Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

En bref...

- La *compilation* se fait en deux ou trois étapes:
  - pré-compilation: substitution des macros, gestion des inclusions => un unique fichier source
  - compilation: vérification des types et production d'un binaire avec possiblement des dépendances (symboles non résolus)
  - édition de liens: résolution des dépendances, inclusion de binaire ou ajout de liens vers de bibliothèques dynamiques (pas de vérification de types)
- Une bibliothèque statique est une archive de binaire (.o)
  - son utilisation ajoute les .o de l'archive nécessaires à la production finale
- Une bibliothèque dynamique est un binaire sans dépendances non résolues.
  - son utilisation ajoute un lien entre le binaire final et la bibliothèque.

L'importance des headers !

- Les *headers* servent de « contrats » entre différentes entités compilées séparément (modules, bibliothèques).
- Les *headers* ne contiennent pas d'implémentation!
- Un *header* est un fichier .h qui contient :
  - la déclaration de fonctions
  - la déclaration de structures
  - les macro
  - les « globales » (**avec extern**)

Les headers

- 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 ;
}
```

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

point.h

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

tout le monde peut allouer des structures et accéder au champs

Les headers

- L'avantage de la pré-déclaration de structure 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
```

Remarque: On pourra utiliser

#pragma once

mais cela ne fait pas parti du standard et peut ne pas être géré par tous les compilateurs.

Les headers

- 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 (édition de liens), 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 headers

header.h :

```
#ifndef HEADER_H_
#define HEADER_H_

extern int v ;

#endif
```

header.c :

```
#include "header.h"

int v ;
```

sample.c :

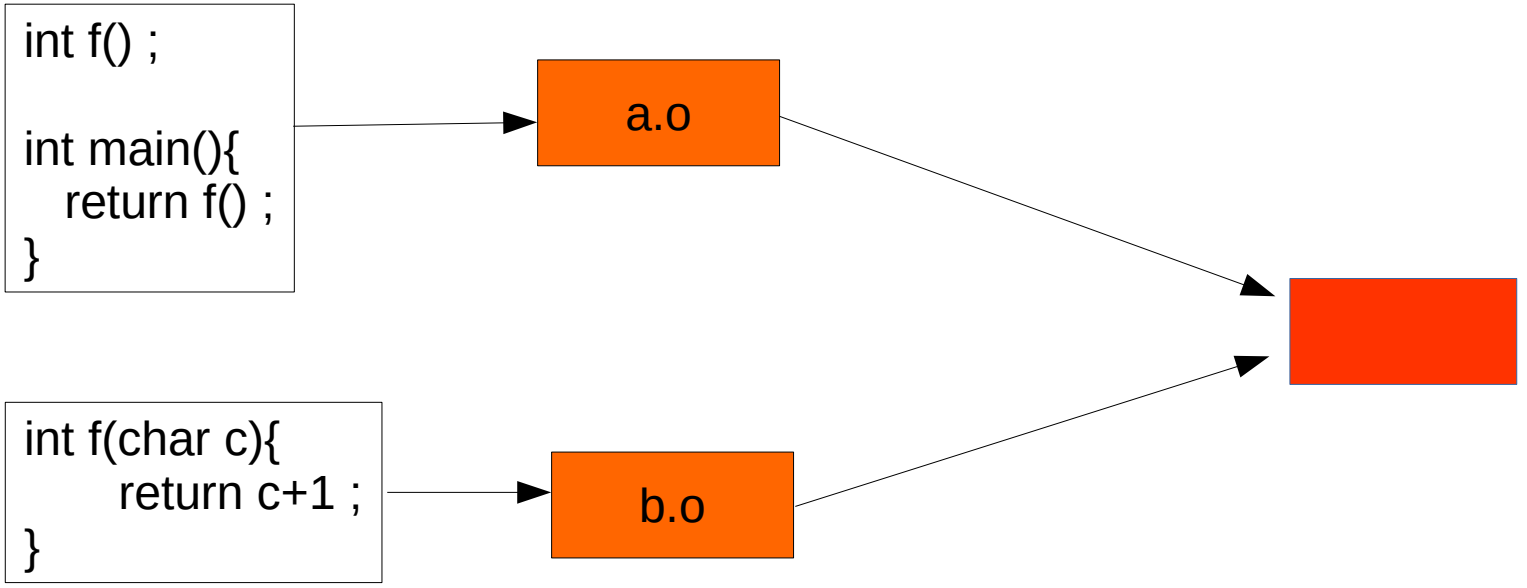
```
#include <header.h>

int main(){return v;}
```

```
$ gcc -c header.c
$ gcc -c -I. sample.c

$ nm header.o sample.o

header.o:
0000000000000000 B v

sample.o:
0000000000000000 T main
                 U v
```

