

Le processus de responsabilité

de Christopher Avery

- Face à une situation problématique, le cerveau passe systématiquement par un certain nombre d'étapes : c'est le processus de responsabilité.
- Quelques exemples de situations :
 - j'ai perdu mes clés
 - mon binôme me dit que mon code ne fonctionne pas
 - j'ai planté mes exams
 - je suis largué en prog.
 - ...

Première étape

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)

Deuxième étape

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)
- L'accusation :
 - Il y a un problème, mais vous rejetez la faute sur un tier (personne, machine...).

Troisième étape

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)
- L'accusation :
 - Il y a un problème, mais vous rejetez la faute sur un tier (personne, machine...).
- La justification :
 - Il y a un problème, vous en acceptez une par de responsabilité mais vous avez des excuses...

Quatrième étape

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)
- L'accusation :
 - Il y a un problème, mais vous rejetez la faute sur un tier (personne, machine...).
- La justification :
 - Il y a un problème, vous en acceptez une par de responsabilité mais vous avez des excuses...
- La culpabilité :
 - Il y a un problème, vous en acceptez une par de responsabilité et vous vous en voulez...

Cinquième étape

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)
- L'accusation :
 - Il y a un problème, mais vous rejetez la faute sur un tier (personne, machine...).
- La justification :
 - Il y a un problème, vous en acceptez une par de responsabilité mais vous avez des excuses...
- La culpabilité :
 - Il y a un problème, vous en acceptez une par de responsabilité et vous vous en voulez...
- L'obligation :
 - Vous faites ce qu'il y a à faire pour résoudre le problème présent

Sixième étape

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)
- L'accusation :
 - Il y a un problème, mais vous rejetez la faute sur un tier (personne, machine...).
- La justification :
 - Il y a un problème, vous en acceptez une par de responsabilité mais vous avez des excuses...
- La culpabilité :
 - Il y a un problème, vous en acceptez une par de responsabilité et vous vous en voulez...
- L'obligation :
 - Vous faites ce qu'il y a à faire pour résoudre le problème présent
- La responsabilité :
 - Vous analysez les causes du problème de façon à corriger celui-ci durablement.

Processus de responsabilité

FUITE

- le déni :
 - vous niez qu'il y a un problème. (Cela peut être le cas de votre point de vu)
- L'accusation :
 - Il y a un problème, mais vous rejetez la faute sur un tier (personne, machine...).
- La justification :
 - Il y a un problème, vous en acceptez une par de responsabilité mais vous avez des excuses...
- La culpabilité :
 - Il y a un problème, vous en acceptez une par de responsabilité et vous vous en voulez...

RESOLUTION

- L'obligation :
 - Vous faites ce qu'il y a à faire pour résoudre le problème présent
- La responsabilité :
 - Vous analysez les causes du problème de façon à corriger celui-ci durablement.

Processus de responsabilité

- C'est un outil personnel
- En aucun cas il faut l'utiliser directement :
 - « t'es bloqué en accusation là... »
- Identifier le positionnement de votre interlocuteur peut vous permettre de mieux diriger l'échange pour arriver à une résolution du problème :
 - montrer qu'il y a un problème (→ déni)
 - identifier les « responsables » (déni → accusation)
 - Aurais-tu pu faire quelque chose ? (accusation → justification)
 - Tu aurais pu faire ceci ou cela... (justif → culpabilité)
 - Corrigeons le problème (→ obligation)
 - Analysons les racines de ce problème et trouvons une solution sur le long terme. (→ responsabilité)

Les bibliothèques statiques

- Les bibliothèques statiques sont un regroupement d'objets
- Elles peuvent être créées avec l'outil « ar »



```
ar rcs ma_bibilo.a b.o c.o d.o

nm ma_bibilo.a
b.o:
000000000000000000 T b

c.o:
000000000000000000 T c

d.o:
                                U c
000000000000000000 T d
```

- Lors de l'utilisation, si un objet d'une bibliothèque apporte un symbole manquant, alors il est intégralement inclus (l'objet par la bibliothèque).

```
nm a.out | grep -v _
000000000040051b T c
0000000000601038 b completed.7259
000000000040050b T d
00000000004004f6 T main
```

Les bibliothèques dynamiques

- Une bibliothèque dynamique est binaire qui regroupe un ensemble de symboles.
- Lors de l'édition de liens, si un symbole manquant est fourni par un biblio. dyn. alors un lien est créé vers cette bibliothèque :

```
gcc -fPIC -shared -o libtoto.so b.c c.c d.c
gcc -c a.c

gcc a.o -ltoto -L.
nm a.out | grep -v _
0000000000601040 b completed.7259
                U d
0000000000400696 T main

LD_LIBRARY_PATH=. ldd a.out
    linux-vdso.so.1 => (0x00007fff8e522000)
    libtoto.so => ./libtoto.so (0x00007f63a7ea2000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f63a7ac4000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f63a80a6000)
```



Question

#QDLE#Q#ABC*D#60#

- Laquelle de ces affirmations est fausse :
 - Les bibliothèques dynamiques permettent la correction de bug sans toucher aux exécutables qui les utilisent.
 - L'utilisation des bibliothèques statiques produit des exécutables plus volumineux
 - L'utilisation des bibliothèques dynamiques est plus « sécurisé »
 - La suppression d'une bibliothèque dynamique sur mon système va empêcher l'exécution de programme qui l'utilisent.

