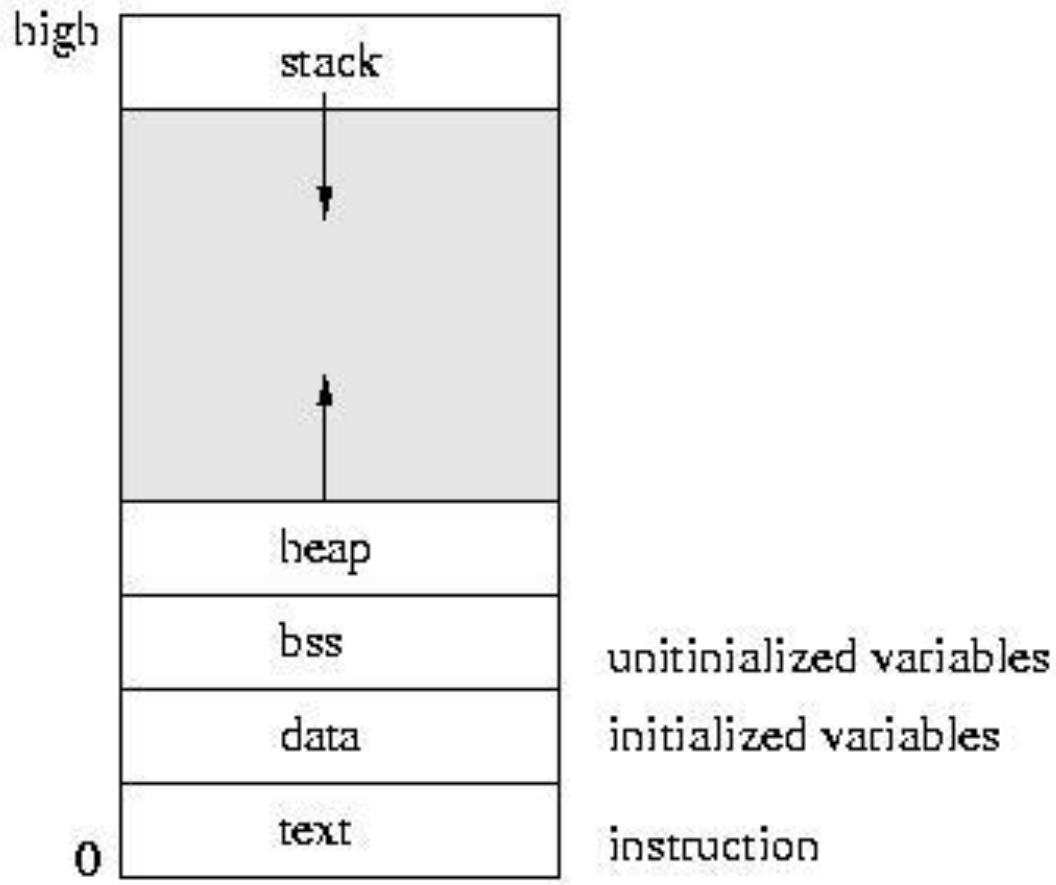
Alocarea dinamică a memoriei

Reminder Stivă (stack) vs Heap



Reminder Stivă (stack) vs Heap

Stivă

- acces ff rapid
- se curăță automat la terminarea funcției
- spațiul gestionat eficient (nu se fragmentează)
- doar pentru variabile locale
- dimensiune (mai mică decât heap-ul) dependentă de sistemul de operare [1]
- dimensiunea variabilelor nu poate fi modificată

Heap

- variabilele pot fi accesate global
- dimensiune nelimitată*
- ceva mai lentă
- trebuie să gestionăm noi memoria (rezervăm/alocăm și eliberăm)
- nu este gestionată eficient (se fragmentează)
- putem redimensiona variabilele

*depinde totuși de RAM/swap

Stivă (stack) vs Heap

Când folosim stiva?

- date de dimensiuni mici
- le folosim doar într-o funcție
- le știm de la început dimensiunea maximă (și aceasta nu se schimbă pe parcursul programului)

Când folosim heap-ul?

