Zarządzanie pamięcią

dr inż. Paweł Kłeczek

2021

Rodzaje alokacji pamięci

Dwie podstawowe metody alokacji pamięci w C++:

- **alokacja statyczna** – ilość pamięci z góry określona na etapie pisania programu (poprzez odpowiednie deklaracje zmiennych); pamięć zwalniana automatycznie – po wyjściu z bloku programu, w którym była deklarowana (z wyj. obiektów `static`) lub po zakończeniu programu

```
int a;
char str[] = "Hello!";
float tab[5];
```

- **alokacja dynamiczna** – program przydziela dowolne ilości pamięci w trakcie pracy w momencie użycia odpowiednich funkcji/operatorów; pamięć musi być jawnie zwolniona (za pomocą funkcji/operatora)

Dynamiczna alokacja pamięci pozwala przede wszystkim na:

- utworzenie tablicy o rozmiarze obliczanym dopiero w trakcie programu
- dostęp do zmiennych utworzonych wewnątrz funkcji po wyjściu z nich

Organizacja pamięci programu

Podstawowe segmenty pamięci:

- **pamięć statyczna** – przechowuje zmienne o statycznym okresie trwania
- **stos** (ang. stack) – przechowuje zmienne o automatycznym okresie trwania; struktura danych typu LIFO (Last-In-First-Out)
  - definiowanie nowej zmiennej automatycznej $\Rightarrow$ operacja *push*
    usuwanie tej zmiennej $\Rightarrow$ operacja *pop*
  - zarządzaniem zajmuje się procesor, a nie programista
  - stos ma ograniczoną, niewielką pojemność
    zbyt wiele zmiennych? $\Rightarrow$ **przepełnienie stosu** (ang. stack overflow)
- **sterta** (ang. heap) – pula pamięci dużych rozmiarów, która może być używana w sposób dynamiczny
  - zarządzaniem zajmuje się „ręcznie” programista
  - do obiektów zaalokowanych na stercie można się odwoływać w dowolnym miejscu w programie
  - jedyne ograniczenie na rozmiar sterty – fizyczny rozmiar pamięci operac.

Zarządzanie pamięcią „surowymi” wskaźnikami

Dwie pary operatorów do dynamicznego alokowania i zwalniania pamięci:

- `new` i `delete` – dla pojedynczych obiektów
- `new[]` i `delete[]` – dla tablic obiektów.

W przybliżeniu:

- [C++] operatory `new` i `new[]` $\approx$ [C] `malloc()`
- [C++] operatory `delete` i `delete[]` $\approx$ [C] `free()`

„Ręczne” zarządzanie pamięcią z użyciem powyższych operatorów jest *bardzo* podatne na błędy!

Dynamiczna alokacja pamięci

operator `new` – wykonuje trzy operacje:

- przydziela pamięć dla alokowanego obiektu
- (opcjonalnie) dokonuje jego inicjalizacji
- zwraca wskaźnik na typ zgodny z typem operandu
  (czyli alokowanego obiektu)

Dynamicznie zaalokowane obiekty są inicjalizowane w sposób domyślny:

- obiekty typów wbudowanych mają niezdefiniowaną wartość
- obiekty klas są inicjalizowane z użyciem właściwego konstruktora
  (w tym potencjalnie domyślnego)

Dynamiczna alokacja pamięci

Przykłady użycia operatora **new**:

```
// inicjalizacja "w domyślny sposób" = brak inicjalizacji
//   (`p` wskazuje na obiekt niezainicjalizowany)
int* p = new int;

// wywołanie konstruktora domyślnego
//   (obiekt wskazywany będzie mieć pusty łańcuch - "")
std::string* ps1 = new std::string;

// wywołanie konstruktora jednoargumentowego
//   (inicjalizacja łańcuchem "X")
std::string* ps2 = new std::string("X");
```