L'importance des header !

- Les header servent de « contrats » entre différentes entités compilées séparément (module, bibliothèques).
- Les header ne contiennent pas de code !
- Un header est un fichier .h qui contient :
 - la déclaration de fonction
 - la déclaration de structures
 - les macro
 - les « globales »

Les header

- La déclaration de fonction :
 - ex : `int fonction_foo(char) ;`
 - Le nommage des paramètres n'est pas obligatoire.
- Les structures :
 - Selon que l'on souhaite donner accès au champs de la structure, on pourra soit pré-déclarer soit déclarer :

point.h

```
struct Point ;
```

point.c

```
struct Point{  
    int x ;  
    int y ;  
}
```

point.h

```
struct Point{  
    int x ;  
    int y ;  
}
```

seul point.c peut créer des struct Point et accéder aux champs. Les autres doivent obligatoirement utiliser des pointeurs.

tout le monde peut allouer des structures et accéder au champs

Les header

- L'avantage de la pré-déclaration est de masquer l'implémentation et de pouvoir modifier l'implémentation sans impacter les utilisateurs (on en reparle plus tard avec les API).
- Les macros (déjà vu). Pour éviter les inclusions multiples, on utilisera la technique :

```
#ifndef ID
```

```
#define ID
```

```
....
```

```
#endif
```

Les header

- Les globales, c'est à dire les variables dont on souhaite que tout le monde puisse y accéder.
- On évitera de telle variable (effet de bord...).
- Les variables globales doivent être déclarées dans le « .c » car un symbole est généré pour cette variable.
- Afin d'indiquer que ce symbole existera, on le déclare en « extern » dans le .h :

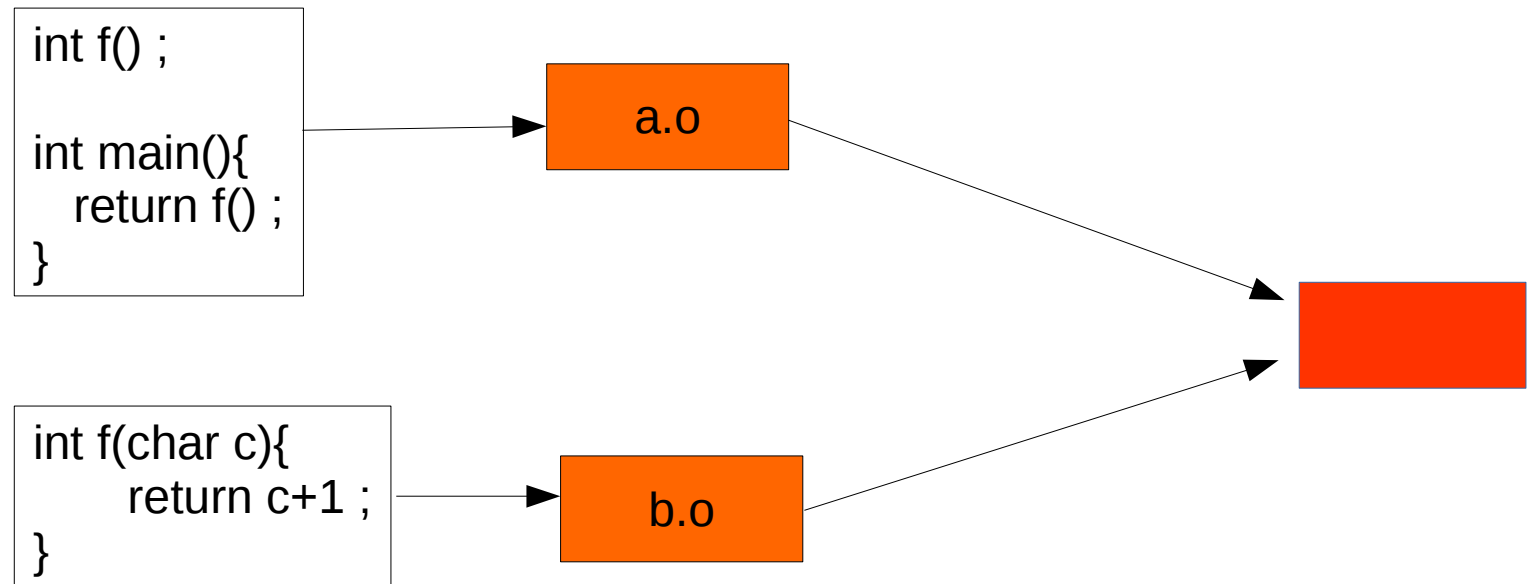
```
extern int v ;           int v ;
```

- Ainsi l'utilisateur du .h aura le symbole « v » comme manquant dans le .o. Lors de l'édition de lien, il y aura association entre le symbole v manquant et le symbole v fournit.

