HE2B ESI

MICL : TD04 : Variables globales

ABS – BEJ – DBO – HAL – NVS – SRE – YVO *

Année académique 2021 – 2022

Dans ce TD, la notion de variable globale et sa mise en œuvre avec `nasm` sont étudiées. Nous voyons comment déclarer, initialiser et utiliser les types de données classiques nombres et caractères.

1 Définition

Les variables globales [1] sont des variables généralement déclarées au début du code source assembleur et utilisables ensuite dans *tout* le fichier source. C'est de cette utilisation *globale* dans tout le source suivant qu'elles tirent leur nom. Cette caractéristique est à mettre en opposition aux variables *locales* [2] (à une fonction) étudiées le TD07.

Dans le cours d'algorithmique, les variables globales sont tues. En Java [3], cette notion n'existe pas. La raison en est qu'il est difficile de résoudre une erreur liée à une variable globale puisque celle-ci peut avoir été modifiée presque partout dans le programme qui l'utilise. Nous essayons dans les labos microprocesseur d'en limiter l'usage aux emplois légitimes, mais cela n'est pas toujours le cas. La surcharge de travail et de difficultés pour utiliser des variables locales en langage d'assemblage l'expliquent.

*Et aussi, lors des années passées : DHA – DWI – EGR – ELV – FPL – JDS – MBA – MCD – MHI – MWA.

1. https://fr.wikipedia.org/wiki/Variable_globale (consulté le 17 février 2020).
2. https://fr.wikipedia.org/wiki/Variable_locale (consulté le 17 février 2020).
3. https://stackoverflow.com/a/4646605 (consulté le 17 février 2020).

2 Sections dédiées aux variables

Un fichier binaire exécutable au format `elf` [4] est divisé en plusieurs sections. Avec `nasm`, la directive `section` [5] permet de les définir dans le code source du programme. Jusqu'à présent, nous n'avons utilisé que la section `.text`. Elle contient les instructions exécutables du programme. Les sections `elf` standard [6] pour les variables globales sont au nombre de trois : `.data`, `.rodata` et `.bss`.

Un point commun partagé par les variables qui résident dans ces trois sections est que leur durée de vie égale celle de leur programme. Elles sont créées en mémoire lors du chargement de l'exécutable et y résident jusqu'à sa fin.

Les sections `.data` et `.rodata` servent aux variables globales explicitement initialisées dans le code source. La section `.bss` est utilisée pour les variables implicitement initialisées à zéro au démarrage du programme.

Une différence essentielle entre variables des sections `.data` ou `.rodata` et `.bss` est que les premières existent *telles quelles* dans le *fichier exécutable*, tandis que les secondes n'y apparaissent que sous la forme de directives de création. Toutes sont créées *en mémoire* au démarrage du programme. Les premières sont initialisées avec les valeurs qu'elles ont dans le fichier exécutable, les secondes sont mises à zéro.

La syntaxe de définition des variables diffère entre ces deux types de sections. Par contre, ce qui est commun aux deux, c'est que les variables en langage d'assemblage *ne sont pas typées* [7] [8]. La seule caractéristique à fournir lors de la définition d'une variable est sa *taille*. Une valeur initiale est en outre exigée dans les sections `.data` et `.rodata`.

Les variables définies dans ces trois sections sont placées en mémoire à des endroits différents. Par contre, au sein de chaque section, les variables sont rangées en mémoire dans l'ordre de leurs définitions dans le code source, à la queue leu leu, sans trou entre elles, sauf mention contraire [9] [10].

La table Table 1 résume les quatre sections que nous utilisons dans les codes source des laboratoires microprocesseur.

