Minikurs języka C

Lista zadań nr 3

Na zajęcia 8 kwietnia 2020
grupa AKu

Za każde zadanie można otrzymać od 0 do 10 punktów.

Zadanie 1. (autor MGa[1]) W tym zadaniu zaimplementujesz strukturę podobną do `vector` z biblioteki standardowej `C++`.

Skorzystaj z następującego typu danych:

```
typedef struct vector {
    size_t size;
    size_t capacity;
    size_t element_size;
    void* data;
} vector;
```

Struktura ta działa tak samo jak tablica (oferuje dostęp do i-tego el. w czasie stałym), ale dodatkowo można dodawać elementy na koniec i usuwać elementy z końca. Realizowana jest ona jako tablica `data`, która na początku ma pewien ustalony rozmiar `capacity`. Dodawanie elementów to po prostu wstawianie ich do tej tablicy. W momencie kiedy dodajemy element, na który nie ma miejsca w tablicy (czyli `size==capacity` w momencie kiedy chcemy wstawić element), usuwamy całą tablicę i tworzymy nową, dwa razy większą.

Można pokazać, że dodawanie elementów do naszej struktury będzie nas kosztować amortyzowany czas stały na każdy z dodanych elementów. Wynika to z tego, że jeśli po wykonaniu wszystkich operacji nasza struktura ma rozmiar n, to w sumie podczas wykonania programu zaalokowaliśmy nie więcej niż $2n$ komórek pamięci (zastanów się dlaczego tak jest).

Twoja implementacja powinna obsługiwać następujące funkcje:

```
void vector_init(vector* v, size_t capacity, size_t element_size)
    // inicjalizuje nowy, pusty vector
void vector_copy(vector* destination, vector* source)
    // kopiuje zawartosc vectora source do vectora destination
void vector_assign(vector *v, int i, void* elem) //modyfikuje element na i-tej pozycji
void* vector_at(vector *v, int i) //zwraca wskaznik na i-ty element
void* vector_front(vector *v) //zwraca wskaznik na pierwszy element
void* vector_back(vector *v) //zwraca wskaznik na ostatni element
void vector_push_back(vector* v, void* elem) // dodaje element na koniec
void* vector_pop_back(vector* v) // usuwa element z konca oraz zwraca na niego wskaznik
void vector_clear(vector* v) // czysci strukture
void vector_resize(vector* v, int n) // ustala rozmiar struktury na n
void vector_reserve(vector* v, int n)
    // ustala rozmiar zarezerwowanej tablicy na przynajmniej n
size_t vector_size(vector* v) // zwraca rozmiar struktury
size_t vector_capacity(vector* v) // zwraca rozmiar zaalokowanej tablicy
int vector_empty(vector* v) // sprawdza czy struktura jest pusta
```

Jeśli masz wątpliwości co do tego jak mają działać powyższe funkcje, to warto zobaczyć specyfikację ich odpowiedników z biblioteki standardowej `C++`. Jest ona dostępna np. tutaj: `http://www.cplusplus.com/reference/vector/vector/`.

Utwórz plik nagłówkowy `vector.h` z odpowiednimi deklaracjami oraz plik `vector.c` z implementacją. Ponadto przetestuj też swoją implementację. W tym celu należy dostarczyć też plik `main.c`, w którym powinny być umieszczone odpowiednie testy.

[1] Zmodyfikowany wariant zadania zaproponowanego przez MGa

Zadanie 2. (autor PWit) W tym zadaniu zaimplementujesz kilka standardowych operacji na napisach (łańcuchach znaków) w języku `C`. Dla przypomnienia:

- Znak ma typ `char`, a stałe znakowe pisze się w apostrofach, np. instrukcja `char znak = 'c'`; deklaruje zmienną znak przypisując jej wartość początkową `'c'`.
- Napis to ciąg znaków zakończonych znakiem końca napisu o kodzie `ASCII 0`, czyli znakiem `'\0'`.
- Stała napisowa to ciąg znaków ograniczonych znakami `"`, np. stałą napisową jest `"Nietoperz"`.
- Stała napisowa jest napisem, zatem występuje w niej niewidoczny znak końca napisu. Stąd np. `sizeof("Nietoperz")` jest równe 10, a nie 9.
- Zmienna napisowa to a) tablica znaków lub b) wskaźnik do bloku pamięci zawierającego ciąg znaków. W obydwu wypadkach końcowym znakiem w tablicy/bloku musi być znak końca napisu. Przykłady: a) `char napis1[10] = "Nietoperz"`, b) `char *napis2 = "Nietoperz"`.
- Zmienne napisowe można modyfikować, o ile nie ma ku temu przeciwskazań. Dla przykładu, instrukcja `napis1[0] = 'n'` sprawi, że napis zapamiętany w tablicy `napis1` zmieni się na `"nietoperz"`. Instrukcja `napis2[0] = 'n'` może (ale nie musi – to zależy od środowiska wykonania) wygenerować błąd czasu wykonania. A to dlatego, że wskaźnik `napis2` może wskazywać na niemodyfikowalną pamięć.
- Zmienną napisową będącą wskaźnikiem można traktować jak każdy inny wskaźnik, np. przypisywać jej nowoprzydzieloną pamięć za pomocą `malloc` lub za pomocą przypisania adresu już istniejącego bloku w pamięci. Oto przykłady: `char *napis3 = (char*) malloc(10)` oraz `char *napis4 = &napis1`. Te dwa wskaźniki wskazują na modyfikowalną pamięć, zatem poprawne są instrukcje `napis3[5] = 'n'` czy `napis4[0] = 'n'`.
- Napis można wypisać na ekran przy pomocy funkcji `printf` (np. `printf("%s", napis1)`) oraz wczytać z klawiatury przy pomocy funkcji `scanf` (np. `scanf("%s", napis1)`). W przypadku wczytywania z klawiatury musisz uważać na liczbę wczytanych znaków!
- Tego, czy dwie zmienne napisowe zawierają ten sam napis nie da się stwierdzić przy pomocy operatora `==` (wyrażenie `napis1 == napis2` nie zadziała). Napisy trzeba porównywać znak po znaku. Podobnie, napisów nie da się przypisywać za pomocą operatora przypisania `=`. Trzeba je kopiować znak po znaku. Nie ma również możliwości przypisania czy wyodrębnienia fragmentu napisu (analogu Pythonowego operatora `[:]`).

Mając powyższe na uwadze, zaimplementuj dane funkcje zgodnie ze specyfikacją podaną na stronie `http://www.cplusplus.com/reference/cstring/`. W rozwiązaniu nie można używać żadnych funkcji z nagłówka `string.h`. W operacjach na łańcuchach znaków używaj wyłącznie **składni wskaźnikowej** (arytmetyka na wskaźnikach, porównywanie ich oraz operacje dereferencji). Używanie **składni tablicowej** (operator indeksowania `[·]`) jest zabronione.

```
char * strcat ( char * destination, const char * source );
char * strncat ( char * destination, const char * source, size_t num );
int strcmp ( const char * str1, const char * str2 );
int strncmp ( const char * str1, const char * str2, size_t num );
```

Możesz założyć, że wskaźniki `destination` podawane jako argumenty w powyższych funkcjach zawsze wskazują na bloki pamięci o wystarczającej długości. Następnie napisz program, który testuje powyższe funkcje, (np. pobierając dane z wejścia). Przekonaj siebie samego i sprawdzającego te zadanie, że funkcje są poprawnie zaimplementowane dobierając "złośliwe" przypadki testowe – wypisz te przypadki w komentarzu.

Zadanie 3. Pojawi się w systemie SKOS.