

TP01

Fait par: Robin Viollet et Yaël Tramier

Ex1

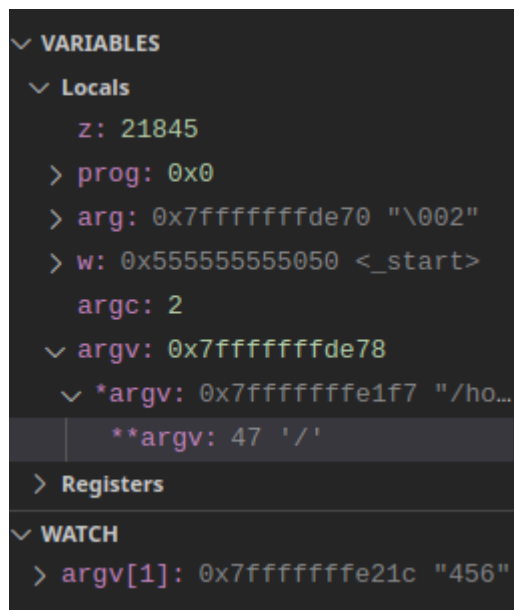
- Regardez le contenu de argc et argv. Est-ce cohérent avec le contenu de launch.json ?

argc vaut 2, ce qui est juste puisque 2 arguments ont été passés: le nom du programme et 456

```
"name": "(gdb) Launch debugSimple",  
"type": "cppdbg",  
"request": "launch",  
"program": "${workspaceFolder}/debugSimple",  
"args": ["456"],
```

- Quel est le lien entre le caractère / et 47 ?

47 est la valeur ascii de /



The screenshot shows a debugger's variable window with the following content:

```
✓ VARIABLES  
  ✓ Locals  
    z: 21845  
    > prog: 0x0  
    > arg: 0x7fffffffde70 "\002"  
    > w: 0x555555555050 <_start>  
    argc: 2  
    ✓ argv: 0x7fffffffde78  
      ✓ *argv: 0x7fffffffe1f7 "/ho...  
        **argv: 47 '/'  
    > Registers  
  ✓ WATCH  
    > argv[1]: 0x7fffffffe21c "456"
```

Ex2

- Avancez pas à pas jusqu'à la ligne 11. Observez les valeurs de prog et arg. Notez qu'une info bulle affiche leur contenu quand vous passez sur la variable dans l'éditeur.

prog pointe vers le premier argument: le nom du programme

arg pointe vers le deuxième argument: 456

```
✓ VARIABLES
  ✓ Locals
    z: 2
    ✓ prog: 0x7fffffffef1f7 "/home...
      *prog: 47 '/'
    ✓ arg: 0x7fffffffef21c "456"
      *arg: 52 '4'
    > w: 0x7fffffffdd64
      argc: 2
    ✓ argv: 0x7fffffffde78
      ✓ *argv: 0x7fffffffef1f7 "/ho...
```

Ex3

- On remarque que `*w = 2` ainsi que `z`, ce qui est normal puisque `*w=z`. On aimerait voir le contenu de `&w`. On peut pour cela ajouter un espion. Cliquer + dans la zone ESPION et tapez `&w`. Dépliez la valeur de `&w`. Comprenez-vous les relations entre : `&w`, `*&w` et `**&w`

`&w`: adresse de `w`

`*&w`: valeur de `w`

`**&w`: valeur de `z`

```
✓ WATCH
  > argv[1]: 0x7fffffffef21c "456"
  ✓ &w: 0x7fffffffdd68
  ✓ *&w: 0x7fffffffdd64
  **&w: 2
```

Ex4

```
C debugSimple.c > main(int, char * [])
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int z =  argc;
7      char *prog = argv[0];
8      char *arg = argv[1];
9
10     int *w = &z;
11
12     printf("bonjour\n");
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

bonjour

Ex5

RUN ... (gdb) Launch ...

debugSimple.c x launch.json

debugSimple.c > main(int, char * [])

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int z =  argc;
7      char *prog = argv[0];
8      char *arg = argv[1];
9
10     int *w = &z;
11
12     printf("bonjour\n");
13 }
```

