System typów

dr inż. Paweł Kłeczek

2021

Aliasy

alias – inna nazwa dla istniejącego już typu danych, traktowana jako nowy, pełnoprawny typ danych

Definicja aliasu od C++11:

```
using identyfikator_nowego_typu = opis_istniejacego_typu;
```

Przykład #1:

```
using byte = uint8_t;
byte b;  // równoważnie: uint8_t b;
```

Przykład #2:

```
using pesel_t = unsigned char[11];  // alias na 11-elementową
pesel_t p = {4, 4, 0, 5, 1, 4, 0, 1, 4, 5, 8};  //  tablicę
```

Zasięg aliasu zależy od położenia instrukcji **using** (zasady są podobnie jak m.in. w przypadku definiowania zmiennych).

Przenośność oprogramowania

Przykłady uzyskania przenośności za pomocą aliasów:

- `sizeof` – „zwraca wartość typu całkowitego”
- `time()` – „zwraca wartość typu całkowitego”

. . . ale konkretny użyty typ całkowity zależy od konfiguracji sprzętowej.

Przykład:

```
time_t time(time_t*);
```

Jak zdefiniowano typ `time_t`?

- maszyna #1 – `unsigned int`
- maszyna #2 – `unsigned long`

Alias `size_t`

alias `size_t` (wg standardu języka C):

> „odnosi się do typu całkowitego bez znaku będącego rezultatem działania pewnych operatorów, m.in. **`sizeof`**”

$\Rightarrow$ standard nie określa konkretnego użytego typu – wymaga jednak, aby był on w stanie przechowywać maksymalną liczbę bajtów zajmowaną przez teoretycznie możliwy do utworzenia obiekt

Stosuj alias `size_t` jako typ zmiennych wyrażających liczbę elementów (np. tablicy, kontenera) – nie stosuj do tego typów wbudowanych.

Niejawne konwersje typów dla klas

konstruktor konwertujący (ang. converting constructor) – *każdy* konstruktor, który przyjmuje pojedynczy argument

```
struct Foo {
  // konstruktor KONWERTUJĄCY
  //   (bo przyjmuje pojedynczy argument typu innego niż `Foo`)
  Foo(const std::string& s) : n(s.size()) {}
  std::size_t n;
};

void bar(const Foo& f) { /* ... */ }

int main() {
  std::string str = "abc";
  bar(str);  // `bar()` oczekuje argumentu typu `Foo`, więc
             //    przekazując obiekt typu `std::string`
             //    wywołany zostanie niejawnie konstruktor
             //    konwertujący `Foo(const std::string&)`
  return EXIT_SUCCESS;
}
```

Niejawne konwersje typów dla klas

Kompilator jest w stanie wykonać tylko jedną niejawną konwersję na raz!

```
// BŁĄD: wymagane dwie zdefiniowane przez użytkownika konwersje:
//   (1) konwertuj "9-999-99999-9" typu `const char[]`
//         na typ `std::string`
//   (2) konwertuj ten (tymczasowy) obiekt typu `std::string`
//         na typ `Foo`
bar("9-999-99999-9");
```

W takich przypadkach musimy jedną z konwersji określić jawnie, z użyciem odpowiedniego konstruktora:

```
// OK: jawna konwersja do std::string, niejawna do Foo
bar(std::string("9-999-99999-9"));
// OK: niejawna konwersja do std::string, jawna do Foo
bar(Foo("9-999-99999-9"));
```

Operatory rzutowania

rzutowanie (ang. cast) – wymuszenie interpretacji wartości typu `T1` jako wartości typu `T2`

Problem:

```
int i = 1;
int j = 2;
double slope = i / j;   // chcemy potraktować `i` i `j`
                        //   jako wartości typu `double`
```

Choć w pewnych przypadkach jawne rzutowanie jest niezbędne, co do zasady jest to bardzo niebezpieczna konstrukcja języka C++.

Ogólna postać wszystkich operatorów rzutowania w C++:

```
XXX_cast<type>(expression);
```

(`XXX_cast` – rodzaj rzutowania, `type` – typ docelowy, `expression` – rzutowana wartość)

Operator `static_cast`

`static_cast` – pozwala m.in. na jawne dokonanie tych konwersji, które zwykle są dokonywane niejawnie oraz na konwersję między wskaźnikami na powiązane klasy (i to nie tylko „w górę”, ale także „w dół”)

Przykład:

```
int i = 1;
int j = 2;
double slope = static_cast<double>(i) / j;
```

Kompilator nie weryfikuje, czy żądana konwersja faktycznie ma sens!

Przykład:

```
class Base {};
class Derived: public Base {};
Base* a = new Base;
Derived* b = static_cast<Derived*>(a); // NIEBEZPIECZNE!
```

Operator `dynamic_cast`

`dynamic_cast` – umożliwia konwersję „w górę” i „w dół” dla wskaźników i referencji na klasy; dodatkowo gwarantuje, że wynikiem rzutowania będzie poprawny (kompletny) obiekt pożądanego typu

Podstawowe formy operatora **`dynamic_cast`**:

```
dynamic_cast<type*>(expression)
dynamic_cast<type&>(expression)
```

gdzie `type` musi być typu klasowego, a `expression` to:

- albo obiekt klasy dziedziczącej *publicznie* po `type`,
- albo *publiczna* klasa macierzysta klasy `type`,
- albo ta sama klasa co `type`.

Jeśli `expression` nie spełni tych kryteriów, **`dynamic_cast`** odpowiednio:

- zwróci wskaźnik pusty w przypadku rzutowania na wskaźnik, albo
- rzuci wyjątek **`bad_cast`** w przypadku rzutowania na referencję.