2.1 Sections `.data` et `.rodata`

Les variables définies dans la section `.data` d'un programme peuvent être modifiées au cours de son exécution. Celles définies dans sa section `.rodata` sont constantes [11]

4. https://devarea.developpez.com/linux-processus-construction/ (consulté le 17 février 2020).
5. https://www.nasm.us/doc/nasmdoc6.html#section-6.3 (consulté le 17 février 2020).
6. https://www.tortall.net/projects/yasm/manual/html/objfmt-elf-section.html#elf-standard-sections (consulté le 17 février 2020).
7. https://fr.wikipedia.org/wiki/Type_(informatique) (consulté le 17 février 2020).
8. Ce sont les traitements qu'on applique aux variables, ou aux données en général, qui leur donnent une signification, donc, entre autre, un type.
9. https://www.nasm.us/doc/nasmdoc4.html#section-4.11.13 (consulté le 17 février 2020).
10. Cela n'est jamais le cas lors des labo micro.
11. https://fr.wikipedia.org/wiki/Variable_(informatique)#Variables_et_constantes (consulté le 17 février 2020).

Nom	Rôle
`.text`	instructions exécutables du programme
`.data`	variables globales explicitement initialisées
`.rodata`	constantes globales explicitement initialisées
`.bss`	variables globales implicitement initialisées à 0

TABLE 1 – Sections utilisées lors des laboratoires microprocesseur.

Taille (en *bytes*)	Pseudo-instruction	Signification
1	`DB`	***D**efine **B**yte*
2	`DW`	***D**efine **W**ord*
4	`DD`	***D**efine **D**oubleword*
8	`DQ`	***D**efine **Q**uadword*

TABLE 2 – Pseudo-instructions pour la déclaration de variables initialisées.

(*read only*). Une tentative de les modifier provoque une *erreur lors de l'exécution* du programme.

Les variables des sections `.data` et `.rodata` doivent être initialisées. Le nom d'une variable est une étiquette [12]. Il n'est cependant pas nécessaire de mettre deux points (:) pour indiquer à l'assembleur qu'il s'agit d'une étiquette.

La table TABLE 2 reprend certaines des pseudo-instructions `nasm` [13] [14] pour définir des variables dans les sections `.data` et `.rodata`. Dans le code source, ces directives sont suivies par une valeur initiale [15] donnée à la variable.

Exemple Voici un code source montrant la mise en œuvre de ces pseudo-instructions :

```
1  section .data
2      i1      DB      -1                  ; 1 byte initialisé
3      i2      DW      23                  ; 2 bytes initialisés
4      i4      DD      -1                  ; 4 bytes initialisés
5
6      i8      DQ      0x80_00_00_00_00    ; 8 bytes initialisés
7                                          ; 0x00_00_00_80_00_00_00_00
8
9  section .rodata
10     ci8     DQ      93229
11
```

12. https://en.wikipedia.org/wiki/Label_(computer_science) (consulté le 17 février 2020).
13. https://www.nasm.us/doc/nasmdoc3.html#section-3.2.1 (consulté le 17 février 2020).
14. Il ne s'agit pas d'instructions à destination du microprocesseur, mais de directives pour l'assembleur.
15. Ou plusieurs dans le cas d'un tableau comme nous le voyons dans le TD06.

Taille (en *bytes*)	Pseudo-instruction	Signification
1	RESB	***Res**erve **B**yte*
2	RESW	***Res**erve **W**ord*
4	RESD	***Res**erve **D**oubleword*
8	RESQ	***Res**erve **Q**uadword*

TABLE 3 – Pseudo-instructions pour la déclaration de variables non initialisées.

```
12 section .text
13 ; ...
14
```

2.2 Section `.bss`

Les variables de la section `.bss` ne sont pas littéralement présentes dans le fichier exécutable. Elles sont créées et mises à zéro lors du démarrage du programme.

La table TABLE 3 reprend certaines des pseudo-instructions `nasm`[16] pour définir des variables dans la section `.bss`. Dans le code source, ces directives sont suivies d'un entier pour indiquer combien de *bytes*, *words*, *double words* ou *quad words* réserver au démarrage du programme. Ceci est très utile pour la définition de tableaux. Ces derniers sont simplement évoqués dans ce TD. On y revient en détail dans le TD06.

