

Konstruowanie obiektów klas

Wykład 5

Konstruktory

- Konstruktor jest specjalną funkcją składową uruchamianą automatycznie przy definiowaniu każdego obiektu danej klasy
- Służy do nadawania wartości początkowych składnikom tworzonych obiektów
- Może być przeładowany
 - Bardzo częsta praktyka
- Nie zwraca nic
- Nie może być typu `static`
- Nie może być typu `virtual`
- Może być wywoływany do tworzenia obiektów z przydomkiem `const` i `volatile`

Sposoby wywoływania konstruktorów

■ Obiekty lokalne

- Automatyczne - powstają gdy program napotka ich definicję, a przestają istnieć kiedy program wychodzi poza blok, w którym one powstały
- Statyczne - konstruktor wykonany zostanie na początku programu przed wywołaniem funkcji `main()`
 - Zakres ważności odnosi się do bloku
 - Czas życia równy jest czasowi pracy programu

Sposoby wywoływania konstruktorów...

- Obiekty globalne - tworzone są przed wykonaniem funkcji `main()` tak jak statyczne. Czas życia równy czasowi wykonywania programu. Zakres ważności cały plik
- Obiekty tworzone operatorem `new` - konstruktor wywoływany przy tworzeniu. Czas życia od momentu wykonanie `new` do końca programu lub wywołania `delete`. Zakres ważności, wszędzie tam gdzie jest dostępny choć jeden wskaźnik pokazujący na dany obiekt
 - UWAGA utrata wskaźnika nie oznacza zniszczenia obiektu!!!

Konstruktor domniemany

- Konstruktor domniemany to taki, który można wywołać bez żadnych argumentów
 - Może nie mieć żadnych argumentów
 - Może mieć wszystkie argumenty domyślne
 - Klasa może mieć tylko jeden konstruktor domniemany
- Jeśli klasa nie ma żadnego konstruktora to sam kompilator automatycznie wygeneruje konstruktor domniemany
 - Jeżeli klasa ma zdefiniowany chociaż jeden niedomyślny konstruktor to kompilator go już nie utworzy

Lista inicjalizacyjna

- Jeżeli w klasie mamy obiekty z przydomkiem **const** to
 - Nie możemy nadać wartości w ciele klasy w sposób (nie dotyczy stałych statycznych)
const float f = 3.14;
 - W nowym standardzie C++11 możemy nadawać wartości w ciele klasy np.:
float f = 3.14;
 - Do nadawania wartości obiektom **const** służy lista inicjalizacyjna konstruktora
 - **klasa::klasa(args) : lista {...}**
- Wykonanie konstruktora odbywa się w dwóch etapach
 - Inicjalizacja składników
 - Wykonanie instrukcji w ciele konstruktora
 - **Funkcje wirtualne?!**

Lista inicjalizacyjna...

- Listę zapisujemy w następujący sposób
 - `klasa::klasa(int a, float f) :
 zm_const_int(a), zm_const_float(f)
 {}`
- Na liście inicjalizacyjnej mogą się znaleźć także obiekty nie-**const**
- Lista inicjalizacyjna nie może inicjalizować obiektów typu **static**
- Argument którym inicjalizujemy nie musi być wartością może być wyrażeniem nawet z funkcją
- Przykład `cpp_5.1`

Tworzenie obiektów ze składnikiem z innej klasy

- Sytuacja bardzo częsta kiedy obiekt jednej klasy jest składnikiem innej klasy
 - Taki składnik może być inicjalizowany tylko za pomocą listy inicjalizacyjnej
 - Jeżeli nie posiada konstruktora to jest wywoływany automatycznie stworzony konstruktor domyślny
 - Jeżeli nie posiada konstruktora domyślnego to jej inicjalizacja musi pojawić się na liście
 - Konstruktory składników są wywoływane przed wywołaniem konstruktora klasy, w której się znajdują (destrukторы odwrotnie)
 - W jakiej kolejności?
- Przykład cpp_5.2

Konstruktor kopiujący

- Konstruktor kopiujący jest wywoływany z jednym argumentem będącym referencją do tej samej klasy
 - `klasa::klasa(klasa&);`
 - `klasa::klasa(klasa&, typ=1);`
- Służy do stworzenia kopii istniejącego obiektu danej klasy
- Konstruktor kopiujący nie jest obowiązkowy
 - Jeżeli nie zostanie przez nas zdefiniowany to kompilator stworzy go sam
 - Nie ma sensu i powodu tworzyć konstruktora kopiującego jeżeli będzie on taki sam jak ten, który wygeneruje kompilator
 - **A kiedy jest?**
- Przykład `cpp_5.3`

Konstruktor kopiujący...

