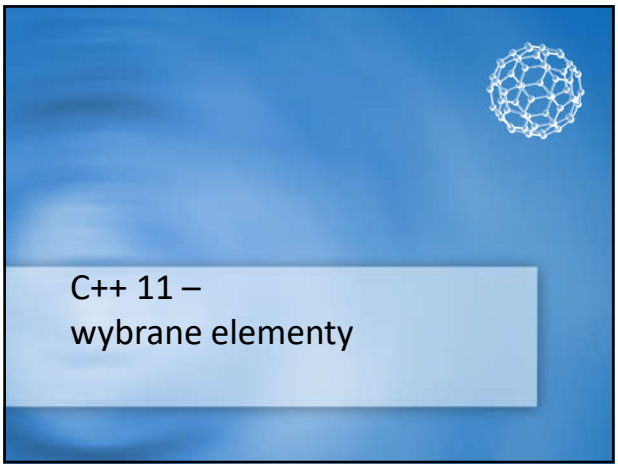


1

C++11

Lista rozszerzeń C++11 obecnych w VC2013:

1. Referencje do *rvalue*,
2. Jednolite inicjowanie i delegowanie konstruktorów,
3. Konstruktory delegujące
4. Jednolita inicjalizacja
5. Klasa `initializer_list`
6. `static_assert`
7. Domyślne wartości parametrów szablonu dla szablonów funkcji
8. Szablony ze zmienną liczbą parametrów
9. `typedef` i aliasy
10. Szablony zewnętrzne
11. Pętla **`for`** oparta na zakresie,
12. Dedukcja typów danych,
13. Sprytne wskaźniki,
14. **`nullptr`**
15. Kontenery mieszające (kontenery z haszowaniem).

2

C++xx

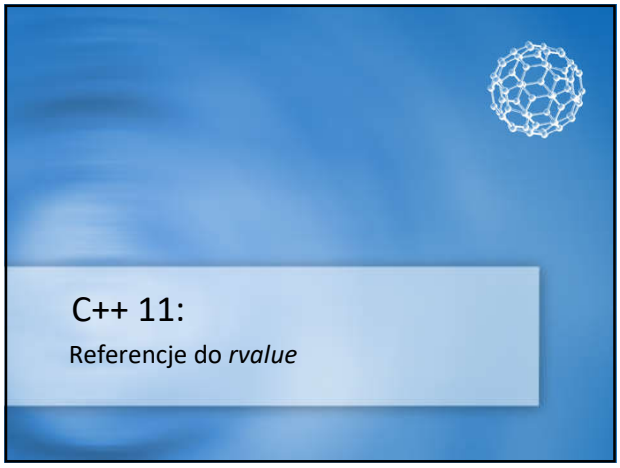
- Do MSVC w kolejnych wersjach środowiska dodawane są kolejne rozszerzenia składni z C++14, C++17, C++20

Więcej informacji na stronie:

Microsoft C++ language conformance table

https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-language-conformance?view=msvc-160

3



4

C++11: referencje do *rvalue*

lvalue i rvalue

lvalue - wyrażenie które odnosi się do obszaru pamięci (obiektu, który ma nazwę) i pozwala na pozyskanie adresu tego obszaru za pomocą operatora &

rvalue - wyrażenie nie będące *lvalue*

```
int foobar();

int j = 0;
j = 42;                  // ok, 42 jest rvalue
j = foobar();            // ok, foobar() jest rvalue
int* p2 = &foobar();     // błąd, nie można pobrać adresu
```


5

C++11: referencje do *rvalue*

Przykładowy problem

Przyjmijmy, że mamy klasę w której jedno z pól jest wskaźnikiem **`m_pResource`** np. do pewnego kontenera. Przeciążony operator kopiowania dla tej klasy:

```
X& X::operator=(X const & rhs) {
  // [...]
  // zniszcz zasób wskazywany przez m_pResource
  // sklonuj to, na co wskazuje rhs.m_pResource
  // dołącz klon do m_pResource.
  // [...]
}
```

Teraz można wywołać kod:

```
X foo();   // dana jest pewna funkcja foo..

X x;
x = foo(); // wywołanie przeciążonego operatora przypisania
```


