1 C++ memory managment

1.1 Wskaźniki

Używanie prostych wskaźników jest niebezpieczne ze względu na łatwość wystąpienia wycieku pamięci. Nie tylko przez możliwość zapomnienia o ręcznym zwolnieniu pamięci ale także przez wystąpienie błędów przed zwolnieniem.

Preferowane jest używanie smart pointerów: `std::unique_ptr<T>` - jest jedynym właścicielem danego obiektu, nie można go skopiować a jedynie przenieść; `std::shared_ptr<T>` - może być kopiowany i służy do udostępniania danych dla wielu obiektów, zawiera licznik referencji i zostanie zwolniony gdy ostatni wskaźnik zostanie usunięty. Istnieje także `weak_ptr<T>` który "podczepiaśię pod istniejacy `shared_ptr` i udostępnia możliwość podglądu czy pamięc została zwolniona.

Nadal istnieje możliwość niezwolnienia pamięci z użyciem smart pointerów w momencie gdy w konstruktorze pointera tworzymy wskaźnik operatoren `new` a sam konstruktor używany jest jako argument. Np. `void foo(shared_ptr<int>(new int(5)), boo())` Ze względu na brak specyfikacji kolejności ewaluacji argumentów, najpierw może być utworzony wskaźnik, a następnie może być wywołana funkcja `boo()`, która wyrzuci wyjątek i pamięć nie zostanie zwolniona.

Problem można rozwiązać używając funkcji `std::make_{unique/shared}`, która gwarantuje poprawne utworzenie smart pointera.

1.1.1 make_shared a shared_ptr

Oprócz samego obiektu, `shared_ptr` alokuje także blok kontrolny. W przypadku użycia konstruktora ze wskaźnikiem, blok kontrolny zawiera wskaźnik na ten obiekt, `make_shared` alokuje jeden większy blok zawierające sam obiekt i blok kontrolny. W pierwszym przypadku, możliwa jest dealokacja obiektu przed dealokacją samego bloku kontrolnego, nie jest to możliwe w przypadku drugiem. `make_shared` za to alokuje pamięc tylko raz (nie dwa razy) oraz ze względu na `memory aligment` zajmuje on mniej miejsca.

1.2 Konstruktory

Wyróżnia się 5 różnych rodzajów konstruktorów:

1. Default constructor `Class();`
2. Copy constructor`Class(const Class& other);`
3. Copy assignment `Class& operator=(const Class& other);`
4. Move constructor `Class(Class&& other);`
5. Move assignment `Class& operator=(Class&& other);`

Są one automatycznie generowane przez kompilator, może to jednak prowadzić do błędnych zachowań (shallow copy zamiast deep copy).

1.2.1 Rule of Three

Zakłada, że implementujemy konstruktory kopiujące.

1.2.2 Rule of Five

Zakłada, że implementujemy wszystkie pięć.

1.2.3 Rule of Zero

Przy poprawnym używaniu stosu, stosowaniu smart pointerów, możliwe jest pomięcie używania konstruktorów kopijących i przenoszących.

1.2.4 Automatyczne generowanie konstruktorów

Zależnie od napisanych implementacji konstruktorów, kompilator może nie generować innych konstruktorów. Możemy sprecyzować które konstruktory chcemy wygenerować a które nie przy pomocy `= default` lub `= delete`.