- Sytuacje w których jest wywoływany
 - Przy wywoływaniu funkcji w której argument przesyłany jest przez wartość
 - Gdy funkcja zwraca jako rezultat przez wartość obiekt klasy
- Konstruktor kopiujący powinien zawsze odbierać obiekt z przydomkiem **const**
 - Jeśli nie to może się okazać, że nie będziemy mogli zrobić kopii obiektu który jest **const**
 - Przykład cpp_5.4
- Przykład kiedy konstruktor kopiujący jest niezbędny (domyślnie wygenerowany działa źle)
 - Przykład cpp_5.5

Konstruktory niepubliczne

- Konstruktor podlega regułom dostępu takim jak wszystkie inne funkcje składowe
 - Najczęściej konstruktor jest publiczny
 - Jeżeli klasa nie posiada publicznego konstruktora to nazywamy ją klasą prywatną
 - Ale po co?
- Konstruktor niepubliczny może wywołać funkcja lub klasa zaprzyjaźniona
- Jeżeli konstruktor jest **protected** to może być wywołany jak wyżej oraz z klasy pochodnej
- Przykład cpp_5.6

„Nazwane konstruktory”

- Jest to technika umożliwiająca użytkownikom klasy na bardziej intuicyjne i/lub bezpieczniejsze tworzenie jej obiektów
- Problem ze "zwykłymi" konstruktorami jest taki, że mają one tą samą nazwę, co klasa
 - Z tego powodu można je rozróżnić tylko dzięki odmiennym listom parametrów formalnych
 - Jeżeli konstruktorów pewnej klasy jest zbyt dużo, różnice między nimi stają się subtelne, co może stać się przyczyną różnych błędów
- Technika „*Named Constructor Idiom*” polega na tym, że wszystkie konstruktory deklaruje się jako prywatne lub chronione, a następnie definiuje się publiczne i statyczne metody, które tworzą i zwracają nowy obiekt
 - Generalnie definiuje się po jednej takiej statycznej metodzie dla każdego sposobu na utworzenie obiektu danej klasy
 - W ten sposób można stworzyć klasę finalną tzn. taką, po której się nie da dziedziczyć
- Przykład `cpp_5.7`

Destruktory

- Destruktor wywoływany jest automatycznie przy likwidacji obiektu
- Destruktor nie zwraca nic
- Destruktor nie może być przeciądowany
- Jawne wywołanie destruktora wykonuje polecenia w nim zawarte, ale nie niszczy samego obiektu
- Destruktor „**musi**” być zdefiniowany jeśli np. w konstruktorze tworzymy obiekty za pomocą **new** to w destruktor musi te obiekty usunąć za pomocą **delete**
- Przykład cpp_5.8

Tablice obiektów

- Inicjalizacja tablic obiektów danej klasy
 - Kiedy tablica jest agregatem
 - Nie ma składników `private` i `protected`
 - Nie ma konstruktorów
 - Nie ma klas podstawowych
 - Nie ma funkcji wirtualnych
 - Przykład cpp_5.9
 - Kiedy tablica nie jest agregatem
 - Przykład cpp_5.10
 - Kiedy tablica jest tworzona przy pomocy operatora `new`
 - Nie mogą mieć jawnie wypisanej inicjalizacji
 - Musi mieć konstruktor domniemany!!!
 - Przykład cpp_5.11
- W c++11 wprowadzona ujednoliconą inicjalizację
 - `std::initializer_list<T>`
 - Przykład cpp_5.11a

