Uniwersytet Bielsko-Bialski

Wydział budowy maszyn i informatyki

Architektura komputerów

(Laboratorium №2)

Temat ćwiczenia: Systemy kodowania liczb

Data wykonania ćwiczenia: 08.11.2023

Igor Gawłowicz

Zadanie 1

W poniższym programie, przeanalizuj adres zmiennej tmp dla różnych typów danych, w tym: char, int, float, double, __m128i (związany z technologią SIMD SSE) i __m256i (związany z technologią SIMD AVX). Kod ten jest próbą sprawdzenia dopasowania adresu zmiennej "tmp" do różnych granic, które są potęgami dwójki (2 do potęgi "i", gdzie "i" przyjmuje wartości od 0 do 7). Dla każdej iteracji pętli, obliczamy granicę, pobieramy adres zmiennej "tmp", a następnie sprawdzamy, czy ten adres jest podzielny przez badaną granicę bez reszty.

```
#include <iostream>
#include <emmintrin.h>
#include <immintrin.h>
int main() {
    float tmp; // Deklarujemy zmienną "tmp" typu float
    // Pętla dla wartości "i" od 0 do 7
    for (int i = 0; i < 8; ++i) {
        int granica = 1 << i; // Obliczamy granicę jako potęgę dwójki (2 do potęgi
"i")
        // Pobieramy adres zmiennej "tmp" i zapisujemy jego wartość w zmiennej
"adres"
        long long adres = (long long)&tmp;
        // Wypisujemy informacje o adresie i badanej granicy
        std::cout << "Czy adres: ";
        std::cout << adres;
        std::cout << " jest dopasowany do granicy ";
        std::cout << granica;
        std::cout << " bajtów? ";
        std::cout << std::endl;
        // Sprawdzamy, czy adres jest dopasowany do badanej granicy
        if (adres % granica) {
            std::cout << "Adres nie jest dopasowany do badanej granicy " <<
granica << " bajtów.";
        }
        else {
            std::cout << "Adres jest dopasowany do badanej granicy " << granica <<
" bajtów.";
        }
        std::cout << std::endl;
        std::cout << std::endl;
    }
    return 0;
}
```

Wyniki dla typu `float`

```
Czy adres: 140727845299020 jest dopasowany do granicy 1 bajtów?
Adres jest dopasowany do badanej granicy 1 bajtów.

Czy adres: 140727845299020 jest dopasowany do granicy 2 bajtów?
Adres jest dopasowany do badanej granicy 2 bajtów.

Czy adres: 140727845299020 jest dopasowany do granicy 4 bajtów?
Adres jest dopasowany do badanej granicy 4 bajtów.

Czy adres: 140727845299020 jest dopasowany do granicy 8 bajtów?
Adres nie jest dopasowany do badanej granicy 8 bajtów.

.............................................................
```

Wynii dla typu `int`

```
Czy adres: 140735299315564 jest dopasowany do granicy 1 bajtów?
Adres jest dopasowany do badanej granicy 1 bajtów.

Czy adres: 140735299315564 jest dopasowany do granicy 2 bajtów?
Adres jest dopasowany do badanej granicy 2 bajtów.

Czy adres: 140735299315564 jest dopasowany do granicy 4 bajtów?
Adres jest dopasowany do badanej granicy 4 bajtów.

Czy adres: 140735299315564 jest dopasowany do granicy 8 bajtów?
Adres nie jest dopasowany do badanej granicy 8 bajtów.

.............................................................
```

Wyniki dla typu `double`

```
Czy adres: 140734915953176 jest dopasowany do granicy 1 bajtów?
Adres jest dopasowany do badanej granicy 1 bajtów.

Czy adres: 140734915953176 jest dopasowany do granicy 2 bajtów?
Adres jest dopasowany do badanej granicy 2 bajtów.

Czy adres: 140734915953176 jest dopasowany do granicy 4 bajtów?
Adres jest dopasowany do badanej granicy 4 bajtów.

Czy adres: 140734915953176 jest dopasowany do granicy 8 bajtów?
Adres jest dopasowany do badanej granicy 8 bajtów.

Czy adres: 140734915953176 jest dopasowany do granicy 16 bajtów?
```

```
Adres nie jest dopasowany do badanej granicy 16 bajtów.

.............................................................
```

Wyniki dla typu `char`

```
Czy adres: 140734381101919 jest dopasowany do granicy 1 bajtów?
Adres jest dopasowany do badanej granicy 1 bajtów.

Czy adres: 140734381101919 jest dopasowany do granicy 2 bajtów?
Adres nie jest dopasowany do badanej granicy 2 bajtów.

.............................................................
```

Na podstawie powyższych wyników widzimy zależnośc wyniku obliczeń od typu danych. Dzieje się tak ponieważ w języku c++ dostęp do pamięci oraz zarządzanie wskaźnikami zależy od typu zmiennej więc zmieniając zmienną, która pośredniczy w procesie działania programu wpływamy na zużycie zasobów i tym zamym na zwracany wynik w programie.

