Wykład 15: Auto. Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

dr inż. Andrzej Stafiniak

Wrocław 2023

Automatyczna dedukcja typu zmiennej (C++11/C++14)

- **Mechanizm automatycznej dedukcji typów** został wprowadzony do języka C++ od standardu C++11, a następnie od standardu C++14 można również korzystać z tego mechanizmu w odniesieniu do typu zwracanego z funkcji.

- Mechanizm ten pozwala na definiowanie zmiennych z użyciem słowa kluczowego `auto`, delegując kompilatorowi odpowiedzialność za dedukcje/wybór właściwego typu danych dla zmiennych.

- Kompilator określa typ podczas inicjalizacji zmiennej (na podstawie typu argumentu podczas przypisania). Dlatego dokonując tylko deklaracji zmiennej z wykorzystaniem słowa kluczowego `auto` otrzymamy błąd kompilacji (niemozliwa automatyczna dedukcja typu, zmienna niezainicjalizowana).

Automatyczna dedukcja typu zmiennej (C++11/C++14)

Poprawne zastosowanie mechanizmu automatycznej dedukcji typu danych z wykorzystaniem słowa kluczowego **auto:**

```
// Od C ++14. Typem zwracanym będzie float
auto mul (float x, float y){
   return x * y;
}

// Od C ++11
// Zmienna x jest typu int
auto x = 2 + 5;

// Zmienna y jest typu float
auto y = mul (2.3f, 7.1f);

// Zmienna text jest typu const char *
auto text = "Ala ma kota ";

// Zmienna ref jest typu const int &
const auto & ref = x;

//Zmienna jest typu std::chrono::time_point<std::chrono::high_resolution_clock>
auto timePoint = std::chrono :: high_resolution_clock::now ();
```

Automatyczna dedukcja typu zmiennej (C++11/C++14)

Poprawne zastosowanie mechanizmu automatycznej dedukcji typu danych z wykorzystaniem słowa kluczowego **auto:**

```
// Od C ++14. Typem zwracanym będzie float
auto mul (float x, float y){
   return x * y;
}

// Od C ++11
// Zmienna x jest typu int
auto x = 2 + 5;

// Zmienna y jest typu float
auto y = mul (2.3f, 7.1f);

// Zmienna text jest typu const char *
auto text = "Ala ma kota ";

// Zmienna ref jest typu const int &
const auto & ref = x;

//Zmienna jest typu std::chrono::time_point<std::chrono::high_resolution_clock>
auto timePoint = std::chrono :: high_resolution_clock::now ();
```

Dobrą praktyka jest niestosowanie słowa kluczowego **auto** w przypadku typów podstawowych (*int*, *float*, *char*, *long*, itp.), a ograniczenie go do dedukcji skomplikowanych typów złożonych.

Ale jeżeli wyłącznym celem stosowania **auto** jest skrócenie nazw typów, to alternatywna metoda może być poznany mechanizm aliasów typów.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

- Tworzenie aplikacji w C/C++ nierozłącznie wiąże się z zastosowaniem wskaźników (idea języków C/C++). Ale używanie wskaźników bywa nieintuicyjne i może prowadzić do błędów. Rezultatem tych błędów mogą być **wycieki pamięci** czy **niezdefiniowane zachowanie**.
- Podstawowym problemem są niejednoznaczne/nieprecyzyjne deklaracje funkcji zwracających wskaźniki. Przykład przedstawiono poniżej:

```
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

- `Config` stanowi typ zmiennej przechowującej konfiguracje aplikacji.
- Funkcja `getConfig()` zwraca wskaźnik na typ `Config`, który przetwarzany następnie jest przez funkcje `processConfig()`.
- Problem powstaje w momencie zwalniania pamięci.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

Patrząc na sygnaturę czy nawet nagłówek funkcji `getConfig()` nie ma możliwości określić czy funkcja alokuje pamięć dla pojedynczego obiektu czy tablicy obiektów (który z operatorów zastosować `delete` czy `delete[]`):

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;

Config * getConfig() {
    return new Config{};
}
```

- Patrząc na sygnaturę czy nawet nagłówek funkcji `getConfig()` nie ma możliwości określić czy funkcja alokuje pamięć dla pojedynczego obiektu czy tablicy obiektów (który z operatorów zastosować `delete` czy `delete[]`):
- Trzeba zobaczyć implementację funkcji (ale ta może być dla nas niedostępna, ponieważ np. jest częścią biblioteki)

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

- Patrząc na sygnaturę czy nawet nagłówek funkcji `getConfig()` nie ma możliwości określić czy funkcja alokuje pamięć dla pojedynczego obiektu czy tablicy obiektów (który z operatorów zastosować `delete` czy `delete[]`):

```
Config * getConfig() {
    return new Config{};
}
```

- Trzeba zobaczyć implementację funkcji (ale ta może być dla nas niedostępna, ponieważ np. jest częścią biblioteki)

```
Config * getConfig() {
    static Config config;
    return &config;
}
```

