

## Tema de casă 3 - Alocator de memorie

---

**În această temă veți implementa un alocator simplu de memorie, similar sistemului malloc/free.**

### Responsabili:

- Darius Neațu (2016) [mailto:darius.neatu@cti.pub.ro]
- Rareș Cheșeli (2016) [mailto:rares.cheseli@stud.acs.upb.ro]

**Autor inițial:** Ștefan Bucur [mailto:stefan.bucur@gmail.com]

**Deadline soft: 20.12.2016 Deadline hard: 12.01.2017**

**Ultima modificare: 28.11.2016**

Menționez că pentru testare (pe vmchecker) se folosește o mașină virtuală pe 32 de biți. Arhiva de test folosește un astfel de binar. În caz că sistemul vostru de operare de pe mașina fizică este pe 64 de biți, sugerez să faceți testarea finală și pe o mașină (virtuală sau nu) de 32 de biți.

## Obiective

---

În urma realizării acestei teme, studentul va fi capabil:

- să lucreze cu pointerii pentru a manipula date stocate într-o memorie;
- să lucreze cu mecanismul de alocare de memorie din biblioteca standard C;
- să înțeleagă și să implementeze conceptele din spatele unui alocator de memorie;
- să lucreze cu funcții de manipulare a șirurilor de caractere;
- să înțeleagă și să genereze hărți de memorie.

## Recomandări

---

### ATENȚIE!

- Recomandăm să recapitulați noțiunile menționate anterior.
- Enunțul temei este lung. Înțelegerea enunțului face parte din rezolvarea temei. NU răspundem pe forum la întrebări pentru care răspunsul există deja în enunț. Recomandăm să citiți enunțul de mai multe ori până când sunteți siguri că l-ați înțeles.
- Recomandăm să vă gândiți foarte bine la abordare înainte să implementați și să modularizați cât mai bine codul.
- Pentru coding style puteți citi aici [http://ocw.cs.pub.ro/courses/programare/coding-style].

## Enunțul temei

---

Așa cum ați aflat deja acum două săptămâni, locuitorii din Codeland sunt în război deschis cu barbarii din Common Town. Aceștia din urmă sunt însă cei care, cu mulți ani înainte de trecerea la un stil de viață beligerant, erau recunoscuți pentru înclinația lor către știință. Acea a fost

perioada în care au scris biblioteca *stdlib*. Odată cu începerea războiului, locuitorii din Codeland s-au văzut nevoiți să renunțe la folosirea acestei biblioteci. Totuși, pentru a putea efectua calculele necesare pentru viața de zi cu zi, ei sunt obligați de împrejurări să își scrie ei înșiși funcțiile de care au nevoie. Momentan, prioritatea o reprezintă funcțiile pentru alocarea dinamică a memoriei. Ajutați-i pe locuitorii din Codeland să implementeze funcțiile de care au nevoie.

## Cerința temei

---

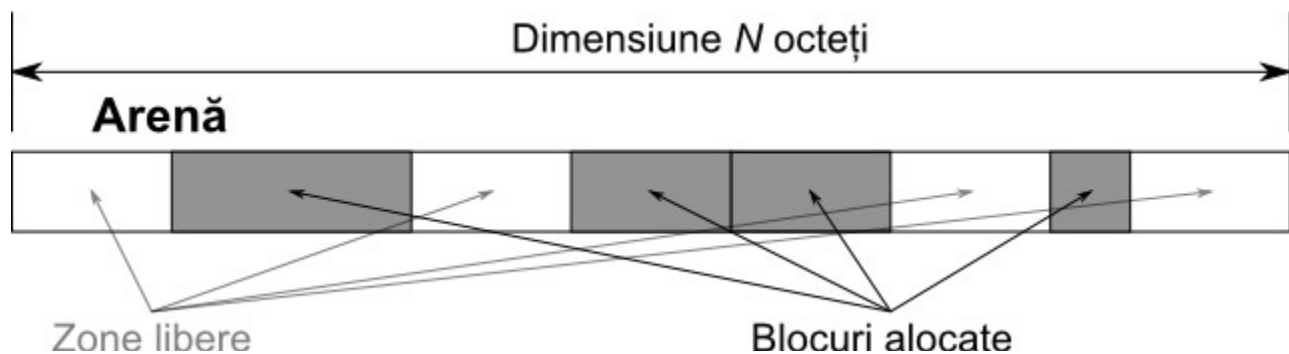
Programul vostru va trebui să realizeze o simulare a unui sistem de alocare de memorie. Programul va primi la intrare comenzi de alocare, alterare, afișare și eliberare de memorie, și va trebui să furnizați la ieșire rezultatele fiecărei comenzi. Nu veți înlocui sistemul standard `malloc()` și `free()`, ci vă veți baza pe el, alocând la început un bloc mare de memorie, și apoi presupunând că acela reprezintă toată "memoria" voastră, pe care trebuie s-o gestionați.