Zadanie 2

W poniższym programie została zdefiniowana struktura, która zawiera 3 pola:

- int a; (rozmiar 4B)
- double b; (rozmiar 8B)
- int c; (rozmiar 4B) Sprawdź jaki rozmiar zostanie zarezerwowany w pamięci w oparciu o obiekt struktury składającej się z powyższych pól:
- w pierwszej kolejności spróbuj wyznaczyć samodzielnie rozmiar struktury sumując rozmiar poszczególnych jej elementów (4B+8B+4B),
  Na pierwszy rzut oka struktura powinna zajmować 16B
- a następnie uruchom poniższy program i sprawdź rzeczywisty rozmiar jaki struktura tmpData zarezerwuje w pamięci.

```
#include <iostream>
struct tmpData {
    int a;
    double b;
    int c;
};
int main() {
    std::cout << sizeof(tmpData) << std::endl;
    return 0;
}
```

W rzeczywistości jednak zajmuje 24B, jest to spowodowane tym w jaki sposób działa język c++, gdyż pomimo tego że struktura oryginalnie powinna zajmować te 16B program przyrównuje sobie pamięć do największej w

puli czyli w tym przypadku do typu double, który sam w sobie zajmuje 8B, żeby na wszelki wypadek nie zabrakło pamięci dlatego w teorii int zajmuję 4B ale w powyższym kontekście zajmuje ich 8 tak samo jak double co daje nam 24B.

Spróbuj zmienić kolejność występujących po sobie pól struktury, np.:

```
struct tmpData {
    int a;
    int c;
    double b;
};
```

Tym razem ilość uzyskanej pamięci wynosi 16B jest to spowodowane tym że jeśli największa zmienna w puli jest pierwszą lub ostatnią w kolejności program nie ma potrzeby, żeby stosować bajtów wyrównujących, więc rozmiar w rzeczywistości i w teorii zostaje taki sam.

Zmień kolejność pól składowych w strukturze tmpData tak aby jej obiekt zajmowała jak najmniejszą przestrzeń w pamięci operacyjnej komputera.

```
struct tmpData {
    char a;
    double b;
    int c;
    double d;
    char e;
};
int main() {
    std::cout << sizeof(tmpData) << std::endl;
    return 0;
}
```

Aby uzyskać najmniejszą ilosc bajtów w pamięci musimy ułożyć alokacje zmiennych w taki sposób aby konkretne typy danych były alokowane jeden za drugim, najlepiej w sposób taki żeby zacząć od największych typów lub najmniejszych i iść dalej rosnąco lub malejąco.

Wstępnie powyższa struktura zajmuje 40B ponieważ program wyrównał wszystkie do wielkości typu double. Gdy ułożymy zmienne rosnącą lub malejąco jak poniżej

```
// opcja a
struct tmpData {
    char a;
    char e;
    int c;
    double b;
    double d;
};
// opcja b
```

```
struct tmpData {
    double b;
    double d;
    int c;
    char a;
    char e;
};
```

To wynik będzie wynosił 24B ponieważ teraz program nie miał potrzeby stosowania bajtów wyrównujących

Wnioski

Podczas laboratorium, kluczowym wnioskiem jest zrozumienie wpływu typów danych i wyrównań w pamięci na zarządzanie zasobami w programowaniu w języku C++. W eksperymencie z zadania 1 obserwowaliśmy, że różne typy danych mają różne rozmiary i wyrównania, co wpływa na adresowanie w pamięci. W zadaniu 2 dostrzegliśmy, że rozmiar struktury może przekraczać sumę rozmiarów jej pól z powodu wyrównań. Warto podkreślić, że optymalizacja pamięci może być osiągnięta poprzez odpowiednią kolejność pól w strukturze. Ogólnie rzecz biorąc, te doświadczenia podkreślają znaczenie starannego wyboru typów danych i uwzględniania wyrównań pamięci podczas projektowania i optymalizacji programów komputerowych.