Alokator pamięci, wymagania na ocenę 3,0:

Celem projektu jest przygotowanie managera pamięci do zarządzania stertą własnego programu. W tym celu należy przygotować własne wersje funkcji `malloc`, `calloc`, `free` oraz `realloc`. Całość należy uzupełnić dodatkowymi funkcjami narzędziowymi, pozwalającymi na monitorowania stanu, spójności oraz defragmentację obszaru sterty.

Przygotowane funkcje muszą realizować następujące funkcjonalności:

- Standardowe zadania alokacji/dealokacji zgodne z API rodziny `malloc`. Należy dokładnie odwzorować zachowanie własnych implementacji z punktu widzenia wywołującego je kodu.
- Możliwość resetowania sterty do stanu z chwili uruchomienia programu.
- Możliwość samoistnego zwiększania regionu sterty poprzez generowanie żądań dla systemu operacyjnego.
- Płotki.

Przestrzeń adresowa pamięci, dla której należy przygotować managera, będzie zawsze zorganizowana jako ciąg stron o długości 4KB.

Funkcje alokujące pamięć muszą uwzględniać płotki bezpośrednio *przed* i bezpośrednio *po* bloku przydzielanym użytkownikowi - między nimi nie może być pustych bajtów.

Zadaniem płotków jest ułatwienie wykrywania błędów typu *One-off* w taki sposób, iż każdy płotek ma określoną i znaną zawartość oraz długość. Jego naruszenie (zamazanie wartości) oznacza, że kod użytkownika niepoprawnie korzysta z przydzielonego mu bloku pamięci i powienien zostać przerwany/poprawiony. Płotek powinien mieć co najmniej `1` bajt, ale zaleca się aby był potęgą `2` i `>= 2`.

Przykład:

```
Pusty obszar sterty: (...) ---------------------------------------- (...)
Obszar po alokacji:  (...) CHHHHbbbTTTT.CCCCHHHHbbbbbTTTT...CCCCHHHHbb (...)
                                          ^
                                    |<----- blok ----->|
```

Legenda:

- `C` - struktura kontrolna bloku (nagłowek bloku),
- `H` - płotek górny (head),
- `T` - płotek dolny (tail),
- `b` - blok użytkownika,

- `^` - pierwszy bajt bloku użytkownika, na który wskazują wskaźniki zwracane przez funkcje `heap_*`.

Płotki muszą być ułożone w pamięci bloku w taki sposób, aby między nimi a blokiem uzytkownika (`b`) **nie było wolnej przestrzeni**. Zwróć zatem uwagę na kilka wolnych bajtów (`...`) między płotkiem dolnym (`T`) a nagłówkiem następnego bloku. W przypadku, gdy położenie bloków nagłówkowych wyrównane jest do 4/8 bajtów (tzn. adresy bloków położone są w adresach podzielnych, bez reszty, przez 4 i 8) to miejsce na to wyrównanie (ang. *paddign*) znajduje się między ostatnim bajtem płotka dolnego (`T`) a pierwszym bajtem nagłówka kolejnego bloku (`C`).

Przykłady (wyrównanie do 4 znaków; wyłącznie na potrzeby ilustracji):

```
(...) ...CCCCHHHHbTTTT...CCCCTTTT...              // malloc(1)
(...) ...CCCCHHHHbbTTTT..CCCCTTTT...              // malloc(2)
(...) ...CCCCHHHHbbbTTTT.CCCCTTTT...              // malloc(3)
(...) ...CCCCHHHHbbbbTTTTCCCCTTTT...              // malloc(4)
(...) ...CCCCHHHHbbbbbTTTT...CCCCTTTT...  // malloc(5)
(...) ...CCCCHHHHbbbbbbTTTT..CCCCTTTT...  // malloc(6)
(...) ...CCCCHHHHbbbbbbbTTTT.CCCCTTTT...  // malloc(7)
(...) ...CCCCHHHHbbbbbbbbTTTTCCCCTTTT...  // malloc(8)
                ^
```

Funkcje do implementacji

Przedstawione poniżej funkcje należy zaimplementować zgodnie z podaną specyfikacją. 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`.

```c
int heap_setup(void);
```

Funkcja `heap_setup` inicjuje (organizuje) stertę w obszarze przeznaczonej do tego pamięci. Wielkość obszaru pamięci dostępnej dla sterty nie jest znana w chwili startu programu.