6

C++11: referencje do *rvalue*

Przykładowy problem (*move semantics*)

Zdecydowanie mniejszy nakład pracy dałby taki przeciążony operator:

```
X& X::operator=(X const & rhs) {
// [...]
// swap ( m_pResource , rhs.m_pResource )
// [...]
```

To jest właśnie **semantyka przenoszenia** (*move semantics*) – *pozwolenie pewnemu obiektowi, pod pewnymi warunkami przejąć kontrolę nad zewnętrznymi zasobami innego obiektu. Dzięki temu unikamy kosztownych kopiowań.*

Żeby `swap` zadziałało, potrzebujemy, aby argument operatora był **referencją**.

Jeżeli argument jest *lvalue* – nie ma problemu. A jeżeli jest *rvalue*?..

.. wtedy przydałby się drugi referencyjny typ danych, który byłby wybierany, kiedy argumentem jest *rvalue*: `&&`

C++11: referencje do *rvalue*

Przykładowy problem (*move semantics*)

Rozwiązanie:

```
void foo(X& x) { .. };     // wersja z referencją do lvalue
void foo(X&& x) { .. };    // wersja z referencją do rvalue

X x;
X foobar() { .. ; return ..; };

foo(x);         // argument jest lvalue: wywołuje foo(X&)
foo(foobar()); // argument jest rvalue: wywołuje foo(X&&)
```

Na etapie kompilacji rozstrzyga się, jaki argument zostanie podany w wywołaniu.

C++11: referencje do *rvalue*

Przykładowy problem (*move semantics*)

Rozwiązanie dla przeciążonego operatora przypisania:

```
X& X::operator=(X const & rhs); // klasyczna implementacja

X& X::operator=(X&& rhs) {
  // Move semantics: zamiana zawartości pomiędzy this i rhs
  // ..
  return *this;
}
```

Uwaga: tak, to prawda, że `&&` można stosować w dowolnej funkcji, ale przyjmuje się stosowanie wyłącznie do przeciążonych operatorów przypisania i konstruktorów kopiujących.

C++11: referencje do *rvalue*

Semantyka przenoszenia danych

Czy zmienna referencyjna do *rvalue* jest traktowana jako *rvalue*?

Przykład:

```
void foo(X&& x) {
  X anotherX = x; // który operator przypisania zadziała?
  // ...
}
```

(`x` jest referencją do *rvalue*, więc powinien `operator=(X&& x)`..)

Nie.

Prosta zasada mówi: jeżeli coś ma nazwę, to jest *lvalue*. W przeciwnym razie – jest *rvalue*.

C++ 11:

Konstruktory delegujące

C++11: konstruktory delegujące

Delegowanie:

Problem: w kodzie napisanym w C++98 jest dana klasa, która ma kilka konstruktorów. Często część kroków inicjalizujących powtarza się w każdym z nich, np.:

```
class A{
public:
   A(): num1(0), num2(0) {average=(num1+num2)/2;}
   A(int i): num1(i), num2(0) {average=(num1+num2)/2;}
   A(int i, int j): num1(i), num2(j) {average=(num1+num2)/2;}
private:
   int num1;
   int num2;
   int average;
};
```

C++11: konstruktory delegujące

Delegowanie:

W C++11 dostajemy możliwość stworzenia jednego konstruktora docelowego i kilku delegujących:

```
class A{
   public:
      A(): A(0){}              // A() deleguje do A(0)
      A(int i):  A(i, 0){}     // A(int i) deleguje do A(i,0)
      A(int i, int j) { num1=i; num2=j; average=(num1+num2)/2; }
   private:
      int num1;
      int num2;
      int average;
   };
```

Uwaga: w konstruktorach delegujących nie wolno w liście inicjalizatorów konstruktora dodawać instrukcji inicjalizacji pól

C++11: konstruktory delegujące

Delegowanie i wyjątki:

Wywołanie docelowego konstruktora może być zamknięte w bloku **try**:

```
class A{
public:
   A();
   A(int i);
   A(int i, int j);
private:
   int num1;
   int num2;
   int average;
};
```

```
A:: A()try: A(0) {
   cout << "A() body"<< endl;
}
catch(...) {
   cout << "A() catch"<< endl;
}

A::A(int i) try: A(i, 0){
   cout << "A(int i) body"<< endl;
}
catch(...) {
   cout << "A(int i) catch"<< endl;
}
```


