

## Programmation

Pr. Olivier Gruber

[olivier.gruber@univ-grenoble-alpes.fr](mailto:olivier.gruber@univ-grenoble-alpes.fr)

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

- La notion d'exécution
- Debugging

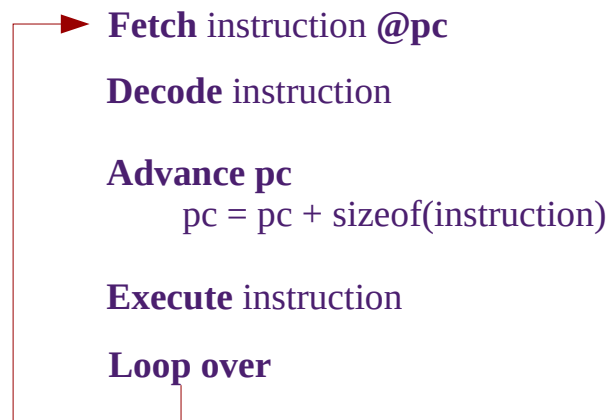
# Exécution vue par le matériel

3

## Où se trouve l'exécution ?

- Où dans le code ?
- Où dans les données ?

### Processor :

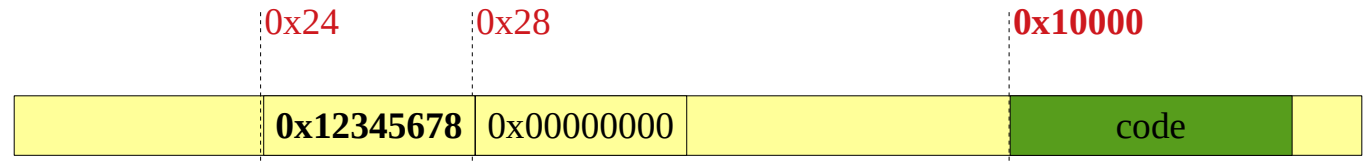


0x10000	mov r0, #0x24
0x10004	ldrb r1, [r0]
0x10008	mov r2, #0x28
0x1000C	str r1, [r2]
0x10010	mov pc, r3

### Processor

pc	0x10008
r0	0x24
r1	0x12345678
r2	???
r3	0x10000

### Memory:



# Debug – Points d'arrêt

4

- Où se trouve l'exécution dans le code ?
  - Le debugger vous le montre...
  - N'oubliez pas de compiler avec les informations de debug (option **-g**)

source file  
« fact.c »

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```

```
36 int main(int argc, char** argv) {  
37     int n = fact(6);  
38     return 0;  
39 }
```

```
$ gcc -g -o fact fact.c  
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb)
```

A propos des points d'arrêt

```
(gdb) breakpoint fact.c:37  
(gdb) breakpoint fact  
(gdb) br fact  
(gdb) info br // liste les brkpt  
(gdb) delete 2 // delete brkpt 2  
(gdb) d 2 // delete brkpt 2
```

- Commande : next

- Continue l'exécution jusqu'au prochain statement, sans montrer l'exécution des appels de fonction

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```

```
36 int main(int argc, char** argv) {  
• 37     int n = fact(6);  
➔ 38     return 0;  
39 }
```

```
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb) next  
38 return 0;  
(gdb)
```

Différentes formes:

```
(gdb) next  
(gdb) n  
  
(gdb) nexti // next assembly instruction  
(gdb) ni
```

# Debug – Afficher des données


6

- Commande : print

- Continue l'exécution jusqu'au prochain statement, sans montrer l'exécution des appels de fonction
- Les appels ont été exécuté !

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```

```
36 int main(int argc, char** argv) {  
● 37     int n = fact(6);  
38     return 0;  
39 }
```



```
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb) next  
38 return 0;  
(gdb) print n  
$1 = 720  
(gdb)
```

Pour afficher:

```
(gdb) print /fmt c-expr  
c-expr: any C expression  
fmt: format d'affichage,  
      inspiré des formats de printf  
p /x exp // hexadecimal  
p /f exp // float  
p /c exp // character
```

# Debug – Relâcher le contrôle de l'exécution

7

- Commande : continue
  - Continue l'exécution...
  - Jusqu'à la fin ou le prochain point d'arrêt

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```

```
36 int main(int argc, char** argv) {  
● 37     int n = fact(6);  
→ 38     return 0;  
39 }
```

```
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb) next  
38 return 0;  
(gdb) print n  
$1 = 720  
(gdb) continue // ou "cont"  
[Inferior 1 (process 260331) exited normally]  
(gdb) quit  
$
```

# Debug – Tuer l'exécution

8

- Commande : kill

- Permet de tuer l'exécution
- En fait, cela tue le processus
- Le processus qui abrite l'exécution

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```

```
36 int main(int argc, char** argv) {  
● 37     int n = fact(6);  
→ 38     return 0;  
39 }
```

```
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb) next  
38 return 0;  
(gdb) print n  
$1 = 720  
(gdb) kill // ou "k"  
Kill the program being debugged? (y or n) y  
[Inferior 1 (process 260331) killed]  
(gdb) quit  
$
```




# Debug – Pas à pas

9

- Commande : step
  - Continue l'exécution
  - Montre l'exécution des appels

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```



```
36 int main(int argc, char** argv) {  
● 37     int n = fact(6);  
38     return 0;  
39 }
```


```
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb) step
```

Différentes formes:


```
(gdb) step  
(gdb) s
```

```
(gdb) stepi // assembly instruction level  
(gdb) si
```

- Commande : step
  - Continue l'exécution
  - Montre l'exécution des appels



```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```




```
36 int main(int argc, char** argv) {  
• 37     int n = fact(6);  
38     return 0;  
39 }
```

```
$ gdb fact  
(gdb) breakpoint fact.c:37  
Breakpoint 1 at fact.c:37  
(gdb) run  
Breakpoint 1, fact(v0=3) at fact.c:37  
37 int n = fact(6);  
(gdb) step  
13 int fact(int v0) {  
(gdb) where  
#0 fact(v0=6) at fact.c:13  
#1 main(argc=1, argv=0x7fffdd48) at fact.c:37  
(gdb)
```


# Debug – Observer la pile d'appels

11

- Commandes : up / down
  - Déplace le point de vue du debugger
  - Change la visibilité des symboles
  - Attention aux variables non visibles



```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```



```
36 int main(int argc, char** argv) {  
• 37     int n = fact(6);  
38     return 0;  
39 }
```

(gdb) **where**

#0 fact(v0=6) at fact.c:13

#1 main(argc=1, argv=0x7ffdd48) at fact.c:37

(gdb) **print v0**

\$1=6


(gdb) **p argc**

No symbol "argc" in current context


# Debug – Observer la pile d'appels

12

- Commandes : up / down
  - Attention aux variables non initialisées



```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```



