

Programmation C avancée

Concepts & Outils
pour le développement



Introduction

- Sur l'informatique et la mémoire...



```
1001000110100110110010000010010011001111
01101100000100001111000011110100110010110
0000110100111101001001111100111000001101
10111110011000001110000110110011001011100
00111100111110101111001011001011000001110
10011011111000001110011111000011001011100
00111010111000001110100110111110000011110
01110111111101011000001110100110111111001
00110000111110011011101000001001001100000
11010001101111111000011001011000001111001
11011111110101100000110010111011101101010
11011111111001100000110100111101001000010
```

- Un ordinateur n'est qu'une machine qui transforme et 0 en 1 et inversement
- Tout est dans l'interprétation des suites de 0 et de 1

Objectifs du module

- Maîtrise des outils de compilation en C
- Bonne compréhension des mécanismes de gestion de la mémoire
- Maîtrise des outils de développement :
 - Gestion des sources
 - Tests
 - Intégration continue
 - ...
- Écriture de programmes : sains, maintenables, robustes, évolutifs

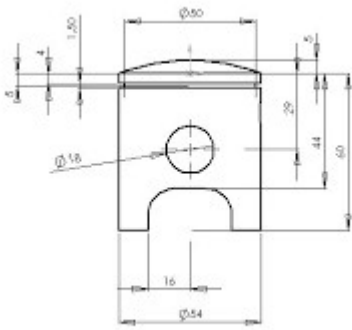


Plan

- Des sources à la mémoire
- Allocations et analyses
- Convention de code, Documentation
- Gestion des sources, dépôt et compilation auto
- L'intégration continue
- Les tests : types, utilisation et framework
- Couverture & intégration continue (suite)
- Performance : localité et analyse
- Pointeurs de fonction, chargement dynamique

Des sources à la mémoire

- Un fichier source est un texte exprimé dans un langage de programmation.
- Un binaire est un fichier qui contient des instructions machines.
- Un exécutable est un binaire ayant un point de début d'exécution.



SOURCE
Hello.c



binaire
Hello.o



exécutable
Hello

Script shell

```
#!/bin/bash
```

```
echo « hello »  
echo « world »  
echo $USER
```

```
#!/usr/bin/python
```

```
import sys  
  
print sys.argv
```

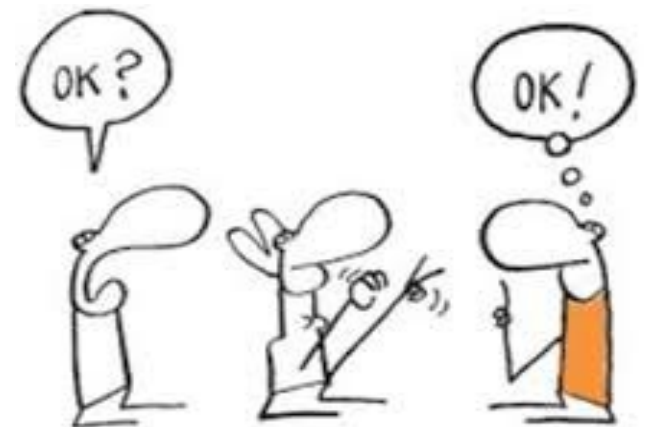
- Un script est
 - un fichier source ?
 - un fichier binaire ?
 - un fichier binaire exécutable ?

Un script est un fichier source qui est interprété.

Dans le cas d'un script shell, l'interpréteur est le shell. Dans celui de python c'est le programme python.

La programmation

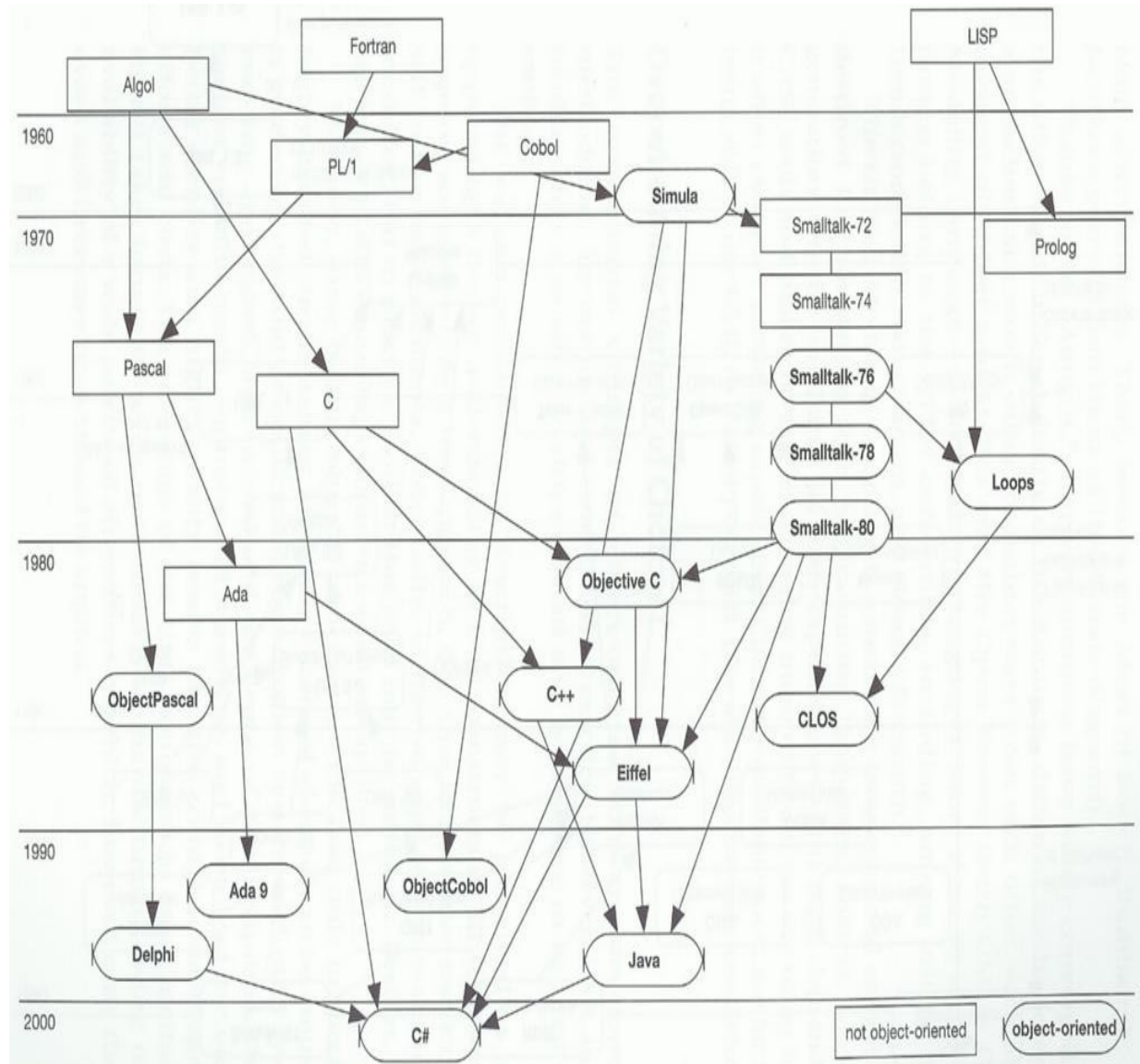
- Un langage de programmation nous aide à structurer la mémoire et son interprétation.
- Il se place entre le programmeur, dont la vision est très haut niveau et la machine dont la « vision » est très bas niveau.
- Il existe de nombreux langages : typés, non typés, compilés, interprétés, impératifs, fonctionnels,....
- La langage C est un langage impératif typé et compilé.



Combien existe-t-il de langage de programmation ?

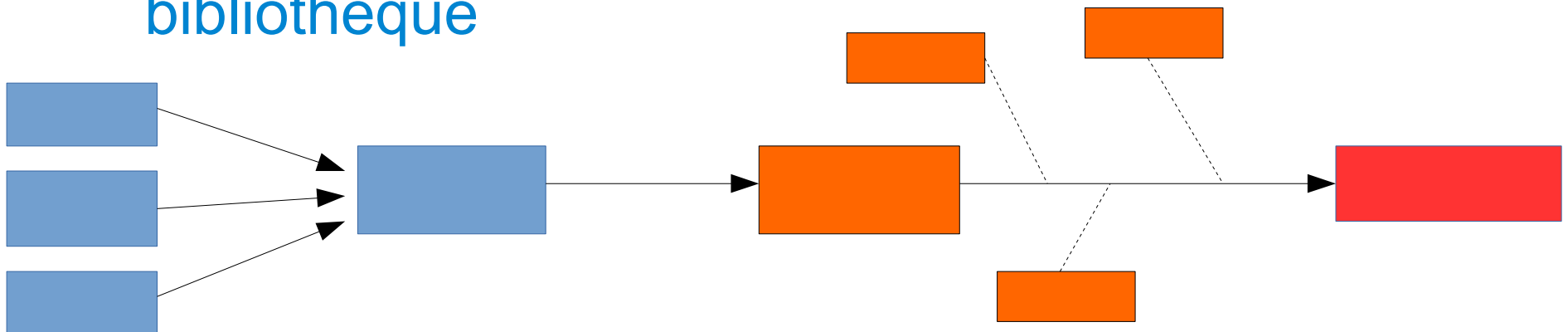
- A. plus de 2000
- B. plus de 1000
- C. plus de 500
- D. moins de 500

wikipedia
référence
à ce jour
698 langages



Fichier source et compilation

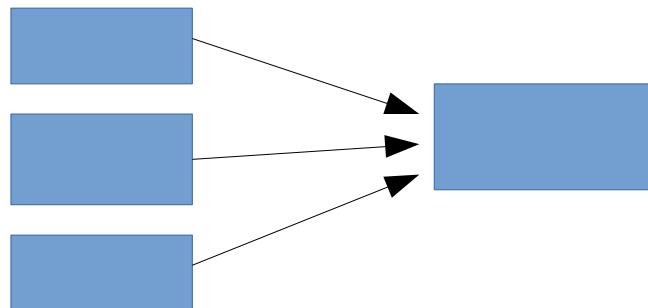
- Le compilateur est un programme qui lit des sources et produit un binaire possiblement exécutable
- La compilation comporte trois étapes :
 - Le pré-processing : sources => source
 - La compilation : source => binaire
 - L'édition de lien : binaires => exécutable, bibliothèque



La pré-compilation



- La pré-compilation prend un ou plusieurs fichiers sources en entrée et produit un unique fichier source exempt de macro :
 - Tous les « #include » sont remplacés par leurs contenus
 - Les macro (#define, #ifdef....) sont interprétées et remplacées par leurs évaluations
- Le résultat est un unique fichier source C sans dépendance



La pré-compilation : exemple

```
#define N 10
```

```
int main(){  
    int i,j=0;  
    for(i=0;i<N;++i)  
        j+=i;  
    return j;  
}
```

```
# 1 "exemple.c"
```

```
# 1 "<built-in>"
```

```
# 1 "<command-line>"
```

```
# 1 "/usr/include/stdc-predef.h" 1 3 4
```

```
# 1 "<command-line>" 2
```

```
# 1 "hello.c"
```

```
int main(){  
    int i,j=0;  
    for(i=0;i<10;++i)  
        j+=i;  
    return j;  
}
```



gcc -E exemple.c

La pré-compilation : exemple 2

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

#define MESSAGE "hello\n"

int main(){
    printf(MESSAGE);
    return EXIT_SUCCESS;
}
```

gcc -E hello.c

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
.....
extern int printf (const char *__restrict __format, ...);
.....
# 3 "hello.c" 2

int main(){
    printf("hello\n");
    return 0;
}
```

