Laboratorium VII

Wstęp

Dzisiejsze zajęcia dotyczyły będą wyjątków. Stanowią one mechanizm, za pomocą którego możemy w C++ obsługiwać niekrytyczne błędy programu. Przykłady takich błędów to: - niepowodzenie alokacji pamięci (za mało RAMu) - brak połączenia sieciowego - plik, który próbujemy otworzyć nie istnieje

Są to sytuacje, które jesteśmy w stanie z góry przewidzieć i po wystąpieniu których chcemy potencjalnie mieć możliwość odzyskania normalnego trybu pracy programu.

Przypomnijmy, jak radziliśmy sobie z tego typu błędami w języku C. Powszechnie przyjęte podejście polega na zwracaniu z funkcji kodów błędu. Innymi słowy, zwracamy z funkcji, których działanie może się nie powieść, liczbę całkowitą. Na podstawie wartości tej liczby stwierdzamy następnie, czy (i jaki) błąd nastąpił. Podejście to ma następujące wady:

- tracimy możliwość zwracania z funkcji wyniku, musimy robić to przez argument będący wskaźnikiem
- musimy ręcznie propagować kody błędów przez stos wywołań funkcji (jeżeli funkcja A woła B, B woła C, itd., to jeżeli funkcja Z może zwrócić błąd, jesteśmy zmuszeni ręcznie propagować jego wartość przez wszystkie funkcje A-Y)
- obsługiwanie wyjątków zakłóca nieco czytelność kodu - zamiast wołać funkcję i pracować z wynikiem jej wywołania, musimy w samym środku kodu umieścić blok sprawdzający wartość kodu błędu

Łapanie wyjątków

W C++ mamy do dyspozycji wyjątki. Korzystamy z nich w następujący sposób:

```
try
{
    // Tutaj potencjalnie problematyczne instrukcje
}
catch(const T1& wyjatek)
{
    // Tutaj instrukcje obsługujące sytuację, w której rzucony zostanie
}
catch(const T2& wyjatek)
{
    // Tutaj instrukcje obsługujące sytuację, w której rzucony zostanie wyjąte
}
// Obsłuż inne typy wyjątków...
catch(...)
{
    // Tutaj instrukcje obsługujące sytuację, w której rzucony zostanie wyjąte
    // typu innego niż obsłużone powyżej (T1, T2, itd.)
}
```

Zanim powiemy o tym jak rzucać wyjątki i jakie mogą mieć typy, zobaczmy konkretny przykład sytuacji, w której wyjątek jest rzucany przez bibliotekę standardową.

Zadanie 1 Napisz program, który wczytuje z klawiatury liczbę całkowitą. Następnie w bloku `try` stwórz wektor wypełniony zerami o długości podanej z klawiatury. W bloku `catch(...)` wyświetl wiadomość informującą, że został rzucony wyjątek. Sprawdź co stanie się, gdy podasz małą liczbę, a co gdy podasz liczbę przekraczającą pamięć RAM dostępną na Twoim komputerze.

Rozróżnianie wyjątków, `std::exception`

W zadaniu 1, blok `catch(...)` pozwolił nam na wykrycie, że *pewien* wyjątek został rzucony. Często chcemy jednak, aby sam wyjątek niósł ze sobą jakąś informację. Mamy możliwość łapania *konkretnych typów* wyjątków. Na przykład, klasy i funkcje biblioteki standardowej w przypadku nieudanej alokacji pamięci rzucają wyjątek typu `std::bad_alloc`.

Zadanie 2 Wykonaj zadanie 1, tym razem łapiąc konkretny wyjątek typu `std::bad_alloc`. Wyświetl do konsoli wiadomość zawartą w tym wyjątku (użyj metody `what`).

Może się zdarzyć, że w bloku `try` mogą zostać rzucone różne typy wyjątków. W zależności od tego, jaki błąd wystąpił, mamy wtedy możliwość wykonania innego zestawu instrukcji. Zobaczmy to na przykładzie `std::bad_alloc` i `std::bad_variant_access`. `std::bad_variant_access` jest typem wyjątku, który rzucany jest gdy próbujemy dostać się do wariantu poprzez typ, którego obecnie nie trzyma.