```
36 int main(int argc, char** argv) {  
• 37     int n = fact(6);  
38     return 0;  
39 }
```

what ???



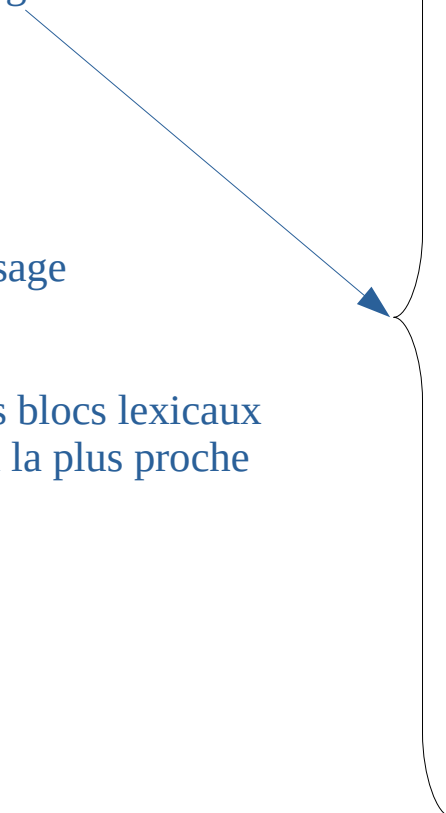
```
(gdb) where  
#0 fact(v0=6) at fact.c:13  
#1 main(argc=1, argv=0x7ffdd48) at fact.c:37  
(gdb) print v0  
$1=6  
(gdb) up  
#1 main(argc=1, argv=0x7ffdd48)  
37 int n = fact(6);  
(gdb) p argc  
$2 = 1  
(gdb) print n  
$2 = 32767  
(gdb)
```

Le debugger suit les mêmes règles de visibilité des symboles que le compilateur...

## Dans le langage C:

En partant de l'endroit d'un usage du nom d'une variable :

- a) remonter dans l'arbre des blocs lexicaux pour trouver la définition la plus proche d'une variable locale.
- b) arguments de la fonction
- c) variables globales



```
(gdb) where
#0 fact(v0=6) at fact.c:13
#1 main(argc=1, argv=0x7ffdd48) at fact.c:37
(gdb) print v0
$1=6
(gdb) print argc
No symbol "argc" in current context

(gdb) up
#1 main(argc=1, argv=0x7ffdd48)
37  int n = fact(6);

(gdb) print argc
$2 = 1
(gdb) print n
$2 = 32767

(gdb) down
#0 fact(v0=6) at fact.c:13
12  int fact(int v0) {

(gdb) print n
No Symbol "n" in current context.
```

# Règles de visibilité pour les variables – A comprendre

14

```
#include <stdio.h>
#include <string.h>
int n = 3;
int main(int argc, char** argv) {
    int n=1;
    int r;
    char* s;
    if (argc==0) {
        int n;
        char* s = "hello";
        n = strlen(s);
        r = n;
    } else {
        s = argv[n];
        r = strlen(argv[n]);
    }
    printf("arg:%s len=[%d]\n", s, r);
    return 0;
}
```

Une variable peut en cacher une autre...

C'est correcte... ou pas...

Et le compilateur ne voit pas toujours l'erreur...

```
$ gcc -Wall -o a a.c
$ ./a toto
arg:toto length=[4]
$ ./a
Segmentation fault (core dumped)
$
```

Usage de « s » incorrect si argc==0

# Factoriel en récursif – Un Exemple...

15

- Solution de l'exécution de la fonction factorielle récursive
  - Pour tout entier  $n > 0$ ,  $n! = (n - 1)! \times n$ .
  - Pour bien comprendre la pile d'appels

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

Où se trouve l'exécution ?

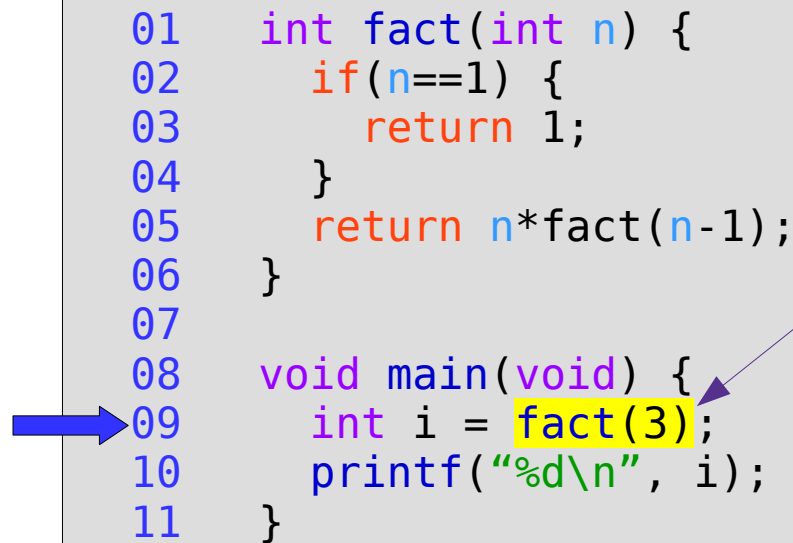
- Où dans le code ?
- Où dans les données ?

→ Il faut parler  
de la pile d'appels  
(Call Stack)

# Debug – Pile d'appels – Les étapes d'un appel

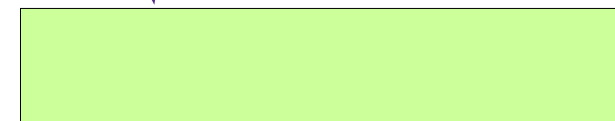
16

- Exécution :
  - Fonction main, ligne 09, **site d'appel** de la fonction « fact »
    - La nouvelle « stack frame » est allouée sur la pile (stack)



```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

L'exécution de l'appel va provoquer le calcul et la capture des valeurs des paramètres et l'empilement d'une nouvelle « frame ».



```
main(void)  
09  fact(3)
```

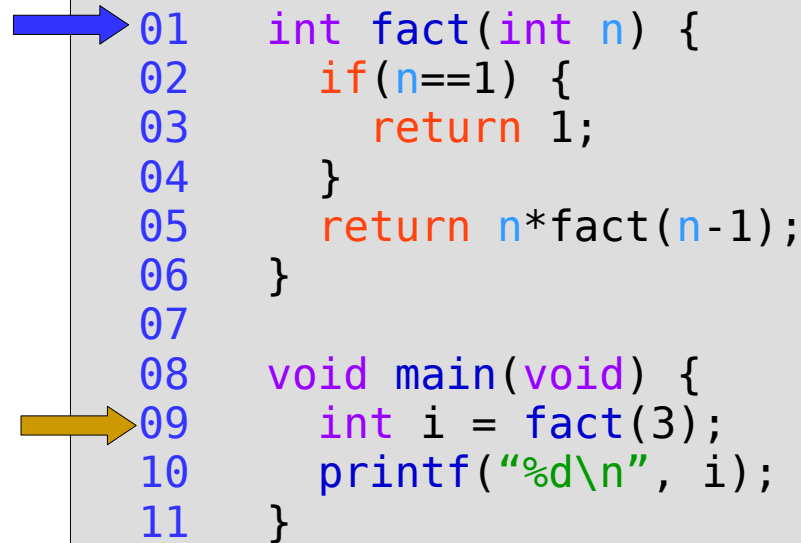
Call Stack / Pile d'appels



# Debug – Pile d'appels – Les étapes d'un appel

17

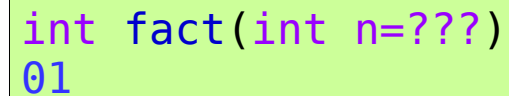
- Exécution :
  - Fonction main, ligne 09, **site d'appel** de la fonction « fact »
    - La nouvelle « stack frame » est allouée sur la pile (stack)
    - L'exécution se déplace dans la fonction « fact »



A diagram showing a snippet of C code with line numbers 01 to 11. A blue arrow points to line 01, and a yellow arrow points to line 09. The code defines a recursive factorial function 'fact' and a 'main' function that calls 'fact(3)'.