Exemple Voici un exemple de code source où sont mises en œuvre ces pseudo-instructions :

```
1 section .bss
2     ; tout est créé et initialisé à 0
3     x1      RESB    10      ; 10 × 1 byte réservés
4     x2      RESW    6       ; 6 × 2 bytes réservés
5     x4      RESD    100     ; 100 × 4 bytes réservés
6     x8      RESQ    2       ; 2 × 8 bytes réservés
7
```

3 Accès à une variable

3.1 Adresse et déréférencement

Le nom d'une variable est une *étiquette* (un *label* en anglais). Son utilisation dans un code source est remplacée par `nasm` par l'*adresse* de l'emplacement où l'étiquette est placée. Cela correspond à l'adresse de la variable lorsqu'on est dans les sections `.data`,

16. https://www.nasm.us/doc/nasmdoc3.html#section-3.2.2 (consulté le 17 février 2020).

`.rodata` ou `.bss`. Rappelons que nous travaillons en 64 bits. Les adresses de variables ont donc toujours cette taille, soit 8 *bytes*.

Pour accéder au *contenu* de la variable, il faut réaliser un *déréférencement*[17] : atteindre ce qui se trouve à l'adresse de la variable. Pour indiquer à `nasm` qu'on désire déréférencer un *pointeur*[18] [19], on place l'adresse entre crochets[20] (`[ ]`).

Exemple Voici un extrait de source qui illustre l'accès au contenu d'une variable :

```
1  section .data
2      i4  DD   42          ; entier sur 4 bytes
3
4  section .text
5      ; ...
6      mov rax, i4          ; rax <-- _adresse_ (8 bytes) de i4
7
8      mov ebx, [i4]        ; ebx <-- _contenu_ de ce qui se trouve
9      ; à l'adresse i4 et s'étend sur 4 bytes (car ebx), soit 42
10
11     mov ecx, [rax]       ; rax contient l'adresse de i4, donc
12     ; [rax] = [i4] et ici ecx <-- _contenu_ de ce qui se trouve
13     ; à l'adresse rax = i4 et s'étend sur 4 bytes (car ecx), soit 42
```

3.2 Boutisme

Lorsqu'il s'agit de stocker en mémoire une donnée s'étendant sur plusieurs *bytes*, il existe essentiellement deux conventions d'ordonnancement des *bytes*[21] :

— gros boutisme[22] (*big endian*) : le *byte* de *rang le plus élevé* est stocké à l'*adresse la plus petite* ;

— petit boutisme[23] (*little endian*) : le *byte* de *rang le plus petit* est stocké à l'*adresse la plus petite*.

L'architecture x86 adopte[24] le petit boutisme.

Exemple Voici un extrait de code source qui montre explicitement l'utilisation du petit-boutisme :

