Programowanie Zaawansowane FM i NI
Ćwiczenia 7

Zadanie 1 (klasy, w tym konstruktor, pole wskaźnikowe)

W doświadczeniu na sprężynie zawieszano ciężarki o różnych masach. Wyniki pomiarów (wartości mas *m* oraz długości wychylenia *L*) zestawia Tabela.

Tabela. Wychylenia *L* sprężyny dla danych mas *m* zawieszonych na niej ciężarków.

m [g]	13.04	23.69	34.32	44.95	55.61
L [cm]	47.0	83.5	122.0	160.0	198.0

W ramach modelu zależności proporcjonalnej $L = a \cdot m$, wzory *regresji liniowej* podają wartości $a \pm s_a$ dla prostej najlepszego dopasowania:

$$a = \frac{\sum_{i=1}^{N} m_i L_i}{\sum_{i=1}^{N} m_i^2} \quad , \qquad s = \sqrt{\frac{\sum_{i=1}^{N} \left(L_i - a \cdot m_i\right)^2}{N-1}} \quad , \qquad s_a = \frac{s}{\sqrt{\sum_{i=1}^{N} m_i^2}} \quad .$$

W funkcji `main` umieść dane dot. `m` i `L` w osobnych tablicach elementów typu `double`.

Następnie napisz klasę `Regresja`, której zadaniem będzie wyznaczenie `a` oraz `s` na podstawie danych, które przyjmie w konstruktorze. Klasa ma posiadać:

- w polach prywatnych wskaźniki `double* m` i `double* m`, gotowe do przypisania im adresów tablic odpowiednich danych, oraz zmienną `int size`, mieszczącą rozmiar tablicy.
- zmienne double `a`, `S` oraz `Sa`, mogące przechować rezultaty obliczeń wg wzorów.
- konstruktor przyjmujący przez 2 wskaźniki adresy tablic z danymi oraz przez `int` – rozmiar tablicy. *Uwaga*: klasa `Regresja` nie ma zawierać tablic z danymi, a jedynie adresy tablic z zewnątrz.
- metody `void wyznacz_a ()`, `wyznacz_Sa ()` i `wyznacz_S ()`, obliczające powyższe wartości i wstawiające wyniki do pól: `a`, `S` i `Sa`.
- metodę `void Analiza()`, wywołującą kolejno powyższe metody
- metodę `void Status()`, wypisującą na ekran obliczoną `a` oraz `Sa`

W funkcji `main` utwórz obiekt klasy `Regresja` na danych tablicach `m` i `L`, a następnie wywołaj na tym obiekcie metodę `Analiza` i `Status`.

Zadanie 2 (struct – stos, pole z obiektem własnej klasy, operator -> , alokacja)

Twoim zadaniem jest prosta implementacja stosu obiektów, gdzie każdy obiekt posiada jeden napis. Dla pojedynczego obiektu utwórz `struct Item`, a dla stosu – `class Stack`. Każdy `Item` dokłada się „na górę" `Stack`'u, a zdejmować ze `Stack`'u można tylko z góry.

`struct Item` powinna posiadać:
- pole z napisem typu `string`
- wskaźnik `prev` na adres obiektu typu `Item`. Wskaźnik docelowo ma przechowywać adres tego obiektu `Item`, który jest poprzednikiem na stosie.
- konstruktor dwuargumentowy, inicjujący powyższe pola.

`class Stack` powinna posiadać:
- prywatne pole `Top`, będące wskaźnikiem na obiekt „na szczycie"
- konstruktor bezargumentowy, inicjujący stos jako pusty (użyj `nullptr`)
- metodę `Stack& Push (string)` , która alokuje dynamicznie nowy obiekt `Item`, kopiując do niego napis i podając adres obiektu, który przed chwilą był na szczycie. Następnie wstaw adres nowego obiektu na szczyt.
  Na koniec metoda ma zwrócić nasz stos.
- metodę `bool Pop (string&)` , która zdejmuje ze stosu wierzchni napis, ale podaje jego treść przez argument wejścia.
  Uwaga: zdejmowanie ma uwzględniać dealokację tego obiektu.
  Jeżeli jednak stos był pusty (nie można zdjąć napisu), to `Pop` ma natychmiast zwrócić `false`. W przypadku sukcesu – metoda ma zwracać `true`.

W funkcji `main` utwórz stos. Niech użytkownik podaje z klawiatury słowa, które będą umieszczane kolejno na szczycie stosu, aż użytkownik wpisze (umowne) 0. Następnie niech komputer pozdejmuje napisy ze stosu, zarazem wypisując je na ekran.