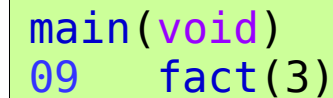
```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

L'exécution rentre dans « fact », elle est à la ligne 1 du source. Notez que le paramètre « n » n'a pas encore pris sa valeur (3)



A diagram of the call stack showing two frames. The top frame is for 'fact(int n=???)' at line 01. A purple arrow points from the text above to this frame.

```
int fact(int n=???)  
01
```



The bottom frame of the call stack is for 'main(void)' at line 09, showing the call to 'fact(3)'.

```
main(void)  
09  fact(3)
```

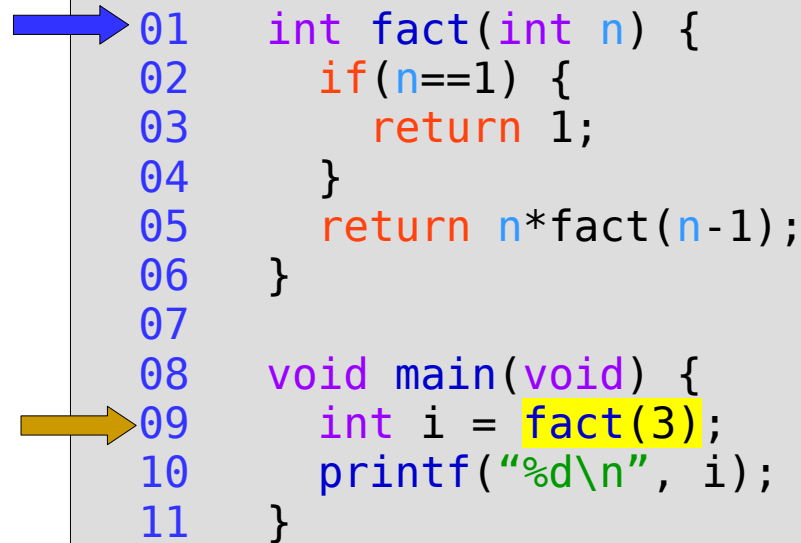
Call Stack / Pile d'appels

# Debug – Pile d'appels – Les étapes d'un appel

18

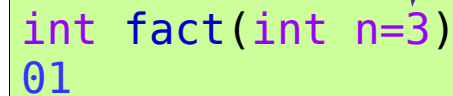
- Exécution :

- Fonction main, ligne 09, **site d'appel** de la fonction « fact »
  - La nouvelle « stack frame » est allouée sur la pile (stack)
  - L'exécution se déplace dans la fonction « fact »
  - Le prologue de la fonction « fact » s'exécute, les arguments prennent leur valeur



```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

Le prologue copie les valeurs  
données par le site d'appel  
dans les arguments.



```
int fact(int n=3)  
01
```

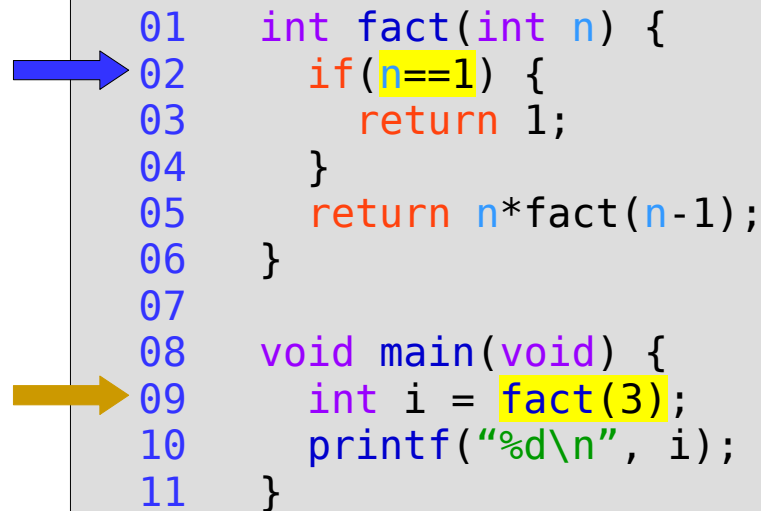
```
main(void)  
09  fact(3)
```

Call Stack / Pile d'appels

# Debug – Pile d'appels – Les étapes d'un appel

19

- Exécution :
  - Fonction main, ligne 09, site d'appel de la fonction « fact »
  - Fonction « fact », ligne 02, le prologue est fini
  - L'exécution du code de la fonction commence...



```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

```
int fact(int n=3)  
02  (n==1)
```

```
main(void)  
09  fact(3)
```

Call Stack / Pile d'appels

- Exécution :
  - Fonction main, ligne 09, site d'appel de la fonction « fact »
  - Fonction « fact », ligne 05, **site d'appel** récursif de la fonction « fact »
  - La nouvelle valeur du paramètre « n » va être calculée...
    - C'est la valeur (n-1), soit 2

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

L'exécution va d'abord évaluer l'expression (n-1), puis faire l'appel

