Pointeurs

François Trahay

CSC4103 – Programmation système

2021–2022

1 Espace mémoire d'un processus

- Espace mémoire dans lequel un processus peut stocker des données/du code
- Séparé en plusieurs parties (*segments*), dont:
  - *pile* (*stack*): les variables locales et paramètres de fonctions
  - *tas* (*heap*): les variables globales
  - *segment de code* : le code (binaire) du programme

2 Adresse mémoire

- On peut faire référence à n'importe quel octet de l'espace mémoire grace à son adresse
- Adresse mémoire virtuelle codée sur k bits[a]
  - donc 2^k octets accessibles (de `00...00` à `11...11`)
- exemple: à l'adresse `0x1001` est stocké l'octet `0x41`
  - peut être vu comme un `char` (le caractère `A`)
  - peut être vu comme une partie d'un `int` (par exemple l'entier `0x11412233`)

valeur	0x11	0x41	0x22	0x33	0xab	0x12	0x50	0x4C	0x4F	0x50	0x21	0x00
Adresse	0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	0x1008	0x1009	0x100A	0x100B

a. k dépend de l'architecture. Sur les processeurs modernes (64 bits), on a $k = 64$.

2.1 Adresse d'une variable

- Adresse d'une variable: `&var`
  - `&var` désigne l'adresse de `var` en mémoire
  - affichable avec %p dans `printf`:

    ```
    printf("adresse de var: %p\n", &var);
    ```

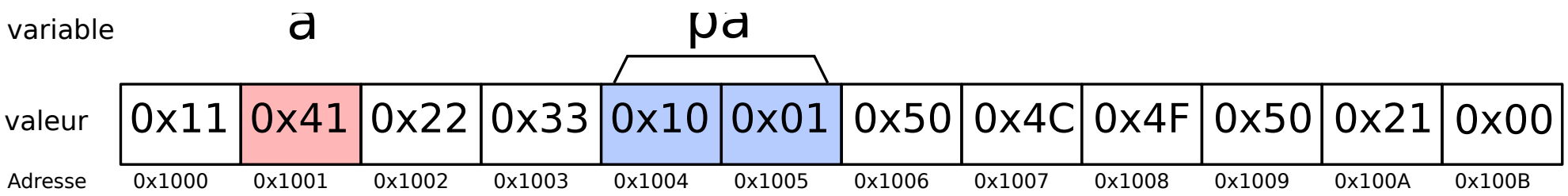
    affiche:

    ```
    adresse de var: 0x7ffe8d0cbc7f
    ```

3 Pointeur

- Variable dont la valeur est une adresse mémoire
- valeur binaire codée sur k bits (k dépend de l'architecture du processeur)
- déclaration: `type* nom_variable;`
  - `type` désigne le type de la donnée "pointée"
- exemple: `char* pa;` crée un pointeur sur une donnée de type `char`:

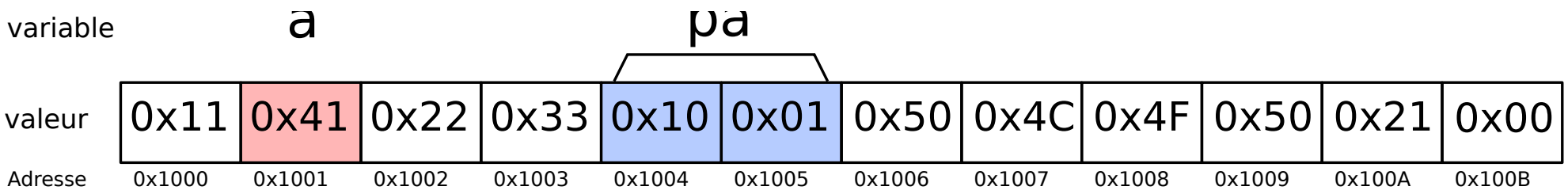
```
// pour l'exemple, les adresses sont codees sur 32 bits
char a = 'A'; // a est stocke a l'adresse 0x0000FFFF
              // la valeur de a est 0x41 ('A')
char* pa = &a; // pa est une variable de 32 bits stockee
               // aux adresses 0xFFFB a 0xFFFE
               // la valeur de pa est 0x0000FFFF (l'adresse de a)
```



