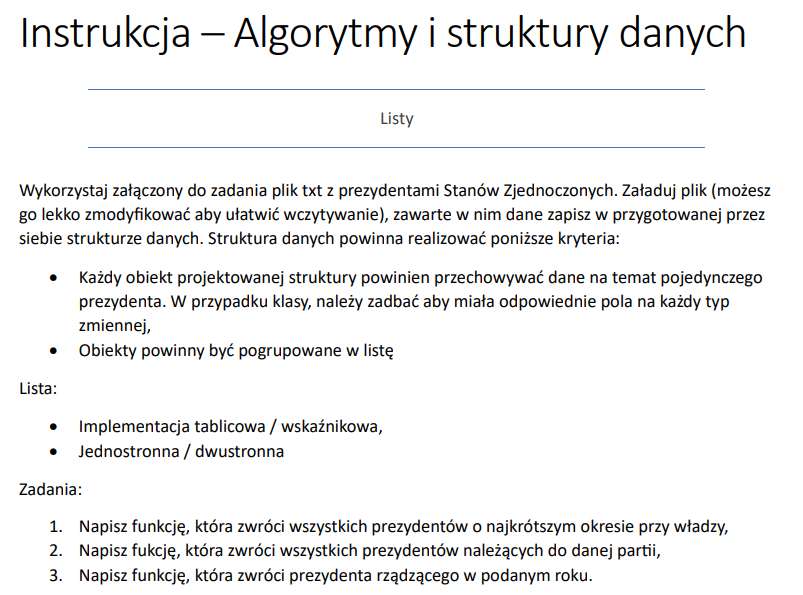
# Sprawozdanie, Zajęcia nr 1 – Listy, stosy i kolejki.

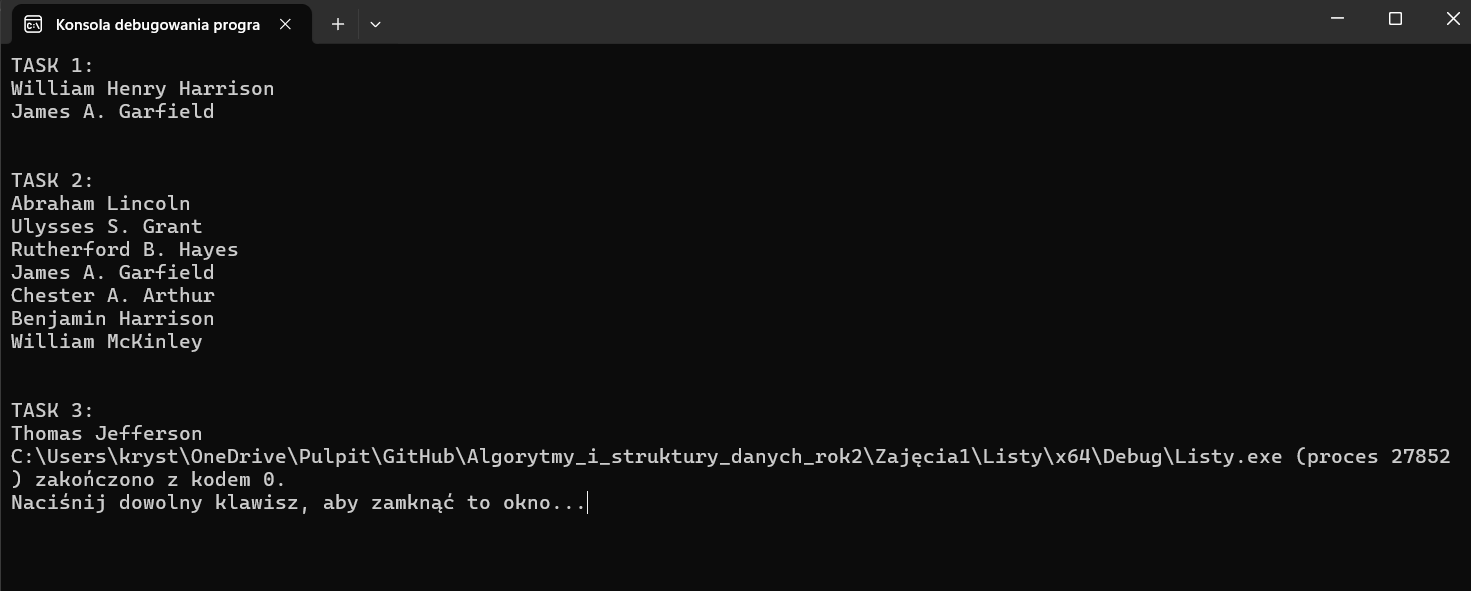
## Krystian Kostrzewa, 418845. WIMiIP, Informatyka Techniczna, sn. Data zajęć: 23.03.2024