17. https://fr.wikibooks.org/wiki/Programmation_C-C%2B%2B/D%C3%A9r%C3%A9f%C3%A9rencement,_indirection (consulté le 17 février 2020).
18. https://fr.wikipedia.org/wiki/Pointeur_(programmation) (consulté le 17 février 2020).
19. Un pointeur est une mémoire qui contient une adresse.
20. https://www.nasm.us/doc/nasmdoc2.html#section-2.2.2 (consulté le 17 février 2020).
21. https://fr.wikipedia.org/wiki/Boutisme (consulté le 16 février 2020).
22. https://fr.wikipedia.org/wiki/Boutisme#Gros-boutisme (consulté le 16 février 2020).
23. https://fr.wikipedia.org/wiki/Boutisme#Petit-boutisme (consulté le 16 février 2020).
24. https://en.wikipedia.org/wiki/Endianness#Current_architectures (consulté le 17 février 2020).

```
1  section .data
2      vw      DW   0x0102
3      ; à l'adresse vw     : 0x02
4      ; à l'adresse vw + 1 : 0x01
5      ; ---> petites adresses --->
6      ;                      vw
7      ;           ... | ... | 0x02 | 0x01 | ... | ...
8      ;                                ---> grandes adresses --->
9
10     vq      DQ   0x1122334455667788
11     ; à l'adresse vq     : 0x88
12     ; à l'adresse vq + 1 : 0x77
13     ; à l'adresse vq + 2 : 0x66
14     ; à l'adresse vq + 3 : 0x55
15     ; à l'adresse vq + 4 : 0x44
16     ; à l'adresse vq + 5 : 0x33
17     ; à l'adresse vq + 6 : 0x22
18     ; à l'adresse vq + 7 : 0x11
19     ; ---> petites adresses --->
20     ;     vq
21     ; .. | 0x88 | 0x77 | 0x66 | 0x55 | 0x44 | 0x33 | 0x22 | 0x11 | ..
22     ;                                          ---> grandes adresses --->
23
24     ; vue complète de la section .data :
25     ; ---> petites adresses --->
26     ;     vw            vq
27     ; .. | 0x02 | 0x01 | 0x88 | 0x77 | 0x66 | 0x55 | 0x44 |
28     ; 0x33 | 0x22 | 0x11 | ..
29     ;                                          ---> grandes adresses --->
30
31 section .text
32     ; ...
33     mov r8b, [vw]        ; r8b <-- 0x02
34     mov r9b, [vw + 1]    ; r9b <-- 0x01
35     mov r10b, [vq + 1]   ; r10b <-- 0x77
36     mov r11b, [vq + 5]   ; r11b <-- 0x33
37     mov r12b, [vq + 7]   ; r12b <-- 0x11
38
39     ; pas de trou entre les variables :
40     ;       vw est directement suivie par vq
41     mov r13b, [vw + 2]   ; r13b <-- 0x88
42     mov r14b, [vw + 3]   ; r14b <-- 0x77
43     mov r15b, [vq - 1]   ; r15b <-- 0x01
```

Taille (en *bytes*)	Spécificateur
1	`byte`
2	`word`
4	`dword`
8	`qword`

TABLE 4 – Spécificateurs de taille de variables.

```
44      ; ...
45
```

3.3 Amnésie de `nasm`

Comme il a été mentionné en tout début de section 3.1, le nom d'une variable dans un code source `nasm` est une étiquette. La seule information à laquelle on accède en fournissant un nom de variable est l'adresse à laquelle cette étiquette a été placée. Aucune autre information n'est rendue disponible par `nasm`. Cette limitation amène quelques complications détaillées dans la suite de cette section.

3.3.1 Problème de taille

L'assembleur `nasm` ne retient pas [25] la taille des variables. Lorsqu'on accède au contenu d'une variable, le nombre de *bytes* déréférencés à partir de l'adresse entre crochets est déduit de la taille du second opérande, s'il existe et s'il ne s'agit pas d'un immédiat. Dans le cas contraire, il faut renseigner la taille de la donnée à l'aide d'un des spécificateurs de taille [26] de la table TABLE 4. Lorsque la taille du second opérande est déjà connue, il est également possible de préciser le spécificateur de taille. Ainsi le compilateur peut vérifier la compatibilité des deux opérandes et s'assurer qu'on ne commet pas d'erreur de taille.