La pré-compilation

- Quelques mots sur les macros :

substitution :

```
#define NAME VALUE
```

```
#define NAME(arg) VALUE
```

mise en chaine :

```
#define s(x) #x
```

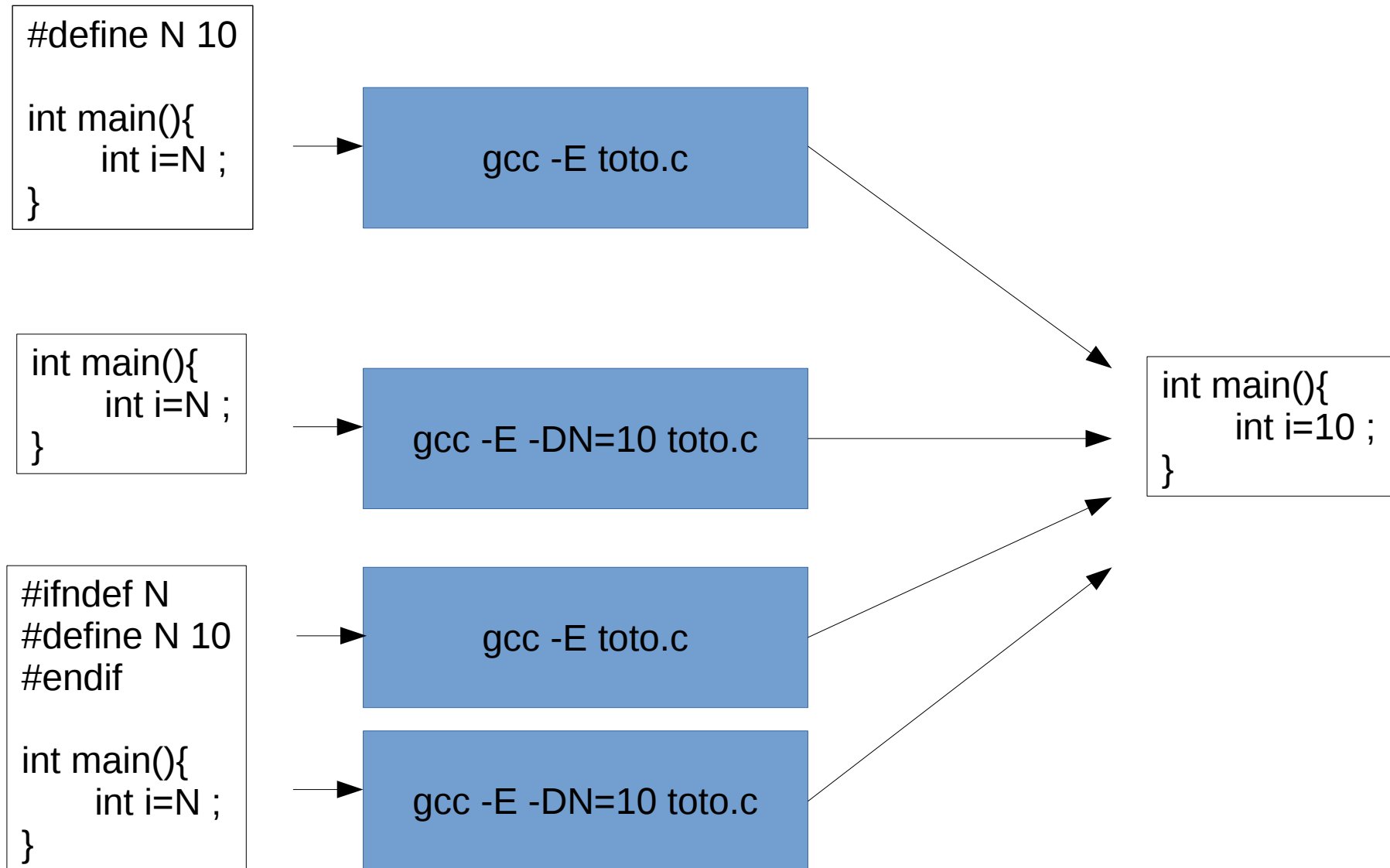
Retrait :

```
#undef
```

- Branchement conditionnel :

```
#ifdef, #ifndef, #endif, #else, #elif, #defined
```

Macros : substitution



macros : concaténation et mise en chaine

```
#define str(x) struct point_##x { \  
    x value ; \  
}
```

```
str(int) ; \  
str(float) ;
```

gcc -E toto.c

```
struct point_int { \  
    int value ; \  
};
```

```
struct point_float{ \  
    float value ; \  
};
```

```
#define msg(x) if (x) { \  
    printf(#x) ; \  
}
```

```
int main(){ \  
    int i=0 ; \  
    msg(i+1) ; \  
}
```

gcc -E toto.c

```
int main(){ \  
    int i=0 ; \  
    if (i+1) { printf("i+1") ; } ; \  
}
```

La compilation

- Traduction d'un fichier source en un fichier binaire.



- Les instructions sont transformées en code machine, regroupé par fonctions.
- Les fonctions non implémentées sont indiquées comme « symboles à résoudre ». Seul le nom est indiqué.
- Les variables globales externes sont déclarées
- Pour compiler, toute fonction doit être soit implémentée, soit déclarée.

Compilation : objdump

```
int f(int ) ;
```

```
int main(){  
    int i =0;  
    return f(i) ;  
}
```

gcc -c toto.c

toto.o

objdump -t toto.o

toto.o: file format elf64-x86-64

SYMBOL TABLE:

0000000000000000	df *ABS*	0000000000000000	toto.c
0000000000000000	d .text	0000000000000000	.text
0000000000000000	d .data	0000000000000000	.data
0000000000000000	d .bss	0000000000000000	.bss
0000000000000000	d .note.GNU-stack	0000000000000000	.note.GNU-stack
0000000000000000	d .eh_frame	0000000000000000	.eh_frame
0000000000000000	d .comment	0000000000000000	.comment
0000000000000000 g	F .text	000000000000001b	main
0000000000000000	*UND*	0000000000000000	f

Compilation

```
extern float x ;  
int f(int );  
int i ;  
static int j ;  
int g(){  
    static int k=0;  
    return k++ ;  
}  
  
int main(){  
    int i =0;  
    return f(i) ;  
}
```

gcc -c a.c

a.o

nm a.o

```
                U f  
0000000000000000 T g  
0000000000000004 C i  
0000000000000000 b j  
0000000000000004 b k.1748  
0000000000000015 T main  
                U x
```

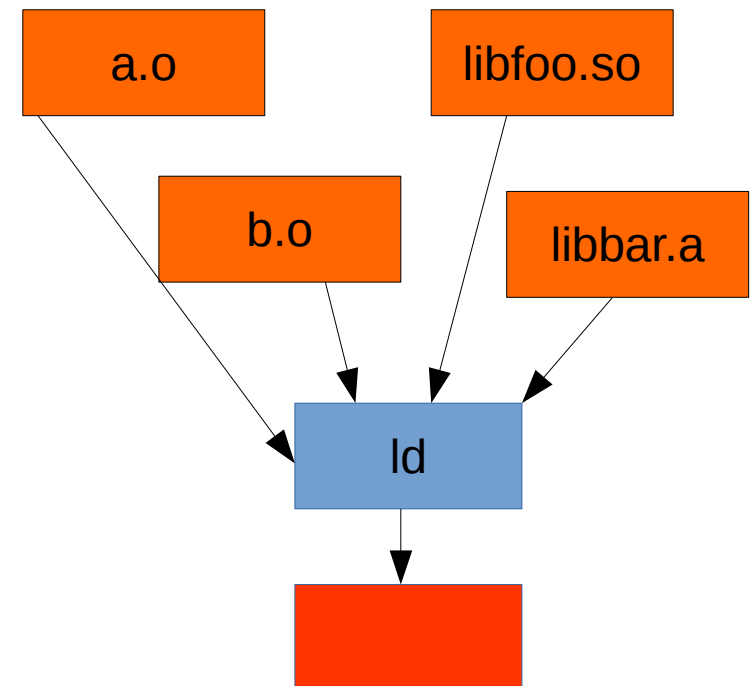
U : The symbol is undefined.
T : The symbol is in the text (code) section
C : The symbol is common. Common symbols are uninitialized data.
b : The symbol is in the uninitialized data section (known as BSS).

- A la fin de cette étape, on dispose d'un unique fichier dit « objet » qui contient :
 - Un ensemble de variables globales
 - Un ensemble de variables statiques
 - Un ensemble de fonction : nom + instructions
 - Un ensemble de symboles « manquants »

L'édition de liens



- L'édition de lien consiste à interconnecter différents « objets » au sein d'une bibliothèque ou bien d'un binaire exécutable.
- En entrée, l'éditeur de lien (ld) prend :
 - des fichiers objets
 - des bibliothèques statiques
 - des bibliothèques dynamiques
- En sortie on obtient :
 - une bibliothèque **dynamique**
 - un binaire exécutable



L'édition de liens



- L'éditeur de lien énumère l'ensemble des symboles fournis et manquants
- Lorsqu'un symbole est manquant dans un objet mais fournit dans un autre, alors les deux sont liés et le symbole est **résolu**
- Si un symbole manquant n'est fournit par aucun autre objet alors il est dit manquant :

```
gcc a.o -o a
```

```
a.o: In function `main':  
a.c:(.text+0xa): undefined reference to `f'  
collect2: error: ld returned 1 exit status
```



L'édition de liens

#QDLE#Q#AB*C#30#



```
> cat a.c
int main(){
    return f();
}
> gcc -c a.c
> nm a.o :
                 U f
00000000 T main
```

```
gcc -c b.c
nm b.o :

00000004 C f
```

gcc a.o b.o

```
0000000000601038 B __bss_start
0000000000601038 b completed.7259
0000000000601028 D __data_start
0000000000601028 W data_start
0000000000400430 t deregister_tm_clones
00000000004004b0 t __do_global_dtors_aux
0000000000600e18 t __do_global_dtors_aux_fini_array_entry
0000000000601030 D __dso_handle
0000000000600e28 d _DYNAMIC
0000000000601038 D _edata
0000000000601040 B _end
000000000060103c B f
0000000000400584 T _fini
00000000004004d0 t frame_dummy
0000000000600e10 t __frame_dummy_init_array_entry
00000000004006b8 r __FRAME_END__
0000000000601000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
00000000004003a8 T _init
0000000000600e18 t __init_array_end
0000000000600e10 t __init_array_start
0000000000400590 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000600e20 d __JCR_END__
0000000000600e20 d __JCR_LIST__
                 w _Jv_RegisterClasses
0000000000400580 T __libc_csu_fini
0000000000400510 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000004004f6 T main
0000000000400470 t register_tm_clones
0000000000400400 T _start
0000000000601038 D __TMC_END__
```

→ segfault, pourquoi ?

- A. parce que f n'est pas initialisée dans b.c
- B. parce que f est une fonction dans a.c mais une variable globale dans b.c
- C. parce que f est une variable globale dans a.c mais une fonction dans b.c

L'édition de liens

- Aucune cohérence de type des symboles n'est effectuée lors de l'édition
 - > d'où l'importance de fichier d'entête s'assurant au moment de la compilation de cette cohérence
- Si un symbole est présent en plusieurs versions, une erreur est signalée :

```
gcc a.o b.o c.o
```

```
c.o: In function `pow':
```

```
c.c:(.text+0x0): multiple definition of `pow'
```

```
b.o:/tmp/b.c:1: first defined here
```

```
collect2: error: ld returned 1 exit status
```

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_biblio.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 _
0000000000040051b T c
00000000000601038 b completed.7259
0000000000040050b T d
000000000004004f6 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)
```



Les bibliothèques dynamiques

- création :

- Il faut un ensemble d'objets compilés avec l'option -fPIC (code repositionnable)

```
gcc -fPIC -c b.c
```

- La création se fait avec l'option -shared :

```
gcc -shared -o libnom.so b.o
```

- Sous UNIX un bibliothèque a pour nom lib....so

- L'utilisation se fait avec l'option -l

```
gcc a.o -lnom
```

- L'éditeur va chercher « libnom.so » dans des répertoires par défaut. L'option -L permet d'en ajouter :

```
gcc a.o -lnom -L.
```

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 ;  
}
```

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 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 :