C++11: konstruktory delegujące

Delegowanie i wyjątki:

Wywołanie docelowego konstruktora może być zamknięte w bloku **try**:

```
class A{
public:
   A();
   A(int i);
   A(int i, int j);
private:
   int num1;
   int num2;
   int average;
};
```

```
A::A(int i, int j) try {
   num1=i;
   num2=j;
   average=(num1+num2)/2;
   cout << "A(int i, int j) body"<< endl;
   throw 1;
}
catch(...) {
   cout << "A(int i, int j) catch"<< endl;
}
```


C++ 11:

Jednolita inicjalizacja

C++11: jednolita inicjalizacja

Jednolita inicjalizacja za pomocą nawiasów klamrowych:

Możliwa do zastosowania do dowolnej klasy, struktury czy unii, o ile istnieje (zdefiniowany jawnie lub *implicite*) konstruktor domyślny:

```
class class_a {
public:
    class_a() {} // konstruktor domyślny
    class_a(string str) :
       m_string{ str } {}
    class_a(string str, double dbl) :
       m_string{ str },
       m_double{ dbl } {}

double m_double;
string m_string;
};
```

```
int main()
{
    class_a c1{};
    class_a c1_1;
    class_a c2{ "ww" };
    class_a c2_1("xx");
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}
```

Kolejność argumentów jak w konstruktorze, ponieważ ta inicjalizacja de facto wywołuje konstruktor

C++11: jednolita inicjalizacja

Jednolita inicjalizacja za pomocą nawiasów klamrowych:

Jeżeli nie zdefiniowano żadnego konstruktora, domyślny porządek argumentów taki jak pól w deklaracji klasy/struktury/unii:

```
class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};
```

```
int main()
{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // błąd
    class_d d5{ "string", 'c', 2.0 }; // błąd
}
```

C++11: jednolita inicjalizacja

Jednolita inicjalizacja za pomocą nawiasów klamrowych:

Można jej używać w dowolnym miejscu kodu, np.:

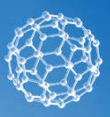
```
class_d* cf = new class_d{4.5};
```

albo

```
kr->add_d({ 4.5 });
```

albo

```
return { 4.5 };
```

C++ 11:

Klasa **initializer_list**

C++11: initializer_list

Lista obiektów do inicjalizacji:

Klasa reprezentuje listę obiektów jednego, określonego typu, które mogą być używane w konstruktorze i innych inicjalizujących kontekstach

```
#include <initializer_list>
..
initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "tak!" << endl; // spodziewamy się "tak!"
```

Standardowe klasy kontenerów z STL, a także np. **string** mają zdefiniowane konstruktory przyjmujące jako argument listę obiektów.

C++11: initializer_list

Lista obiektów do inicjalizacji:

Można używać list w konstruktorach własnych klas:

```
class C1
{
public:
 list<int> L;
 C1(const std::initializer_list<int>& v) : L(v.begin(), v.end()) {}
};

int main() {
 C1 c{ 3, 5, 7, 11 }; // inicjalizujemy listą wartości
 ..
```


C++11: initializer_list

Lista obiektów do inicjalizacji:

Można używać list do inicjalizacji klas kontenerów wyrażeniami z nawiasami klamrowymi

```
vector<int> v1{ 9, 10, 11 };

map<int, string> m1{ {1, "a"}, {2, "b"} };

string s{ 'a', 'b', 'c' };
```


C++ 11:

static_assert

C++11: static_assert

Asercja statyczna:

Testuje zadane wyrażenie na etapie kompilacji

Składnia:

```
static_assert(
    constant-expression,
    string-literal
);
```

Jeżeli **constant-expression** zwraca **false**, wypisywany jest komunikat **string-literal**, a kompilacja kończy się niepowodzeniem,

25

C++11: static_assert

Asercja statyczna:

Przykłady użycia

Jeżeli nieprawda, że rozmiar =4B, czyli, jeżeli architektura nie jest 32-bitowa, tzn. że musi być 64-bitowa, a 64-bitowej platformy nie wspieramy..

```
static_assert(sizeof(void *) == 4,
   "64-bit code generation is not supported.");
```

