

Kurs języka C++

15. Operatory, funkcje i klasy narzędziowe z biblioteki standardowej

Spis treści

- ▶ Pary i tuple
- ▶ Sprytne wskaźniki
- ▶ Ograniczenia liczbowe
- ▶ Minimum i maksimum
- ▶ Zamiana wartości
- ▶ Operatory porównywania
- ▶ Typy wyliczeniowe

Pary

- ▶ Szablon struktury `pair<>` (zdefiniowany w `<utility>`) umożliwia potraktowanie dwóch wartości jako pojedynczego elementu.
- ▶ Para posiada dwa pola: `first` i `second`.
- ▶ Do pól pary `first` i `second` można się dowoływać za pomocą funkcji szablonowej odpowiednio `get<0>(p)` i `get<1>(p)`, gdzie `p` to para.
- ▶ Para posiada konstruktor dwuargumentowy oraz domyślny i kopiujący.
- ▶ Pary można porównywać (operatory `==` i `<`).
- ▶ Istnieje szablon funkcji `make_pair()` do tworzenia pary (typy danych są rozpoznawane przez kompilator po typach argumentów).

Pary

► Przykłady:

```
void f (std::pair<int, const char*>);  
void g (std::pair<const int, std::string>);  
...  
std::pair<int, const char*> p(44, "witaj");  
f(p); // wywołuje domyślny konstruktor kopiujący  
g(p); // wywołuje konstruktor szablonowy  
g(std::make_pair(44, "witaj")); // przekazuje dwie  
    // wartości jako parę z wykorzystaniem konwersji  
    // typów
```

► Pary są wykorzystywane w kontenerach `map` i `multimap`.

Tuple

- ▶ W C++11 zdefiniowano tuple do przechowywania wielu wartości a nie tylko dwóch (szablon `tuple<>` jest analogią do szblonu pary `pair<>`).
- ▶ Tupla posiada wiele ponumerowanych pól, do których dostęp mamy za pomocą funkcji `get<i>`.
- ▶ Tupla posiada konstruktor wieloargumentowy oraz domyślny i kopiujący.
- ▶ Tuple można porównywać za pomocą operatorów porównań (porównywanie leksykograficzne).
- ▶ Istnieje szablon funkcji `make_tuple()` do tworzenia tupli (typy danych są rozpoznawane przez kompilator po typach argumentów).
- ▶ Istnieje szablon funkcji `tie()` do tworzenia tupli z referencjami (jako argumenty podaje się zmienne).
- ▶ Szablon `tuple_size<tupletype>::value` służy do podania liczby elementów w tupli.
- ▶ Szablon `tuple_element<idx, tupletype>::type` służy do podania typu elementu o indeksie `idx` w tupli.

Tuple - przykład

```
std::tuple<double, char, std::string> get_student(int id) {  
    if (id == 0)  
        return std::make_tuple(3.8, 'A', "Lisa Simpson");  
    if (id == 1)  
        return std::make_tuple(2.9, 'C', "Milhouse Houten");  
    if (id == 2)  
        return std::make_tuple(1.7, 'D', "Ralph Wiggum");  
    throw std::invalid_argument("id");  
}  
// ...  
auto student0 = get_student(0);  
std::cout << "ID: 0, "  
            << "GPA: " << std::get<0>(student0) << ", "  
            << "grade: " << std::get<1>(student0) << ", "  
            << "name: " << std::get<2>(student0) << '\n';
```

Sprytne wskaźniki

- ▶ Sprytne wskaźniki są zdefiniowane w pliku nagłówkowym `<memory>`.
- ▶ Zastąpienie szablonu `auto_ptr<>`.
- ▶ Szablon klasy `shared_ptr<>` - wiele takich sprytnych wskaźników może przechowywać wskaźnik do tego samego obiektu, tak że obiekt ten oraz związane z nim zasoby zostaną zwolnione dopiero po likwidacji ostatniego sprytnego wskaźnika.
- ▶ Szablon klasy `unique_ptr<>` - tylko jeden sprytny wskaźnik może przechowywać wskaźnik do tego danego obiektu.

Sprytne wskaźniki

- ▶ Wskaźniki typu `shared_pointer<>` implementują semantykę sprzątania po niepotrzebnym już obiekcie.
- ▶ Ostatni istniejący wskaźnik współdzielony odnoszący się do tego samego obiektu jest odpowiedzialny za zwolnienie tego obiektu i jego zasobów (użycie operatora `delete`).
- ▶ Inicjalizacja wskaźnika współdzielonego:
 - ▶ za pomocą listy wartości, na przykład:

```
shared_ptr<string> pNico{  
    new string("nico")};
```
 - ▶ za pomocą funkcji `make_shared()`:

```
shared_ptr<string> pJutta =  
    make_shared<string>("jutta");
```


Ograniczenia liczbowe

- ▶ Typy numeryczne posiadają ograniczenia zależne od platformy i są zdefiniowane w szablonie `numeric_limits<>` (zdefiniowany w `<limits>`, stałe preprocesora są nadal dostępne w `<climits>` i `<cfloat>`).
- ▶ Wybrane składowe statyczne szablону `numeric_limits<>`:
`is_signed`, `is_integer`, `is_exact`,
`is_bounded`, `is_modulo`, `has_infinity`,
`has_quiet_NaN`,
`min()`, `max()`, `epsilon()`.
- ▶ Przykłady:
`numeric_limits<char>::is_signed;`
`numeric_limits<short>::is_modulo;`
`numeric_limits<long>::max();`
`numeric_limits<float>::min();`
`numeric_limits<double>::epsilon();`