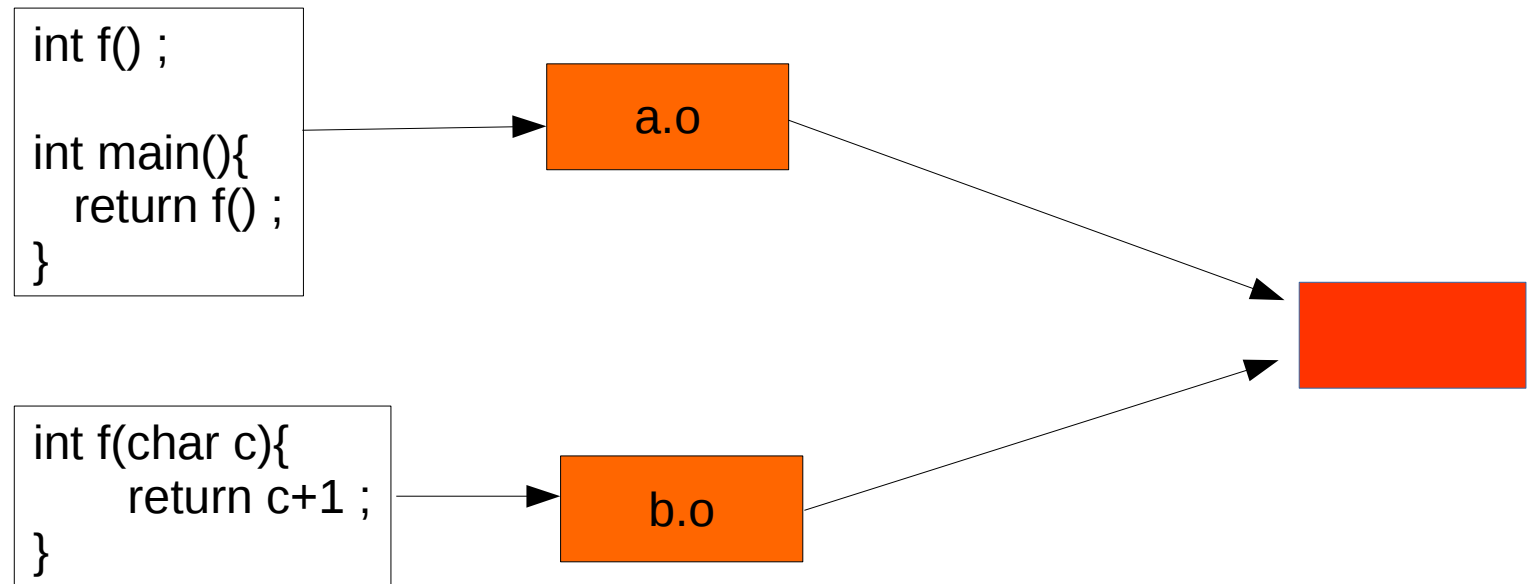
```
extern int v ;
```

```
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.

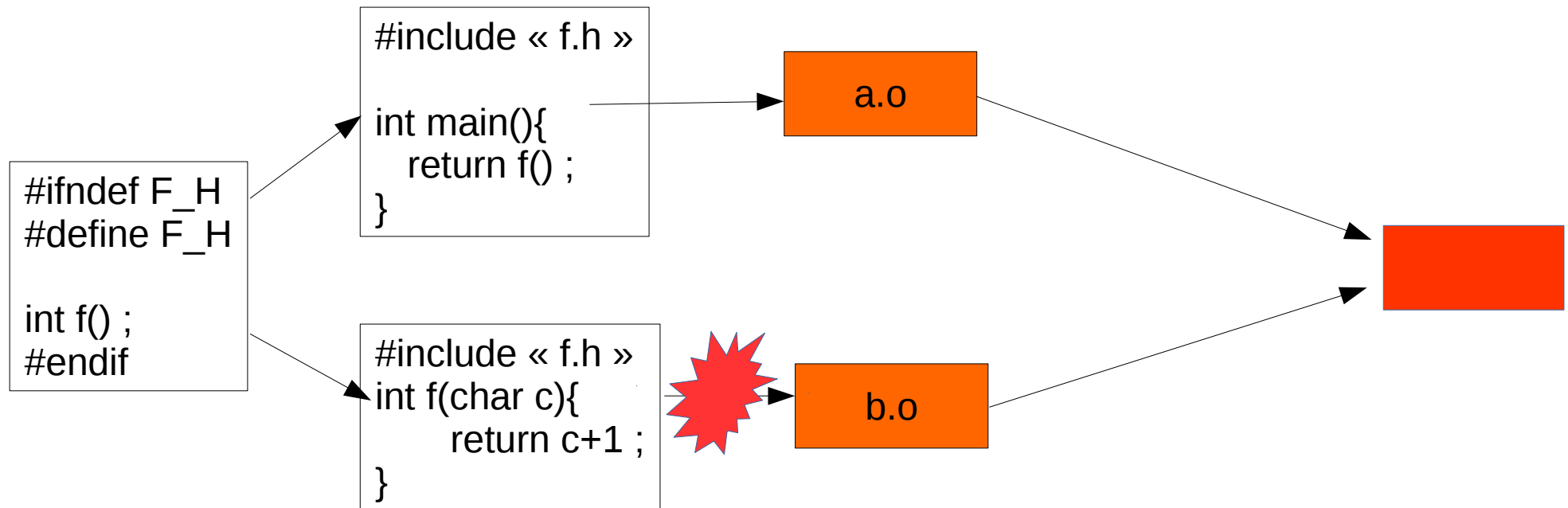
Les header par l'exemple



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.

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...

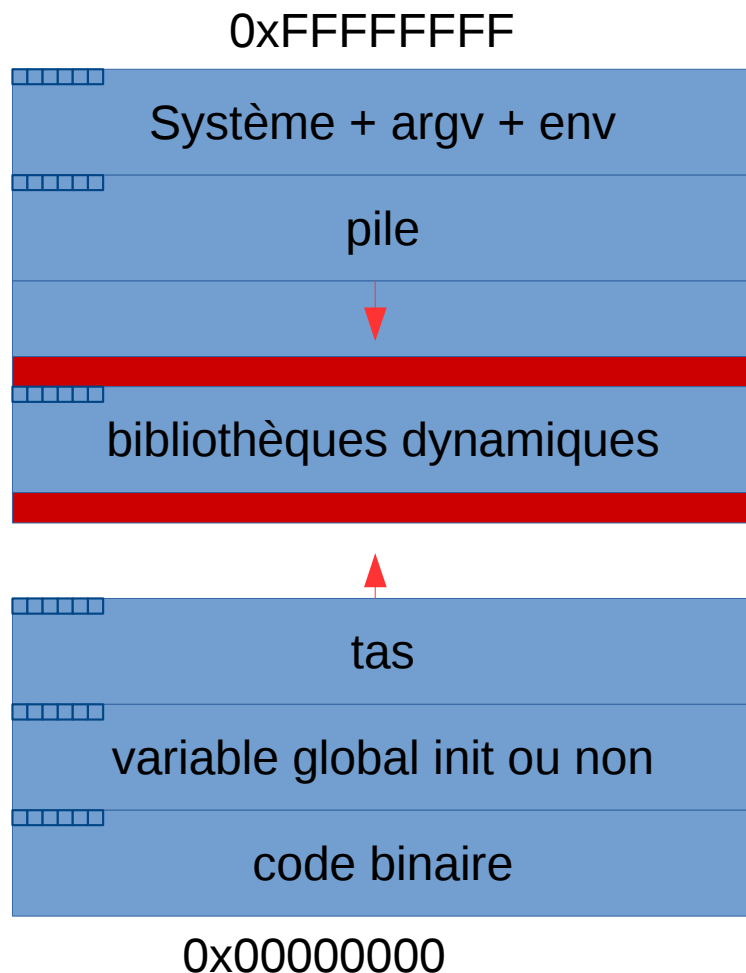
- La taille des pages est fixe :

```
> 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

Programme, processus et mémoire...

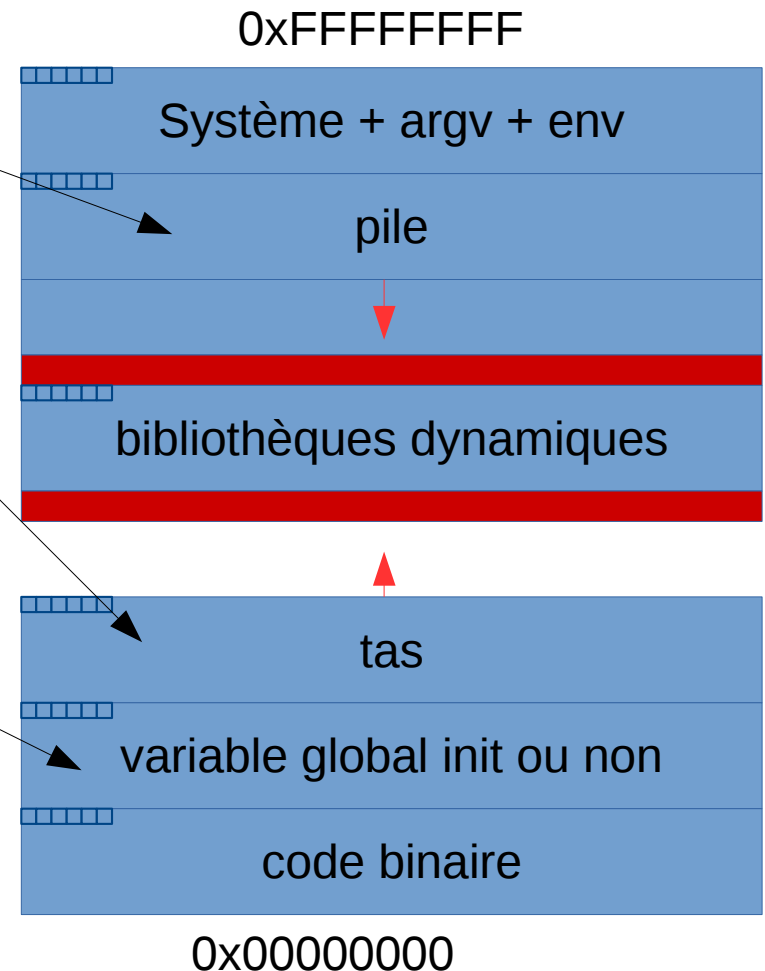
- Un accès incompatible mène à l'émission d'un signal qui souvent se traduit par un SEGVFAULT



- 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...

- 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 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

- 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

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 changer au cours de l'exécution.

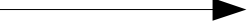
Les variables statiques

```
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
```

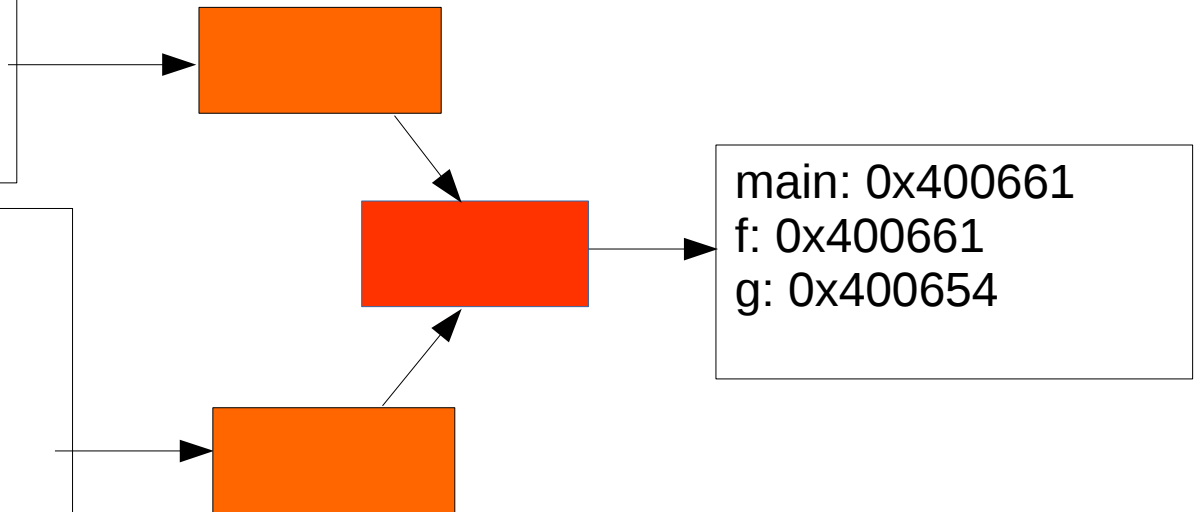
Au lancement, `sizeof(int)` octets sont réservés à l'adresse 0x601044 et sont initialisés à la valeur (int)0.

