Langage C

ENSTA - TC 1ère année

François Pessaux

U2IS

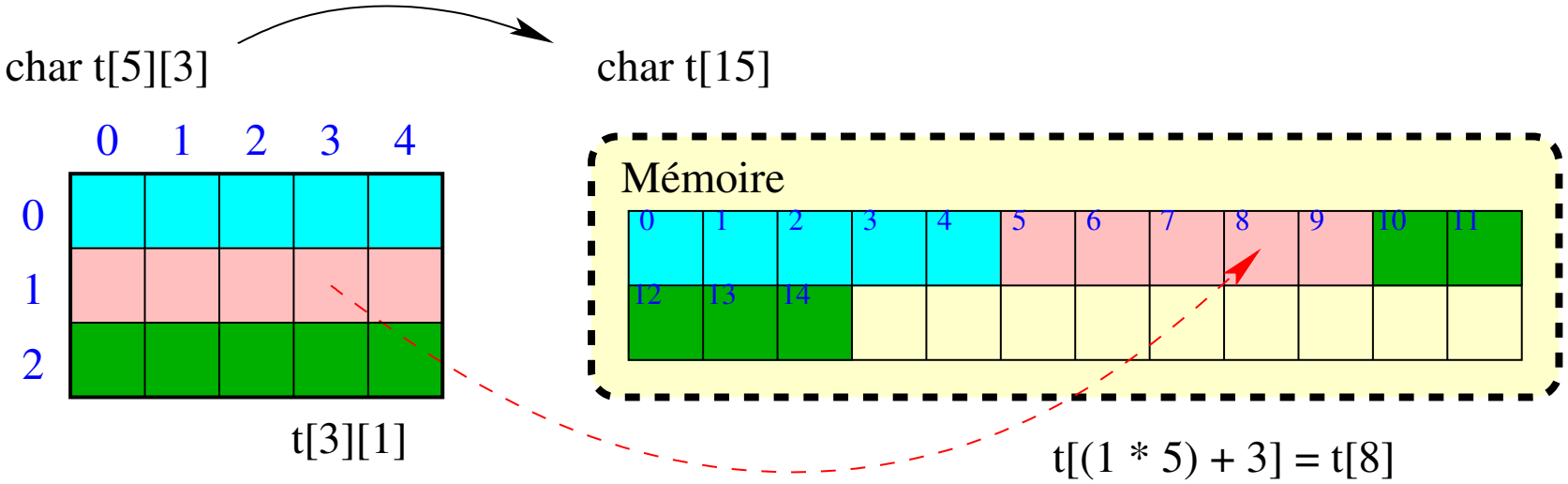
2016-2017

francois.pessaux@ensta-paristech.fr

Tableaux dynamiques à plusieurs dimensions

Les tableaux dynamiques à plusieurs dimensions (1)

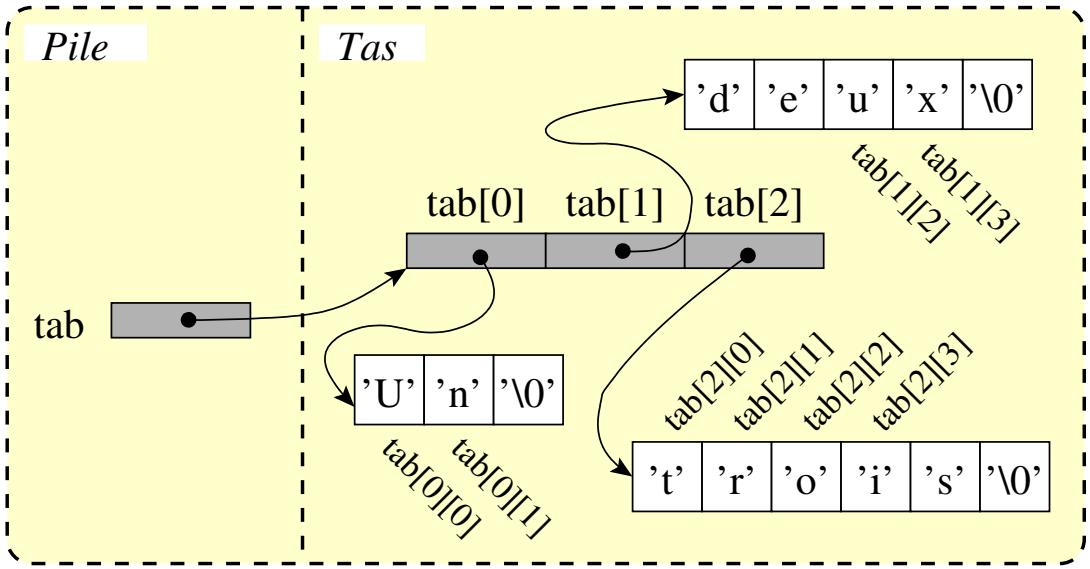
- Vus précédemment : `char t[5][3] ;`
- Comment faire si dimensions non fixes ? → Allocation dynamique.
- Solution 1 :
  - Un grand tableau de taille $DIM_1 * DIM_2 = 5 * 3$.
  - Linéarisation de l'adressage : $t[x][y] \longmapsto t[y * DIM_1 + x]$.



