Zbudujemy klasę

Definicję klasy zapiszmy w pliku **tstring.h**

```
#ifndef TSTRING_H
#define TSTRING_H
#include <cstring>
 // w pliku nagłówkowym NIE
 // otwieramy przestrzeni std
class TString {
 public:
  // interfejs
 private:
  // implementacja
  // składowe klasy
 protected:
  // póki nie będziemy dziedziczyć,
  // to pole nas nie interesuje
}; // pamiętaj o średniku
#endif
```

pola dostępu do klasy z zewnątrz

Zapiszmy też prosty plik **main.cpp**

```
#include "tstring.h"
#include <iostream>
using namespace std;
int main () {
  TString s1;
}
```

size_t jest nazwą (typedef) na bezznakowy typ całkowity wystarczająco pojemny aby opisać wielkość dowolnego obiektu
[m.in. zwracany przez **sizeof**]

```
class TString {
 private:
   char* ptr = nullptr;
   std::size_t len = 0;
};
```

Zdefiniujmy konstruktor

Zdeklarujmy konstruktor (**c-tor**) :

```
class TString {
 public:
   TString( const char* s = nullptr );
};
```

Metody **definiujemy** w **tstring.cpp**

```
#include "tstring.h"
#include <iostream>

using namespace std;

TString::TString( const char* s ) :
                    ptr(nullptr), len(0) {
if (s > 0) {
   len = strlen(s);
   ptr = new char[ len + 1 ];
   strcpy( ptr, s );
}
#ifdef DEBUG
   cout << "TString c-tor " << len << " - " << ( ptr ? ptr : "pusty" ) << endl;
#endif
}
```

Kompilujemy dodatkowo dodając w linii
opcję –D *definiowananazwa*
czyli –D DEBUG (może być bez spacji), przykładowo:
g++4.9.2 –std=c++14 –DDEBUG
main.cc tstring.cpp –o prog

Możemy teraz dopisać **w main** kolejny obiekt, np.
TString s2("inicjalizacja slowem");

Zdefiniujmy destruktor

Zdeklarujmy destruktor (**d-tor**):

```
class TString {
 public:
   TString( const char* s = nullptr );
  ~TString();
};
```

Definicję destruktora, jak i wszystkich kolejnych metod składowych klasy, dopisujemy jako ciąg dalszy (czyli poniżej definicji konstruktora) w pliku **tstring.cpp**

w pliku tstring.cpp jako dalsza część

```
TString::~TString() {

#ifdef DEBUG
  cout << "TString d-tor " << len << " - " << ( ptr ? ptr : "pusty" ) << endl;
#endif

delete [] ptr;
}
```

Śledzenie pokrokowe programu (debuger) **gdb**
Kod trzeba skompilować z flagą –g (oraz nie używać flag optymalizujących takich jak –O –O2 itd.)
http://www.yolinux.com/TUTORIALS/GDB-Commands.html

Zdefiniujmy konstruktor kopiujący

Zdeklarujmy konstruktor kopiujący (**cc-tor**):

```
class TString {
 public:
   TString( const char* s = 0 );
   TString( const TString& s );
  ~TString();
};
```

Możemy dopisać w **main.cpp**

```
// poniżej to nie jest przypisanie
TString s3 = s2;
// albo tak:
TString s3 ( s2 );
// albo tak:
TString s3 { s2 };
```

```
TString::TString( const TString& s ) :
                ptr(nullptr), len( s.len ) {
  if (len > 0) {
    ptr = new char[ len + 1 ];
    strcpy( ptr, s.ptr );
  }

#ifdef DEBUG
  cout << "TString cc-tor " << len << " - " << ( ptr ? ptr : "pusty" ) << endl;
#endif
}
```

w pliku tstring.cpp jako dalsza część

Operacja podobna do tej z konstruktora.