Les headers

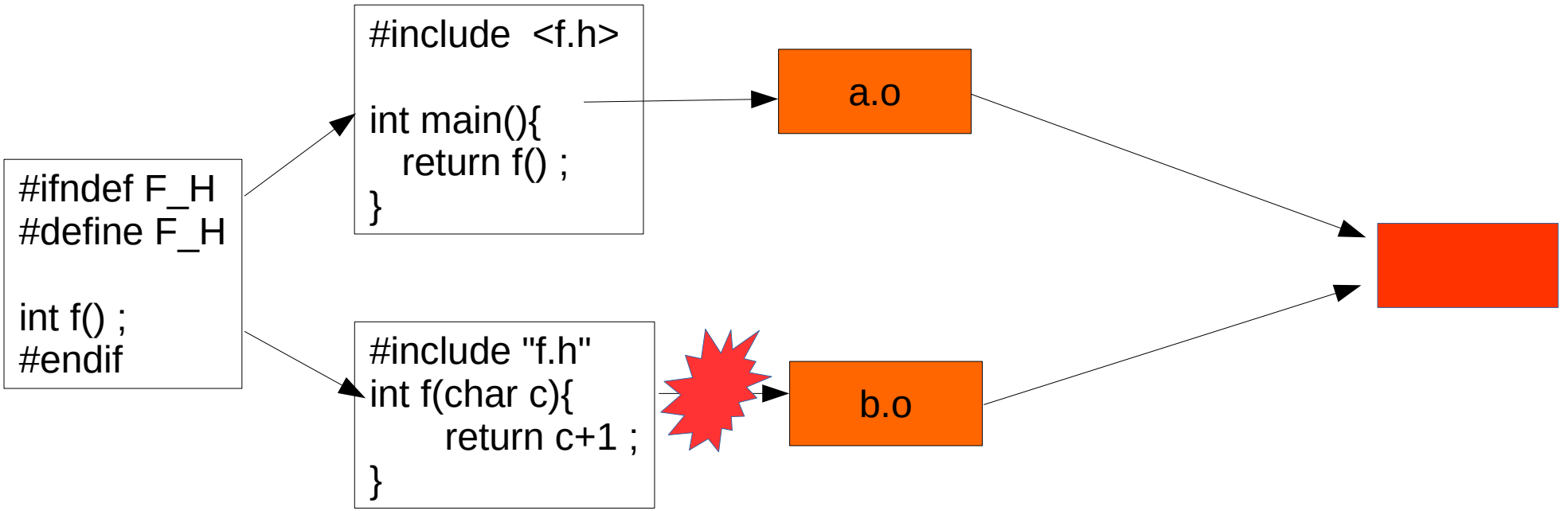
- Après l'étape de pré-compilation, tous les header inclus (directement ou indirectement) sont regroupés en un seul source.
- Si un macro est définie plusieurs fois, il y a erreur dans la pré-compilation.
- Si une structure, globale ou fonction est définie plusieurs fois, il y a erreur lors de la compilation,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

Les headers 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 headers 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 headers: qq règles

- Toujours protéger contre l’inclusion multiple
- Mettre le minimum nécessaire d’include dans le .h
- Ne mettre que les fonctions faisant parties de l’API du module
- Pas de globales directement: extern
- On utilise #include "foo.h" quand on implémente le contenu de foo.h ou que notre code fait parti du même *module*
- On utilise #include <foo.h> quand on utilise/dépend du contenu de foo.h