Exemple Voici un exemple d'utilisation des spécificateurs de taille de variables :

```
1  ; taille.asm
2  global _start
3
4  section .data
5      i1      DB  -1          ; 1 byte initialisé
6      i2      DW  23          ; 2 bytes initialisés
7      i4      DD  -1          ; 4 bytes initialisés
8      i8      DQ  130_761_944 ; 8 bytes initialisés
9      ; contenu de la mémoire (byte par byte, hexadécimal) :
```

25. https://www.nasm.us/doc/nasmdoc2.html#section-2.2.3 (consulté le 17 février 2020).
26. https://www.nasm.us/doc/nasmdoc3.html#section-3.7 (consulté le 17 février 2020).

```
10      ; ---> petites adresses --->
11      ;       i1   i2       i4                  i8
12      ;  ... | FF | 17 | 00 | FF | FF | FF | FF | D8 | 44 | CB |
13      ;       07 | 00 | 00 | 00 | 00 | ...
14      ;                               ---> grande adresses --->
15  section .text
16  _start:
17      mov al, [i1]     ; al <-- 0xFF : ok
18      mov rax, [i1]    ; rax <-- 0xD8_FF_FF_FF_FF_00_17_FF : problème !
19      ; rem. : il s'agit d'une erreur logique, pas d'une erreur lors de
20      ;        l'assemblage ou de l'édition des liens, ni d'un plantage
21      ;        lors de l'exécution
22
23      mov [i4], eax    ; dword [i4] <-- 0xFF_00_17_FF : ok
24      ; contenu de la mémoire (byte par byte, hexadécimal) :
25      ; ---> petites adresses --->
26      ;       i1   i2       i4                  i8
27      ;  ... | FF | 17 | 00 | FF | 17 | 00 | FF | D8 | 44 | CB |
28      ;       07 | 00 | 00 | 00 | 00 | ...
29      ;                               ---> grande adresses --->
30      mov [i2], eax    ; dword [i2] <-- 0xFF_00_17_FF : problème !
31      ; contenu de la mémoire (byte par byte, hexadécimal) :
32      ; ---> petites adresses --->
33      ;       i1   i2       i4                  i8
34      ;  ... | FF | FF | 17 | 00 | FF | 00 | FF | D8 | 44 | CB |
35      ;       07 | 00 | 00 | 00 | 00 | ...
36      ;                               ---> grande adresses --->
37      mov dword [i2], eax ; on peut spécifier la taille de l'opérande
38  ;   mov  word [i2], eax ; impossible de mélanger 16 et 32 bits
39
40  ;   mov [i1], 12        ; error: operation size not specified
41                          ; car ni [i1], ni 12 n'ont de taille
42
43      mov byte [i1], 12    ; byte [i1] <-- 12 : ok
44      ; contenu de la mémoire (byte par byte, hexadécimal) :
45      ; ---> petites adresses --->
46      ;       i1   i2       i4                  i8
47      ;  ... | 0C | FF | 17 | 00 | FF | 00 | FF | D8 | 44 | CB |
48      ;       07 | 00 | 00 | 00 | 00 | ...
49      ;                               ---> grande adresses --->
50      mov word [i2], 42    ; word [i2] <-- 42 : ok
51      ; contenu de la mémoire (byte par byte, hexadécimal) :
52      ; ---> petites adresses --->
53      ;       i1   i2       i4                  i8
```

```
54      ;  ... | 0C | 2A | 00 | 00 | FF | 00 | FF | D8 | 44 | CB |
55      ;        07 | 00 | 00 | 00 | 00 | ...
56      ;                                 ---> grande adresses --->
57      mov dword [i4], 34  ; dword [i4] <-- 34 : ok
58      ; contenu de la mémoire (byte par byte, hexadécimal) :
59      ; ---> petites adresses --->
60      ;       i1   i2        i4                  i8
61      ;  ... | 0C | 2A | 00 | 22 | 00 | 00 | 00 | D8 | 44 | CB |
62      ;        07 | 00 | 00 | 00 | 00 | ...
63      ;                                 ---> grande adresses --->
64
65      mov qword [i8], 0x80_00_DD_FF   ; mov m64, imm64 n'existe pas !
66      ; on a : mov m64, imm32 où imm32 >> imm64 par extention de signe
67      ; contenu de la mémoire (byte par byte, hexadécimal) :
68      ; ---> petites adresses --->
69      ;       i1   i2        i4                  i8
70      ;  ... | 0C | 2A | 00 | 22 | 00 | 00 | 00 | FF | DD | 00 |
71      ;        80 | FF | FF | FF | FF | ...
72      ;                                 ---> grande adresses --->
73
74      mov qword [i8], 0x7F_00_DD_FF   ; mov m64, imm64 n'existe pas !
75      ; on a : mov m64, imm32 où imm32 >> imm64 par extention de signe
76      ; contenu de la mémoire (byte par byte, hexadécimal) :
77      ; ---> petites adresses --->
78      ;       i1   i2        i4                  i8
79      ;  ... | 0C | 2A | 00 | 22 | 00 | 00 | 00 | FF | DD | 00 |
80      ;        7F | 00 | 00 | 00 | 00 | ...
81      ;                                 ---> grande adresses --->
82
83      mov rax, 0x80_00_DD_FF  ; mov r64, imm64 existe
84      mov qword [i8], rax     ; mov m64, r64 existe, qword redondant
85      ; contenu de la mémoire (byte par byte, hexadécimal) :
86      ; ---> petites adresses --->
87      ;       i1   i2        i4                  i8
88      ;  ... | 0C | 2A | 00 | 22 | 00 | 00 | 00 | FF | DD | 00 |
89      ;        80 | 00 | 00 | 00 | 00 | ...
90      ;                                 ---> grande adresses --->
91
92  ;    mov rcx, dword [i8] ; error: mismatch in operand sizes
93                          ; car rcx fait 8 bytes et dword 4 bytes
94
95      mov rax, 60
96      mov rdi, 0
97      syscall
```

```
98
```