Zadanie 3 (klasy, konstruktor, destruktor , operatory () i < , wskaźnik w polu, alokacja)

Zaprojektuj klasę `MyString`, której obiekt przechowuje C-string i posiada konstruktor, destruktor oraz operatory: () oraz < . Skorzystaj z tego wzorca .
Klasa powinna mieć:

- w polach prywatnych: wskaźnik `char* Tab`, gotowy do przyjęcia adresu początku tablicy znaków oraz `int len` – do przechowania długości napisu.
- konstruktor `MyString ( char* )` , przejmujący C-string, alokujący dynamicznie nową tablicę `char`'ów (*uwaga*: powinna mieścić też `char` kończący o wartości 0) , przypisujący jej znaki ze wzorca i ustawiający wartość `len`.
- gettery: `size()` zwracająca wartość `len` oraz `c_str()`, zwracająca adres początku tablicy znaków.
- destruktor – po to, aby przed skasowaniem obiektu skasować alokację tablicy

– `char& operator() (int N)` , który zwraca N-ty znak. Zwracanie przez referencję umożliwia użycie tego operatora do przypisania w to miejsce znaku. *Uwaga*: klasa posiadająca operator() czyni takie obiekty *funktorami*. Oznacza to, że wolno napisać `obiekt(..)` , a więc „wywołać" go jak funkcję.
– `bool operator< (MyString& S2)` . To tzw. *komparator*. Jego zadaniem jest zwrócenie `true`, gdy nasz napis jest alfabetycznie ściśle wcześniejszy od napisu, w przeciwnym wypadku – ma zwrócić `false`. Operator ten ma również zwrócić `true`, gdyby po kolejnych identycznych literach okazało się, że nasz napis jest krótszy, zaś `false` – gdy krótszym okaże się `S2`. (Por. `strcmp` w bibliotece `<cstring>`).

Zadanie 4 (klasy, w tym konstruktory, destruktor, operatory, wskaźnik w polu, alokacja)

Zaprojektuj klasę `Notes`, która zarządza tablicą obiektów klasy `Osoba`, alokowaną dynamicznie. W tym celu utwórz klasę `Osoba`, przechowującą informację o (dla prostoty) imieniu i numerze telefonu znajomej osoby. Obiekt klasy `Notes` ma pobierać, wyświetlać, kasować i sortować wpisy o osobach. Możesz skorzystać z tego wzorca .

Klasa `Osoba` powinna zawierać:
– pola prywatne: `string Imie, int tel`
– konstruktory: bezargumentowy, jednoargumentowy dla imienia, dwuargumentowy oraz kopiujący. W każdym wypadku zainicjuj pola wartościami.
– settery i gettery (np. `ustawImie`, `podajTel`)
– metodę `Wypisz`, wyświetlającą w jednej linijce imię i nr telefonu
– `bool operator== (Osoba& O2)`, zwracający `true`, gdy nasza osoba ma te same dane, co `O2`. Jeśli nie, zwraca `false`. *Uwaga*: przyda się `string::compare` .
– `bool operator< (Osoba& O2)`, zwracający `true`, gdy nasza osoba jest alfabetycznie wcześniejsza od osoby `O2`. Jeśli nie – zwraca `false`. Przyda się `string::compare`

Klasa `Notes` powinna zawierać:
– w prywatnych polach: wskaźnik `Osoba* Tab`, do którego konstruktory przypiszą adres zaalokowanej tablicy obiektów klasy `Osoba`. Ma też posiadać 2 zmienne rozmiarowe typu `int`: `size` i `capacity`, zainicjowane w deklaracji do, odpowiednio, `0` i `3`. `size` ma mieścić rzeczywistą liczbę wpisów. `capacity` – rozmiar zaalokowanej tablicy.

Reguła: jeżeli poszerzanie wpisów w `Tab` ma spowodować, że `size` przekroczy `capacity`, to osobna prywatna metoda `Extend` ma podwoić `capacity`, następnie zaalokować nową tablicę o rozmiarze `capacity`, przelać osoby ze starej tablicy do nowej, zdealokować tablicę starą, a adres nowej przypisać wskaźnikowi `Tab`. Celem reguły jest zmniejszenie częstości realokowania tablicy, które jest czasochłonne przy dużych tablicach.

– konstruktory: bezargumentowy, dwa 1-argumentowe: przyjmujący 1 osobę oraz przyjmujący cały notes, 2-argumentowy przyjmujący adres tablicy osób i jej rozmiar, jak też konstruktor kopiujący.

- destruktor, dealokujący tablicę
- `operator+= (Osoba& Os)` do dopisania nowej osoby do tablicy
- `operator+= (Notes& N2)` do dopisania wpisów z notesu N2 do tablicy
- `operator-= (Osoba& Os)`, wyszukujący w tablicy pierwsze wystąpienie wpisu `Os` i „kasujący je" poprzez cofnięcie osób „dalszych" o 1 element i dekrementację `size`. Tu przyda się `Osoba::operator==`
- metodę `Wypisz`, wyświetlającą cały notes, korzystając z metody `Osoba::Wypisz`
- metodę `SortAlfa`, sortująca wpisy zgodnie z alfabetycznym porządkiem imion. Możesz użyć sortowania bąbelkowego. Używając (`Osoba1 < Osoba2`), wywołasz `operator<` w klasie `Osoba`.