Zawartość:

- struktury i klasy jako foremki na dane

- unie

- tablice 2D

Struktury i Klasy

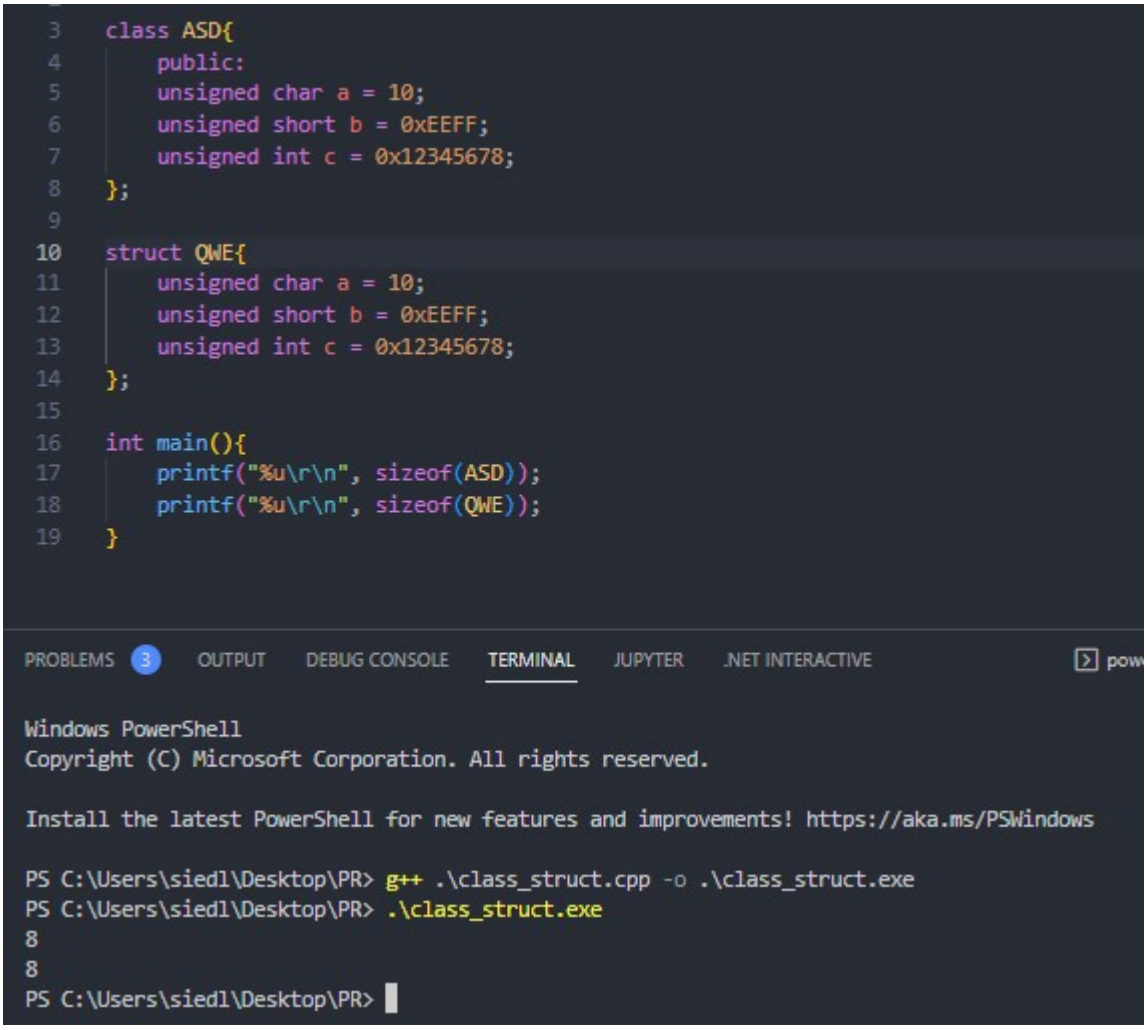
Sztandarowe pytanie z programowania brzmi następująco: czym różni się klasa od struktury?

