Wprowadzenie do programowania w języku C
grupa RKr, wtorek 16:15-18:00
lista nr 7 *(na pracownię 26.11.2019) (wersja 1)*

Zadanie 1. [10p na pracowni lub 5p po pracowni**]**

[3.33(3)p | **A]** W języku C to Ty zarządzasz pamięcią, którą alokujesz i użytkujesz. Gdy pamięć jest niepotrzebna musisz ją zwolnić (w wielu językach robi to za Ciebie tzw. Garabge Collector). Gdy jedna z wielu operacji w dużej funkcji się nie powiedzie trzeba zwolnić wszystkie alokacje, które zostały utworzone do momentu powstania błędu, wołając funkcję free(.). Napisz funkcję **tidyMemkeeping**(), która wykonuje 8 alokacji pamięci (8 razy woła funkcję malloc(.), ale bez użycia pętli), a gdy jedna z nich się nie powiedzie (malloc(.) da w wyniku NULL), zwolni wszystkie alokacje, które do tego momentu odniosły sukces. Twoim wyzwaniem jest użycie funkcji free(.) dokładnie jeden raz dla każdej alokacji, co daje w sumie 8 wywołań.
*wskazówka: Wykorzystaj umiejętnie instrukcję **goto**, i nie przejmuj się ludźmi, którzy mówią, że nie powinna istnieć :)*

[3.33(3)p | **B]** Napisz makro **memberoff__**(T, m), które w wyniku da położenie pola m w strukturze T, położenie jest to dystans (uwzględniający padding) wyrażony w bajtach mierzony od początku struktury.
*wskazówka: Skorzystaj z możliwości dereferencji NULL jako r-wartości oraz pobrania adresu pola struktury porpzez **&**.*
Lub ogólniej, wykorzystaj fakt, że arytmetyka adresów na stałych wartościach działa w czasie kompilacji.

[3.33(3)p | **C]** W języku C, jeśli nie zainicjujemy stworzonego obiektu lub uszkodzimy pamięć, w której się znajduje, to będzie on zawierał przypadkowe/błędne wartości. Do typu **Box** dodaj pole, które pomoże Ci rozstrzygnąć z bardzo wyskokim pradwopodobieństwem, czy pamięć jest zainicjowana oraz nieuszkodzona. Napisz funkcję o sygnaturze „**bool isValid**(**Box*** b)”, która sprawdzi poprawność pamięci zawierającej b.
wskazówka: Dodaj pole, które jest wskaźnikiem i zastanów się na co powinien wskazywać.
Pole przyjmujące wartości 0 lub 1 nie jest dobrym rozwiązaniem.

Zadanie 2. [10p] A
Język C nie posiada szablonów (*ang. templates*) znanych z języka C++, jednak makra i szablony są podobne. Szablon można napisać w formie makro rozwijające się do implementacji funkcji. Przykładowa implementacja i użycie poniżej.
uwaga: Język C nie pozwala na aliasowanie funkcji, trzeba do nazwy funkcji doklejać rozróżniający identyfikator.

```
#define TEMPLATE_next(T) static T next##_##T(T num) { return num + (T)1; }
TEMPLATE_next(long) // instancjacja szablonu
long x = next_long(6); // x = 7
```

Napisz szablonową funkcję sortującą tablicę n liczb działającą dla typów: **byte**, **short**, **int**, **long**, **float**, **double**. Wybierz metodę sortowania, która nadaje się do rekurencyjnej implementacji, np. mergesort lub quicksort.

Zadanie 3. [10p] Dostępne w serwisie SKOS.

Zadanie 2. [10p] B

Napisz funkcję o sygnaturze „**void pack3_fp32**(**float** f0, **float** f1, **float** f2, **long long*** fff_out)” oraz funkcję o sygnaturze „**void unpack3_fp32**(**float*** f0, **float*** f1, **float*** f2, **long long** fff_in)”. Pierwsza spakuje trzy liczby typu **float** do jednego 8-bajtowego słowa (typu **long long**), a druga wykona operację odwrotną.
Zauważ, że osiągamy redukcję z 12 bajtów do 8 bajtów, asymptotycznie redukujemy pamięć o połowę.
Taka forma kompresji zadziała zwykle wtedy, gdy trzy pakowane liczby będą wielokrotnościami pewnej liczby uzyskanymi poprzez mnożenia przez potęgi gwójki. Przykładowo, jeśli jako bazę wybierzemy liczbę π, to trzy liczby do pakowania możemy wybrać spośród liczb: π·(0.125), π· (0.25), π· (0.5), π·1, π·2, π·4, π·8.
Jeśli podane 3 liczby nie posiadają takiej właściwości, to funkcja pack3_fp32(.) powinna zgłosić błąd.
Możesz eksperymentować wybierając liczby o innej bazie, np. liczba Eulera, pierwsza lub „zwyczajna” liczba.