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

- Patrząc na sygnaturę czy nawet nagłówek funkcji `getConfig()` nie ma możliwości określić czy funkcja alokuje pamięć dla pojedynczego obiektu czy tablicy obiektów (który z operatorów zastosować `delete` czy `delete[]`):

```
Config * getConfig() {
    return new Config{};
}
```

- Trzeba zobaczyć implementację funkcji (ale ta może być dla nas niedostępna, ponieważ np. jest częścią biblioteki)

```
Config * getConfig() {
    static Config config;
    return &config;
}
```

- A co gdyby było tak? Wywołanie operatora `delete` na wskaźniku do zmiennej statycznej kończy się **niezdefiniowanym zachowaniem**.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

- Problem nr 2: Wskaźnik `config` przekazany jest jako argument do funkcji `processConfig()`, a następnie zostaje zwolniona pamięć po nim z użyciem operatora `delete`.
- Co jednak w sytuacji, gdy funkcja `processConfig()` zgłosi **wyjątek**?

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

Ponieważ w powyższym kodzie nie widzimy obsługi wyjątków, starowanie będzie oddawane do funkcji nadrzędnych, co spowoduje pominięcie instrukcji zwolnienia pamięci ze wskaźnika `config`.

- Problem nr 2: Wskaźnik `config` przekazany jest jako argument do funkcji `processConfig()`, a następnie zostaje zwolniona pamięć po nim z użyciem operatora `delete`.
- Co jednak w sytuacji, gdy funkcja `processConfig()` zgłosi **wyjątek**?
- Działanie funkcji `processConfig()` zostanie przerwane, a sterowanie zostanie zwrócone do funkcji wywołującej.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
Config * getConfig();
// Pobranie konfiguracji programu
Config * config = getConfig();
// Przetworzenie konfiguracji
processConfig(config);
// Zwolnienie pamieci: delete czy delete[]?
delete config;
```

- Problem nr 2: Wskaźnik `config` przekazany jest jako argument do funkcji `processConfig()`, a następnie zostaje zwolniona pamięć po nim z użyciem operatora `delete`.
- Co jednak w sytuacji, gdy funkcja `processConfig()` zgłosi **wyjątek**?

Aby zapobiec takiej sytuacji można poprawić kod np. w taki sposób:

```
try {
    // Moze zglosic wyjatek std::runtime_error
    processConfig(config);
    delete config;
} catch (const std::runtime_error &) {
    std::cout << "processConfig() threw an exception
   " << std::endl;
    delete config;
}
```

Do obsługi wyjątków stosujemy bloki instrukcji `try - catch`. W bloku `try` wywołujemy instrukcję, która potencjalnie może wyrzucić wyjątek, a w bloku `catch` przechwytujemy wyjątek spełniający warunek, wykonując instrukcje bloku.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
try {
    // Moze zglosic wyjatek std::runtime_error
    processConfig(config);
    delete config;
} catch (const std::runtime_error &) {
    std::cout << "processConfig() threw an exception
   " << std::endl;
    delete config;
}
```

Trochę o wyjątkach:

wyjątek związany jest z nietypową sytuacją - błędem, a sam wyjątek w zamyśle ma być komunikatem co się stało.

Do wyrzucenia wyjątku korzystamy z instrukcji **`throw`**, a jako wyjątek wyrzucić możemy cokolwiek jednak zaleca się stosowanie ustandaryzowanych nazw typów zdefiniowanych w nagłówku `<stdexcept>`

Np.:

std::logic_error	- błąd ogólny - naruszono założenie
std::runtime_error	- błąd ogólny - przekroczony czas wykonania
std::invalid_argument	- przekazano nieprawidłowy argument
std::length_error	- przekroczenie maksymalnego dozwolonego rozmiaru czegoś

……

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
void printConfig(Config * config) {
    if (config != nullptr) {
        // Dereferencja wskaznika config
        std::cout << *config << std::endl;
    }
}
```

- Problem nr 3: związany jest z weryfikacją wartości wskaźnika.
- Funkcja `printConfig()` przyjmuje wskaźnik na typ `Config`, sprawdza czy jego wartość jest różna od `nullptr` i w przypadku pozytywnej weryfikacji wypisuje na ekranie konfiguracje aplikacji. – *niby nie ma zagrożenia*.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
void printConfig(Config * config) {
    if (config != nullptr) {
        // Dereferencja wskaznika config
        std::cout << *config << std::endl;
    }
}
try {
    // Moze zglosic wyjatek std::runtime_error
    processConfig(config);
    delete config;
} catch (const std::runtime_error &) {
    std::cout << "processConfig() threw an exception
   " << std::endl;
    delete config;
}


   // config != nullptr;
   printConfig(config);
```