Les header

- Après l'étape de pré-compilation, tout les header inclus (directement ou indirectement) sont regroupé en un seul source.
- Si un macro est définie plusieurs fois, il y a erreur dans la pré-compilation.
- Si une structure, globale ou fonction est définie plusieurs fois, il y a erreur lors de la compilation, d'où l'importance de se protéger contre les inclusions multiples.

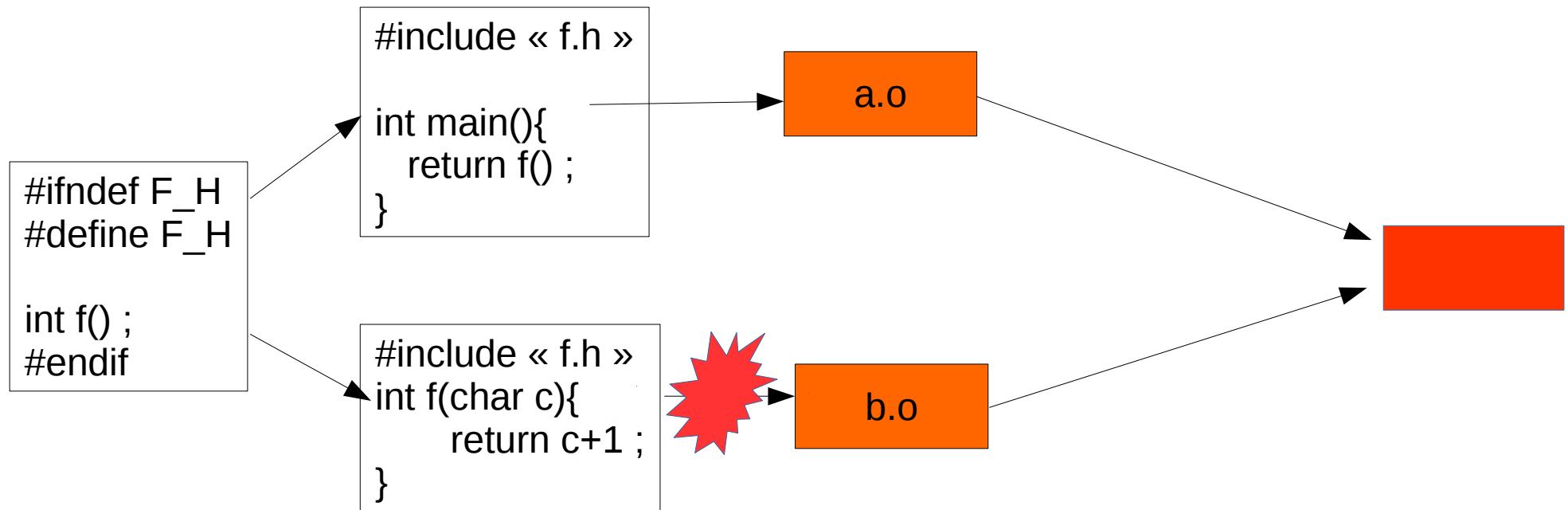
Les header par l'exemple



Les deux sources compilent sans problème.

Il n'y a pas de header, la cohérence n'est pas garantie, le comportement final n'est pas prédictible.

Les header par l'exemple



La code de la fonction f dans le fichier b.c n'est pas cohérent avec la déclaration dans f.h => erreur à la compilation.

Les header

#QDLE#Q#ABC*#30#

- Lorsqu'un fichier header est modifié, je dois :
 - A. recompiler l'ensemble de mon projet
 - B. recompiler les fichiers .c qui incluent ce header
 - C. recompiler les fichier .c qui incluent ce header directement ou indirectement.

Question

#QDLE#Q#A*BCD#30#

- Laquelle de ces affirmations est fausse :
 - A. Le langage C est un langage impératif
 - B. Le langage C est un langage compilé
 - C. Le langage C est un langage fonctionnel
 - D. Le langage C n'est pas un langage interprété



Exécution et chargement

#QDLE#Q#AB*#20#

- Lors du chargement d'un exécutable toutes les bibliothèques dont il dépend sont d'abord chargés.
- Si un bibliothèque est manquante :

```
./a.out: error while loading shared libraries: libtoto.so: cannot open shared object file: No such file or directory
```

- Si un symbole est manquant :

```
./a.out: symbol lookup error: ./a.out: undefined symbol: d
```

- Cette erreur peut provenir d'une biblio. statique ?
 - oui
 - non

Programme, processus et mémoire...

- Lors de l'exécution du binaire, un espace mémoire particulier est associé à cette exécution : c'est le processus
- La mémoire de ce processus est divisée en pages de taille fixe organisées en zones :
 - la pile
 - le tas
 - le code du binaire
 - le code des bibliothèques dynamiques liées
 - les variables globales (initialisées et non initialisées)

Programme, processus et mémoire...

- Chaque processus a son propre espace d'adressage.
- Le système maintient une table d'association entre les pages du processus et leurs localisations effectives en mémoire (RAM ou swap par ex.).
- Les valeurs d'adresses possibles sur une machine 32 bits vont de 0x00000000 à 0xFFFFFFFF

Programme, processus et mémoire...