Jeżeli nieprawda, że wersja > 2, tzn. że musi być 2 lub mniej, a tak starych wersji już nie wspieramy..

```
static_assert(SomeLibrary::Version > 2,
   "Old versions of SomeLibrary are missing the foo
   functionality. Cannot proceed!");
```


26

C++11: static_assert

Asercja statyczna:

Przykłady użycia

```
class Foo {
    public:
        static const int bar = 3; // prowokujemy błąd..
};
static_assert(Foo::bar > 4, "Foo::bar is too small :(");

int main() {
    return Foo::bar;
}
```


27

C++11: static_assert

Asercja statyczna:

Przykłady użycia

```
template < class T, int Size >
class Vector {
   static_assert(Size > 3, "Vector size is too small!");
   T m_values[Size];
};
int _tmain(int argc, _TCHAR* argv[]) {
   Vector< int, 4 > four;  // prawda, że: Size>3
   Vector< short, 2 > two; // nieprawda, że: Size>3 - błąd
   return 0;
}
```


28

C++ 11:

Domyślne wartości parametrów szablonu dla szablonów funkcji

29

C++11: domyślne argumenty szablonu

Domyślne wartości parametrów szablonu dla szablonów funkcji:

W C++98 w szablonach klas dozwolone były domyślne argumenty szablonu. W szablonach funkcji – nie.

Zaczynając od VS2013 wolno nam:

```
template <typename T = int> void Foo(T t = 0) { }

Foo(12L);     // Foo<long>
Foo(12.1);    // Foo<double>
Foo('A');     // Foo<char>
Foo();        // Foo<int> (!)
```


30

C++11: domyślne argumenty szablonu

Domyślne wartości parametrów szablonu dla szablonów funkcji:

```
template <typename T>
class Manager {
public:
  void Process(T t) { }
};

template <typename T>
class AltManager {
public:
  void Process(T t) { }
};
```

```
template <typename T, typename M=Manager<T>>
void Manage(T t) {
  M m;
  m.Process(t);
}

Manage(25);
        // Manage<int, Manager<int>>

Manage<int, AltManager<int>>(25);
        // Manage<int, AltManager<int>>
```


C++11: domyślne argumenty szablonu

Domyślne wartości parametrów szablonu dla szablonów funkcji:

Trzeba jednak uważać na niejednoznaczności:

```
template <typename T = int>
void Foo(T t = 0) { }

template <typename B, typename T = int>
void Foo(B b = 0, T t = 0) { }

Foo(12L);     // nie skompiluje się
Foo(12.1);    // nie skompiluje się
Foo('A');     // nie skompiluje się
Foo();        // Foo<int>
```


C++ 11:

Metody usunięte i jawnie domyślne

C++11: metody usunięte i jawnie domyślne

W C++11 kompilator generuje automatycznie dla przykładowej klasy **`Klasa`**:

Konstruktor domyślny: **`Klasa()`**
Konstruktor kopiujący: **`Klasa(const Klasa& k)`**
Operator przypisania: **`operator=(const Klasa& k)`**
Destruktor: **`~Klasa()`**
Konstruktor przenoszący: **`Klasa(const Klasa&& k)`**
Operator przypisania przenoszący: **`operator=(const Klasa&& k)`**

C++11: metody usunięte i jawnie domyślne

Reguły tworzenia:

1. **Jeżeli** choć jeden konstruktor został utworzony jawnie, **to** żadne inne domyślne nie są tworzone
2. **Jeżeli** wirtualny destruktor został utworzony jawnie, **to** domyślny destruktor nie jest tworzony
3. **Jeżeli** konstruktor przenoszący lub operator przypisania przenoszący został utworzony jawnie, **to** nie są tworzone domyślnie konstruktor domyślny ani operator przypisania domyślny
4. **Jeżeli** jawnie stworzone zostały: konstruktor kopiujący lub przenoszący, operator przypisania zwykły lub przenoszący, lub destruktor, **to** nie są tworzone domyślnie konstruktor przenoszący ani operator przypisania przenoszący

C++11: metody usunięte i jawnie domyślne

Reguły tworzenia:

Te reguły rzutują na obecność składowych klas w przypadku dziedziczenia.