Standard C++03 umożliwił inicjalizację obiektów *każdego* typu (także prostego) z użyciem nawiasów okrągłych – jak w przypadku konstruktorów:

```
int* x = new int();    // `x` wskazuje na obiekt o "wartości
                       //   domyślnej" dla danego typu (tu: 0)
int* y = new int(2);   // `y` wskazuje na obiekt o wartości 2
```

Dynamiczna alokacja pamięci

Standard C++11 umożliwił inicjalizację za pomocą tzw. jednolitej inicjalizacji:

```
// "uniform initialization" => tablica zawiera 10 elementów
//   o wartościach: 1, 2, 3, 0, 0, ...
int* pi = new int[10]{1,2,3};

// "uniform initialization" => kontener zawiera elementy
//   o wartościach 0, 1 i 2
std::vector<int>* pv = new std::vector<int>{0,1,2};
```

Można dynamicznie alokować obiekty o typie z kwalifikatorem `const`:

```
const int *pci = new const int(1024);
```

Aby uniknąć niezdefiniowanych wartości *zawsze* inicjalizuj nowo tworzony obiekt – także w przypadku alokacji dynamicznej.

Zwalnianie dynamicznie przydzielonej pamięci

Aby zapobiec wyciekom pamięci, zwalniaj dynamicznie przydzieloną pamięć za pomocą operatora **`delete`** (jego operandem jest wskaźnik na obiekt, który należy usunąć i zwolnić odpowiadającą mu pamięć).

Wskaźnik przekazany do **`delete`** musi wskazywać na obiekt zaalokowany dynamicznie albo mieć wartość `nullptr`:

```
delete p;  // `p` musi wskazywać na obiekt zaalokowany
           //   dynamicznie albo mieć wartość `nullptr`
```

Aby zwolnić pamięć przydzieloną z użyciem **`new`**`[]` należy użyć **`delete`**`[]`:

```
int* pi = new int[3]{1,2,3};
delete[] pi;
```

Sytuacje prowadzące do niezdefiniowanego zachowania programu:

- próba usunięcia wskaźnika na pamięć nie przydzieloną dynamicznie
- próba dwukrotnego zwolnienia tego samego miejsca w pamięci

Problemy związane z (nieumiejętnym) zarządzaniem pamięcią

Wycieki pamięci

wyciek pamięci (ang. memory leak) – sytuacja, gdy przydzielony dynamicznie blok pamięci nie został zwolniony, a utracona została informacja o położeniu takiego bloku (co uniemożliwia jego zwolnienie)

$\Rightarrow$ taki blok będzie traktowany jako „zajęty”, bez możliwości jego ponownego użycia przez ten sam lub inne procesy

Sytuacje prowadzące do wycieku pamięci:

- brak zapamiętania adresu dynamicznie przydzielonego bloku pamięci

```
new char;
```

- utrata adresu bloku pamięci poprzez nadpisanie

```
char* ptr = new char;
ptr = nullptr;  // utrata adresu
```

Wycieki pamięci

- utrata adresu bloku pamięci poprzez wyjście z bloku

```
void foo(void) {
  char* ptr = new char;
  /* inne instrukcje, ale nie `delete ptr` */
}
```

System operacyjny wyznacza limit pamięci, jaki może wykorzystać dany program – jego przekroczenie powoduje „ubicie” programu.

Zapominanie o zwolnieniu przydzielonej pamięć prowadzi do jej wycieku i może skutkować przedwczesnym zakończeniem programu (przez system operacyjny).

Błędy naruszenia ochrony pamięci

błąd **naruszenia ochrony pamięci** (ang. memory access violation), żarg. **segfault** (od ang. segmentation fault) – zgłaszany przez sprzęt posiadający mechanizm ochrony pamięci, sygnalizuje systemowi operacyjnemu próbę dostępu do „chronionych” obszarów pamięci (np. takich, do których dany proces nie powinien mieć dostępu)

Sytuacje prowadzące do błędów naruszenia ochrony pamięci:

- odwoływanie się do elementów o indeksach spoza zakresu danej tablicy

```
int arr[3];
arr[-1] = 0;   // BŁĄD
```

- próba uzyskania dostępu do przydzielonego bloku pamięci po jego zwolnieniu (w tym próba ponownego zwolnienia tego samego bloku)

```
int* ptr = new int;
delete ptr;
*ptr = 0;     // BŁĄD     [ tu `ptr` = wiszący wskaźnik
delete ptr;   // BŁĄD              (ang. dangling pointer) ]
```

Diagnostyka błędów

Valgrind – darmowe narzędzie do identyfikowania problemów z zarządzaniem pamięcią, m.in.