Point d'entrée Dans le code ci-dessus, l'étiquette `_start` sert de point d'entrée. Ce *label* est celui utilisé par défaut [27] par l'éditeur de lien `ld`. Dès lors, si `taille.o` est le fichier objet produit par l'assemblage de `taille.asm`, l'exécutable `taille` est obtenu simplement par :

```
ld -o taille taille.o
```

sans devoir spécifier explicitement le point d'entrée.

Extension de signe Toujours dans le code ci-dessus, il est mis en exergue que l'instruction mov [28] avec comme destination un *emplacement mémoire* de 64 bits et comme source un immédiat voit la taille de cet immédiat limitée à 32 bits, cette valeur étant étendue à 64 bits par extension de signe lors de l'exécution. Une telle situation rappelle les cas déjà rencontrés lors des TD02 et 03 des instructions `and`, `or`, `xor` et `cmp` avec un immédiat en opérande de droite, quelle que soit la nature de celui de gauche.

En résumé, la seule instruction acceptant un immédiat sur 64 bits vue au cours des laboratoires microprocesseur jusqu'à présent est `mov` avec un *registre* 64 bits comme destination. Dans les autres cas, si une valeur sur 64 bits est nécessaire et qu'un immédiat est fourni, il est codé par l'assembleur sur 32 bits et étendu par extension de signe lors de l'exécution au sein du microprocesseur.

3.3.2 Problème de section

L'assembleur `nasm` ne retient pas la section dans laquelle une variable est déclarée. Aucune erreur ni avertissement ne sont produits lors de l'assemblage du code source suivant :

```
1  ; rodata.asm
2  global _start
3
4  section .rodata
5      c1      DQ      -1
6
7  section .text
8  _start:
9      mov qword [c1], 0    ; boum ici !
10
11     mov rax, 60
12     mov rdi, 0
13     syscall
14
```

27. https://stackoverflow.com/a/33537191 (consulté le 17 février 2020).
28. https://www.felixcloutier.com/x86/mov (consulté le 17 février 2020).

