24

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 =4, czyli, że architektura nie jest 32-bitowa, tzn. że musi być 64-bitowa,a tej 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 tej 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

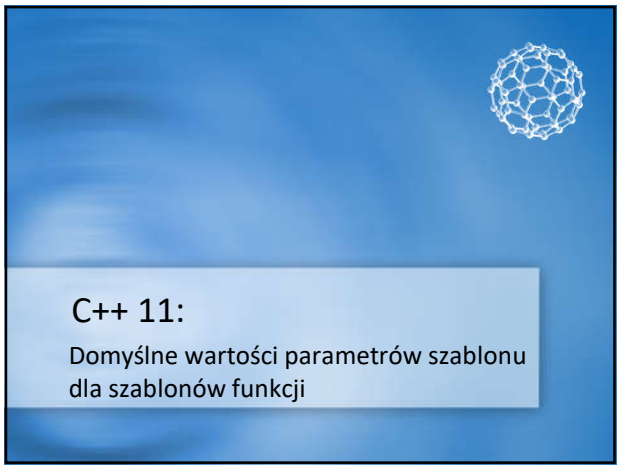
C++11: static_assert

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



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> (!)
```


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`.

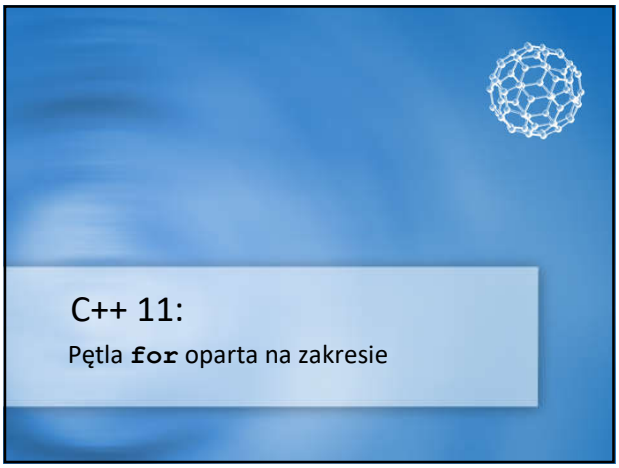
C++11: metody usunięte i jawnie domyślne

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;
}
```

C++11: pętla *for* oparta na zakresie

Nowa składnia pętli **for**

for (deklaracja : wyrażenie) instrukcje

wyrażenie – tablica/kontener/dowolny obiekt implementujący semantykę kontenera (zwraca obiekt w stylu iteratora za pomocą metod **begin** i **end**)

deklaracja zmiennej - typu takiego jak elementy w tablicy lub kontenerze

instrukcje – wykonywane tyle razy, ile elementów przechowuje tablica lub kontener

Przykład:

```
vector<int> vec;
vec.push_back( 10 );
vec.push_back( 20 );

for (int i : vec ) {
    cout << i;
}
```


C++11: pętla *for* oparta na zakresie

Przykłady

Modyfikowanie zawartości kontenera:

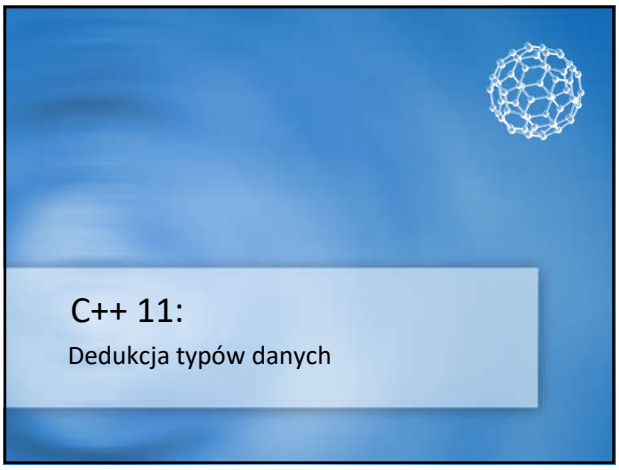
```
vector<int> vec;
vec.push_back( 1 );
vec.push_back( 2 );

for (int& i : vec )
{
    i++; // inkrementuje wartości z wektora
}
```


C++11: pętla *for* oparta na zakresie

Obiekty posiadające zakres to:

- wszelkie kontenery z biblioteki STL , ponieważ:
  1. posiadają iteratory, które wspierają metody **operator*** , **operator!=** i **operator++** (zadeklarowane jako metody własne albo jako funkcje globalne),
  2. posiadają metody **begin** i **end** (zadeklarowane jako metody własne albo jako funkcje globalne), zwracające iteratory na początek i koniec kontenera.
- wszelkie inne obiekty zaimplementowane samemu o funkcjonalności kontenerów STL (obowiązkowa w/w funkcjonalność).