Les variables statiques

- Les chaînes de caractères sont des données statiques correspondant à des tableaux de caractères, 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 caractère 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

- 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 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)

L'allocation dynamique

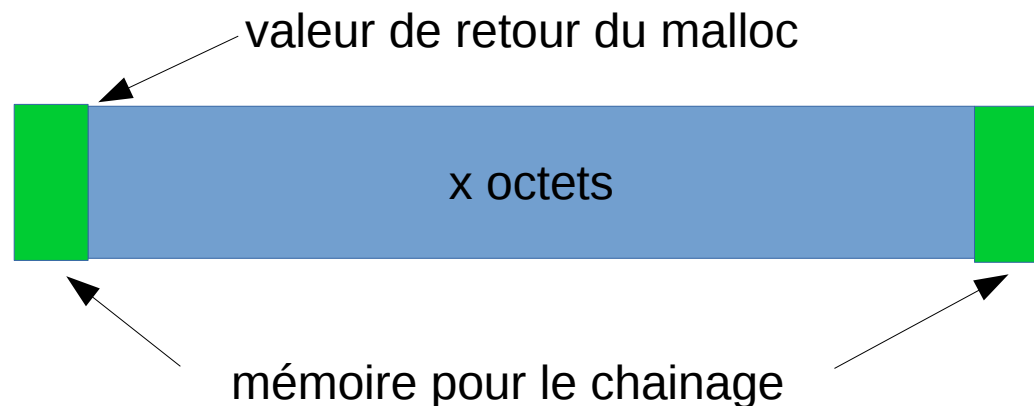
- Au lancement du processus, il y a une plage d'adresse non réservée (aucune page n'y est associée).
- L'appel à la fonction **sbkr** permet d'étendre cette plage d'adresse. La mémoire ainsi réservée peut être utilisée par le processus.
- Les fonctions malloc/free permettent une gestion de cette mémoire.

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 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).

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
0000010 004 \0 > \0 001 \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
0000030 \0 \0 \0 \0 @ \0 8 \0 023 \0 \0 \0 \0 \0 \0 \0
0000040 004 \0 \0 \0 \0 \0 \0 \0 h 004 \0 \0 \0 \0 \0
0000050 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000060 d \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
0000080 \0 020 \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
00000a0 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
00000b0 001 \0 \0 \0 004 \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
00000e0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0
00000f0 \0 0 \0 \0 \0 \0 \0 \0 \0 020 ` \0 \0 \0 \0
0000100 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000110 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000120 001 \0 \0 \0 005 \0 \0 \0 \0 @ \0 \0 \0 \0 \0
0000130 \0 0 <C0> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0
0000140 \0 020 \0 \0 \0 \0 \0 \0 \0 <A0> 033 \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
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
00001a0 \0 <D0> <FB> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0
00001b0 \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
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
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
0000210 \0 0 <FC> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0
0000220 \0 P \0 \0 \0 \0 \0 \0 \0 P \0 \0 \0 \0 \0
0000230 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 005 \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
```

```
0000260 \0 0 002 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000270 001 \0 \0 \0 006 \0 \0 \0 \0 020 001 \0 \0 \0 \0
0000280 \0 <A0> 034 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0
0000290 \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
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
00002d0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
00002e0 001 \0 \0 \0 004 \0 \0 \0 \0 ` 001 \0 \0 \0 \0
00002f0 \0 <A0> 036 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0
0000300 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000310 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0
0000320 \0 p 001 \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
0000340 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000350 001 \0 \0 \0 006 \0 \0 \0 \0 200 001 \0 \0 \0 \0
0000360 \0 <C0> 036 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0
0000370 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000380 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0
0000390 \0 220 001 \0 \0 \0 \0 \0 \0 <F0> 200 \0 <FF> 177 \0 \0
00003a0 \0 \0 \0 \0 \0 \0 \0 \0 \0 002 \0 \0 \0 \0 \0
00003b0 \0 \0 002 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
00003c0 001 \0 \0 \0 005 \0 \0 \0 \0 <B0> 003 \0 \0 \0 \0
00003d0 \0 ` 215 \0 <FF> 177 \0 \0 \0 \0 \0 \0 \0 \0
00003e0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00003f0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 004 \0 \0
0000400 \0 <D0> 003 \0 \0 \0 \0 \0 \0 200 215 \0 <FF> 177 \0 \0
0000410 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000420 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000430 001 \0 \0 \0 005 \0 \0 \0 \0 <F0> 003 \0 \0 \0 \0
0000440 \0 \0 ` <FF> <FF> <FF> <FF> <FF> \0 \0 \0 \0 \0 \0
0000450 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0
0000460 \0 020 \0 \0 \0 \0 \0 \0 005 \0 \0 \0 P 001 \0 \0
0000470 001 \0 \0 \0 C O R E \0 \0 \0 \0 \0 \0 \0
0000480 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000490 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 177 8 \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.

binaire informé

- L'option « -g » de gcc permet d'adjoindre au binaire des informations entre les instructions machine et le code source.
- Lors de la compilation, le compilateur met en place des optimisations qui peuvent rendre cette association difficile.
- L'option -O (lettre o) permet de contrôler les optimisations du compilateur :
 - -O0 ⇒ pas d'optimisations
 - -O1 ⇒ premier niveau
 - -O2 ⇒ deuxième niveau
 - ...
- debug ⇒ -O0

gdb

- Si le binaire contient des symboles de debug alors on peut :
 - lister le code (list)
 - placer un arrêt d'exécution sur une ligne spécifique (break)
 - afficher le contenu d'une variable (print ou display)
 - avancer d'une ligne de code (next ou step)

gdb

- Quelques commandes avancées :
 - unroll : sortie de boucle
 - finish : sortie de fonction
 - cond : conditionne l'activation d'un point d'arrêt
 - watch (surveiller le changement de contenu d'une case mémoire)
 - disa / ena / del : gestion des points d'arrêt.
 - up / down : déplacement dans la pile

gdb : cas d'école !

- mon programme fait un SEGFAULT
- 1 \Rightarrow j'exécute (dans gdb ou bien core)
- 2 \Rightarrow j'identifie la cause directe du problème :

```
Reading symbols from a.out...done.
(gdb) r
Starting program: /tmp/a.out

Program received signal SIGSEGV, Segmentation fault.
__strcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S:682
682  ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S: No such file or directory.
(gdb) bt
#0  __strcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S:682
#1  0x0000000004006cc in main (argc=1, argv=0x7fffffffe058) at s.c:13
(gdb) up
#1  0x0000000004006cc in main (argc=1, argv=0x7fffffffe058) at s.c:13
13      strcpy(p,argv[i]);
(gdb) p p
$1 = 0x0
(gdb) p i
$2 = 0
(gdb) p argv[i]
$3 = 0x7fffffffe38d "/tmp/a.out"
(gdb)
```

gdb : cas d'école !

- je remonte la pile jusqu'à mon code
- je trouve la valeur qui pose problème (ici p)
- je liste le code :

```
(gdb) l
8      l+=1;
9      s=malloc(sizeof(char)*l);
10     if (s=NULL) return EXIT_FAILURE;
11     p=s;
12     for(i=0;i<argc;++i) {
13         strcpy(p,argv[i]);
14         p+=strlen(argv[i]);
15     }
16     puts(s);
17     return EXIT_SUCCESS;
```

- La valeur de p est « fixée » ligne 11. Je vérifie cela :
- « break 11 »
- « run »

gdb : cas d'école !

Starting program: /tmp/a.out

Breakpoint 1, main (argc=1,
argv=0x7ffffffe058) at s.c:11

```
11  p=s;
```

```
(gdb) p s
```

```
$4 = 0x0
```

```
(gdb) l
```

```
6   int i,l=0;
```

```
7   for(i=0;i<argc;++i) l+=strlen(argv[i]);
```

```
8   l+=1;
```

```
9   s=malloc(sizeof(char)*l);
```

```
10  if (s=NULL) return EXIT_FAILURE;
```

```
11  p=s;
```

```
12  for(i=0;i<argc;++i) {
```

```
13      strcpy(p,argv[i]);
```

```
14      p+=strlen(argv[i]);
```

```
15  }
```

```
(gdb) b 9
```

Breakpoint 2 at 0x40066e: file s.c, line 9.

- apparemment la valeur de s est déjà à 0
- s est initialisée ligne 9
- « b 9 »
- « r »

gdb : cas d'école !

```
Breakpoint 2, main (argc=1,
argv=0x7ffffffe058) at s.c:9
9      s=malloc(sizeof(char)*l);
(gdb) next
10  if (s=NULL) return EXIT_FAILURE;
(gdb) p s
$5 = 0x602010 ""
(gdb) display s
1: s = 0x602010 ""
(gdb) n
```

```
Breakpoint 1, main (argc=1,
argv=0x7ffffffe058) at s.c:11
11     p=s;
1: s = 0x0
(gdb)
```

- je passe la ligne avec « next » puis je contrôle la valeur de s
- apparemment s change de valeur (de 0x602010 à 0x0)
- je fais un display puis j'exécute pas à pas.
- après la ligne 10, la valeur a changée...

gdb : autres

- Il existe d'autre interface pour gdb :
 - xxgdb
 - ddd
 - kdbg
 - etc...
- La plus part des environnements de développement propose un debugger intégré :
 - visual studio
 - qtcreator
 - kdevelop
 - etc...

tracer les accès mémoires

- valgrind [vælgrind] est un outil qui intègre de nombreux tests pour l'exécution de programmes.
- Permet de détecter des erreurs d'exécution qui ne sont pas relevées par le système comme le dépassement d'un tableau par exemple.
- L'exécution du programme par valgrind est contrôlée pas à pas, ce qui ralentit beaucoup le programme.

valgrind

- Pour pouvoir fonctionner efficacement avec valgrind, le programme doit avoir été compilé avec -g et -O0
- ensuite, on lance le programme dans valgrind :
`valgrind ./a.out arg1 arg2 ...`
- valgrind va afficher les erreurs mémoires au fur et à mesure de leur apparition
- Il est possible de demander à valgrind de lancer gdb à chaque erreur :
`valgrind --vgdb-error=1 ./a.out`

exemple :

```
allali@hebus:/tmp$ valgrind --vgdb-error=1 ./a.out
==9539== Memcheck, a memory error detector
==9539== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9539== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9539== Command: ./a.out
==9539==
==9539== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==9539== /path/to/gdb ./a.out
==9539== and then give GDB the following command
==9539== target remote | /usr/lib/valgrind/../../bin/vgdb --pid=9539
==9539== --pid is optional if only one valgrind process is running
==9539==
==9539== Invalid write of size 4
==9539== at 0x400570: main (s.c:7)
==9539== Address 0x51fc050 is 0 bytes after a block of size 16 alloc'd
==9539== at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck.so)
==9539== by 0x40054E: main (s.c:4)
==9539==
==9539== (action on error) vgdb me ...
```

```
allali@hebus:/tmp$ gdb a.out
...
Reading symbols from a.out...done.
(gdb) target remote | /usr/lib/valgrind/../../bin/vgdb --pid=9539
...
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x0000000000400570 in main (argc=1, argv=0xfffff98) at s.c:7
7         t[i]=0;
(gdb) p t
$1 = (int *) 0x51fc040
(gdb) p i
$2 = 4
```

