Zadania z klas do poćwiczenia

1. Napisz klasę `Trojkat` o bokach `double A, double B, double C` (pola prywatne). Zaprogramuj w tej klasie metody:

– `void ustawBoki (double, double, double)` ← która ustawia boki trójkątowi
– `double podajA()` i odpowiednio `podajB`, `podajC` ← która zwraca długość danego boku
– `double Obwod()` oraz `double Pole()` ← zwracająca obwód i pole trójkąta.
Pole trójkąta o zadanych bokach możesz wyznaczyć ze wzoru Herona.
– `void Print ()` ← wypisującą na raz długości boków.

W funkcji `main` utwórz trójkąt `T` o bokach (3, 4, 5), a następnie wyświetl jego boki, wykonując na nim metodę `Print()`. Wypisz jego obwód i pole, korzystając z odpowiednich metod.

2. Wzbogać klasę `Trojkat` z zadania 1 o konstruktory:
– bezargumentowy, resetujący boki do 0
– 3-argumentowy do przypisywania boków podczas tworzenia obiektu.
– kopujący na wypadek tworzenia nowego trójkąta przez przypisanie mu istniejącego.
Dodaj też destruktor.

Zmień funkcję `main()` tak, aby ustawić boki przy pomocy konstruktora. Następnie utwórz trójkąt `T2` poprzez kopię trójkąta `T` i wyświetl jego pola.

Teraz, jeśli zaprogramował(e/a)ś konstruktory wprost, przepraw je tak, aby pola wypełniać w liście inicjalizacyjnej konstruktora.

3. Zmienimy teraz klasę `Trojkat` tak, aby działała na punktach. W tym celu utwórz klasę `Punkt` o polach `A` i `B` typu `double`, będących współrzędnymi. Zaimplementuj:

– konstruktor 2-wymiarowy z użyciem *listy inicjalizacyjnej* konstruktora.
– metodę `void Print()`, wyświetlającą informację o punkcie np. tak: `[ -1.5 , 4 ]`,
– metodę `double Distance (const Punkt& P2)`, która zwróci odległość między punktem rodzimym a punktem `P2`.
– dla poćwiczenia, skonstruuj `operator+`, przyjmujący obiekt typu `Punkt` (optymalnie – przez referencję). Operator wytwarza nowy obiekt klasy `Punkt`, którego każda współrzędna jest sumą odpowiednich współrzędnych ze składników działania. Operator zwraca nowoutworzony obiekt.

Następnie zmień pola klasy `Trojkat` tak, aby w polach przechowywała trzy obiekty klasy `Punkt`. Podmień konstruktor, aby móc skonstruować obiekt `Trojkat` z trzech obiektów `Punkt`. Podmień też w klasie `Trojkat` getter'y, metodę `Print` oraz metody `Obwod` i `Pole`, tak aby działały na podstawie punktów. Zauważ, że długości boków można odzyskać, działając metodą `Distance` pomiędzy punktami.

W funkcji `main()` utwórz punkty `P1 (0, 4)`, `P2 (3, 0)` i `P3 (0, 0)`, a następnie utwórz trójkąt `T`, oparty o te punkty. Wyświetl informacje o tym trójkącie, a następnie podaj jego obwód i pole.

Na koniec stwórz trójkąt `T2` poprzez kopię trójkąta `T` i wyświetl jego pola.

4. Po zakończonym zadaniu 3. dysponujesz kodem z klasami `Trojkat` i `Punkt`. Podziel kod na pliki osobno poświęcone każdej z klas (odpowiednio: plik nagłówka i implementacyjny) oraz plik klienta. Następnie skompiluj całość niezależnie na 2 sposoby:
- komendą w jednej linii
- „krok po kroku", tj. najpierw kompilacja do plików obiektowych, a następnie kompilacja pliku klienta z linkowaniem obiektów.

5. Napisz klasę `WektorN` reprezentującą wektor liczb double w N-wymiarowej przestrzeni. Zakoduj tablicę danych poprzez alokację dynamiczna. Klasa powinna mieć prywatne pola :

- wskaźnik na tablicę : `double* tab`
- rozmiar przestrzeni : `int dim`

Zakoduj:

- konstruktor 1-wymiarowy: przyjmuje liczbę typu `double`. Konstruktor alokuje 1-wymiarową tablicę i wstawia do niej liczbę. Ustawia `dim` na 1.