C++11: dedukcja typów danych

Mechanizmy dedukcji typów danych

Pochodzący C++98:

1. **Szablony i ich parametry**

Wprowadzone w C++11:

1. `auto`
2. `decltype`

- Pozwalają uniknąć ciągłego wypisywania typów, które są oczywiste bądź powtarzają się.
- Zmiana typu zmiennej w jednym miejscu kodu propaguje się na cały kod (kompilator sam dedukuje nowy typ danych – *uwaga: może nas zaskoczyć*).

C++11: dedukcja typów danych

Zasady dedukcji dla **`auto`**

Podobne do zasad dedukcji dla parametrów szablonu z C++98

Deklarując `auto` jako typ zmiennej

```
auto x = 10;
```

`auto` odgrywa rolę taką, jak parametr `T` w szablonie:

```
template <typename T>
void fun(T x) ;
fun(10);
```

Stąd:

```
const char name[] = "ABC";  // name: const char[13]
auto arr1 = name;           // arr1: const char*

void Fun(int, double);      // Fun: void(int, double)
auto func1 = Fun;      // func1: void (*)(int, double)
```


C++11: dedukcja typów danych

Pożytek z **`auto`**

Mamy szablon funkcji, która na pewnym zakresie elementów kontenera identyfikowanym przez parę iteratorów, wykonuje „czynności ważne i potrzebne”:

```
template<typename It>   // wykonuje czynności ważne i potrzebne
void czwip(It b, It e)  // na rzecz elementów od 'b' do 'e'
{
  for (; b != e; ++b) {
    typename std::iterator_traits<It>::value_type currValue = *b;
    ...
  }
}
```

Odwołujemy się pracowicie do cech charakterystycznych iteratora, żeby pozyskać typ danych wskazywanych przez te iteratory.

C++11: dedukcja typów danych

Pożytek z **`auto`**

Mamy szablon funkcji, która na pewnym zakresie elementów kontenera identyfikowanym przez parę iteratorów, wykonuje „czynności ważne i potrzebne”:

```
template<typename It>   // wykonuje czynności ważne i potrzebne
void czwip(It b, It e)  // na rzecz elementów od 'b' do 'e'
{
  for (; b != e; ++b) {
    auto currValue = *b;
    ...
  }
}
```

Wersja dla mniej pracowitych. ☺

C++11: dedukcja typów danych

Pożytek z **`auto`**

Jeszcze jeden przykład dla „mniej pracowitych”:

```
map<string, string> address_book;
for ( auto address_entry : address_book )
{
   cout  << address_entry.first << " < " <<
            address_entry.second << ">" << endl;
}
```


C++11: dedukcja typów danych

Pożytek z **`auto`**

Kiedy bezpieczniej jest pisać **`auto`** – przykład: pozyskanie rozmiaru kontenera:

```
std::vector<int> v;
...
unsigned sz = v.size();
```

Co prawda typ zwracany przez **`.size()`** to: **`vector<int>::size_type`**

ale wszyscy wiedzą, że to typ całkowity bez znaku, więc spotyka się **`unsigned`**.

Nie wszyscy jednak wiedzą, że choć w Win32 zmienne tych dwóch typów zajmują te samą liczbę bajtów, to już w Win64 nie (**`unsigned`** zajmuje mniej) ..

Rekompilacja kodu na Win64 powiedzie się, ale dla dostatecznie dużych zbiorów danych działanie może przestać być poprawne. ☹

Dlatego prościej i bezpieczniej pisać: **`auto sz = v.size();`**

C++11: dedukcja typów danych

Niespodzianki z **auto**

Przykład: mamy funkcję, która dla pewnej klasy widgetów zwraca wektor opisujący ich cechy, tj. co potrafią, a czego nie:

```
std::vector<bool> cechy(const Widget& w);
```

Kod programu:

```
Widget w;
...
bool cecha5 = cechy(w)[5];     // czy posiada cechę nr 5
  ..
```

A gdyby tak napisać:

```
auto cecha5 = cechy(w)[5];     // czy posiada cechę nr 5
```

Powinno działać tak samo.