W rzeczywistych przypadkach kod obsługujący stertę korzysta z funkcji systemu operacyjnego `sbrk()`. Jednak na potrzeby tego projektu należy korzystać z funkcji `custom_sbrk()` o prototypie danym plikiem nagłówkowym `custom_unistd.h`. Jest ona zgodna ze swoim odpowiednikiem (`sbrk()`) zarówno co do parametrów jak i zachowania, widocznego z punktu widzenia kodu wywołującego (tutaj - alokatora).

Wartość zwracana:

- `0` – jeżeli sterta została poprawnie zainicjowana, lub

- `-1` – jeżeli sterty nie udało się zainicjować, np. system odmówił przydziału pamięci już na starcie.

Uwaga: Funkcja ta jest wykorzystywana w testach do przywracania warunków początkowych sterty, ale nie jest testowana oddzielnie. Oznacza to, że błędy pojawiające się w kolejnych funkcjach i testach mogą być związane z błędnym działaniem `heap_setup()`.

```
void heap_clean(void);
```

Funkcja `heap_clean` zwraca **całą** przydzieloną pamięć do systemu operacyjnego. Funkcja powinna pracować poprawnie również w przypadku uszkodzonej sterty (np. zamazane płotki).

Innymi słowy zadaniem funkcji `heap_clean` jest:

1. Zwrócenie całej przydzielonej pamięci systemowi operacyjnemu (patrz `sbrk`).
2. Wyzerowane całej pamięci stanowiącej struktury kontrolne sterty.

Wynik działania funkcji `heap_clean` pozostałe funkcje API alokatora muszą widzieć stertę jako *niezainicjowaną*. W takim przypadku, aby możliwe było ponowne skorzystanie ze sterty, należałoby uruchomić funkcję `heap_setup()`.

```
void* heap_malloc(size_t size);
void* heap_calloc(size_t number, size_t size);
void* heap_realloc(void* memblock, size_t count);
void  heap_free(void* memblock);
```

Funkcje `heap_malloc`, `heap_calloc`, `heap_free` oraz `heap_realloc` mają zostać zaimplementowane zgodnie ze specyfikacją Biblioteki Standardowej GNU C Library (glibc), dostępnej pod adresem http://man7.org/linux/man-pages/man3/malloc.3.html.

```
void* heap_malloc(size_t size);
void* heap_calloc(size_t nmemb, size_t size);
```

Funkcja `heap_malloc` przydziela pamięć zgodnie z następującym algorytmem:

1. Jeżeli na stercie dostępny jest wolny blok pamięci, o rozmiarze większym bądź równym rozmiarowi żądanemu przez użytkownika (parametr `size`), to przydzielany jest **pierwszy** napotkany taki obszar (patrząc względem początku sterty).

   Przydzielony obszar nie musi być podzielony na obszar zajęty i wolny (jak ma to miejsce w zadaniu 1.5 Prosty malloc).

2. Jeżeli nie ma dostępnego wolnego bloku o żądanym rozmiarze, to funkcja żąda od systemu operacyjnego rozszerzenia sterty do takiej wielkości, aby operacja alokacji mogła się powieść a następnie posługuje się schematem z punktu 1.
3. W przypadku braku dostępnej pamięci (i odmowy SO na żądanie `sbrk`) funkcja zwraca `NULL`.
4. W przypadku wykrycia uszkodzenia sterty funkcja nie podejmuje żadnej akcji i zwraca `NULL`.