- La taille des pages est fixe :

```
> getconf PAGESIZE  
4096
```

- Aussi, les adresses 0x00000000 à 0x00000FFF appartiennent à la même page.
- Sur une machine 32 bits, l'espace d'adressage est donc divisé en $2^{(32-12)}=1048576$ pages.
- Ainsi, chaque page peut être :
 - non adressable (non associée)
 - adressable :
 - en lecture
 - en écriture
 - en exécution

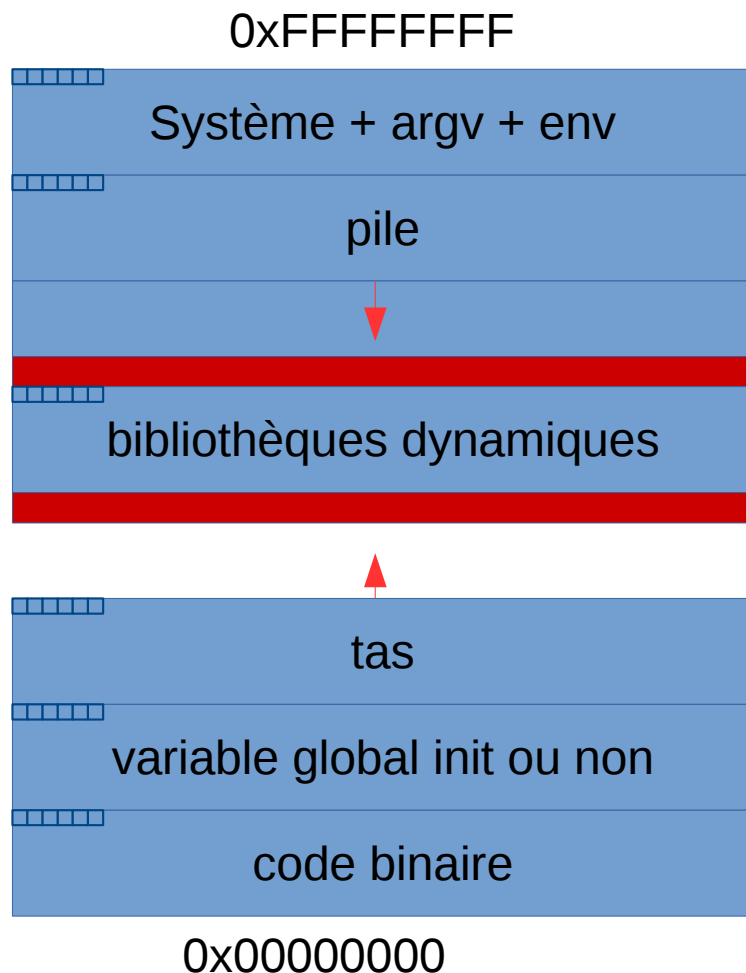
Question

#QDLE#Q#A*B#25#

- Si, sur un système A la taille des pages est de 4096 et que sur le système B, la taille des pages est de 8192, alors
 - A. Les tables d'indirection des processus sur A sont plus grandes que sur B ?
 - B. Les tables d'indirection des processus sur B sont plus grandes que sur A ?

Programme, processus et mémoire...

- Un accès incompatible mène à l'émission d'un signal qui souvent se traduit par un `SEGV`



- Au lancement du processus les pages mémoires pour la pile sont déjà réservées.
- Les pages mémoires pour le tas ne le sont pas. Le programme peut demander à augmenter le tas lors de l'exécution (`brk/sbrk`).

Programme, processus et mémoire...