Par contre, lors de l'exécution du programme résultant, les choses se passent mal [29] :

```
$ nasm -w+all -f elf64 rodata.asm
$ ld -o rodata rodata.o
$ ./rodata
Segmentation fault (core dumped)
$
```

En voici la raison : on y tente de modifier le contenu d'une variable définie dans la section `.rodata` (voir début de la section 2.1).

4 Type de données

Dans cette section, nous allons passer en revue quelques types de données classiques et voir comment les implémenter en langage d'assemblage. Les chaînes de caractères sont introduites dans le TD05. Les tableaux sont explorés dans le TD06.

4.1 Entier

Les entiers sont le cas le plus simple. Nous les avons déjà rencontrés dans nos exemples. Pour définir un entier, il suffit de choisir une taille (1, 2, 4 ou 8 *bytes*) et de réserver l'espace en conséquence.

L'assembleur `nasm` offre de nombreuses possibilités pour indiquer un littéral entier [30].

Exemple Voici un extrait de code source où des variables entières sont déclarées et initialisées dans différentes bases :

```
1 section .rodata
2     vqd     DQ      -1          ; valeur décimale
3     vdh     DD      0x12345678  ; valeur hexadécimale
4     vwo     DW      144q        ; valeur octale
5     vbb     DB      10101010b   ; valeur binaire
6
```

4.2 Flottant

L'assembleur `nasm` prend en charge de nombreuses formes de constantes flottantes [31]. Cependant, dans l'architecture x86, les flottants disposent de registres et d'instructions

29. https://en.wikipedia.org/wiki/Segmentation_fault#Writing_to_read-only_memory (consulté le 20 mars 2020).
30. https://www.nasm.us/doc/nasmdoc3.html#section-3.4.1 (consulté le 17 février 2020).
31. https://www.nasm.us/xdoc/2.14.02/html/nasmdoc3.html#section-3.4.6 (consulté le 16 février 2020).

dédiés [32]. Nous ne les voyons pas au laboratoire microprocesseur [33].

4.3 Caractère

Comme en `Java`, les caractères sont considérés comme un type numérique. La table `ASCII` associe un nombre à chaque caractère. Pour les caractères au delà de la table `ASCII`, `linux1` utilise le codage `UTF-8` [34]. Les caractères accentués, par exemple, sont stockés sur 2 *bytes*.

Exemple Voici un exemple de code source déclarant des variables de type caractère :

```
1 section .data
2     c1      DB      'A' ; c1 contient le code utf-8 du caractère 'A'
3     c2      DB      65  ; c2 contient la valeur 65...
4     ; on désire utiliser c2 comme un caractère
5     ; question : quel est le caractère stocké dans c2 ?
6
```

5 Débogage, variables et sections

5.1 Variables

Pour voir le contenu d'une variable dans `KDbg`, il faut donner son nom *sans aucun crochet* dans la vue *Expressions surveillées* (*Watched Expressions*).

Par défaut, `KDbg` considère que la variable s'étend sur 4 *bytes*. Si on désire inspecter le contenu d'une variable de taille différente de 4 *bytes*, il faut recourir à une syntaxe issue du langage C [35]. En particulier, il faut utiliser l'opérateur de transtypage [36] (*cast*) du langage C.

Avec les variables `i1`, `i2`, `i4` et `i8` telles que définies dans l'exemple de la section 2.1 (p. 3), on peut utiliser :

- `(char) i1`
- `(short) i2`
- `(int) i4`
- `(long long) i8`

pour inspecter leurs contenus dans la vue *Expressions surveillées* (*Watched Expressions*) de `KDbg`.

32. https://en.wikibooks.org/wiki/X86_Assembly/Floating_Point (consulté le 16 février 2020).
33. Il est possible que vous en ayez un aperçu au cours théorique de microprocesseur.
34. https://en.wikipedia.org/wiki/UTF-8 (consulté le 17 février 2020).
35. http://www.open-std.org/JTC1/SC22/WG14/ (consulté le 17 février 2020).
36. https://en.cppreference.com/w/c/language/cast (consulté le 17 février 2020).