Les headers

#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#ABC*D#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

- Lors du chargement d'un exécutable toutes les bibliothèques dont il dépend sont d'abord chargées.
- 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
```

Exécution et chargement

#QDLE#Q#AB*#20#

- Lors du chargement d'un exécutable toutes les bibliothèques dont il dépend sont d'abord chargées.
- 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 tailles fixes organisées en zones :
  - la pile
  - le tas
  - le code du binaire
  - le code des bibliothèques dynamiques liées
  - les variables globales (initialisées et non initialisées)

processus

- chaque processus a un numéro unique qu’on appelle le PID
- On peut avoir la liste des processus en cours d’exécution avec la command $ps
- On peut également voir les ressources utilisées par les processus avec la command $top

Programme, processus et mémoire...

- Chaque processus a son propre espace d'adressage. Cette *isolation* est gérée par le processeur et garantie la sécurité du système.
- Le système (kernel) 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

note: le kernel s'execute à une niveau différent sur le processeur ce qui lui donne accès à l'intégralité des ressources

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)=1 048 576 pages.
- Ainsi, chaque page peut être :
  - non adressable (non associée)
  - adressable avec des droits:
    - en lecture
    - en écriture
    - en exécution

C'est l'accès à une page non adressable qui mène au SEGFAULT, cela indépendamment de la structuration mémoire (variables etc...)

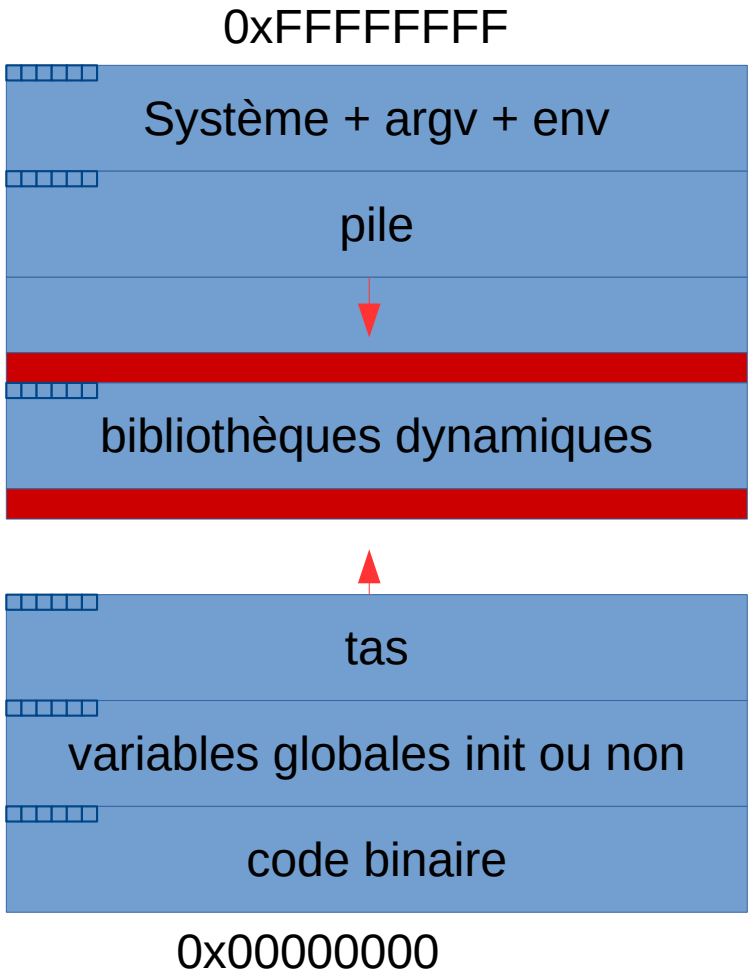
Question

#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...

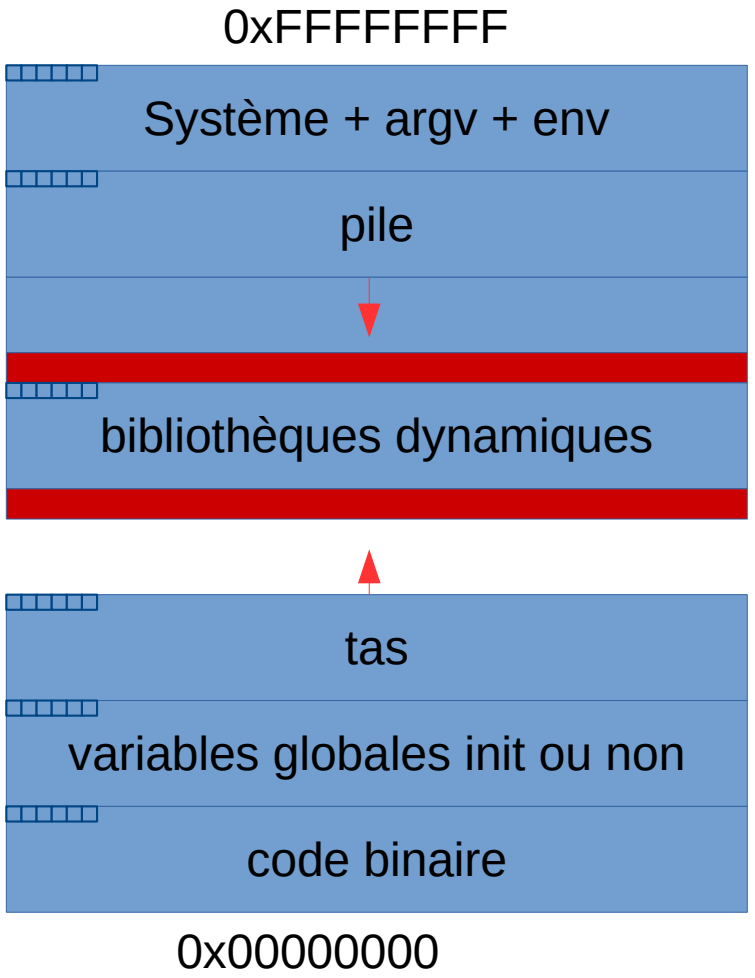
- Lors de l'exécution d'un programme un espace mémoire virtuel est créé.



- Le système (kernel) met en place une table d'adressage utilisée par le processeur.

Programme, processus et mémoire...

