Operatory na rzecz typu TString

Dopiszmy w definicji klasy operator[], dzięki któremu potraktujemy obiekt jak tablicę

```
class TString { public:
   char& operator[]( size_t n );
   const char& operator[]( size_t n ) const;
};
```

```
// w pliku tstring.cpp umieszczamy definicje
char& TString::operator[]( size_t n ) { // włączcie nagłówek <stdexcept>
  // na przykład sprawdźmy czy podane n jest poprawne
  if ( !ptr ) throw invalid_argument("pusty obiekt");
  if ( n >= 0 && n < len ) return ptr[ n ];
  // co zrobić gdy n jest niepoprawne?
  // czy można np. napisać return '0';   ?
  // ponieważ zwracamy „przez referencję” więc
  // jedynym rozsądnym wyjściem jest „zgłosić wyjątek”
  throw out_of_range("In TString::operator[] argument out of scope");
}
const char& TString::operator[]( size_t n ) const {  /* identycznie jak powyżej */ }
```

```
// w programie main np. :
s1 = "zebym nie byl pusty";
cout <<  s1[3] << endl;
// spróbuj też ze złym indeksem
```

TString jako kontener

Każdy kontener ma kilka użytecznych metod… są one krótkie w treści, można je definiować bezpośrednio w klasie:

```
class TString { public:
char* begin() { return ptr; }
char* end() { return ptr+len; }
size_t length() const { return len; } // hmm…
void clear() { delete [] ptr; ptr = nullptr; len = 0; }
bool empty() const { return len ? false : true; }
char& front() { return *ptr; }
const char& front() const { return *ptr; }
char& back() { return *(ptr+len-1); }
const char& back() const { return *(ptr+len-1); }
};
```

```
// tak naprawdę powinniśmy zwrócić
// iterator, ale na razie stworzymy
// uproszczone wersje
```

Teraz np. (w programie main) można użyć pętlę ”range-based loop”

```
for ( auto& n : s1 ) { ++n; } endl ( cout ); // zmieniamy literki
for ( const auto& n : s1 ) cout << n << " – "; cout << endl; // odczytujemy tylko
```

TString – size, insert, erase

Dopiszmy metodę size()

```
size_t size() const { return len; }
```

Czy potrafią Państwo zaimplementować kolejne pożyteczne metody:

```
char* insert( size_t pos, const char* c );
char* insert( size_t pos, char c );
// insert zwraca pierwszy element wstawiony
char* erase( size_t bpos = 0, size_t len = 0 );
// erase zwraca element za ostatnim usuniętym
```

```
// ponownie wersja
// uproszczona, powinny
// być iteratory
```

W programie powinno działać np:

```
s1.insert( 0, "wstawka" );
s1.insert( s1.size(), 'a' );
s1.erase( 0, s1.size() / 2 ); // erase bez argumentów czyli erase(0,0) robi to co clear()
```

TString – przykładowa implementacja insert

```
char* TString::insert( size_t pos, const char* c ) {
  if (pos >=0 && pos <= len) {
    size_t oldlen = len;
    len = len+strlen(c);
    char* tmp = new char[ len+1 ];
    strcpy( tmp, ptr );
    for (size_t i=pos; i<pos+strlen(c); ++i) {
       tmp[i] = c[i-pos];
    }
    for (size_t i=pos; i<oldlen; ++i) {
       tmp[i+strlen(c)] = ptr[i];
    }
    delete [] ptr;
    ptr = tmp;
    return ptr+pos;
  } else {
     throw out_of_range("zly argument");
  }
  return ptr;
}
```

W programie powinno działać:

```
s1.insert( 0, "napoczatek" );
s1.insert( s1.size(), "nakoniec" );
```

PROSZĘ SAMODZIELNIE
ZAIMPLEMENTOWAĆ WERSJĘ
Z ARGUMENTEM char c
ORAZ METODĘ erase

TString – jak wysłać obiekt do strumienia

Aby możliwe było bezpośrednie wstawienie obiektu typu TString do strumienia, potrzeba przeciążyć operator<<

operator<< ma dwa argumenty: **strumień „wyjściowy”**, **obiekt TString**
Taka kolejność argumentów wymusza, że operator<< będzie funkcją globalną.

Zdefiniujemy ją w pliku o nazwie np. operatory.cpp
i dodatkowo zamkniemy w przestrzeni nazw **MojeOperatory**

A gdzie deklaracja? Zwykle w jakimś pliku nagłówkowym, ale teraz…

friend – poprzedzające deklarację funkcji (tutaj: globalnej) jest „deklaracją przyjaźni” – konieczna, ponieważ zewnętrzna (tu: globalna) funkcja **nie ma** dostępu do części prywatnej klasy (czyli pól „ptr” i „len”). Dopiero gdy klasa zadeklaruje przyjaźń, to jest dostęp.

```
// w dowolnym miejscu wewnątrz klasy TString (deklaracja przyjaźni)
friend std::ostream& MojeOperatory::operator<<
                                ( std::ostream& strumien, const TString& s );
```

TString – jak wysłać obiekt do strumienia

Aby możliwe było bezpośrednie wstawienie obiektu typu TString do strumienia, potrzeba przeciążyć operator<<

operator<< ma dwa argumenty: **strumień „wyjściowy”**, **obiekt TString**
Taka kolejność argumentów wymusza, że operator<< będzie funkcją globalną.