- date de dimensiuni mari
- date care pot să-și schimbe dimensiunea în timp
- date care trebuie să aibă durată de viață mare

null pointer

- Cum știm că un pointer este valid (îl putem folosi)?
  - un pointer este o adresă de memorie, dacă scriem la o adresă pe care nu avem dreptul să o accesăm => segmentation fault
- Cum diferențiem un pointer inițializat de un pointer neinițializat?
  - amândoi pointerii punctează la o adresă care poate fi validă

null pointer (NULL)

- avem nevoie de o adresă specială, care să nu fie validă niciodată și care să o atribuim pointerilor ce nu pot fi folosiți
- The language definition states that for each pointer type, there is a special value--the ”null pointer”--which is **distinguishable from all other pointer values** and which is ”**guaranteed to compare unequal to a pointer to any object or function**.” That is, a null pointer **points definitively nowhere**; it is not the address of any object or function. **The address-of operator & will never yield a null pointer [3]**
- in C null pointer-ul este reprezentat ca ***NULL și are valoarea 0***
- *în teorie pot exista și reprezentări diferite de 0 în memorie pt NULL*

null pointer (NULL)

- int *p;
- p=NULL; //p este inițializat cu NULL
- if(p) //sau if(p!=NULL) – verifică dacă p are o adresă validă
- trebuie să avem grijă ca atunci când un pointer nu mai are o adresă validă spre care să puncteze să-i fie asignat NULL
- funcții care nu reușesc să returneze adrese valide folosesc returnează tot NULL

Alocarea dinamică a memoriei

- alocarea dinamică a memoriei – rezervarea unui spațiu de memorie a cărui dimensiune o putem ști la compilare sau la rulare.
- pentru a putea utiliza spațiul rezervat adresa de început a zonei de memorie este asignată unui pointer
- în cazul în care alocarea nu reușește funcțiile de alocare întorc NULL

malloc

- void* malloc (size_t size);
- alocă un număr de *size* octeti
- întoarce pointer la zona de date alocată sau NULL în cazul în care alocarea nu reușește
- pointerul se recomandă a fi convertit la tipul de date pe care vrem să-l alocăm
- definită în *stdlib.h*
- *size_t – unsigned int [4]*

Exemplu malloc

```c
#include <stdio.h>

int main(void)
{
    int i,*p,n;
    printf("introdu n (>0):\n");
    scanf("%d",&n);

    if(n<=0)
        return 0;

    p=malloc(n*sizeof(int));
    if(!p)
    {
        printf("nu s-a putut efectua alocarea\n");
        return 0;
    }
    for (i=0;i<n;i++)
        printf("p[%i]=%i, adresa lui p[%i]=%p\n",i,p[i],i,p+i);

    return 0;
}
~
```

calloc

- void* calloc(size_t num, size_t size);
- num = numărul de elemente
- size = dimensiunea unui element
- toată zona de memorie alocată este inițializată cu 0
- ! ținând cont că NULL poate avea reprezentări pe biți diferite de 0 dacă alocăm un vector de pointeri nu este bine să ne bazăm pe inițializările făcute de calloc

Exemplu calloc

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i,*p,n;
    printf("introdu n (>0):\n");
    scanf("%d",&n);

    if(n<=0)
        return 0;

    p=calloc(n,sizeof(int));
    if(!p)
    {
        printf("nu s-a putut efectua alocarea\n");
        return 0;
    }
    for (i=0;i<n;i++)
        printf("p[%i]=%i, adresa lui p[%i]=%p\n",i,p[i],i,p+i);

    return 0;
}
```

realloc

- void *realloc (void* ptr, size_t size);
- redimensionează zona de memorie spre care punctează ***ptr*** la ***size*** octeți.
- dacă ***size*** este mai mare decât dimensiunea inițială a blocului, zona suplimentară nu este inițializată
- dacă realocarea nu reușește, întoarce NULL

realloc

- In cele mai multe situatii, puteti considera ca

```
ptr1=realloc(ptr2, ...);
```

- Este echivalent cu:

```
ptr1 = malloc(...);
memcpy(ptr1, ptr2, ...);
free(ptr2);
```

Exemplu realloc

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i,*p,*q,n;
    printf("introdu n (>0):\n");
    scanf("%d",&n);

    if(n<=0)
        return 0;

    p=calloc(n,sizeof(int));
    if(!p)
    {
        printf("nu s-a putut efectua alocarea\n");
        return 0;
    }
    for (i=0;i<n;i++)
        printf("p[%i]=%i, adresa lui p[%i]=%p\n",i,p[i],i,p+i);

    n*=2;
    q=realloc(p,n*sizeof(int));
    if(q)
        p=q;//!!! nu folosim p=realloc(p,n*sizeof(int))
    //daca realocarea nu ar reusi s-ar pierde zona de memorie alocata initial
    return 0;
}
~
```

free

- void free(void* ptr);
- eliberează memoria alocată și spre care punctează pointerul ***ptr***
- dacă ***ptr*** punctează către o zonă de memorie care nu a fost alocată cu malloc, calloc, realloc comportarea este nedefinită
- ***ptr*** nu este modificat de free. Este **recomandat** să îi asignăm NULL după apelul free