```
int fact(int n=3)  
05 (n-1)
```

```
main(void)  
09 fact(3)
```

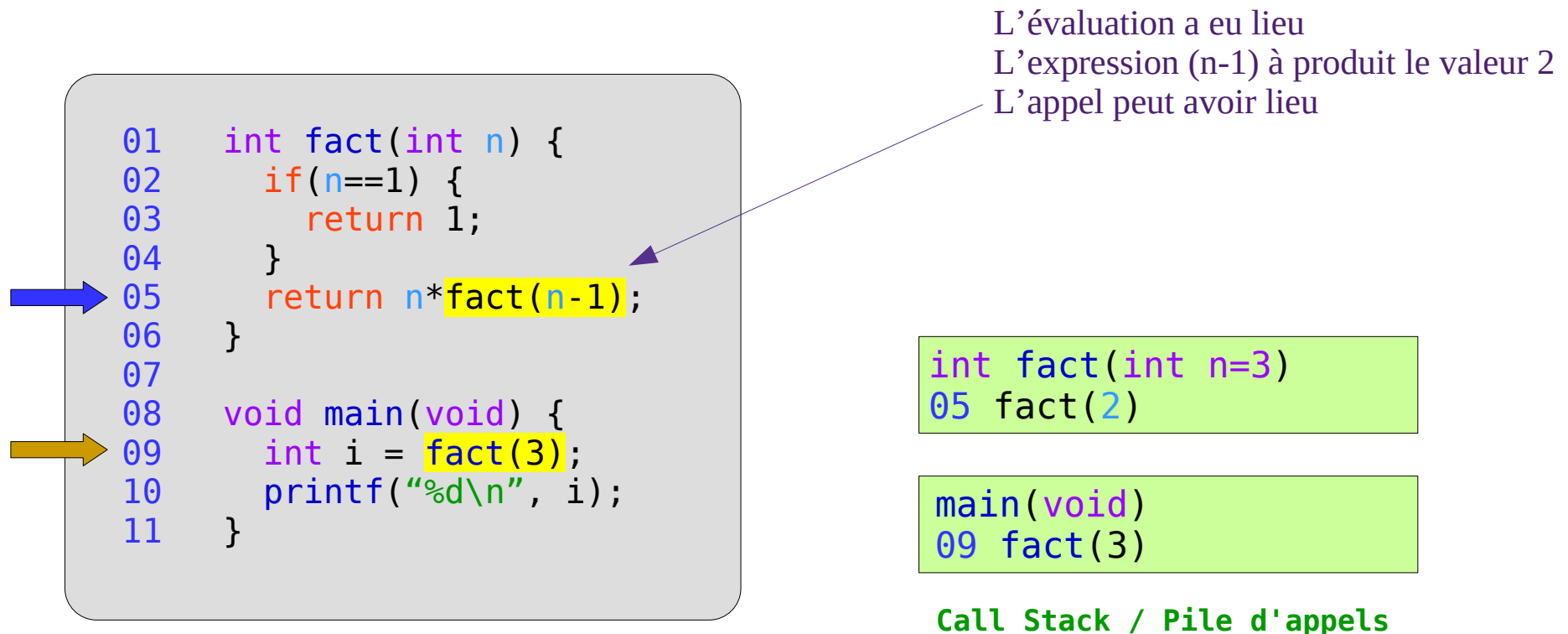
Call Stack / Pile d'appels

# Appels – Calcul des Valeurs & Arguments

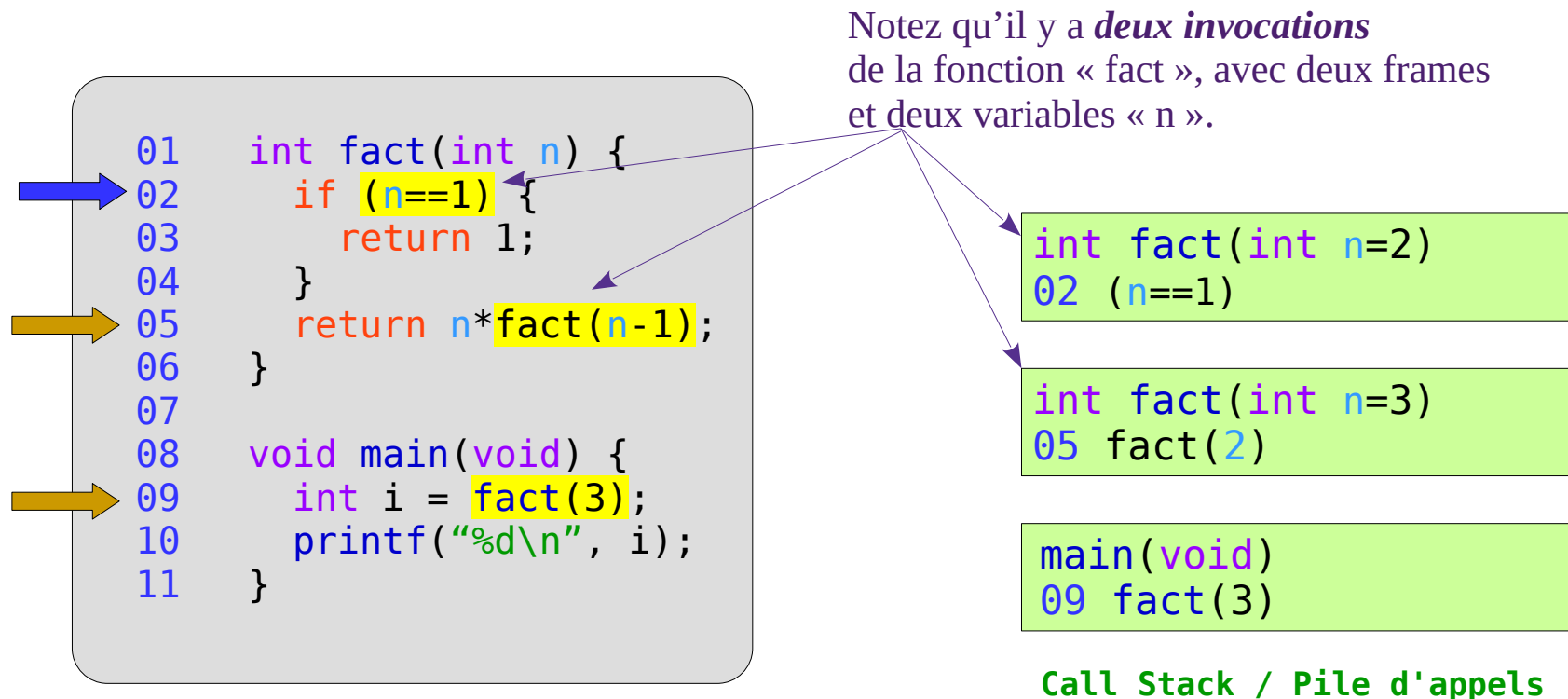
21

- Exécution :

- Fonction main, ligne 09, site d'appel de la fonction « fact »
- Fonction « fact », ligne 05, **site d'appel** récursif de la fonction « fact »
- La nouvelle valeur du paramètre « n » va être calculée... c'est la valeur (n-1), soit 2



- Exécution :
  - Fonction main, ligne 09, site d'appel de la fonction « fact »
  - Fonction « fact », ligne 05, **site d'appel** récursif de la fonction « fact »
  - La nouvelle valeur du paramètre « n » va être calculée... c'est la valeur (n-1), soit 2
  - L'appel a eu lieu... et les mêmes étapes prennent place...



# Pile d'appels & Frames – Pointeurs

23

```
int fact(int n=2)
02 (n==1)
```

```
int fact(int n=3)
05 fact(2)
```

```
main(void)
09 fact(3)
```

Call Stack / Pile d'appels

Deux variables,  
avec deux zones  
mémoire.

```
01 int fact(int n) {
02     if(n==1) {
03         return 1;
04     }
05     return n*fact(n-1);
06 }
07
08 void main(void) {
09     int i = fact(3);
10     printf("%d\n", i);
11 }
```

```
(gdb) run
Breakpoint 1, fact(n=3) at fact.c:2
02 if (n==1) {
(gdb) print n
$1 = 3
(gdb) print &n
$2 = (int *) 0x7fffddec

(gdb) continue
Breakpoint 1, fact(n=2) at fact.c:2
02 if (n==1) {

(gdb) where
#0 fact (n=2) at fact.c:2
#1 in fact (n=3) at fact.c:5
#2 in main (...) at fact.c:9
(gdb) print &n
$4 = (int *) 0x7ffddcc
(gdb) print n
$5 = 2
(gdb)
```

# Pile d'appels & Pointeurs – Correct ?

24

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(dec(&n));  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }  
12  
13  int dec(int* n) {  
14      return *n - 1;  
15  }
```

Est-ce que ce programme est correct ?

