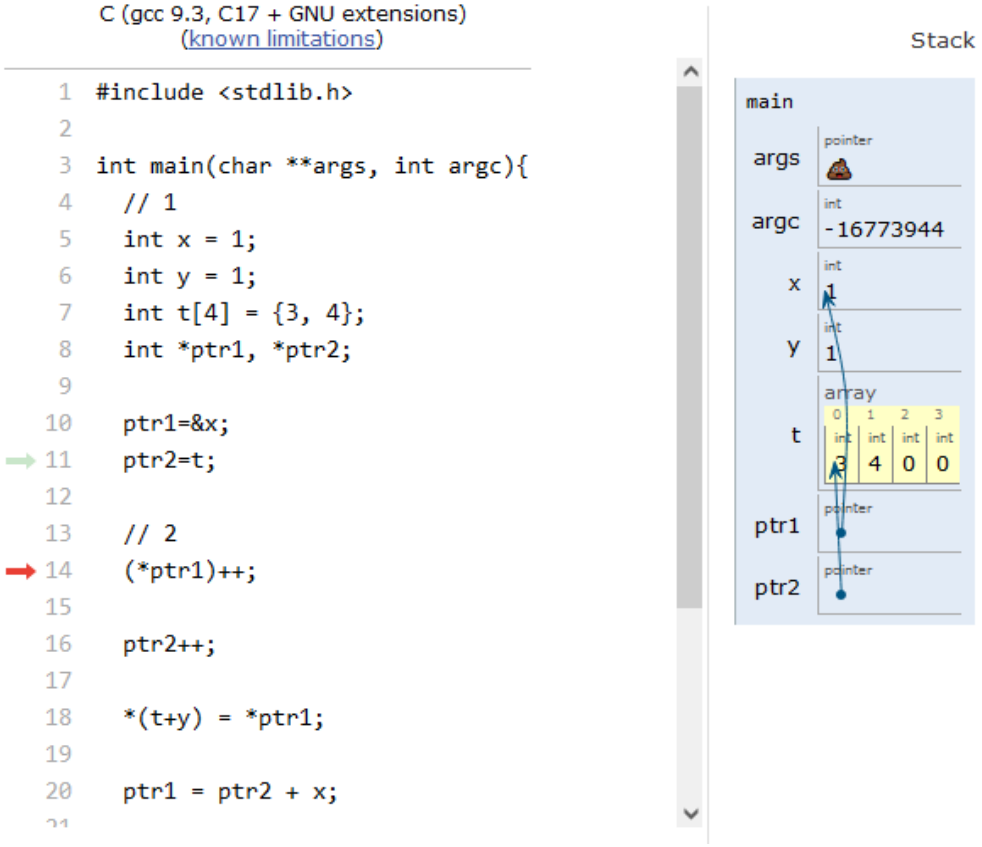
I2010 : langage C (TP3)

1. Pointeurs et gestion dynamique de la mémoire

Pour visualiser le code avec *PythonTutor.com*, exécutez la page suivante : https://tinyurl.com/2nfs9934

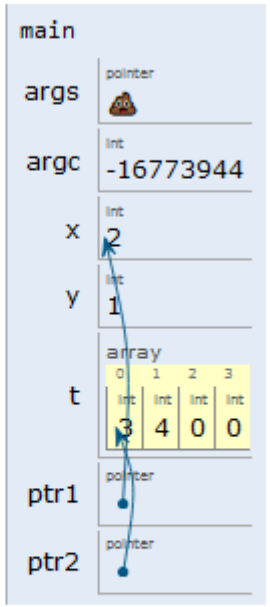
Vert : la dernière instruction exécutée
Rouge : la prochaine instruction à exécuter



