| Akademia Nauk Stosowanych w Nowym Sączu<br>Programowanie Współbieżne i Rozproszone | | | |
|---|---|---|---|
| **Temat: Pomiar czasu za pomocą biblioteki chrono** | | | |
| **Nazwisko i imię: Ciapała Tadeusz** | | **Ocena sprawozdania** | **Zaliczenie:** |
| | | | |
| **Data wykonania ćwiczenia: 07.03.2023** | **Grupa: P3** | | |

## 1) KOD 3:

Na zajęciach pracowaliśmy z biblioteką C++ o nazwie “chrono”

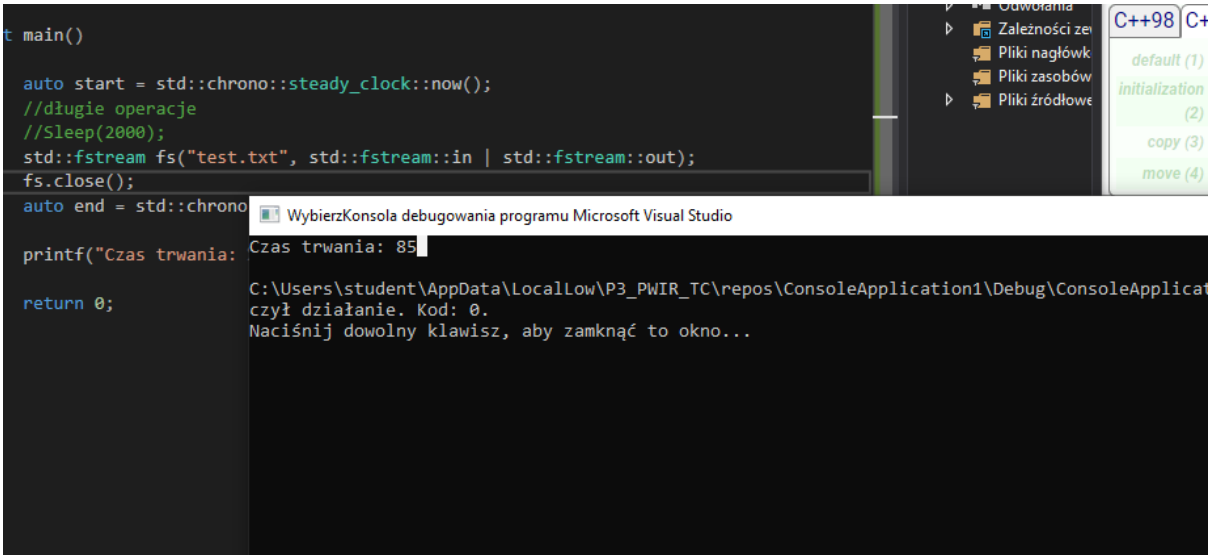
- https://en.cppreference.com/w/cpp/chrono

Biblioteka ta umożliwia wiele operacji na czasie. W naszym przypadku była ona potrzebna do wykonania pomiarów czasu trwania kodów z podanych zadań.

```
#include <chrono>
#include <cstdio>
#include <windows.h>

int main(){
    auto start = std::chrono::steady_clock::now();
    //długie operacje
    Sleep(2000);
    auto end = std::chrono::steady_clock::now();

    printf("Czas trwania: %llu\n",
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count());

    return 0;
}
```

*Listing nr 1: kod nr 3*

Naszym zadaniem na początek było sprawdzenie ile czasu zajmują operacje otwarcia i zamknięcia pliku. Do tego używaliśmy biblioteki fstream. Warto tu zwrócić uwagę na to, że na współczesnych komputerach czas wykonania takich operacji jest prawdopodobnie niezwykle niski, tj. mniejszy niż 10 milisekund. Jednak pierwsze uruchomienie kodu, który sprawdzał te operacje trwało dłużej niż każde następne, w przypadku których licznik pokazywał 0 milisekund. Prawdopodobnie miało to związek ze środowiskiem IDE.

