Podstawy programownia (w języku C++)

Struktury danych

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

4 października 2021

Overview

Po co?

Struktury danych

Struktury (typy) danych wykorzystuje się jeśli zachodzi potrzeba zgrupowania pewnych danych mających *wspólny raison d'être*[1]; reprezentujących zestaw *danych* i *operacji* na tych danych, których przechowywanie osobno nie miałoby większego sensu.

Biblioteka standardowa zawiera strukturę danych `std::vector`, która reprezentuje tablicę o zmiennej długości. Grupuje ona *dane* (zawartość i rozmiar), oraz *operacje* (np. dodawanie, usuwanie i dostęp do elementów).

[1] `https://en.wiktionary.org/wiki/raison_d'être`

Składniki

Struktury danych

Struktury (typy) danych składają się przede wszystkim z **pól** (ang. *fields* lub *member variables*), czasem zwanych zmiennymi lub stałymi składowymi, i **funkcji składowych** (ang. *member functions*).

```
struct Cat {
    size_t number_of_lives { 1 };     // member variable (field)
    bool const brings_luck { false }; // member constant (field)

    auto make_sound() const -> void;  // member function
};

auto Cat::make_sound() const -> void
{
    std::cout << "Meow! I have "
        << number_of_lives << " lives left.\n";
}
```

Pola

Struktury danych

Pola służą do przechowywania *danych*, które dany typ grupuje. Na przykład rozmiar i zawartość wektora.
Pola mogą być zmienne - jeśli ich wartości mogą podlegać modyfikacji w trakcie życia obiektu danego typu (np. rozmiar `std::vector`), albo stałe - jeśli nie podlegają modyfikacji (np. rozmiar `std::array`).

Definiując pole trzeba użyć starego stylu deklaracji, czyli poprzedzić nazwę typem, a nie słowem kluczowym `auto`:

```
size_t number_of_lives { 1 };
```

Funkcje składowe

Struktury danych

Funkcje składowe służą do wykonywania *operacji* na danych zgrupowanych przez dany typ (np. `std::vector::push_back`), lub do ogólnej interakcji z nim (np. `Cat::make_sound`).

`class` vs `struct`

Jedyna różnica to domyślna *widoczność* pól i funkcji składowych. Pola klas (`class`) są prywatne, a struktur (`struct`) publiczne. Widoczność można zmieniać słowami kluczowymi `private` i `public`.

```
class Foo {
    public:
    bool now_you_see_me { true };

    private:
    bool now_you_dont { false };
};
```

Widoczność pól

Struktury danych

Po co jest widoczność? Niektóre pola w strukturze mogą nie być przeznaczone dla "użytkowników" lub wymagać zachowania pewnych warunków. Jeśli jest taka potrzeba zawsze można udzielić dostępu do pola przez funkcje składowe.

```
class Hour {
    unsigned value { 0 };  // private by default
    public:
    auto what_time_is_it() const -> unsigned;
    auto increase_time() -> void;
};
auto Hour::what_time_is_it() const -> unsigned
{
    reurn value;
}
auto Hour::increase_time() -> void
{
    ++value;
    if (value > 23) {  // make sure the hour wraps after 23
        value = 0;
    }
}
```

Konstruktor

Struktury danych

Konstruktor (ang. *constructor*) jest specjalną funkcją składową odpowiedzialną za ”przygotowanie” struktury danych do użycia. Konstruktor może zasygnalizować niemożność utworzenia instancji struktury (np. z powodu niewłaściwych wartości argumentów) używając wyjątków[2].

```
class Hour {
    // code from last slide here...
    public:
    Hour(unsigned);
};
Hour::Hour(unsigned v)
    : value{v} // initialise the field using values from parameters
{
    if (v > 23) { // throw if they don't make sense
        throw std::out_of_range{"hour value cannot exceed 23"};
    }
}
```

[2]ponieważ konstruktor jako takie nie zwraca wartości więc nie ma jak inaczej zasygnalizować błędu

Overview

Zmienne

Pola

Definicja zmiennego pola wygląda tak jak definicja każdej innej zmiennej. Załóżmy, że chcemy stworzyć strukturę opisującą kota. Jak wiadomo, koty mają 9 żyć do wykorzystania:

```
struct Cat {
    constexpr static unsigned MAX_LIVES = 9;
    unsigned lives_left { MAX_LIVES };
};
```

Inicjalizacja zmiennych składowych

Pola

Zmienne składowe można zainicjalizować na kilka sposobów:

1. przypisując im wartość w liście inicjalizującej składowe (ang. *member initialiser list*)
2. przypisując im wartość w ciele konstruktora
3. pozostawiając ich wartości domyślne zdefiniowane w ciele struktury

Inicjalizacja zmiennych składowych (c.d.)