  - Exactement le cas de la mémoire graphique.

Les tableaux dynamiques à plusieurs dimensions (2)

- Comment faire si le tableau n'est pas rectangulaire ?
- ⇒ Tableau de pointeurs.
- Chaque pointeur pointe vers un tableau de taille propre.
- Exemple l'argument `char *argv[]` du `main`.
  - Chaque chaîne peut avoir une longueur différente.



Les tableaux dynamiques à plusieurs dimensions (3)

- Tableau d'`int`s à 2 dimensions, *n*, *m* : tableau de pointeurs sur `int`.
- → `int **t ;`
- On alloue *n* pointeurs (→ 1 allocation) ...
- On alloue *n* fois *m* `int`s (→ *n* allocations).
- Même schéma de libération de mémoire.

```
int i ;
int **t ;

t = malloc (n * sizeof (int*)) ;
for (i = 0; i < n; i++)
  t[i] = malloc (m * sizeof (int)) ;
...
for (i = 0; i < n; i++)
  free (t[i]) ;
free (t) ;
```

(PS : C'est mal : on n'a pas testé le succès des allocations !)

Passage par adresse

Retourner plusieurs valeurs ?

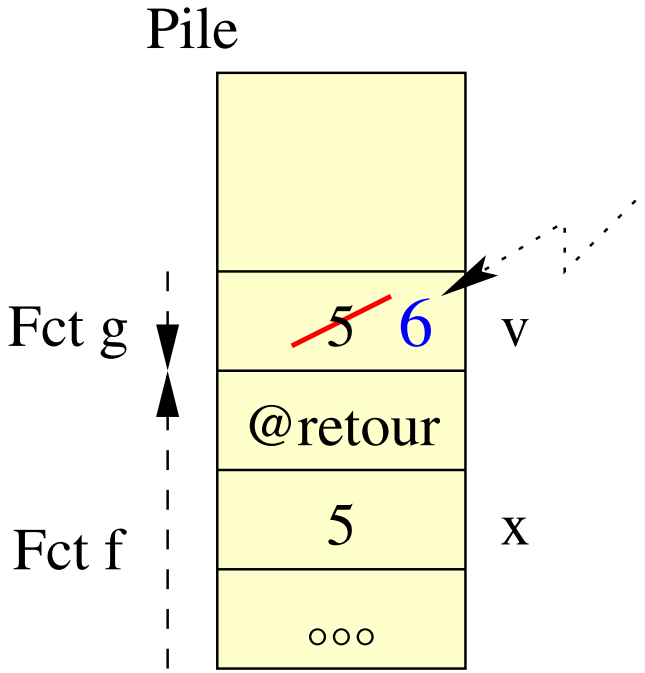
- Parfois une fonction « devrait » fournir plusieurs résultats en sortie.
- Ou un résultat « *composite* ».
- Ex : filtrage d'un point par convolution.
  - Retourne 3 composantes : rouge, vert, bleu.
- En C, `return` ne permet de retourner qu'une valeur.
- Utiliser des variables globales est peu satisfaisant.