1. **Listy**

**Polecenia:**

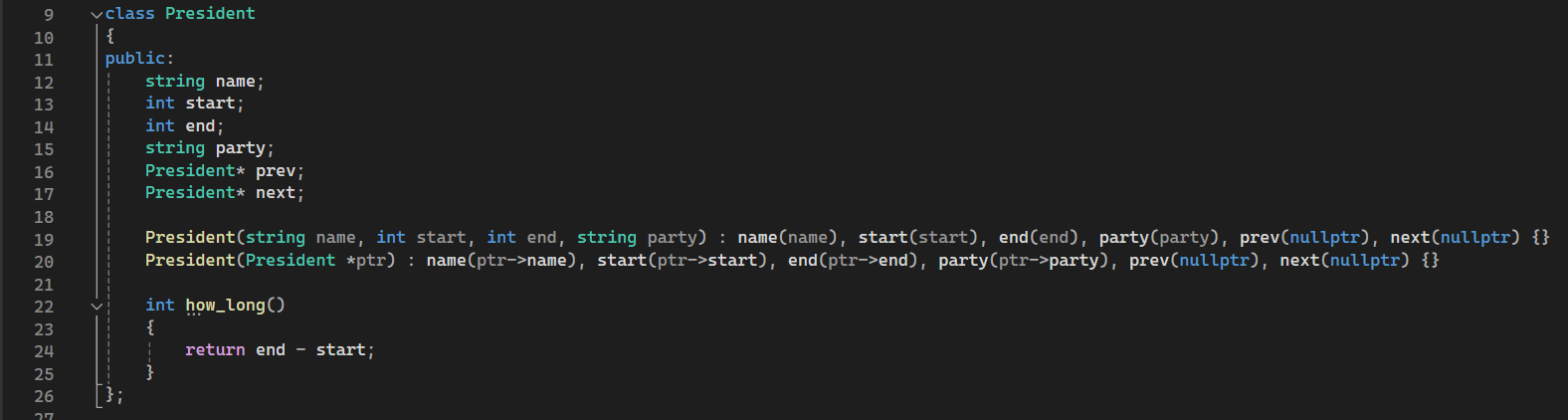


**Wynik w konsoli:**

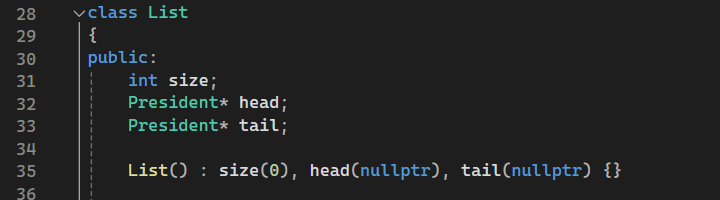


**Kod i opis rozwiązania:**

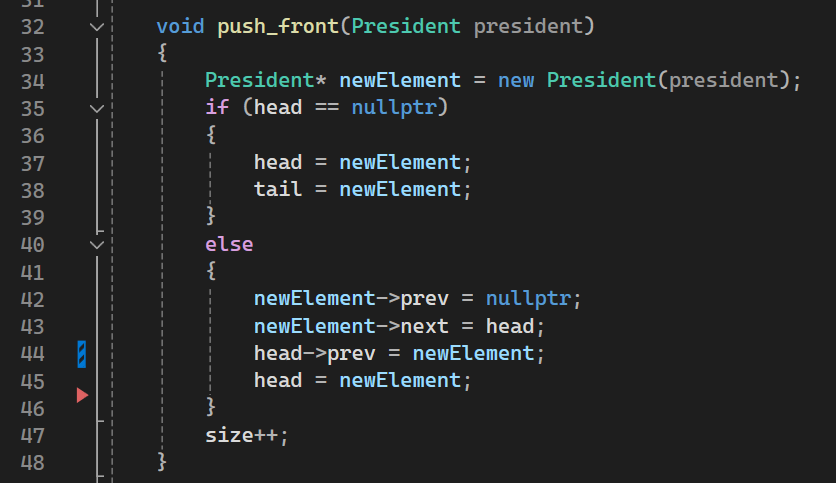
Do mojego kodu wybrałem implementację wskaźnikową dwustronną. Po skończonych pracach nad kodem uważam, że do tego zadania implementacja wskaźnikowa byłaby równie dobra, a może nawet lepsza. Zawsze to parę linii kodu mniej.



Na powyższym zrzucie ekranu przedstawiona jest klasa *President*, której obiekty będą mieć za zadanie przechowywać dane o jednym, konkretnym prezydencie USA takie jak imię i nazwisko, data rozpoczęcia rządzenia, data zakończenia rządzenia oraz partia, którą reprezentował rządząc. Dodatkowo obiekt tej klasy będą przechowywać wskaźniki na poprzedni i następny element listy, dzięki czemu będzie można za pomocą operacji na wskaźnikach przemieszczać się po strukturze danych. Klasa ta posiada dwa konstruktory. Pierwszy posłuży przede wszystkim do importu danych z pliku. Drugi konstruktor będzie używany częściej, głównie w metodach, w których operuję na wskaźnikach.

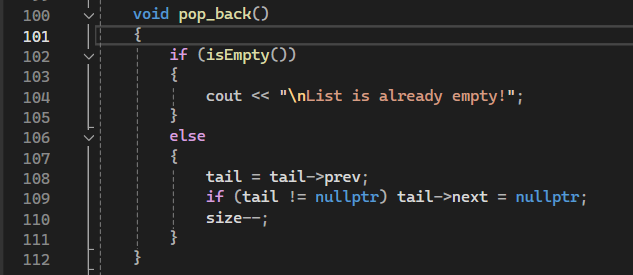
**

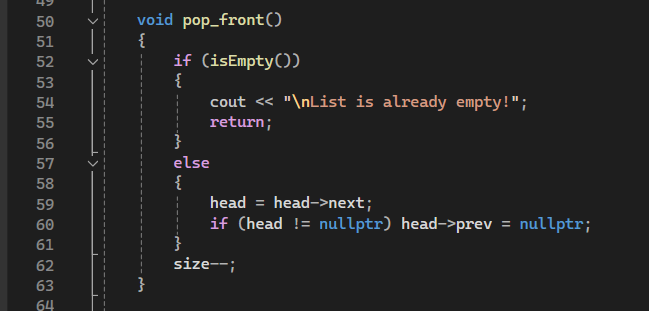
Elementy klasy *List* będą pozwalały zarządzać strukturami danych przechowującymi elementy w taki sposób, jak jest to przewidziane dla listy, czyli dostęp mamy jedynie do pierwszego i ostatniego elementu listy. Dlatego wskaźniki *head*  i *tail* wskazują na kolejno pierwszy i ostatni element struktury. Klasa posiada 1 konstruktor, który tworzy pustą listę. Uzupełniamy ją za pomocą metod.



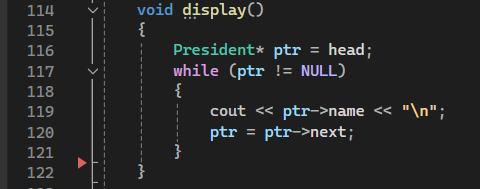


Metody dodające elementy do struktury to *push\_front* i *push\_back* przeciążona, aby można było bez problemu dodawać elementy na końcu listy za pomocą wskaźnika. Na przykładzie funkcji *push\_front* opiszę działanie tych wszystkich metod, gdyż działają analogicznie. Za pomocą konstruktora kopiującego tworzona jest dynamicznie alokowana zmienna *newElement*. Następnie program sprawdza, czy struktura klasy danych jest pusta. Jeśli tak, ustawia element zarówno jako pierwszy i ostatni element listy. Jeśli na liście coś już się znajduje manipulacja wskaźnikami, umieszczając element na 1 miejscu w liście (wskaźnik wskazujący na poprzedni element wskazuje na *nullptr* , a wskazujący na kolejny wskazuje element, który przed wywołaniem metody był headem, już nim nie jest, bo w linii 44 wskaźnik *prev* jest skierowany na inny element (już nie *nullptr*), a w 45 linii już oficjalnie headem zostaje nazwany świeżo dodany element.

