Kurs języka C++

3. Inicjalizacja, kopiowanie i przenoszenie

Spis treści

Składowe statyczne

- Każdy egzemplarz obiektu danej klasy ma swój własny zestaw danych.
- Pole statyczne jest w pamięci tworzone jednokrotnie i jest wspólne dla wszystkich egzemplarzy obiektów danej klasy.
- Składnik statyczny w klasie istnieje nawet wtedy, gdy jeszcze nie zdefiniowano ani jednego egzemplarza obiektu tej klasy.
- Deklarator `static` określa, że pole z takim przydomkiem ma zostać umieszczone w statycznym obszarze pamięci.

Składowe statyczne

- Deklaracja składnika statycznego w ciele klasy nie jest jego definicją.
- Definicję składnika statycznego należy umieścić w pliku źródłowym.
- Do składnika statycznego należy odnieść się poprzez nazwę klasy (możne też wykorzystać zmienną lub wskaźnik do zmiennej tego typu):
  `Klasa::pole_statyczne`
  `Klasa::funkcja_statyczna(…)`
- Funkcja statyczna nie ma dostępu do składowych instancyjnych.

Inicjalizacja typów podstawowych

- Dla typów podstawowych został stworzony konstruktor nadający jakąś określoną wartość – istnieje więc możliwość inicjalizacji nowotworzonej zmiennej za pomocą formy konstruktorowej. Przykłady:

```
int x(17);
double y(PI / 2);
```

- Dla typów podstawowych został stworzony konstruktor domyślny nadający jakąś wartość domyślną, czyli zero, nowotworzonej zmiennej. Przykłady:

```
bool b = bool();
char c = char();
```

Lista inicjalizacyjna w konstruktorze

- Treść konstruktora może być poprzedzona listą inicjalizacyjną, której przeznaczeniem jest inicjalizacja pól w nowopowstającym obiekcie w formie konstruktorowej.
- Lista inicjalizacyjna występuje w definicji konstuktora za nagłówkiem po dwukropku i przed ciałem konstruktora. Przykład:

```
K::K(int a, string s)
: x(a-1), y(a+1), nazwa(s) {…}
```

- Dzięki konstruktorom nadającym wartość typom podstawowym można na liście inicjalizacyjnej umieścić pola typu podstawowego.
- Użycie listy inicjalizacyjnej jest konieczne w przypadku pól stałych i referencji.
- Na liście inicjalizacyjnej może się znaleźć inny konstruktor tego samego typu oddelegowany do inicjalizacji obiektu albo konstruktory klas bazowych.

Jednolita inicjalizacja listą wartości

- Pomysł inicjalizowania obiektów za pomocą list wartości jest zapożyczony z języka C. Ideą jest, by struktura lub tablica były tworzone, podając po prostu listę argumentów o kolejności zgodnej, odpowiednio, z kolejnością definicji składowych struktury lub kolejnymi elementami tablicy.
- C++ posiada możliwość inicjalizowania stanu obiektu za pomocą list wartości – kolejno deklarowane pola są inicjalizowane kolejnymi wartościami z listy.
- Struktura/klasa jest podatna na jednolitą inicjalizację gdy:
  - w strukturze/klasie wszystkie pola są publiczne;
  - na liście znajduje się tyle wartości ile jest pól i typy tych wartości odpowiadają typom pól;
  - w strukturze/klasie nie są zdefiniowane żadne konstruktory (za wyjątkiem generowanych domyślnie).
- Jednolita inicjalizacja może wystąpić w instrukcji `return` w funkcji zwracającej obiekt podatny na jednolitą inicjalizację.

Jednolita inicjalizacja listą wartości

```
struct elem {
    int id;
    string name;
};
…
elem e1 {12, ”calendar”};
…
elem e2 = {24, ”clock”};
…
elem getelem() {
    return {31, ”month”};
}
```

Inicjalizacja listą wartości

- Argumentem konstruktora może być kolekcja `initializer_list<>`, która pozwala użyć formy inicjalizacji za pomocą listy wartości (tego samego typu).
- Przykład:

```
class elem {
    elem(intializer_list<int> lst) {…}
};
…
elem e = {2, 3, 5, 7};
```

Inicjalizacja listą wartości

- Można użyć formy inicjalizacji za pomocą listy wartości (różnych typów) w stosunku do dowolnego konstruktora.
- Przykład:

```
class elem {
    elem(int i, string s) {…}
};
…
elem e = {7, ”week”};
```

Referencja do stałej jako argument w funkcji

- Referencja do stałej może się odnosić do obiektu zewnętrznego (może być zadeklarowany jako stały) ale również do obiektu tymczasowego.
- Przykład referencji do stałej:

```
const int &rc = (2*3-5)/7+11;
```

- Przykład argumentu funkcji, który jest referencją do stałej:

```
int fun (const int &r);
// wywołanie może mieć postać
// fun(13+17);
// gdzie argumentem może być wyrażenie
// fun(x); // x jest zmienną typu int
```

Argumenty tymczasowe

- Obiekty tymczasowe (określane jako r-wartości), to wartości stojące po prawej stronie operatora przypisania (analogicznie zwykła referencja do zmiennej stojącej po lewej stronie przypisania nazywa się l-wartością).
- Argument w funkcji będący referencją do r-wartości definiujemy jako `TYP &&arg`.
- Argument będący r-referencją może być akceptowany jako niestała wartość, co pozwala funkcjom na ich modyfikację.
- Z punktu widzenia funkcji, której argumentem jest referencja do stałej, nie jest możliwe rozróżnienie pomiędzy aktualną r-wartością a zwykłym obiektem przekazanym referencję.
- Referencja do r-wartości jest akceptowana jako niestała wartość, co pozwala obiektom na ich modyfikację i umożliwia stworzenie semantyki przenoszenia.
- Argumenty r-referencyjne umożliwiają zaimplementowanie semantyki przenoszenia za pomocą konstruktorów przenoszących oraz przypisań przenoszących.

