(int i) const { return i < val; }
};
find_if(v.begin(), v.end(), lessthan(4));
```

  - z wykorzystaniem bindingu:

```
find_if(v.begin(), v.end(), std::bind2nd(std::less<int>(), 4));
```

  - z wyrażeniami lambda:

```
find_if(v.begin(), v.end(), _1 < 4);
```

Wybrane pozostałe drobne zmiany

- statyczne asercje

```
static_assert(sizeof(T) > 1);
```

- cechy typów (`type_traits`), np: `is_empty`, `is_abstract`
- problem >> w szablonach:

```
std::vector<std::complex<double>> vec;
```

- `long long` (śladami C99)
- aliasy szablonów

```
template<typename T>
using string_map = std::map<std::string, T>;
```

- bezpieczniejsze typy wyliczeniowe

```
enum class Val : unsigned long { E1, E2, E3 = 10, E4 /* = 11 */ };
Val v = 10;
Val v = Val::E3;
```

- pętle `for_each`:

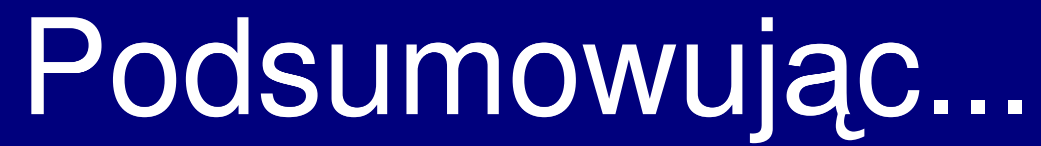
```
std::vector<int> v;
for_each(auto x : v)
    std::cout << x << std::endl;
```

- bardziej „literalne" literały znakowe
  - do tej pory:

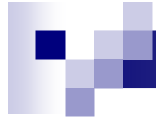
    ```
    "<HEAD>\n"
    "<META HTTP-EQUIV=\"Content-Type\" CONTENT=\"text/html; charset=windows-1252\">\n"
    "</HEAD>\n"
    ```

  - w C++09:

    ```
    R"$\
    <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=windows-1252">
    </HEAD>
    $"
    ```


Podsumowując...

- http://www.open-std.org/JTC1/SC22/WG21/
- http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2122.htm
- http://en.wikipedia.org/wiki/C++0x
- http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=216



Pytania...?

Dziękuję za uwagę!