Passage par valeur

- En C, paramètres passés par valeur :
  1. Ils sont évalués
  2. Ils sont copiés sur la pile avant l'appel.
- ⇒ Impossible de modifier une variable de l'appelant depuis l'appelé.

```
void g (int v)
{
▶ v++ ;
}

void f ()
{
  int x = 5 ;
  g (x) ;
}
```

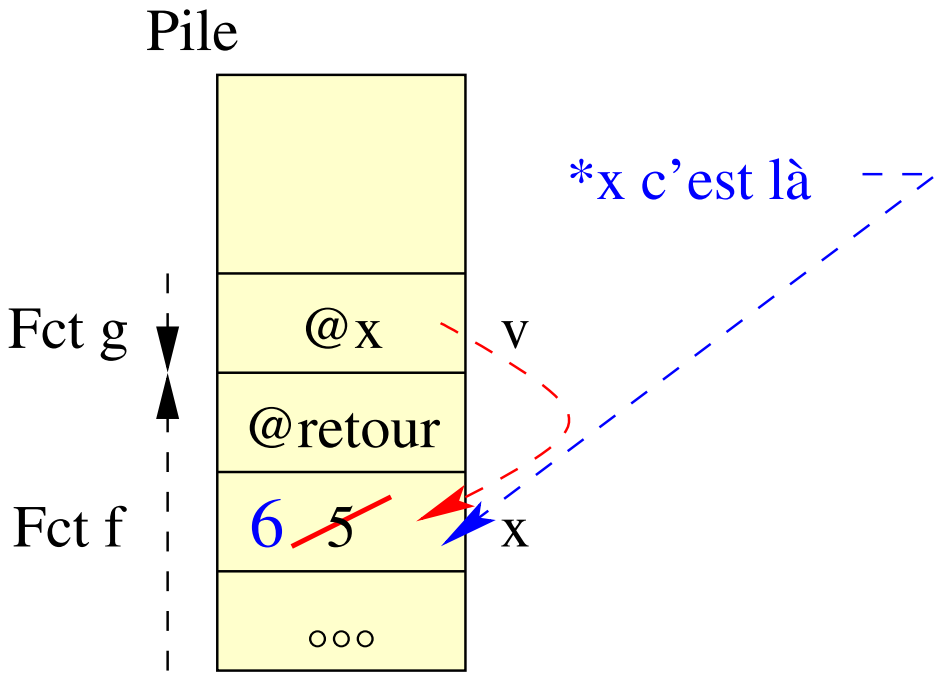


Passage par adresse

- Et si à la place on passait l'adresse de la variable x ?
- C'est l'adresse qui serait copiée.
- Dans g on a donc un pointeur vers x de l'appelant.
- Il suffit de modifier la valeur pointée : `(*v)++`

```
void g (int *v)
{
► (*v)++ ;
}

void f ()
{
  int x = 5 ;
  g (&x) ;
}
```



Exploitation du passage par adresse

- Pour « *retourner* » plusieurs résultats, une fonction n'a qu'à prendre l'adresse de variables où stocker ses résultats.
- Mode de passage de paramètres appelé « *out* » (c.f. `scanf`).

```
int* random_array (int *size) {
  *size = 1 + random () ;
  return (malloc (*size * sizeof (int))) ;
}
```

- La fonction peut aussi utiliser les valeurs initiales de ces arguments avant de les écraser.
- Mode de passage de paramètres appelé « *in/out* ».

```
void scale_point (int *x, int *y, int scale) {
  *x = *x * scale ;
  *y = *y * scale ;
}
```

- Rem : un tableau étant un pointeur sur la 1ère case, il est forcément passé par adresse !

Débugger son programme

Quand ça ne marche VRAIMENT pas . . .

- Votre programme plante sec ou ne donne pas le résultat attendu.
- Trois solutions :
  - Relire votre code et penser très fort
    - Efficace pour les bugs simples.
  - Mettre des `printf` pour tracer ce que fait votre code.
    - Très efficace ! Pour les bugs compliqués.
  - Utiliser un debugger pour inspecter finement l'éxécution.
    - Efficace . . . pour les bugs désespérés.
    - Impeccable si votre programme a fait « *core dumped* » : obtention de l'état du programme au moment du plantage !