Toute erreur signalée par valgrind mérite une analyse et, à de très rares exceptions, doit être corrigée !

```
1 #include<stdlib.h>

3 int main(int argc, char **argv) {
4     int *t=malloc(sizeof(int)*4);
5     int i;
6     for(i=0;i<6;++i)
7         t[i]=0;
8     return 0;
9 }
```

valgrind : fuite mémoire

- A la fin de l'exécution, valgrind liste les blocs non libérés et les classe selon :
 - que plus rien ne pointe sur ce segment (definitively lost)
 - qu'un autre segment de mémoire pointe encore dessus (indirectly lost)
 - encore accessible par une variable de pile (still reachable)

valgrind : definitively lost

```
allali@hebus:/tmp$ valgrind --leak-check=full ./a.out
==9900== Memcheck, a memory error detector
==9900== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9900== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9900== Command: ./a.out
==9900==
==9900==
==9900== HEAP SUMMARY:
==9900==   in use at exit: 1 bytes in 1 blocks
==9900== total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==9900==
==9900== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9900==   at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9900==   by 0x400543: main (s.c:4)
==9900==
==9900== LEAK SUMMARY:
==9900==   definitely lost: 1 bytes in 1 blocks
==9900==   indirectly lost: 0 bytes in 0 blocks
==9900==   possibly lost: 0 bytes in 0 blocks
==9900==   still reachable: 0 bytes in 0 blocks
==9900==     suppressed: 0 bytes in 0 blocks
==9900==
==9900== For counts of detected and suppressed errors, rerun with: -v
==9900== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
#include<stdlib.h>

int main(){
    malloc(sizeof(char));
}
```


valgrind : indirectly lost

```
allali@hebus:/tmp$ valgrind --leak-check=full --show-reachable=yes ./a.out
==9973== Memcheck, a memory error detector
==9973== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9973== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9973== Command: ./a.out
==9973==
==9973==
==9973== HEAP SUMMARY:
==9973==   in use at exit: 9 bytes in 2 blocks
==9973== total heap usage: 2 allocs, 0 frees, 9 bytes allocated
==9973==
==9973== 1 bytes in 1 blocks are indirectly lost in loss record 1 of 2
==9973==   at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9973==   by 0x400555: main (s.c:5)
==9973==
==9973== 9 (8 direct, 1 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==9973==   at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9973==   by 0x400547: main (s.c:4)
==9973==
==9973== LEAK SUMMARY:
==9973==   definitely lost: 8 bytes in 1 blocks
==9973==   indirectly lost: 1 bytes in 1 blocks
==9973==   possibly lost: 0 bytes in 0 blocks
==9973==   still reachable: 0 bytes in 0 blocks
==9973==   suppressed: 0 bytes in 0 blocks
==9973==
==9973== For counts of detected and suppressed errors, rerun with: -v
==9973== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

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

```
int main(){
    char **p=malloc(sizeof(char *));
    p[0]=malloc(sizeof(char ));
}
```

valgrind : still reachable

```
allali@hebus:/tmp$ valgrind --leak-check=full --show-reachable=yes ./a.out
==9996== Memcheck, a memory error detector
==9996== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9996== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9996== Command: ./a.out
==9996==
==9996==
==9996== HEAP SUMMARY:
==9996==   in use at exit: 1 bytes in 1 blocks
==9996== total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==9996==
==9996== 1 bytes in 1 blocks are still reachable in loss record 1 of 1
==9996==   at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9996==   by 0x400543: main (s.c:5)
==9996==
==9996== LEAK SUMMARY:
==9996==   definitely lost: 0 bytes in 0 blocks
==9996==   indirectly lost: 0 bytes in 0 blocks
==9996==   possibly lost: 0 bytes in 0 blocks
==9996==   still reachable: 1 bytes in 1 blocks
==9996==   suppressed: 0 bytes in 0 blocks
==9996==
==9996== For counts of detected and suppressed errors, rerun with: -v
==9996== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
#include<stdlib.h>

char *p;
int main(){
    p=malloc(sizeof(char ));
}
```

valgrind

- options :
 - `--leak-check=full --show-reachable=yes`
 - `--malloc-fill=<hexnumber>`
 - `--free-fill=<hexnumber>`
 - ...
- Attention : un programme n'ayant pas de problème avec valgrind n'est pas nécessairement un programme sans bug !

Convention de codage

- Une convention de codage est un document qui liste les règles d'écriture de code source pour un projet / une entreprise :
 - nommage des fonctions, variables, macro...
 - nommage et organisation de fichiers
 - langue pour le code et les commentaires
 - formatage spécifique (boucle, tests...)
 - techniques de programmation
- Une convention est un document vivant qui doit être mis à jour si nécessaire.

convention de codage : exemple

- <https://www.kernel.org/doc/Documentation/CodingStyle>
 - indentation
 - taille max de lignes
 - positionnement des accolades et parenthèses
 - espaces
 - nommage : C is a Spartan language, and so should your naming be....
 - typedef
 - fonctions
 - sortie de fonctions
 - commentaires
 -

Documentation

- La documentation est primordiale pour un projet sur le long terme
- Plusieurs niveaux de doc :
 - très haut niveau : présentation large, vue d'ensemble, éléments d'architecture
 - modules : aspects fonctionnels, périmètre
 - fonction : description des paramètres, valeurs de retour, spécification
 - code : astuces mises en œuvre, point d'algorithmique non trivial, justification de choix.

Documentation

- Le maintien d'une documentation peut être un travail long et il arrive souvent qu'il y a divergence entre la documentation et le code.
- Pour cela, on rapproche la documentation du code en l'incluant dans celui-ci
- Utilisation des commentaires et de générateurs de documentation
- Ne permet de faire toute la documentation (en particulier, la doc de haut niveau, manuel d'utilisation...).

doxygen

- **doxygen** permet de générer la documentation au format html/pdf/latex... à partir des commentaires dans le code source.
- doxygen peut également intégrer de la documentation au format **Markdown**
- Le résultat est une documentation séparée des sources mais synchronisée avec celles-ci.

doxygen

- doxygen utilise des commentaires suivant certaines règles d'écriture :

```
//! power function
```

```
/*! The pow() function returns the value of x raised to the power of y.
```

```
* \param x a real in double format
```

```
* \param y a real in double format
```

```
* \return x raised to power of y or NaN if wrong arguments
```

```
*/
```

```
double pow(double x, double y) ;
```

- doxygen gère d'autres formats (javadoc par exemple).

pow.c File Reference

#include "pow.h"

Include dependency graph for pow.c:



[Go to the source code of this file.](#)

Functions

double [pow](#) (double x, double y)

Function Documentation

```
double pow ( double x,  
             double y  
             )
```

power function The [pow\(\)](#) function returns the value of x raised to the power of y.

Parameters

- x** a real in double format
- y** a real in double format

Returns

x raised to power of y or NaN if wrong arguments

Definition at line [5](#) of file [pow.c](#).

doxygen et README.md

- Il est souhaitable pour un projet d'avoir un fichier qui donne des informations générales de haut niveau :
 - auteurs
 - objectif de l'application
 - pré-requis
 - installation
 - utilisation
 - ...
- Une technique consiste à intégrer ces éléments dans un fichier README.md à la racine du projet. Ce fichier pourra être intégré à la documentation par doxygen :

```
My great project      {#mainpage}  
=====
```

```
About  
-----
```

My Project

Main Page

Files

▼ My Project



My great project

► Files

My great project

Author

Julien Allali

About

example for PG106 course.

Commentaires : qq règles

- Les commentaires doivent être **utiles**
- Pour une fonction :
 - ce que fait la fonction (spécification)
 - éventuellement, comment elle le fait (algo, complexité, coût mémoire...)
 - domaine de valeur des paramètres
 - cas d'erreurs
- Les commentaires dans le code doivent servir à suivre la logique de celui-ci, par ex :
 - `// set default value into the matrix`
 - ...
 - `// fill the matrix according to the formula : $M[i][j] = \min(M[i-1][j], M[i][j-1])$`
 - ...
 - `// backtrace to compute the alignment`
 - ...

Commentaires : qq règles

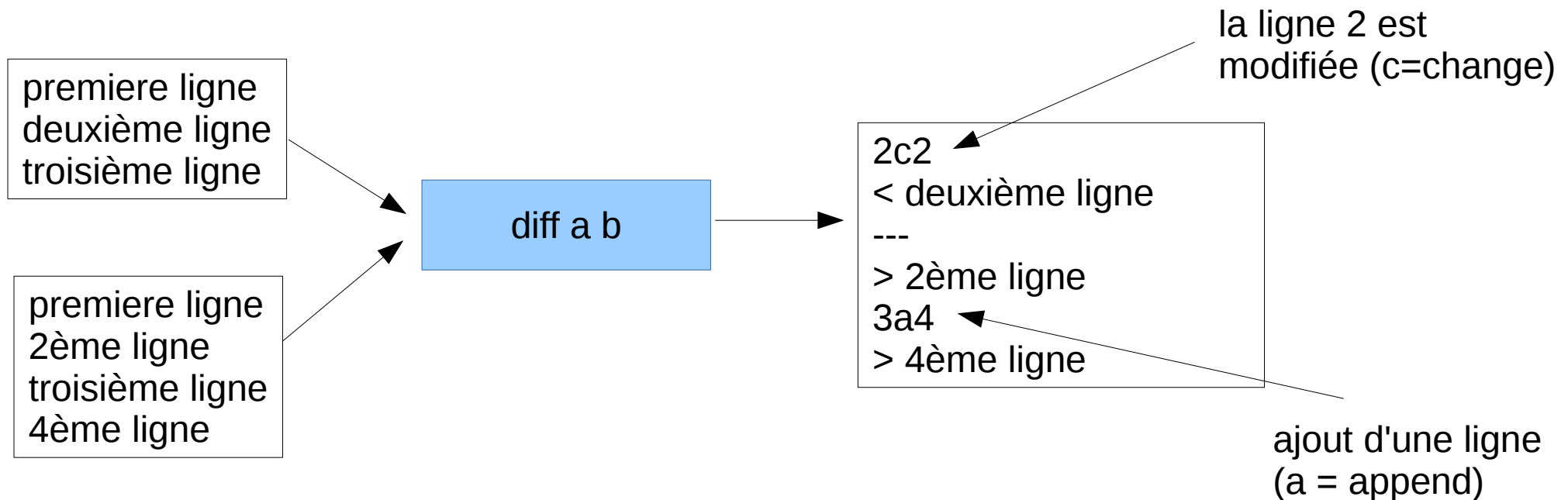
- Pour les modules, penser à ajouter une description générale de ce que fait le module, avec un code d'exemple d'utilisation.
- Au début des fichiers d'implémentation, un entête spécifie :
 - les auteurs,
 - la licence, (copyright si rien)
 - une liste datée des modifications

gestion de sources

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs...) :

diff

- **diff** est un outil d'analyse de texte qui compare deux fichiers entre eux et produit le nombre minimum d'édition à faire sur le premier fichier pour obtenir le second :



diff : side by side

- on peut afficher les deux fichiers cote à cote :

```
diff -y a b
```

```
premiere ligne  
deuxième ligne  
troisième ligne
```

```
premiere ligne  
| 2ème ligne  
troisième ligne  
>4ème ligne
```

- diff permet la comparaison récursive de deux arborescences.

diff : recursif

- Je souhaite modifier le code source d'un projet :
 1. je fais une copie de sauvegarde des sources d'origine
 2. j'effectue mes modifications
 3. à tout moment, je visualise mes modifications avec
`diff : diff -r projet projet_new`

diff : recursif

- Je souhaite modifier le code source d'un projet :
 1. je fais une copie de sauvegarde des sources d'origine
 2. j'effectue mes modifications
 3. à tout moment, je visualise mes modifications avec
`diff : diff -r projet projet_new`

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLib_new/TaskParser.cpp
15c15
<     throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>     throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLib_new/TaskParser.hpp
36a37,38
>  /*! TaskParser: build a task from a json description.
>  */
```

Ajout d'un commentaire

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

MOTELS ARE NOT HELL!

MIROIR TU ES LA!

- Sur cet exemple « MO T » est une sous séquence commune.

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

MOTELS_ARE_NOT_HELL!

MIROIR_TU_ES_LA!