Minimum i maksimum

- ▶ Obliczanie wartości minimalnej oraz maksymalnej:

```
template <class T>
inline const T& min (const T &a, const T &b)
    { return b<a ? b : a; }
template <class T>
inline const T& max (const T &a, const T &b)
    { return a<b ? b : a; }
```

- ▶ Istnieją też wersje tych szablonów z komparatorami (funkcja lub obiekt funkcyjny):

```
template <class T, class C>
inline const T& min (const T &a, const T &b, C comp)
    { return comp(b, a) ? b : a; }
template <class T>
inline const T& max (const T &a, const T &b, C comp)
    { return comp(a, b) ? b : a; }
```

Minimum i maksimum

▶ Przykład 1:

```
bool int_ptr_less (int *p, int *q) {  
    return *p < *q; }  
  
...  
int x = 33, y = 44;  
int *px = &x, *py = &y;  
int *pmax = std::max(px, py, int_ptr_less);
```

▶ Przykład 2:

```
int i;  
long l;  
  
...  
// niezgodne typy argumentów  
// l = max(i, l); // BŁĄD  
  
...  
l = std::max<long>(i, l); // OK
```

Zamiana wartości

- ▶ Zamiana dwóch wartości:

```
template <class T>
inline void swap (T &a, T &b) {
    T tmp(move(a)); a = move(b); b = move(tmp);
}
```

- ▶ Przykład:

```
int x = 33, y = 44;
...
std::swap(x, y);
```

Operatory porównywania

- ▶ Cztery funkcje szablonowe (zdefiniowane w `<utility>`) na podstawie operatorów `==` i `<` definiują operatory porównań `!=`, `<=`, `>=` i `>`.
- ▶ Funkcje te są umieszczone w przestrzeni nazw `std::rel_ops`.

- ▶ **Przykład:**

```
namespace std { namespace rel_ops { struct porown {}; } }

class X : private std::rel_ops::porown {
    // ...
public:
    bool operator== (const X &x) const noexcept { ... }
    bool operator< (const X &x) const noexcept { ... }
};

...
// using namespace std::rel_ops;
X x1, x2;

...
if (x1 >= x2) { ... }

...
if (x1 != x2) { ... }

...
```

Operator statku kosmicznego

- ▶ W wersji c++20 wprowadzono operator porównania trójwartościowego `<=>`, który dla porównania `x<=>y` zwraca jedną z trzech wartości:
 - ▶ `strong_ordering::less`, gdy `x < y`;
 - ▶ `strong_ordering::equal`, gdy `x == y`;
 - ▶ `strong_ordering::greater`, gdy `x > y`.

Wyliczenia enum z zasięgiem

- ▶ Deklarowanie nazwy wewnątrz nawiasów klamrowych ogranicza widoczność nazwy do zasięgu definiowanego przez te nawiasy klamrowe. Nie jest tak w przypadku wyliczeń deklarowanych w stylu C++98 za pomocą **enum**.

- ▶ Przykład:

```
enum Color {black, white, red};  
// black, white, red są w tym samym zasięgu co Color  
auto white = false; // błąd!  
// white ma już deklarację w tym zasięgu
```

- ▶ W C++11 wyliczenia enum z zasięgiem nie powodują wyciekania nazw w ten sposób:

- ▶ Przykład:

```
enum class Color {black, white, red};  
// black, white, red mają zasięg Color  
auto white = false; // dobrze  
// nie ma innego white w zasięgu  
Color c = white; // błąd!  
// brak wyliczenia o nazwie white w tym zasięgu  
Color c = Color::white; // dobrze  
auto c = Color::white; // też dobrze
```

Wyliczenia enum z zasięgiem

- ▶ Redukcja zanieczyszczenia przestrzeni nazw oferowana przez wyliczenia enum z zasięgiem jest powodem, aby wybierać je zamiast enum bez zasięgu.
- ▶ Wyliczenia enum z zasięgiem są znacznie mocniej typowane - brak jest niejawnej konwersji na inny typ (wyliczenia dla enum bez zasięgu są w sposób niejawny konwertowane na typy całkowite).
- ▶ W standardzie C++11 wyliczenia enum z zasięgiem mogą być deklarowane z wyprzedzeniem. Przykład:
`enum class Color;`

Wyliczenia enum z zasięgiem

- ▶ W celu wydajnego użycia pamięci kompilatory często chcą wybierać najmniejszy podstawowy typ całkowitoliczbowy dla wyliczenia **enum** bez zakresu, który wystarcza do reprezentacji zakresu wartości wyliczenia. Przykład:

```
enum Status {  
    good = 0,  
    faled = 1,  
    incomplete = 100,  
    corrupt = 200,  
    indeterminate = 0xFFFFFFFF  
};
```

- ▶ Domyślnie typem podstawowym wyliczeń enum z zasięgiem jest `int`. Jeżeli domyślny typ nam nie odpowiada, możemy go nadpisać. Przykład:

```
enum class Status: std::uint32_t;
```

Wyliczenia enum z zasięgiem

▶ Przykłady:

▶ `using UserInfo = std::tuple< // alias typu
 std::string, // nazwa
 std::string, // email
 std::size_t> ; // reputacja`

`...
UserInfo uInfo; // obiekt typu tuple`

`...
auto val = std::get<1>(uInfo); // pobierz email`

▶ `enum UserInfoFields {uiName, uiEmail, uiReputation};
UserInfo uInfo; // jak poprzednio`

`...
auto val = std::get<uiEmail>(uInfo); // pobierz email`