Le debugger : gdb

- Nombreuses variantes avec ou sans IHM (`gdb`, `xxgdb`, `cgdb`, ...).
- Nécessite de compiler avec l'option `-g` de `gcc`.
- Permet de :
  - Démarrer un programme.
  - Mettre des points d'arrêt.
  - Avancer instruction par instruction.
  - Surveiller des variables.
  - Et bien d'autres choses encore...
- S'invoque en ligne de commande (avec le nom de l'exécutable).

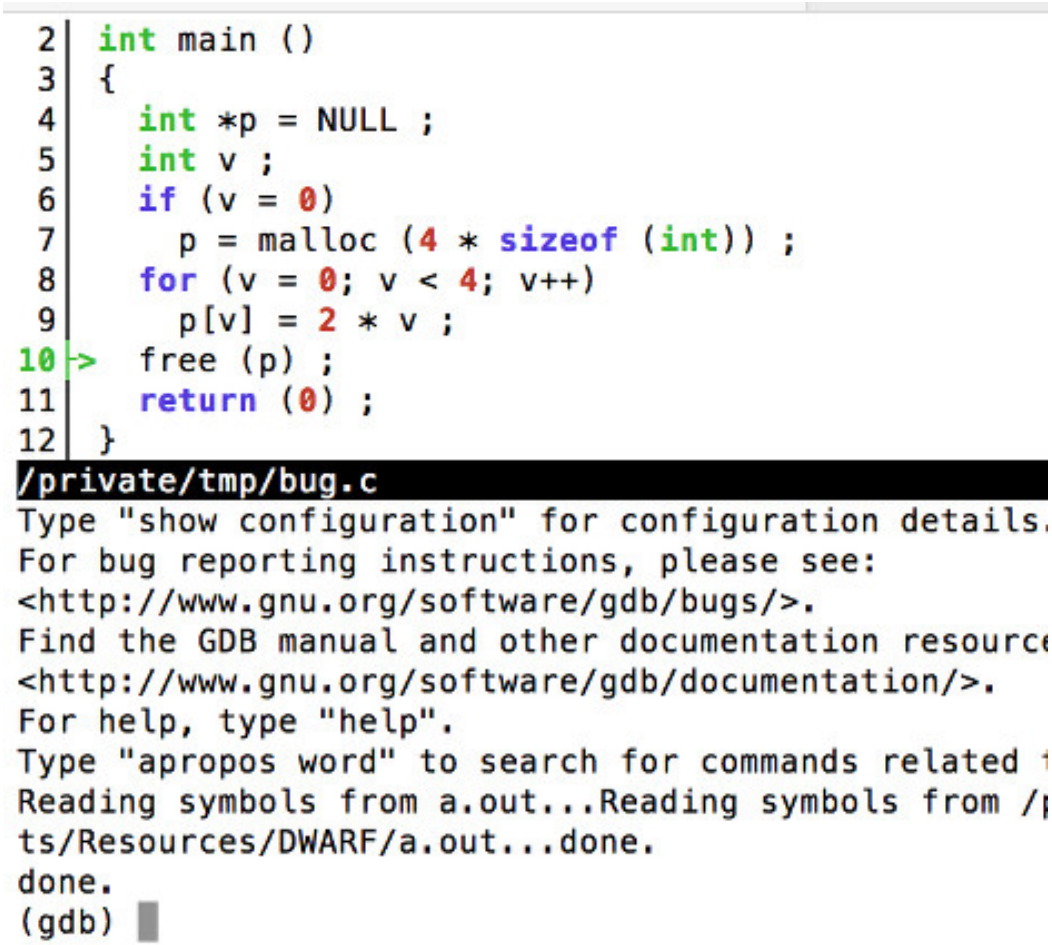
gdb : lancer un programme

```
#include <stdlib.h>
int main ()
{
  int *p = NULL ;
  int v ;
  if (v = 0)
    p = malloc (4 * sizeof (int)) ;
  for (v = 0; v < 4; v++)
    p[v] = 2 * v ;
  free (p) ;
  return (0) ;
}
```

```
mymac:$ gcc -g bug.c
mymac:$ gdb ./a.out
```