VARIABLES

- Locals
 - z: 2
 - > prog: 0x7fffffffef1f7 "/home..."
 - > arg: 0x7fffffffef21c "456"
 - > w: 0x555555555050 <_start>
 - argc: 2
 - > argv: 0x7fffffffefde78
- Registers

WATCH

- > argv[1]: 0x7fffffffef21c "456"
- > &w: 0x7fffffffdd68
- > *w: 0x555555555050 <_start>

Ex7

rax	0x7fffffffef21c	140737488347676
rbx	0x0	0
rcx	0x7ffff7fa6718	140737353770776
rdx	0x7fffffffde90	140737488346768
rsi	0x7fffffffde78	140737488346744
rdi	0x2	2
rbp	0x7fffffffdd80	0x7fffffffdd80
rsp	0x7fffffffdd50	0x7fffffffdd50
r8	0x0	0
r9	0x7ffff7fe21b0	140737354015152
r10	0x3	3
r11	0x2	2
r12	0x55555555050	93824992235600
r13	0x0	0
r14	0x0	0
r15	0x0	0
rip	0x55555555161	0x55555555161 <main+44>
eflags	0x206	[PF IF]
cs	0x33	51
ss	0x2b	43
ds	0x0	0
es	0x0	0
fs	0x0	0
gs	0x0	0

Ex8

3. Tracez pas à pas jusqu'à avoir exécuté l'instruction `char *ci=tab;`. Que déduisez-vous de l'organisation mémoire ? Comment sont placés les octets à l'intérieur d'un int ?

Les 4 octets qui composent un int sont stockés séquentiellement dans la mémoire.

4. Y a-t-il une différence entre `*ci@16` et `*si@8` ?

`*ci@16` et `*si@8` indiquent la même valeur (12) mais ne l'obtiennent pas de la même manière. `*ci@16` interprète la valeur comme un char (1 octet) alors que `*si@8` l'interprète comme un short (2 octets). Comme 12 tient sur un seul octet, ils obtiennent la même valeur.

6. Tracez pas à pas jusqu'à la fin du programme et observez bien ii, si et ci. Pour quelle raison ii++ et si++ n'ont pas le même effet sur la valeur de ii et si ? De même, pourquoi ii++ et ci++ n'ont pas le même effet sur la valeur de ii et ci ?

`ii` est de type `int*`, `ii++` fait avancer le pointeur de 4 octets.
`si` est de type `short*`, `si++` fait avancer le pointeur de 2 octets.
`ci` est de type `char*`, `ci++` fait avancer le pointeur de 1 octet.

Ex9

- Tentez de découvrir quel est le bug de ce programme.

La boucle for itère 11 fois sur un tableau de 10 éléments. Pour corriger le bug, changer la condition à `i < 10`.