## Funcțiile unui alocator de memorie

---

Un alocator de memorie poate fi descris, în termenii cei mai simpli, în felul următor:

- Primește un bloc mare, compact (fără "găuri"), de memorie, pe care trebuie să-l administreze. Acest bloc, în terminologia de specialitate, se numește *arenă*. De exemplu, sistemul de alocare cu `malloc()` are în gestiune *heap*-ul programului vostru, care este un segment special de memorie special rezervat pentru alocările dinamice.
- Utilizatorii cer din acest bloc, porțiuni mai mici, de dimensiuni specificate. Alocatorul trebuie să găsească în arenă o porțiune continuă liberă (nealocată), de dimensiune mai mare sau egală cu cea cerută de utilizator, pe care apoi o marchează ca ocupată și întoarce utilizatorului adresa de început a zonei proaspăt marcată drept alocată. Alocatorul trebuie să aibă grijă ca blocurile alocate să nu se suprapună (să fie *disjuncte*), pentru că altfel datele modificate într-un bloc vor altera și datele din celălalt bloc.
- Utilizatorii pot apoi să ceară alocatorului să elibereze o porțiune de memorie alocată în prealabil, pentru ca noul spațiu liber să fie disponibil altor alocări.
- La orice moment de timp, arena arată ca o succesiune de blocuri libere sau ocupate, ca în figura de mai jos.



O problemă pe care o are orice alocator de memorie este cum este ținută evidența blocurilor alocate, a porțiunilor libere și a dimensiunilor acestora. Pentru această problemă există în general două soluții:

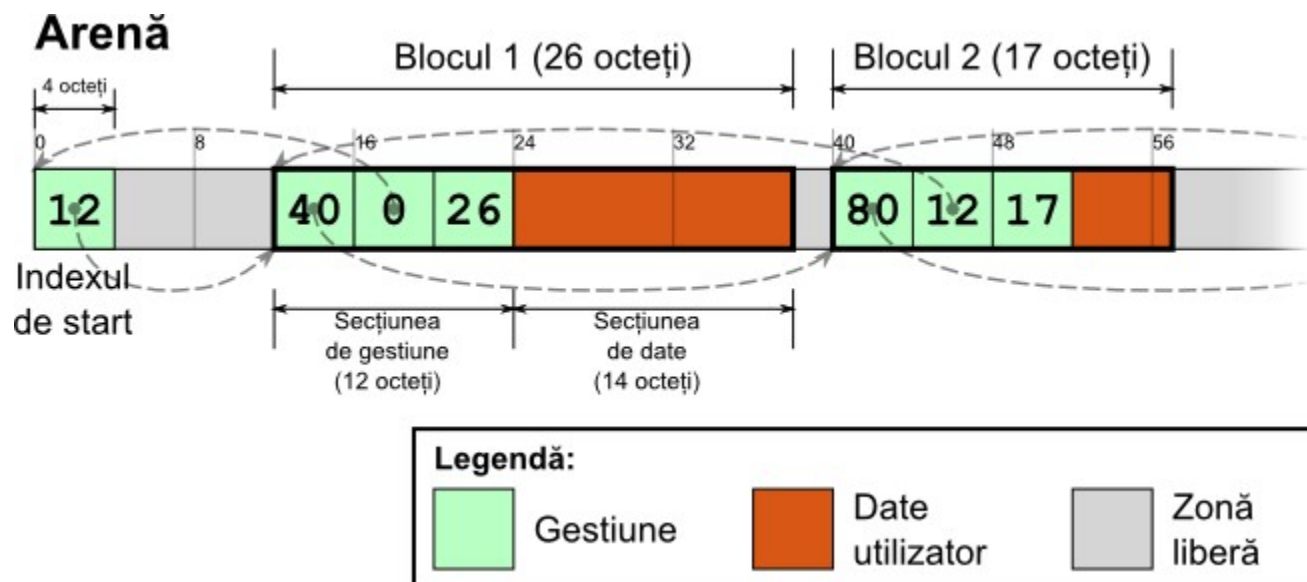
- Definirea unor zone de memorie separate de arenă care să conțină liste de blocuri și descrierea acestora. Astfel, arena va conține doar datele utilizatorilor, iar secțiunea separată va fi folosită de alocator pentru a găsi blocuri libere și a ține evidența blocurilor alocate.
- Cealaltă soluție, pe care voi o veți implementa în această temă, folosește arena pentru a stoca informații despre blocurile alocate. Prețul plătit este faptul că arena nu va fi disponibilă în totalitate utilizatorilor, pentru că va conține, pe lângă date, și informațiile de gestiune, însă avantajul este că nu are nevoie de memorie suplimentară și este în general mai rapidă decât prima variantă.