Wskaźniki do składników klas i metod

- Zwykłym wskaźnikiem można pokazywać na składnik klasy, który jest publiczny
 - `int *a = &klasa.skladnik_int;`
- Wskaźnik do składowej, który pokazuje nie na konkretne miejsce w pamięci, ale na odległość od początku obiektu danej klasy
 - `int K::*wsk = &K::skladowa_int`
- Wskaźniki do składników statycznych mogą być tylko zwykłymi wskaźnikami
- Podobnie możemy sobie zdefiniować wskaźnik do funkcji składowej danej klasy
 - `int (K::*wsk)(float)`
- Również możemy deklarować i używać tablic wskaźników
 - Zarówno do składowych jak i metod klasy

Konwersje (typów)

- Konwersje (typów) - innymi słowy przekształcenia obiektów jednego typu na inny
 - Konwersje niejawne
 - Konwersje jawne
- Konwersje typu jakim jest klasa muszą zostać zdefiniowane przez programistę, kompilator nie wykona konwersji jeśli nawet wydaje się nam oczywista

Konwersje niejawne

- Przy wywołaniu funkcji
 - Przy niezgodności argumentów wywołania z formalnymi, ale istnieje jednoznaczna metoda pozwalająca na usunięcie niedopasowania
- Przy zwracaniu rezultatu funkcji
 - Jeśli przy słowie `return` stoi inny typ niż funkcja ma zwrócić i jednocześnie istnieje możliwość jednoznacznej konwersji
- W obecności operatorów
 - Np. przy operatorze `+` kompilator spodziewa się, że będą stały obiekty jednego typu
- W instrukcjach
 - `if`, `switch`, `while`, `for`
- Przy wrażeniach inicjalizujących
- Należy unikać niejawnych konwersji, szczególnie nietypowych

Konstruktor jako konwerter

- Jednoargumentowy konstruktor określa sposób konwersji od danego typu (argumentu, który przyjmuje) do obiektu tej klasy
- Typem z którego dokonywana jest konwersja może być typ inny niż wbudowany (klasa)
 - Klasa z której dokonywana jest konwersja musi dawać dostęp do koniecznych przy konwersji swoich składników
 - Przez zadeklarowanie ich jako **public** - nie zalecane
 - Przez deklarację przyjaźni z konstruktorem
 - Przez publiczne funkcje składowe
- Przykład `cpp_5.12`

Cechy konstruktora konwertującego

- Nie można zdefiniować konstruktora dla typu wbudowanego
- Nie można napisać konstruktora dla obcej klasy
- Musi mieć dostęp do odpowiednich składowych klasy
- Argument w konstruktorze musi pasować dokładnie
 - Nie można polegać na standardowych konwersjach
- Konstruktor się nie dziedziczy
- **Konstruktor jednoargumentowy** z przydomkiem **explicit** zapewnia, że nie zostanie użyty do **niejawnej konwersji**
- Przykład cpp_5.13

Funkcja konwertująca

- Możemy także wyobrazić sobie sytuację odwrotną
 - Np. chcemy wywołać funkcję działającą na określonym typie, a wywoływaną z obiektem klasy innego typu
 - `void fun(float x);`
`Fraction aFraction;`
`fun(aFraction);`
 - Musimy naszą klasę wyposażyć w funkcję składową, która przekształci obiekt tej klasy np. w typ `float`
 - `K::operator T();`
 - `T` - oznacza nazwę typu
- Przykład `cpp_5.14` i `cpp_5.15`
- Standard `c++11` funkcja konwertująca w wersji
 - `explicit K::operator T();`
 - Przykład `cpp_5.15a`

Funkcja konwertująca...

- Musi być funkcją składową klasy
- Nie posiada określenia typu rezultatu jaki zwraca
- Zawsze zwraca typ taki jak się sama nazywa
- Nie zawiera żadnych argumentów wywołania
 - Powinna być const
- Jest dziedziczona
 - W przeciwieństwie do konstruktorów konwertujących
- Może być funkcją wirtualną
- Działa odwrotnie niż konstruktor konwertujący
 - Konwertuje z typu danej klasy, której jest składową na inny typ

Który sposób wybrać?

