Pamięć alokowana dynamicznie w języku C++. Wprowadzenie do klas i obiektów

W tym odcinku zamiast dodawać do projektu plik z rozszerzeniem .c, należy obowiązkowo dodać plik o rozszerzeniu .cpp (np. main.cpp). Spowoduje to użycie kompilatora C++ zamiast C, początkowo różnice między tymi językami będą prawie niezauważalne, ale pod koniec tego artykułu pojawi się to, czego nie ma język C, a mianowicie klasy i obiekty.

Zaczniemy zupełnie nieobiektowo, od naszej ulubionej struktury danych, jaką jest Student:

```
struct Student
{
	int nralb;
	char imie[100];
};

int main()
{
	Student s;
	printf("nralb: %d, imie: %s\n", s.nralb, s.imie);
	return 0;
}
```

Wyniki:

```
nralb: -858993460, imie: ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠ł ↕
```

1) Ponieważ używamy C++, nie jest konieczne każdorazowe dodawanie słowa kluczowego struct, wystarczy sama nazwa typu strukturalnego (Student).
2) Podobnie jak w C, zmienna lokalna nie ma określonej wartości początkowej, należy więc zadbać o jej stan początkowy samodzielnie.
3) Warto przy okazji zdefiniować specjalizowaną funkcję do wyświetlania dowolnego studenta, aby w przyszłości, wraz ze zwiększaniem liczby pól struktury, nie było konieczne modyfikowanie wszystkich miejsc w kodzie, gdzie chcemy Studenta wyświetlić.

```
void Student_ustaw(Student &s, int nralb, char imie[100])
{
	s.nralb = nralb;
	strcpy(s.imie, imie);
}

void Student_wyswietl(Student s)
{
	printf("nralb: %d, imie: %s\n", s.nralb, s.imie);
}
```

```
void main()
{
     Student s;
     Student_ustaw(s, 12345, "Jasiu");
     Student_wyswietl(s);
}
```

Wyniki:

```
nralb: 12345, imie: Jasiu
```

Lepiej, ale nie do końca.

Zwróć uwagę na referencję w funkcji Student_ustaw(…). Jest to trzeci sposób przekazywania argumentów do funkcji. W C używaliśmy sposobu „przez wartość” oraz „przez wskaźnik”. C++ wprowadza bardzo przydatny mechanizm „przez referencję”. Używamy go wtedy, gdy chcemy aby funkcja mogła bezpośrednio modyfikować przekazaną do niej zmienną, a nie tylko jej kopię.

To, co na pewno będzie wymagało poprawienia, to przekazywanie parametru `char imie[100]` do funkcji Student_ustaw(…). Nie jest też dobrym pomysłem użycie `Student_wyswietl(Student s)`– należałoby raczej posłużyć się wskaźnikiem albo referencją, zmniejszyłoby to liczbę kopiowanych bajtów i pozwoliłoby zaoszczędzić pamięć stosu oraz przyspieszyłoby działania kodu. Sama struktura Student też nie jest zbyt dobrze zaprojektowana, pole imie zawsze zjamuje 100 bajtów niezależnie od tego, czy długość imienia wynosi 0, 10 czy 99 znaków. Co gorsza, gdybyśmy chceli użyć imienia dłuższego niż 99 znaków to nie będzie to możliwe w tak zdefiniowanej strukturze. Lepiej byłoby użyć wskaźnika znakowego, zaś pamięć potrzebną do przechowania imienia można alokować dynamicznie, w dokładnie takiej ilości, jaka jest potrzebna, ani mniej, ani więcej.

Dynamiczna alokacja pamięci narzuca też na nas obowiązek zwalnianie tej pamięci, gdy już nie będzie potrzebna. Pojawi się więc kolejna funkcja Student_usun(…), która zadba o to.

```
struct Student
{
     int nralb;
     char *imie;
};

void Student_inicjalizuj(Student &s)
{
     s.nralb = 0;
     s.imie = NULL;
}

void Student_ustaw(Student &s, int nralb, char *imie)
{
     s.nralb = nralb;
```

```
    if(s.imie != NULL)     //jeśli student już posiada imię,
        free(s.imie);      //to najpierw się go pozbywamy,
    s.imie = strdup(imie); //a następnie dynamicznie tworzymy
    //duplikat (kopię) przekazanego argumentu wywołania funkcji
}

void Student_wyswietl(Student *s)
{
    printf("nralb: %d, imie: %s\n", s->nralb, s->imie);
}

void Student_usun(Student *s)
{
    printf("zwalniam pamiec studenta o nralb=%d\n", s->nralb);
    if(s->imie != NULL)
        free(s->imie);
}

void main()
{
    Student s;
    Student_inicjalizuj(s);
    Student_ustaw(s, 12345, "Jasiu");
    Student_wyswietl(&s);
    Student_usun(&s);
}
```

W języku C++, w odróżnieniu od C, istnieje mechanizm **przeciążania (przeładowania) funkcji**. Nie jest konieczne nazywanie tych funkcji rozwlekle – zamiast Student_inicjalizuj(s); można krócej napisać inicjalizuj(s);. Nawet wtedy, gdy w naszym programie zdefiniujemy inną strukturę (np. struct Browar {…}), i też zechcemy mieć funkcję inicjalizującą zmienne tego typu, to możemy śmiało zapisać to tak:

```
struct Browar {
    char *marka;
    double cena, procenty;
};

void inicjalizuj(Browar &b)
{
    b.marka = NULL;
    b.cena = b.procenty = 0.0;
}

void inicjalizuj(Student &s)
{
    s.nralb = 0;
    s.imie = NULL;
}

void main()
{
    Student s;
```

```
        Browar b;

        inicjalizuj(s);
        inicjalizuj(b);
        …
}
```

Mamy 2 funkcje o identycznej nazwie inicjalizuj(…), różniące się jednak typem argumentów. To w zupełności wystarczy, aby kompilator C++ poradził sobie z odróżnianiem tych funkcji.