W C++ można powiedzieć, że niczym. Jedyna mało znacząca różnica polega na tym, że w strukturze wszystko jest domyślnie publiczne a w klasie prywatne.

W C# struktury są typami wartościowymi, a klasy typami referencyjnymi. Tutaj różnica jest ogromna.

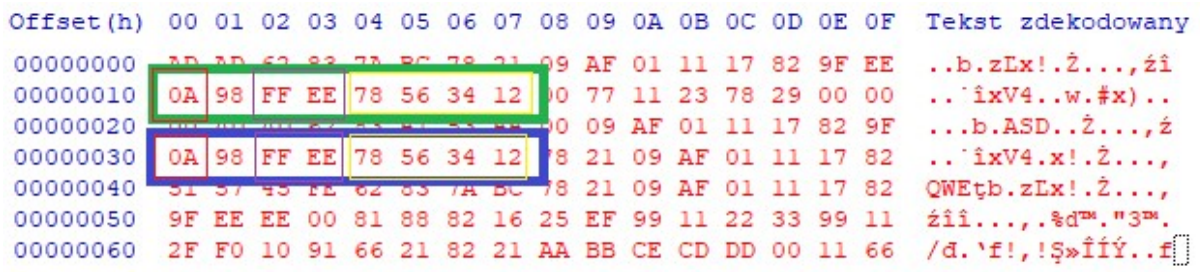
W Javie i Pythonie nie mamy struktur.

Chciałbym poruszyć temat struktur i klas w kontekście „foremki" na dane. Czyli traktujemy je tutaj jako zbiór pól bez logiki. Takie klasy stosuje się np. jako model do ORM w przypadku baz relacyjnych.



To co można zauważyć na powyższym obrazku to rozmiar obu „foremek". 1 bajt na chara, 2 bajty na short, 4 bajty na inta daje nam 7 a na ekranie wyświetliło się 8. Wynika to z tego, że kompilatory lubią umieszczać dane na parzystych adresach. Jeżeli char jest umieszczony na offsecie 0 to short umieszczony jest nie na 1 a na dwa a int na offsecie 4. Wynika to z tego, że

procesory piszą i czytają szybciej z parzystych adresów. Oczywiście da się to wyłączyć. Zobaczmy jak to wygląda w pamięci:



Na zielono zaznaczono klasę, na niebiesko strukturę. Kolorami zaznaczono wszystkie trzy pola. Jak widać struktura i klasa nie przechowuje w sobie nic poza tymi zmiennymi, które w niej umieściliśmy.

Jeżeli dodamy metody, dziedziczenie, polimorfizm to w klasie/stukturze pojawi się jeszcze coś takiego jak wskaźnik na vTable czyli tablicę metod. Jest to wskaźnik funkcje przypisane do danej klasy/struktury.

ASD a;

ASD* a_ptr = &a; //a_ptr ma adres 0x00000010

unsigned char* a_a_ptr = &a.a; //a_a_ptr ma adres 0x00000010

Wniosek jest prosty. Adres całej struktury/klasy == adres pierwszego pola. Działa to tak samo jak w tablicach. Kastowanie wskaźnika na inne typy może nam pozwolić na wyciąganie różnych wartości ze środka zielonego i niebieskiego prostokąta.