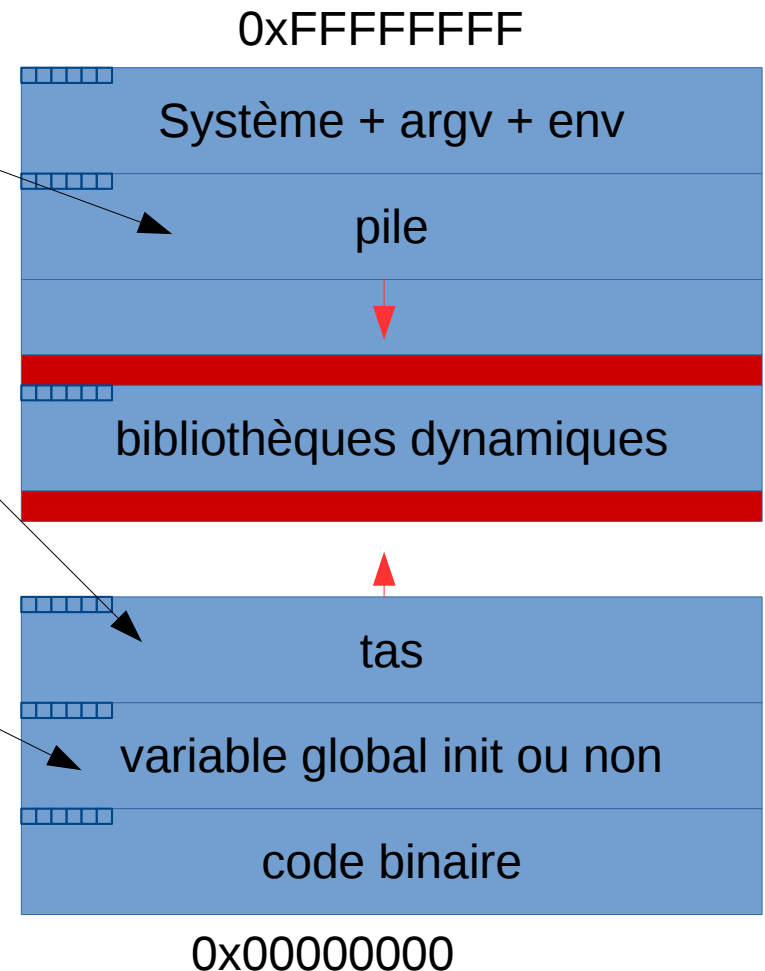
#QDLE#Q#AB*CD#25#



- La mémoire où est chargée le code binaire à les droits :
 - A. en lecture /écriture
 - B. en lecture/exécution
 - C. en écriture/exécution
 - D. en execution seul.

Programme, processus et mémoire...

- Il existe principalement 3 modes d'allocation mémoires :
 - L'allocation automatique
 - L'allocation dynamique
 - L'allocation statique
- Ces 3 modes correspondent à trois types de gestions différents.



Les variables

- Une variable dans le code source est un outil pour le programmeur qui permet de manipuler une zone mémoire et d'y associer une interprétation :

```
int i ;           float k ;           void *data ;  
char c ;         double l ;         char *toto ;
```

`sizeof(type)` ⇒ la quantité de mémoire nécessaire pour stocker le type

`sizeof(var)` ⇒ idem pour le type de « var »

Les variables

- A l'exécution, le **nom** des variables et des fonctions n'existent pas : il n'y pas que des adresses.
- Ainsi, lors de l'utilisation d'une variable v :
 - v correspond à l'interprétation de la mémoire
 - $\&v$ correspond à l'adresse de cette mémoire
- Cas de pointeur :
 - Contient une adresse, ainsi sa taille est toujours la même quel que soit le type de la donnée pointée :
`sizeof(void *)=sizeof(char *)=sizeof(int *)....`
 - Comme toute variable, un pointeur a une adresse propre.

3 modes de réservation mémoire

- Les variables permettent d'associer une interprétation à de la mémoire.
- Il y a trois façons de réserver de la mémoire :
 - Allocation statique : variables globales
 - Allocation automatique : variables de contexte
 - Allocation dynamique : réservation et libération explicite par le programmeur.

Les variables statiques

- Les variables statiques sont créées dans le binaire. Lorsque ce binaire est chargé en mémoire alors ces variables le sont également.
- Les adresses sont fixes et restent valides du lancement jusqu'à la fin du processus.

```
> cat a.c
```

```
char msg[] = « ceci est un  
tres tres .....  
tres.....  
tres long message » ;
```

```
> ls -l a.c  
25447 a.c
```

```
> gcc -c a.c  
> ls -l a.o  
26392 a.o  
> hexdump -c a.o  
00000000 177 E L F 002 001 001 \0 \0 \0 \0 \0 \0 \0 \0  
00000010 001 \0 > \0 001 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0  
00000020 \0 \0 \0 \0 \0 \0 \0 \0 <D8> d \0 \0 \0 \0 \0 \0  
00000030 \0 \0 \0 \0 @ \0 \0 \0 \0 \0 @ \0 \t \0 006 \0  
00000040 c e c i e s t u n t r e s  
00000050 t r e s t r e s t r e s  
00000060 l o n g m e s s a g e c e c i  
00000070 e s t u n t r e s t r e  
00000080 s t r e s t r e s l o n g  
...  
> strings a.o  
ceci est un tres tres tres....
```

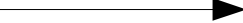
Les variables statiques

```
int i=0;

void f(){
    i+=1;
    printf("f: %p %d\n",&i,i);
}

void g(){
    i+=2;
    printf("g: %p %d\n",&i,i);
}

int main(){
    printf("main: %p %d\n",&i,i);
    f();
    printf("main: %p %d\n",&i,i);
    g();
    printf("main: %p %d\n",&i,i);
    return i;
}
```



```
main: 0x601044 0
f: 0x601044 1
main: 0x601044 1
g: 0x601044 3
main: 0x601044 3
```

Au lancement, `sizeof(int)` octets sont réservés à l'adresse `0x601044` et sont initialisés à la valeur `(int)0`.

Allocation automatique

- L'allocation automatique concerne les variables réservées dans la **pile**.
- Cette allocation est particulière du fait qu'elle réserve automatiquement l'espace nécessaire lors de la création de variables, et libère automatiquement cet espace lorsque la variable est « détruite ».
- Elle correspond à des variables de « contextes »...



Allocation automatique

#QDLE#Q#ABC*#45#

- Exemple : les variables de fonction.

```
void f() {  
    int i ;  
    char j[5] ;  
    double *m ;  
}
```

Dans cet exemple, i, j et m sont des variables dites « locales » (par opposition aux variables globales).

Lors du chargement du programme, avant l'appel à la fonction f, que représente i, j et m ?

- A. des adresses mémoires ?
- B. le contenu de cases mémoires ?
- C. autre.