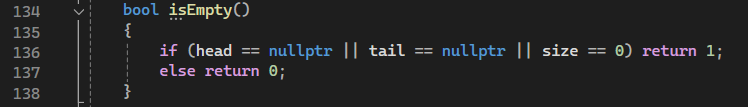


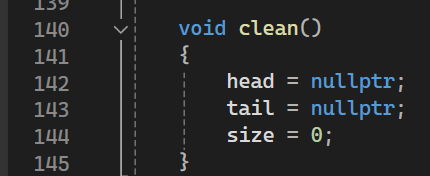


Kolejnymi metodami są *pop\_back* i *pop\_front*, które służą do usuwania kolejno ostatniego lub pierwszego elementu listy. Na przykładzie *pop\_front* opiszę działanie tych metod. Funkcja w pierwszej kolejności sprawdza, czy może lista nie jest pusta, jeśli tak nie robi nic, tylko zwraca odpowiedni komunikat i opuszcza ją. Jeśli znajdują się w niej jakieś elementy funkcja przekierowuje wskaźnik *head* na kolejny element listy i ustawia jego *prev* na *nullptr,* przez co nie ma połączenia z żadnym elementem przed nim.

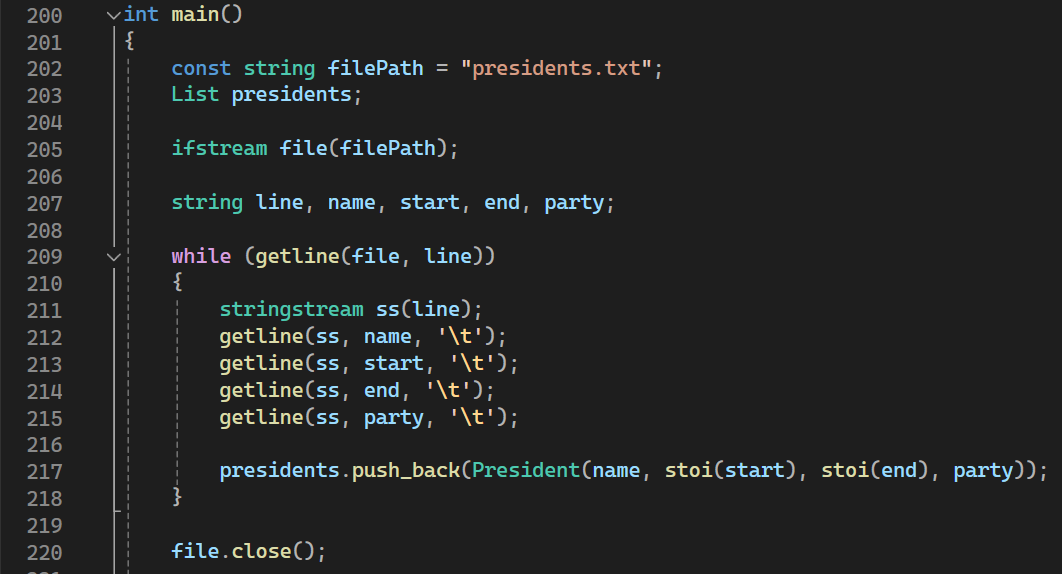


Metoda *display* wyświetla imię i nazwisko każdego prezydenta znajdującego się na liście. Zaczynając od głowy funkcja wypisuje *name* elementu, na który aktualnie skierowany jest wskaźnik *ptr* i przechodzi do następnego elementu na liście aż napotka nullptr.



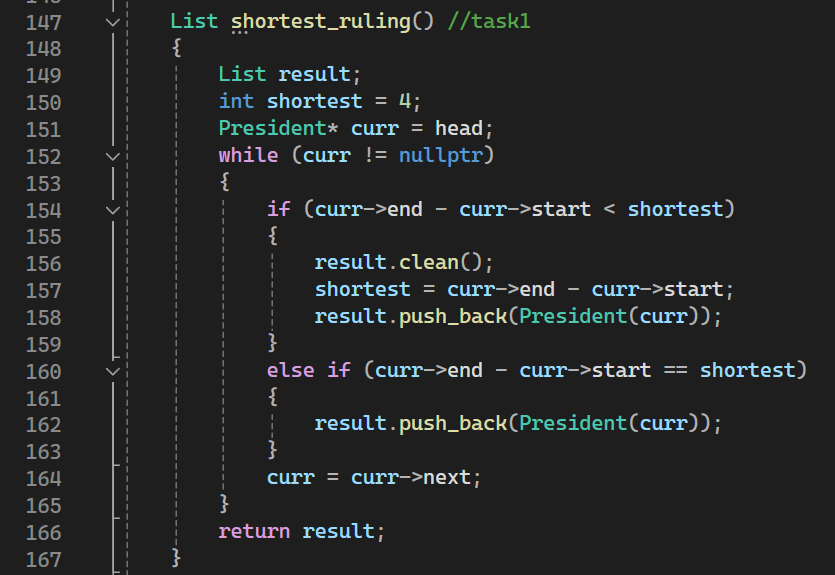


Metoda *isEmpty* po prostu sprawdza, czy w jakikolwiek sposób struktura danych wskazuje na to, że jest pusta. Jeśli tak - zwraca 1 (true), jeśli jednak nie – zwraca 0 (false). Natomiast *clean* służy do opróżniania zawartości listy poprzez ustawienie własności struktury danych na początkowe.

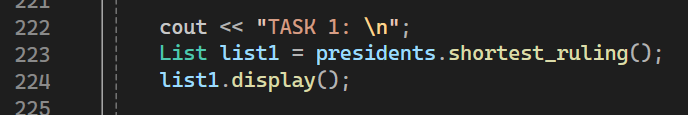


Powyższa część kodu służy do odczytu danych z pliku, zaimplementowanie ich w odpowiedni sposób do elementów klasy *President* oraz połączenia całości w liście klasy List. W skrócie działa to tak, że z całego pliku, linia po linii sczytywany jest tekst, zapisywany do zmiennej *ss*, następnie wszystkie dane oddzielone znakiem tabulacji przypisywane są do odpowiednich zmiennych, a potem za pomocą konstruktora klasy *President* tworzone są elementy tej klasy i przy użyciu metody klasy *List* dodawane są do listy *presidents*.