- Problem nr 3: związany jest z weryfikacją wartości wskaźnika.
- Funkcja `printConfig()` przyjmuje wskaźnik na typ `Config`, sprawdza czy jego wartość jest różna od `nullptr` i w przypadku pozytywnej weryfikacji wypisuje na ekranie konfiguracje aplikacji.
- Jednak gdy wywołamy funkcje `printConfig()` po zwolnieniu pamięci wskazywanej przez `config`, tu pojawia się problem.
- Jaki ? Przecież mamy zabezpieczenia. `if (config != nullptr)`

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

```
void printConfig(Config * config) {
    if (config != nullptr) {
        // Dereferencja wskaznika config
        std::cout << *config << std::endl;
    }
}
try {
    // Moze zglosic wyjatek std::runtime_error
    processConfig(config);
    delete config;
} catch (const std::runtime_error &) {
    std::cout << "processConfig() threw an exception
   " << std::endl;
    delete config;
}


   // config != nullptr;
   printConfig(config);
```

- Tak jak już było wspomniane operator `delete`, tak jak funkcja `free()` nie zeruje, ani tego co wskazuje wskaźnik ani samego wskaźnika.
- W efekcie, wskaźnik przejdzie pozytywną weryfikację (różną od nullptr) i wykonana zostanie na nim dereferencja / wyłuskanie.
- A odwołanie się (dereferencja) do zwolnionego obszaru pamięci skutkuje **niezdefiniowanym zachowaniem**.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

- Kolejna hipotetyczna sytuacja:
- Najczęściej korzystamy ze wskaźnika w celu uchwycenia zalakowanej dynamicznie pamięci.
- Wskaźnik jest zmienną lokalną odkładaną na **stosie**.
- Pamięć przydzielana dynamicznie odkładana jest na **stercie**.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

- Kolejna hipotetyczna sytuacja:
- Najczęściej korzystamy ze wskaźnika w celu uchwycenia zalakowanej dynamicznie pamięci.
- Wskaźnik jest zmienną lokalną odkładaną na **stosie**.
- Pamięć przydzielana dynamicznie odkładana jest na **stercie**.
- Gdy sterowanie wychodzi z bloku, w którym zadeklarowany jest wskaźnik, **automatycznie** czyszczona jest po nim pamięć na **stosie**.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

- Kolejna hipotetyczna sytuacja:
- Najczęściej korzystamy ze wskaźnika w celu uchwycenia zalakowanej dynamicznie pamięci.
- Wskaźnik jest zmienną lokalną odkładaną na **stosie**.
- Pamięć przydzielana dynamicznie odkładana jest na **stercie**.
- Gdy sterowanie wychodzi z bloku, w którym zadeklarowany jest wskaźnik, **automatycznie** czyszczona jest po nim pamięć na **stosie**.
- W takiej sytuacji tracimy uchwyt (nasz wskaźnik), tracimy możliwość zwolnienia dynamicznie przydzielonej pamięci. Dochodzi do tzw. **wycieku pamięci**.

Wskaźniki, niezdefiniowane zachowanie, wycieki pamięci

- **Wyciek pamięci** - Jak temu przeciwdziałać:
- Rozwiązaniem jest opakowanie „surowego” wskaźnika (ang. raw pointer) w obiekt automatycznie zwalniający pamięć podczas destrukcji. Oznacza to przekazanie zarzadzania pamięcią alokowana na stercie do obiektu alokowanego na stosie.
- Taki mechanizm został wprowadzony od standardu C++11 pod nazwą **inteligentne wskaźniki** (ang. smart pointers) i stanowi jeden z ważniejszych elementów nowoczesnego **programowania obiektowego**.

Kolokwium, kolokwium, kolokwium

Wy1	Wprowadzenie do przedmiotu. Standardy języka C i C++. Cykl budowania oprogramowania. Typy danych i typy zmiennych
Wy2	Podstawowe operatory i wyrażenia. Instrukcje sterujące
Wy3	Łańcuchy znakowe
Wy4	Tablice jedno i wielowymiarowe. Wskaźniki. Działania na wskaźnikach
Wy5	Obsługa standardowego wejścia i wyjścia
Wy6	Obsługa plików
Wy7	Przydział i zarządzanie pamięcią
Wy8	Funkcje. Wskaźniki funkcyjne
Wy9	Rekurencja
Wy10	Typy złożone: struktura, unia
Wy11	Struktury danych
Wy12	Algorytmy i złożoność obliczeniowa
Wy13	Statyczne i dynamiczne biblioteki programistyczne
Wy14	Wyjątki, wskaźniki inteligentne, niezdefiniowane zachowanie, wycieki pamięci

Wyjątki tylko w C++

- Wyjątki występują w „wyjątkowych” sytuacjach, sytuacje standardowe są powszechne, a program przetwarzany jest liniowe zgodnie z kolejnością instrukcji.

- Wyjątek związany jest z nietypową sytuacją - błędem, a sam wyjątek jest komunikatem co się stało. Do wyrzucenia wyjątku korzystamy z instrukcji **`throw`**, a jako wyjątek wyrzucić możemy cokolwiek jednak zaleca się stosowanie ustandaryzowanych nazw typów zdefiniowanych w nagłówku `<stdexcept>`

-