- Przy konwersji obiektu klasy A na typ wbudowany tylko funkcja konwertująca
- Przy konwersji obiektu klasy A na obiekt klasy B
 - Po pierwsze funkcja konwertująca w klasie A
 - Tylko w sytuacji braku możliwości zmian klasy A konstruktor konwertujący w klasie B (pokazane wcześniej wymagania co do klasy A)
 - Nie należy stosować obu metod na raz bo przy próbie niejawniej konwersji kompilator zaprotestuje!!!

Inne aspekty konwersji

- Nie należy mnożyć konwersji
- W dobrym stylu jest zdefiniowanie tylko jednego operatora konwersji
 - Użycie słowa kluczowego `explicit` jest bardzo wskazane
- Jeżeli potrzeba więcej konwersji to lepiej napisać odpowiednie funkcje składowe
 - Przyjęte nazwy `toType()` , `asType()`
- Należy pamiętać czy konwersja ma w ogóle sens

Konwersje jawne

- Przejęte z języka C
 - Za pomocą ()
 - `float a = 3.14;`
`int b = (int) a;`
 - Nie zalecane ze względu na możliwość niejednoznaczności, używać tylko w sytuacjach oczywistych, a najlepiej w ogóle
- Wprowadzone w C++
 - `static_cast<typ>(zmienna);`
 - Bezpieczniejszy, (uwzględnia budowę klasy przy rzutowaniu wskaźników w przeciwieństwie do rzutowanie wymuszonego)
 - Może wywoływać funkcje
 - `const_cast<typ>(zmienna);`
 - Służy on do rzutowania wskaźników lub referencji różniących się modyfikatorem `const`
 - Pozwala wysłać stały obiekt do funkcji, która nie gwarantuje że nie zmieni obiektu
 - Należy unikać, chyba że nie ma innej możliwości
 - Reinterpretacja bitów na nie-`const` (dyrektywa kompilatora)
 - `reinterpret_cast<typ>(zmienna);`
 - Generalnie służy on do rzutowania wskaźników lub typów całkowitych
 - Reinterpretacja bitów tak jak `const_cast`
- Przykład `cpp_5.16` i `cpp_5.16a`

Przestrzenie nazw

- Przestrzenie nazw - **namespace**
 - Służą logicznemu grupowaniu typów i funkcji w języku C++
 - Możemy umieszczać w nich dowolne symbole
 - Symbole mogą być umieszczane w różnych plikach
 - Użycie symbolu poza przestrzenią nazw wymaga poprzedzenia go odpowiednią nazwą
 - W ten sposób unikamy konfliktu nazw oraz określamy przynależność symbolu do określonej przestrzeni nazw (inaczej tworzymy pakiety lub komponenty)
- Przykład
 - ```
namespace A {
 class Nazwa{...}; //definiuje klasę A::Nazwa
}
```

# Przestrzenie nazw...

- Postępując się symbolami z określonej przestrzeni nazw możemy uniknąć wielokrotnego stosowania operatora zakresu
  - Za pomocą deklaracji `using`
    - `using OOP::Point;`
    - W bieżącym zakresie `Point` staje się synonimem `OOP::Point`
  - Za pomocą dyrektywy `using`
    - `using namespace OOP;`
    - Wszystkie symbole przestrzeni nazw `OOP` traktowane są jak zmienne globalne w bieżącym zakresie
- Należy pamiętać o możliwości wystąpienia dwuznaczności
- Dyrektywy `using` nie należy stosować w kontekstach, w których nie jest jasne jakie symbole są dostępne globalnie
  - Na pewno nie należy tej dyrektywy umieszczać w plikach nagłówkowych
- Przykład `cpp_5.17`

# Obiekty stałe jako zmienne

- Czasami zachodzi potrzeba przekształcenia stałej w zmienną
  - Np. operujemy obiektem statym na pliku, który ma określoną liczbę wierszy, ale nieznaną w czasie kompilacji, normalnie nie moglibyśmy zmienić składnika takiego obiektu
  - Jest to możliwe jeżeli taki składnik zadeklarujemy ze słowem `mutable`
- Składnik z przydomkiem `mutable` nie ma wpływu na logiczną stałość obiektu, dzięki temu może być modyfikowany nawet przez stałe funkcje składowe
- Przykład `cpp_5.18`