Rozwiązanie zadania 1:

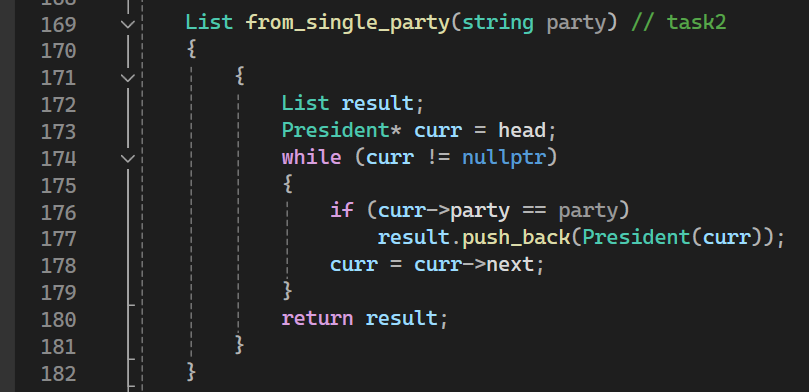


Do wykonania 1 polecenia przygotowałem metodę klasy *List*, która za pomocą wskaźnika *curr* (który po każdym wykonaniu pętli while wskazuje na kolejny element listy, aż napotka *nullptr*) sprawdza, czy aktualnie rozpatrywany prezydent panował najkrócej spośród wcześniej sprawdzanych. Jeśli tak, czyści listę z poprzednich i dopisuje element do listy *result*. Jeśli jednak rządził równie długo – nie usuwa pozostałych prezydentów będących już na liście, tylko dopisuje go, jako kolejny element. Na końcu zwracana jest tablica, która przechowuje wszystkich najkrócej rządzących prezydentów.

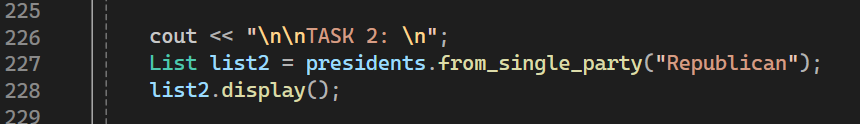


Powyżej wywołanie metody na liście przechowującej wszystkich prezydentów z pliku oraz wyświetlenie jej za pomocą metody *display* w funkcji *main*.

Rozwiązanie zadania 2:

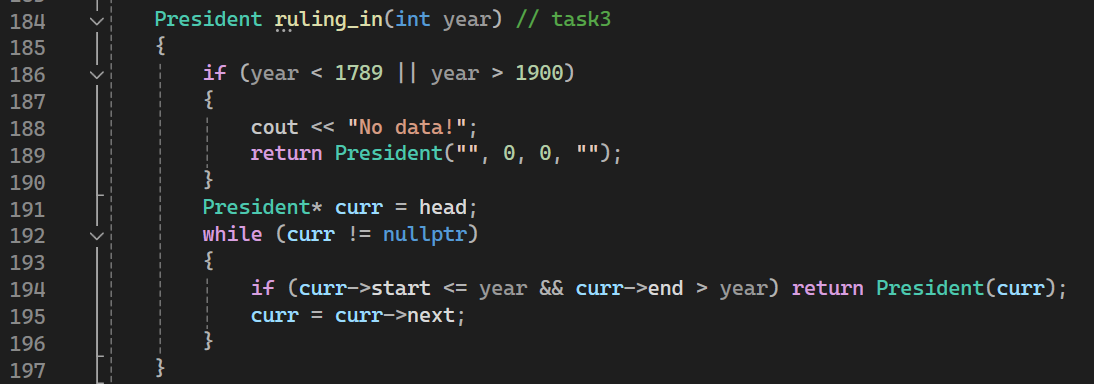


Do wykonania 2 polecenia przygotowałem metodę klasy *List*, która za pomocą wskaźnika *curr* (który po każdym wykonaniu pętli while wskazuje na kolejny element listy, aż napotka *nullptr*) sprawdza, czy aktualnie rozpatrywany prezydent należał do partii, której nazwę przekazaliśmy w argumencie, przy wywołaniu metody. Jeśli tak - dopisuje element do listy *result*. Na końcu zwracana jest tablica, która przechowuje wszystkich prezydentów, którzy należeli do wskazanej partii.

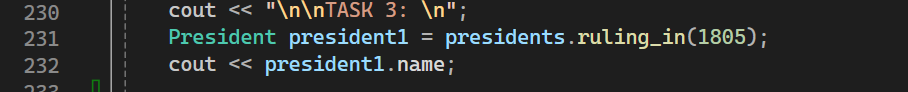


Powyżej wywołanie metody na liście przechowującej wszystkich prezydentów z pliku oraz wyświetlenie jej za pomocą metody *display* w funkcji *main*.

Rozwiązanie zadania 3:



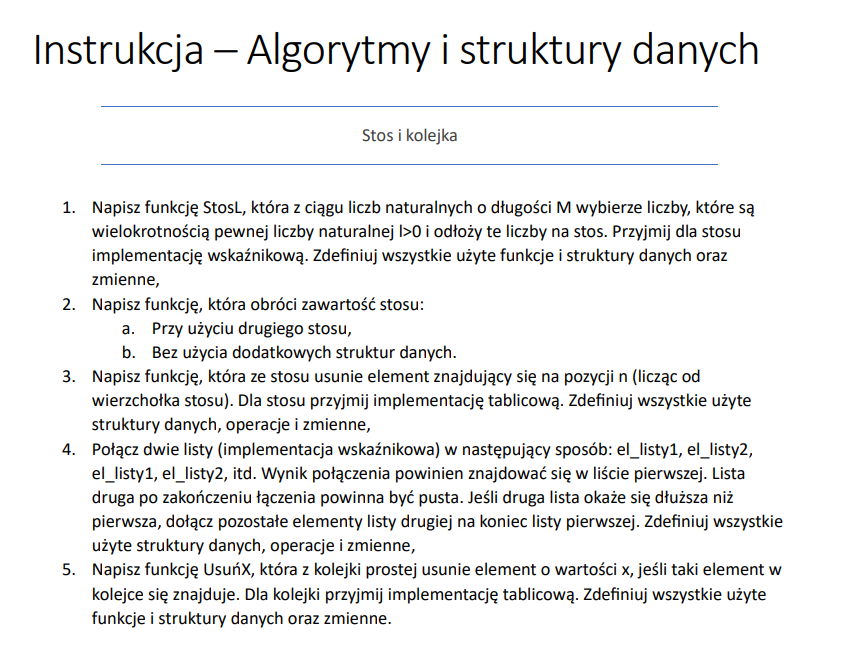
Do wykonania 2 polecenia przygotowałem metodę klasy *List*, która w pierwszej kolejności sprawdza, czy rok wpisany przez użytkownika mieści się w przedziale lat, zgodnym z danymi z pliku wejściowego, jeśli nie -zwracany jest pusty element klasy *President* oraz odpowiedni komunikat o braku danych. Jeśli jednak rok znajduje się w odpowiednim przedziale sprawdzane jest, czy zmienna *party* w tym elemencie jest równa argumentowi podanemu przy wywołaniu funkcji. Jeśli tak – element klasy President jest zwracany i następuje wyjście z funkcji.