Există mai multe metode prin care se poate ține evidența blocurilor alocate în arenă, în funcție de performanțele dorite. Voi va trebui să implementați un mecanism destul de simplu, care va fi prezentat în secțiunea următoare. Deși nu este extrem de performant, se va descurca destul de bine pe dimensiuni moderate ale arenei (de ordinul MB).

## Structura arenei

În continuare vom considera arena ca pe o succesiune (vector) de  $N$  octeți (tipul de date `unsigned char`). Fiecare octet poate fi accesat prin indexul său (de la 0 la  $N-1$ ). Vom considera că un index este un întreg cu semn pe 32 de biți (tipul de date `int` pe un sistem de operare pe 32 de biți). De asemenea, va fi nevoie câteodată să considerăm 4 octeți succesivi din arenă ca reprezentând valoarea unui index. În această situație, vom considera că acel index este reprezentat în format 'little-endian' (revedeți [exercițiile](#) de la laboratorul de pointeri pentru mai multe detalii și citiți acest articol [<http://en.wikipedia.org/wiki/Endianness>] mult mai descriptiv), și astfel vom putea face cast de la un pointer de tip `unsigned char *` la unul de tip `int*`, pentru a accesa valoarea indexului, stocată în arenă.

Figura de mai jos ilustrează structura detaliată a arenei, în decursul execuției programului:



## Structura unui bloc

Se poate observa că fiecare bloc alocat de memorie (marcat cu un chenar îngroșat) constă din două secțiuni:

- Prima secțiune, de `gestiune`, este reprezentată de 12 octeți (`3 * sizeof(int)`) împărțiți în 3 întregi (a câte 4 octeți fiecare). Cei trei întregi reprezintă următoarele:
  - Primul întreg reprezintă indexul de start al blocului următor de memorie din arenă (aflat imediat "la dreapta" blocului curent, dacă privim arena ca pe o succesiune de octeți de la stanga la dreapta). Se consideră că un bloc începe cu secțiunea de gestiune, și toți indicii la blocuri vor fi tratați ca atare. Dacă blocul este ultimul în arenă (cel mai "din dreapta"), atunci valoarea primului întreg din secțiune va fi 0.
  - Cel de-al doilea întreg din secțiune reprezintă indexul de start al blocului imediat anterior din arenă. Dacă blocul este primul în arenă, atunci valoarea acestui întreg va fi 0.
  - Cel de-al treilea întreg din secțiune reprezintă lungimea totală a blocului, adică lungimea celor două secțiuni la un loc (nu doar a datelor alocate utilizatorului).
- A doua secțiune conține datele `efective` ale utilizatorului. Secțiunea are lungimea în octeți egală cu dimensiunea datelor cerută de utilizator la apelul funcției de alocare. Indicele returnat de alocator la o nouă alocare reprezintă începutul acestei secțiuni din noul bloc, și 'nu' începutul primei secțiuni, întrucât partea de gestiune a memoriei trebuie să fie complet transparentă pentru utilizator.