Allocation automatique

- Règle : un fonction ne doit jamais renvoyer directement ou indirectement l'adresse d'une variable locale.
- En effet, cette adresse devient non réservée dès que l'on « sort » de cette fonction.

```
int *getTab(){  
    int t[10] ;  
    return t ;  
}
```

Ce code génère un Warning à la compilation !

Allocation automatique & pile

- La taille de la pile d'un programme est limitée et fixée lors de l'exécution.
- Sous linux, la commande
\$ ulimit -s
8192
- permet de connaitre et fixer la taille de la pile (en ko).



```
#include<stdio.h>
int c=0;

int main(){
  int i=0;
  int *p=&i;
  printf("%d\n",(int)sizeof(int));
```

```
while(1){
  *p=0;
  p--;
  c+=1;
  printf("%d\n",c);
}
```

```
4
1
2
.....
2094516
Segmentation fault (core dumped)
```

La récursivité

#QDLE#Q#ABC*D#45#

- Soit le code ci-dessous :

```
#include<stdio.h>
int c=0;

void f(void){
    c+=1;
    printf("%d\n",c);
    f();
}

int main(){
    f();
}
```

```
$ ulimit -s
8192
$ ./stack
1
2
....
523783
Segmentation fault (core dumped)
```

- Quel est l'espace minimal pris dans la pile par un appel de fonction ?
 - A. 4 octets
 - B. 8 octets
 - C. 16 octets
 - D. 32 octets

L'allocation dynamique

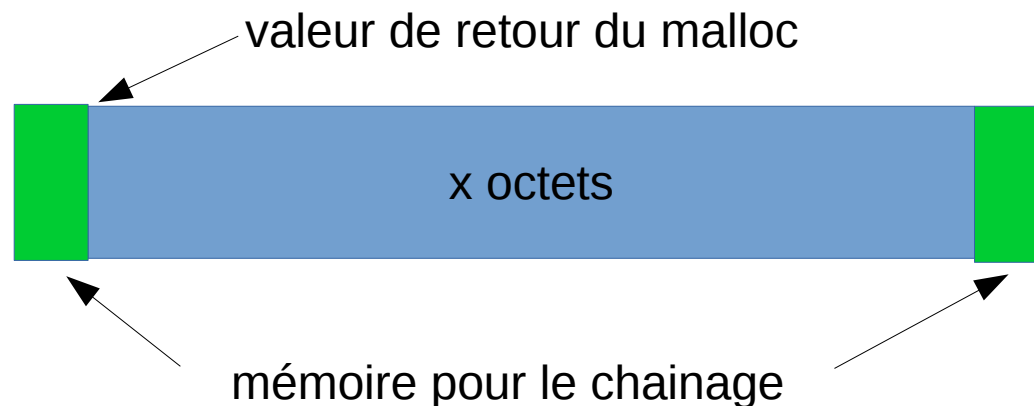
- Au lancement du processus, il y a une plage d'adresse non réservée (aucune page n'y est associée).
- L'appel à la fonction **sbkr** permet d'étendre cette plage d'adresse. La mémoire ainsi réservée peut être utilisée par le processus.
- Les fonctions malloc/free permettent une gestion de cette mémoire.

L'allocation dynamique

- La structuration et la gestion du tas est liée à l'interprétation de cette mémoire par le gestionnaire : ici malloc.
- Il est tout à fait possible de remplacer ce gestionnaire par un autre.
- Tant qu'aucun appel à **sbrk** n'est fait pour rendre au système des pages mémoires, du point de vue du système, les pages correspondent à des zones accessibles (valides).
- malloc et free fonctionnent sur un système de liste chaînée

malloc

- L'appel à `malloc(x)` permet de demander à celui-ci l'adresse d'une zone mémoire de `x` octets.
- `malloc` va réserver cette espace dans son système et renvoyer l'adresse de cette zone.
- La zone va être préfixée et post fixée par des pointeurs correspondant au chainage de `malloc` :



malloc

- démonstration :

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>

int main(){
    int i=0;
    for(i=0;i<1000000000;++i)
        malloc(4);
    sleep(60);
}
```

100M Allocations par malloc de 4 octets

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>

int main(){
    int i=0;
    malloc(4*1000000000);
    sleep(60);
}
```

Allocation par malloc de 400 Moctets

```
$ top -p
PID USER      PR  NI  VIRT  RES  SHR S  %CPU  %MEM    TIME+  COMMAND
23020 allali    20   0 3129296 2,981g 1020 S   0,0  38,8   0:04.24 mem
23139 allali    20   0  394824    648   568 S   0,0   0,0   0:00.00 mem2
```

Allocation dynamique

- Tant que l'on a pas indiqué à malloc qu'une zone mémoire n'est pas réutilisable, il ne va pas sans servir pour de nouvelles allocations.
- Contrairement à l'allocation automatique, la « durée de vie » de l'allocation dynamique n'est pas liée à l'appel ou au retour de fonction.
- Si à un moment il n'existe plus de variable dans la pile (ou static) qui pointe directement ou indirectement vers une allocation dynamique alors il y a une « fuite mémoire » (memory leak).