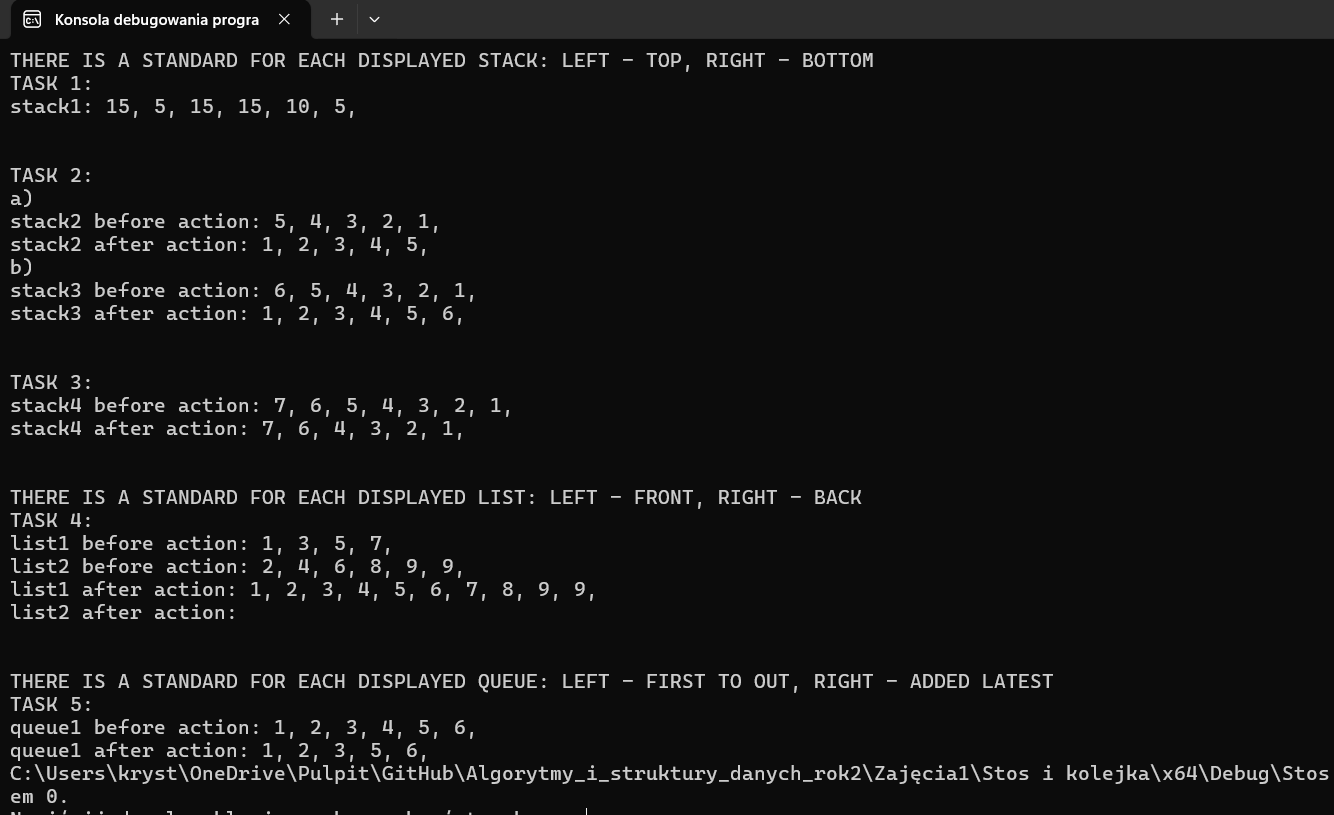
Powyżej wywołanie metody na liście przechowującej wszystkich prezydentów z pliku oraz wyświetlenie jej za pomocą metody *display* w funkcji *main*.

1. **Listy**

**Polecenia:**

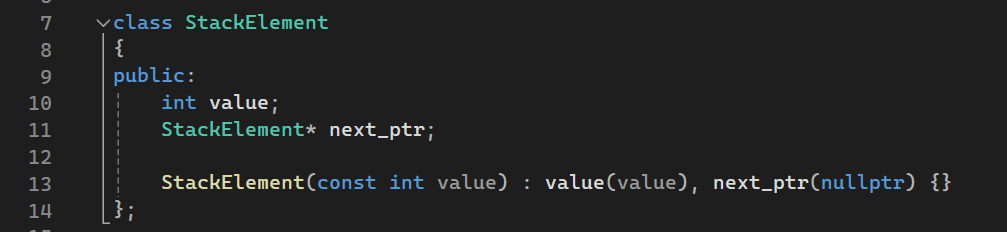


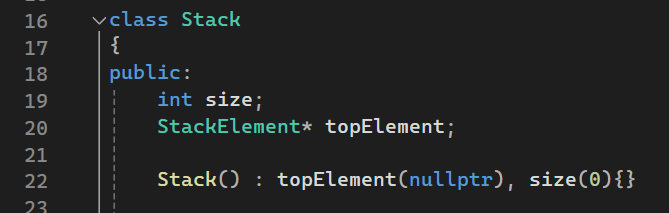
**Wynik w konsoli:**



**Kod i opis rozwiązania:**

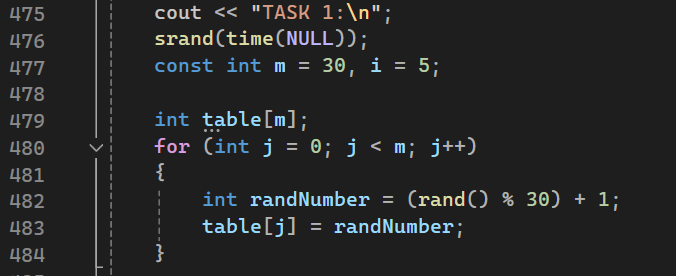
Do wykonania zadań 1 i 2 użyłem elementów klas *StackElement* oraz *Stack,* która przechowuje dane i daje do nich dostęp na bardzo podobnej zasadzie jak *List* z pierwszej części laboratoriów. Jest to jednak przykład implementacji wskaźnikowej jednostronnej.



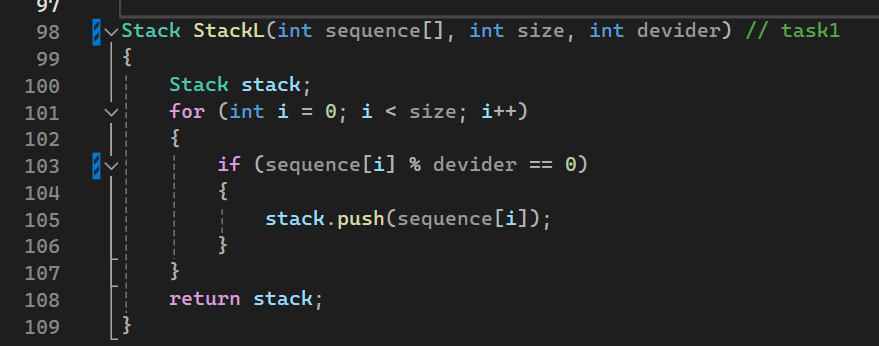


Powyższe zrzuty ekranu pokazują definicje klas *StackElement* oraz *Stack* (bez jej metod, bo za dużo miejsca by to wszystko zajęło, a są bliźniacze do tych z pierwszej części zajęć).

