Podstawy programownia (w języku C++)

Struktury danych

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

19 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.

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

Wyliczenia

Struktury danych

Pierwszym rodzajem struktur danych w C++ są typy wliczeniowe.
Są ich dwa rodzaje – silne (`enum class`) i słabe (`enum`):

```
enum class Strong {
    CONAN_THE_BARBARIAN,
    JOHN_CARTER_OF_MARS,
};

enum Weak {
    KOBOLD,
    GNOLL,
};
```

Wyliczenia silne definiują nowy typ danych i pozwalają jedynie na porównywanie wartości. Słabe są *de facto* wartościami typu `int` w przebraniu, i pozwalają na wszystko na co pozwala `int` (w tym automatyczne konwersje między wartościami typu `int` i słabymi wyliczeniami).

Struktury

Struktury danych

Drugim rodzajem struktur danych w C++ są struktury.

```
struct Player_character {
    std::string name;
    short int   proficiency_bonus;
    int         hit_points;
    short int   hit_dice;
};
```

Definiują one nowy typ danych i pozwalają na definiowanie dla nich dowolnych operacji.

Składniki

Wyliczenia

Wyliczenia jawnie definiują wartości jakie mogą przybrać.
Silny typ wyliczeniowy reprezentuje logiczną sumę (alternatywę, *albo*) wszystkich swoich wartości: wartość A *albo* wartość B, nigdy obie naraz.
Słaby typ wyliczeniowy to nieco bardziej śliski temat, i może "mieszać" deklarowane przez siebie wartości – jest w stanie reprezentować wartość, która nie jest jawnie wyliczona.

Silne wyliczenia

Wyliczenia

Przydatne do zdefiniowania dyskretnych, "niemieszalnych" wartości:

```
enum class Emotion {
    ANGER,
    FRUSTRATION,
    FEAR,
    JOY,
    HOPE,
};

auto es = std::set<Emotion>{ Emotion::FEAR, Emotion::HOPE };
```

Jeśli zmienna powinna być w stanie reprezentować kilka takich wartości naraz można użyć `std::set` (z nagłówka `<set>`) do ich przechowania.

Słabe wyliczenia

Wyliczenia

Przydatne do zdefiniowania "mieszalnych" wartości lub flag:

```
enum Emotion {
    ANGER       = (1 << 0),
    FRUSTRATION = (1 << 1),
    FEAR        = (1 << 2),
    JOY         = (1 << 3),
    HOPE        = (1 << 4),
};

auto ew = FEAR | HOPE;
```

Mieszanie wartości można osiągnąć za pomocą operatora sumy bitowej[2] czyli | (tzw. *pipe*).

[2] ale wtedy sztuczka z przesunięciami bitowymi (ang. *bit shift*) jest konieczna, bo każde pole musi mieć zapalony inny bit

Wyliczenia silne *vs* słabe

Wyliczenia

Reprezentacja kilku wartości naraz:

```
// enumeration, strong
auto es = std::set<Emotion>{ Emotion::FEAR, Emotion::HOPE };

// add new value to the mix
es.insert(Emotion::ANGER);
```

vs

```
// enumeration, weak
auto ew = FEAR | HOPE;

// add new value to the mix
ew |= ANGER;
```

Wyliczenia silne *vs* słabe

Wyliczenia

Sprawdzenie czy zmienna zawiera daną wartość:

```
auto es = std::set<Emotion>{ Emotion::FEAR, Emotion::HOPE };

if (es.count(Emotion::ANGER)) { /* ... */ }
```

vs

```
auto ew = FEAR | HOPE;

if (ew & ANGER) { /* ... */ }
```

Wyliczenia silne *vs* słabe

Wyliczenia

Usunięcie wartości ze zmiennej:

```
auto es = std::set<Emotion>{ Emotion::FEAR, Emotion::HOPE };

es.erase(Emotion::FEAR);
```

vs

```
auto ew = FEAR | HOPE;

ew = (ew ^ FEAR);
```

Wyliczenia silne *vs* słabe

Wyliczenia

Wyliczenia silne i słabe pozwalają na osiągnięcie tego samego celu, ale na różne sposoby.

Zaletą wyliczeń silnych jest to, że definiują nowy typ danych więc nie ma szans na przypadkową automatyczną konwersję, która sprawi, że w programie pojawi się "niewyliczona" wartość.

Składniki (1)

Struktury

Struktury (typy) danych składają się przede wszystkim z **pól** (ang. *fields* lub *member variables*), czasem zwanych zmiennymi lub stałymi składowymi.
Typ strukturalny reprezentuje logiczny iloczyn (koniunkcję, *i*) wartości reprezentowalnych przez typy swoich pól: wartość A *i* wartość B.

Struktury mogą też zawierać **funkcje składowe** (ang. *member functions* lub *methods*) czasem zwanych metodami.