C++11: dedukcja typów danych

Niespodzianki z **auto**

Dla:

```
auto cecha5 = cechy(w)[5];
```

Typ zmiennej **cecha5** to **vector<bool>::reference** (klasa zagnieżdżona w **vector<bool>**).

Powód jest natury technicznej: klasa **vector<bool>** została tak zaprojektowana, aby przechowywać wartości w postaci spakowanej – jeden bit na każdy element (a w C++ nie ma referencji do pojedynczych bitów).

Stąd **operator[]** dla **vector<bool>** zwraca obiekt, który *działa tak jak* **bool&**

Oczywiście, wśród jego wielu funkcjonalności, istnieje też możliwość konwersji tego obiektu do **bool**.

C++11: dedukcja typów danych

Niespodzianki z **auto**

Dla:

```
auto cecha5 = cechy(w)[5];
```

Typ zmiennej **cecha5** to **vector<bool>::reference** (klasa zagnieżdżona w **vector<bool>**).

Teraz zmienna **cecha5** jest kopią obiektu przechowującego referencję do właściwej informacji – bitu nr 5.

Niestety, funkcja **cechy** zwraca referencję do obiektu, który za chwilę zniknie (funkcja cechy zwraca kontener – obiekt tymczasowy, więc jego elementy też są tymczasowe..).

Dlatego **cecha5** będzie przechowywać referencję do obiektu, który nie istnieje. ☹

C++11: dedukcja typów danych

Niespodzianki z **auto**

Klasa:

```
vector<bool>::reference
```

to tzw. klasa proxy (*proxy class*). Takich klas jest wiele.

Przestroga: należy unikać sytuacji w kodzie, gdzie:

```
auto pewnazmienna = wyrażenie typu 'klasa proxy';
```

No tak, tylko że one zostały tak napisane, żeby być jak najmniej widoczne (w założeniu: mają być całkiem niewidoczne).

Aby się nie dać oszukać, należy:

sprawdzać nagłówki funkcji i metod (np. w źródłach bibliotek), albo ..

```
auto cecha5 = static_cast<bool>(cechy(w)[5]);
```


C++11: dedukcja typów danych

Dedukcja typu przez **decltype**

zwraca typ zmiennej lub wyrażenia podanego w argumencie:

```
const int i = 0;            // decltype(i): const int
bool f(const Widget& w);    // decltype(w): const Widget&
                            // decltype(f): bool(const Widget&)

struct Point {
  int x, y;                 // decltype(Point::x): int
};                          // decltype(Point::y): int

Widget w;                   // decltype(w): Widget

if (f(w)) …                 // decltype(f(w)): bool
```


C++11: dedukcja typów danych

Dedukcja typu przez **decltype**

Przydaje się szczególnie kiedy szablon funkcji ma zwrócić typ danych zależny od typu parametru (ale nie będący typem parametru)

Przykład: funkcja zwracająca element z kontenera po uprzedniej autoryzacji dostępu do danych dla bieżącego użytkownika:

```
template<typename Container, typename Index>
… (?) authAndAccess(Container& c, Index i) {
                …
```

Funkcja powinna zwracać typ taki, jak ma **c[i]**

Jak go pozyskać?

C++11: dedukcja typów danych

Dedukcja typu przez **`decltype`**

Dygresja:

W C++11 wprowadzono możliwość podania typu zwracanego przez funkcję w miejscu tuż za jej nagłówkiem (tzw. *trailing return type*); bardzo przydatne, kiedy typ zwracanych danych ma skomplikowany zapis, np. :

```
auto fpif(int) -> int(*)(int)
```

Koniec dygresji.

Drugie zastosowanie: kiedy zwracany typ zależy argumentów wywołania (*trt*):

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i) -> decltype(c[i])
{
  authenticateUser();      // autoryzacja dostępu
  return c[i];             // dostęp do danych
}
```