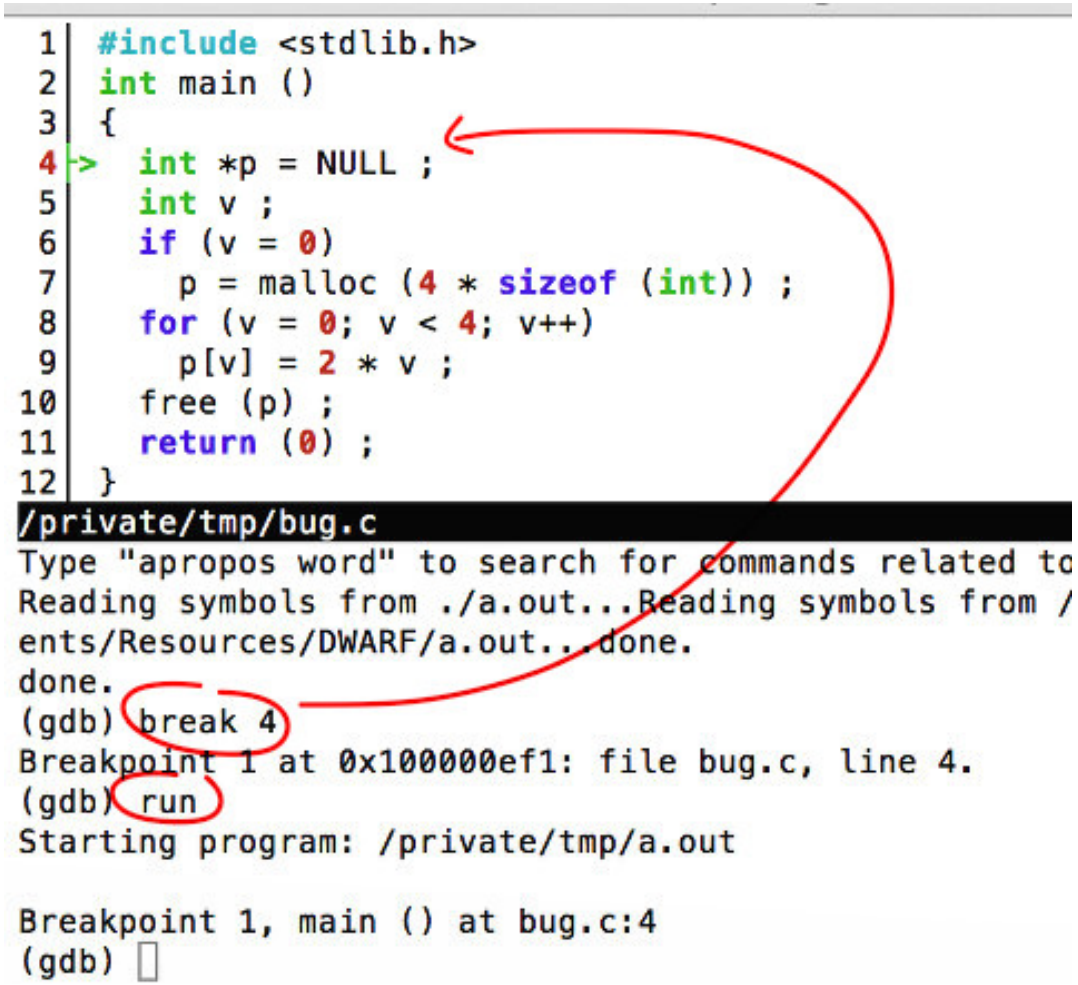
Si arguments à passer :

```
(gdb) set args toto tata titi
```



gdb : points d'arrêt et exécution

- Point d'arrêt : instruction où arrêter l'exécution.
  - Instruction au point d'arrêt non encore exécutée.
- Mettre un point d'arrêt
  - `break` n° ligne
  - `break` fichier : n° ligne
  - `break` nom fonction
  - `break` fichier : nom fonction
- Lancer le programme
  - `run`