Uwagi

- Funkcja `heap_malloc` zwraca adres pierwszego bajta pamięci dostępnej dla użytkownika, a nie struktury kontrolnej lub płotków.
- W testach jednostkowych funkcji `heap_malloc` wykorzystywane są następujące funkcje:
    - `heap_setup` - Do zainicjowania wewnętrznych struktur alokatora.
    - `heap_clean` - Do zwrócenia całej użytej pamięci do systemu i ustawienia sterty w stan "niezainicjowana". Po zakończeniu każdego z testów cała wykorzystana pamięć powinna być zawsze zwrócona do systemu.
    - `heap_free` - Do zwalniania zaalokowanej pamięci.
    - `heap_validate` - Do sprawdzenia spójności sterty po wykonywanych operacjach.
    - `get_pointer_type` - Do sprawdzenia poprawności typu wskaźnika, zwracanego przez funkcję `heap_malloc`.

```
void* heap_calloc(size_t number, size_t size);
```

Funkcja `heap_calloc` ma zostać zaimplementowana, aby w pełni oddać zachowanie funkcji `calloc` z Biblioteki Standardowej C. Należy pamiętać, że różnicą między `malloc` a `calloc` jest inicjalizacja przydzielonej pamięci (w przypadku tej drugiej).

```
void* heap_realloc(void* memblock, size_t size);
```

Funkcja `heap_realloc` zmienia rozmiar bloku pamięci `memblock`.

Jeżeli `size` z wywołania `heap_realloc` jest **mniejszy** od wielkości bloku `memblock` to funkcja powinna jedynie zmniejszyć jego rozmiar. Ponadto,
jeżeli `size` jest **równa** wielkości bloku `memblock` to funkcja nie podejmuje żadnej akcji, zwracając jednocześnie niezmieniony wskaźnik `memblock`.

W przypadku, gdy `size` jest **większy** od bieżącej wielkości bloku `memblock` funkcja `heap_realloc` przydziela nową pamięć, zgodnie z następującym algorytmem:

1. Jeżeli za blokiem pamięci, wskazywanym przez `memblock`, dostępny jest obszar/blok wolnej pamięci o rozmiarze większym, bądź równym rozmiarowi żądanemu przez użytkownika `count` minus aktualny rozmiar `memblock` **to** obszar wskazywany przez `memblock` jest powiększany.

   Przykład: Jeżeli na stercie są dwa bloki: **A**(zajęty, size=100 bajtów) i zaraz po nim **B**(wolny, size=300 bajtów) a użytkownik chce rozszerzyć wielkość **A** do 150 bajtów, to `heap_realloc` zwiększa rozmiar bloku **A** kosztem przesunięcie granicy **AB** w głąb bloku **B**. Po takiej operacji blok **A** będzie miał długość 150 bajtów a blok **B** 250 bajtów. Pamiętaj o tym, że wszystkie bloki muszą mieć swoje struktury kontrolne a bloki zajęte jeszcze płotki!

2. Jeżeli obszar wskazywany przez `memblock` jest na końcu sterty a wielkość sterty jest zbyt mała na pomyślne zwiększenie wielkości bloku `memblock` do `size` bajtów, to należy poprosić system o dodatkową pamięć (patrz `sbrk()`).
3. Jeżeli wskaźnik `memblock` jest równy `NULL` to funkcja pozostaje tożsama z funkcją `heap_malloc`.
4. Jeżeli obszar wskazywany przez `memblock` nie może zostać powiększony do `size` bajtów (bo pamięć znajdująca się w kierunku powiększania jest już zajęta) to funkcja musi przydzielić nową pamięć na `size` bajtów w innym miejscu sterty, następnie przenieść zawartość poprzedniego bloku do nowego. Osierocony blok musi zostać zwolniony 😉.
   - Jeżeli operacja `heap_malloc` się nie powiedzie to funkcja zwraca `NULL` i nie modyfikuje obszaru pamięci `memblock` (nie modyfikuje sterty).

- W przypadku wykrycia uszkodzenia sterty funkcja nie podejmuje żadnej akcji i zwraca `NULL`.