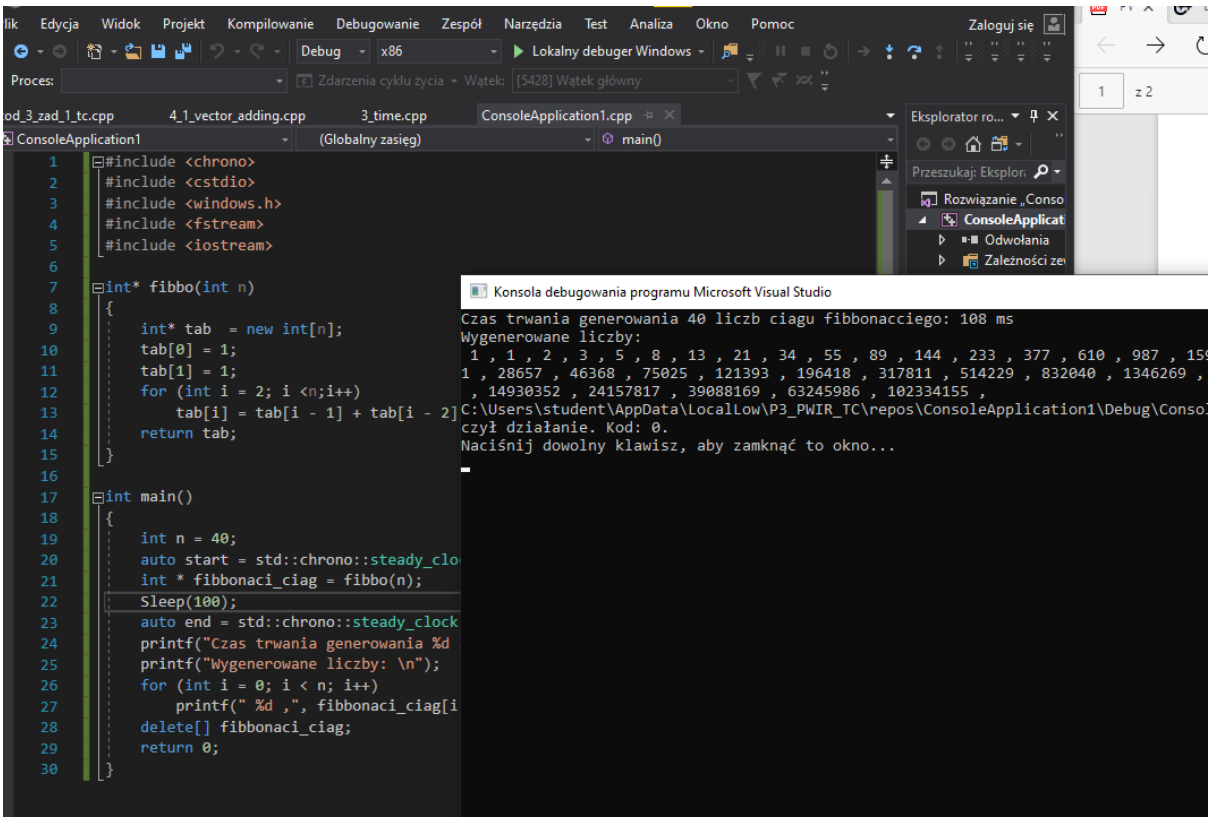



*Zrzut nr 1: test operacji otwarcia i zamknięcia pustego pliku*

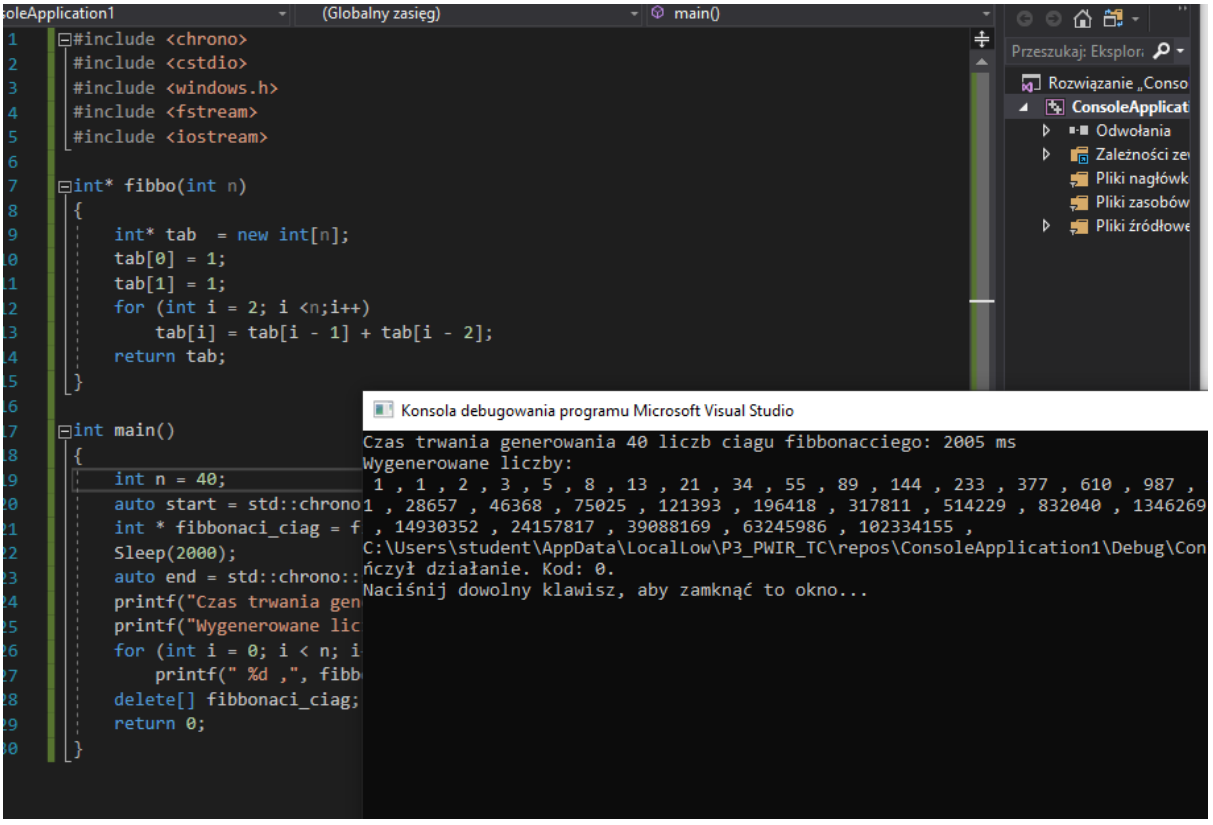
```
#include <chrono>
#include <cstdio>
#include <windows.h>
#include <fstream>
#include <iostream>

int main()
{
	std::fstream fs;
	auto start = std::chrono::steady_clock::now();
	fs.open("test.txt", std::fstream::in | std::fstream::out |
std::fstream::app);
	fs.close();
	auto end = std::chrono::steady_clock::now();
	printf("Czas trwania: %llu ms\n",
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count());
	return 0;
}
```

*Listing nr 2: wykonanie zadania 1*

Drugi zadaniem było sprawdzenie ile czasu zajmie wygenerowanie 40 elementów ciągu fibbonacciego. Współczesny komputer nie miał żadnych problemów z takim zadaniem. Tutaj trwało to prawdopodobnie mniej niz 1 milisekunda. Problemem z generowaniem liczb ciągu fibbonacciego jest ich przetrzymywanie w pamięci. Dla większej ilości wyrazów dochodzi do przepełnienia typu int i zakłamania liczonych informacji. Pomóc może zmiana typu int na unsigned int lub na jeszcze wieksze, inne typy. Najlepszy rozwiązaniem jest napisanie własnej metody do przechowywania w pamięci większych liczb naturalnych.




*Zrzut nr 2: generowanie liczb ciągu fibbonaciego*




*Zrzut nr 3: testy czasu ciągu liczb fibbonaciego - po dodaniu Sleep na 2 sekundy do kodu, program zarejestrował jeszcze 5 milisekund*

```
#include <chrono>
#include <cstdio>
#include <windows.h>
#include <fstream>
#include <iostream>

int* fibbo(int n)
{
	int* tab = new int[n];
	tab[0] = 1;
	tab[1] = 1;
	for (int i = 2; i < n; i++)
		tab[i] = tab[i - 1] + tab[i - 2];
	return tab;
}

int main()
{
	int n = 40000;
	auto start = std::chrono::steady_clock::now();
	int * fibbonaci_ciag = fibbo(n);
	Sleep(2000);
	auto end = std::chrono::steady_clock::now();
	printf("Czas trwania generowania %d liczb ciagu fibbonacciego: %llu
ms\n", n, std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count());
	printf("Wygenerowane liczby: \n");
	for (int i = 0; i < n; i++)
		printf(" %d ,", fibbonaci_ciag[i]);
	delete[] fibbonaci_ciag;
	return 0;
}
```