Składniki (2)

Struktury

```
struct Cat {
    size_t number_of_lives { 9 };     // member variable (field)
    bool const brings_luck { false }; // member constant (field)

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

auto Cat::make_sound(bool const loud) const -> void
{
    std::cout << (loud ? "MEOW!!!" : "Meow!")
        << " I have " << number_of_lives << " lives left, and I "
        << (brings_luck ? "do" : "don't") << " bring luck.\n";
}
```

Pola

Struktury

Pola służą do przechowywania *danych*, które dany typ grupuje – na przykład rozmiar i zawartość wektora w `std::vector`.

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 { 9 };
```

Funkcje składowe

Struktury

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

Funkcja składowa ma dostęp zarówno do wartości przekazanych jej w argumentach, jak i do wartości pól obiektu, na którym jest wywoływana (patrz przykład kodu na slajdzie 14).

Funkcje składowe mogą też pełnić specjalną rolę, np. konstruktor lub przeładowany operator (patrz slajdy 33 i 31).

`class` vs `struct`

Struktury

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

```
struct S {
    bool now_you_see_me { true };

private:
    bool now_you_dont { false };
};

class C {
    bool you_dont_see_me { false };

public:
    bool now_you_do { true };
};
```

Widoczność pól

Struktury

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;
    }
}
```

Overview

Zmienne

Pola

Definicja zmiennego pola wygląda tak jak definicja każdej innej zmiennej, z tym wyjątkiem, że trzeba użyć starego stylu deklaracji (tj. bez `auto`).
Załóżmy, że chcemy stworzyć strukturę opisującą kota. Jak wiadomo, koty mają 9 żyć do wykorzystania:

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

Stałe

Pola stałe (stałe składowe)

Definicja stałego pola wygląda podobnie do definicji zmiennego pola. Trzeba dodać jedynie słowo kluczowe `const`. Kontynuując przykład z kotem:

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

    std::string const name;
};
```

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

[3] Oh, rly? Ciekawostka na slajdzie 26.

Inicjalizacja pól

Pola

Pola można zainicjalizować na kilka sposobów:

1. przypisując im wartość w liście inicjalizującej składowe (ang. *member initialiser list*) (patrz slajd 23)
2. przypisując im wartość w ciele konstruktora
3. pozostawiając ich wartości domyślne zdefiniowane w ciele struktury

Zmienne składowe można zainicjalizować na każdy z powyższych sposobów.
Stałe składowe można zainicjalizować jedynie w sposób 1. lub 3.

Inicjalizacja pól (c.d.)

Pola

```
struct Cat {
    /* fields of Cat omitted */

    Cat() = default; // default ctor
    Cat(std::string, unsigned const);
};

Cat::Cat(std::string n, unsigned const ll)
    : name{n}        // only member initialiser list for constants
    , lives_left{ll} // for variables you can use member initialiser list...
{
    lives_left = ll; //...or assignment in constructor's body
}
```

Jeśli definiujemy konstruktor to kompilator nie wygeneruje konstruktora domyślnego (z pustą listą parametrów).
Domyślny konstrukor można w takim wypadku zdefiniować ręcznie, lub wymusić jego automatyczne utworzenie za pomocą konstrukcji "`= default`".

Dostęp

Pola

Aby dostać się do pola struktury należy użyć operatora dostępu czyli . (kropka).
Jeśli pole jest zmienne, można je zmodyfikować lub przypisać mu całkiem nową wartość:

```
auto a_cat = Cat{};       // an ordinary cat
a_cat.lives_left -= 1;    // the cat died and lost a life
a_cat.lives_left  = 666; // now it's a cat from hell
```

Stałych pól nie można modyfikować, a kompilator w takim przypadku wygeneruje błąd:

```
auto mr_snuggles = Cat{ 9, "Mr. Snuggles" };
std::cout << mr_snuggles.name << "\n";
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

Funkcje składowe należy *zadeklarować* wewnątrz struktury, do której mają należeć.

```
struct Cat {
    // declaration, usually in header file: Cat.h
    auto make_sound(bool const) const -> std::string;
};

// definition, usually in implementation file: Cat.cpp
auto Cat::make_sound(bool const loud) const -> std::string
{
    /* ... */
}
```

Definicje funkcji składowych znajdują się (zazwyczaj) poza definicją struktury.

Wywoływanie funkcji składowych

Funkcje składowe

Wywoływanie funkcji składowych odbywa się podobnie jak wywoływanie funkcji wolnych (ang. *free function*), z tym wyjątkiem, że ich pierwszym parametrem jest obiekt, *na którym* są wywoływane. Obiekt ten pojawia się przed operatorem dostępu (tak jak w dostępie do pól).

```
auto a_cat = Cat{};
std::cout << a_cat.make_sound(false) << "\n";
```

"Ten" obiekt

Funkcje składowe

Funkcja składowa jako ukryty argument otrzymuje informację o tym *na jakim obiekcie* została wywołana.