Co ważne. Z punktu widzenia procesora i pamięci nie ma czegoś takiego jak pole prywatne. Nie da się czegoś ukryć w pamięci. To, że coś jest publiczne albo prywatne jest informacją tylko i wyłączeni na czas pisania kodu.

```
3   class ASD{
4       private:
5       unsigned char a = 10;
6       unsigned short b = 0xEEFF;
7       unsigned int c = 0x12345678;
8   };
9
10  > struct QWE{ ···
15
16  int main(){
17      ASD a;
18      printf("%u\r\n", *((unsigned char*)&a));
19  }
```

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER   .NET INTER

PS C:\Users\siedl\Desktop\PR> .\class_struct.exe
10
```

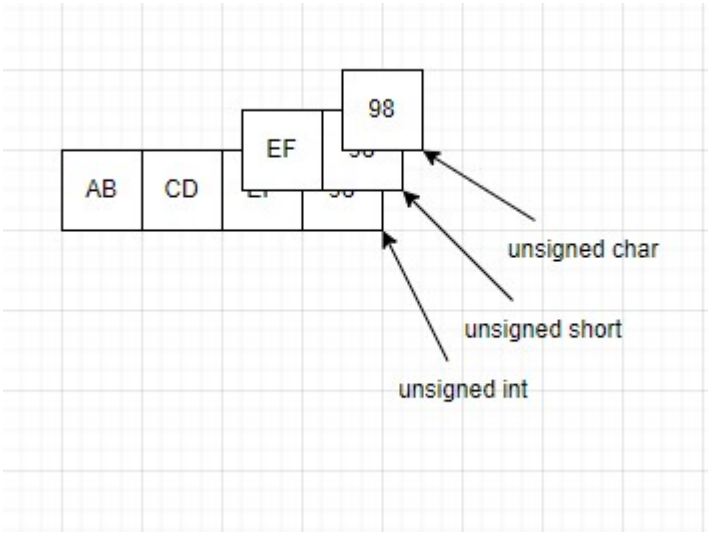
Unia

Unia to struktura unikalna dla C/C++. Idea tej struktury polega na tym, iż wszystkie typy nakładają się na siebie.

```
16  union Unia{
17      unsigned char a;
18      unsigned short b;
19      unsigned int c;
20  };
21
22  int main(){
23      Unia unia;
24      unia.c = 0xABCDEF98;
25      printf("%X\r\n", unia.a);
26      printf("%X\r\n", unia.b);
27      printf("%X\r\n", unia.c);
28  }
```

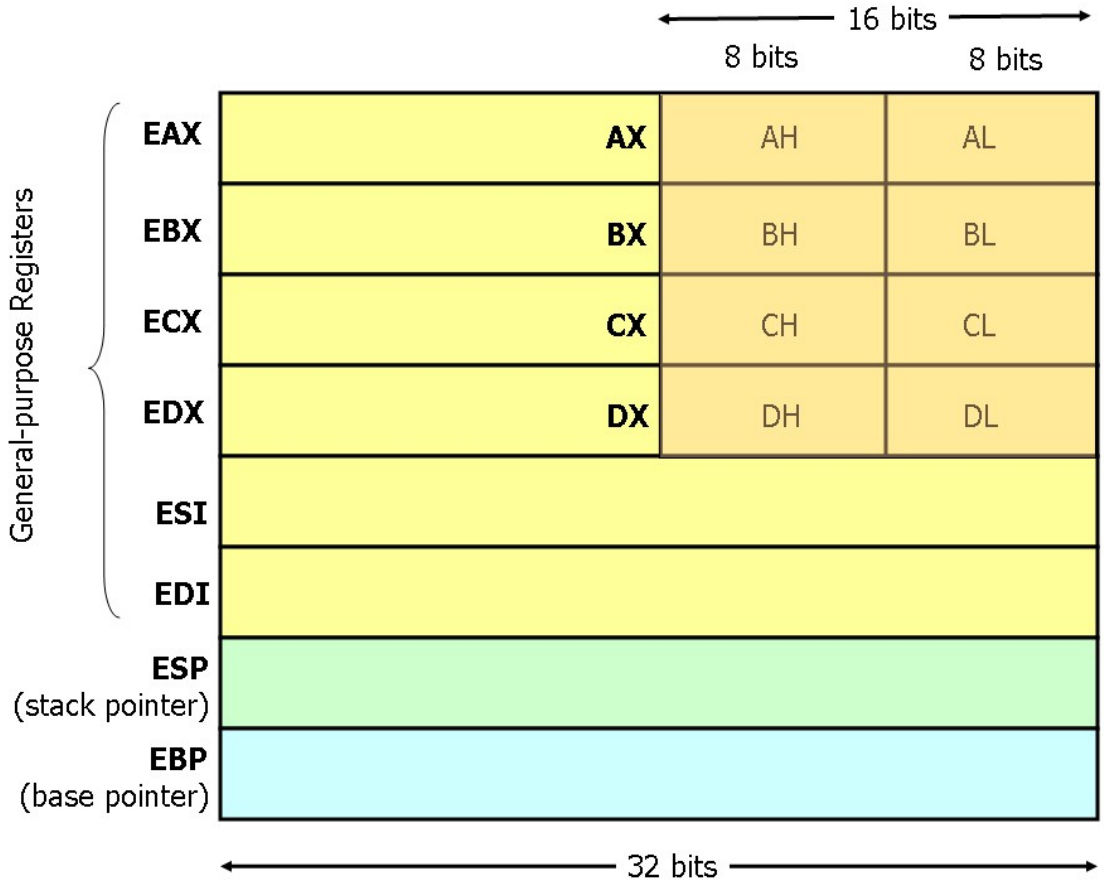
```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER   .NET INTERACTIVE