Argumenty tymczasowe

- Przykład (1):

```
class Simple {
  void *Memory; // The resource
public:
  Simple() {
    Memory = nullptr; }
  // the MOVE-CONSTRUCTOR
  Simple(Simple&& sObj) {
    // Take ownership
    Memory = sObj.Memory;
    // Detach ownership
    sObj.Memory = nullptr; }
  Simple(int nBytes) {
    Memory = new char[nBytes]; }
  ~Simple() {
    if(Memory != nullptr) delete[] Memory; }
};
```

Argumenty tymczasowe

- Przykład (2):

```
Simple GetSimple() {
  Simple sObj(10);
  return sObj; }

// R-Value NON-CONST reference
void SetSimple(Simple&& rSimple) {
  // performing memory assignment here
  Simple object;
  object.Memory = rSimple.Memory;
  rSimple.Memory = nullptr;
  // Use object...
  delete[] object.Memory; }
```

Konstruktor kopiujący

- Konstruktor kopiujący służy do utworzenia obiektu, który będzie kopią innego już istniejącego obiektu.
- Konstruktorem kopiującym jest konstruktor w klasie `Klasa`, który można wywołać z jednym argumentem typu:

```
Klasa::Klasa (Klasa &);
Klasa::Klasa (const Klasa &);
```

- Wywołanie konstruktora może nastąpić:
  - w sposób jawny:

```
Klasa wzor;
//…
Klasa nowy = wzor;
```

  - niejawnie, gdy wywołujemy funkcję z argumentem danej klasy przekazywanym przez wartość;
  - niejawnie, gdy wywołana funkcja zwraca wartość w postaci obiektu danej klasy.

Konstruktor kopiujący

- Jeśli programista nie zdefiniuje konstruktora kopiującego to może wygenerować go kompilator (wtedy wszystkie pola są inicjalizowane za pomocą swoich konstruktorów kopiujących).
- Kompilator nie wygeneruje konstruktora kopiującego, jeśli dla pewnego pola w klasie nie będzie można zastosować konstruktora kopiującego.

Przypisanie kopiujące

- Przypisanie kopiujące służy do skopiowania do obiektu danych z innego obiektu.
- Przypisanie kopiujące w klasie `Klasa` może być zdefiniowane z jednym (prawym względem `=`) argumentem (drugim argumentem, lewym względem `=`, jest bieżący obiekt):

```
Klasa & Klasa::operator= (Klasa &);
Klasa & Klasa::operator= (const Klasa &);
```

- Przypisanie kopiujące powinno zwrócić referencję do bieżącego obiektu (aby umożliwić kaskadowe wykorzystanie operatora `=`).
- Jeśli programista nie zdefiniuje przypisania kopiującego to może wygenerować go kompilator (wtedy wszystkie pola są kopiowane za pomocą operatora przypisania).

Kopiowanie a RVO

- Technika RVO (ang. Return Value Optimalization) to optymalizacja kodu w stosunku do wartości zwracanej (pomijanie kopiowania).
- Idea RVO: jeśli funkcja na końcu zwraca przez wartość instrukcją `return` obiekt roboczy stworzony w ciele funkcji, to obiekt roboczy jest tworzony w miejscu przeznaczonym na wynik i pomijane jest kopiowanie.
- Technikę RVO można stosować zarówno przy zwracaniu rezultatu przez funkcję, jak i podczas wysyłania argumentów do funkcji.

Referencja do r-wartości

- R-wartość nie ma swojej nazwy, ponieważ jest to chwilowy rezultat pewnego wyrażenia – nie można się więc dowiedzieć jaki jest adres w pamięci (inaczej niż dla l-wartości, która odnosi się do zmiennej i która ma jakiś adres w pamięci).
- Kompilator umie rozpoznać, czy do przeciążonej funkcji wysłano obiekt (l-wartość), czy chwilową wartość wyrażenia (r-wartość).
- Funkcja biblioteczna `std::move()` nie przenosi a tylko rzutuje!

Konstruktor przenoszący

- Konstruktor przenoszący służy do utworzenia obiektu, który przejmie dane z obiektu tymczasowego.
- Konstruktorem przenoszącym jest konstruktor w klasie `Klasa`, który można wywołać z jednym argumentem typu:
  `Klasa::Klasa (Klasa &&);`
- Wywołanie konstruktora może nastąpić gdy podamy mu jako argument obiekt tymczasowy.
- Gdy w klasie nie ma konstruktora przenoszącego to zostanie użyty konstruktor kopiujący.

Przypisanie przenoszące

- Przypisanie przenoszące służy do przeniesienia do obiektu danych z obiektu tymczasowego.
- Przypisanie przenoszące w klasie `Klasa` może być zdefiniowane z jednym (prawym względem `=`) argumentem:
  `Klasa & Klasa::operator= (Klasa &&);`
- Gdy w klasie nie ma przypisania przenoszącego to zostanie użyte przypisanie kopiujące.