gdb : gestion des points d'arrêt

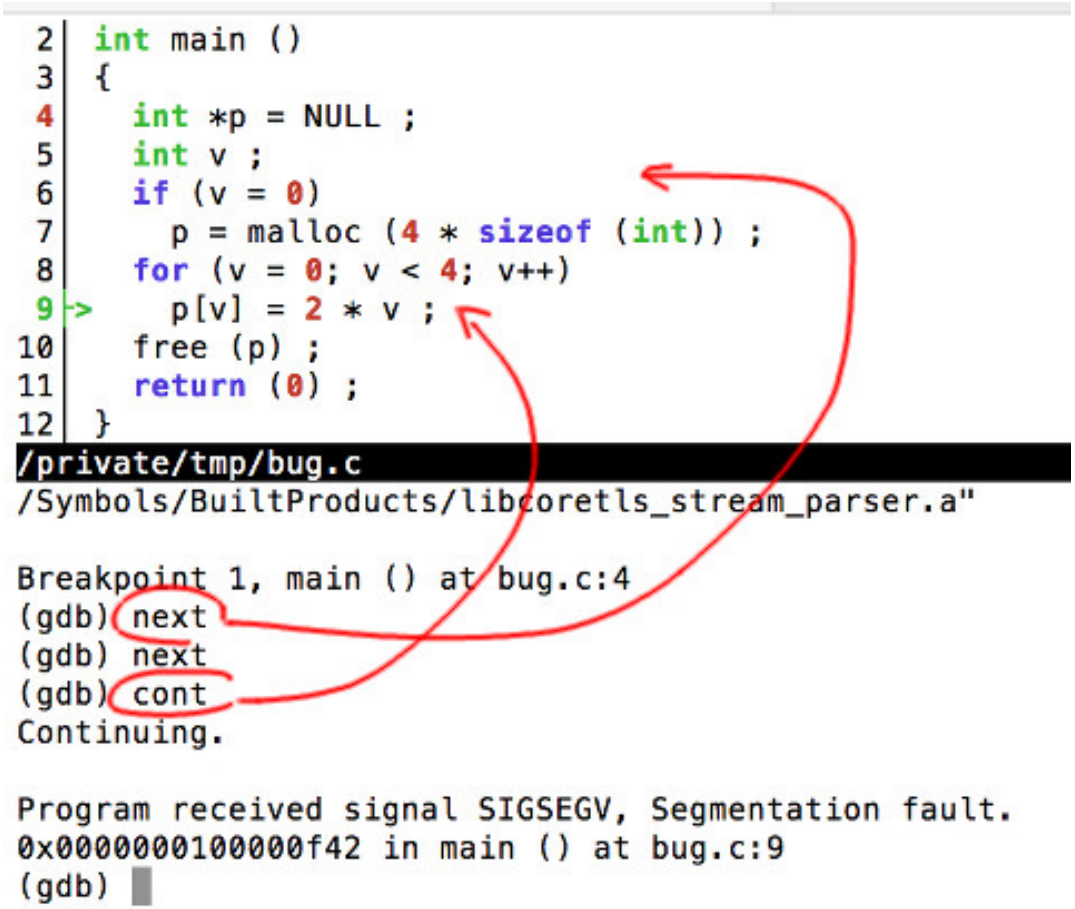
- Liste des points d'arrêt : `info break`

```
/private/tmp/bug.c
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000100000ef3 in main at bug.c:4
        breakpoint already hit 1 time
(gdb)
```

- Supprimer un point d'arrêt : `delete` n° pt arrêt
- Désactiver un point d'arrêt : `disable` n° pt arrêt
- Ré-activer un point d'arrêt : `enable` n° pt arrêt

gdb : continuer l'exécution

- Instruction suivante
  - `next` (appel fonction ≡ 1 instruction)
  - `step` (descend dans appel fonction)
- Continuer → prochain point d'arrêt :
  - `cont`



gdb : step vs next (1/2)

- `step` : rentre dans la fonction.

```
 3  int plus_one (int p)
 4  {
 5    return (p + 1) ;
 6  }
 7
 8  int main ()
 9  {
10->  int v = 5 ;
11    v = plus_one (v) ;
12    return (0) ;
13  }
/private/tmp/foo.c
For help, type "help".
Type "apropos word" to search for commands related to
Reading symbols from a.out...Reading symbols from /pr
ts/Resources/DWARF/a.out...done.
done.
(gdb) break 10
Breakpoint 1 at 0x100000f7f: file foo.c, line 10.
(gdb) run
Breakpoint 1, main () at foo.c:10
(gdb)
```

```
 1  #include <stdlib.h>
 2
 3  int plus_one (int p)
 4  {
 5->  return (p + 1) ;
 6  }
 7
 8  int main ()
 9  {
10    int v = 5 ;
11    v = plus_one (v) ;
/private/tmp/foo.c
/Symbols/BuiltProducts/libcoretls_ha
warning: Could not open OSO archive
/Symbols/BuiltProducts/libcoretls_re
warning: Could not open OSO archive
/Symbols/BuiltProducts/libcoretls_st

Breakpoint 1, main () at foo.c:10
(gdb) step
(gdb) step
plus_one (p=5) at foo.c:5
(gdb)
```

gdb : step vs next (2/2)