## Înlănțuirea blocurilor

După cum se poate observa din figura de mai sus, la începutul arenei sunt rezervați 4 octeți care reprezintă `indicele de start` - indicele primului bloc (cel mai "din stânga") din arenă. Dacă arena nu conține nici un bloc (de exemplu, imediat după inițializare), acest indice este 0.

Indicele de `start` marchează începutul lanțului de blocuri din arenă: din acest indice putem ajunge la începutul primului bloc, apoi folosind secțiunea de gestiune a primului bloc putem găsi începutul celui de-al doilea bloc, și așa mai departe, până când ajungem la blocul care are indexul blocului următor 0 (este ultimul bloc din arenă). În acest mod putem traversa toate blocurile din arenă, și de asemenea să identificăm spațiile libere din arenă, care reprezintă spațiile dintre două blocuri succesive.

Este de remarcat faptul că lanțul poate fi parcurs în ambele sensuri: dintr-un bloc putem ajunge atât la vecinul din dreapta, cât și la cel din stânga.

De asemenea, atunci când este alocat un bloc nou sau este eliberat unui vechi, '`lanțul de blocuri trebuie modificat`'. Astfel, la alocarea unui nou bloc de memorie, trebuie să ținem cont de următoarele:

- Spațiul liber în care este alocat noul bloc este mărginit de cel mult două blocuri vecine. Secțiunile de gestiune ale acestor vecini trebuie modificate astfel:
  - Indexul blocului următor din structura de gestiune a blocului din stânga trebuie să indice către noul bloc. Dacă blocul din stânga nu există, atunci este modificat indicele de start.

- Indexul blocului precedent din structura de gestiune a blocului din dreapta trebuie să indice către noul bloc. Dacă blocul din dreapta nu există, atunci nu se întâmplă nimic.
- Secțiunea de gestiune a noului bloc va conține indicii celor doi vecini, sau 0 în locul vecinului care lipsește.

La eliberarea unui bloc, trebuie modificate secțiunile de gestiune a vecinilor într-o manieră similară ca la adăugare.

## Funcționarea programului

---

Programul vostru va trebui să implementeze o serie de operații de lucru cu arena, care vor fi lansate în urma comenzilor pe care le primește la intrare. Fiecare comandă va fi dată pe câte o linie, și rezultatele vor trebui afișate pe loc. Secțiunea următoare prezintă sintaxa comenzilor posibile și formatul de afișare al rezultatelor.

Întrucât pentru testare comenzile vor fi furnizate prin redirectare dintr-un fișier de intrare, iar rezultatele vor fi stocate prin redirectare într-un alt fișier, programul vostru nu va trebui să afișeze nimic altceva în afara formatului specificat (de exemplu, nu trebuie să afișați mesaje de tipul "Introduceți comanda: ").

Folosiți funcțiile de manipulare a șirurilor de caractere pentru a citi și interpreta comenzile date la intrare. Este recomandată combinația `fgets()` și `strtok()` pentru o implementare elegantă.

Pentru o mai bună organizare a codului vostru, implementați execuția fiecărei comenzi într-o funcție separată. De asemenea, gândiți-vă ce variabile trebuie păstrate globale, iar pe restul declarați-le local.

## Formatul comenzilor

---

Programul vostru va trebui să accepte următoarele comenzi la intrare:

### 1. **INITIALIZE <N>**

- Această comandă va fi apelată prima, și va trebui să realizeze inițializarea unei arene de dimensiune `N` octeți. Prin inițializare se înțelege alocarea dinamică a memoriei necesare stocării arenei, setarea fiecărui octet pe 0, și inițializarea lanțului de blocuri (setarea indicelui de start pe 0).
- Comanda nu va afișa nici un rezultat.

### 2. **FINALIZE**

- Această comandă este apelată ultima, și va trebui să elibereze memoria alocată la inițializare și să încheie programul.
- Comanda nu va afișa nici un rezultat.

