Instrukcja I

Wstęp

Współczesny C++

Niniejszy tekst rozpoczyna cykl instrukcji stanowiących kurs wprowadzający do języka C++. Język ten, pomimo swojego dojrzałego wieku, wciąż bardzo dynamicznie się rozwija. W chwili pisania tej instrukcji, jego najnowszą odsłonę stanowi niedawno zatwierdzony standard C++20. Komitet ISO, który go wydaje, pracuje obecnie w trzyletnim cyklu; dotychczasowe wydania standardu to C++98, C++03, C++11, C++14, C++17 i C++20 (następna planowana wersja to C++23). Wspominamy o tym, gdyż dobrze napisany kod w ww. języku wygląda dziś znacząco inaczej, niż kilkanaście lat temu. W czasie trwania zajęć postaramy się przedstawić czytelnikowi możliwie jak najbardziej współczesne elementy języka oraz schematy programowania. Postaramy się zaznaczać, w którym standardzie pojawił się dany element i dlaczego wyparł on ten wcześniej używany (lub jaką lukę wypełnia). Tym samym sugerujemy czytelnikowi zaopatrzyć się w kompilator wspierający możliwie jak najnowszy standard. Absolutne minimum to wsparcie dla C++11. Od tej edycji zwykło się mówić o "współczesnym C++" (ang. *modern C++*), gdyż fundamentalnie zmieniła ona paradygmat programowania w C++.

Dlaczego C++?

Na samym początku chcielibyśmy zaznaczyć do czego służy C++ oraz kiedy należy sięgnąć po inny język. Niewątpliwie największą jego siłę stanowi wydajność, zarówno w zakresie wykorzystywanej pamięci, jak i czasu wykonania programu. Pisząc w C++ sami zarządzamy pamięcią, zatem napisany program nigdy nie wykorzysta jej więcej, niż sobie tego zażyczymy - sami decydujemy o swoim losie. Dzięki dostępnym niskopoziomowym narzędziom oraz wyrafinowaniu optymalizatorów dostępnych we współczesnych kompilatorach możemy mieć duży stopień pewności, że dany algorytm zaimplementowany w C++ wykona się co najmniej tak samo szybko, jak w jakimkolwiek innym języku. Jedną z przyświecających twórcom tego języka maksym jest "*leave no room for a language between C++ and assembly*". Jednocześnie, C++ pozwala na wspięcie się na relatywnie wysoki poziom abstrakcji. Przez abstrakcję rozumiemy tutaj wyrażanie stosunkowo skomplikowanych operacji w zwięzły sposób. Zamiast opisywać niskopoziomowe operacje na bitach, wtajemniczenie w które wymaga czasu i skupienia, opisujemy interakcje między obiektami, których znaczenie widoczne jest na pierwszy rzut oka. Pozwala to na pisanie przejrzystego, łatwo sprawdzalnego kodu, co przyspiesza proces tworzenia oprogramowania. Z tego powodu często mówi się, że jedną z zalet C++ są "darmowe abstrakcje" (ang. *zero-cost abstractions*), tzn. abstrakcje, które nie pociągają za sobą kosztu w wydajności kodu.

C++ nie jest jednak magicznym narzędziem, które najlepiej nadaje się do wszystkiego. Pomimo wspomnianych abstrakcji, kod potrzebny do wykonania danego zadania będzie przeważnie dłuższy (w znaczeniu liczby linijek) od kodu napisanego w wysokopoziomowym języku, np. Pythonie. Jeżeli naszym celem jest napisanie szybko małego programu, którego czas wykonania wynosi kilka sekund, prawdopodobnie należy sięgnąć po inne narzędzie. Z tego powodu, często w dużych systemach w C++ napisane są kernele (jądra), a bardziej peryferyjna funkcjonalność (frontend, interfejsy) zaimplementowana jest w innym języku. C++ możemy współcześnie znaleźć np. w branżach takich jak automotive, aerospace i game development, w bazach kodu do obliczeń na superkomputerach (HPC), ale także w kodzie źródłowym dużych serwisów jak Facebook czy Google, w których, ze względu na skalę, nawet mikrooptymalizacje mogą pociągać za sobą milionowe oszczędności.