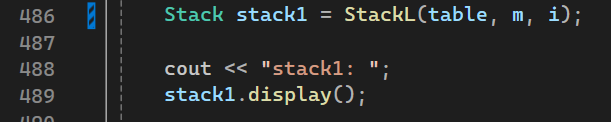
Rozwiązanie zadania 1:



Pokazana wyżej część kodu służy do utworzenia tablicy typu int oraz uzupełnieniu jej *m* losowymi wartościami liczbowymi w zakresie od 1 do 30

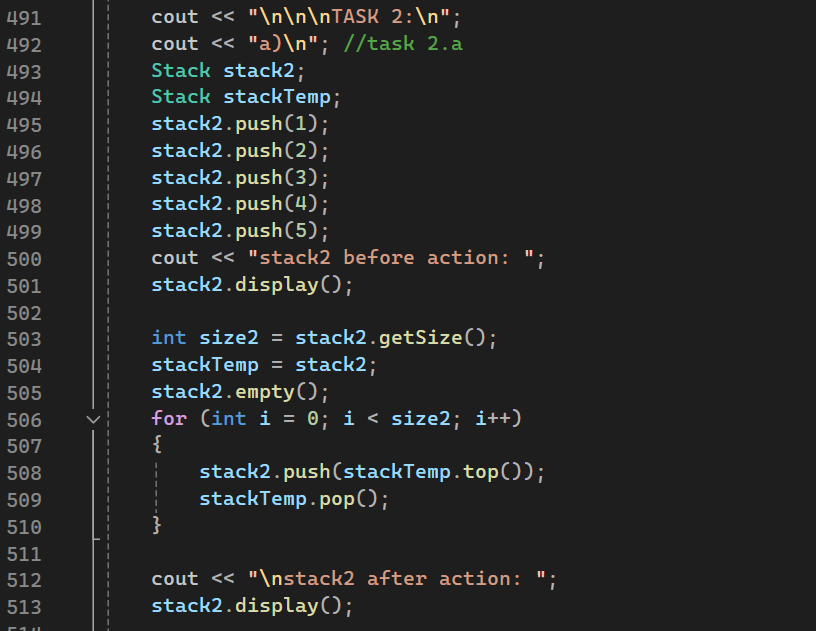


Do wykonania zadnia 1 utworzyłem funkcję *StackL*,która przechodzi po każdym elemencie podanej w argumencie tablicy, sprawdza czy dany element jest podzielny przez podaną na początku zmienną *i* (do funkcji podana jako *devider*) i jeśli tak – dodaje go na wierzchołek stosu. Na końcu zwracany jest stos klasy *Stack*.



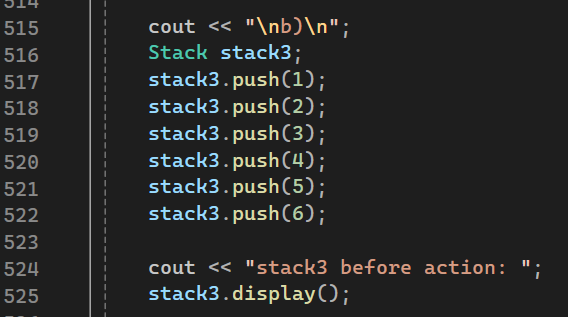
Powyżej wywołanie funkcji oraz wyświetlenie stosu za pomocą metody *display* w funkcji *main*.

Rozwiązanie zadania 2a:

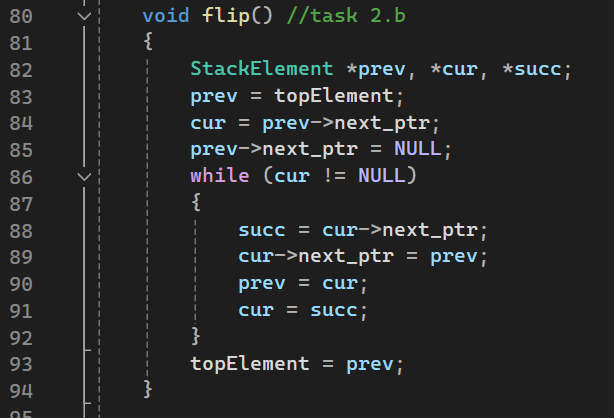


Tym razem całość rozwiązania znajduje się w funkcji *main*. Najpierw następuje uzupełnienie stosu i wyświetlenie go. Następnie stos pomocniczy zostaje kopią *stack2*. Metodą *empty* zostaje opróżniony *stack2*, po czym elementy ze stosu pomocniczego są przerzucane z powrotem na *stack2* w wyniku czego elementy wróciły na stos w odwrotnej kolejności. Na końcu zostaje wyświetlony stos po wykonanych działaniach.

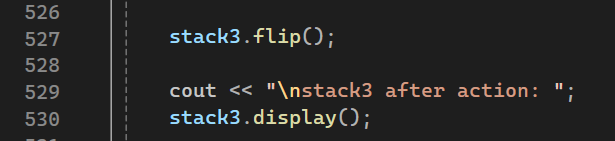
Rozwiązanie zadania 2b:



Na początku, jak w poprzednim zadaniu został przygotowany, usupełniony i wyświetlony stos *stack3.*

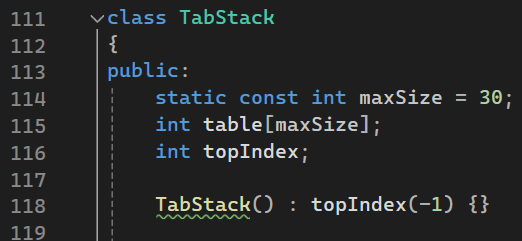


Do wykonania zadania 2b przygotowałem metodę *flip* klasy *Stack*. Zaczyna ona od wierzchołka stosu (pierwszego elementu na stosie) i idzie do kolejnych elementów. Podczas przechodzenia przez te elementy, zmienia kierunek wskazywania: zamiast wskazywać na następny element, wskazuje na poprzedni. Kiedy dotrze do ostatniego elementu, ustawia go jako nowy wierzchołek stosu. W ten sposób zmienia kolejność elementów na stosie odwracając ją bez używania innych struktur pamięci.