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
1  #include <stdlib.h>
2
3  int main(char **args, int argc){
4    // 1
5    int x = 1;
6    int y = 1;
7    int t[4] = {3, 4};
8    int *ptr1, *ptr2;
9
10   ptr1=&x;
11   ptr2=t;
12
13   // 2
14   (*ptr1)++;
15
16   ptr2++;
17
18   *(t+y) = *ptr1;
19
20   ptr1 = ptr2 + x;
```

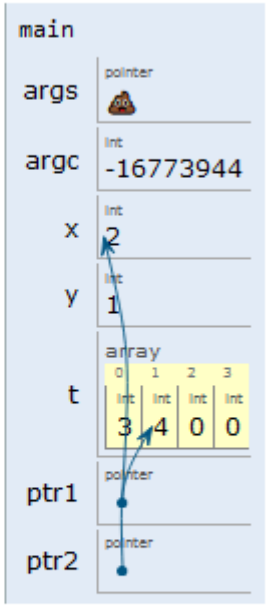
Stack



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
1  #include <stdlib.h>
2
3  int main(char **args, int argc){
4    // 1
5    int x = 1;
6    int y = 1;
7    int t[4] = {3, 4};
8    int *ptr1, *ptr2;
9
10   ptr1=&x;
11   ptr2=t;
12
13   // 2
14   (*ptr1)++;
15
16   ptr2++;
17
18   *(t+y) = *ptr1;
19
20   ptr1 = ptr2 + x;
```

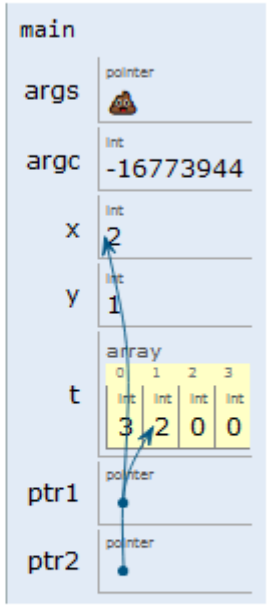
Stack



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
1  #include <stdlib.h>
2
3  int main(char **args, int argc){
4    // 1
5    int x = 1;
6    int y = 1;
7    int t[4] = {3, 4};
8    int *ptr1, *ptr2;
9
10   ptr1=&x;
11   ptr2=t;
12
13   // 2
14   (*ptr1)++;
15
16   ptr2++;
17
18   *(t+y) = *ptr1;
19
20   ptr1 = ptr2 + x;
```

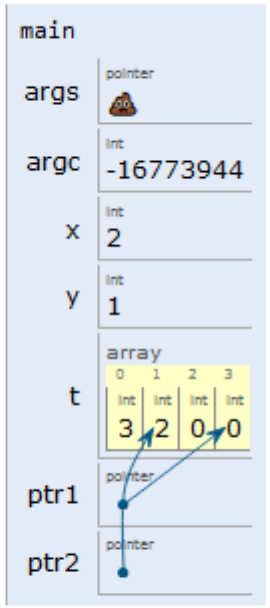
Stack



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
7    int t[4] = {3, 4};
8    int *ptr1, *ptr2;
9
10   ptr1=&x;
11   ptr2=t;
12
13   // 2
14   (*ptr1)++;
15
16   ptr2++;
17
18   *(t+y) = *ptr1;
19
20   ptr1 = ptr2 + x;
21
22   ptr1 = &(t[x+1]);
23
24   y = (*ptr1)++;
25
26   x = ptr1-t;
27 }
```

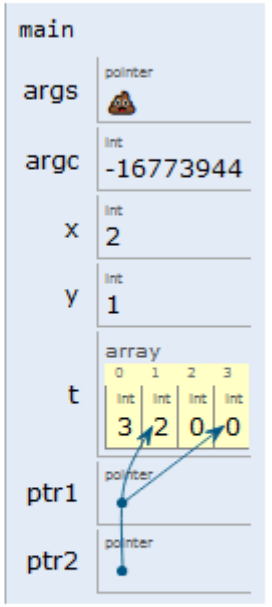
Stack



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
7    int t[4] = {3, 4};
8    int *ptr1, *ptr2;
9
10   ptr1=&x;
11   ptr2=t;
12
13   // 2
14   (*ptr1)++;
15
16   ptr2++;
17
18   *(t+y) = *ptr1;
19
20   ptr1 = ptr2 + x;
21
22   ptr1 = &(t[x+1]);
23
24   y = (*ptr1)++;
25
26   x = ptr1-t;
27 }
```

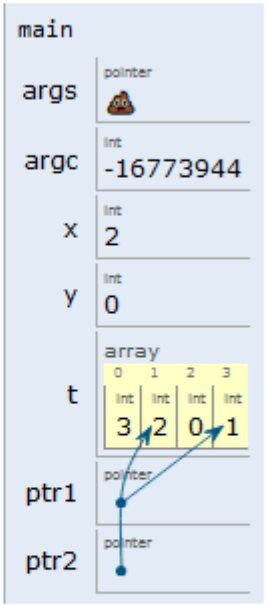
Stack



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
7    int t[4] = {3, 4};
8    int *ptr1, *ptr2;
9
10   ptr1=&x;
11   ptr2=t;
12
13   // 2
14   (*ptr1)++;
15
16   ptr2++;
17
18   *(t+y) = *ptr1;
19
20   ptr1 = ptr2 + x;
21
22   ptr1 = &(t[x+1]);
23
24   y = (*ptr1)++;
25
26   x = ptr1-t;
27 }
```