### 3. **DUMP**

- Această comandă va afișa întreaga hartă a memoriei, așa cum se găsește în acel moment, octet cu octet. Vor fi afișați câte 16 octeți pe fiecare linie, în felul următor:
  - La începutul liniei va fi afișat indicele curent, în format hexazecimal, cu 8 cifre hexa majuscule.

- Apoi este afișat un TAB ( '\t' ) , urmat de 16 octeți, afișati separați printr-un spațiu și în format hexazecimal, cu 2 cifre hexa majuscule fiecare. Între cel de-al 8-lea și cel de-al 9-lea octet se va afișa un spațiu suplimentar.
- Pe ultima linie, indiferent de numărul de octeți din arenă, se va afișa indexul ultimului octet din arenă + 1 (practic, dimensiunea arenei), în format hexazecimal cu 8 cifre hexa majuscule.

#### 4. **ALLOC** <SIZE>

- Comanda va alocă SIZE octeți de memorie din arenă. Ea va trebui să găsească o zonă liberă suficient de mare (care să încapă SIZE octeți + secțiunea de gestiune), și să rezerve un bloc 'la începutul' zonei (nu în mijloc, nu la sfârșit). Va trebui folosită prima zonă liberă validă, într-o căutare de la stânga la dreapta.
- Comanda va afișa, în format zecimal, indexul de început al blocului alocat în arenă, sau 0 dacă nu a fost găsită nici o zonă liberă suficient de mare în arenă.  
'Atenție:' Va trebui să afișați indexul secțiunii de date din noul bloc, și nu al secțiunii de gestiune.

#### 5. **FREE** <INDEX>

- Comanda va elibera blocul de memorie al cărei secțiuni de date începe la poziția INDEX în arenă. Practic, INDEX va fi o valoare care a fost întoarsă în prealabil de o comandă 'ALLOC'. În urma execuției acestei comenzi, spațiul de arenă ocupat de vechiul bloc va redeveni disponibil pentru alocări ulterioare.
- Comanda nu va afișa nici un rezultat.

#### 6. **FILL** <INDEX> <SIZE> <VALUE>

- Comanda va seta SIZE octeți consecutivi din arenă, începând cu indexul INDEX, la valoarea VALUE, cuprinsă între 0 și 255 inclusiv. Atenție, această comandă poate modifica și octeți de gestiune, nu numai octeți de date. În acest caz, se garantează ca arena nu va deveni coruptă după o serie de comenzi FILL consecutive.
- Comanda nu va afișa nici un rezultat.

#### 7. **SHOW** <INFO>

- Comanda va afișa informații statistice despre starea arenei. INFO poate fi una din următoarele:

##### a. **FREE**

- Vor fi afișați (în format zecimal) numărul de octeți liberi din arenă, împreună cu numărul de regiuni (zone continue) libere din arenă sub forma următoare:

```
<nblocks> blocks (<nbytes> bytes) free
```

##### b. **USAGE**

- Vor fi afișați, pe câte o linie:
  - Numărul de octeți folosiți din arenă (numai secțiunile de date)
  - Eficiența utilizării (în procente), egală cu numărul de octeți folosiți raportat la numărul de octeți rezervați (care nu sunt liberi)

- Fragmentarea (în procente), egală cu numărul de zone libere - 1, raportat la numărul de blocuri alocate. Pentru o arenă fără nici un bloc alocat, fragmentarea va fi considerată 0.
- Formatul afișării este:

```
<nblocks> blocks (<nbytes> bytes) used
<eff>% efficiency
<fragm>% fragmentation
```

### c. ALLOCATIONS

- Vor fi afișate pe câte o linie, zonele libere și alocate, în ordinea în care sunt așezate în arenă. Fiecare linie va fi de forma:

```
{FREE|OCCUPIED} <N>
```

- unde { .. | .. } reprezintă faptul că doar una dintre valori va fi afișată. N reprezintă dimensiunea (nenulă), în octeți, a zonei respective.

Nu este nevoie să vă preocupați de eventualele comenzi invalide. Veți presupune că toate comenzile introduse vor fi corecte.