Zadanie 3 Napisz program, który wczytuje z klawiatury liczby całkowite `a` i `b`. Następnie w bloku `try` stwórz wektor wypełniony zerami o długości `a` oraz wariant typu `std::variant<int, std::string>`. Jeżeli `b` jest parzyste, przypisz do wariantu wartość 42, a jeżeli nie, wartość `"nieparzyste"`. Spróbuj wyświetlić wartość trzymaną przez wariant jako `int` (`std::cout << std::get<int>(v)`). Napisz 2 bloki `catch`, jeden łapiący `std::bad_alloc`, jeden łapiący `std::bad_variant_access`, w których wyświetlisz wiadomość o rzuconym wyjątku. Zbadaj zachowanie programu w zależności od podanych z klawiatury zmiennych.

Wyjątki rzucane przez bibliotekę standardową są polimorficznymi typami, które dziedziczą po klasie `std::exception` i nadpisują jej wirtualną metodę `what` (vide dokumentacja). W przypadku, gdy nie zależy nam na innym zestawie instrukcji dla różnych typów wyjątków, a jedynie informacji jaki typ wyjątku został rzucony, możemy to wykorzystać, pisząc tylko jeden blok `catch` łapiący klasę bazową.

Zadanie 4 Wykonaj ponownie zadanie 3, tym razem pisząc tylko jeden blok `catch`, łapiący wyjątek typu `std::exception`. Wyświetl w nim informację zwróconą przez metodę `what`. Zbadaj zachowanie programu w zależności od podanych z klawiatury zmiennych.

W przypadku, gdy istnieje więcej niż 1 blok `catch`, wyjątki dopasowywane są do pierwszego bloku (w kolejności, w której występuje w kodzie), do którego jest to możliwe. W związku z tym, jeżeli wyjątki są polimorficzne, przeważnie najlepiej umieścić klasy bazowe na końcu, aby obsłużyć wyjątek w możliwie jak najbardziej "wyspecjalizowany" sposób. W praktyce, pisząc bloki `catch` powinniśmy trzymać się następującej hierarchii:

1. Wyjątki konkretnych typów, zdefiniowanych przez nas lub biblioteki, z których korzystamy
2. Wyjątki STL klas pochodnych (np. `std::bad_alloc` i `std::bad_variant_access`
3. `std::exception`
4. Blok `catch(...)`

Rzucanie wyjątków

Oczywiście możliwość rzucania wyjątków ma nie tylko biblioteka standardowa, ale także my. Wyjątki rzucamy za pomocą słowa kluczowego `throw`, np.:

```
void foo(const T& argument)
{
    if(!warunek(argument))
    {
        T_wyjatek wyjatek{/* ... */};
        throw wyjatek;
    }
    // Teraz przystępujemy do pracy z argumentem mając pewność, że spełnia pos
}
```

Zwróćmy uwagę, że rzucenie wyjątku przerywa funkcję i przechodzi to "najbliższego" bloku `catch`. W związku z tym w powyższym kawałku kodu nie jest potrzebny blok `else`.

Zadanie 5 Napisz funkcję `podziel`, która przyjmuje dwie liczby całkowite i zwraca ich iloraz. Jeżeli podany mianownik jest równy zero, niech funkcja rzuca wyjątek typu `int` o wartości 0. W funckji `main` wywołaj `podziel` w blokach `try-catch`

Rozwijanie stosu (*stack unwinding*) i RAII

Po rzuceniu wyjątku, a przed wejściem do bloku `catch`, następuje bardzo ważny etap: rozwijanie stosu. Oznacza to, że wołane są destruktory wszystkich zmiennych ze scope'ów poprzedzających blok `catch`. Jeżeli mamy stos wywołań (*call stack*) funkcji A-Z (funkcja A woła B, B woła C, itd.), to możemy śmiało rzucić wyjątek w funkcji Z i złapać go w funkcji A. Funkcje B-Y nie muszą nic wiedzieć o możliwości zaistnienia wyjątku. Dzięki rozwijaniu stosu, stworzone dotychczas zmienne w B-Y zostaną automatycznie i poprawnie zniszczone, co pozwoli nam uniknąć wycieku zasobów i innych nieprzyjemnych sytuacji. Jest to kolejny argument za stosowaniem podejścia RAII.

Zadanie 6 Napisz klasę `Informer`, która drukuje informację o konstrukcji i destrukcji jej obiektów (możesz użyć tej napisanej na wcześniejszych zajęciach). Wykonaj dowolne z wcześniejszych zadań zawartych w tej instruckji, poprzedzając miejsce w którym może zostać rzucony wyjątek stworzeniem obiektu typu `Informer`. Zweryfikuj, czy jest on niszczony zgodnie z oczekiwaniami.