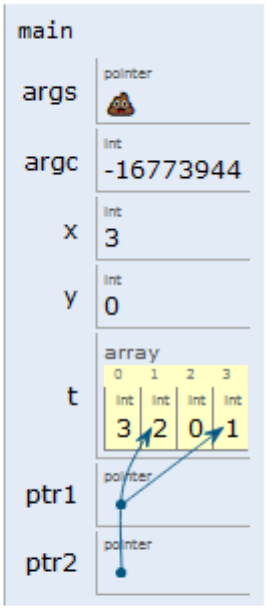
Stack



C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
7   int t[4] = {3, 4};
8   int *ptr1, *ptr2;
9
10  ptr1=&x;
11  ptr2=t;
12
13  // 2
14  (*ptr1)++;
15
16  ptr2++;
17
18  *(t+y) = *ptr1;
19
20  ptr1 = ptr2 + x;
21
22  ptr1 = &(t[x+1]);
23
24  y = (*ptr1)++;
25
26  x = ptr1-t;
27 }
```

Stack



4. Exercices d'observation et debugging

- ***to_debug_stack_1.c***
  → on lit en dehors de la plage du tableau (*buffer overread*) → *garbage value*
  → aucune erreur signalée, ni à la compilation, ni à l'exécution → gdb n'est d'aucune aide

- ***to_debug_stack_smashing.c***
  → on écrit en dehors de la plage du tableau (*buffer overflow*) → *stack smashing*[1]
  → cette erreur est détectée par gcc à la fin de l'exécution du programme[2] → gdb n'est d'aucune aide

- ***to_debug_segmentation_fault_1.c***
  → on n'alloue pas le tableau et on tente d'accéder à l'adresse 0x0 (*NULL mistake*)
  → gdb permet d'identifier quelle ligne du programme a provoqué une SEGFAULT

- ***to_debug_segmentation_fault_2.c***
  → on parcourt le tableau à l'envers en décrémentant i → on n'arrête jamais → boucle infinie
  → gdb permet de signaler quand `t[i]` provoque une SEGFAULT en sortant de la mémoire virtuelle du processus

- ***to_debug_stack_smashing_2.c***
  → on parcourt t2 comme des *double* alors que le tableau est alloué comme *int* → écriture en dehors des limites du tableau (*buffer overflow*)
  → cette erreur est détectée par gcc à la fin de l'exécution du programme → gdb n'est d'aucune aide

- ***to_debug_doublette.c***
  → on libère deux fois la même zone de mémoire dynamique : tab1 (*double free*) → l'exécution peut varier : rien ne se passe (sous Ubuntu) ou execution aborted (sous WSL) ; cf. `man 3 free` : « if free(ptr) has already been called before, undefined behavior occurs. »
  → gdb permet d'identifier la ligne du programme qui provoque l'arrêt prématuré du programme

---

[1] Voir mécanisme de sécurité informatique *Stack Smashing Protection (SSP)*: https://www.arsouyes.org/blog/2019/57_Smashing_the_Stack_2020

[2] Remarque : Le mécanisme de *Stack Smashing Protection* ne sera pas activé si une instruction `exit(code)` interrompt l'exécution de votre programme. En effet, `exit` renverra le status code « `code` » au bash avant que la vérification de la stack ne soit réalisée (et renvoie éventuellement le code d'erreur 132 en cas de détection de corruption de la stack). Conseil : ne pas ajouter d'instruction `exit(0)` ou `exit(EXIT_SUCCESS)` dans un programme si celle-ci n'est pas indispensable.