Np. **jeżeli** klasa bazowa nie ma własnego konstruktora domyślnego, który mógłby zostać wywołany w klasie pochodnej, **to** w klasie pochodnej kompilator nie wygeneruje jej własnego konstruktora domyślnego.

Dotychczas, aby jawnie zarządzać tworzeniem lub zakazem tworzenia konstruktorów stosowana była ich deklaracja w sekcji **`private`**:

```
private:
CMySingleton() {}
~CMySingleton() {}
CMySingleton(const CMySingleton&);     // Prevent copy-construction
CMySingleton& operator=(const CMySingleton&); // Prevent assignment
```

C++11: metody usunięte i jawnie domyślne

Wady dotychczasowego rozwiązania:

1. Aby ukryć konstruktor kopiujący, trzeba go zadeklarować prywatnie, a skoro jest – trzeba też zadeklarować domyślny, choćby nic nie robił, bo inaczej sam się nie utworzy.
2. Nawet jeżeli domyślny nic nie robi, to jest zadeklarowany, a w związku z tym daje większy koszt obliczeniowy wywołania, niż ten utworzony automatycznie przez kompilator.
3. Konstruktor kopiujący i operator przypisania są zadeklarowane prywatne, ale w metodach klasy **`friend`** oraz w metodach tej klasy można próbować je wywołać – wygenerują błąd linkera (nie mają kodu).
4. Cała konstrukcja klasy dla kogoś, kto nie uważał na wykładach z PO i ZTP, jest niejasna i nieoczywista.

C++11: metody usunięte i jawnie domyślne

Składnia w C++98:

```
struct niekopiowalny {
  niekopiowalny() {};
private:
  niekopiowalny(const niekopiowalny&);
  niekopiowalny& operator=(const niekopiowalny&);
};
```

Składnia w C++11:

```
struct niekopiowalny {
  niekopiowalny() =default;
  niekopiowalny(const niekopiowalny&) =delete;
  niekopiowalny& operator=(const niekopiowalny&) =delete;
};
```


C++11: metody usunięte i jawnie domyślne

Składnia w C++11 – korzyści:

1. Generowanie konstruktora domyślnego nadal jest wstrzymane z racji deklaracji konstruktora kopiującego, ale.. można to zmienić korzystając z deklaracji **`=defaut`**.
2. Konstruktor kopiujący i operator kopiowania są publiczne, ale usunięte, więc przy próbie ich wywołania lub definiowania pojawi się błąd kompilacji.
3. Intencja autora kodu staje się czytelniejsza, nawet ktoś, kto nie uczęszczał na PO i ZTP, ma szansę się domyślić, co wolno a czego nie wolno używać.

C++11: metody usunięte i jawnie domyślne

Określanie „**`=default`**” poza deklaracją klasy:

```
struct widget {
  widget()=default;
  inline widget& operator=(const widget&);
};

inline widget& widget::operator=(const widget&) =default;
```

Konstruktor lub operator można wskazać jako „default” poza deklaracją klasy tylko pod warunkiem, że został zadeklarowany jako **`inline`**.

Wskazówka: ze względu na wyższą efektywność, tam gdzie to możliwe, należy zamieniać puste ciała konstruktorów i operatorów na polecenia **`=default`**

C++11: metody usunięte i jawnie domyślne

Blokowanie niechcianych wywołań:

Jeżeli nie chcemy, aby obiekty danego typu były tworzone dynamicznie:

```
struct widget  {
  void* operator new(std::size_t) = delete;
};
```

Jeżeli nie chcemy określonych konwersji argumentów w wywołaniach metod lub funkcji:

```
void call_with_true_double_only(float) = delete;
void call_with_true_double_only(double param) { return; }
```

Podanie argumentu typu **`int`** wywoła jego konwersję do **`double`**.

C++11: metody usunięte i jawnie domyślne

Blokowanie niechcianych wywołań:

Jeżeli nie chcemy żadnych konwersji argumentów w wywołaniach metod lub funkcji i chcemy aby argumenty jednego i tylko jednego typu były akceptowane:

```
template < typename T >
void call_with_true_double_only(T) = delete;

void call_with_true_double_only(double param) {
  ...
  return;
}
```