- usage correct des pointeurs ?
- calcul correctement factoriel ?

```
int dec(int* n)  
14 (*n-1)
```

```
int fact(int n=3)  
05 dec(&n)
```

```
main(void)  
09 fact(3)
```

Call Stack / Pile d'appels



# Pile d'appels & Pointeurs – Correct ?

25

```
01  int fact(int n) {  
●02      if(n==1) {  
03          return 1;  
04      }  
→05      return n*fact(dec(&n));  
06  }  
07  
08  void main(void) {  
→09      int i = fact(3);  
10      printf("%d\n", i);  
11  }  
12  
13  int dec(int* n) {  
→14      return *n = *n - 1;  
15  }
```

Est-ce que ce programme est correct ?

- usage correct des pointeurs ?
- calcul correctement factoriel ?

```
int dec(int* n)  
14 (*n=)
```

```
int fact(int n=3)  
05 dec(&n)
```

```
main(void)  
09 fact(3)
```

Call Stack / Pile d'appels

# Factoriel en récursif – La fin de la récursion, les « return »

26

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

```
int fact(int n=1)  
03  return 1
```

```
int fact(int n=2)  
05  fact(1)
```

```
int fact(int n=3)  
05  fact(2)
```


```
main(void)  
09  fact(3)
```

Call Stack / Pile d'appels

# Pile d'appels – Return

27

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```



```
int fact(int n=1)  
03  return 1
```

```
int fact(int n=2)  
05  n*1
```

```
int fact(int n=3)  
05  fact(2)
```

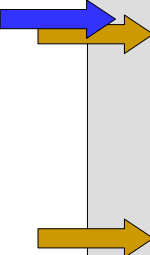
```
main(void)  
09  fact(3)
```

Call Stack / Pile d'appels

# Pile d'appels – Return

28

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```



```
int fact(int n=2)  
05 return 2
```

```
int fact(int n=3)  
05 fact(2)
```

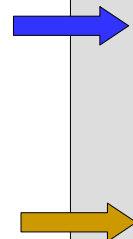
```
main(void)  
09 fact(3)
```

Call Stack / Pile d'appels

# Exécution Récursive – A finir par vous même...

29

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```



```
int fact(int n=2)  
05 return 2
```

```
int fact(int n=3)  
05 n*2
```

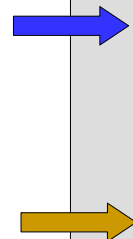
```
main(void)  
09 fact(3)
```

Call Stack / Pile d'appels

# Exécution Récursive – A finir par vous même...

30

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```




```
int fact(int n=3)  
05 return 6
```

```
main(void)  
09 fact(3)
```

Call Stack / Pile d'appels

# Exécution Récursive – A finir par vous même...

31

```
01  int fact(int n) {  
02      if(n==1) {  
03          return 1;  
04      }  
05      return n*fact(n-1);  
06  }  
07  
08  void main(void) {  
09      int i = fact(3);  
10      printf("%d\n", i);  
11  }
```

```
int fact(int n=3)  
05 return 6
```

```
main(void)  
09 i=6
```

Call Stack / Pile d'appels

# Exécution vue par le matériel

32

## Où se trouve l'exécution ?

✓ - Où dans le code ?

➡ - Où dans les données ?

### Processor :

➡ **Fetch** instruction @pc


**Decode** instruction

**Advance pc**

pc = pc + sizeof(instruction)

**Execute** instruction

**Loop over**



<b>0x10000</b>	mov r0, #0x24
0x10004	ldrb r1, [r0]
<b>0x10008</b>	<b>mov r2, #0x28</b>
0x1000C	str r1, [r2]
0x10010	<b>mov pc, r3</b>

### Processor

➡ pc	<b>0x10010</b>
r0	0x24
r1	0x12345678
r2	0x28
r3	0x10000

### Memory:

	0x24	0x28		0x10000
	0x12345678	0x00000000		code



- L'appel de fonction avec pointeur

```
06 int _strlen(char *s) {  
07     int count=0;  
08     while (*s != '\0') {  
09         s = s + 1;  
10         count = count+1;  
11     }  
12     return count;  
13 }
```

```
26 char* s1 = "Hello";  
27 char* s2 = "World";  
  
28 void main(void) {  
29     int length = _strlen(s1);  
30 }
```

**Règle :** Les valeurs fournis par le site d'appel sont toujours copiées dans les arguments.  
Même les pointeurs !  
Leur valeur qui est copiée est l'adresse.

```
int _strlen(char *s=0x78afedec)  
07     int count=0;
```

```
main(void)  
09 int length = _strlen(s1);
```

Call Stack / Pile d'appels

```
(gdb) run  
Breakpoint 1, strlen(n=3) at strlen.c:7  
07 int count=0;  
(gdb) print s  
$1 = (char *) 0x78afedec  
(gdb) up  
29 int length=strlen(s1);  
(gdb) print s1  
$2 = (char *) 0x78afedec  
(gdb)
```

copied address

- L'appel de fonction avec pointeur

```
int _strlen(char *s) {  
    int count=0;  
    while (*s != '\0') {  
        s = s + 1;  
        count = count+1;  
    }  
    return count;  
}
```

Où se trouve l'exécution ?

- Où dans le code ?
- Où dans les données ?

Pour le code :

Regardez la pile d'appels

Pour les données :

Regardez les pointeurs !

```
char* s1 = "Hello";  
char* s2 = "World";  
int length = _strlen(s1) + _strlen(s2);
```

Ici, deux appels de la même fonction  
qui manipulent des données différentes

Si un appel de \_strlen retourne une longueur  
fausse, est-ce parce que le code est faux ou  
bien les données le sont ?

# Point d'arrêt sur les données – Exemple I

35

- Point d'arrêt avec condition
  - *Permet de s'arrêter dans une boucle*

File: string.c

```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

File: main.c

```
07 char *s1 = "Hello Fred";  
08 char *s2 = "Hello Anais";  
09 int main(int argc, char **argv) {  
10     int c1 = _strlen(s1);  
11     int c2 = _strlen(s2);  
12     return c1+c2;  
13 }
```

```
(gdb) br string.c:24  
Breakpoint 4 at string.c:24  
(gdb) condition 4 (*s=='A')
```

```
(gdb) run  
Breakpoint 4, _strlen(...) at string.c:24  
24  s = s + 1;
```

```
(gdb) print *s  
$1 = 65 'A'
```

```
(gdb) print s  
$2 = 0x55556009 "Anais"
```

```
(gdb) up
```

```
...
```

```
(gdb) print s2  
$2 = 0x55556003 "Hello Anais"
```

```
(gdb) condition 4  
Breakpoint 4 now unconditional.
```

# Point d'arrêt sur les données – Exemple II

36

- Point d'arrêt avec condition
  - Permet de s'arrêter dans une boucle
  - **Permet de s'arrêter dans un certain appel**

File: string.c