MORTEL!

diff et communication

- Ainsi, si l'on souhaite communiquer une modification, il suffit d'envoyer le résultat d'un diff récursif : `diff -rupN original new > patch`
- On appelle ce fichier un patch.
- Le destinataire peut lire ce fichier et comprendre vos modifications
- Il peut également appliquer ces modifications en local grâce au programme `patch`
- *les options upN sont nécessaires au fonctionnement de patch, elles ajoutent des informations de contexte (u), de fonction (p), d'ajout de fichier (N)*

patch

- le programme patch permet d'appliquer les modifications identifiées par diff.
- Ainsi sur l'exemple précédent je peux faire :

```
$ cp -R GenTaskLib GenTaskLibMod
```

```
$ cd GenTaskLibMod
```

```
$ patch < ../patch
```

```
patching file TaskParser.cpp
```

```
patching file TaskParser.hpp
```

```
$ cd .. ; diff -r GenTaskLib GenTaskLibMod
```

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLibMod/TaskParser.cpp
15c15
<     throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>     throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLibMod/TaskParser.hpp
36a37,38
>  /* TaskParser: build a task from a json description.
>  */
```

diff & patch

- Dans le monde open source, diff et patch sont extrêmement utilisés

Bugzilla@Mozilla

New Account | Log In | Forgot Password

Home New Browse Search [help] Reports Product Dashboard

Bug List: (1 of 307) First Last Prev Next Show last search results

Bug 328174 - ISP files: can't preselect server type choice [Last Comment](#)

Status: REOPENED
Whiteboard:
Keywords: fixed1.8.1
Product: Thunderbird ([show info](#))
Component: Account Manager ([show other bugs](#)) ([show info](#))
Version: unspecified
Platform: All All
Importance: -- normal (vote)
Target Milestone: ---
Assigned To: Yann Rouillard
QA Contact:
Mentors:
URL:
Depends on:
Blocks: [Show dependency tree / graph](#)

Reported: 2006-02-22 02:29 PST by Yann Rouillard
Modified: 2009-11-11 19:21 PST ([History](#))
CC List: 4 users ([show](#))
See Also:
Crash Signature:
Project Flags:
Tracking Flags:

Attachments

ISP example file to test the bug. (2.25 KB, application/rdf+xml)	no flags	Details
Patch to preselect imap in server page if server type was set to imap in isp rdf file (11 KB, patch)	mozilla: review-	Details Diff Review
cumulative patch (2.73 KB, patch)	mozilla: review+ mscott: superreview+ mscott: approval-branch-1.8.1+	Details Diff Review
Allow isp rdf to force use of incoming username for outgoing server (4.65 KB, patch)	mozilla: review+ mkmelin+mozilla: superreview-	Details Diff Review
rdf file to test smtpUseIncomingUsername (2.32 KB, application/rdf+xml)	no flags	Details
Add an attachment (proposed patch, testcase, etc.)		Show Obsolete (1) View All

liste de 3 patchs correctifs

4 juin 1996

- Vous êtes ingénieur dans à l'agence spatiale européenne.
- Depuis 9 ans, vous travaillez sur les différents modules embarqués dans la fusée Ariane 5
- Aujourd'hui est le grand jour de lancement de la fusée :
 - Kourou / Guyane
 - 370 Millions de \$ d'investissement

Les gestionnaires centralisés

- Historiquement, cvs (concurrent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
 - checkout, update, infos, status, commit, diff, revert

svn

- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cette méthodologie dans tout projet de développement (open source ou entreprise).

svn : méthodologie

- Il est possible d'utiliser SVN juste pour le répertoire de développement principal.
- Seulement, comment faire si on a un « gros » développement à produire: si on transmet les modifications intermédiaires, le programme devient instable (ne compile plus par exemple...)
- Comment faire également pour gérer les versions:
 - On sort une version 1.0 qui évolue en 1.1 puis 1.2
 - On sort la version 2.0 (« casse » la compatibilité avec 1.x)
 - Un bug est trouvé dans la version 2.0: il faut le corriger dans la version courante mais également dans la version 1.x!

svn : méthodologie

- Aussi, il est nécessaire de maintenir plusieurs « répertoires » de développement parallèle.
- Une méthodologie classique organise les sources ainsi:
 - / « racine »
 - /trunk : contient la version en cours des sources (dev.)
 - /branches/: des copies de trunk (« svn copy »)
 - /branches/1.0 : copie de trunk pour release (tests), retour de modif dans le trunk avec « svn merge » si compatible
 - /tags/1.0.0: version **figée** d'une branche, sert de référence, est diffusée. La branche correspondante est étiquetée (tag). **pas de commit/modifs dans ce répertoire**
 - /branches/modif_allali_155/: copie temporaire pour de « grosses » modification avec retour dans le trunk par « merge ». Synchronisation régulière depuis le trunk (« merge »)

svn : la création de dépôt

- Le création d'un dépôt se fait à l'aide de la commande « `svnadmin create nom_de_depot` »
- possibilité d'ajouter l'exécution de script avant/pendant et après les « commits »
- possibilité d'envoi de mails lors des commits
- facile à mettre en place sur son compte:
- `cd ~/.depots/ ; svnadmin create SVN`
- `cd ~/; svn co file:/// $HOME/.depots/SVN`
- via ssh:
- `svn co svn+ssh://allali@ssh.enseirb.fr/.depots/SVN`

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt centrale.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modification (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
 - clone, add, commit, push, pull, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt centrale).
- On peut transmettre un ensemble de commit à un autre utilisateur avec la commande push
- On peut réceptionner un ensemble de commit depuis un autre utilisateur avec la commande pull.

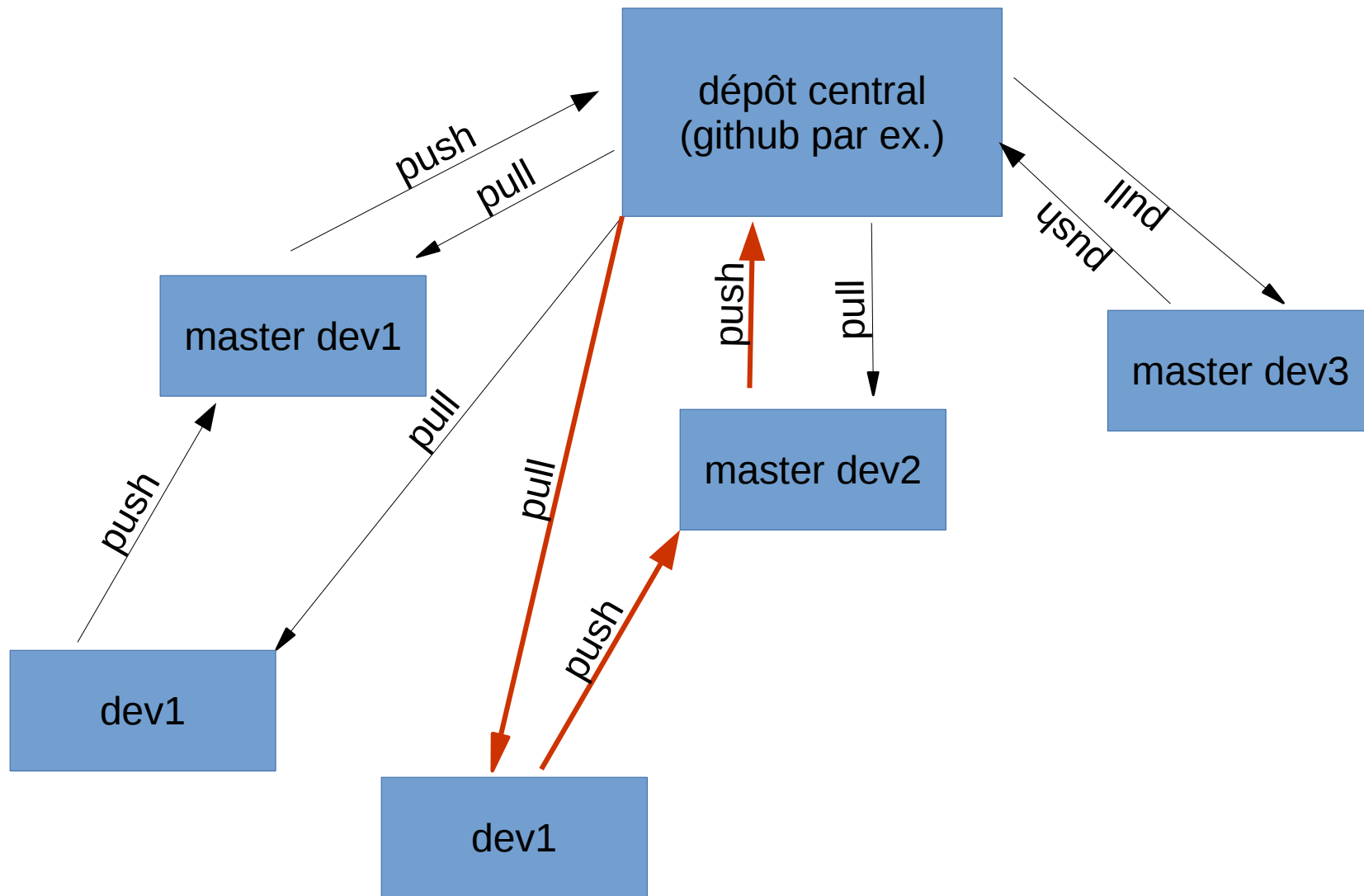
git

- git intègre une gestion de branches :
 - création :
 - git branch b2
 - git checkout b2
 - ou bien git checkout -b b2
 - la fusion :
 - on bascule dans la branche qui doit recevoir les modifs
 - git checkout master ; git merge b2
 - Les modifications doivent avoir été commité dans b2
 - la déléction : git branch -d b2
- La branche par défaut s'appelle **master**

re-centralisation

- L'avantage d'être en décentraliser et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plus part de projet, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github par exemple).
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits ».

git



contrôle dans svn

- Il est aussi possible d'avoir cette approche hiérarchique dans svn en subdivisant le répertoire « branches » en répertoire et en ajustant les droits :
 - seuls les masters peuvent commiter dans « trunk »
 - les dev travaillent dans des branches
- Différences majeures entre svn et git est :
 - centralisé / dé-centralisé
 - support natif du système de branches dans git
 - commit locaux dans git

Les autres gestionnaires

	Open Source	Centralisé	Décentralisé
CVS	•	•	
SVN	•	•	
GIT	•		•
SourceSafe		•	
Mercurial	•		•
Bazaar	•		•
BitKeeper			•
Team Foundation Server		•	

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



Makefile

- make est un outil qui permet d'automatiser la création et la mise à jour de fichiers.
- make repose sur un fichier de description : **Makefile**
- Un Makefile est un ensemble de règles formatées :

```
cible : source1 source2 source3 ...
```

```
    commande1
```

```
    commande2
```

```
    ...
```

- Si « cible » n'existe pas ou est moins récente que l'une des sources, alors les commandes sont exécutées.

Makefile : exemple

```
image_thumbail.jpg : image.jpg
    convert -size 80x80 image.jpg image_thumbnail.jpg
image.jpg : image.png
    convert image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

- make est récursif : si un fichier source n'existe pas, il cherche une règle pour le créer.
- initialement, make cherche à créer une seule cible : la première du Makefile ou celle(s) indiquée(s) sur la ligne d'appel : `make image.jpg`
- Si une source est manquante et qu'il n'y a pas de règle pour la créer : erreur
- Si une des commandes renvoie une valeur différente de 0, make s'arrête (tips : `commande || true`)

Makefile : variable

- make supporte la déclaration de variable :

`NOM=VALEUR`

- valeur peut comporter des espaces

`NOM - =VALEUR`

- Si NOM est dans l'environnement, alors c'est la valeur de l'environnement qui est utilisée, sinon c'est VALEUR.
- `$(NOM)` est remplacé par VALEUR.