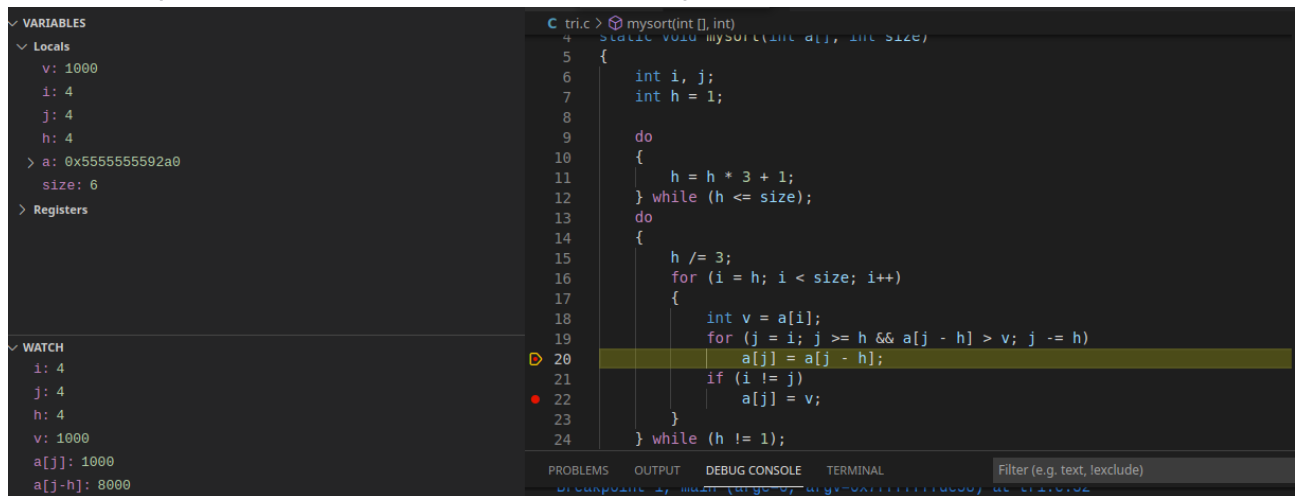
Ex10

- Quel algorithme de tri implémente la fonction `mysort` ?

Shellsort ?

Ex11

- Placer des points d'arrêts et des observateurs d'expression.



Ex12

- Corrigez maintenant le problème et recompilez avec `ctrl-shift-b`.

```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

-   mysort(a, argc);
+   mysort(a, argc - 1);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
}
```

```
    free(a);  
    return 0;  
}
```

Ex13

- Dans les programmes fournis dans l'archive, vous avez le programme `list.c`. Tentez de le déboguer le plus rapidement possible.

```
void InsertAtTail(int x) {  
-   struct Node* temp = head;  
    struct Node* newNode = GetNewNode(x);  
+   if (head == NULL){  
+       head = newNode;  
+       return;  
+   }  
+   struct Node* temp = head;  
    while(temp->next != NULL) temp = temp->next; // Go To last Node  
    temp->next = newNode;  
    newNode->prev = temp;  
}
```

Ex14

- Exécutez votre programme de tri corrigé avec ltrace. Quelles est (sont) la (les) bibliothèque(s) partagées que votre programme utilise. Quelles sont les fonctions de cette (ces) bibliothèque(s) qui sont utilisées ?

Bibliothèques appelées :

- `stdlib.h`
- `stdio.h`

Fonctions appelées :

- `malloc()`
- `atoi()`
- `printf()`
- `putchar()`
- `free()`

Ex15

- Utilisez le programme strace pour trouver où se trouve(nt) la (les) bibliothèque(s) qui sont chargées.

Ex16

- Problème #1 :

La boucle `while` dans `rechercheBinaire` est infinie si la valeur recherchée n'est pas présente dans le tableau.

```
int rechercheBinaire(int tab[],int x,int i, int j){
    int gau,droite,milieu;
    gau=i;droite=j;
-   while (gau<=droite) {
+   while (gau<droite-1) {
        milieu = (gau+droite)/2;
        if (tab[milieu]==x)
            return milieu;
        if (tab[milieu]>x)
            droite = milieu-1;
        else
            gau = milieu;
    }
    return -1;
}
```

- Problème #2 :

Si aucun argument n'est passé au programme, une erreur de segmentation apparaît.

```
int main(int argc, char *argv[])
{
+   if (argc > 2){
        int *a;
        int i;

        // (int *) est un cast. indispensable ?
        a = (int *)malloc((argc - 2) * sizeof(int));

        for (i = 0; i < argc - 2; i++)
            a[i] = atoi(argv[i + 1]);

        int val = atoi(argv[i + 1]);

        int trouve = recherche(a, val, argc - 1);
        if (trouve!=-1)
```

```
        printf("la valeur %i est à l'indice %i",val,trouve);
    else
        printf("valeur %i pas trouvée",val);
    free(a);
    return 0;
+ }
+ printf("usage: %s n... v\n", *argv);
+ printf("n being one or many intergers separated");
+ printf(" by a space and v the integer for which to find");
+ printf(" the index in the previous list.");
+ return 1;
}
```