Wyraźnie oddzielenie definicji typu danych (struktury) od algorytmów które ich używają (funkcji) niekoniecznie najlepiej odzwierciedla rzeczywistość. Wydaje się, że bardziej naturalne byłoby przypisanie studentowi nie tylko cech ilościowych (nr albumu, wiek, imię, nazwisko, oceny…), ale także zdefiniowanie umiejętności, jakie student powinien posiadać (zakuwanie, imprezowanie, sprzątanie po sobie). Od strony czysto praktycznej wygodniej by też było, gdy zmienne same potrafiły się inicjalizować oraz zwalniać pamięć automatycznie (wtedy, gdy już nie są nam one więcej potrzebne).

Takie możliwości daje nam użycie klasy zamiast struktury. Podstawowa różnica między strukturą a klasą jest taka, że oprócz pól w tej drugiej można definiować także funkcje (w tym przypadku będziemy je nazywać metodami).

```
class Student
{
public:
    int nralb;
    char *imie;

    void ustaw(int nralb, char *imie)
    {
        this->nralb = nralb;
        if(this->imie != NULL)
            free(this->imie);
        this->imie = strdup(imie);
    }

    void wyswietl()
    {
        printf("nralb: %d, imie: %s\n", this->nralb, this->imie);
    }

    void usun()
    {
        printf("zwalniam pamiec studenta o nralb=%d\n", nralb);
        if(imie != NULL)
            free(imie);
    }

    void inicjalizuj()
    {
```

```
            nralb = 0;
            imie = NULL;
      }
};


void main()
{
      Student s;
      s.inicjalizuj();
      s.ustaw(9999, "Malgosia");
      s.wyswietl();
      s.usun();
}
```

Wyniki:

```
nralb: 9999, imie: Malgosia
zwalniam pamiec studenta o nralb=9999
```

Zdecydowanie lepiej! Czy można usprawnić nasz kod jeszcze bardziej? To, co najbardziej się może przydać, to umiejętność automatycznego ustawiania własnych pól i automatyczne zwalnianie pamięci, którą zajęliśmy. Służą do tego konstruktory i destruktor. Każda klasa w C++ może mieć 0 lub więcej konstruktorów i 0 lub 1 destruktor.

```
class Student
{
public:
…
      Student() //konstruktor
      {
            printf("inicjalizacja pol obiektu klasy Student\n");
            nralb = 0;
            imie = NULL;
      }
      ~Student() //destruktor
      {
            printf("zwalniam pamiec studenta o nralb=%d\n", nralb);
            if(imie != NULL)
                  free(imie);
      }
};

void main()
{
      Student s;
      s.ustaw(9999, "Malgosia");
      s.wyswietl();
}
```

Wyniki:

```
inicjalizacja pol obiektu klasy Student
nralb: 9999, imie: Malgosia
zwalniam pamiec studenta o nralb=9999
```

A gdyby tak jeszcze bardziej ułatwić życie programiście? W klasie możemy zdefiniować dodatkowy konstruktor następującej postaci:

```
class Student {
public:
     …
     Student(int nralb, char *imie) //b. podobne do metody ustaw(…)
     {
          printf("tworze obiekt klasy Student (%d, %s)\n", nralb, imie);
          this->nralb = nralb;
          this->imie = strdup(imie);
     }
};
void main()
{
     Student s(9999, "Malgosia"); //krócej i ładniej się nie da
     s.wyswietl();
}
```

Wyniki:

```
tworze obiekt klasy Student (9999, Malgosia)
nralb: 9999, imie: Malgosia
zwalniam pamiec studenta o nralb=9999
```

c.d.n.

Zadanie na dzisiaj

1) zbadaj zachowanie konstruktorów i destruktora. W którym momencie jest wywoływany każdy z nich?

2) Mamy dany poniższy fragment kodu. Co zostanie wyświetlone na ekranie? Jaki napis pojawi się jako pierwszy? A jaki jako ostatni? DLACZEGO?

```
Student s(9999, "Malgosia");
void main()
{
     printf("Witamy w funkcji main()!\n");
     s.wyswietl();
}
```

3) Idąc tropem z punktu (2), zbadaj zachowanie aplikacji, kiedy masz do dyspozycji więcej zmiennych obiektowych różnych typów (różnych klas). Co decyduje o kolejności wykonania konstruktorów? Aby zrealizować ten podpunkt, przygotuj kompletną klasę inną niż Student{}, może to być np. class SokOwocowy{} i utwórz kilkoro Studentów oraz kilka SokówOwocowych.

4) zmodyfikuj dotychczasowy proceduralny kod swojej aplikacji z Lab08 tak, aby użyć klasy zamiast struktury i zbioru funkcji.

5) Soki owocowe są na tyle zdrowe, że warto nauczyć Studenta aby pił je (z umiarem) nawet kilka razy w ciągu dnia. Pomyśl, w postaci jakich danych można reprezentować obiekt klasy Studenta z plecakiem wypełnionym obiektami klasy SokOwocowy? Co jeszcze może student nosić w plecaku? Czy są to tylko napoje? Jak najogólniej można zdefiniować klasę, która będzie pasowała do dowolnego elementu wyposażenie studenta?

6) Celem tego zadania jest zapoznanie się z pojęciem „model obiektowy" lub „programowanie zorientowane obiektowo". Zamodeluj obiektowo wycinek rzeczywistości, jakim jest ekipa dobrze wyposażonych studentów wraz z opiekunami na najbliższym Rajdzie.