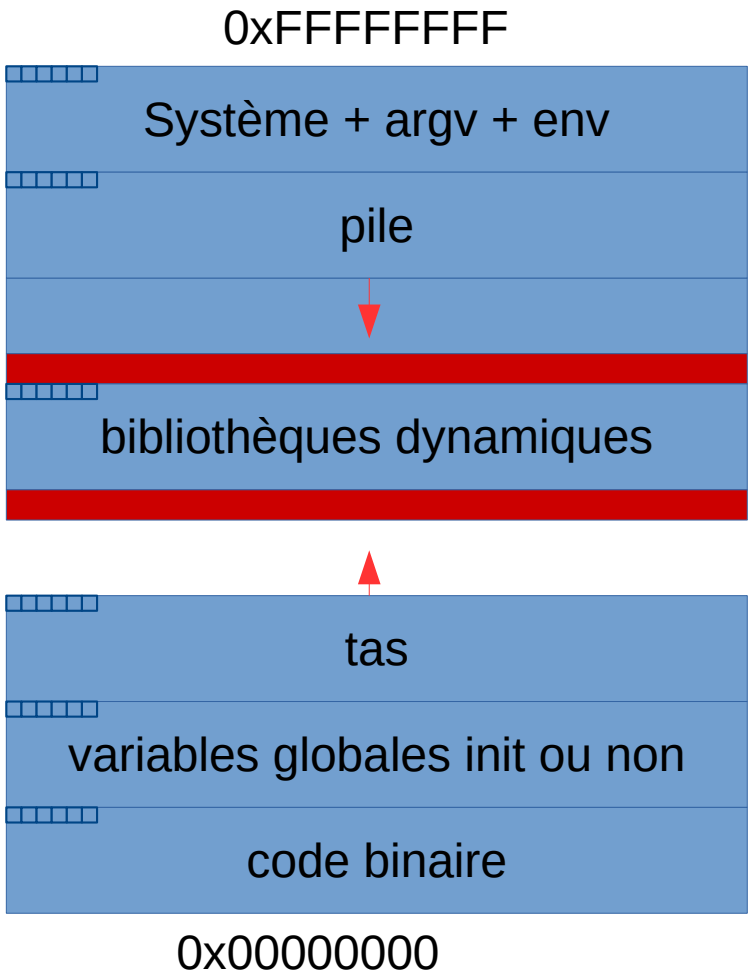
- Un accès incompatible mène à l'émission d'un signal qui souvent se traduit par un SEGFAULT



- 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...

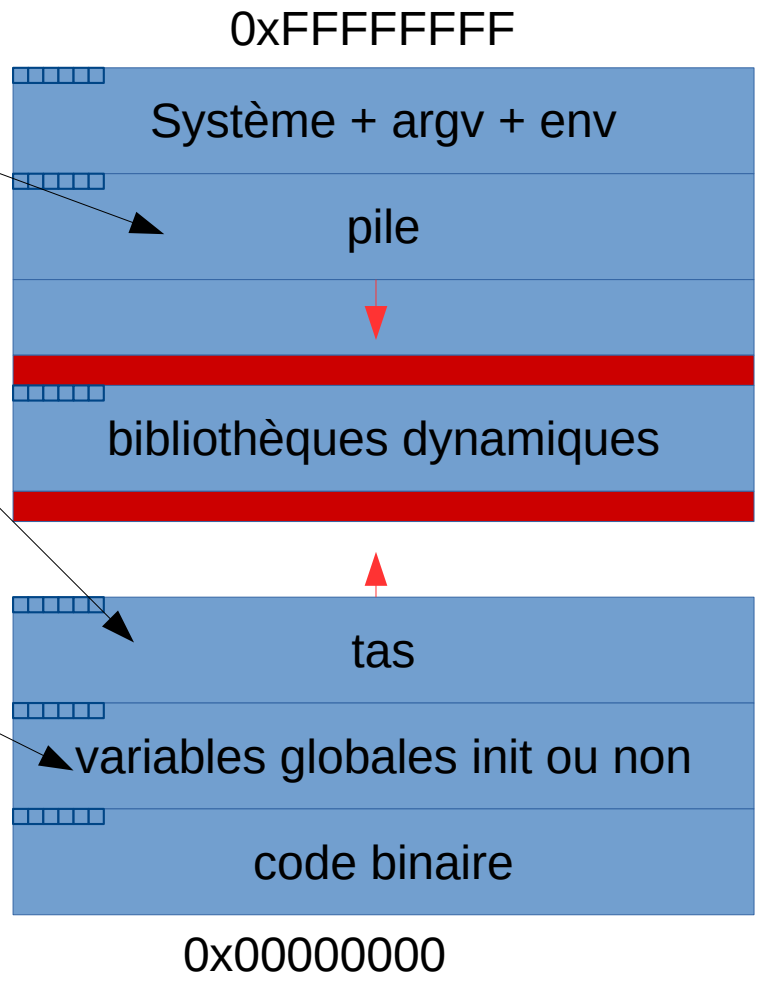
#QDLE#Q#AB*CD#25#



- 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 exécution seul.

Programme, processus et mémoire...

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



Les variables

- En C les variables sont typées. Le type d'une variable fixe la taille mémoire requise pour stocker la valeur de la variable
- A tout moment de l'exécution du programme, si vous pouvez accéder à une variable c'est qu'il y a un espace mémoire réservé pour stocker celle-ci: cela est **garanti** par le langage.

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.
- A l'exécution, le nom des variables et des *fonctions* n'existent pas.
- 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
- *sizeof* permet de connaître la quantité d’espace mémoire requise pour stocker une variable ou un type: sizeof(i) ou sizeof(int) etc...

Les pointeurs

- Les pointeurs sont des variables qui stockent une adresse: tous les pointeurs utilisent la même quantité d'octets: 4 sur une architecture 32 bits et 8 sur une architecture 64 bits.
- C'est pour cela qu'on peut déclarer un pointeur vers une structure déclarée mais non définie.
- Le type pointé permet d'interpréter la mémoire à l'adresse conservée dans le pointeur: cela suppose une certaine quantité d'octets réservés.

void *

- Comme tous les pointeurs utilisent la même quantité de mémoire (celle nécessaire au stockage d'une adresse), il est possible de déclarer un pointeur sans préciser la nature du type à l'adresse stockée:
  - void *p
- Dans ce cas, il n'est pas possible de faire de l'arithmétique (p+5 ou p[3]) car on ne sait pas de quelle quantité il faut se déplacer
- Cela permet le stockage d'adresses mémoires mais suppose à un moment d'utiliser le *cast* pour exploiter l'adresse conservée dans le pointeur.