Nu trebuie să verificați semantica operațiilor cerute programului vostru. De exemplu va trebui să executați întocmai comenzi care cer scrierea în zone de memorie nealocate sau rezervate gestiunii, exact așa cum în C puteți realiza operații invalide cu pointeri și programul să vă dea Segmentation Fault.

## Exemple

În această secțiune sunt ilustrate câteva exemple de rulare a programului, pentru a înțelege mai bine modul în care programul vostru trebuie să se comporte. Fiecare exemplu este urmat apoi de câteva explicații.

### Exemplul 1

```
INITIALIZE 100
ALLOC 13
16
FILL 16 13 255
DUMP
00000000 04 00 00 00 00 00 00 00 00 00 00 00 19 00 00 00
00000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00
00000064
FREE 16
ALLOC 50
16
ALLOC 40
0
ALLOC 30
0
ALLOC 20
```

```

78
FILL 78 20 127
DUMP
00000000 04 00 00 00 42 00 00 00 00 00 00 3E 00 00 00
00000010 FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 04 00 00 00 20 00 00 00 7F 7F
00000050 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F
00000060 7F 7F 00 00
00000064
FREE 16
FREE 78
FINALIZE

```

Observații:

- Comenzile (intrarea) este marcată cu albastru, iar ieșirea cu negru. Programul vostru nu trebuie să afișeze nimic altceva în afara a ce este ilustrat în acest exemplu.
- În output-ul de DUMP, au fost marcate cu verde zonele de gestiune. În primul dump, în primul chenar se poate recunoaște indexul de start, în al doilea chenar secțiunea de gestiune a blocului alocat. Cu roșu au fost figurate datele utilizatorilor.
- Se poate observa că cererile de alocare prea mari au fost respinse, afișându-se 0 (care este un index de date invalid, pentru că pe poziția 0 stă indexul de start, și nu se poate alocă memorie în acea zonă).

## Exemplul 2

---

```

INITIALIZE 100
ALLOC 10
16
ALLOC 10
38
ALLOC 10
60
ALLOC 10
82
ALLOC 10
0
FREE 16
FREE 60
FILL 38 10 255
FILL 82 10 255
DUMP
00000000 1A 00 00 00 1A 00 00 00 00 00 16 00 00 00
00000010 00 00 00 00 00 00 00 00 46 00 00 00 00 00
00000020 00 00 16 00 00 00 FF FF FF FF FF FF FF FF FF
00000030 46 00 00 00 1A 00 00 00 16 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 1A 00 00 00 16 00
00000050 00 00 FF FF FF FF FF FF FF FF FF 00 00 00 00
00000060 00 00 00 00
00000064
FINALIZE

```

În acest exemplu se poate observa modul în care sunt actualizate secțiunile de gestiune ale vecinilor blocurilor eliberate, pentru a le înlătura din lanțul de blocuri. De asemenea, se poate observa în dump faptul că un bloc eliberat nu este modificat (resetat pe 0, de exemplu).

## Exemplul 3

---

```

INITIALIZE 100
ALLOC 20
16
ALLOC 20

```



```

48
SHOW FREE
1 blocks (32 bytes) free
SHOW USAGE
2 blocks (40 bytes) used
58% efficiency
0% fragmentation
SHOW ALLOCATIONS
OCCUPIED 4 bytes
OCCUPIED 32 bytes
OCCUPIED 32 bytes
FREE 32 bytes
FREE 16
SHOW FREE
2 blocks (64 bytes) free
SHOW USAGE
1 blocks (20 bytes) used
55% efficiency
100% fragmentation
SHOW ALLOCATIONS
OCCUPIED 4 bytes
FREE 32 bytes
OCCUPIED 32 bytes
FREE 32 bytes
FINALIZE

```

În acest exemplu se poate observa cum se modifică statisticile arenei pe măsură ce blocurile sunt alocate și eliberate.