PS C:\Users\siedl\Desktop\PR> g++ .\class_struct.cpp -o .\class_struct.exe
PS C:\Users\siedl\Desktop\PR> .\class_struct.exe
98
EF98
ABCDEF98
PS C:\Users\siedl\Desktop\PR>
```

Skąd się to wzięło spróbuje wyjaśnić tym rysunkiem.



Unia przyjmuje rozmiar największego typu wewnętrznego. Wszystkie typy są na siebie nakładane wyrównując do LSB.

Ustawiając pole C ustawiamy wszystkie zmienne. Ustawiając A, zmieniamy tylko LSB w B oraz C. **Dla unii raz mi się udało znaleźć sensowne zastosowanie gdy implementowałem emulator procesora i rejestrów.**



W rejestrach jest tak, że EAX ma 4 bajty. AX to dolne dwa bajty z tego EAX. AH to górny bajt AX a AL to dolny bajt AX.

Tablice statyczne i dynamiczne

Tablica statyczna to tablica, której rozmiar znany jest w momencie kompilacji.

Tablica dynamiczna to tablica, której rozmiar jest różny i zależny od przebiegu programu lub tego co zrobi użytkownik.

```
int tablica_statyczna[10];

int* tablica_dynamiczna = new int[10];
delete[] tablica_dynamiczna;
```

Co to znaczy, że rozmiar tablicy statycznej musi być znany w czasie kompilacji. Oznacza to, że musi być stałą a nie zmienną:

```
#define SIZE 5

int main(){
    int size = 10;
    int tablica_statyczna[size]; //error - to nie jest stała czasu kompilacji

    const int size1 = 20;
    int tablica_statyczna1[size1]; //ok

    int tablica_statyczna2[SIZE]; //ok - to jest najpopularniejsze rozwiazanie
```

Makro OK, const OK, ale zmienna NIE OK. Proste

Tablice dynamiczne są alokowane dynamicznie na stercie i trzeba jest zwalniać. Rozmiar może być dowolny (no chyba, że braknie sterty).