Allocation dynamique

- Sous linux, le fonctionnement de malloc peut être paramétré via des variables d'environnement ou la fonction **mallopt** :
 - `MALLOC_CHECK_` : vérification au moment de la libération (double free)
 - `MALLOC_PERTURB_` : permet de remplir la mémoire avec une valeur à l'allocation et de la ré-initialiser à une autre valeur à la libération. Très utile !

Allocation : conclusion

- **IMPORTANT** : toute variable est soit dans la pile, soit dans le segment de donnée (static).
- En aucun cas une variable de votre programme peut avoir une adresse qui soit dans le tas.
- L'utilisation de la mémoire dynamique suppose donc l'utilisation de pointeurs permettant de gérer les adresses mémoires.
- La structuration des plages n'est pas connu par malloc : il est important d'interpréter ces zones correctement (taille).

malloc : question

#QDLE#Q#A*B#30#

- L'utilisation de malloc au profit de l'allocation automatique dans des fonctions récursives permet d'élargir la capacité de traitement de celle-ci :
 - A. vrai
 - B. faux



Question

#QDLE#Q#AB*C#40#

```
void rand(int *i) ; //génère un nombre aléatoire
int main(){
    int *j=NULL ;
    rand(j) ;
    return 0 ;
}
```

- Ce code :
 - A – compile et fonctionne sans erreur
 - B – compile mais génère une erreur à l'exécution
 - C – ne compile pas

En bref :

- Allocation statique :
 - Variables globales
 - Chaînes constantes
 - ⇒ dans le binaire, adresse fixe dans la zone data
- Allocation automatique :
 - Variables locales
 - Paramètre de fonction
 - ⇒ réservé dans la pile en début de bloc, libéré en sortie de bloc. Adresses variables
- Allocation dynamique :
 - malloc / free / sbrk
 - ⇒ Mémoire réservée à la demande dans le tas.

Les pointeurs

- Pour toute variable *foo i*, *&i* est de type *foo **
- Les pointeurs sont des variables qui contiennent des adresses mémoire.

*foo *p*

- La variable *p* contient une adresse
- L'adresse contenue dans *p* correspond au début de *sizeof(foo)* octets qui seront interprétés selon *foo*.

Les pointeurs : exemple

int i ;

- *i* représente 4 (variable) octets interprétés comme un entier, cette interprétation peut changer selon la plateforme.
- *&i* est une adresse sur 8 (variable) octets

*int *p=i ;*

- *p* représente une adresse, ici celle du début des 4 octets accessibles via la variable *i*
- *&p* représente une adresse : celle d'un pointeur sur un entier.

*int **q=&p ;*

- *q* représente une adresse, l'adresse contenue dans *q* est interprétée comme l'adresse de début d'une zone de 8 octets qui seront interprétés comme l'adresse de début d'une zone de 4 octets qui seront interprétés selon l'encodage des entiers de la plateforme.
-

The compiling trivia

#QDLE#Q#ABCD*E#25#

- Quelle commande permet de compiler le fichier toto.c en objet ?
 - A. gcc toto.c
 - B. gcc -E toto.c
 - C. gcc -S toto.c
 - D. gcc -c toto.c
 - E. gcc -fPIC toto.c



The compiling trivia

#QDLE#Q#AB*CD#25#

- Quelle commande permet de récupérer le source après l'étape de pré-compilation ?
 - A. gcc toto.c
 - B. gcc -E toto.c
 - C. gcc -S toto.c
 - D. gcc -c toto.c



The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en un exécutable ?
 - A. gcc toto.o main.o
 - B. gcc -shared toto.o main.o
 - C. gcc -static toto.o main.o
 - D. gcc -exec toto.o main.o



The compiling trivia

#QDLE#Q#ABCD*#25#

- Quelle commande permet de lier toto.o et main.o en une bibliothèque statique?
 - A. gcc -shared -o libfoo.a toto.o main.o
 - B. gcc -static -o libfoo.a toto.o main.o
 - C. gcc toto.o main.o -lfoo
 - D. ar rcs libfoo.a toto.o main.o



The compiling trivia

#QDLE#Q#ABCDEF*G#30#

- Pour pouvoir regrouper des objets (.o) dans une bibliothèque dynamique, il faut les compiler avec la ou les options :
 - A. -c
 - B. -E
 - C. -s
 - D. -fPIC
 - E. réponse A&B
 - F. réponse A&D
 - G. réponse B&E



The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en une bibliothèque dynamique?
 - A. gcc -shared -o libfoo.so toto.o main.o
 - B. gcc -dynamic -o libfoo.a toto.o main.o
 - C. gcc toto.o main.o -lfoo
 - D. ar rcsD libfoo.so toto.o main.o



Core

- Lors d'une faute mémoire (accès invalide à une page), le système génère une copie de la mémoire du processus sur le disque : un fichier **core**
- La commande `ulimit` permet de contrôler cette option : `ulimit -c / ulimit -c unlimited`

```
#include <stdlib.h>

int main(){
    char msg[]="ceci est un tres long message...";
    char *p=NULL;
    *p=0;
}
```

```
$ gcc bug.c
$ ulimit -c unlimited
$ ./a.out
Segmentation fault (core dumped)
$ ls -l core
-rw----- 1 allali 262144 core
$ strings core | grep ceci
ceci estH
ceci estH
ceci est un tres long message...
```