## BONUS

---

Se va acorda un bonus de **20 de puncte** (pe lângă faimă și respect :) ), dacă se vor implementa, pe lângă comenzile standard prezentate mai devreme, și următoarele comenzi:

### 1. **ALLOCALIGNED <SIZE> <ALIGN>**

- Această comandă va funcționa ca `ALLOC`, cu excepția faptului că indexul returnat va trebui să fie aliniat la `ALIGN` octeți, unde `ALIGN` este o putere a lui 2 (poate fi 1, 2, 4, 8, etc.). Un index `INDEX` este aliniat la `ALIGN` octeți dacă `INDEX % ALIGN == 0`.

### 2. **REALLOC <INDEX> <SIZE>**

- Această comandă va realoca o zonă de memorie întoarsă în prealabil la adresa `INDEX` într-un nou spațiu de memorie de dimensiune `SIZE` și va afișa indexul secțiunii de date a noului bloc alocat. De asemenea, va copia datele aflate în vechiul bloc în noul bloc. Dacă `SIZE` este mai mic decât dimensiunea originală, vor fi copiați numai `SIZE` octeți (va avea loc o `trunchiere`).
- Atenție:** Pentru găsirea unei zone de memorie libere va trebui să reluați procedura de căutare de la stânga la dreapta. Nu este valid să verificați că în locul curent există deja spațiu pentru expansiune / micșorare.

### 3. **SHOW MAP <LENGTH>**

- Comanda va afișa pe mai multe linii un șir de `LENGTH` caractere, fiecare caracter fiind fie `"*"` sau `". "`, care va descrie zonele libere sau ocupate din arenă. Un caracter este `"*"` dacă în zona descrisă de el se află cel puțin un octet rezervat, altfel el va fi `". "`. Fiecare linie va afișa maxim 80 de astfel de caractere. Dacă dimensiunea arenei este `N`, atunci un caracter va reprezenta  $x = N / LENGTH$

octeți. Dacă cel puțin unul din cei x octeți este ocupat, se va afișa "\*", altfel ".".  
Atenție, x poate fi și subunitar.

## Exemplu

---

```
INITIALIZE 100
ALLOCALIGNED 10 32
32
SHOW ALLOCATIONS
OCCUPIED 4 bytes
FREE 16 bytes
OCCUPIED 22 bytes
FREE 58 bytes
FILL 32 10 255
SHOW MAP 50
**      *****
*****
SHOW MAP 31
**      *****
*****
SHOW MAP 2
*
SHOW MAP 200
*****
*****
****
*****
*****
DUMP
00000000 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010 00 00 00 00 00 00 00 00 00 00 00 00 16 00 00
00000020 FF FF FF FF FF FF FF FF FF FF 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00
00000064
REALLOC 32 50
16
DUMP
00000000 04 00 00 00 00 00 00 00 00 00 00 00 3E 00 00
00000010 FF FF FF FF FF FF FF FF FF FF 00 00 16 00 00
00000020 FF FF FF FF FF FF FF FF FF FF 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00
00000064
SHOW MAP 100
*****
*****
*****
FINALIZE
```

## Testare

---

Testarea temei se va face folosind un script de evaluare automata, ce poate fi descărcat la această adresă [[http://ocw.cs.pub.ro/courses/\\_media/programare/teme\\_2016/checker\\_allocator.zip](http://ocw.cs.pub.ro/courses/_media/programare/teme_2016/checker_allocator.zip)].

## Instrucțiuni de utilizare

---

- Arhiva se dezarchivează în directorul vostru de lucru (acolo unde este compilat executabilul ./allocator).
- Arhiva conține un fișier `Makefile.test`, din care se pornește operația de testare. Pentru a rula varianta standard de testare (fără bonus), rulați:
  - `make -f Makefile.test test`
- Pentru a rula varianta de testare cu bonus, rulați:

- `make -f Makefile.test test-bonus`
- Pentru a curăța toate fișierele de ieșire generate pe parcursul testării, rulați:
  - `make -f Makefile.test clean`
- Dacă apar erori și testele eșuează, puteți să vă uitați în directorul `_tests` și să comparați rezultatele voastre (fișierele `.out`), cu cele ale implementării de referință (fișierele `.out.ref`).
- Arhiva include și un binar al implementării de referință (`./reference`), care este folosit intern de tester. Puteți de asemenea să-l rulați separat și să experimentați cu el și alte situații.