D'autre part, pour changer le format de l'affichage des valeurs, il faut utiliser les options de formatage en sortie [37] de `gdb`.

5.2 Sections

Pour voir le contenu d'une section dans `KDbg`, il faut donner le nom de la première variable qui y est déclarée *précédé d'une esperluette* (`&`) dans la vue *Mémoire* (*Memory*). Il s'agit à nouveau d'une syntaxe issue du langage C. L'esperluette est l'opérateur adresse de [38].

Avec les sections `.data` et `.rodata` telles que définies dans l'exemple de la section 2.1 (p. 3), on peut utiliser :

— `&i1` pour inspecter le contenu de la section `.data` ;
— `&ci8` pour inspecter le contenu de la section `.rodata` ;

dans la vue *Mémoire* (*Memory*) de `KDbg`.

Avec la section `.bss` telle que définie dans l'exemple de la section 2.2 (p. 4), on peut utiliser `&x1` pour inspecter le contenu de la section `.bss` dans la vue *Mémoire* (*Memory*) de `KDbg`.

6 Exercices

Ex. 1 Remplissez les pointillés puis exécutez le programme dans `KDbg` pour vérifier vos réponses.

```
1  global _start
2
3  section .data
4      var1   DB   1
5      var2   DB   2
6      var3   DW   0x0304
7      var4   DQ   0x0000000008000FFFF
8      ; la section des données occupe ..... bytes
9      ; son contenu est ....................................
10
11 section .text
12 _start:
13     mov rax, var1      ; rax contient ........................
14     mov al,  [var1]    ; al  contient ........................
15     mov ax,  [var1]    ; ax  contient ........................
```

37. http://sourceware.org/gdb/current/onlinedocs/gdb/Output-Formats.html#Output-Formats (consulté le 17 février 2020).
38. https://en.cppreference.com/w/c/language/operator_member_access (consulté le 9 mars 2021)

```
16      mov al,  [var3]    ; al  contient ..........................
17      mov ax,  [var3]    ; ax  contient ..........................
18      mov rax, -1        ; rax contient ..........................
19      mov eax, [var4]    ; rax contient ..........................
20
21      mov rax, 60
22      mov rdi, 0
23      syscall
24
```

Ex. 2 Écrivez un code source complet qui déclare une variable `nb` de taille 4 *bytes*. Il place ensuite l'adresse de cette variable dans `rax` et son contenu dans `rbx`.

Testez dans `KDbg` avec des valeurs de variable de bits de signe différents.

Ex. 3 Écrivez un code source complet qui déclare une variable sur 8 *bytes* implicitement initialisée à 0 puis lui assigne la valeur 42.

Ex. 4 Soient les déclarations suivantes :

```
1  section .data
2      b0      DB      0
3      b1      DB      0
4      b2      DB      0
5      b3      DB      0
6  section .rodata
7      nb      DD      0x12345678
```

Écrivez un code source complet qui stocke dans `b0` le *byte* de rang 0 de `nb`, dans `b1` celui de rang 1, dans `b2` celui de rang 2 et finalement dans `b3` celui de rang 3.

Comme nous n'étudions pas d'instructions permettant d'utiliser deux opérandes de type *variable*, utilisez un ou des registres intermédiaires.

Ex. 5 Écrivez un code source complet qui :

1. déclare deux variables initialisées aux valeurs de votre choix ;
2. échange les contenus de ces variables.

Ex. 6 Écrivez un code source complet qui déclare trois variables dont deux sont constantes et explicitement initialisées mais pas la troisième. Le contenu de cette dernière est calculé. Il est égal au minimum des deux autres.

Références

[1] *Intel© 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, octobre 2017. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[2] Igor Zhirkov. *Low-Level Programming*. Apress, 2017. https://www.apress.com/gp/book/9781484224021.