Zakres kursu

W trakcie zajęć nauczymy się podstaw języka C++ oraz zilustrujemy na jego przykładzie ideę programowania obiektowego. Położymy także nacisk na naukę dobrych praktyk i wytłumaczymy, czemu służą. Po ukończeniu niniejszego kursu, czytelnik powinien być w stanie samemu napisać proste programy, ale także potrafić poruszać się po bardziej skomplikowanych bibliotekach. Nie ukrywamy jednak, że kursowi temu daleko do bycia wyczerpującym. C++ stanowi obszerny temat, którego zgłębienie wymaga wiele czasu, wysiłku i przede wszystkim pracy nad własnym kodem (dlatego prawdopodobnie najbardziej rozwijającą częścią tego kursu są projekty domowe). Nadzieja autorów jest taka, że czytelnik, który w przyszłości potrzebował będzie wykorzystać ten język, posiadał będzie solidny fundament wiedzy, znał będzie "filozofię" programowania w C++ i wiedział będzie gdzie szukać zasobów do rozwijania swoich umiejętności.

Prerekwizytami do niniejszego kursu są:

- Znajomość języka C: zmienne, pętle, funkcje, zarządzanie pamięcią (czyli czym różni się stos od sterty, `malloc`, `free`) itp.
- Dostęp do kompilatora wspierającego standard C++11 (a najlepiej C++17). Instrukcje przygotowane są w sposób niezależny od platformy, ale sugestia autorów to gcc lub clang na systemie Linux i MSVC na systemie Windows

(dostępne przez IDE Visual Studio). Ostatnią deskę ratunku stanowią kompilatory online, wśród których króluje niewątpliwie Compiler Explorer.
- Dostęp do internetu w trakcie zajęć: wątpliwości co do standardu (oraz biblioteki standardowej) najlepiej wyjaśniać zaglądając do dokumentacji. Będzie ona także dostępna w czasie zaliczenia.

Zaznaczamy też, że, przystępując do laboratorium, czytelnik powinien być zaznajomiony z treścią odpowiedniego wykładu. Opisy zawarte w instrukcjach nie są wyczerpujące, stanowią one jedynie zwięzłe przypomnienie i mają za zadanie skupić uwagę czytelnika na najważniejszych aspektach omawianego zagadnienia.

Klasy

Pola i metody

Fundamentalnym pojęciem dla C++ i programowania obiektowego jest klasa. Definiując klasy oraz tworząc ich instancje (obiekty), możemy wyrazić operacje na bitach pamięci w sposób abstrakcyjny i zrozumiały dla człowieka. Klasy deklarujemy przy użycia słowa kluczowego `class` lub `struct`. Różnią się one jedynie tym, że domyślnie wszystkie pola klasy zadeklarowanej jako `class` są prywatne, a `struct` publiczne (co to dokładnie znaczy omówimy za chwilę).

```
class A
{
    // definicje i/lub deklaracje pól i metod, domyślnie prywatne
};

struct B
{
    // definicje i/lub deklaracje pól i metod, domyślnie publiczne
};
```

Zadeklarowawszy takie (puste) klasy, możemy stworzyć ich instancje:

```
int main()
{
    A obiekt_typu_A;
    B obiekt_typu_B;
}
```

Oczywiście aby nasze klasy były jakkolwiek użyteczne, musimy wyposażyć je w pola i/lub metody:

```
struct Human
{
    int    age;
    double height;
};
```

Możemy teraz stworzyć człowieka:

```
int main()
{
    Human Alice;
    Alice.height = 175.5;
    Alice.age    = 35;
}
```

Jak widzimy, powyższa klasa pozwoliła nam związać ze sobą parę parametrów w jeden obiekt. Pracując nad nim (np. podając go do funkcji), oszczędzamy czas, gdyż nie musimy za każdym razem mówić o parze parametrów, ale o jednym tworze. Takie struktury były już dostępne w C. W C++ klasy mogą mieć także metody:

```
#include <iostream>
struct Human
{
    void printAge() { std::cout << age << '\n'; }

    int    age;
    double height;
};

int main()
{
    Human Alice;
    Alice.height = 175.5;
    Alice.age    = 35;
    Alice.printAge();
}
```

Zadanie 1 Napisz klasę `Wektor2D`, która przechowuje (jako publiczne zmienne) współrzędną x i y dwuwymiarowego wektora. Dodaj do niej metodę `norm`, zwracającą normę wektora, oraz `print`, drukującą (w ładnym formacie) jego współrzędne.

