C++ i STL zaawansowane techniki

Nowe elementy w składni języka

Literatura

- N.M.Josuttis: *C++. Biblioteka standardowa. Podręcznik programisty. Wydanie 2. Rozdział 3: nowe elementy języka.* Wydawnictwo Helion, Gliwice 2014.
- J.Galowicz: *C++17 STL. Receptury. Rozdział 1: Nowe funkcje w C++17.* Wydawnictwo Helion, Gliwice 2018.
- Raw String Literal in C++: https://www.geeksforgeeks.org/raw-string-literal-c/

Nowe standardy

- C++98, to pierwszy historycznie standard C++ przyjęty w roku 1998. Numer dokumentu: ISO/IEC 14882:1998.
- C++03, to rewizja techniczna standardu C++, przyjęta w 2003 roku. Spisuje ona pomniejsze poprawki do standardu C++98. Tak więc zarówno C++98, jak i C++03 składają się na pierwszy standard C++. Numer dokumentu: ISO/IEC 14882:2003.
- Język C++ zyskał wiele usprawnień wraz z wydawaniem nowych standardów C++11, C++14, C++17 i C++20. Obecnie jest to zupełnie inny język w porównaniu do tego, którym był jeszcze przed 2011 rokiem.
- Standard C++ nie dotyczy jedynie języka, który musi być obsługiwany przez kompilatory, ale również standardowej biblioteki szablonów STL.

Odstępy w wyrażeniach szablonowych

- Zniknęło od dawna krytykowane wymaganie rozdzielania odstępami znaków nawiasów kątowych zamykających wyrażenie szablonowe.
- Przykład starej notacji:
```
vector<list<int> > lst;
```
- Przykład nowej notacji:
```
vector<list<int>> lst;
```

Wskaźnik pusty nullptr zamiast NULL albo 0

- Typ wskaźnikowy to nie to samo co liczba całkowita `int`.
- Literał `0` i symbol `NULL` jest typu `int`, nie jest to wskaźnik.
- Wartość `nullptr` nie ma typu całkowitoliczbowego. Nie ma też typu wskaźnikowego, ale możemy myśleć o niej jako o wskaźniku wszystkich typów. Typem wartości `nullptr` faktycznie jest `std::nullptr_t`, zdefiniowany w nagłówku `<cstddef>`.
- Typ `std::nullptr_t` niejawnie konwertuje się na wszystkie typy wskaźników surowych, a to sprawia, że `nullptr` działa tak, jakby był wskaźnikiem wszystkich typów.
- Typ `std::nullptr_t` zaliczany jest do typów podstawowych.

Surowe łańcuchy znakowe

- Możemy uniknąć specjalnego traktowania „znaków specjalnych” w literałach znakowych stosując surowe literały łańcuchowe.
- Surowy łańcuch znakowy zaczyna się od `R"(`, a kończy się `)"`.
  Przykłady:

```
std::string no_newlines = R"(\n\n)";
std::string cmd(R"(cd "C:\new folder\text")");
std::string with_newlines(R"(Line 1 of text...
                             Line 2...
                             Line 3...)");
```

- W niektórych sytuacjach ograniczniki mogą posiadać unikalną nazwę.
  Przykład:

```
std::string ala = R"bolek(Ala ma kota.)bolek";
```

Aliasy zamiast definiowania typów

- Instrukcja `typedef` i deklaracja `using` służą dokładnie do tego samego celu – wprowadzenia krótkiej nazwy dla złożonego typu.
- Definicja typu za pomocą `typedef`:

```
typedef std::unique_ptr<std::unordered_map<
    std::string, std::string>> UPtrMapSS;
```

- Deklaracja aliasu za pomocą `using`:

```
using UPtrMapSS =
    std::unique_ptr<std::unordered_map<
    std::string, std::string>>;
```

- Deklaracje aliasów (za pomocą `using`) mogą być używane w formie szablonów (w takim przypadku są nazywane szablonami aliasów), a definicje typów (za pomocą `typedef`) nie:

```
template<typename T>
using UMapST =
    std::unordered_map<std::string, T>>;
```

Aliasy zamiast definiowania typów

- Przykład:

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;
…
MyAllocList<Widget> lw;
…
template<typename T>
class Widgets {
private:
    MyAllocList<T> list;
    …
};
```

Deklaracje typu z auto

- Deklaracje zmiennych z użyciem słowa kluczowego `auto` umożliwiają automatyczną dedukcję typu zmiennej przez kompilator (`auto` jest słowem kluczowym, które w C++11 otrzymało nowe znaczenie).
- Słowem kluczowym `auto` można zadeklarować zmienną albo obiekt bez jawnego określania jego typu, który będzie wydedukowany na podstawie inicjajizatora.
- Przykłady:

```
auto i = 42; // i jest typu int
double f();
auto d = f(); // d jest typu double
```

Deklaracje typu z auto

- Definiując zmienną z użyciem `auto` można dodawać modyfikatory `const`, `volatile` czy `static` oraz stosować referencje lub wskaźniki.
- Stosowanie `auto` przydaje się przede wszystkim tam, gdzie typ zmiennej jest skomplikowanym albo nadmiernie długim wyrażeniem.
- Przykład:

```
vector<string> v;
// pos jest typu vector<string>::iterator
auto pos = v.begin();
// lam jest typu lambda z argumentem int
// i wartością zwracaną bool
auto lam = [] (int x) -> bool { … };
```

Jednolita inicjalizacja

- Celem jednolitej inicjalizacji w C++ jest jeden uniwersalny sposób inicjalizacji danych, włącznie z kontenerami.
- Składnia inicjalizacji z wykorzystaniem nawiasów klamrowych jest teraz dozwolona we wszystkich przypadkach.
- Przykłady:

```
int var1 {5};
const int con2 {10};
int arr3[] {1, 2, var1, var1 + con1};
const Point pt4 {10, 20};
std::complex<double> c5 {4.0, 2.0};
std::vector<std::string> cities6 = {
    "Wroclaw",
    "Cracow"
};
```

Jednolita inicjalizacja

- Jednolita inicjalizacja zabrania inicjalizacji z wykorzystaniem zawężających konwersji, za wyjątkiem niejawnej konwersji, która zawęża typ.
- Przykład:

```
int arr1[] = { 1, 2, 4.5 };
// OK in C++98; error in C++11
```

- Jednolita inicjalizacja wymusza inicjalizację wartością (ang. value initialization), co oznacza, że nawet lokalne zmienne typów podstawowych, które posiadają niezdefiniowaną wartość początkową, są inicjalizowane zerem (albo wartością `nullptr` w przypadku wskaźników).
- Przykłady:

```
double d{}; // d jest równe 0.0
char *ptr{}; // ptr jest równe nullptr
int tab[4]{}; // tab jest równe [0, 0, 0, 0]
```

Jednolita inicjalizacja

- Inicjalizacja agregatów za pomocą klamer przebiega dokładnie w taki sam sposób jak w C++98 i powoduje inicjalizację składowych według kolejności ich definicji w agregacie; liczba elementów na liście musi być równa lub mniejsza od ilości elementów w agregacie.
- Jednolita inicjalizacja klas czy struktur nie będących agregatami powoduje wywołanie konstruktora (z ilością i typem parametrów pasujących do konstruktora).

Listy inicjalizacyjne

- Aby umożliwić inicjalizację kontenerów standardowych za pomocą składni z nawiasami klamrowymi C++11 wprowadzono nowy typ `std::initializer_list<>`.
- Typ `std::initializer_list<>` może być również wykorzystany do inicjalizacji obiektu typu zdefiniowanego przez użytkownika.
- Klasa `std::initializer_list<>` jest zdefiniowana w pliku `<initializer_list>`.
- Obiekt `std::initializer_list<>` przechowuje elementy listy inicjalizującej w niemutowalnej tablicy i implementuje ograniczony interfejs umożliwiający dostęp do elementów przy pomocy iteratorów.

Listy inicjalizacyjne

- Jeśli klasa posiada wiele wersji konstruktora, przy wywołaniu konstruktora poprzez nawiasy klamrowe preferowany jest konstruktor z `std::initializer_list<>`.
- Przykład:

```
class Gadget {
public:
    Gadget(int, int);
    Gadget(int, std::string);
    Gadget(std::initializer_list<int>);
// …
};
// …
Gadget g1(77, "a"); // calls Gadget::Gadget(int, string)
Gadget g2{77, "a"}; // calls Gadget::Gadget(int, string)
Gadget g3={77, "a"}; // calls Gadget::Gadget(int, string)
Gadget g4(33, 22);
// calls Gadget::Gadget(int, int)
Gadget g5{33, 22};
// calls Gadget::Gadget(initializer_list)
Gadget g6={33, 22};
// calls Gadget::Gadget(initializer_list)
```

Jednolita inicjalizacja a listy inicjalizacyjne

- Kiedy typ definiuje zarówno konstruktory dla listy inicjalizacyjnej, jak i konstruktor z powtórzonymi parametrami tego samego typu, to przy wywołaniu preferowana jest wersja z listą inicjalizacyjną.
- Przykład:

```
class P {
public:
    P(int,int);
    P(std::initializer_list<int>);
    //...
};
P p(77,5); // wywołanie P::P(int, int)
P q{77,5}; // wywołanie P::P(initializer_list)
P r{77,5,42}; // wywołanie P::P(initializer_list)
P s={77,5}; // wywołanie P::P(initializer_list)
```

Ograniczanie zasięgu zmiennej w instrukcji warunkowej if-else

- Dobrym zwyczajem w programowaniu jest maksymalne ograniczanie zasięgu zmiennej. Jednak czasami trzeba najpierw otrzymać wartość, która będzie mogła być dalej przetwarzana tylko po spełnieniu określonego warunku.
- Przykład:

```
if (auto itr (char_map.find(c)); itr != char_map.end()) {
    // itr jest poprawny i zostanie wykorzystany do pewnych operacji
}
else {
    // itr jest iteratorem końcowym i nie wolno z niego korzystać
}
```

- Zmiennej lokalnej zadeklarowanej w tuż przed warunkiem w instrukcji warunkowej można potem używać w dalszych zagnieżdżonych albo kaskadowo połączonych instrukcjach warunkowych.

Ograniczanie zasięgu zmiennej w instrukcji wyboru switch-case

- Instrukcje if-else i switch-case wraz z inicjalizatorami to lukier syntaktyczny (osiągnięcie takiego efektu jest możliwe po ujęciu kodu w dodatkowy nawias klamrowy).
- Przykład:

```
switch (char c (getchar()); c) {
case 'a': move_left(); break;
case 's': move_back(); break;
case 'w': move_fwd(); break;
case 'd': move_right(); break;
case 'q': quit_game(); break;
//…
}
```

- Krótszy cykl życiowy zmniejsza liczbę zmiennych w zasięgu, co z kolei przekłada się na większą przejrzystość kodu i jego łatwiejszą refaktoryzację.

Pętle zakresowe

- C++11 wprowadza nową postać pętli `for`, iterującej po wszystkich elementach podanego zakresu (tablicy bądź kolekcji) – pętla ta jest często nazywana pętlą foreach.
- Składnia zakresowej pętli for:
  for (element: kolekcja) {
  instrukcje
  }
  gdzie element to deklaracja pomocniczej zmiennej w pętli tego samego typu co elementy w kolecji. – w każdej iteracji pomocnicza zmienna jest inicjalizowana kolejną wartością z kolecji.
- Przykład:

```
for (int x: {2, 3, 4, 5, 7, 8, 9})
    std::cout << x << std::endl;
```

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
  ```

Wiązania strukturalne

- Standard C++17 oferuje mechanizm **strukturalnego wiązania**. Pomaga ono w przypisywaniu poszczególnym zmiennym wartości pochodzących z par, krotek i struktur.
- W innych językach programowania to zadanie nosi nazwę **rozpakowywania**.
- Przykład:

```
auto [fraction, remainder] =
divide_remainder(16, 3);
std::cout << "16 / 3 to "
<< fraction << " plus reszta z dzielenia "
<< remainder << endl;
```

decltype

- Instrukcja decltype pozwala na określenie przez kompilator typu wyrażenia:
  decltype(wyrażenie)
- Jednym z zastosowań wyrażenia decltype jest deklarowanie typów wartości zwracanych przez funkcje; stosuje się je też w metaprogramowaniu.
- Przykład:

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container &&c, Index i)
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

Nadpisywanie metod wirtualnych

- Wśród podstawowych filarów programowania obiektowego znajduje się polimorfizm, czyli implementacje funkcji wirtualnych w klasach dziedziczących poprzez nadpisanie implementacji swoich odpowiedników z klasy bazowej.
- Przykład:

```
class Base {
public:
    virtual void doWork(); // funkcja wirtualna
                           // klasy bazowej
    …
};
class Derived: public Base {
public:
    virtual void doWork(); // nadpisuje funkcję
                           // Base::doWork
    …                      // (słowo "virtual"
};                         // jest opcjonalne)
std::unique_ptr<Base> upb =
    std::make_unique<Derived>();

…// użycie!!!
```

Nadpisywanie metod wirtualnych

- Aby nadpisanie miało miejsce, muszą być spełnione pewne warunki:
  - Funkcja klasy bazowej musi być wirtualna.
  - Nazwy funkcji bazowej i potomnej muszą być identyczne (z wyjątkiem destruktorów).
  - Typy parametrów funkcji bazowej i potomnej muszą być identyczne.
  - Stałość (const) funkcji bazowej i potomnej musi być identyczna.
  - Zwracane typy i specyfikacje wyjątków funkcji bazowej i potomnej muszą być kompatybilne.
  - **Kwalifikatory odwołań** funkcji muszą być identyczne.