```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

File: main.c

```
07 char *s1 = "Hello Fred";  
08 char *s2 = "Hello Anais";  
09 int main(int argc, char **argv) {  
10     int c1 = _strlen(s1);  
11     int c2 = _strlen(s2);  
12     return c1+c2;  
13 }
```

```
(gdb) br string.c:22  
Breakpoint 4 at string.c:22
```

```
(gdb) condition 4 (s==s2)
```

```
(gdb) run  
Breakpoint 1, _strlen (s="Hello Anais")  
at string.c:22
```

```
(gdb) print *s  
$1 = 72 'H'
```

```
(gdb) print s  
$2 = 0x55556003 "Hello Anais"
```

```
(gdb) condition 4  
Breakpoint 4 now unconditional.
```

# Point d'arrêt sur les données – Problème

37

- Point d'arrêt avec condition
  - Permet de s'arrêter dans une boucle
  - **Permet de s'arrêter dans un certain appel**

File: string.c

```
21 int _strlen(char *s) {  
• 22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```


File: main.c

```
07 int main(int argc, char **argv) {  
08     char *s1 = "Hello Fred";  
09     char *s2 = "Hello Anais";  
➔ • 10     int c1 = _strlen(s1);  
11     int c2 = _strlen(s2);  
12     return c1+c2;  
13 }
```

```
(gdb) br main.c:10  
Breakpoint 1 at main.c:10
```

```
(gdb) run  
Breakpoint 1,  
main (argc=1, argv=0x7ffdb58) at main.c:10  
10 int c1 = strlen(s1);
```

```
(gdb) br string.c:22  
Breakpoint 2 at string.c:22
```

```
(gdb) condition 2 (s==s2)   
No symbol "s2" in current context  
(gdb)
```

*Ici, les variables sont locales à « main »  
et ne sont pas visible depuis le context  
du point d'arrêt 4...*

*Il nous faut passer par une variable locale  
du debugger, les fameuses variables \$x...*

# Point d'arrêt sur les données – Solution (gdb variables)

38

- Point d'arrêt avec condition
  - Permet de s'arrêter dans une boucle
  - **Permet de s'arrêter dans un certain appel**

File: string.c

```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

File: main.c

```
07 int main(int argc, char **argv) {  
08     char *s1 = "Hello Fred";  
09     char *s2 = "Hello Anais";  
10     int c1 = _strlen(s1);  
11     int c2 = _strlen(s2);  
12     return c1+c2;  
13 }
```

(gdb) **br main.c:10**

Breakpoint 1 at main.c:10

(gdb) **run**

Breakpoint 1,

main (argc=1, argv=0x7fffdb58) at main.c:10

10 int c1 = strlen(s1);

(gdb) **br string.c:22**

Breakpoint 2 at string.c:22

(gdb) **print s2**

**\$1** = 0x5555600f "Hello Anais"

(gdb) **condition 2 (s==\$1)**

(gdb) **cont**


Continuing...

# Point d'arrêt sur les données – Solution (gdb variables)

39


- Point d'arrêt avec condition
  - Permet de s'arrêter dans une boucle
  - **Permet de s'arrêter dans un certain appel**

File: string.c



```
21 int _strlen(char *s) {
22     int count=0;
23     while (*s != '\0') {
24         s = s + 1;
25         count = count+1;
26     }
27     return count;
28 }
```

File: main.c



```
07 int main(int argc, char **argv) {
08     char *s1 = "Hello Fred";
09     char *s2 = "Hello Anais";
10     int c1 = _strlen(s1);
11     int c2 = _strlen(s2);
12     return c1+c2;
13 }
```

(gdb) **br main.c:10**

Breakpoint 1 at main.c:10

(gdb) **run**

Breakpoint 1,  
main (argc=1, argv=0x7ffdb58) at main.c:10

10 int c1 = strlen(s1);

(gdb) **br string.c:22**

Breakpoint 2 at string.c:22

(gdb) **print s2**

**\$1 = 0x5555600f "Hello Anais"**

(gdb) **condition 2 (s==\$1)**

(gdb) **cont**

Continuing...

Breakpoint 2,


\_strlen (s=0x5555600f "Hello Anais") at string.c:22

22 int count = 0;

(gdb)

- Afficher le contenu de la mémoire
  - Par une variable pointer
  - En forgeant un pointeur

File: string.c



```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

```
(gdb) br string.c:24
```

```
Breakpoint 4 at string.c:24
```

```
(gdb) run
```

```
Breakpoint 4, _strlen(...) at string.c:24
```

```
s = s + 1;
```

```
(gdb) p s
```

```
$2 = 0x55556044 "Hello Fred"
```

```
(gdb) p (char*)0x55556044
```

```
$3 = 0x55556044 "Hello Fred"
```

```
(gdb) p *(char*)0x55556044
```

```
$4 = 72 'H'
```

```
(gdb) p *(uint8_t*)s
```

```
$5 = 72
```

```
(gdb) p /c *(uint8_t*)$2
```

```
$6 = 72 'H'
```

```
(gdb) p /x *(uint8_t*)$2
```

```
$6 = 0x48
```

```
(gdb) p *(char*)(s+1)
```

```
$6 = 101 'e'
```



- Afficher le contenu de la mémoire
  - Par une variable pointer
  - En forgeant un pointeur
  - Examiner la mémoire


```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

```
(gdb) run  
Breakpoint 4, _strlen(...) at string.c:24  
s = s + 1;  
  
(gdb) print s  
$1 = 0x55556044 "Hello Fred"  
  
(gdb) x /5cb s  
0x55556004:  72 'H' 101 'e' 108 'l' 108 'l' 111 'o'  
(gdb) x /5xb s  
0x55556004:  0x48 0x65 0x6c 0x6c 0x6f  
  
(gdb) p /x *(uint32_t*)s  
$5 = 0x6c6c6548
```

← **Petit boutisme  
(little endian)**

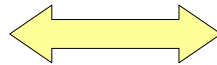
Pour avoir la syntaxe de la commande « x »,  
« x » pour « examine »,  
faire « help x » sous gdb.

- Watch points (points de surveillance)
  - S'arrête lorsqu'une location en mémoire est modifiée
    - Watch une variable : **watch count**
    - Watch une adresse : **watch \*0x55556004**
  - Utilité
    - Lorsqu'une structure de données en mémoire est corrompue par un mauvais usage d'un pointeur



```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

Exemple trivial  
pour comprendre...