## Barem corectare

---

- Criteriile de notare sunt următoarele:
  - **70 puncte** - testele automate din arhiva de testare.
  - **20 puncte** - calitatea și eficiența implementării, utilizarea corespunzătoare a pointerilor/memoriei.
  - **10 puncte** - explicațiile din README și aspectul codului sursă.
  - **20 puncte** - implementarea bonusului.
- Teste **basic**: 30 puncte
- Teste **advanced**: 25 puncte
- Teste **random**: 15 puncte
- Implementare **bonus**: 20 puncte

### Mențiuni suplimentare:

- dacă pentru o anumită categorie de teste nu au trecut toate testele, am punctat astfel:
  - teste **basic**: 10 puncte fiecare test trecut
  - teste **advanced**: 5 puncte fiecare test trecut
  - teste **random**: 5 puncte fiecare test trecut
  - teste **bonus**: 5 puncte fiecare test trecut
- cele 10 puncte pentru README și aspectul codului sursă s-au împărțit astfel:
  - existența unui README relevant: 2 puncte
  - claritatea codului sursă: 8 puncte
- cele 20 puncte pentru calitatea și eficiența implementării, utilizarea corespunzătoare a pointerilor/memoriei s-au împărțit astfel:
  - calitatea și eficiența implementării: 10 puncte
    - am scăzut 5 puncte dacă codul nu este modularizat
      - am scăzut 2 puncte dacă codul este modularizat, dar folosește funcții foarte lungi
    - am scăzut 5 puncte pentru warning-uri de compilare
  - utilizarea corespunzătoare a pointerilor/memoriei: 10 puncte
    - am scăzut 5 puncte dacă memoria nu a fost alocată dinamic
    - am scăzut 2 puncte dacă memoria a fost alocată dinamic, dar nu s-a eliberat la final memoria

- am scăzut 5 puncte dacă programul are accese nevalide la memorie (testat cu valgrind)

## Regulament teme

---

- Regulamentul general al temelor se găsește aici  
[[http://ocw.cs.pub.ro/courses/programare/regulament-ca#temele\\_de\\_casa](http://ocw.cs.pub.ro/courses/programare/regulament-ca#temele_de_casa)].
- Tema se va implementa DOAR în limbajul C. Va fi compilată și testată DOAR într-un mediu LINUX. Nerespectarea acestor reguli aduce un punctaj nul pe temă.
- Tema va fi trimisă atât pe vmchecker [<https://elf.cs.pub.ro/vmchecker/>] cât și pe moodle [<http://cs.curs.pub.ro>], sub forma unei arhive ZIP. Nerespectarea acestei reguli aduce un punctaj NUL pe temă.
- Fișierele temei trebuie OBLIGATORIU împachetate într-o arhivă de tip '.zip', cu numele 'Grupa\_NumePrenume\_Tema3.zip'.
- Arhiva va trebui să conțină în directorul RADACINA doar următoarele:
  - 1. Codul sursă al programului vostru (fișierele .c și eventual .h).
  - 2. Un fișier Makefile care să conțină regulile build și clean. Regula build va compila programul într-un executabil cu numele **allocator**. Regula clean va șterge executabilul și eventual toate binarele intermediare (fișiere obiect) generate de voi.
  - 3. Un fișier README care să conțină prezentarea implementării alese de voi. Dacă ați implementat și bonusul, menționați acest lucru în README. NU copiați bucăți din enunț în README.
  - 4. Un fișier gol bonus dacă ați implementat și bonus-ul (folosit intern de vmchecker pentru a determina dacă trebuie să ruleze sau nu și aceste teste).
  - Nerespectarea regulilor 1 și 2 aduce un punctaj nul pe temă.
- Arhiva temei NU va conține fișiere binare.
- O temă care NU compilează pe vmchecker [<https://elf.cs.pub.ro/vmchecker/>] NU va fi punctată.
- O temă care compilează, dar care NU trece niciun test pe vmchecker [<https://elf.cs.pub.ro/vmchecker/>], NU va fi punctată. Punctele pe teste sunt cele de pe vmchecker [<https://elf.cs.pub.ro/vmchecker/>].