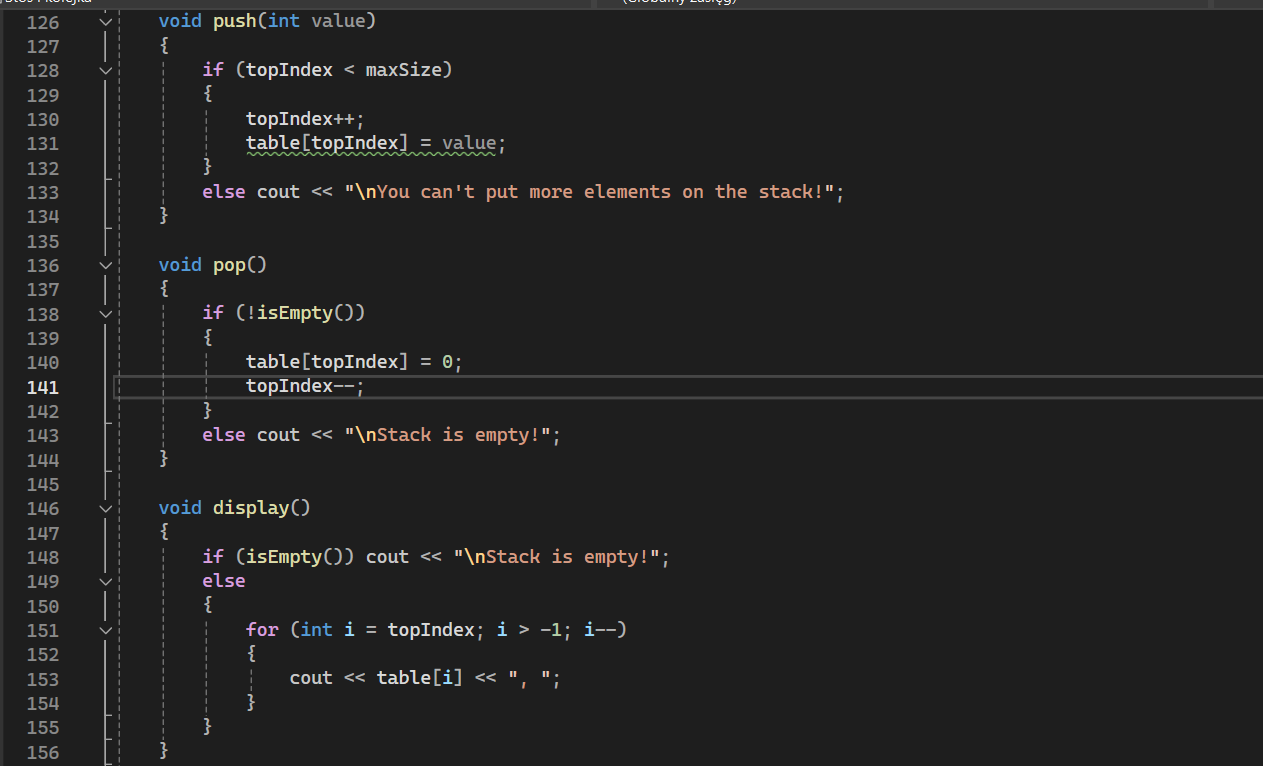


I na końcu wywołanie metody *flip* dla stosu *stack3* oraz wyświetlenie jego zawartości po dokonaniu akcji.

Do wykonania zadań 1 i 2 użyłem elementów klasy TabStack, która przechowuje dane i daje do nich dostęp dzięki implementacji tablicowej.

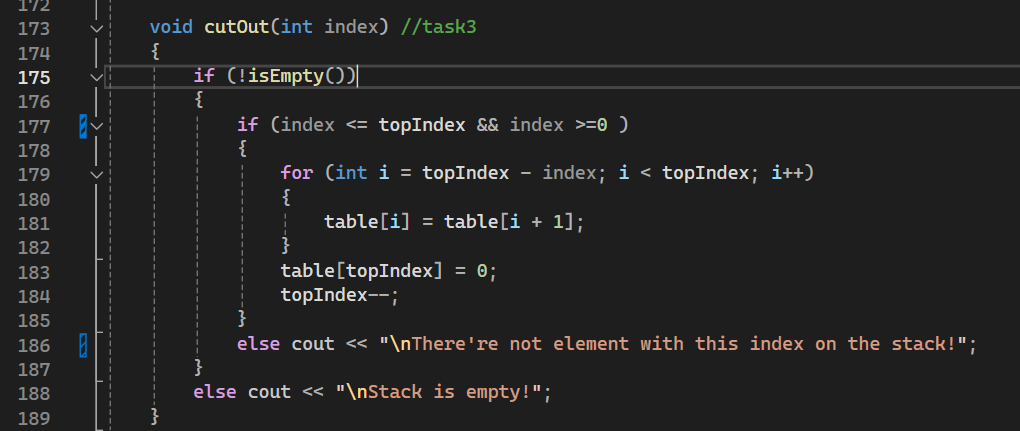


Powyższe zrzuty ekranu pokazują definicje klasy *TabStack*, której metody nie wykorzystują wskaźników, ale stale śledzą zachowanie górnych elementów stosu i dzięki zmiennej *topIndex*  mamy stały dostęp do elementu na górze stosu, jak i do rozmiaru stosu, co wykorzystujemy np. tak jak w przykładowych metodach:



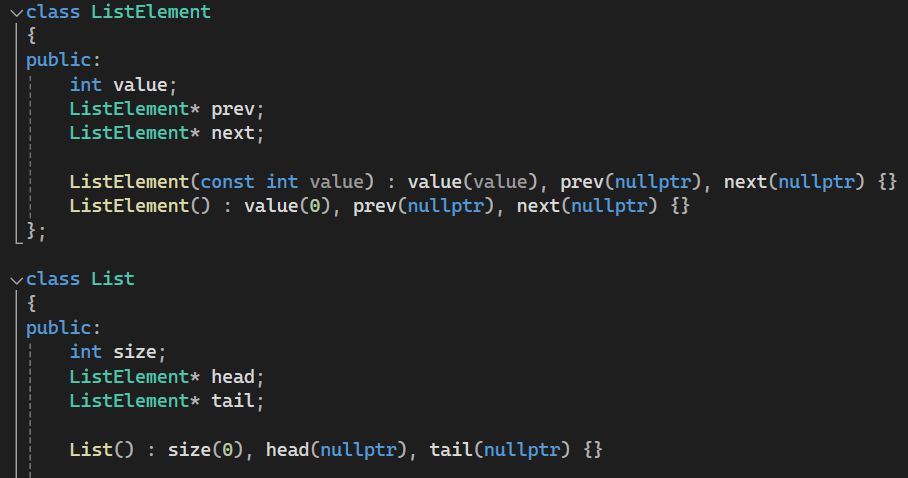
Rozwiązanie zadania 3:

Najpierw oczywiście w funkcji *main* zadeklarowałem, wypełniłem i wyświetliłem stos stack4 klasy *TabStack*.