- sprawdza ile pamięci zostało przydzielone
- sprawdza ile pamięci zostało zwolnione
- wychwytuje próby nieuprawnionego dostępu do w miejsc w pamięci

Po co nam inteligentne wskaźniki?

Najważniejsze błędy przy ręcznym zarządzaniu pamięcią:

- wyciek zasobów
- dostęp przez niewłaściwy wskaźnik
- naruszenie pamięci
- niepewność odnośnie typu wiązania obiektu
- niewłaściwe zwalnianie pamięci

Rozwiązanie: **inteligentne wskaźniki**

Inteligentne wskaźniki

inteligentne wskaźniki (ang. smart pointers) – abstrakcyjne typy danych zachowujące się jak tradycyjne, **„surowe” wskaźniki** (ang. raw pointers), lecz jednocześnie zapewniające dodatkową funkcjonalność, m.in.:

- automatyczne zarządzanie pamięcią

  (tj. automatyczne zwalnianie pamięci użytej do przechowywania wskazywanego obiektu)

- sprawdzanie zakresu

  (tj. czy nie próbujemy uzyskać dostępu do „cudzej” pamięci)

W C++:

- inteligentne wskaźniki są zdefiniowane jako szablony klasy o parametrze będącym typem wskazywanego obiektu
  $\Rightarrow$ możesz utworzyć inteligentny wskaźnik do obiektu dowolnego typu
- inteligentny wskaźnik przechowuje (jako pole szablonu klasy) „surowy” wskaźnik do obiektu wskazywanego
- funkcjonalność „surowego” wskaźnika jest symulowana za pomocą przeciążenia odpowiednich operatorów (m.in. operatora dereferencji $*$)

Inteligentne wskaźniki

własność (ang. ownership) – oznacza to, który fragment kodu odpowiada za zwolnienie przydzielonego zasobu (pamięci, uchwytu do pliku itp.)

Korzystając z „surowych” wskaźników właścicielem zasobu przydzielonej pamięci jest obiekt albo zasięg efektywnie wykonujący operację zwolnienia (za pomocą operatora `delete` albo funkcji `free()`).

inteligentny wskaźnik $\equiv$ obiektowe opakowanie dla „surowego” wskaźnika:

- w konstruktorze otrzymuje „surowy” wskaźnik na dynamicznie zaalokowany obiekt
- w destruktorze zwalnia tę pamięć (co dzięki RAII gwarantuje „szczelność” takiego rozwiązania)

$\Rightarrow$ inteligentny wskaźnik jest właścicielem wskazywanego obiektu, wyręcza nas w zwalnianiu pamięci przydzielonej na przechowanie tego obiektu (sam obiekt inteligentnego wskaźnika może być alokowany statycznie)

Inteligentne wskaźniki

Rodzaje inteligentnych wskaźników w C++11 (zob. `<memory>`):

- `std::unique_ptr`
  Implementuje posiadanie zasobu „na wyłączność” – w danym momencie tylko jeden wskaźnik tego rodzaju może być właścicielem danego obiektu. Gdy wskaźnik ten zostanie zniszczony, obiekt wskazywany zostaje automatycznie zwolniony.
  Stosuj domyślnie właśnie ten rodzaj inteligentnych wskaźników.