Zdefiniujemy ją w pliku o nazwie np. operatory.cpp
i dodatkowo zamkniemy w przestrzeni nazw **MojeOperatory**

A gdzie deklaracja? Zwykle pliku nagłówkowym:

plik operatory.h

```
#ifndef OPERATORY_H
#define OPERATORY_H
#include <iostream>
#include <fstream>
class TString; // wystarczy deklaracja

namespace MojeOperatory {  // definiujemy naszą przestrzeń nazw
  std::ostream& operator<<( std::ostream& strumien, const TString& s );
}
#endif
```

TString – wysyłamy do strumienia, czytamy ze strumienia…

```
#include "tstring.h"
using namespace std;
ostream& MojeOperatory::operator<<( ostream& strumien, const TString& s ) {
    return strumien << ( s.ptr ? s.ptr : "pusty" );
  }
```

dopiszmy w tstring.h

#include "operatory.h"

plik operatory.cpp

Proszę pamiętać o dopisaniu tego pliku (oprócz main.cc i tstring.cc) w linii kompilatora!

```
// w programie teraz możemy bezpośrednio wysłać do strumienia
// pamiętając o otwarciu naszej przestrzeni nazw!!!   using namespace MojeOperatory;
cout << "zawartość obiektu s5: " << s5 << endl;
```

Aby możliwe było bezpośrednie czytanie strumienia do obiektu typu TString, potrzeba przeciążyć operator>>

operator>> ma dwa argumenty: **strumień „wejściowy”**, **obiekt TString** **bez przydomka const !**

Deklarujemy w przestrzeni nazw MojeOperatory, definiujemy w pliku **operatory.cpp**

Pamiętajmy o „deklaracji przyjaźni” (wewnątrz klasy TString)

```
friend std::istream&
        MojeOperatory::operator>>( std::istream& strumien, TString& s );
```

TString – jak wczytać zawartość strumienia do obiektu?

Idea czytania ze strumienia podobna do operacji przypisania, tzn. stara zawartość usuwana, a nowa umieszczana w odpowiednio pojemnej tablicy (do wskaźnika ptr).

```
// uprościmy sobie zadanie, korzystając z tymczasowego obiektu string, dzięki
// któremu nie będziemy się martwić o to, jak dużo wczytamy ze strumienia…
istream& MojeOperatory::operator>>( istream& strumien, TString& s ) {
  string tmp; // pamiętajmy o nagłówku <string>
  getline( strumien, tmp ); // wszystko najpierw ląduje do tmp
  // teraz trzeba przepisać
  delete [] s.ptr;
  s.len = tmp.length();
  if ( s.len > 0 ) {
    s.ptr = new char[ s.len + 1 ];
    strcpy( s.ptr, tmp.c_str() ); // metoda c_str() zwraca const char* z obiektu string
  } else {
     s.ptr = nullptr;
  }
  return strumien;
}
```

Dlaczego obiekt typu std::string nie ma bezpośredniej konwersji do typu **char*** / **const char***
Ponieważ jeśli dzieje się to niejawnie,
to jest to z reguły niepożądane!

TString – dodajmy c_str() i pomyślmy o konwersji…

Dobrym pomysłem jest dodanie definicji metod c_str() w wersji zwykłej i stałej, na wzór metody istniejącej w klasie std::string

```
// napiszmy te definicje szybko wewnątrz klasy TString, w części publicznej
char* c_str() { return ptr; }
const char* c_str() const { return ptr; }
```

Przypomnijmy sobie, że w naszej klasie TString coś na kształt „niejawnej konwersji” już ma miejsce, spowodowane to jest istnieniem konstruktora o jednym argumencie. Wtedy program „ma wiedzę” jak zbudować TString z const char*

```
// jak pamiętamy, możliwe jest
TString s6 = "konwersja, ja nie jestem TString";
// by zablokować takie zjawisko, należy poprzedzić deklarację c-tor specyfikatorem explicit
explicit TString( const char* c = nullptr );
```

Napiszmy teraz operator konwersji (metoda składowa w klasie TString):

```
operator char*() { return ptr; } // sprawdź też explicit operator char*();
operator const char*() const { return ptr; } // j.w. z explicit
// wtedy w programie możliwe jest również zapisanie np.
char* ch = s6;
char tabl[100];
strcpy( tabl, s6 );
```

proszę sprawdzić co się stanie z tymi przykładami gdy poprzedzimy powyższe deklaracje **explicit**

A gdy mamy też użyty operator[] zobaczymy… błąd.