*Listing nr 3: wykonanie zadania 2*

**2) KOD 4:**

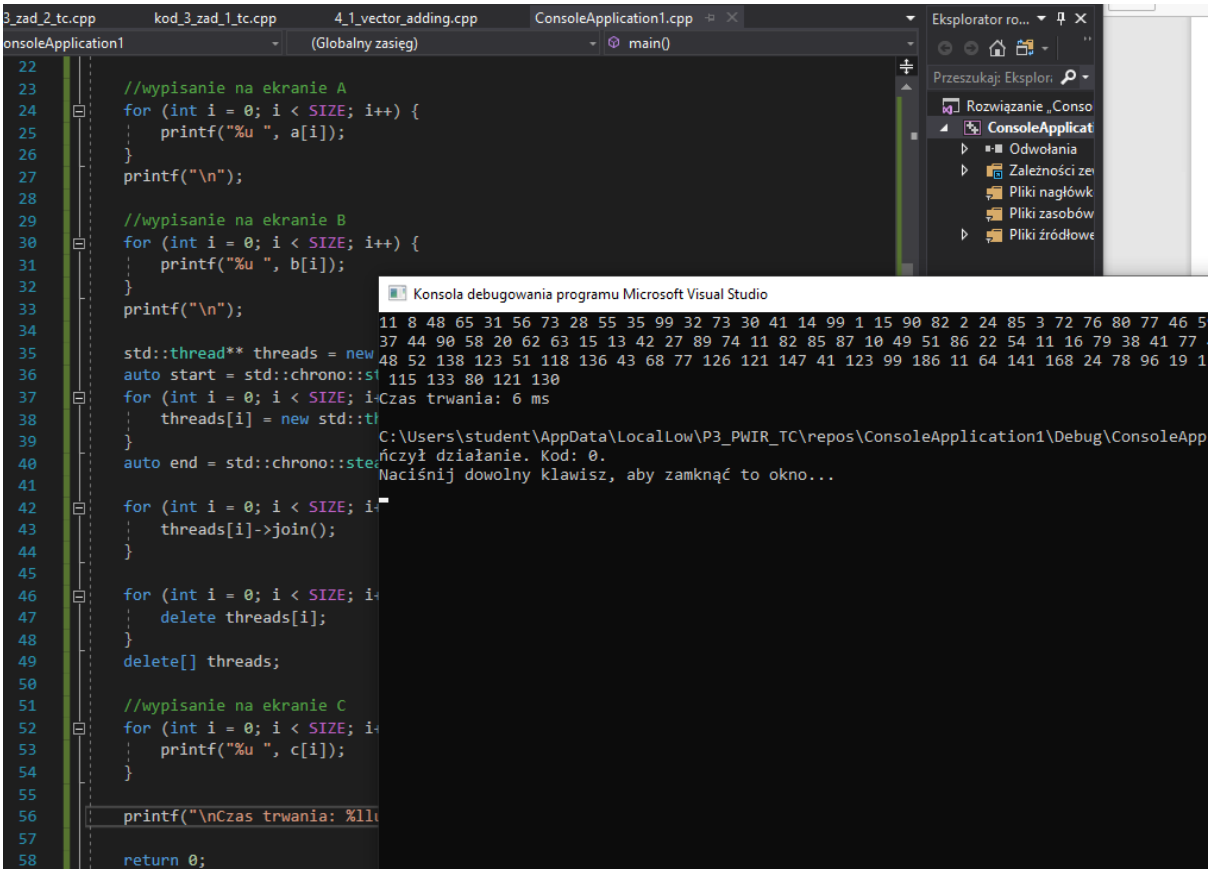
Następny kod przedstawiony na zajęciach miał za zadanie tworzyć tablice o ustalonym rozmiarze. Kolejną rzeczą było ich wypełnienie losowymi liczbami naturalnymi z zakresu od 1 do 100. Program otwierał tyle pobocznych wątków, ile było elementów tablicy (tutaj na początku ustalone było 100 elementów). Wątki wykonywały dodawanie liczb, gdzie każdy wątek miał nadzór nad odpowiadającym mu elementem (równe indeksy).

```
#include <cstdio>
#include <cstdlib>
#include <time.h>
#include <thread>

#define SIZE 40

void add(int id, int* a, int* b, int* c){
    c[id] = a[id] + b[id];
}

int main(){
    srand(time(NULL));
    int a[SIZE];
    int b[SIZE];
    int c[SIZE];
    for(int i = 0;i<SIZE;i++){
```

```
        a[i] = rand() % 100 + 1; //1 do 100
        b[i] = rand() % 100 + 1;
    }

    //wypisanie na ekranie A
    for(int i = 0;i<SIZE;i++){
        printf("%u ", a[i]);
    }
    printf("\n");

    //wypisanie na ekranie B
    for(int i = 0;i<SIZE;i++){
        printf("%u ", b[i]);
    }
    printf("\n");

    std::thread** threads = new std::thread*[SIZE];
    for(int i = 0;i<SIZE;i++){
        threads[i] = new std::thread(add, i, a, b, c); //wykorzystuje i
jako id danego wątku
    }

    for(int i = 0;i<SIZE;i++){
        threads[i]->join();
    }

    for(int i = 0;i<SIZE;i++){
        delete threads[i];
    }
    delete[] threads;

    //wypisanie na ekranie C
    for(int i = 0;i<SIZE;i++){
        printf("%u ", c[i]);
    }

    return 0;
}
```