Exemplu free

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i,*p,*q,n;
    printf("introdu n (>0):\n");
    scanf("%d",&n);

    if(n<=0)
        return 0;

    p=calloc(n,sizeof(int));
    if(!p)
    {
        printf("nu s-a putut efectua alocarea\n");
        return 0;
    }
    for (i=0;i<n;i++)
        printf("p[%i]=%i, adresa lui p[%i]=%p\n",i,p[i],i,p+i);

    n*=2;
    q=realloc(p,n*sizeof(int));
    if(q)
        p=q;//!!! nu folosim p=realloc(p,n*sizeof(int))
    //daca realocarea nu ar reusi s-ar pierde zona de memorie alocata initial

    free(p);
    return 0;
}
```

alocare spatiu vectori

- tip_vector *v;
- v=(tip_vector*)malloc(nr_elemente*sizeof(tip_vector));
- sau
- v=(tip_vector*)calloc(nr_elemente,sizeof(tip_vector));

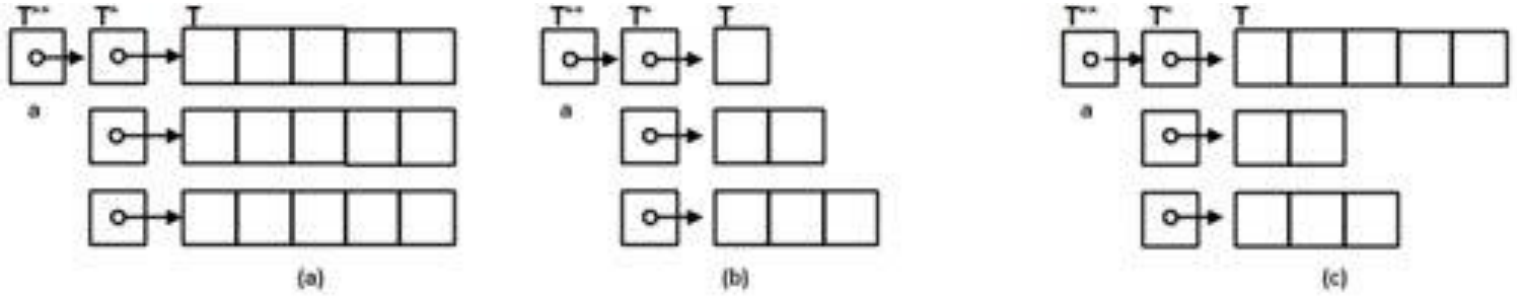
alocare spatiu matrice

- vrem sa alocam spatiu pentru o matrice [m][n]
- putem aloca un vector cu m*n elemente
  - trebuie sa avem grija sa accesam elementele corect (v[n*i+j]⇔v[i][j]
- alocam un vector de m pointeri si pentru fiecare pointer alocam un vector de n elemente

alocare matrice ca vector

alocare matrice ca vector de pointeri

- From [5]



Representing two-dimensional arrays using a pointer to a pointer: (a) Regular matrix (b) Lower triangular matrix (c) Ragged matrix

Greșeli frecvente alocare

- nu se verifică succesul alocării
  - v=(int*)malloc(n*sizeof(int));
  - if(!v) //if(v==NULL) – nu s-a putut efectua alocarea se tratează eroarea
- nu se eliberează memoria
  - aveti grija sa faceti free de fiecare dată când folosiți alocare dinamică

Greseli frecvente

- erori de logică
  - se utilizează o zonă de memorie după ce a fost eliberată
    - nu se asigneaza pointerului NULL dupa eliberare si se acceseaza o zona de memorie care poate fi curatata sau poate sa mentina valorile (cel mai periculos)
    - nu se asigneaza pointerului NULL dupa eliberare si ajungem sa accesam o zona la care nu mai avem dreptul (seg fault)
    - asignam pointerului NULL dupa eliberare sau dupa eroare alocare dar nu verificam daca pointerul este diferit de NULL si incercam sa-l mai folosim

VLA vs vectori alocati dinamic

- VLA

```
int* f(int n)
{
        int v[n];
        …..
        return v;
}
```

v puncteaza catre o zona de memorie “curatata” la parasirea functiei

spatiul pentru v este rezervat pe stiva

spatiul pt v este limitat la dimensiunea stivei

- vectori alocati dinamic

```
int *f(int n)
{
        int *v=(int*)
malloc(n*sizeof(int));
        …
        return v;
}
```

spatiul rezervat pentru v exista si la incheierea functiei

v este alocat pe heap

Bibliografie

1. http://www.cs.nyu.edu/exact/core/doc/stackOverflow.txt
2. http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html
3. http://c-faq.com/null/index.html
4. http://en.wikipedia.org/wiki/C_data_types#stddef.h
5. http://ecomputernotes.com/what-is-c/function-a-pointer/two-dimensional-arrays-using-a-pointer-to-pointer