```
(gdb) br _strlen  
Breakpoint 4, _strlen(...) at string.c:22
```

```
(gdb) run  
Breakpoint 4, _strlen(...) at string.c:22  
22 int _strlen(char* s)
```

```
(gdb) watch count  
Hardware watchpoint 7: count
```

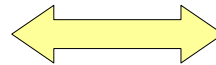
```
(gdb) continue  
Continuing.
```

- Watch points (points de surveillance)
  - S'arrête lorsqu'une location en mémoire est modifiée
    - Watch une variable : **watch count**
    - Watch une adresse : **watch \*0x55556004**
  - Utilité
    - Lorsqu'une structure de données en mémoire est corrompue par un mauvais usage d'un pointeur



```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

Exemple trivial  
pour comprendre...



```
(gdb) br _strlen  
Breakpoint 4, _strlen(...) at string.c:21
```

```
(gdb) run  
Breakpoint 4, _strlen(...) at string.c:21
```

```
(gdb) watch count  
Hardware watchpoint 7: count
```


```
(gdb) continue  
Continuing.
```

```
Hardware watchpoint 7: count  
Old value = 21845  
New value = 0  
_strlen(...) at string.c:22
```

# Exécution – Watch Point

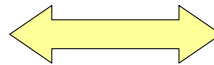
44

- Watch points (points de surveillance)
  - S'arrête lorsqu'une location en mémoire est modifiée
    - Watch une variable : **watch count**
    - Watch une adresse : **watch \*0x555555556004**
  - Utilité
    - Lorsqu'une structure de données en mémoire est corrompue par un mauvais usage d'un pointeur



```
21 int _strlen(char *s) {  
22     int count=0;  
23     while (*s != '\0') {  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

Exemple trivial  
pour comprendre...



```
(gdb) br _strlen  
Breakpoint 4, _strlen(...) at string.c:21
```

```
(gdb) run  
Breakpoint 4, _strlen(...) at string.c:21
```

```
(gdb) watch count  
Hardware watchpoint 7: count
```

```
(gdb) continue  
Continuing.
```

```
Hardware watchpoint 7: count  
Old value = 21845  
New value = 0  
_strlen(...) at string.c:22
```

```
(gdb) continue  
Continuing.
```

```
Hardware watchpoint 7: count  
Old value = 0  
New value = 1  
_strlen(...) at string.c:25
```

# Exemple de données corrompues

45

- Un exemple plus réel...

- L'exécution est **fausse...**
- Mais elle ne plante pas !

```
20 int strset(char *s, char c) {  
21     int count=0;  
22     while (*s != '\0') {  
23         *s = c;  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```

```
30 int c1,c2;  
31  
32 int main(int argc, char** argv) {  
33     char s1[] ="Hello Fred";  
34     char s2[] ="Hello Anaïs";  
35     c1 = strlen(s1)+strlen(s2);  
36     c2 = 0x00FFFFFF;  
37     strset((char*)&c2, 'z');  
38     return c1+c2;  
39 }
```

```
(gdb) run  
Breakpoint 4, main(...) at 36  
36     c2 = 0x00FFFFFF;  
(gdb) next  
37     strset((char*)&c2,'z');  
(gdb) print /x c2 ← valeur correcte  
$1 = 0xffffffff  
(gdb) x /4xb &c2 // confirm little-endian  
0x55558014 <c2>:  0xff 0xff 0xff 0x00  
(gdb)
```

# Exemple de données corrompues

46

- Un exemple plus réel...

- L'exécution est **fausse...**
- Mais elle ne plante pas !

```
20 int strset(char *s, char c) {
21     int count=0;
22     while (*s != '\0') {
23         *s = c;
24         s = s + 1;
25         count = count+1;
26     }
27     return count;
28 }
```

```
30 int c1,c2;
31
32 int main(int argc, char** argv) {
33     char s1[] ="Hello Fred";
34     char s2[] ="Hello Anaïs";
35     c1 = strlen(s1)+strlen(s2);
36     c2 = 0x00FFFFFF;
37     strset((char*)&c2, 'z');
38     return c1+c2;
39 }
```

```
(gdb) run
Breakpoint 4, main(...) at 36
36     c2 = 0x00FFFFFF;
(gdb) next
37     strset((char*)&c2,'z');
(gdb) print /x c2
$1 = 0xffffffff
(gdb) x /4xb &c2           // confirm little-endian
0x55558014 <c2>:  0xff 0xff 0xff 0x00
(gdb) next
38     return c1+c2;
(gdb) x /4xb &c2
0x55558014 <c2>:  0x7a 0x7a 0x7a 0x00
(gdb) print /x c2
$2 = 0x7a7a7a
(gdb)
```

← **c2 a été vérolée**  
**'z' = 0x7a**

# Exemple de données corrompues – la solution...

47

- Un exemple plus réel...
  - Utilisons un watchpoint
  - Pour voir où la variable c2 est vérolée

```
20 int strset(char *s, char c) {  
21     int count=0;  
22     while (*s != '\0') {  
23         *s = c;  
24         s = s + 1;  
25         count = count+1;  
26     }  
27     return count;  
28 }
```


```
30 int c1,c2;  
31  
32 int main(int argc, char** argv) {  
33     char s1[] ="Hello Fred";  
34     char s2[] ="Hello Anaïs";  
35     c1 = strlen(s1)+strlen(s2);  
36     c2 = 0x00FFFFFF;  
37     strset((char*)&c2, 'z');  
38     return c1+c2;  
39 }
```

```
(gdb) run  
Breakpoint 4, main(...) at 36  
36     c2 = 0x00FFFFFF;  
(gdb) next  
37     strset((char*)&c2,'z');  
(gdb) p /x c2  
$1 = 0xffffffff  
(gdb) x /4xb &c2  
0x55558014 <c2>:  0xff 0xff 0xff 0x00  
(gdb) watch c2  
Hardware watchpoint 2: c2  
(gdb) cont  
Continuing.
```

# Exemple de données corrompues – la solution...

48

- Un exemple plus réel...
  - Utilisons un watchpoint
  - Pour voir où la variable c2 est vérolée



```
20 int strset(char *s, char c) {
21     int count=0;
22     while (*s != '\0') {
23         *s = c;
24         s = s + 1;
25         count = count+1;
26     }
27     return count;
28 }
```

```
30 int c1,c2;
31
32 int main(int argc, char** argv) {
33     char s1[] ="Hello Fred";
34     char s2[] ="Hello Anaïs";
35     c1 = strlen(s1)+strlen(s2);
36     c2 = 0x00FFFFFF;
37     strset((char*)&b>c2, 'z');
38     return c1+c2;
39 }
```

```
(gdb) run
Breakpoint 4, main(...) at 36
36     c2 = 0x00FFFFFF;
(gdb) next
37     strset((char*)&c2,'z');
(gdb) p /x c2
$1 = 0xffffffff
(gdb) x /4xb &c2
0x55558014 <c2>:  0xff 0xff 0xff 0x00
(gdb) watch c2
Hardware watchpoint 2: c2
(gdb) cont
Continuing.
```

```
Hardware watchpoint 2: c2
Old value = 16777215
New value = 16777082
24     s=s+1;
(gdb) p /x c2
$3 = 0xffff7a // little endian: so first byte is 0x7a
(gdb) p /x 'z'
$4 = 0x7a
```



# Exécution vue par le matériel

49

## Où se trouve l'exécution ?

- ✓ - Où dans le code ?
  - ✓ - Où dans les données ?
- } **Et les pointeurs de fonction ?**

### Processor :

➔ **Fetch** instruction @pc


**Decode** instruction

**Advance pc**

pc = pc + sizeof(instruction)

**Execute** instruction

**Loop over**



0x10000	mov r0, #0x24
0x10004	ldrb r1, [r0]
0x10008	mov r2, #0x28
0x1000C	str r1, [r2]
0x10010	mov pc, r3

### Processor

➔ pc	0x10010
r0	0x24
r1	0x12345678
r2	0x28
r3	0x10000

### Memory:

	0x24	0x28		0x10000
	0x12345678	0x00000000		code

- L'appel de fonction via pointeur de fonction

```
int count(char *s, bool (*valid)(char)) {  
    int count=0;  
    while (*s != '\0') {  
        if (valid(*s))  
            count = count+1;  
        s = s + 1;  
    }  
    return count;  
}
```

Calcule le nombre de caractères valides dans une chaîne de caractères, en utilisant un **pointeur de fonction** pour décider si un caractère est valide ou pas.