3.1 Déréférencement

- **Déréférencement** : consulter la valeur stockée à l'emplacement désigné par un pointeur
- `* ptr`
  - exemple:

```
char a = 'A'; // valeur 0x41 (cf. codage ASCII)
char* pa = &a;
printf("pa = %p\n", pa); // affiche "pa = 0xFFFF"
printf("*pa = %c\n", *pa); // affiche "*pa = A"
*pa = 'B';    // modifie l'emplacement memoire 0xFFFF
              // (donc change la valeur de a)
printf("a = %c\n", a); // affiche "a = B"
```



3.2 Tableaux et pointeurs (1/3)

- si un tableau est **un argument de fonction**
  - la déclaration est **remplacée** par celle d'un pointeur
  - `void f(int x[])` <=> `void f(int* x)`
  - un accès effectue un décalage + déréférencement
    - `tab[i]` réécrit en `*(tab + i)`
    - exemple: si `tab = 0x1000` et `i=5`
    - `tab[i]` calcule `0x1000 + (5*sizeof(int)) = 0x1000 + 0x14 = 0x1014`
  - `sizeof(tab)` donne la taille d'un pointeur
  - Remarque : `&tab` donne l'adresse de `int[] tab`, donc `&tab != tab`

3.3 Tableaux et pointeurs (2/3)

- si un tableau est **une variable locale ou globale**
  - le tableau **n'est pas remplacé par un pointeur**
  - le tableau doit avoir une taille connue
    - `int tab[3];` alloue 3 `int`
      - `tab` est le nom de cet espace mémoire
    - `int tab[] = { 1, 2, 3 };` idem + initialisation
    - `int tab[];` **interdit**
  - `sizeof(tab)` renvoie la taille du tableau

3.4 Tableaux et pointeurs (3/3)

- si un tableau est **une variable locale ou globale** (suite)
  - `&tab` donne l'adresse du tableau
    - Remarque : `&tab == &tab[0]` car `tab` et `tab[0]` désignent les mêmes emplacements mémoires
  - `tab` est **implicitement transtypé vers son pointeur** au besoin
  - Exemple :
    - `int* tab2 = tab;` réécrit en `int* tab2 = &tab`
    - `if(tab == &tab)` récrit en `if(&tab == &tab)`
    - `f(tab)` réécrit en `f(&tab)`
    - `printf("%p %p\n", tab, &tab);` réécrit en `printf("%p %p\n", &tab, &tab);`
    - `tab[i]` réécrit en `(&tab)[i]` puis en `*(&tab + i)`
    - `*(tab + i)` réécrit en `*(&tab + i)`

3.5 Passage par référence

Rappel:

- *Passage par référence: une référence vers l'argument de l'appelant est donné à l'appelé* (cf. CI2)
- Cette référence est un pointeur

```
void f(int* px) {
  *px = 666;          // la variable pointee par px est modifiee
}

int main() {
  int x = 42;
  f(&x);                    // l'adresse de x est donnee à f
          //  => le x de main est modifié par f
  printf("x = \%d\n", x); // la nouvelle valeur de x : 666
  return EXIT_SUCCESS;
}
```

4 Allocation dynamique de mémoire

- `void* malloc(size_t nb_bytes);`
- Alloue `nb_bytes` octets et retourne un pointeur sur la zone allouée
- usage:
- `char* str = malloc(sizeof(char)* 128);`
- renvoie `NULL` en cas d'erreur (par ex: plus assez de mémoire)

Attention ! Risque de "fuite mémoire" si la mémoire allouée n'est jamais libérée

4.1 Libération de mémoire

- `void free(void* ptr);`
- Libère la zone allouée par `malloc` est situé à l'adresse `ptr`
- Attention à ne pas libérer plusieurs fois la même zone !

4.2 Notions clés

- L'espace mémoire d'un processus
- Les pointeurs
  - Adresse mémoire d'une variable (`&var`)
  - Pointeur sur `type` : `type* ptr;`
  - Arithmétique de pointeurs (`ptr++`)
  - Adresse nulle: `NULL`
  - Déréférencement d'un pointeur:
    - types simples: `*ptr`
    - structures: `ptr->champ`
    - tableaux: `ptr[i]`
  - Passage de paramètre par référence
- Allocation dynamique de mémoire
  - allocation: `int* ptr = malloc(sizeof(int)*5);`
  - désallocation: `free(ptr);`