Konstruktory i destruktory

Szczególne typy metod to konstruktory i destruktory. Konstruktory to metody służące do tworzenia obiektów. Klasa może mieć dowolną liczbę konstruktorów (rozróżnianych typami podawanych argumentów, dokładnie tak samo jak przeciążalibyśmy każdą inną funkcję). Destruktor to metoda wywoływana przy niszczeniu obiektów danej klasy.

```
#include <iostream>
struct Human
{
    Human(int a, double h, std::string n)
    {
        age    = a;
        height = h;
        name   = n;
        std::cout << "Hello, " << name << "!\n";
    }
    ~Human()
    {
        std::cout << "Goodbye, " << name << "...\n";
    }

    int         age;
    double      height;
    std::string name;
};

int main()
{
    Human Alice(35, 175.5, "Alice");
}
```

Nieco niuansów inicjalizacji Zauważmy, że teraz próba konstrukcji `Human Alice;` spowodowałaby błąd kompilacji. Dzieje się tak dlatego, że w przypadku, w którym nie zdefiniujemy żadnego konstruktora, kompilator spróbuje za nas stworzyć konstruktor domyślny (tzn. taki bez argumentów). Ponieważ zdefiniowaliśmy `Human(int, double, std::string)`, kompilator nie doda konstruktora domyślnego. Nieco więcej o tym i o tzw. "Rule of 5" powiemy w następnej instrukcji. Warto też tutaj powiedzieć, że niestety z przyczyn historycznych składnia konstrukcji w C++ jest dość zagmatwana (zainteresowanych szczegółami odsyłamy np. do tego wykładu). Poniższe inicjalizacje zmiennej typu `int` są dokładnie równoważne:

```
int a1 = 0; // Nie, tutaj nie ma przypisania, int od razu inicjalizowany jest
int a2(0);  // Tutaj bez niespodzianki, ale co się stanie, gdy zawołamy domyśl
int a3{0};  // Od C++11 preferujemy nawiasy klamrowe!
```

Współcześnie, silnie preferujemy inicjalizację (konstrukcję) przy pomocy nawiasów klamrowych. Wynika to między innymi z faktu, że wołanie konstruktora domyślnego (tzn. tego bez argumentów) przy pomocy nawiasów okrągłych skutkuje dość nieoczekiwanym zachowaniem, tzw. *most vexing parse*.

```
struct Human
{
    Human() { age = 0; height = 40.; name = "Nameless"; }
    // Reszta jak wyżej...
};

int main()
{
    Human no_name1{}; // Tutaj tworzymy obiekt no_name1 przy użyciu konstrukto
    Human no_name2;   // Jak  wyżej
    Human no_name3(); // Tutaj deklarujemy funkcję no_name3, która przyjmuje (
                      // Most vexing parse!
}
```

Powodem tego zjawiska jest to, że w C++ wszystko, co może tylko być deklaracją, jest interpretowane jako deklaracja. Inicjalizacja nawiasami klamrowymi może w pewnym przypadku sprawić kłopoty, o czym powiemy na zajęciach dotyczących kontenerów (kłopoty wynikają ze szczególnych zasad dotyczących `std::initializer_list`). Póki co starajmy się jednak wyrabiać dobre nawyki i pozostańmy przy podawaniu argumentów konstruktorów wewnątrz `{ }`.

Zadanie 2 Dodaj do klasy `Wektor2D` konstruktor dwuargumentowy, który nadaje wartości współrzędnym wektora, a następnie je drukuje. Napisz destruktor, który

także drukuje tę informację. Stwórz kilka różnych wektorów. W jakiej kolejności są one niszczone? W którym miejscu w kodzie następuje destrukcja?

Zadanie 3 Napisz klasę `Informer`, która posiada konstruktor domyślny, drukujący informację o konstrukcji, oraz destruktor, drukujący informację o destrukcji. Dodaj do klasy `Wektor2D` pole typu `Informer`. Które destruktory wołane są przy zniszczeniu wektora? W jakiej kolejności? Zastanów się, jakie ma to implikacje dla komponowania większych obiektów z mniejszych obiektów.