Qu'est-ce qu'un pointer de fonction ?  
C'est un pointeur dont l'adresse est l'adresse de la première instructions du corps de la Fonction.

C'est aussi un pointeur que l'on utilise pour faire un appel de fonction et non pas lire ou écrire dans la mémoire.

# Exécution – Pointeurs de fonction – Exemple

51

- L'appel de fonction via pointeur de fonction

```
int count(char *s, bool (*valid)(char)) {  
    int count=0;  
    while (*s != '\0') {  
        → if (valid(*s))  
            count = count+1;  
        s = s + 1;  
    }  
    return count;  
}
```

Deux appels, avec deux pointeurs  
de fonctions différents

```
bool isLetter(char c);  
bool isDigit(char c);
```

```
char* s1 = "Hello";  
char* s2 = "World";
```

```
int length=count(s1,isLetter)  
            + count(s2,isDigit);
```

# Exécution – Pointeurs de fonction – Exemple

52

Nous sommes dans le **premier** appel à « count »,  
avec le pointeur sur la fonction « isLetter »

```
void main(void) {  
    char* s1 = "Hello";  
    char* s2 = "World";  
1 → int length=count(s1,isLetter)  
        + count(s2,isDigit);  
}
```

```
2 → int count(char *s, bool (*valid)(char)) {  
    int count=0;  
    while (*s != '\0') {  
        if (valid(*s))  
            count = count+1;  
        s = s + 1;  
    }  
    return count;  
}
```

```
3 bool isLetter(char c)  
    (c>='a' && c<='z')
```

```
2 int count(s,valid)  
    valid(*s)
```

```
1 main(void)  
    count(s1,isLetter)
```

Call Stack / Pile d'appels

```
3 → bool isLetter(char c) {  
    return (c>='a' && c<='z') ||  
           (c>='A' && c<='Z');  
}  
bool isDigit(char c) {  
    return (c>='0' && c<='9');  
}
```

# Exécution – Pointeurs de fonction – Exemple

53

Nous sommes dans le **second** appel à « count », avec le pointeur sur la fonction « isSymbol »

```
void main(void) {  
    char* s1 = "Hello";  
    char* s2 = "World";  
1 → int length=count(s1,isLetter)  
        + count(s2,isDigit);  
}
```

```
2 → int count(char *s, bool (*valid)(char)) {  
    int count=0;  
    while (*s != '\0') {  
        if (valid(*s))  
            count = count+1;  
        s = s + 1;  
    }  
    return count;  
}
```

```
3 bool isDigit(char c)  
    (c>='0' && c<='9')
```

```
2 int count(s,valid)  
    valid(*s)
```

```
1 main(void)  
    count(s1,isLetter)
```

Call Stack / Pile d'appels

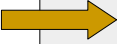
```
bool isLetter(char c) {  
    return (c>='a' && c<='z') ||  
           (c>='A' && c<='Z');  
}  
3 → bool isDigit(char c) {  
    return (c>='0' && c<='9');  
}
```

# Debug – Pointeurs de fonction & Points d'arrêt


54

```
char* s1 = "Hello";
char* s2 = "World";

int length=count(s1,isLetter)
    + count(s2,isDigit);
```



```
64 int count(char *s, bool (*valid)(char)) {
65     int count=0;
66     while (*s != '\0') {
67         if (valid(*s))
68             count = count+1;
69         s = s + 1;
70     }
71     return count;
72 }
```



File: count.c

```
bool isLetter(char c) {
    return c>='a' && c<='z' ||
           c>='A' && c<='Z';
}
bool isDigit(char c) {
    return (c>='0' && c<='9');
}
```

```
(gdb) br count
Breakpoint 4 ... at count.c: 64
(gdb) condition 4 (valid==isDigit)
(gdb) run
Breakpoint 4,
    count (s=0x555600a "World",
           valid=0x555551a8 <isDigit>)
    at count.c:65
65     int count=0;
```

# Utilité des pointeurs de fonction & Points d'arrêt ?

55

```
char* s1 = "Hello";
char* s2 = "World";

int length=count(s1,isLetter)
    + count(s2,isDigit);
```

```
64 int count(char *s, bool (*valid)(char)) {
65     int count=0;
66     while (*s != '\0') {
67         if (valid(*s))
68             count = count+1;
69         s = s + 1;
70     }
71     return count;
72 }
```

File: count.c

```
bool isLetter(char c) {
    return c>='a' && c<='z' ||
           c>='A' && c<='Z';
}
bool isSymbol(char c) {
    return !isLetter(c);
}
```

```
(gdb) br count
Breakpoint 4 ... at count.c: 64
(gdb) condition 4 (valid==isDigit)
(gdb) run
Breakpoint 4,
    count (s=0x555600a "World",
          valid=0x555551a8 <isDigit>)
    at count.c:65
65     int count=0;
```

Pourquoi utiliser un point d'arrêt dans « count » avec une condition sur le pointeur de fonction passé en argument ?

Pourquoi ne pas utiliser un point d'arrêt dans la fonction « isSymbol » ?

# Debug – Pointeurs de fonction & Points d'arrêt

56

```
char* s1 = "Hello";
char* s2 = "World";

int length=count(s1,isLetter)
➔      + count(s2,isDigit);
```

```
➔ 64 int count(char *s, bool (*valid)(char)) {
65     int count=0;
66     while (*s != '\0') {
67         if (valid(*s))
68             count = count+1;
69         s = s + 1;
70     }
71     return count;
72 }
```

File: count.c

```
bool isLetter(char c) {
    return c>='a' && c<='z' ||
           c>='A' && c<='Z';
}
bool isSymbol(char c) {
    return !isLetter(c);
}
```

```
(gdb) br count
Breakpoint 4 ... at count.c: 64
(gdb) condition 4 (valid==isDigit)
(gdb) run
Breakpoint 4,
      count (s=0x555600a "World",
            valid=0x555551a8 <isDigit>)
at count.c:65
65     int count=0;
```

Si on utilise un point d'arrêt dans la fonction « isSymbol », l'exécution va s'arrêter dans tous les appels, pas seulement ceux depuis la fonction « count ».



- Comment travailler pour réussir ?
  - Il faut reprendre les transparents
  - Il faut donc poser les questions nécessaires pour bien comprendre
  - Il faut aussi faire, mettre en pratique
  - Comprendre ne sera pas suffisant
  - Il faudra y revenir pour mémoriser sur le long terme
- Rappelez-vous
  - Le but n'est pas simplement d'avoir une note
  - Mais d'apprendre un savoir-faire pérenne