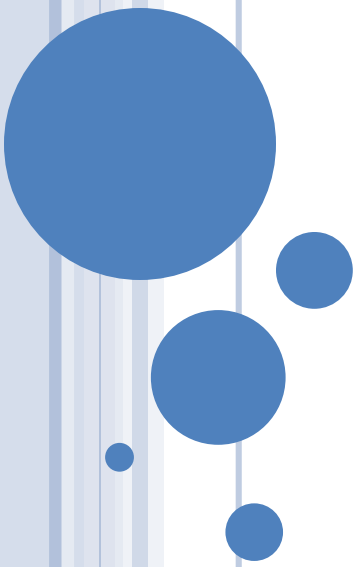

Kurs języka C++ – wykład 3 (19.03.2019)

Inicjalizacja, kopiowanie i przenoszenie

Spis treści

- Inicjalizacja typów podstawowych – konstr.kop. i dom.
- Lista inicjalizacyjna w konstruktorze (klasa ze stałymi lub referencjami)
- Jednolita inicjalizacja listą wartości
- Inicjalizacja listą wartości `initializer_list<>`
- Argument będący referencją do stałej
- Argumenty tymczasowe
- Konstruktor kopiujący i przenoszący
- Przypisanie kopiujące i przenoszące
- Blokowanie konstruktorów kopiujących i przypisań kopiujących generowanych przez kompilator
- Konstruktory delegatowe
- Wnioskowanie typu (auto, decltype)
- [Pola stałe, zawsze modyfikowalne i ulotne] poprz.wykl.

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 inicjowania 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.
- 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};
```

- 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
```

Argumenty tymczasowe

- Obiekty tymczasowe (określane jako r-wartości), mogą być przekazywane do funkcji, ale tylko jako referencje do stałej.
- Z punktu widzenia funkcji, której argumentem jest referencja do stałej, nie jest możliwe rozróżnienie pomiędzy aktualną r-wartością a zwykłym obiektem przekazanym referencję.
- Referencję do r-wartości definiujemy:
  `TYP &&zm`
- Referencja do r-wartości może być akceptowana jako niestała wartość, co pozwala obiektom na ich modyfikację.
- Taka zmiana umożliwia obiektom na stworzenie semantyki przenoszenia.

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.
- Argumenty r-referencyjne umożliwiają pewnym obiektom na stworzenie 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.
- Jeśli programista nie zdefiniuje konstruktora kopiującego to może wygenerować go kompilator (wtedy wszystkie pola są kopiowane za pomocą operatora przypisania).
- Kompilator nie wygeneruje konstruktora kopiującego, jeśli dla pewnego pola w klasie nie będzie można zastosować operatora przypisania `=` (na przykład do pola stałego).

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).

Konstruktor przenoszący i przypisanie przenoszące

- 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 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 &&);`

Stały wskaźnik i wskaźnik do stałej

- Wskaźnik do stałej pokazuje na obiekt, którego nie można modyfikować. Przykład:

```
int a=7, b=5;
const int *p = &a;
// *p = 12; to jest błąd
p = &b; // ok
```

- Stały wskaźnik zawsze pokazuje na ten sam obiekt. Przykład:

```
int a=13, b=11;
int *const p = &a;
*p = 12; // ok
// p = &b; to jest błąd
```

- Można również zdefiniować stały wskaźnik do stałej. Przykład:

```
int c=23;
const int *const p = &c;
```

Pola stałe w klasie

- W klasie można zdefiniować pola stałe z deklaratorem `const`. Przykład:

```
class zakres {
     const int MIN, MAX;
public:
     zakres(int mi, int ma);
     // …
};
```

- Inicjalizacji pola stałego (i nie tylko stałego) można dokonać tylko poprzez **listę inicjalizacyjną** w konstruktorze (po dwukropku za nagłówkiem). Przykład:

```
zakres::zakres(int mi, int ma) : MIN(mi), MAX(ma) {
     if (MIN<0||MIN>=MAX)
          throw string("złe zakresy");
}
```

  Inicjalizacja pól na liście ma postać konstruktorową.

- Konstruktor kopiujący nie zostanie wygenerowany automatycznie tylko wtedy, gdy w klasie nie ma pól stałych.

Stałe funkcje składowe

- W klasie można zadeklarować stałe funkcje składowe z deklaratorem `const`. Przykład:

```
class zakres {
    const int MIN, MAX;
public:
    int min () const;
    int max () const;
    // …
};
```

- Stała funkcja składowa gwarantuje nam, że nie będzie modyfikować żadnych pól w obiekcie (nie zmieni stanu obiektu). Przykład:

```
int zakres::min () const { return MIN; }
int zakres::max () const { return MAX; }
```

- Na obiektach stałych możemy działać tylko stałymi funkcjami składowymi.

Pola zawsze modyfikowalne

- Jeśli obiekt zostanie zadeklarowany jako stały, to można na nim wywoływać tylko stałe funkcje składowe, które nie zmieniają stanu obiektu.
- W klasie można jednak zdefiniować zawsze modyfikowalne pola składowe za pomocą deklaratora `mutable`. Przykład:

```
class zakres
{
    mutable int wsp;
public:
    void nowyWsp (int w) const;
    // …
};
```

- Pole zawsze modyfikowalne może być zmieniane w stałym obiekcie przez stałą funkcję składową. Przykład:

```
void zakres::nowyWsp (int w) const
{
    if (w<0||w<wsp/2||w>wsp*2)
        throw string("zły współczynnik");
    wsp = w;
}
```

Ulotne funkcje składowe

- W klasie można również zadeklarować ulotne funkcje składowe z deklaratorem `volatile`. Przykład:

```
class licznik {
    volatile int ile;
public:
    int ilosc () volatile;
    // …
};
```

- Ulotna funkcja składowa gwarantuje nam, że nie będzie optymalizować kodu przy korzystaniu z pól w obiekcie (nie przechowywać stanu obiektu w podręcznej pamięci). Przykład:

```
int licznik::ilosc () volatile {
    return ile;
}
```

- Na obiektach ulotnych możemy działać tylko ulotnymi funkcjami składowymi.