C++17 i STL

Nowe elementy w składni języka

Nowe standardy

- Język C++ zyskał wiele usprawnień wraz z wydaniem standardów C++11, C++14 i C++17. Obecnie jest to zupełnie inny język w porównaniu do tego, którym był jeszcze dekadę temu.
- Standard C++ nie dotyczy jedynie języka, który musi być obsługiwany przez kompilatory, ale również standardową bibliotekę szablonów STL.

Wskaźnik pusty nullptr zamiast 0

- Literał 0 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**.
- Typ std::nullptr_t niejawnie konwertuje się na wszystkie typy wskaźników surowych, a to sprawia, że nullptr działa tak, jakby był wskaźnikiem wszystkich typów.

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)");
```

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 mogą być używane w formie szablonów (w takim przypadku są nazywane szablonami aliasów), a definicje typedef nie.

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;
    …
};
```

Jednolita inicjalizacja

- Celem jednolitej inicjalizacji w C++ jest jeden uniwersalny sposób inicjalizacji danych, włącznie z kontenerami.
- Jednolita inicjalizacja zabrania inicjalizacji z wykorzystaniem zawężających konwersji, za wyjątkiem niejawnej konwersji, która zawęża typ.
- Składnia inicjalizacji z wykorzystaniem nawiasów klamrowych jest teraz dozwolona we wszystkich przypadkach.
- Przykłady:

```
int var1 {5};
const int con1 {10};
int arr1[] = {1, 2, var1, var1 + con1};
const Point pt1 {10, 20};
std::complex<double> c {4.0, 2.0};
std::vector<std::string> cities = {
    "Wroclaw",
    "Cracow"
};
```

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.
- Przykłady:

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

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)
```

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
  ```

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).

- Definiując zmienną z użyciem auto można dodawać modyfikatory const, volatile oraz stosować referencje lub wskaźniki.

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

- decltype(wyrażenie) zwraca typ danych

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 deklaracja funkcji

- Auto f() -> typ