Les tableaux

- Le déclaration d'un tableau de type:
  - int t[10]
- L'espace mémoire associé est dans la pile
- Il n'y a pas de vérification en cas de dépassement du tableau
- t désigne l'adresse de début du tableau
- &t désigne également cette adresse
- sizeof(t) correspond au nombre d'octet pour conserver l'intégralité du tableau

ainsi t ne doit pas être vu comme un pointeur puisqu'il n'y a pas un espace mémoire réservé pour conserver l'adresse: t désigne directement le début du tableau.

Fonction

- Le nom d'une fonction est une variable non modifiable.
- void f(int i){....}
- Le symbole *f* correspond à l'adresse de la première instruction du code associé à la fonction.
- L'opération parenthèse: f(...) correspond à un saut à l'adresse associée à f
- Il est possible de consulter l'adresse via l'utilisation de f: printf(« adresse de la fonction %p\n »,f);

Les variables statiques

- Dans le cas des variables statiques
- v représente le contenu d'une zone mémoire fixe.
- L'adresse mémoire &v sera toujours la même au long de l'exécution.
- L'interprétation de cette zone dépend du type de v.
- La « valeur » (c'est à dire le contenu de la mémoire) peut charger au cours de l'exécution.

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 (accessibles) 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
0000000 177   E   L   F 002 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000010 001  \0   >  \0 001  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0   <D8>   d  \0  \0  \0  \0  \0  \0
0000030  \0  \0  \0  \0   @  \0  \0  \0  \0  \0   @  \0  \t  \0 006  \0
0000040   c   e   c   i       e   s   t       u   n       t   r   e   s
0000050       t   r   e   s       t   r   e   s       t   r   e   s
0000060   l   o   n   g       m   e   s   s   a   g   e   c   e   c   i
0000070       e   s   t       u   n       t   r   e   s       t   r   e
0000080   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
```

