Programowanie Zaawansowane FM i NI
Ćwiczenia 8

Zadanie 1 (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`. Wskazówka: `strcmp` w bibliotece `cstring`.

Zadanie 1 (b) Podziel kod z zadania 1 na pliki klasy oraz plik klienta.

- Skompiluj wpierw całość jednym poleceniem.
- Teraz skompiluj plik klasy do pliku obiektowego, a następnie skompiluj plik klienta, łącząc go („linkując") z plikiem obiektowym klasy do aplikacji.

Ciekawostka: na systemach Unixowych możesz wypisać spis obiektów pliku obiektowego, jak też aplikacji:

```
nm -C mystring.o
nm -C mystring.exe
```

Możesz również spróbować tak:

```
objdump -TC mystring.o
```

Zadanie 2 (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`.
- 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 (rodzaj kopiującego, ale arg. wejścia nie musi być const), oraz konstruktor 2-argumentowy, przyjmujący adres tablicy osób i jej rozmiar
- 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`.

Zadanie 2 (b) Podziel kod z zadania 2 na pliki dla dwóch klas osobno i plik klienta. Skompiluj wpierw całość jednym poleceniem. Następnie skompiluj pliki klas do plików obiektowych i skompiluj plik klienta oraz *zlinkuj* z plikami obiektowymi klas do aplikacji.

Zadanie 3 `MathTool` – statyczne metody klasy

Napisz klasę `MathTool`, która za pomocą metod statycznych udostępni użytkownikowi(-czce) narzędzia numeryczne dla podanej przez niego(-ej) funkcji, działające na danym przedziale:

– wartość najmniejszą funkcji (`MinVal`),
– miejsce zerowe (`Root`),
– całkę oznaczoną (`Integral`)
– przebieg pochodnej numerycznej (`Deriv`).

Algorytmy numeryczne powinny być jak najprostsze: skupiamy się na programowaniu. Każda z metod ma przyjmować funkcję w argumencie wejścia. Klasa ma też posiadać statyczne pole `double dx`, zainicjowane np. do 0.01, które będzie używane w narzędziach jako krok skanowania i które użytkownik może zmieniać.

Aby zgrabnie podawać funkcję, napisz klasę `Funkcja`, której polami będą: wskaźnik funkcyjny typu `double (*myfun) (double)` i dziedzina funkcji (`double xmin, xmax`). Klasa ta powinna posiadać odpowiedni konstruktor oraz:

– `double operator() (double x)`, który zwraca wartość funkcji dla danego `x`.

Uwaga: w ten sposób obiekt tej klasy staje się *funktorem* . Oznacza to, że np. dla obiektu

```
Funkcja myFun ( cos , -5. , 5. );
```

można napisać:

```
cout << myFun ( 0.5 ) ;
```

Metody klasy `MathTool` mogą działać na typie `double`. Metody te powinny przyjmować obiekt klasy `Funkcja` i przedział pracy [`x1`, `x2`] .
Metoda `Deriv` ma na celu wypróbkowanie przebiegu pochodnej (ilorazu różnicowego) w przedziale pracy co krok `dx`, co oznacza pewne `N` punktów. Przed jej wywołaniem w `main` użytkownik powinien wytworzyć dwie tablice o liczbie elementów przynajmniej `N`, gotowe do wypełnienia kolejnymi X_i i $F'(X_i)$, których adres poda przez argumenty wejścia `Deriv` . Metoda ma wypełnić te tablice wartościami.

W funkcji `main` utwórz obiekt klasy `Funkcja`, któremu przypisz wskaźnik funkcji `sin`.

– Wykorzystując narzędzia, wypisz wartość najmniejszą na przedziale [-1, 1].
– Następnie zmniejsz dx do np. 1e-4 i wypisz wynik całki oznaczonej w przedziale [0, π] oraz miejsce zerowe w przedziale [2 , 4].
– Następnie, zwiększ `dx` do 0.1 i wytwórz dwie *puste* tablice, do których metoda `Deriv` wpisze zestaw X_i i $F'(X_i)$ w przedziale [-1, 1].
– W funkcji `main` wypisz zestaw tych punktów.

Na koniec: podziel kod na pliki poświęcone klasom (2 x 2 pliki) i plik klienta.

Uwaga: jeżeli klasa zawiera pole statyczne, to przypisanie jemu wartości nie może być w pliku .h, tylko w pliku implementacyjnym .C tej klasy.