- konstruktor 2-wymiarowy: pobiera rozmiar i tablicę danych typu `double` poprzez wskaźnik. Konstruktor alokuje tablicę o odpowiednim rozmiarze i przypisuje jej elementy. Ustawia `dim` na właściwą wartość.

- konstruktor kopiujący: przyjmuje obiekt typu `WektorN` jako wzorzec. Konstruktor tworzy dynamicznie tablicę i wypełnia ją danymi ze wzorca.

- konstruktor inicjujący ze zbioru liczb, poprzez `initializer_list<double>` (por. wykład). Pozwala, aby zadeklarować obiekt np. tak: `WektorN V = {1, 2, 3, 4, 5} ; .`

- destruktor: zwalnia zaalokowaną tablice

- gettery: rozmiaru oraz adresu początku tablicy danych

- `operator+` , przyjmuje obiekt klasy `WektorN` (optymalnie - przez referencję). Pozwala na: `W1 + W2` , o ile rozmiar `W1` = rozmiar `W2`. Jeśli rozmiary się różnią, wyświetla komunikat i przerywa program. Operator wytwarza nowy obiekt klasy `WektorN`, z wynikiem działania. Zwraca ten obiekt.

- `operator+=` , przyjmuje liczbę typu `double`. Pozwala na: `W += 1.23`. Zwraca rodzimy obiekt.

- `operator++` , pozwala na: `W++` . Efektem jest zwiększenie o 1 każdego elementu tablicy ale jako postinkrementacja. Zwraca rodzimy obiekt.

- `operator++` , pozwala na: `++W` . Efektem jest zwiększenie o 1 każdego elementu tablicy ale jako preinkrementacja. Zwraca rodzimy obiekt.

- `operator*` , przyjmuje liczbę typu `double` oraz obiekt klasy `WektorN`. Pozwala na: `W2 = 1.23 * W1` (przemyśl, gdzie powinien zostać umieszczony ten operator). Aby miał on dostęp wprost do pól klasy `WektorN`, zastosuj mechanizm `friend` . Wytwarza nowy obiekt klasy `WektorN`, w którym każdy element jest starym elementem, przemnożonym przez liczbę. Zwraca ten nowy obiekt.

- `operator==` , przyjmuje `obiekt klasy WektorN` (optymalnie - przez referencję). Pozwala na sprawdzenie, czy `W1 == W2` . Zwraca w typie `bool` odpowiedź:
  `true` - gdy oba wektory są identyczne co do rozmiarów i zawartości,
  `false` - w każdym innym przypadku.

- `print()` , wypisuje elementy wektora (zadbaj o czytelność)

- `length()` , zwraca długość wektora, tj. sqrt ($w_0 \cdot w_0$ + ...)

W funkcji `main` zademonstruj działanie każdej z metod powyższej klasy.

Pamiętaj o tym, że za alokowanie pamięci i jej zwalnianie odpowiada programista.

6. Struktura `Napis` w polach posiada wskaźnik (`Text`) na c-string oraz zmienną rozmiaru `Size`, opisującą liczbę liter napisu (bez finalnego char = 0).

Napisany jest konstruktor, przyjmujący poprzez c-string konkretny napis (tzw. stała napisowa wymaga, aby zmienna przyjmująca była const), alokujący dynamicznie tablicę i przelewający do niej tekst.

W podobnym duchu napisz metodę `void AppendAtEnd (const char* _Text)`, która do dotychczasowego napisu ma na końcu dokleić nowy. Zrealizuj to tak, aby zaalokować nową tablicę (przemyśl, jak wydobyć jej właściwy rozmiar), przypisać jej właściwe litery, a na końcu skasuj starą alokację i przypisz do wskaźnika `Text` adres nowej tablicy.

Co powinno być w destruktorze? Zakoduj go.

Przećwicz działanie Twojej funkcji, dając w funkcji `main` jakiś prosty przykład.

```
struct Napis {
  char* Text = nullptr ;
  int Size   = 0 ;
  Napis (const char*) ;
};

Napis::Napis (const char* _Text) {
  while (_Text[Size] != 0) Size++ ;
  Text = new char[ Size+1 ] ;
  for (int i = 0; i <= Size; i++)
    Text[i] = _Text[i];
  }
};
```

7. Wybierz klasy rozważane w tym zestawie i uogólnij je w szablony klas.
Następnie, podziel kod na pliki nagłówka i implementacyjne oraz plik klienta.
Upewnij się, że wiesz, jak kompilować taki podzielony kod.