Makefile : variable, exemple

```
CONVERT=convert
THUMB_SIZE-=80x80
image_thumbail.jpg : image.jpg
    $(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
image.jpg : image.png
    $(CONVERT) image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

\$ THUMB_SIZE=60x60 make

Makefile : spéciales

- Quelques variables spéciales pour l'écriture des commandes :

`cible : source1 source2`

- `$@` : cible
- `$<` : source1
- `$$` : source1 source2

- **Ainsi :**

```
image_thumbnail.jpg : image.jpg
$(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
```

- **Devient :**

```
image_thumbnail.jpg : image.jpg
$(CONVERT) -size $(THUMB_SIZE) $< $@
```

Makefile : règles génériques

- Il est possible d'écrire des règles génériques :

```
%.jpg : %.png
```

```
    convert $< $@
```

- Permet de convertir tout fichier png en jpg à l'aide de convert

```
%_thumb.jpg : %.jpg
```

```
    convert -size 80x80 $< $@
```

- Il est enfin possible de séparer la liste des dépendances et la règle de création.

Makefile et compilation

```
CC=gcc
CFLAGS=-Wall

prog : prog.o hash.o
    $(CC) -o $@ $^
prog.o : prog.c hash.h
hash.o : hash.c hash.h
%.o :
    $(CC) $(CFLAGS) $< -o $@
```

- La dernière règle explique comment générer un fichier .o
- les deux règles au dessus donnent les dépendances

Makefile : PHONY

- On peut vouloir écrire des règles qui ne génèrent pas de fichier :

`all`

`install`

`clean`

`distclean`

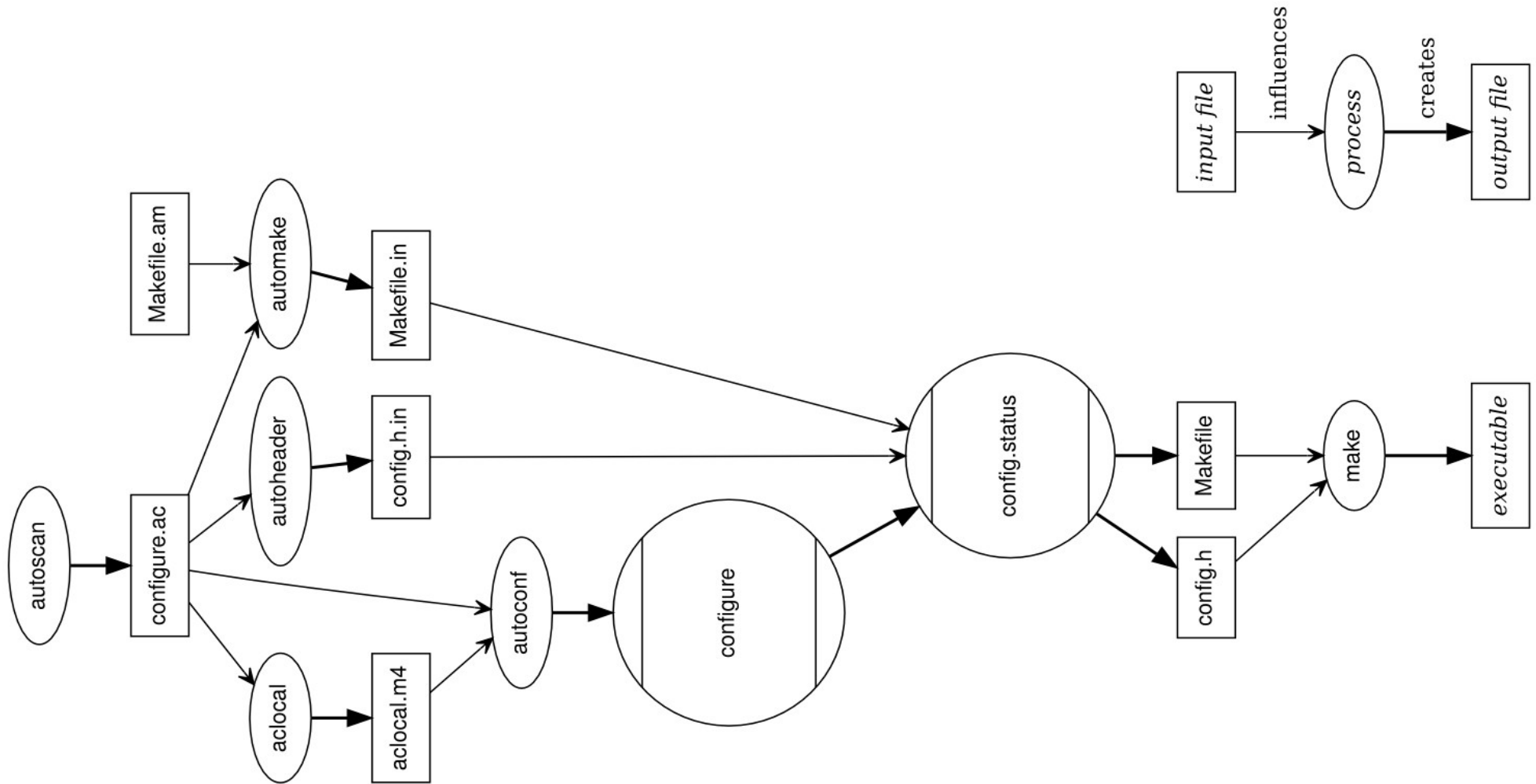
- On indique cela à make :

```
.PHONY=all install clean distclean
```

Makefile, automake...

- Makefile n'est pas spécifique à la compilation de programme
- Il ne gère pas, entre autre, la dépendance entre les fichiers sources (on peut utiliser `gcc -MM source1.c source2.c ...` pour cela)
- La prise en compte de l'environnement (compilateur, options, bibliothèque...) peut ce faire :
 - par l'édition des variables du Makefile (options de compilations, répertoire d'installation...)
 - par l'écriture de plusieurs Makefile (un par système)
 - par l'utilisation d'un générateur :
 - automake / autoconf
 - cmake

auto-tools : automake / autoconf



cmake

- Cmake est un outil simplifié permettant la compilation de sources C et C++.
- C'est un outil multi-plaforme sous licence BSD
- Nécessite la présence d'un fichier CMakeLists.txt
- gestion automatique des dépendances
- peut générer des makefile
- Simple d'utilisation / facile à prendre en main
- Exemple: une projet <<HELLO>> avec une bibliothèque dans le répertoire Hello et un programme d'exemple dans le répertoire Demo

cmake

- `./CMakeLists.txt`:

```
cmake_minimum_required (VERSION 2.6)
project (HELLO)
```

```
add_subdirectory (Hello)
add_subdirectory (Demo)
```

- `./Hello/CMakeLists.txt`

```
add_library (Hello hello.c)
```

- `./Demo/CMakeLists.txt`

```
include_directories (${HELLO_SOURCE_DIR}/Hello)
link_directories (${HELLO_BINARY_DIR}/Hello)
add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo Hello)
```

cmake:cross platform make

- cmake est un système de compilation cross-platforme. Il ne compile pas directement mais génère des fichiers dans différents formats :
 - Makefile
 - projet Visual Studio
 - Borland Makefile
 - projet Xcode
 - Kate
 - ...
- cmake utilise les fichiers CMakeLists.txt et génère des fichiers en fonction de la plate-forme de compilation (Makefile, visual, xcode....).

cmake : les bases

- La déclaration de variables :
 - `set(NAME VALUE)`
 - `${NAME}`
 - lors de l'appel à cmake : `cmake -DNAME=VALUE`
- Variables standards :
 - `CMAKE_INCLUDE_PATH` (pour les .h)
 - `CMAKE_LIBRARY_PATH` (pour la recherche de .so)
 - `DESTDIR` (pour l'installation)
 - `CMAKE_BUILD_TYPE` (Debug, Release)
- Dans le CMakeLists.txt :

– <code>CMAKE_C_FLAGS</code>	– <code>CMAKE_CURRENT_SOURCE_DIR</code>
– <code>CMAKE_C_FLAGS_DEBUG</code>	– <code>CMAKE_CURRENT_BINARY_DIR</code>
– <code>CMAKE_C_FLAGS_RELEASE</code>	– <code>CMAKE_SOURCE_DIR</code>

cmake : les bases (2)

- `add_executable(name sources)`
- `add_library(name STATIC sources)`
- `add_library(name SHARED sources)`
- `target_link_libraries(name libs)`
- `include_directories(dir1 dir2...)`
- `add_custom_command`

cmake : utilisation

- cmake support l'out-source building : c'est à dire la compilation en dehors du répertoire des sources :
- On suppose : projet/CMakeLists.txt
- alors on peut faire :