Listing nr 4: kod nr 4

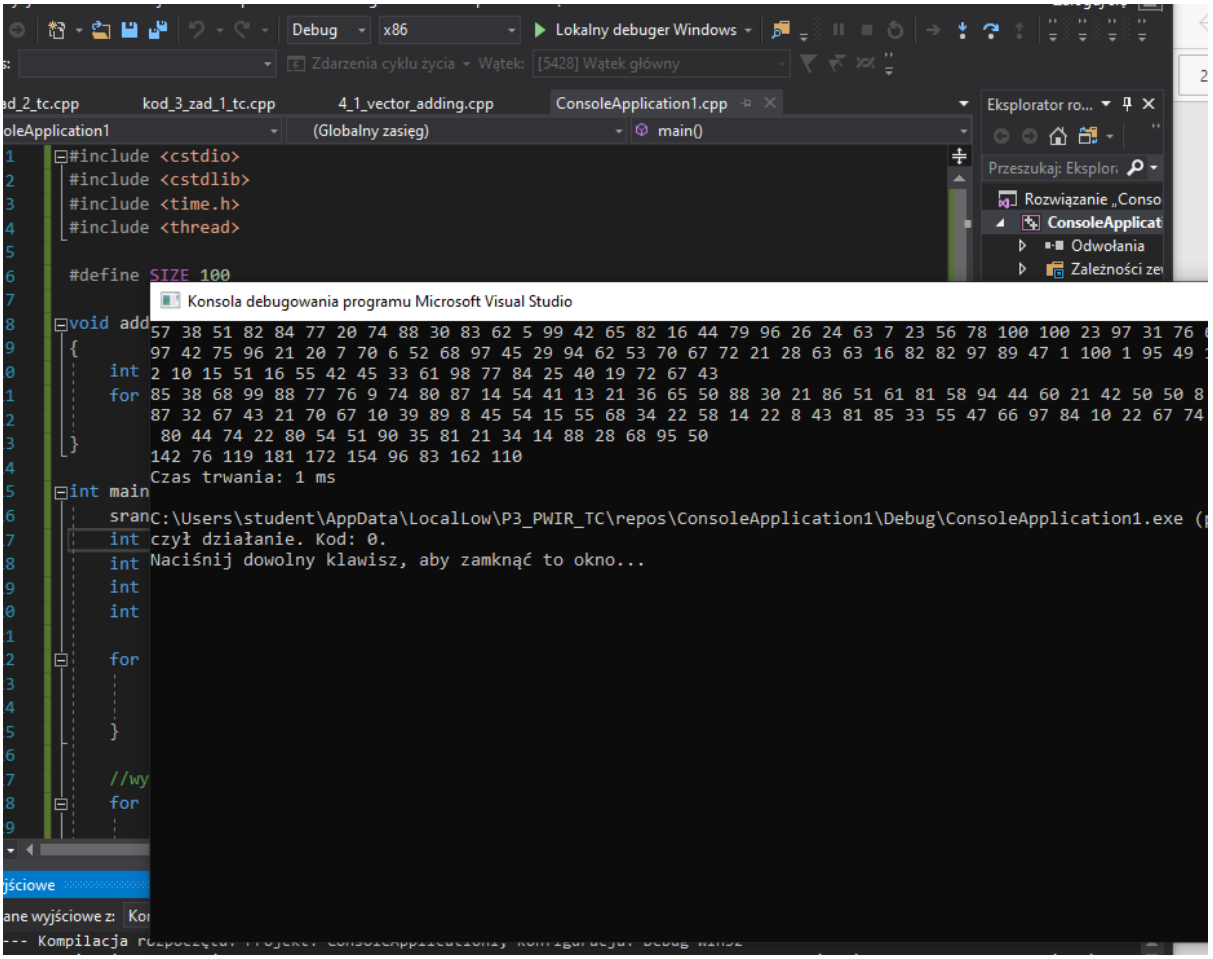
Pierwszym zadaniem było dodanie pomiaru czasowego, czyli wykorzystanie kodu nr 3. należało go dodać po linii z zadeklarowanym wskaźnikiem “threads”.



