Alokator pamięci, wymagania na ocenę 5,0:

Celem jest rozbudowanie alokatora o następujące funkcjonalności:

1. Wszystkie funkcje alokujące pamięć
   (tj. `heap_malloc`, `heap_calloc` i `heap_realloc`) muszą zwracać adres pamięci będący wielokrotnością długości słowa maszynowego. Pamiętaj, że długość słowa maszynowego jest zależna od kompilatora i platformy docelowej. Sugerowany sposób sprawdzania to `sizeof(void*)`. Długość słowa jest zawsze potęgą liczby `2`.
2. Alokator, wyszukując miejsce w pamięci na alokację żądanego przez użytkownika bloku, musi względnić każde wolne miejsca pamięci. Przykład:
   1. Stan wejściowy (trzy zaalokowane wcześniej bloki).

```
AaaaaaaaaaBbbbbbbbbbbbbbbbbbbCccccccc
free=0     free=0              free=0
size=9     size=18             size=7
```

   5. Blok `B` zostaje zwolniony (`heap_free(B)`).

```
AaaaaaaaaaFfffffffffffffffffCccccccc
free=0     free=1              free=0
size=9     size=18             size=7
```

   9. Zaalokowany zostaje blok `D`: `D = heap_malloc(8)`.

```
AaaaaaaaaaDddddddddFfffffffffCccccccc
free=0     free=0    free=1    free=0
size=9     size=8    size=9    size=7
```

      W wyniku tej operacji w wolnym obszarze (`Fff`) utworzone zostały dwa bloki. Pierwszy to `D` o wielkości żądanej przez użytkownika (`8`) a drugi to blok wolny (size=`9`). Zwróć uwagę, że suma długości obu bloków jest równa długości obszaru wolnego sprzed wywołania `heap_malloc`.

   13. Zaalokowany zostaje blok `E`: `E = heap_malloc(8)`.

```
AaaaaaaaaaDddddddddEeeeeeeee.Cccccccc
free=0     free=0    free=0    free=0
size=9     size=8    size=8    size=7
```

      W wyniku tej alokacji w wolnym obszarze utworzony został blok `E`. Pozostała przestrzeń do bloku `C` (oznaczona `.`) jest zbyt mała aby zmieścić w niej strukturę kontrolną wolnego bloku. Przy zwalnianiu bloku `E` nieużyty obszar (`.`) należy odzyskać:

   17. Zwolniony zostaje blok `E`: `heap_free(E)`.

```
AaaaaaaaaaDddddddddFfffffffffCccccccc
free=0     free=0    free=1    free=0
size=9     size=8    size=9    size=7
```

Zwolnienie `E` spowodowało odzsykanie nieużytego obszaru (zmiana `size` z `8` na `9`).

3. **Legenda**:

- `A`, `B`, `C`, `D`, `E`, `F` - Struktury kontrolne.
- `a`, `b`, `c`, `d`, `e` - Obszary danych poszczególnych bloków
- `f` - Nieużywany obszar bloku wolnego o nagłówku `F`.
- `.` - Nieużywany obszar pamięci w danym bloku, między obszarem przydzielonym użytkownikowi (o długości `size`) a strukturą kontrolną następnego bloku.

Ponadto należy zaimplementować następujące funkcje debugujące:

```
void* heap_malloc_debug(size_t count, int fileline, const char* filename);
void* heap_calloc_debug(size_t number, size_t size, int fileline, const char*
filename);
void* heap_realloc_debug(void* memblock, size_t size, int fileline, const
char* filename);

void* heap_malloc_aligned_debug(size_t count, int fileline, const char*
filename);
void* heap_calloc_aligned_debug(size_t number, size_t size, int fileline,
const char* filename);
void* heap_realloc_aligned_debug(void* memblock, size_t size, int fileline,
const char* filename);
```

Rodzina funkcji `heap_*_debug` działa jak ich odpowiedniki bez przyrostka `_debug`, przy czym działanie to jest rozszerzone o mechanizm opisywania alokowanych/modyfikowanych bloków. Do każdego zmodyfikowanego/zaalokowanego bloku funkcje dodają informacje o nazwie pliku źródłowego (`filename`) oraz linii (`fileline`) z której zostały wywołane. Przykładowe wywołanie takich funkcji to:

```
void* ptr = heap_malloc_debug(current_heap, 1024, __FILE__, __LINE__);
```

lub poprzez makro:

```
#define MALLOC(__size) heap_malloc_debug(current_heap, (size_t)(__size),
__FILE__, __LINE__);
(....)
uint8_t* some_data = (uint8_t*)MALLOC(1234);
```

Wszystkie funkcje API sterty, wraz z definicjami struktur i typów danych, należy umieścić w pliku nagłówkowym `heap.h`. Natomiast faktyczne implementacje należy umieścić w pliku źródłwym `heap.c`.

Uwagi

- W tym zadaniu funkcja `main` nie jest testowana. Wykorzystaj ją do testów.
- Funkcja `custom_sbrk()` dostępna jest zarówno w raportach z kompilacji (Dante dołącza ją automatycznie) jak i w repozytorium GitHuba https://github.com/tomekjaworski/SO2/tree/master/heap_sbrk-sim.
- Nie używaj rzeczywistej funkcji `sbrk()`. W przypadku Biblioteki Standardowej `libc` za jej wykorzystanie odpowiada standardowa implementacja alokatora pamięci (znany już `malloc` itp). Ręczne uruchamianie `sbrk()` spowoduje desynchronizację informacji, posiadanych przez tę bibliotekę, i fizycznie przydzielonej pamięci. Uniemożliwi to poprawne działanie wszystkim funkcjom Biblioteki Standardowej, korzystających z wbudowanego alokatora (np. `fopen`).
- Link do pliku CMake dla środowiska CLion: https://pastebin.com/DGr27FLE.

Przydatne informacje:

- https://medium.com/@andrestc/implementing-malloc-and-free-ba7e7704a473
- https://danluu.com/malloc-tutorial/