Pola

```
struct Cat {
    constexpr static unsigned MAX_LIVES = 9;
    unsigned lives_left { MAX_LIVES }; // default value provided

    Cat() = default; // default ctor
    Cat(unsigned const);
};
Cat::Cat(unsigned const ll)
    : lives_left{ll} // use member initialiser list
{}
```

Jeśli definiujemy konstruktor to kompilator nie wygeneruje konstruktora domyślnego (z pustą listą parametrów). Jeśli chcemy w takiej sytuacji skorzystać z domyślnego konstruktora to możemy albo zdefiniować go samodzielnie, albo użyć konstrukcji = `default`.

Dostęp

Pola

Aby dostać się do pola struktury należy użyć operatora '.' (kropki):

```
auto a_cat = Cat{}; // an ordinary cat

std::cout << a_cat.lives_left << "\n";
```

Jeśli pole jest zmienne, można je zmodyfikować:

```
a_cat.lives_left -= 1; // the cat died
```

lub przypisać mu całkiem nową wartość:

```
a_cat.lives_left = 666; // cat from hell
```

Stałe

Pola stałe (stałe składowe)

Definicja stałego pola wygląda tak jak definicja każdej innej stałej. Kontynuując przykład z kotem:

```
struct Cat {
    constexpr static unsigned MAX_LIVES = 9;
    unsigned lives_left { MAX_LIVES };

    std::string const name;

    Cat() = default;
    Cat(unsigned const);
};
```

Wartości pół stałych nie można[3] zmienić, nawet w ciele konstruktora. Do ich inicjalizacji służy specjalna notacja.

[3] Oh, rly?

Inicjalizacja stałych składowych

Pola stałe (stałe składowe)

Żeby mieć możliwość inicjalizacji stałego pola różnymi wartościami w różnych instancjach struktury potrzebny jest konstruktor przyjmujący jako parametr wartość, którą stałe powinno być zainicjalizowane:

```
struct Cat {
    constexpr static unsigned MAX_LIVES = 9;
    unsigned lives_left { MAX_LIVES };

    std::string const name;

    Cat() = default;
    Cat(unsigned const, std::string);
};
Cat::Cat(unsigned const ll, std::string n)
    : lives_left{ll}
    , name{std::move(n)}
{}
```

Dostęp

Pola stałe (stałe składowe)

Do stałych pól struktury można się dostać za pomocą operatora '.' (kropki):

```
auto mr_snuggles = Cat{ 9, "Mr. Snuggles" };

std::cout << mr_snuggles.name << "\n";
```

Możliwy jest jednak dostęp wyłącznie do odczytu, a kompilator zapobiegnie ich przypadkowej modyfikacji:

```
mr_snuggles.name = "Evil Elvis"; error!
```

Dostęp - modyfikacja stałych składowych?!

Pola stałe (stałe składowe)

Rysunek: Hackerman[4]

[4] Kung Fury (2015), https://www.imdb.com/title/tt3472226/

Dostęp - modyfikacja stałych składowych?!

Pola stałe (stałe składowe)

Jeśli chce się zaimponować cioci w odwiedzinach, albo babci na święta to można pokazać im jak oszukać kompilator i zmodyfikować stałe składowe!

```
std::cout << mr_snuggles.name << "\n";

auto wait_its_illegal = &mr_snuggles.name; // pointer to member
*const_cast<std::string*>(wait_its_illegal) = "Evil Elvis";

std::cout << mr_snuggles.name << "\n";
```

Przy okazji widać też jak można pobrać *wskaźnik do składowej* (ang. *pointer to member*). Nie jest to coś co często się przydaje, ale na pewno częściej niż modyfikacja stałych składowych.

Overview

Definiowanie funkcji składowych

Funkcje składowe

Funkcję należy zadeklarować wewnątrz struktury, a następnie zdefiniować poza strukturą.

Definicje struktur (ich nazwy, pola, funkcje składowe) znajdują się zazwyczaj w plikach nagłówkowych, np. `Cat.h` (wyjątek to statyczne zmienne składowe, które trzeba dodatkowo zadeklarować w pliku z implementacją).
Definicje funkcji składowych znajdują się zazwyczaj w plikach z ich implementacją, np. `Cat.cpp` (wyjątek to np. funkcje `inline` albo szablony).

Definiowanie funkcji składowych (c.d.)

