Celem zadania jest implementacja **prostego kalkulatora**, który może wykonać „zaprogramowane" **operacje**, np. **dodawanie** podanej listy argumentów, **odejmowanie**, **mnożenie**, **dzielenie** oraz **potęgowanie** (1-argumentowe), a na wyświetlaczu pokaże wynik.

Funkcja **main** jest gotowa i nie wolno jej modyfikować poza komentowaniem/odkomentowywaniem.
Przykładowe wyniki zadania znajdują się w oddzielnym pliku **wyniki9a.pdf**.

W zadaniu **zabronione** jest stosowanie jawnego rozpoznawania typów (tj. **nie wolno** korzystać z mechanizmu **RTTI**, itp.).
Wykorzystujemy również właściwości klasy bazowej (**nie** redefiniujemy kodu)

Program ma składać się z następujących klas:

ETAP 1

Klasa: Operation – gotowa, tylko obejrzyj
Klasa abstrakcyjna, posiadająca:

- prywatne stałe pole ***name*** przechowujące nazwę działania (tj. **Addition**, **Subtraction**, **Multiplication**, **Division** lub **Power**)
- chronione stałe pole całkowite ***n*** przechowujące liczbę argumentów operacji
- publiczny konstruktor z parametrami: liczba argumentów operacji oraz nazwa operacji
- wirtualny destruktor
- publiczną **czysto wirtualną** metodę ***Result***, która otrzymuje (tylko) wskaźnik do tablicy **n** liczb rzeczywistych i zwraca wynik operacji na nich
- publiczną **czysto wirtualną** metodę ***Clone***, która odpowiada za tworzenie poprawnych kopii obiektów w klasach pochodnych
- zaprzyjaźniony **operator<<**, wykorzystujący wirtualną metodę ***Print***:
  name, number of arguments: ***m***
  (np.: Addition, number of arguments: 3)

Zaimplementuj publiczne klasy pochodne od klasy **Operation:**
Addition
Subtraction
Multiplication
Division
Power

Każda klasa pochodna posiada:

- konstruktor z parametrem określającym **liczbę argumentów**, domyślnie wartość 2 (wywołaj odpowiedni konstruktor klasy bazowej, w którym podaj też nazwę operacji – czyli nazwę tej klasy)
- w klasie **Power** parametr konstruktora (p>0) oznacza **wartość potęgi**, a liczba argumentów działania jest 1
- tylko klasa **Power** posiada prywatne pole przechowujące całkowity parametr p
- metodę ***Result*** *(parametrem jest wskaźnik tablicy argumentów)*
- metodę ***Clone***, tworzącą dynamiczną kopię obiektu klasy

W metodzie ***Result*** do pierwszego elementu tablicy (o indeksie 0) stosowany jest odpowiedni operator dla następnych argumentów z tablicy, np.

```
double arg[] = { 18,3,2,2 };
Subtraction   S3(3);
cout<<S3.Result(arg)<<endl;      //=18-3-2
```

ETAP 2a

Klasa: Calculator posiada:

- prywatne pole rzeczywiste ***value***, przechowujące obecnie pamiętaną wartość (bieżący wynik obliczeń na wyświetlaczu kalkulatora)
- prywatne stałe pole statyczne ***N*** zainicjowane wartością 10 (maksymalna liczba operacji w kalkulatorze)
- prywatną tablicę ***N*** elementową ***operation***, zawierającą wskaźniki do klasy bazowej **Operation**
- publiczny konstruktor bezparametrowy, w którym ustawia pole ***value*** na ***0*** oraz inicjalizuje tablicę ***operation*** wskaźnikami **nullptr**
- publiczny destruktor
- publiczna metoda ***AddOperation***, która – jeśli jest miejsce - wstawia wskaźnik przekazanego w parametrze obiektu jako kolejny element tablicy ***operation*** (wykorzystaj metodę ***Clone***)
- publiczna metoda ***Start*** wyświetla MENU, odpowiada za obliczenia i komunikację z użytkownikiem, w kolejnych krokach powinna wyświetlać ***value*** i dawać użytkownikowi możliwość wyboru kolejnych obliczeń.

W celu poprawnego działania metody ***Start*** zaimplementuj prywatną metodę ***Running***, która:

- wypisze dostępne operacje (zawartość tablicy ***operation***)
- wczyta wybór operacji (0÷9)
- wczyta wszystkie potrzebne argumenty operacji do dynamicznej tablicy (jako pierwszy element o indeksie 0 ustaw bieżącą ***value*** kalkulatora)
- wykona operację (met. ***Result***), której wynik zapisze w polu ***value***

ETAP 2b

W klasie **Division**, gdy wykryto jakikolwiek dzielnik równy 0 (w tablicy argumentów, poza jej pierwszym elementem), to należy **wyrzucić wyjątek** *(obiekt własnej klasy).*

Zdefiniuj własną klasę wyjątku jako **publiczną klasę pochodną** od ***exception***, gdzie w prywatnym polu klasy należy przechować ***indeks*** pierwszego niepoprawnego dzielnika (który spowodował błąd dzielenia).
Do projektu należy **dołączyć** nowy plik **`.h`**, w którym **zdefiniuj własną klasę wyjątku**.

Wyjątek należy przechwycić i obsłużyć w metodzie ***`Running`***.
W przypadku, gdy zaistnieje sytuacja wyjątkowa, należy wyświetlić **komunikat**:
`Bad argument i=1` (jeśli niepoprawne dzielenie spowodował argument o indeksie 1) i kontynuować MENU.

ETAP 3

W funkcji main zakomentuj ETAP 1 oraz ETAP 2 !!!

Należy „przerobić” hierarchię klas na klasy szablonowe:
Operation
Addition
Subtraction
Multiplication
Division
Power

Mają to być szablony klas zależne od **jednego parametru** określającego typ. Przewidujemy, że parametr będzie reprezentować domyślny typ **double** lub **complex<double>**
Należy włączyć nagłówek **complex**

Tak jak w ETAPie 1 i ETAPie 2 - **wewnątrz wszystkich klas** mają pozostać tylko nagłówki. Implementację szablonów wykonaj **poza** klasą.

Uwaga w pochodnych klasach szablonowych do odziedziczonego pola ***n*** (liczba argumentów operacji) należy odwoływać się: ***this->n***

ETAP 4

Należy „przerobić” klasę **Calculator** na klasę szablonową, podobnie jak klasy w ETAPie 3

Tak jak w ETAPie 1 i ETAPie 2 - **wewnątrz klasy** („ćwiczeniowo”) mają pozostać tylko nagłówki. Implementację szablonu wykonaj **poza** klasą.