```
#define SIZE 5

int main(){
    int size = 10;
    int tablica_statyczna[size]; //error - to nie jest stała czasu kompilacji

    const int size1 = 20;
    int tablica_statyczna1[size1]; //ok

    int tablica_statyczna2[SIZE]; //ok - to jest najpopularniejsze rozwiazanie

    int* tablica_dynamiczna = new int[size];
    int* tablica_dynamiczna1 = new int[size1];
    int* tablica_dynamiczna2 = new int[SIZE];
```

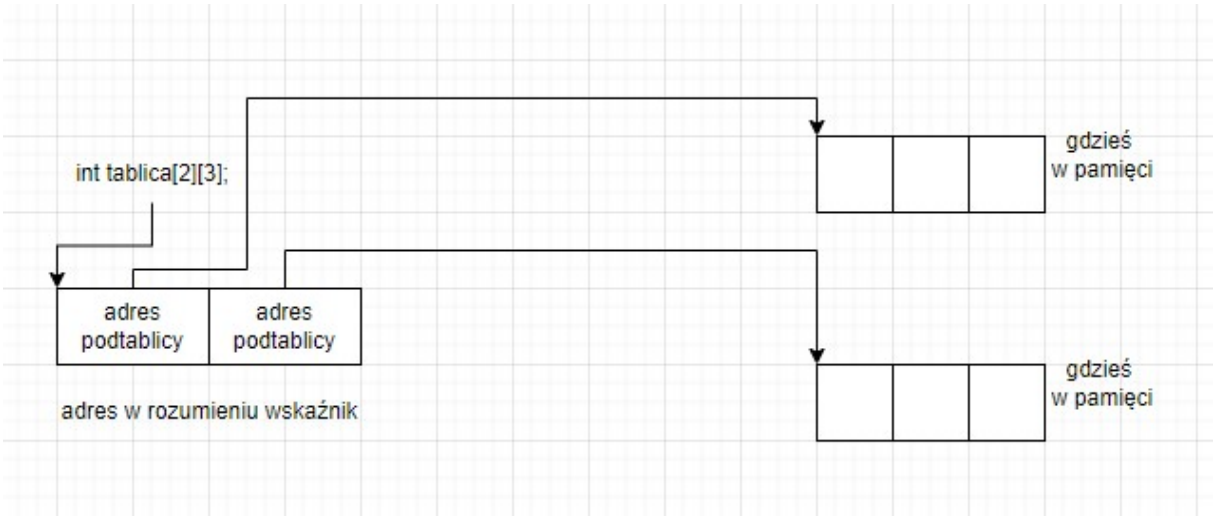
W C#, Pythonie, Javie każda tablica jest dynamiczna. Takie obiekty są typami referencyjnymi i nie trzeba ich usuwać (nawet niezbyt jest jak), ponieważ zajmuje się tym GC.

Tablica 2D

Tablica 2D – najlepiej to opisać jako tablica wskaźników na tablice. Prościej się nie da. Zdanie to ma sens nawet dla 10D. Po prostu robi się nam taki łańcuch wskaźników.

Statyczna tablica:

unsigned int tablica[2][3]; //tablica dwóch wskaźników na dwie tablice, które mają po trzy inty



Dynamiczna tablica:

```
int** tablica = new int*[2];
for(int i = 0;i<2;i++){
    tablica[i] = new int[3];
}
```

Alokujemy tablicę na dwa wskaźniki na inta.

Później alokujemy każdą pod tablicę, a wskaźnik przypisujemy do komórek nadrzędnej tablicy.

Mówiąc prościej na bazie powyższego schematu. Alokujemy tablicę po lewej. A później te po prawej. Alokując tablice po prawej wskaźniki na nie zapisujemy w tablicy po lewej.

Kasowanie pamięci zaalokowanej w ten sposób przebiega odwrotnie. Najpierw w pętli robimy delete[] tablica[i], a na końcu delete tablicy nadrzednej.

Dynamiczne tablice 2D mają przewagę nad statycznymi polegającą na tym, iż pozwalają alokować podtablice o różnym rozmiarze. W przykładzie każdy new w pętli alokuje 3 elementy, ale nie musi tak być.

W C# tablice 2D alokujemy na jeden z dwóch sposobów:

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Ważne by pamiętać, że podtablica jest typem referencyjny, więc ewentualne kopie trzeba wykonać na głęboko.