Rafał Mielniczuk
28.03.2022 r.

Zarządzanie pamięcią w C++

Cele:

Opanowanie praktyk dobrego zarządzania pamięcią w C++ z przykładowymi ćwiczeniami.

Wstęp:

Głównym zagadnieniem wykładu było zaprezentowanie wyższości nowego standardu (począwszy od C++11) zarządzania pamięcią, skuteczne sposoby wykorzystania go oraz analiza ich działania, aby uniknąć błędów w ich implementacji. Ponadto zachęcenie do stosowania metodyki RAII i w miarę możliwości wykorzystywania stosu zamiast sterty.

Valgrind:

Narzędzie programistyczne do sprawdzania nieprawidłowości w zarządzaniu pamięcią. Może skutecznie wskazać miejsce wycieku danych oraz ich rozmiar. Dodatkowo pokazuje pozwala zlokalizować pomniejsze błędy jak brak zamknięcia FILE.

Proste wskaźniki:

Mogą prowadzić do wielu błędów w kodzie i wymagają specjalnej uwagi, aby nie doprowadzić do potencjalnych wycieków. Są pozostałością C i w nowym standardzie powinny być używane tylko w ostateczności.

Unique_ptr:

- **Główny następca prostych wskaźników. Pozwala na beztroską implementację bez obaw o wycieki pamięci. Unique pointer w momencie opuszczania stosu dopilnowuje, aby obiekt, na który wskazuje, został usunięty.**
- **Może być tylko jeden unique pointer (stąd nazwa) dla danego obiektu, tym samym dopuszcza się tylko jego przenoszenie (std::move()), bez możliwości kopiowania.**
- **Może stosować własny deleter, który zwiększa jego elastyczność.**
- **Obawy związane z jego szybkością bądź rozmiar w porównaniu do prostego wskaźnika są niewskazane, ponieważ ma on praktycznie identyczne parametry.**

Shared_ptr:

- **Jest odpowiednikiem unique pointer z możliwością jednoczesnego udzielania dostępu do zasobu wielu „użytkownikom”. Liczbę przydzielonych referencji utrzymuje w bloku kontrolnym i w momencie, gdy spadnie ona do zera, uwalnia zasób.**
- **Może stosować własny deleter, który zwiększa jego elastyczność.**
- **Trzeba zaznaczyć, że w miarę możliwości należy stosować unique pointer ze względu na jego nadrzędną wydajność. Shared pointer w kwestiach wydajnościowych potrzebuje dodatkowych pamięci i alokacji na blok kontrolny (nie dotyczy gdy make_shared).**

Weak_ptr:

- **Jest wskaźnikiem pomocniczym dla shared pointera. Służy do obserwacji wskazywanego obiektu, ale nie jest „współwłaścicielem” obiektu jak shared pointer.**
- **Nic nie stoi na przeszkodzie, aby zasób został zwolniony, gdy obserwują go weak pointery. Jest to użyteczne, aby móc stwierdzić, że zasób na pewno został zwolniony.**

Make_ptr:

- **Metody std::make_unique oraz std::make_shared są bardzo dobrą praktyką. Pozwalają wyeliminować potencjalne błędy związane z użyciem operatora new. Np. przekazanie obiektu przez new do konstruktora i rzucenie przez niego wyjątku może doprowadzić do wycieku pamięci (destruktor nie zostanie zawołany). Natomiast make pozwala na kontrolę tworzonego obiektu przez wskaźnik.**
- **W przypadku make_shared zaoszczędza pamięć. Zamiast tworzenia oddzielnie w pamięci bloku kontrolnego, przyłącza go do tworzonego obiektu.**

Optional:

- **Służy do reprezentowania potencjalnie nullowych wartości. Spełnia to zadanie lepiej niż dotychczasowy null_ptr ze względu na obecność na stosie oraz kolejność działania.**
- **Ustawiając wskaźnik na pustą przestrzeń adresową najpierw go alokujemy i następnie stwierdzamy brak wartości, natomiast std::optional pozwala na zweryfikowanie wartości zasobu i na podstawie tej informacji stworzenie go.**

Konstruktory i dekstruktory:

- **Prawidłowe użycie mechanizmu polimorfizmu wymaga od nas implementacji destruktora wirtualnego.**
- **Istniejąca w obecnym standardzie zasada pięciu (The Rule of Five) wymaga od nas własnej implementacji każdej metody w przypadku zmiany chociaż jednej z nich, mianowicie: destruktor, konstruktor kopiujący, kopiujący operator przypisania, konstruktor przenoszący, przenoszący operator przypisania.**
- **Dopełniając tego obowiązku możemy również posłużyć flagami =default i =delete, które oddelegowują odpowiednie żądanie do kompilatora.**
- **Metodyka RAII pozwala na posługiwanie się zasadą zeru. Jej stosowanie skutecznie eliminuje potrzebę własnej implementacji którejkolwiek z tych metod.**

Obsługa błędów:

- **Wiedząc, że rzucony wyjątek rozpoczyna procedurę odwijania stosu należy pamiętać, aby wszelkie niezwolnione zasoby specjalnie uwolnić, np. w bloku catch.**
- **Rzucenie wyjątku przez konstruktor odwija stos bez użycia odpowiedniego destruktora, tym samym należy zadbać w konstruktorze o odpowiednie rozliczenie się z alokowanymi zasobami.**

RAII:

Metodyka w nowoczesnym C++ zachęcająca do zarządzania pamięcią na stosie. Jak mówi nazwa (Resource Acquisition Is Initialization), tworzenie zasobów następuje poprzez konstruktory, tym samym wyjście z lokalnego zasięgu woła destruktor danego zasobu. Gwarantuje to poprawność zarządzania na poziomie języka.

Wnioski:

C++ zdecydowanie się zmienił na przestrzeni lat i nie przypomina on starego C++ zwłaszcza sprzed standardu C++11. Przeżytki C zostały zastąpione równie wydajnymi, ale co najważniejsze, bezpieczniejszymi rozwiązaniami. Moim zdaniem największy wpływ na dzisiejszy kod ma metodyka RAII. Rozwiązuje ona wiele problemów przeszłości oraz zachęca do wykorzystywania stosu, który jest o wiele bezpieczniejszy od sterty. Tym samym znikają operatory new i delete. Obecnie widuje się je w znikomym stopniu w nowoczesnym oprogramowaniu. Dodatkowo wzrasta przejrzystość kodu, który stał się krótszy i treściwszy. Łatwiej w nim zrozumieć intencje programisty oraz naprawić potencjalne błędy. Oczywiście obok RAII należy również docenić unique oraz shared pointery, które skutecznie pomagają realizować bezwyciekową perspektywę języka C++.