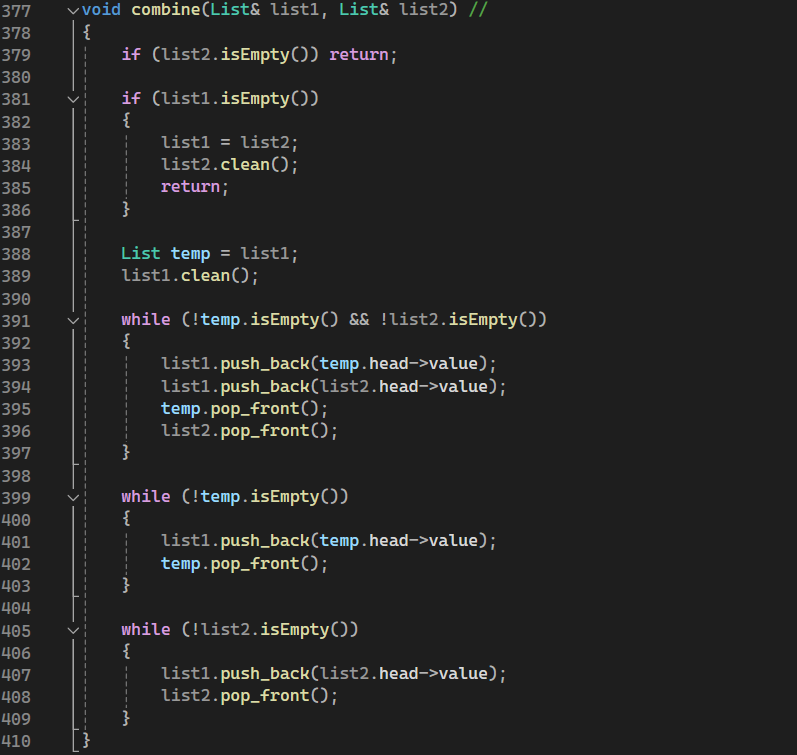
Do wykonania zadania 3 przygotowałem metodę *cutOut* klasy *TabStack*, która usuwa ze stosu element o indeksie podanym w argumencie funkcji przy jej wywołaniu. Następnie od wybranego elementu w górę nadpisuje wszystkie elementy następnym, a na końcu zmniejsza topIndex. Na końcu w funkcji main wywołałem metodę dla *stack4* i wyświetliłem stos po operacji.

Poniższe zrzuty ekranu pokazują definicje klas *ListElement* oraz *List* (bez jej metod, bo za dużo miejsca by to wszystko zajęło). Działa ona na identycznej zasadzie jak lista z pierwszej części zajęć.



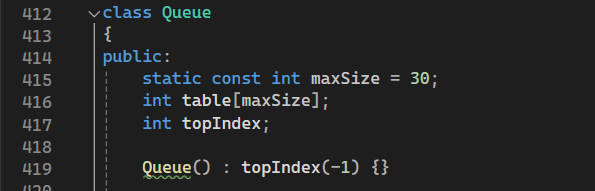
Rozwiązanie zadania 4:

Najpierw w funkcji *main* zadeklarowałem, wypełniłem i wyświetliłem listy *list1* i *list2* klasy *List*.



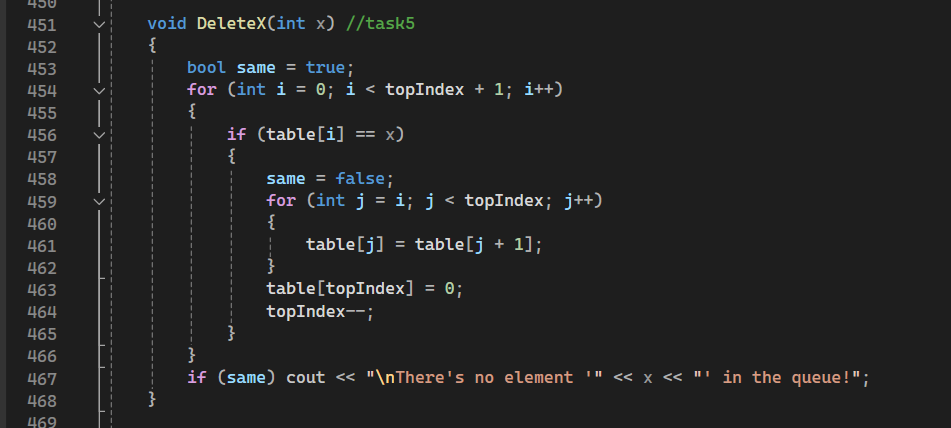
Do wykonania zadania 4 przygotowałem funkcję *combine*, która jako argumenty przyjmuje 2 elementy klasy *List*. Działa ona w ten sposób, że do pustej listy dodaje kolejno, naprzemiennie pierwsze elementy tych list, a następnie usuwa te elementy, aż któraś nie będzie pusta. Finalnie dopisuje ewentualnie dłuższą listę na końcu. Po wywołaniu funkcji *combine* w mainie zostały również wyświetlone obydwie listy.

Poniższe zrzuty ekranu pokazują definicje klasy *Queue* (bez jej metod, bo za dużo miejsca by to wszystko zajęło). Działa ona na bardzo podobnej zasadzie jak klasa *TabStack*, tylko na zasadzie kolejki, first in first out.



Rozwiązanie zadania 5:

Najpierw w funkcji *main* zadeklarowałem, wypełniłem i wyświetliłem kolejkę *queue1* klasy *Queue.*



Do wykonania zadania 5 przygotowałem metodę *DeleteX* klasy Q*ueue.* Przy użyciu pętli for przechodzi ona przez każdy element kolejki, sprawdzając czy jest on równy argumentowi podanemu przy wywoływaniu metody. Jeśli wystąpi taki element zmieniana jest wartość zmiennej *same*, a potem element jest usuwany z kolejki jak w metodzie *cutOut* klasy *TabStack*, która była już opisana wyżej. Jeśli zmienna *same* nie uległa zmianie (czyli element o danej wartości nie znajdował się w kolejce) wyświetlany jest odpowiedni komunikat.