TString – implementacje paru składowych…

Często można kolejne metody implementować w oparciu o już napisaną. Np.

```
// deklaracja w klasie TString
char* insert( size_t pos, char c );
// implementacja w oparciu o insert( size_t, const char* );
char* TString::insert( size_t pos, char c ) {
  return insert( pos, string( { c } ).c_str() ); // { } uniwersalna inicjalizacja
}
// metody push_back – jedna z tych wspólnych metod kontenerów standardowych
void push_back( char c ) { insert( len, c ); }
void push_back( const char* c ) { insert( len, c ); }
```

Dodajmy też licznik kontrolujący liczbę wyprodukowanych obiektów TString

```
// deklaracja pola statycznego i definicja metody statycznej w klasie TString
static size_t count; // w części prywatnej
static size_t getN() { return count; } // w części publicznej
// gdzieś poza klasą… np. przed main() definicja tej składowej statycznej
size_t TString::count;
```

Czy potrafisz teraz dopisać w odpowiednich funkcjach ++ i -- stanu zmiennej count?

TString – przeciążmy jeszcze przydatne operatory + oraz +=

Zdeklarujmy (dwuargumentowy) operator+ oraz operator+= jako funkcje globalne:

```
// deklaracja w przestrzeni nazw MojeOperatory
TString operator+( const TString& a, const TString& b );
TString& operator+=( TString& a, const TString& b );
Proszę zwrócić uwagę na to, co zwraca + oraz co zwraca +=
// implementacja (w pliku operatory.cc)
TString MojeOperatory::operator+( const TString& a, const TString& b ) {
  // musi powstać obiekt tymczasowy – lokalny
  // proszę ponownie zaimplementować za pomocą insert…
}
TString& MojeOperatory::operator+=( TString& a, const TString& b ) {
  return …… ; // zapiszmy w jednej linijce, korzystając z +
}
```

Oczywiście napiszmy również kilka linii testów w programie…

```
cout << s1+s2 << endl; // obserwujmy komunikaty konstruktorów i destruktorów…
s1 = s1 + "ten argument niedopasowany, a dziala";
s1 += s1; // dodanie do samego siebie
```

TString – przeciążmy jeszcze przydatne operatory + oraz +=

Zdeklarujmy (dwuargumentowy) operator+ oraz operator+= jako funkcje globalne:

```
// deklaracja w przestrzeni nazw MojeOperatory
TString operator+( const TString& a, const TString& b );
TString& operator+=( TString& a, const TString& b );
Proszę zwrócić uwagę na to, co zwraca + oraz co zwraca +=
// implementacja (w pliku operatory.cc)
TString MojeOperatory::operator+( const TString& a, const TString& b ) {
  TString tmp ( a );
  tmp.insert( tmp.size(), b );   // konwersja albo trzeba b.c_str() jako drugi argument
  return tmp;
}
TString& MojeOperatory::operator+=( TString& a, const TString& b ) {
  return a = a + b;
}
```

Oczywiście napiszmy również kilka linii testów w programie…

```
cout << s1+s2 << endl; // obserwujmy komunikaty konstruktorów i destruktorów…
s1 = s1 + "ten argument niedopasowany, a dziala";
s1 += s1; // dodanie do samego siebie
```

TString – obiekt funkcyjny

Bardzo ważną i efektywną częścią programowania jest używanie / pisanie obiektów funkcyjnych. W roli takich obiektów najczęściej występują klasy (struktury) mające przeciążony operator oraz tzw. wyrażenia lambda. Przeciążmy operator()

```
// musi to być składowa klasy, zatem w części publicznej:
void operator()( const char& c ) {
  push_back( c );
}
// może robić „cokolwiek”, ja wymyśliłem, że będzie wstawiać na koniec pojemnika
// „użycie” takiego operatora wygląda jak wywołanie funkcji – ale – używamy obiektu:
s6('A'); //całość „wygląda” jak wywołanie metody s6(char), ale s6 to obiekt TString
// w rzeczywistości jest to takie wywołanie: s6.operator()( ’A’ );
```

Ciekawsze będzie jednak użycie algorytmu… np. przepiszmy z obiektu typu std::string literka po literce, wstawiając do obiektu TString. Użyjemy for_each

```
// uwaga! potrzebny nagłówek <algorithm>
string ss("abcdefghijkl");
s2 = for_each(ss.begin(), ss.end(), s2 ); // proszę sprawdzić co się stanie jeśli nie będzie s2 =
cout << s2 << endl; // algorytm for_each „przenosi” s2… więc bez s2 = zostałoby puste !
```