Uwaga: Powyższy algorytm jest *oficjalnym i powszechnie przyjętym sposobem* działania funkcji realloc wszędzie tam, gdzie jest ona implementowana.

```
size_t   heap_get_largest_used_block_size(void);
```

Funkcja `heap_get_largest_used_block_size` zwraca rozmiar największego bloku, przydzielonego użytkownikowi. Zwraca wartość `0` jeżeli:

- sterta nie została zainicjowana,
- żaden blok nie został przydzielony użytkownikowi,

- dane w obszarze sterty są uszkodzone (np. zamaznane płotki, uszkodzona struktura bloku).

```
enum pointer_type_t get_pointer_type(const void* const pointer);
```

Funkcja `get_pointer_type` zwraca informację o przynależności wskaźnika `pointer` do różnych obszarów sterty. Funkcja ta, na podstawie informacji zawartych w strukturach sterty, klasyfikuje wskaźnik `pointer` i zwraca jedną z wartości typu wyliczeniowego `pointer_type_t`:

```
enum pointer_type_t
{
    pointer_null,
    pointer_heap_corrupted,
    pointer_control_block,
    pointer_inside_fences,
    pointer_inside_data_block,
    pointer_unallocated,
    pointer_valid
};
```

Wartości typu `pointer_type_t` mają następującą interpretację:

- `pointer_null` – Przekazany wskaźnik jest pusty – posiada wartość `NULL`.
- `pointer_heap_corrupted` - Sterta jest uszkodzona.
- `pointer_control_block` – Przekazany wskaźnik wskazuje na obszar struktur wewnętrznych/kontrolnych sterty.
- `pointer_inside_fences` - Przekazany wskaźnik wskazuje na bajt, będący częścią dowolnego płotka dowolnego zajętego bloku.
- `pointer_inside_data_block` – Przekazany wskaźnik wskazuje na *środek* któregoś z bloków, zaalokowanych przez użytkownika. Przez *środek* należy rozumieć adres bajta innego niż pierwszego danego bloku.
- `pointer_unallocated` – Przekazany wskaźnik wskazuje na obszar wolny (niezaalokowany) lub poza stertę. Typ ten dotyczy zarówno przestrzeni wolnej (wnętrze bloku wolnego) jak i niewielkich przestrzeni bajtowych, pozostałych po wyrównianiach do długości słowa danych CPU (do rozszerzenia na 4,0+).
- `pointer_valid` – Przekazany wskaźnik jest poprawny. Wskazuje on na pierwszy bajt dowolnego bloku, przydzielonego użytkownikowi. Każdy wskaźnik, zwracany przez `heap_malloc`/`heap_calloc`/`heap_realloc` musi być typu `pointer_valid`. I tylko takie wskaźniki ma przyjmować funkcja `heap_free`.

```
int heap_validate(void);
```

Funkcja wykonuje sprawdzenie spójności sterty.

Wartość zwracana:

- `0` – Jeżeli sterta jest poprawna/nieuszkodzona.
- `1` – Jeżeli sterta jest uszkodzona - naruszone zostały płotki (obszar przed i za obszarem pamięci, przydzielonej użytkownikowi).
- `2` – Jeżeli sterta nie jest zainicjowana (patrz funkcja `heap_setup`).
- `3` – Jeżeli sterta jest uszkodzona w taki sposób, że zamazany został obszar zajmowany przez struktury kontrolne sterty.

Uwaga! Wbrew pozorom jest to funkcja najtrudniejsza do napisania. **Musi ona być odporna na wszelkie możliwe uszkodzenia sterty.** Jej wywołanie **nie ma prawa** doprowadzić do przerwania działania programu ze względu na błędne dane w obszarze sterty.

Podstawowe pytanie, jakie musi zadać sobie projektant takiej funkcji jest następujące: *W jaki sposób sprawdzić, czy struktura opisująca stertę nie została uszkodzona?*

Ponieważ jeżeli nie została uszkodzona, to można z niej odczytać granice regionu przydzielonego stercie (patrz funkcja `heap_setup`) i względem tej granicy walidować każdy wskaźnik w strukturach wewnętrznych sterty. Bez tej informacji i braku walidacji wskaźników zapisanych w strukturach sterty, ich dereferencja może zakończyć się zatrzymaniem procesu (*Segmentation fault*).

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, korzystającym 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/