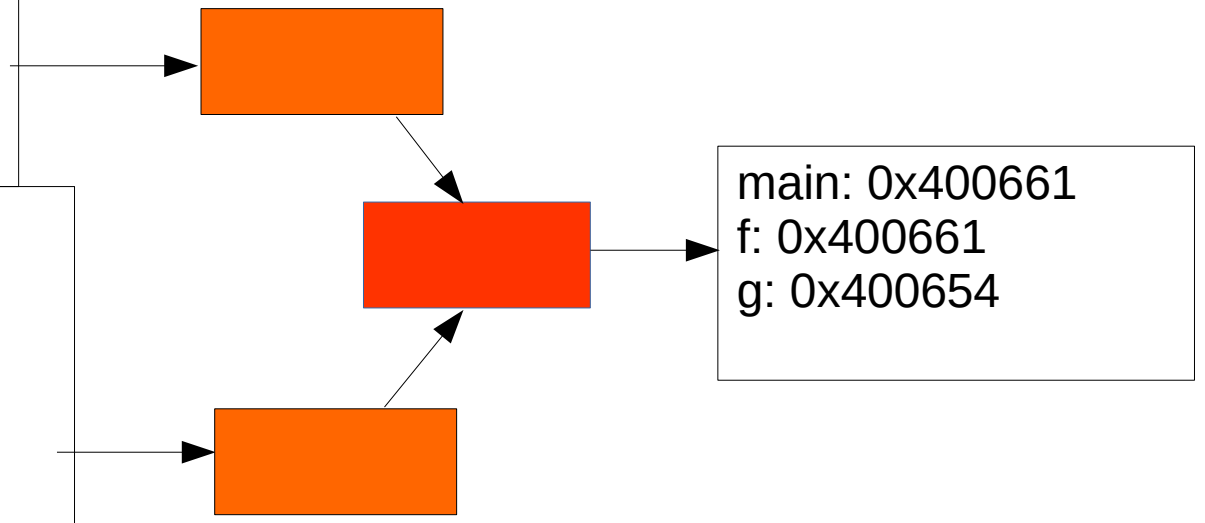
Au lancement, sizeof(int) octets sont réservés à l'adresse 0x601044 et sont initialisés à la valeur (int)0.