Zdefiniujmy operator przypisania kopiujący

Zdeklarujmy operator= kopiujący:

```
class TString { public:
TString& operator=
        ( const TString& s );
};
```

Możemy dopisać w **main.cpp**

```
// poniżej jest przypisanie, bo obiekt
// po lewej już istnieje
s3 = "alfa beta";
s3 = s2;
```

```
TString& TString::operator=
                (const TString& s ) {
 if ( this != &s ) {  // if ( *this != s ) {
     delete [] ptr;      ptr = nullptr;      len = s.len;
     if ( len > 0 ) {
        ptr = new char[ len + 1 ];
        strcpy( ptr, s.ptr );
     }
   }
#ifdef DEBUG
   cout << "TString copy operator= " << len << " - " << ( ptr ? ptr : "pusty" ) << endl;
#endif
return *this; // nie zapomnij zwrócić obiektu!
}
```

this – specjalny wskaźnik, który otrzymuje każda niestatyczna składowa klasy, a w którym zapisany jest adres bieżącego obiektu, na którego argumentach działać ma metoda

Zdefiniujmy konstruktor przenoszący

Konstruktor przenoszący (**mvc-tor**):

```
class TString {
 public:
    TString( TString&& s );
};
```

Możemy dopisać w **main.cpp**

```
// move „maskuje” tożsamość obiektu
TString s4 = std::move( s2 );
// std::move będzie niepotrzebne
// jeśli inicjalizować będzie obiekt
// tymczasowy np. zwracany przez
// funkcję jako wartość
```

```
TString::TString( TString&& s ) :
                    ptr(s.ptr), len(s.len) {
// obiekt źródłowy zostaje pozbawiony zasobów
// ale pozostawiony w stanie do dalszego użytku czyli można coś np. do niego przypisać
  s.ptr = nullptr;
  s.len = 0;

#ifdef DEBUG
  cout << "TString mvc-tor " << len << " - " << ( ptr ? ptr : "pusty" ) << endl;
#endif
}
```

Operacja przenoszenia dzieje się automatycznie wtedy, gdy obiekt źródłowy „nie ma nazwy” i „nie ma adresu”

Zdefiniujmy operator przypisania przenoszący

Zdeklarujmy operator= przenoszący:

```
class TString { public:
 TString& operator=
                ( TString&& s );
};
```

Możemy dopisać w **main.cpp**

```
// ponownie „ukrywamy obiekt”
// za pomocą std::move
s3 = std::move( s1 );
```

```
TString& TString::operator=
                ( TString&& s ) {
 if ( this != &s ) {
     delete [] ptr;  // usuń dotychczasowy zasób
     len = s.len;   // typy proste się tylko (po prostu) kopiuje
     ptr = s.ptr;  // tu zabieramy adres wskaźnika (przeniesienie praw własności)
     s.len = 0;  // obiekt, któremu zabraliśmy, zerujemy
     s.ptr = nullptr;  // wskaźnik również zerujemy
 }
#ifdef DEBUG
   cout << "TString move operator= " << len << " - " << ( ptr ? ptr : "pusty" ) << endl;
#endif
  return *this;  // nie zapomnij zwrócić obiektu!
}
```

Copy elision, obiekty do przenoszenia

copy elision – mechanizm pozwalający kompilatorowi na optymalizację polegającą na ominięciu **cc-tor** lub **mvc-tor** i bezpośrednim zbudowaniu / umieszczeniu obiektu w docelowym miejscu. Mimo tego formalnie tam, gdzie wymagane są **cc-tor** czy **mvc-tor** (choć niewykorzystane), muszą one być poprawnie sformułowane.

en.cppreference.com/w/cpp/language/copy_elision

Można wyłączyć optymalizację za pomocą flagi
-fno-elide-constructors

Jak najczęściej dostajemy obiekt „do przeniesienia”?

Poprzez funkcję zwracającą go przez wartość. Prosty przykład

```
// funkcja zwracająca wartość
TString fun(const char* c) {
  return TString(c);
}
```

```
// funkcję fun definiujemy jako globalną, np.
// przed programem main
// następnie w programie spróbujmy:
TString s5 = fun("konstruktor przenoszący");
TString s6 = s4; // konstruktor kopiujący
// zaobserwujmy wynik gdy działa optymalizacja
// oraz gdy (używając powyższej flagi) jest
// ona wyłączona
```

```
// spróbujmy C++14
auto fun(const char* c) {
  return TString(c);
}
```

UWAGA: kompilacja z flagą
-std=c++14