Funkcje składowe

Deklaracja funkcji wewnątrz definicji struktury, definicja funkcji poza:

```
// Cat.h
struct Cat {
    auto make_sound() const -> void;
};

// Cat.cpp
auto Cat::make_sound() const -> void
{
    std::cout << ("Meow! (" + name + ")\n");
}
```

Wewnątrz definicji funkcji można używać jej pól (stała składowa `name` na przykładzie).

Overview

Zadanie: student

Zaimplementować strukturę danych opisującą studenta. Struktura powinna składać się z:

1. pól (imię, nazwisko, numer indeksu, aktualny semest, średnia ocen)
2. funkcji składowej '`to_string() const`' zwracającej `std::string`, którym opisuje studenta
3. konstruktora

Niech w funkcji main będzie utworzony obiekt reprezentujący was, a na `std::cout` wydrukowany będzie wynik działania funkcji `Student::to_string` na tym obiekcie.

Kod źródłowy w plikach `include/s1234/Student.h` (nagłówek) i `src/s03-Student.cpp` (implementacja i funkcja `main`).

Zadanie: czas

Zaimplementować strukturę danych opisującą czas. Struktura powinna składać się z:

1. pól (godzina, minuta, sekunda)
2. funkcji składowych:
   2.1 'to_string() const' zwracającej std::string pokazującej czas w formacie HH:MM:SS
   2.2 next_hour(), next_minute(), i next_second() (wszystkie zwracające void) zwiększających czas
3. konstruktora

Niech w funkcji main pojawi się kod pozwalający na zweryfikowanie działania waszej struktury danych dla godziny 23:59:59 (np. niech drukuje godzinę, zwiększy mintę, wydrukuje znowu, itd.).

Kod źródłowy w plikach include/s1234/Time.h (nagłówek) i src/s03-Time.cpp (implementacja i funkcja main).

Zadanie: pora dnia

Do struktury opisującej czas dodać funkcję składową '`time_of_day() const`' zwracającą porę dnia (rano, dzień, wieczór, noc). Pora dnia powinna być opisana typem wyliczeniowym `enum class Time_of_day`.
Napisać funkcję `to_string(Time_of_day)` zwracającą `std::string`, która zamieni wartość wyliczeniową na napis.
W funkcji main dodać kod pozwalający na zweryfikowanie działania dodanych funkcji.

Kod źródłowy w plikach `include/s1234/Time.h` (nagłówek) i `src/s03-Time.cpp` (implementacja i funkcja `main`).

Zadanie: arytmetyka

Do struktury opisującej czas dodać funkcje składowe:

```
auto operator+ (Time const&) const -> Time;
auto operator- (Time const&) const -> Time;
auto operator< (Time const&) const -> bool;
auto operator> (Time const&) const -> bool;
auto operator== (Time const&) const -> bool;
auto operator!= (Time const&) const -> bool;
```

Umożliwią one arytmetykę (dodawanie, odejmowanie, porównywanie, itd.) na czasie. Do funkcji `main` dodać kod, który pozwoli na zweryfikowanie działania.

Kod źródłowy w plikach `include/s1234/Time.h` (nagłówek) i `src/s03-Time.cpp` (implementacja i funkcja `main`).

Zadanie: sekundy do północy

Do struktury opisującej czas dodać funkcje składowe:

```
auto count_seconds() const -> uint64_t;
auto count_minutes() const -> uint64_t;
auto time_to_midnight() const -> Time;
```

Funkcje `count_seconds()` i `count_minutes()` liczą sekundy *od* godziny 00:00:00.
Funkcja `time_to_midnight()` zwraca czas pozostały *do* północy.
Do funkcji `main` dodać kod, który pozwoli na zweryfikowanie działania.

Kod źródłowy w plikach `include/s1234/Time.h` (nagłówek) i `src/s03-Time.cpp` (implementacja i funkcja `main`).

Overview

OOP

OOP – programowanie zorientowane obiektowo (ang. *object-oriented programming*).

Na początku lat 70. XX wieku powstał język Smalltalk[5]. Jego głównym zamysłem była swobodna *wymiana wiadomości* między *obiektami* (dlatego "zorientowane obiektowo") współistniejącymi w systemie.
W późnych latach 80. XX wieku powstał język Erlang[6], którego jednym z głównych mechanizmów była *asynchroniczna* wymiana wiadomości.

[5] `https://en.wikipedia.org/wiki/Smalltalk`
[6] `https://en.wikipedia.org/wiki/Erlang_(programming_language)`

OOP (c.d.)

Na początku lat 60. XX wieku został opracowany język Simula[7], który z językiem Smalltalk nie miał zbyt wiele wspólnego, ale za to posiadał klasy i dziedziczenie.