Les variables statiques

- Les chaînes de caractères sont des données statiques correspondantes à des tableaux de caractères (octets), elles sont mutualisées par unité de compilation (.c).

```
void g(){
  char *s="hello";
  printf("g: %p\n",s);
}
```

```
void g();
void f(){
  char *s="hello";
  printf("f: %p\n",s);
}
int main(){
  char *s="hello";
  printf("main: %p\n",s);
  f();
  g();
}
```



Le tableau de caractères associé aux chaînes statiques se termine par le caractère '\0' (0x00).

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

- Il existe un mot clé dans la norme C : auto

  auto int i ; → int i ;

Allocation automatique

#QDLE#Q#AB*#20#

- Il existe un mot clé dans la norme C : auto

  auto int i ; → int i ;

- Soit la fonction f suivante :

```
int f(int i){
}
```

- Lors d'un appel à f, l'adresse de la variable i est toujours la même ?
  - A. vrai
  - B. faux

Allocation automatique

- Règle : une 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 connaître 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

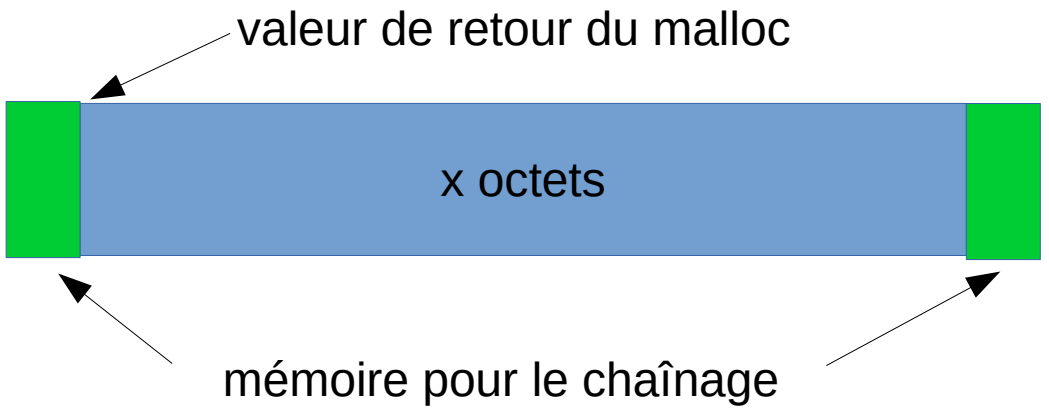
- Au lancement du processus, il y a une plage d'adresse non réservée (aucune page n'y est associée).
- L'appel à la fonction sbkr permet d'étendre cette plage d'adresse. La mémoire ainsi réservée peut être utilisée par le processus.
- Les fonctions malloc/free permettent une gestion de cette mémoire.

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 vu du système, les pages correspondent à des zones accessibles (valides).
- malloc et free fonctionnent sur un système de listes chaînées

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 correspondants au chaînage de malloc :



malloc

- démonstration :

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

int main(){
  int i=0;
  for(i=0;i<100000000;++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*100000000);
  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 statique) 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 (statique).
- 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 allocations n'est pas connue par malloc : il est important d'interpréter ces zones correctement (taille).