- `std::shared_ptr`
  Służy do (współ)dzielenia się zasobem – dowolna liczba inteligentnych wskaźników tego rodzaju może współdzielić obiekt, a współdzielony obiekt zostanie zniszczony dopiero, gdy ostatni inteligentny wskaźnik będący (współ)właścicielem zostanie zniszczony.

- `std::weak_ptr`
  Sam nie „posiada” obiektu – służy do *obserwacji* obiektu zarządzanego przez wskaźniki współdzielone. Pozwala uniknąć cyklicznych zależności.

Szablon klasy `std::unique_ptr`

`std::unique_ptr<T>` – realizuje funkcjonalność inteligentnego wskaźnika na typ `T` „na wyłączność”

Nie można stworzyć kopii instancji `std::unique_ptr`, gdyż:

- konstruktor kopiujący oznaczony jako „`= delete`”
- operator przypisania oznaczony jako „`= delete`”

Szablon klasy `std::unique_ptr`

Przykładowe problemy w przypadku klasycznych, „surowych” wskaźników:

```
Foo *p = new Foo("a useful object");
make_use(p);
```

- Co się stanie ze wskaźnikiem po wywołaniu `make_use()`?
- Czy `make_use()` stworzy kopię wskaźnika, z której później będzie korzystać?
- Czy własność wskaźnika zostanie przeniesiona do funkcji `make_use()`, która dokona zwolnienia pamięci przechowującej wskazywany obiekt, czy też funkcja ta pozostawia kwestię zwolnienia `p` w gestii wywołującego?

$\Rightarrow$ odpowiedź wymaga inspekcji kodu/dokumentacji `make_use()`

Rozwiązanie: `std::unique_ptr<T>` + `std::make_unique<T>()`

Szablon klasy `std::unique_ptr`

`std::make_unique` [C++14] – wewnętrznie dokonuje dynamicznej alokacji i od razu zwraca obiekt inteligentnego wskaźnika `std::unique_ptr`

`std::make_unique()` to funkcja szablonowa, jej wywołanie ma postać

```
std::make_unique<T>(args)
```

- `T` to typ wskazywanego obiektu
- `args` to (opcjonalne) argumenty, które zostaną przekazane do konstruktora obiektu klasy `T`

Przykładowo:

```
// Utwórz unique-pointer na obiekt typu `Foo` utworzony za
//   pomocą wywołania konstruktora jednoargumentowego: Foo(42)
std::unique_ptr<Foo> q = std::make_unique<Foo>(42);
```

`std::unique_ptr` a wskazywane obiekty

Używając `std::unique_ptr` wciąż możesz udostępniać wskazywany zasób za pomocą metody `get()` zwracającej „surowy wskaźnik” na ten zasób:

```
void use_unique_ptrs_resource(int* up) {
  std::cout << *up << std::endl;
}

// ...

auto up = std::make_unique<int>(1);
use_unique_ptrs_resource(up.get());
```

Koncepcja „wyłączności” w przypadku inteligentnych wskaźników dotyczy jedynie *praw własności*, a nie samego *dostępu do danych*!

Kiedy jak przekazywać `std::unique_ptr`?

Zgodnie z *konwencją* użycia kontenera `std::unique_ptr`:

- Chcąc przenieść prawa własności do obiektu typu
  `std::unique_ptr<T>`
  przekazuj go do funkcji przez wartość.
- Chcąc tylko skorzystać ze wskazywanego przez niego obiektu, przekaż argument typu `T*` albo `const T*` (zgodnie z zasadami „const correctness”).

Szablon klasy `std::shared_ptr`

`std::shared_ptr` działa podobnie jak `std::unique_ptr` – przechowuje wskaźnik, udostępnia podstawowy interfejs dla konstruowania i korzystania ze wskaźnika, oraz zapewnia zwolnienie pamięci przy niszczeniu obiektu.

ALE...