W 1985 powstał język C++[8], który zmiksował składnię i funkcjonalność języka C z mechanizmami języka Simula (klasy i dziedziczenie).

[7] `https://en.wikipedia.org/wiki/Simula`
[8] `https://en.wikipedia.org/wiki/C++`

OOP (c.d.)

W 1995 powstał język Java[9], który wzorował się na Simuli i C++ - też miał klasy i dziedziczenie. W Java wszystkie klasy mają wspólną klasę bazową - nazwaną, pechowo, `Object`. Nie jest to jedyna pechowa decyzja w tym języku...

Dział marketingu Oracle uznał, że to oznacza, że język jest zorientowany obiektowo i od tamtego momentu OOP zmieniło się nie do poznania[10].
Zamiast oznaczać swobodną, asynchroniczną wymianę wiadomości zaczęło oznaczać programowanie oparte na klasach i wzorce projektowe usiłujące zamaskować biedę semantyczną języka Java.

[9] `https://en.wikipedia.org/wiki/Java_(programming_language)`
[10] na zajęciach OOP będzie oznaczać tą drugą, zdegenerowaną formę, a nie oryginalną ideę

Dziedziczenie – po co?

OOP

Dziedziczenie (ang. *inheritance*) polega na zbudowaniu jednego typu (*pochodnego*; ang. *derived type*), w oparciu o drugi typ (*bazowy*; ang. *base type*). Mówimy, że typ pochodny *dziedziczy po* typie bazowym.

Typ pochodny posiada "z automatu" wszystkie pola i funkcje składowe typu bazowego, ale dostęp ma tylko publicznych[11], czyli typ pochodzny *dziedziczy implementację*[12] typu bazowego.

[11] Tak naprawdę ma jeszcze dostęp do pól chronionych (słowo kluczowe `protected`), ale pola chronione to coś czego nie polecam używać. Dziedziczenie implementacji jest brudnym mechanizmem nawet bez dodatkowych komplikacji.

[12] To czy odziedziczone pola są widoczne "na zewnątrz" typu pochodnego zależy od tego czy dziedziczył prywatnie (wtedy nie są) czy publicznie (wtedy są).

Polimorfizm – po co?

OOP

Polimorfizm (ang. *polymorphism*) polega na wykorzystaniu wspólnego interfejsu dla wielu typów. Typy prezentujące ten interfejs mogą być używane zamiennie.

Polimorfizm w odniesieniu do funkcji oznacza, że może ona działać na wielu typach.[13]

W C++ żeby móc wykorzystać polimorfizm potrzebe są wskaźniki lub referencje (na etapie działania programu, ang. *run time*), lub szablony (na etapie kompilacji, ang. *compile time*).

[13] Jest to *de facto* druga strona tego co powiedzieliśmy o typach, bo można powiedzieć, że funkcja akceptuje typy spełniające pewien interfejs i w ten sposób zatoczyć koło.

Po co? (moim zdaniem)

OOP

Dziedziczenie w C++ jest sposobem na reprezentację relacji między typami danych. Czyli jego *pierwszorzędną* rolą jest zapewnienie hierarchii typów, a dopiero *drugorzędną* (niemal przy okazji) dziedziczenie implementacji.

Najprościej jak się da, nie ma co sobie komplikować życia. Jeśli chcemy współdzielić implementację mamy do tego dedykowany mechanizm - *procedural abstraction*[14] czyli funkcje.

[14] patrz pierwszy wykład

Overview

Zadanie: kalkulator

Rozwinąć kalkulator napisany w stylu obiektowym tak, żeby nie odbiegał funkcjonalnością od kalkulatora napisanego w stylu proceduralnym zaimplementowanego w ramach poprzednich zajęć.
Kod do wystartowania znajduje się w repozytorium z materiałami[15] w plikach `src/04-rpn-calculator-oo.cpp` (implementacja) i `include/RPN_calculator.h` (plik nagłówkowy).

Wasz kod powinien znaleźć się w plikach `src/s04-rpn-calculator-oo.cpp` (implementacja) i `include/RPN_calculator.h` (plik nagłówkowy).

[15] `https://git.sr.ht/~maelkum/education-introduction-to-programming-cxx`

Overview

Podsumowanie

Student powinien umieć:

1. samodzielnie zaprojektować własny typ danych, jego pola i funkcje składowe
2. wytłumaczyć czym jest i jak działa funkcja składowa, oraz czym jest `this`
3. powiedzieć jaka jest rola konstruktora

Zadania

Podsumowanie

Zadania znajdują się na slajdach 24, 25, 26, 27, 28, 37.