Kwalifikatory odwołań

- Kwalifikatory odwołań umożliwiają ograniczenie użycia funkcji składowych tylko do l-wartości lub tylko do r-wartości.
- Funkcje składowe nie muszą być wirtualne, aby korzystać z kwalifikatorów odwołań.
- Przykład:

```
class Widget {
public:
…
void doWork() &; // ta wersja funkcji doWork
                 // jest stosowana tylko wtedy,
                 // gdy *this to l-wartość
void doWork() &&; // ta wersja funkcji doWork jest
}; // stosowana tylko wtedy, gdy *this to r-wartość
```

Nowa składnia deklaracji funkcji

- Niekiedy typ wartości zwracanej z funkcji jest zależny od typu wyrażenia zastosowanego wobec argumentów wywołania funkcji. W C++11 udostępniono więc alternatywną składnię deklarowania typu wartości zwracanej przez funkcję, za listą parametrów:
  auto fun(params) -> typ;
- Przykład:

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x + y);
```

- Taka sama składnia pozwala deklarować typ wartości zwracanej w lambdach.

Operatory porównań

- Standardowe algorytmy, takie jak `std::sort`, a także kontenery, takie jak `std::set`, oczekują, że `operator<` zostanie domyślnie zdefiniowany dla typów dostarczonych przez użytkownika oraz oczekują, że zaimplementuje on ścisłą kolejność (a zatem spełnia wymagania porównania).
- Idiomatycznym sposobem na zaimplementowanie ścisłego uporządkowania struktury jest użycie porównania leksykograficznego dostarczonego przez `std::tie`.
- Szablon `std:tie<>` tworzy tuplę referencji do l-wartości będących argumentami wywołania, na przykład:

```
std::set<S> set_of_s; // S is LessThanComparable
S value{42, "Test", 3.14};
std::set<S>::iterator iter;
bool inserted;
// unpacks the return value of insert into iter and inserted
std::tie(iter, inserted) = set_of_s.insert(value);
```

Operatory porównań

- Przykład:

```
struct Record
{
   std::string name;
   unsigned int floor;
   double weight;

   friend bool operator<(const Record& l, const Record& r) {
      return std::tie(l.name, l.floor, l.weight)
          < std::tie(r.name, r.floor, r.weight); // keep the same order
   }
};
```

Trójstanowy operator porównania <=>

- Trójstanowy operator porównania w C++20 (ang. three-way comparison operator), zwany również operatorem statku kosmicznego (ang. spaceship operator) ze względu na swój kształt przypominający latający talerz <=>, ma za zadanie porównać dwa obiekty i rozstrzygnąć jaka relacja między nimi zachodzi (mniejszy, równy, większy).
- Operator <=> zwraca obiekt, który można porównywać z 0. Dla wyrażenia `lhs <=> rhs` dostaniemy obiekt, który:
  - jest <0, gdy `lhs < rhs`,
  - jest >0, gdy `lhs > rhs`,
  - jest ==0, gdy `lhs == rhs`.

Trójstanowy operator porównania <=>

- Operator porównania <=> jest uogólnieniem wszystkich innych operatorów porównania (dla zbiorów z porządkiem liniowym): >, >=, ==, <=, <.
- Przykład:

```
if (auto ans = v1 <=> v2; ans < 0) {
    // v1 < v2
}
else if (ans == 0) {
    // v1 == v2
}
else { // ans > 0
    // v1 > v2
}
```

Domyślny operator porównania <=>

- Domyślne przeciążenie operatora <=> umożliwi również porównanie typu za pomocą operatorów <, <=, > i >=.
- Jeśli operator <=> ma wartość domyślną, a operator == w ogóle nie jest zadeklarowany, to operator == jest wygenerowany domyślnie.
- Przykład:

```
struct Record {
    std::string name;
    unsigned int floor;
    double weight;
    auto operator<=>(const Record&) const = default;
};
// Obiekty typu Record mogą być teraz porównywane
// za pomocą ==, !=, <, <=, > i >=.
```