`std::shared_ptr` dodatkowo pozwala na kopiowanie swej instancji, a zarazem zapewnia, że wskazywany obiekt zostanie zniszczony dokładnie w chwili, gdy zostaje zniszczony ostatni obiekt `std::shared_ptr` będący (współ)właścicielem tego obiektu (bądź gdy wszystkie obiekty zwolnią ten wskaźnik)

Szablon klasy `std::shared_ptr`

Przykład:

```
struct MyClass {
  MyClass(int id_);
};

auto ptr = std::make_shared<MyClass>(1);

std::shared_ptr<MyClass> another_ptr = ptr;
// Teraz i `another_ptr`, i `ptr` wskazują na obiekt o id_=1.

ptr.reset();
// Teraz `ptr` przestaje wskazywać na obiekt o id_=1, lecz
//   obiekt ten nie zostaje zniszczony, gdyż `another_ptr`
//   wciąż się do niego odwołuje.

another_ptr.reset();
// Teraz żaden obiekt typu shared_ptr nie odwołuje się do
//   obiektu o id_=1, więc obiekt ten zostaje zniszczony.
```

Kiedy `std::unique_ptr`, kiedy `std::shared_ptr`

Zasada, którą warto stosować przy wyborze typu inteligentnego wskaźnika:

Nie masz pewności, jaki typ wybrać? – wybierz `std::unique_ptr`.

$\Rightarrow$ zawsze możesz później zamienić go na `std::shared_ptr`

Wskaźnik typu `std::shared_ptr` powinien być stosowany *wyłącznie*, gdy faktycznie potrzeba współdzielić *prawa własności* do zasobów (a nie tylko sam *dostęp* do zasobów).

Kiedy `std::unique_ptr`, kiedy `std::shared_ptr`

Argumenty za tym, by `std::unique_ptr` był *domyślnym* wyborem:

- Zawsze należy wybierać najprostsze rozwiązania wystarczające do osiągnięcia celu.

  (zob. zasada KISS)

- Używając `std::unique_ptr` wciąż możesz udostępniać wskazywany zasób – za pomocą metody `get()`.

  („wyłączność” dotyczy jedynie praw własności, nie zaś samego dostępu do danych)

- `std::unique_ptr` jest wydajniejszy i zużywa mniej pamięci niż `std::shared_ptr`

  (`std::unique_ptr` zapewnia niemal identyczną wydajność, co „surowy” wskaźnik)

- Użycie `std::unique_ptr` pozostawia otwartą możliwość konwersji obiektu takiego typu na typ `std::shared_ptr`, z użyciem `std::move()`. Odwrotna operacja nie jest możliwa.

Operacje przenoszące dla `std::unique_ptr`

Konstruktor przenoszący i przenoszący operator przypisania dla `std::unique_ptr` *gwarantują*, że po ich wykonaniu argument będzie wskazywał na `nullptr`:

```
auto u1 = std::make_unique<int>(1);
auto u2 = std::move(u1);
// w tym miejscu zachodzi: u1 == nullptr
```

$\Rightarrow$ niezmiennik unikalnej własności zostaje zachowany

Ograniczenia w stosowaniu inteligentnych wskaźników

Aby uniknąć niezdefiniowanych wyników:

- Korzystaj ze inteligentnych wskaźników tylko do wskazywania na obiekty utworzone dynamicznie.
- Zapewnij, aby każdy zarządzany obiekt posiadał dokładnie jeden obiekt zarządzający.

  Nowo utworzony obiekt powinien zostać momentalnie przekazany do inteligentnego wskaźnika (a pozostałe inteligentne wskaźniki odwołują się poprzez ten pierwszy wskaźnik) – najlepiej z użyciem `std::make_unique` (albo `std::make_shared`).
- Chcesz w pełni wykorzystać możliwości inteligentnych wskaźników? – stosuj „surowe” wskaźniki w odniesieniu do tego samego obiektu wskazywanego przez inteligentny wskaźnik wyłącznie w kontekście, gdzie *nie* stają się one właścicielem obiektu.

Nie stosując się do powyższych zasad stwarzasz ryzyko powstania „wiszących” wskaźników lub podwójnego zwalniania tej samej pamięci!