Używając słowa kluczowego `this` można dostać się do wskaźnika do "tego" obiektu i jawnie go używać:

```
auto Cat::make_sound(bool const) const -> std::string
{
    return ("I have " + std::to_string(this->lives_left)
        + " lives left.");
}
```

Przeładowanie operatorów

Struktury

W C++ możliwe jest "przeładowanie operatorów" (ang. *operator overloading*) czyli zdefiniowanie dla struktur danych funkcji składowych wywoływanych np. przez operatory arytmetyczne + lub -.

```
struct Modulo_arithmetic {
    int const mod { std::numerical_limits<int>::max() };
    int value { 0 };

    Modulo_arithmetic(int v, int m): mod{m}, value{v} {}

    auto operator+ (Modulo_arithmetic const) const -> Modulo_arithmetic;
    auto operator< (Modulo_arithmetic const) const -> bool;
};
```

Definiowanie przeładowanych operatorów

Struktury

Funkcja składowa implementująca operator poza tym, że ma "śmieszną" nazwę niczym się nie różni od zwykłej funkcji składowej:

```
auto Modulo_arithmetic::operator+ (Modulo_arithmetic const o) const
    -> Modulo_arithmetic
{
    // FIXME what if the mods are different?
    return Modulo_arithmetic{(value + o.value) % mod, mod};
}

auto Modulo_arithmetic::operator< (Modulo_arithmetic const o) const
    -> bool
{
    return (value < o.value);
}
```

Konstruktor

Struktury

Konstruktor (ang. *constructor*) jest specjalną funkcją składową odpowiedzialną za "przygotowanie" struktury danych do użycia.

```
struct Hour {
    int value { 0 };

    explicit Hour(unsigned): value{v}
    {
        if (v > 23) { // throw if they don't make sense
            throw std::out_of_range{"hour value cannot exceed 23"};
        }
    }
};
```

Konstruktor może zasygnalizować niemożność utworzenia instancji struktury (np. z powodu niewłaściwych wartości argumentów) używając wyjątków[5].

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

Destruktor

Struktury

Destruktor (ang. *destructor*) jest specjalną funkcją składową odpowiedzialną za "zniszczenie" struktury danych. Jest uruchamiany w momencie tuż przed końcem czasu życia obiektu.

```
struct Network_connection {
    ~Network_connection()
    {
        shutdown(sock, SHUT_RDWR);
        close(sock);
    }
};
```

Destruktory definiuje się zazwyczaj dla typów mających "opakować" jakiś zasób, na przykład pamięć, uchwyt do pliku, lub połączenie sieciowe.

RAII

Struktury

RAII (ang. *Resource Acquisition Is Initialisation*) jest metodą zarządzania czasem życia obiektów[6] w języku C++, definiującą kiedy obiekty są tworzone, kopiowane, przenoszone, i niszczone. Gwarantuje ona automatyczne wywołanie konstruktora, destruktora, oraz innych funkcji zarządzających obiektami w odpowiednich momentach.

Jest to kluczowa sprawa dla zapewnienia poprawnego zarządzania zasobami oraz poprawności programu.

[6] język Rust zawiera podobny mechanizm przez swoje jawne śledzenia czasów życia – czyli lifetimes

"Rule of zero"

RAII

Zazwyczaj dla klasy definiuje się konstruktor, aby sterować tworzeniem jej wartości. Dla klas, które nie zarządzają zasobami zdefiniowanie konstruktora jest wystarczające i stosuje się "Rule of zero" czyli nie podawanie definicji żadnej z funkcji biorących udział w zarządzaniu czasem życia obiektów.

"Rule of five"

RAII

"Rule of five"[7] mówi natomiast, że jeśli zdefiniowana jest jedna z tych funkcji, to należy najprawdopodobniej zdefiniować wszystkie pięć:

1. konstruktor kopiujący `T(T const&)`
2. konstruktor przenoszący `T(T&&)`
3. kopiujący `operator=` `operator=(T const&) -> T&`
4. przenoszący `operator=` `operator=(T&&) -> T&`
5. destruktor `T()`

[7] `https://en.cppreference.com/w/cpp/language/rule_of_three`

"Rule of five" (c.d.)

RAII

```
class Network_connection {
    int sock { -1 };

    Network_connection(Network_connection const&) = delete;
    Network_connection(Network_connection&& o)
        : sock{std::move(o.sock)}
    {
        o.sock = -1;
    }

    auto operator=(Network_connection const&) = delete;
    auto operator=(Network_connection&& o)
    {
        sock = std::move(o.sock);
        o.sock = -1;
    }

    ~Network_connection()
    {
        shutdown(sock, SHUT_RDWR);
        close(sock);
    }
};
```

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`).

Zadanie: ctor i dtor

Zaimplementować strukturę, która w konstruktorze przyjmie wartość typu `std::string`, a w destruktorze wydrukuje ją na ekran poprzedzoną napisem `"DESTRUCTION!"`.

Kod źródłowy w pliku `src/s04-ctor-dtor.cpp`

Zadanie: this

Zaimplementować strukturę, która będzie miała funkcję składową drukującą na ekran wartość wskaźnika `this`. W funkcji `main()` wywołać tą funkcję na stworzonym obiekcie, oraz pobrać jego adres "ręcznie" operatorem `&` i porównać wyniki.

Kod źródłowy w pliku `src/s04-this.cpp`

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, destruktora
4. znać pojęcia takie jak RAII, Rule of zero, Rule of five

Zadania

Podsumowanie

Zadania znajdują się na slajdach 40, 41, 42, 43, 44, 45, i 46.