Modyfikatory dostępu

Wszystkie pola i metody, z których dotychczas korzystaliśmy, były publiczne, tzn. mieliśmy do nich dostęp spoza klasy (z funkcji `main`). Często możemy jednak chcieć zablokować dostęp do części pól i/lub metod jakiejś klasy. Postawmy się na przykład w pozycji autora/autorki biblioteki do sprawdzania pogody. W tego typu bibliotece gdzieś musi znaleźć się funkcjonalność do komunikacji z serwerem, a następnie interpretowania danych, które od niego otrzymamy. Jednak nie chcemy, aby użytkownik naszej biblioteki musiał ten proces widzieć ani rozumieć. Wolimy, żeby użytkownik wołał po prostu np.

```
#include "biblioteka_pogodowa.h"
int main()
{
    Godzina g{"17:00"};
    Pogoda  p{godzina};
    p.getFromServer();
    p.print();
}
```

Czynimy zatem publicznymi konstruktor oraz metody `getFromServer` i `print`. Natomiast wszystkie inne metody przez nie wołane (np. te służące do komunikacji z serwerem) czynimy prywatnymi (metody mogą wołać wszystkie inne metody danej klasy, w tym te prywatne). Klasa `Pogoda` ma więc dość mały interfejs. Dzięki temu jest łatwa w użyciu oraz trudna do nadużycia. Jak wiadomo duża część pracy programistycznej to pisanie kodu w taki sposób, aby był "idiotoodporny".

Często spotykanym podejściem jest pisanie tzw. "setterów" i "getterów". Polega ono na czynieniu pól klasy prywatnymi i dodawaniu publicznych funkcji `setX` i `getX`. W ten sposób zabezpieczamy się przed nieumyślną zmianą danej wartości. W naszym przypadku mogłoby wyglądać to następująco:

```
class Human
{
public:
    void setAge(int a) { age = a; }
    int  getAge()      { return age; }
    // ...

private:
    int    age;
    double height;
};
```

Zadanie 4 Dodaj do klasy `Wektor2D` metody `setX`, `getX`, `setY` i `getY`, służące do odczytywania i modyfikowania współrzędnych wektora. Uczyń pola opisujące współrzędne prywatnymi. Co stanie się, gdy spróbujesz zawołać np. `std::cout << wektor.x;`?

Przeciążanie operatorów

W języku C++ operatory (tu znajdziesz ich listę) możemy przeciążać tak samo jak wszystkie inne funkcje. Spójrzmy na przykład:

```
class Human{ /* ... */ };
struct Couple
{
    Couple() {}
    Couple(Human p1, Human p2) { person1 = p1; person2 = p2; } // Uwaga: Human
    Human person1;
    Human person2;
};

Couple operator+(Human h1, Human h2)
{
    return Couple{h1, h2};
}

int main()
{
```

```
    Human Alice{35, 175.5, "Alice"};
    Human Sam{34, 174, "Sam"};
    Couple c;
    c = Alice + Sam;
}
```

Pozwala nam to często na pisanie bardziej ekspresyjnego kodu poprzez definiowanie abstrakcyjnych operacji przy pomocy znanych nam intuicyjnie operatorów (`&&`, `+`, `*`, itd.). Więcej na temat szczególnego operatora `=` powiemy na kolejnych zajęciach.

Zadanie 5 Przeciąż operatory `+` i `*` tak, aby były zdefiniowane dla klasy `Wektor2D` (zgodnie z tradycyjną algebrą).

Zadanie 6 Przeciąż operator `<<` tak, aby można było zawołać `std::cout << wektor;`. Następnie przeciąż go tak, aby można było zawołać `std::cout << wektor1 << wektor2 /* << ... */ << wektorn`.

Pola i metody statyczne

Dotychczas definiowaliśmy pola i metody, które operowały na konkretnym obiekcie danej klasy (np. imię jest indywidualną cechą każdego człowieka). Czasem przydatne mogą być także pola i metody statyczne, czyli zdefiniowane dla całej klasy, nie dla jej poszczególnych instancji. Przyjrzyjmy się przykładowi:

```
struct Human
{
    static       int n_humans;
    static const int n_human_heads = 1;

    Human()  { ++n_humans; }
    ~Human() { --n_humans; }
};

int Human::n_humans = 0;
```

Dzięki odpowiedniej definicji konstruktora i destruktora, pole `n_humans` pozwala nam śledzić, ile ludzi żyje w danym momencie wykonania programu. Dodatkowo, pole `n_human_heads` pozwala nam na zapisanie, ile głów ma człowiek (jest to cecha wspólna wszystkich ludzi). W różnych miejscach kodu możemy odnosić się do tej wielkości, a jeżeli na późniejszym etapie pracy zdecydujemy się, że chcemy, aby gatunek ludzki w naszym programie miał jednak więcej głów, wystarczy zmienić jedną wartość. Taki sposób pisania kodu pozwala nam zaoszczędzić pracy oraz uniknąć potencjalnych błędów.