- `next` : « survole » la fonction.

```
 3  int plus_one (int p)
 4  {
 5    return (p + 1) ;
 6  }
 7
 8  int main ()
 9  {
10->  int v = 5 ;
11    v = plus_one (v) ;
12    return (0) ;
13  }
/private/tmp/foo.c
For help, type "help".
Type "apropos word" to search for commands related to
Reading symbols from a.out...Reading symbols from /pr
ts/Resources/DWARF/a.out...done.
done.
(gdb) break 10
Breakpoint 1 at 0x100000f7f: file foo.c, line 10.
(gdb) run
Breakpoint 1, main () at foo.c:10
(gdb)
```

```
 3  int plus_one (int p)
 4  {
 5    return (p + 1) ;
 6  }
 7
 8  int main ()
 9  {
10    int v = 5 ;
11    v = plus_one (v) ;
12->  return (0) ;
13  }
/private/tmp/foo.c
warning: Could not open OSO archive
/Symbols/BuiltProducts/libcoretls_h
warning: Could not open OSO archive
/Symbols/BuiltProducts/libcoretls_r
warning: Could not open OSO archive
/Symbols/BuiltProducts/libcoretls_s

Breakpoint 1, main () at foo.c:10
(gdb) next
(gdb) next
(gdb)
```

gdb : localiser un plantage

- Message d'erreur.
- Fichier + ligne.

⇒ Reprenez votre source. . .

```
 2  int main ()
 3  {
 4    int *p = NULL ;
 5    int v ;
 6    if (v = 0)
 7      p = malloc (4 * sizeof (int)) ;
 8    for (v = 0; v < 4; v++)
 9->    p[v] = 2 * v ;
10    free (p) ;
11    return (0) ;
12  }
/private/tmp/bug.c
/Symbols/BuiltProducts/libcoretls_stream_parser.a"

Breakpoint 1, main () at bug.c:4
(gdb) next
(gdb) next
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000000100000f42 in main () at bug.c:9
(gdb)
```

gdb : inspecter les variables

- `print` expression C
- Valeur de `v` : 0
- Valeur de `p[0]` : erreur !
- "*Cannot access memory at address 0x0*"
  - Pointeur `p` est NULL ☹

```
 2  int main ()
 3  {
 4    int *p = NULL ;
 5    int v ;
 6    if (v = 0)
 7      p = malloc (4 * sizeof (int)) ;
 8    for (v = 0; v < 4; v++)
 9->    p[v] = 2 * v ;
10    free (p) ;
11    return (0) ;
12  }
/private/tmp/bug.c
(gdb) next
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000000100000f42 in main () at bug.c:9
(gdb) print v
$1 = 0
(gdb) print p[0]
Cannot access memory at address 0x0
(gdb)
```

gdb et le « core dumped »

- Parfois, plantage et message « *core dumped* ».

⇒ Fichier `core` généré dans le répertoire.

- Contient l'état mémoire complet du processus au moment du plantage.
- Peut être chargé en même temps que l'exécutable dans `gdb`.

⇒ Débuggage « *post-mortem* ».

```
localhost:/tmp/foo$ ./bug.x
Segmentation fault (core dumped)
localhost:/tmp/foo$ ls
bug.c  bug.x  core.2505
localhost:/tmp/foo$
localhost:/tmp/foo$ gdb bug.x core.2505
GNU gdb (GDB) Fedora 7.9.1-17.fc22
Copyright (C) 2015 Free Software Foundation, Inc.
Reading symbols from bug.x...done.
[New LWP 2505]
Core was generated by `./bug.x'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000000400506 in main () at bug.c:4
4   *p = 456 ;
(gdb)
```