wskazówka: *Wypisz liczby (ze zbioru o wspólnej bazie) w reprezentacji szesnastkowej i znajdź dwa bajty, które są stałe. Wykorzystaj makra PRINT_HEX_VALUE(x) [lista 1] oraz MERGE_4BYTES(by3, by2, by1, by0) [lista 3]. Do operacji bitowych na typie **float** możesz użyć rzutowania „**float** nf; **long*** bf = (**long***)((**float***)&nf);”, gdzie nf (numerical float) to przetwarzana liczba, a bf (bit float) to wskaźnik na reprezentację dziesiętną umożliwiający operacje bitowe, których efekt możesz odczytać z nf.*

Zadanie 2. [10p] C

Napisz funkcję o sygnaturze „**long weakBits_fp32**(**float** f, **int** k)”, która znajdzie k najsłabszych bitów typu **float**. Najsłabsze bity to takie, które niosą najmniejszą ilość informacji, zmiana wartości tych bitów przynosi najmniejsze zmiany. Wartość wynikowa funkcji weakBits_fp32(f, k) ma mieć zapalone tylko te bity, które oznaczają słabe bity typu **float**. Możesz ją wypisać poprzez colorizeBinNybbles(.) i BITWORD_8(.) [lista 3].
Wypisz błąd jaki wprowadza modyfikacja słabych bitów, wybierz jedną małą (np. **0.3333333**) i jedną dużą (np. **333.3333**) liczbę typu **float**. Przykładowe użycie: PRINT_LSB_VALUE(weakBits_fp32(**0.3333333f**, **2**), **4**).

uwaga: *Możesz założyć, że parametr k przyjmuje wartości od 1 do 4. Podane liczby są niereprezentowalne, więc zaakceptuj to, że literał 0.3333333f zamieni się na 0.33333331346511840820312 5. Jeśli wybierzesz inne liczby, uważaj na pułapki w postaci infinity,-infinity oraz NaN.*

wskazówka: *Możesz modyfikować każdy bit i sprawdzać jak bardzo zmieniła się wartość liczby. Zastanów się, który z trzech segmentów bitów (znak, wykładnik, mantysa) będzie zawierał te bity. Do operacji bitowych na typie **float** możesz użyć rzutowania „**float** nf; **long*** bf = (**long***)((**float***)&nf);”, gdzie nf (numerical float) to przetwarzana liczba, a bf (bit float) to wskaźnik na reprezentację dziesiętną umożliwiający operacje bitowe, których efekt możesz odczytać z nf.*

Zadanie 2. [10p] D

Napisz funkcję o sygnaturze „**float*** **buildFloatChain**(**float* const** fArray, **long* const** tArray_**b**, **int** length)”, która zbuduje łańcuch (tablicę) liczb typu **float**, w którym każdy fArray[k] będzie miał wbudowaną etykietę tArray_**b** [k] na najsłabszych bitach typu **float** [wyszukiwane przez weakBits_fp32]. Łańcuch musisz zakończyć wartownikiem (*ang. sentinel*) oznaczającym koniec, to zabierze Ci jeden bit w każdej liczbie lub doda jedną liczbę na końcu łańcucha, zdecyduj, którą opcję zaimplementujesz.
Napisz również funkcję o sygnaturze „**float*** **filterFloatChain**(**float const * const** fChain, **int** tag_**b**)”, która ze zbudowanego wcześniej łańcucha stworzy zupełnie nowy łańcuch poprzez wybranie tylko tych liczb z oryginalnego łańcucha, które posiadają etykietę *tag_**b***.

uwaga: *Możesz założyć, że maksymalna liczba bitów na etykiety to 2 lub 3, czyli za **b** podstaw 2 lub 3. Pozwala Ci to na wyrażenie 2^(**b**) etykiet.*

wskazówka: *Zdefiniuj typ strukturalny, który pozwoli Ci na łatwy dostęp do potrzebnych bitów, możesz korzystać z konceptów takich jak **union**, **struct**, **bit-field** i typów **float** oraz **long**. Alternatywnie możesz użyć sztuczki ze wskaźnikiem: „**float** nf; **long*** bf = (**long***)((**float***)&nf);”.*