Zadanie 7 Dodaj do klasy `Wektor2D` prywatne pole `num_wek` typu całkowitego, które przechowuje bieżącą liczbę istniejących obecnie wektorów. Zainicjalizuj je zerem. Zmodyfikuj odpowiednio konstruktory i dodaj odpowiedni destruktor. Stwórz w `main`ie kilka wektorów, z których część umieścisz w dodatkowym zagnieżdżonym scope'ie (bloku nawiasów `{}`). Zweryfikuj swoją pracę wyświetlając wartość pola `num_wek` na różnych etapach wykonania programu (lub podglądając ją w debuggerze).

Statyczne mogą być także funkcje (metody). W ostatnim zadaniu nic nie stoi na przeszkodzie, abyśmy ręcznie zmodyfikowali zmienną reprezentującą liczbę istniejących wektorów. Naprawmy ten potencjalny problem, wykorzystując fakt, że metody statyczne mają dostęp do prywatnych pól i metod danej klasy (zarówno tych statycznych jak i nie).

Zadanie 8 Uczyń pole `num_wek` prywatnym. Dodaj do klasy `Wektor2D` publiczną, statyczną funkcję o sygnaturze `int populacja()`, która zwraca wartość pola `num_wek`. Zmodyfikuj odpowiednio kod w `main`ie.

Bardzo istotny przykład wykorzystania metod statycznych stanowi wrapper na konstruktor (wrapper to w programowaniu określenie na funkcjonalność, która zawija jedynie inną funkcjonalność, nie dodając zbyt wiele od siebie). Chcielibyśmy teraz zdefiniować dodatkowy konstruktor klasy `Wektor2D`, który przyjmie współrzędne wektora w układzie biegunowym. Konstruktor taki przyjmuje oczywiście 2 liczby zmiennoprzecinkowe, ma zatem sygnaturę `Wektor2D(double, double)`. Niestety, taki konstruktor został już przez nas zdefiniowany. Jak zatem obejść ten problem i umożliwić inicjalizację wektorów w obu układach odniesienia?

Zadanie 9 Uczyń konstruktor wektora o sygnaturze `Wektor2D(double, double)` prywatnym. Napisz publiczną, statyczną metodę `Wektor2D kart(double, double)`, która tworzy wektor na podstawie podanych współrzędnych w układzie kartezjańskim. Teraz dodaj kolejną publiczną, statyczną metodę `Wektor2D`

`bieg(double, double)`, która tworzy wektor na podstawie podanych współrzędnych w układzie biegunowym (musisz skonwertować je do układu kartezjańskiego). Zmodyfikuj odpowiednio kod w `main`ie (wszystkie wektory tworzone bezpośrednio muszą teraz być tworzone przez zawołanie odpowiedniej metody statycznej). Zweryfikuj, czy konwersja współrzędnych z jednego układu współrzędnych na drugi przebiegła (matematycznie) poprawnie. Parę uwag do zadania:

- Funkcje trygonometryczne znajdziesz w nagłówku `#include <cmath>`. Wszystkie nagłówki wykorzystywane w języku C zostały przeniesione do C++ w sposób `nazwa_nagłówka.h` → `cnazwa_nagłówka`.
- Domyślny konstruktor wektora może pozostać publiczny. Punkt (0, 0) pokrywa się w obu układach współrzędnych, nie ma tu dwuznaczności.

Pytania na koniec

- Czym jest `std::cout` (do jakiej kategorii bytów należy)? Jaki ma scope ("zasięg istnienia")?
- Z jakiego konstruktora klasy `std::string` korzystaliśmy w klasie `Human`?
- Czy klasa może mieć więcej niż 1 destruktor? Dlaczego?
- Na ile sposobów możemy zdefiniować `operator+` dla klasy `Wektor2D`? W razie wątpliwości zajrzyj tutaj.
- W którym miejscu programu rozpoczynają swoje życie obiekty statyczne? W którym je kończą?