```
mkdir projet-build
```

```
cmake ../projet
```

```
make
```

- et

```
mkdir projet-debug
```

```
cmake -DCMAKE_BUILD_TYPE=Debug ../projet
```

```
make
```

Question

#QDLE#Q#AB*#20#

- cmake analyse les dépendances :
 - A. en énumérant les fonctions appelées
 - B. en traçant les inclusions

Question

#QDLE#Q#A*BCD#35#

- J'ai une bibliothèque dynamique libtoto.so compilée à partir de toto.c et toto.h. J'ai un programme de démo demo.c qui utilise cette bibliothèque. L'ensemble est compilé. Si je modifie toto.c je dois :
 - A. re-compiler la bibliothèque
 - B. re-compiler l'exécutable
 - C. réponse A & B
 - D. ne rien faire.

L'Intégration Continue

- Pour cela, on dispose d'un ensemble de machines.
 - Solution 1 : avant de transmettre mes modifications, je me connecte sur chacune des machines et je testes.
 - Solution 2 : j'utilise un système qui fait cela automatiquement pour moi !
⇒ C'est ce que l'on appelle l'Intégration Continue.
 - l'IC *garantie* une stabilité des développements au fur et à mesure. Cela permet de contrôler certaines dettes techniques.

L'Intégration Continue

- Il existe plusieurs plateformes d'intégration continue :
 - Jenkins (successeur de Hudson, java-open source)
 - TeamCity (JetBrain, commercial)
 - CruiseControl (java-open source)
 - Team Foundation Server (Microsoft, commercial)
 - etc...

L'Intégration Continue

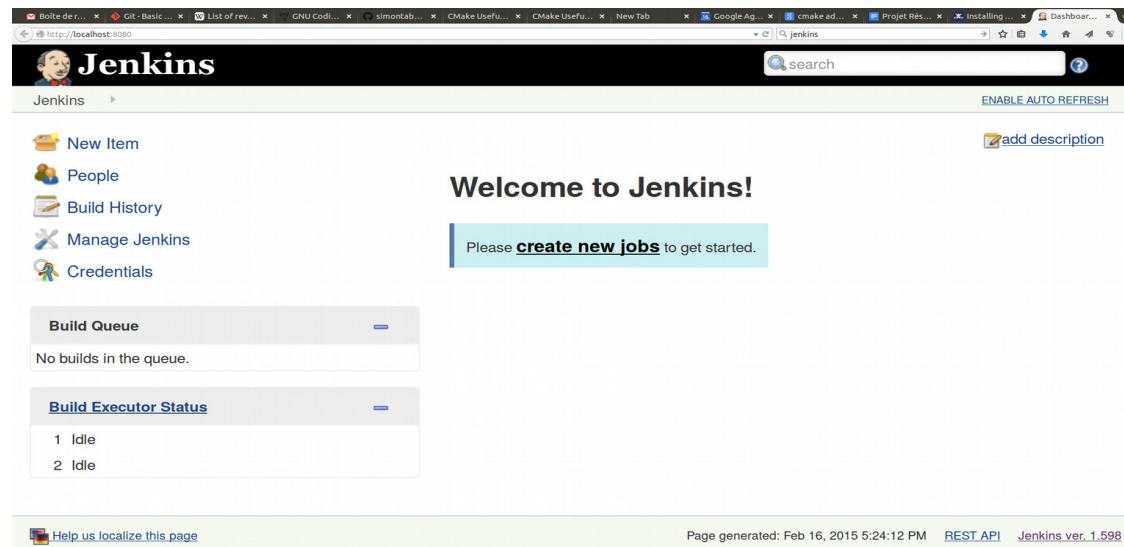
- Un serveur d'intégration continue va se synchroniser avec un dépôt
- A intervalle régulier, il va vérifier que le dépôt est à jour. Si une mise à jour est intervenue, il va effectuer une série de tache (compilation par exemple).
- En fonction du résultat des taches, le serveur va indiquer l'état du projet et possiblement transmettre des alertes.
- Afin de gérer plusieurs environnements, le serveur d'IC va piloter des clients sur lesquels il lancera les taches (via ssh par exemple).

L'Intégration Continue

- La qualité qu'offre l'IC va dépendre principalement de deux facteurs :
 - la nature et la diversité des clients (environnement de validation)
 - la complexité des tâches à réaliser :
 - de la compilation
 - à l'exécution de tâches complexes de validation
- Le serveur d'IC peut également rendre compte de facteur comme les ressources utilisées (cpu, temps, mémoire).

Exemple avec Jenkins

- répertoire projet :
 - projet/ :
 - makefile
 - main.c
- installation de jenkins et connexion au port 8080 sur localhost :



Jenkins : exemple

- création d'un dépôt local avec les sources :
svnadmin create /tmp/projet
checkout + ajout des sources + commit
- Ajout du dépôt dans Jenkins en utilisant comme url *svn+ssh://localhost/tmp/projet/*
- Ajout comme commande de build : make
- Lancement d'un build dans jenkins

Jenkins : statut du projet

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery-... x CMake Useful Variable... x CMake Useful Variable... x Projet [Jenkins] x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/ jenkins subversion "file:///"

Jenkins

Jenkins ▶ Projet ▶ [ENABLE AUTO REFRESH](#)

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

Project Projet

[add description](#)





[Disable Project](#)



[Workspace](#)

[Recent Changes](#)

Build History

[trend](#)

 #4	Feb 16, 2015 5:39 PM
 #3	Feb 16, 2015 5:38 PM
 #2	Feb 16, 2015 5:32 PM
 #1	Feb 16, 2015 5:31 PM

 [RSS for all](#)  [RSS for failures](#)

Permalinks

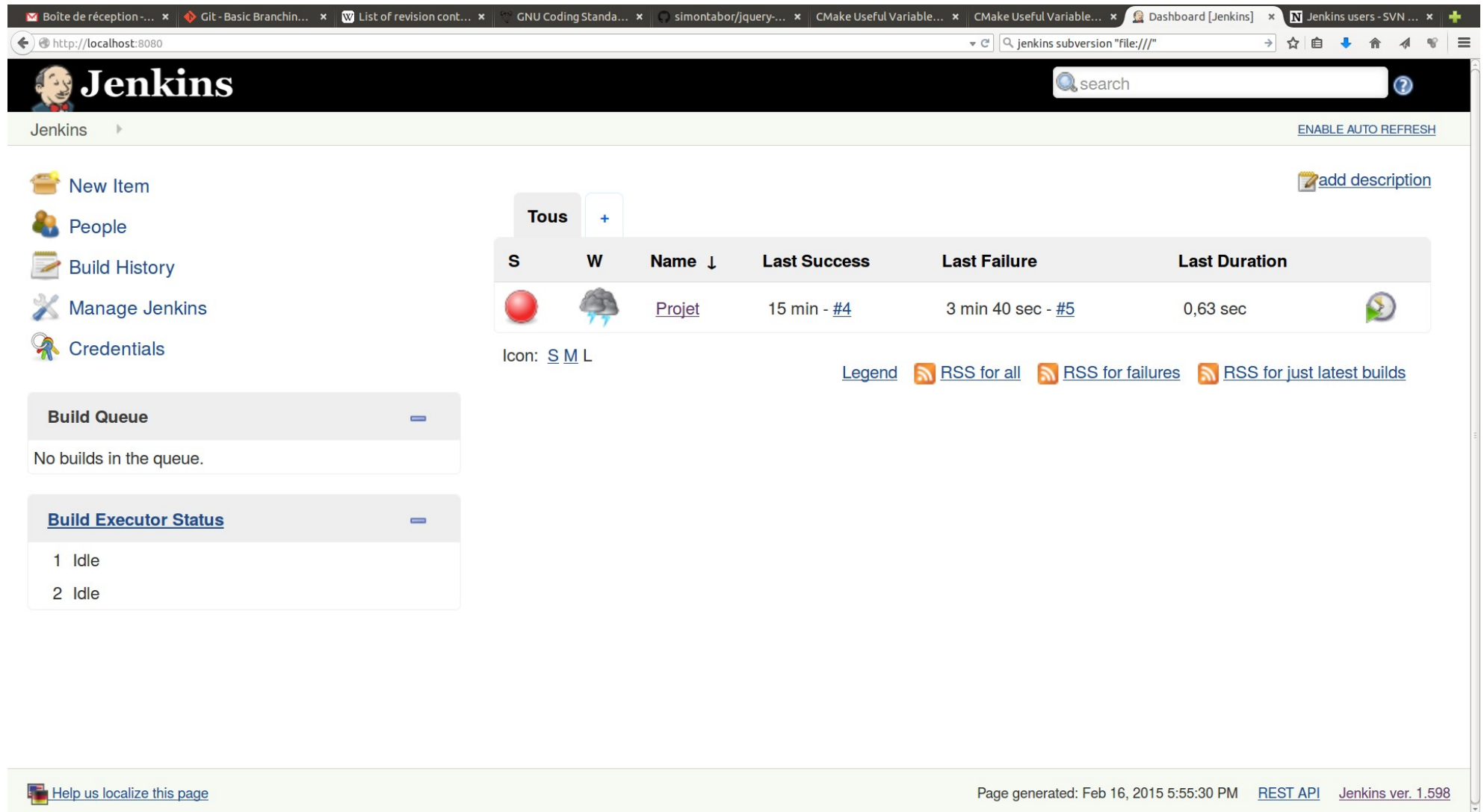
- [Last build \(#4\), 1 min 53 sec ago](#)
- [Last stable build \(#4\), 1 min 53 sec ago](#)
- [Last successful build \(#4\), 1 min 53 sec ago](#)
- [Last failed build \(#3\), 3 min 43 sec ago](#)
- [Last unsuccessful build \(#3\), 3 min 43 sec ago](#)

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:41:46 PM [REST API](#) [Jenkins ver. 1.598](#)

Jenkins : commit

- Ajout d'un bug, commit dans le dépôt



The screenshot shows the Jenkins web interface. The top navigation bar includes the Jenkins logo, a search bar, and a link to 'ENABLE AUTO REFRESH'. The left sidebar contains links for 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main content area displays a 'Build Queue' section with the message 'No builds in the queue.' and a 'Build Executor Status' section showing two idle executors. A table of build history is visible, with columns for status (S, W), name, last success, last failure, and last duration. The table shows a build named 'Projet' that failed. Below the table, there are links for 'Icon: S M L' and 'Legend' with RSS feeds for all builds, failures, and latest builds.

Jenkins

ENABLE AUTO REFRESH




[add description](#)

Build Queue




No builds in the queue.

Build Executor Status

1 Idle
2 Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Projet	15 min - #4	3 min 40 sec - #5	0,63 sec 

Icon: [S](#) [M](#) [L](#)

[Legend](#)  [RSS for all](#)  [RSS for failures](#)  [RSS for just latest builds](#)

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:55:30 PM [REST API](#) Jenkins ver. 1.598

Jenkins : bug

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery... x CMake Useful Variable... x CMake Useful Variable... x Projet #5 [Jenkins] x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/lastFailedBuild/ jenkins subversion "file:///"

Jenkins search

Jenkins ▶ Projet ▶ #5 [ENABLE AUTO REFRESH](#)

[Back to Project](#)

[Status](#)

[Changes](#)


[Console Output](#)

[Edit Build Information](#)

[Delete Build](#)


[Tag this build](#)

[Previous Build](#)


 **Build #5 (Feb 16, 2015 5:51:49 PM)**

Started 5 min 18 sec ago
Took [1 sec](#)

[add description](#)

 Revision: 2
Changes

1. add a bug ([detail](#))

 Started by anonymous user

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:57:07 PM [REST API](#) [Jenkins ver. 1.598](#)

Jenkins : bug

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery-... x CMake Useful Variable... x CMake Useful Variable... x Projet #5 Console [... x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/lastFailedBuild/console jenkins subversion "file:///"

Jenkins

Jenkins ▶ Projet ▶ #5

- Back to Project
- Status
- Changes
- Console Output**
- View as plain text
- Edit Build Information
- Delete Build
- Tag this build
- Previous Build

Console Output

```
Démarré par l'utilisateur anonymous
Building in workspace /var/lib/jenkins/workspace/Projet
Updating svn+ssh://localhost/tmp/projet at revision '2015-02-16T17:51:49.226 +0100'
U      main.c
At revision 2
[Projet] $ /bin/sh -xe /tmp/hudson9096975002011345535.sh
+ make
gcc -c main.c
main.c: In function 'main':
main.c:3:3: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'return'
    return 0;
    ^
Makefile:4: recipe for target 'main.o' failed
make: *** [main.o] Error 1
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
```

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:57:38 PM [REST API](#) [Jenkins ver. 1.598](#)

Intégration Continue

- Tester que le programme compile sous plusieurs environnements est bien mais cela n'offre que peu de garantie quand à l'état fonctionnel du projet.
- Pour *garantir* une qualité tout au long du développement, il faut ajouter des **tests**

Les Tests



Les Tests

- Il existe de nombreux types de tests
- Les tests ont pour objectifs de valider votre code :
 - au niveau d'une fonction : tests unitaires
 - au niveau d'un module : tests fonctionnels
 - entre plusieurs modules : tests d'intégration
 - au niveau général, applicatif : tests de recette
 - sur des cas hors spécifications : test de robustesse

TDD : test driven development

- La méthodologie TDD repose sur l'écriture d'abord de tests puis de code validant les tests.
- La méthode XP (extreme programming) repose en partie sur TDD.
- TDD repose sur des cycles courts consistant :
 - à écrire un test fonctionnel
 - vérifier que le test plante
 - à écrire un test unitaire
 - vérifier que le test plante
 - écrire le code minimal pour que le test fonctionne
 - vérifier que le test passe

TDD

- A la fin de l'écriture d'un code et lorsque tout les tests passent, on peut vouloir refactoriser votre code (copier/coller, simplification, unification, ...)
- Dans ce cas, on ne touche surtout pas aux tests et on remanie le code jusqu'à ce qu'à nouveau il valide l'ensemble des tests.

TDD par l'exemple : bowling

- Supposons que l'on souhaite écrire un module de calcul de feuille de score de bowling :



bowling

- Le joueur a 10 sets
- Pour chaque set, le joueur lance la boule une ou deux fois:
 - Si le joueur élimine les 10 quilles du premier coup, il fait strike et il ne joue pas de 2ème boule
 - Si le joueur élimine les 10 quilles au deuxième coup, il fait spare
- Le score d'un set fait :
 - le score précédent + la somme des deux lancés si pas de strike ni de spare
 - le score précédent + 10 + le score du premier lancé du prochain set si spare
 - le score précédent + 10 + le score du prochain set si strike.
- Pour le 10ème set, le joueur peut avoir un troisième lancé

bowling

- La spécification client est d'avoir un module BowlingGame avoir les méthodes suivantes
 - void roll(BowlingGame *,int nbPinsDown) : enregistre un nouveau lancé
 - int score(BowlingGame *) : renvoie le score actuel

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
 - struct Game
- puis une structure pour modéliser un set
 - struct Set
- Une partie est composée de 10 sets :

```
struct Game {  
    struct Set sets[10] ;  
}
```
- Il faut connaître le set en cours (ajout de currentSet dans Game)
- Pour un set il faut le score de chaque lancé
- Pour le dernier set, il y a peut-être 3 lancers

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
 - struct Game
 - puis une structure pour modéliser un set
 - struct Set
 - Une partie est composée de 10 sets
- ```
struct Game {
 struct Set s[10];
}
```
- Il faut connaître le set en cours (ajout de currentSet dans Game)
  - Pour un set il faut le score de chaque lancé
  - Pour le dernier set, il y a peut-être 3 lancers

**PAS TDD !**

# bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

# bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

```
#include<assert.h>
#include<stdbool.h>

int main(){
 assert(false && « c'est parti ») ;
}
```

ok, le système de test fonctionne !

```
$ make bow
gcc -Wall bow.c -o bow
$./bow
bow: bow.c:5: main: Assertion `0 && "c'est parti"' failed.
```



# bowling : cas vide

- Commençons par le cas vide

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){
 int i;
 struct BowlingGame *game=bg_init();
 for(i=0;i<20;++i)
 bg_roll(game,i);
 assert(bg_score(game)==0 && "test empty");
 bg_free(game) ;
}

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

```
$ gcc -Wall bow.c
bow.c: In function 'test_empty':
bow.c:6:10: warning: implicit declaration of function 'bg_init' [-Wimplicit-function-declaration]
 struct BowlingGame *game=bg_init();
 ^
bow.c:6:28: warning: initialization makes pointer from integer without a cast
 struct BowlingGame *game=bg_init();
```



Ajout de bowling.h

```
#ifndef BOWLING_H
#define BOWLING_H

struct BowlingGame;

struct BowlingGame *bg_init();
void bg_roll(struct BowlingGame *,int);
int bg_score(struct BowlingGame *);
void bg_free(struct BowlingGame *) ;

#endif
```

# bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```



Ajout de bowling.c



# bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```



Ajout de bowling.c

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{};

struct BowlingGame *bg_init(){
 return NULL;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
 return -1;
}
```

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```