Core

- Le core contient l'ensemble des données mémoire : pile, tas, données, code...
- Il est possible de créer un core pour un processus actif avec la commande **gcore**
- Cela permet de contrôler la mémoire d'un processus à divers étapes de son exécution.
- On peut analyser un fichier core avec n'importe quel éditeur, par exemple :

```
$ hexdump -c core | less
```


mais l'interprétation en est très difficile !

core

```
0000000 177 E L F 002 001 001 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000010 004 \0 > \0 001 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000020 @ \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000030 \0 \0 \0 \0 @ \0 8 \0 023 \0 \0 \0 \0 \0 \0 \0 \0
0000040 004 \0 \0 \0 \0 \0 \0 \0 h 004 \0 \0 \0 \0 \0 \0
0000050 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000060 d lv \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000070 \0 \0 \0 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 005 \0 \0 \0
0000080 \0 020 \0 \0 \0 \0 \0 \0 \0 \0 \0 @ \0 \0 \0 \0 \0
0000090 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00000a0 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00000b0 001 \0 \0 \0 004 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00000c0 \0 \0 ` \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00000d0 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00000e0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
00000f0 \0 0 \0 \0 \0 \0 \0 \0 \0 020 ` \0 \0 \0 \0 \0
0000100 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000110 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000120 001 \0 \0 \0 005 \0 \0 \0 \0 @ \0 \0 \0 \0 \0 \0
0000130 \0 0 <C0> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000140 \0 020 \0 \0 \0 \0 \0 \0 \0 <A0> 033 \0 \0 \0 \0 \0
0000150 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 \0 \0 \0
0000160 \0 P \0 \0 \0 \0 \0 \0 \0 <D0> <DB> S <BF> 177 \0 \0
0000170 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000180 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000190 001 \0 \0 \0 004 \0 \0 \0 \0 P \0 \0 \0 \0 \0 \0
00001a0 \0 <D0> <FB> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
00001b0 \0 @ \0 \0 \0 \0 \0 \0 @ \0 \0 \0 \0 \0 \0
00001c0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
00001d0 \0 220 \0 \0 \0 \0 \0 \0 \0 020 <FC> S <BF> 177 \0 \0
00001e0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00001f0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000200 001 \0 \0 \0 006 \0 \0 \0 \0 <B0> \0 \0 \0 \0 \0 \0
0000210 \0 0 <FC> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000220 \0 P \0 \0 \0 \0 \0 \0 P \0 \0 \0 \0 \0 \0
0000230 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 005 \0 \0 \0
0000240 \0 \0 001 \0 \0 \0 \0 \0 \0 200 <FC> S <BF> 177 \0 \0
0000250 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
```

```
0000260 \0 0 002 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000270 001 \0 \0 \0 006 \0 \0 \0 \0 020 001 \0 \0 \0 \0 \0
0000280 \0 <A0> 034 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000290 \0 0 \0 \0 \0 \0 \0 \0 \0 0 \0 \0 \0 \0 \0 \0
00002a0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
00002b0 \0 @ 001 \0 \0 \0 \0 \0 \0 200 036 T <BF> 177 \0 \0
00002c0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00002d0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00002e0 001 \0 \0 \0 004 \0 \0 \0 \0 ` 001 \0 \0 \0 \0 \0
00002f0 \0 <A0> 036 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000300 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000310 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
0000320 \0 p 001 \0 \0 \0 \0 \0 \0 <B0> 036 T <BF> 177 \0 \0
0000330 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000340 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000350 001 \0 \0 \0 006 \0 \0 \0 \0 200 001 \0 \0 \0 \0 \0
0000360 \0 <C0> 036 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000370 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000380 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
0000390 \0 220 001 \0 \0 \0 \0 \0 \0 <F0> 200 lt <FF> 177 \0 \0
00003a0 \0 \0 \0 \0 \0 \0 \0 \0 \0 002 \0 \0 \0 \0 \0 \0
00003b0 \0 002 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
```

gdb

- gdb : GNU Debugger
- permet d'analyser la mémoire avec une couche d'interprétation.
- Il est possible de
 - prendre le contrôle d'un processus en cours d'exécution
 - de lancer un nouveau processus dans gdb
 - d'analyser la mémoire à posteriori hors exécution (core).

gdb : run

- On lance gdb en indiquant le binaire que l'on souhaite analyser, possiblement suivi d'un pid ou d'un fichier core
- gdb va charger le fichier binaire
- Dans le cas d'un pid, il va stopper le processus
- Il est possible de lancer un nouveau processus avec la commande :

```
run arg1 arg2 ...
```

- Ctrl-C permet de suspendre l'exécution du programme, « continue » de la reprendre.

gdb

- En l'absence d'informations additionnelles, gdb ne peut pas associer le contenu mémoire à des variables structurées et donc interpréter la mémoire.
- Il peut cependant indiquer la pile d'appel :

```
(gdb) backtrace
#0  0x00000000004004fa in f ()
#1  0x000000000040050f in g ()
#2  0x0000000000400522 in h ()
#3  0x0000000000400535 in main ()
```

- On voit ici, les adresses dans la pile correspondant aux débuts d'appels.