Zrzut nr 4: mierzenie tylko dodawania (wywołań funkcji add())

Drugim zadaniem była modyfikacja tego kodu w taki sposób, aby było tylko 10 wątków i dany wątek liczył indeksy w przedziałach:

- wątek o id 0 dodaje indeksy 0 z 0, 1 z 1, … , 9 z 9
- wątek o id 1 dodaje indeksy 10 z 10, 11 z 11, …, 19 z 19
- według tego schematu należy kontynuować aż dojdziemy do wątku o id 9, który liczył indeksy 90 z 90, 91 z 91, …, 99 z 99.




Zrzut nr 5: podział dodawania na 10 wątków

Widzimy różnicę w czasie trwania dodawania - po wykonaniu zadania 2 do kodu 4 program “jednocześnie” liczy na 10 wątkach po 10 indeksów, co jest znacznie wydajniejsze niż tworzenie pojedynczych wątków na pojedynczy indeks.

```
#include <cstdio>
#include <cstdlib>
#include <time.h>
#include <thread>

#define SIZE 100

void add(int id, int* a, int* b, int* c)
{
	int i = id * 10;
	for (int j = i; j < i + 10; j++)
		c[j] = a[j] + b[j];
}

int main() {
	srand(time(NULL));
	int tmp = SIZE / 10;
	int a[SIZE];
	int b[SIZE];
	int c[SIZE];
```

```
	for (int i = 0; i < SIZE; i++) {
		a[i] = rand() % 100 + 1; //1 do 100
		b[i] = rand() % 100 + 1;
	}

	//wypisanie na ekranie A
	for (int i = 0; i < SIZE; i++) {
		printf("%u ", a[i]);
	}
	printf("\n");

	//wypisanie na ekranie B
	for (int i = 0; i < SIZE; i++) {
		printf("%u ", b[i]);
	}
	printf("\n");

	std::thread** threads = new std::thread*[10];
	auto start = std::chrono::steady_clock::now();
	for (int i = 0; i < tmp; i++) {
		threads[i] = new std::thread(add, i, a, b, c); //wykorzystuje
i jako id danego wıtku
	}
	auto end = std::chrono::steady_clock::now();

	for (int i = 0; i < tmp; i++) {
		threads[i]->join();
	}

	for (int i = 0; i < tmp; i++) {
		delete threads[i];
	}
	delete[] threads;

	//wypisanie na ekranie C
	for (int i = 0; i < tmp; i++) {
		printf("%u ", c[i]);
	}

	printf("\nCzas trwania: %llu ms\n",
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count());

	return 0;
}
```

Listing nr 5: kod, który jest rozwiązaniem do zadania 1 i 2 dla kodu nr 4

Ostatnim zadaniem było postawione pytanie, co trzeba zmienić, aby rozmiar tablic można było pobierać od użytkownika i dlaczego coś trzeba zmienić. Odpowiedź jest następująca: pobranie danych od użytkownika powinno odbywać się zwyczajnie, czyli np. poprzez funkcję “scanf_s()” bądź obiekt “cin”. Natomiast problem występuje zależnie, gdzie te dane deklarujemy. Jeżeli są tworzone na stosie, to dostaniemy wiadomość od kompilatora, że nasze zmienne nie są stałe i kompilator wyrzuci błąd. Jednak, gdy zostaną one stworzone i umieszczone na stercie, to problem ten znika, a program może korzystać sprawnie z danych podanych przez użytkownika. Wynika to z tego, w jaki sposób działa stos i sterta w programie.