

Base II

Programmation séquentielle en C, 2020-2021

Orestis Malaspinas (A401), ISC, HEPIA

2020-09-23

Inspirés des slides de F. Glück

Structures de contrôle: `switch .. case` (1/2)

```
switch (expression) {  
    case constant-expression:  
        instructions;  
        break; // optionnel  
    case constant-expression:  
        instructions;  
        break; // optionnel  
    // ...  
    default:  
        instructions;  
}
```

Que se passe-t-il si `break` est absent?

Structures de contrôle: `switch .. case` (2/2)

```
int x = 0;
switch (x) {
    case 0:
    case 1:
        printf("0 ou 1\n");
        break;
    case 2:
        printf("2\n");
        break;
    default:
        printf("autre\n");
}
```

Dangereux, mais c'est un moyen d'avoir un "ou" logique dans un case.

Quiz: `switch ... case`

Quiz: `switch ... case`

Structures de contrôle: `for`

La boucle `for`

```
for (expression1; expression2; expression3) {  
    instructions;  
}
```

La boucle `for`

```
int sum = 0; // syntaxe C99  
for (int i = 0; i < 10; i++) {  
    sum += i;  
}  
  
for (int i = 0; i != 1; i = rand() % 4) { // ésotérique  
    printf("C'est plus ésotérique.\n");  
}
```

Structures de contrôle: `continue`, `break`

- `continue` saute à la prochaine itération d'une boucle.

```
int i = 0;
while (i < 10) {
    if (i == 3) {
        continue;
    }
    printf("%d\n", i);
    i += 1;
}
```

- `break` quitte le bloc itératif courant d'une boucle.

```
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        break;
    }
    printf("%d\n", i);
}
```

Représentation des variables en mémoire (1/2)

La mémoire

- La mémoire est:
 - ... un ensemble de bits,
 - ... accessible via des adresses,

bits	00110101	10010000	00110011
addr	2000	2001	4000

- ... gérée par le système d'exploitation.
- ... séparée en deux parties: **la pile** et **le tas**.

Une variable

- Une variable, type `a = valeur`, possède:
 - un type (`char`, `int`, ...),
 - un contenu (une séquence de bits qui encode valeur),
 - une adresse mémoire (accessible via `&a`),
 - une portée.

Représentation des variables en mémoire (2/2)

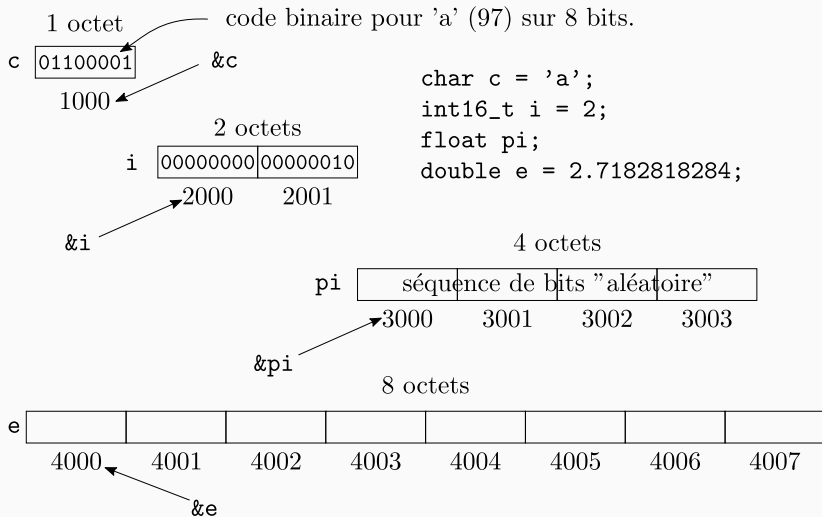


Figure 1 – Les variables en mémoire.

Les fonctions (1/7)

- Les parties indépendantes d'un programme.
- Permettent de modulariser et compartimenter le code.
- Syntaxe:

```
type identificateur(paramètres) {  
    // variables optionnelles  
    instructions;  
    // type expression == type  
    return expression;  
}
```

Les fonctions (2/7)

Exemple

```
int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
int main() {  
    int c = max(4, 5);  
}
```

Les fonctions (3/7)

- Il existe un type `void`, “sans type”, en C.
- Il peut être utilisé pour signifier qu'une fonction ne retourne rien, ou qu'elle n'a pas d'arguments.
- `return` utilisé pour sortir de la fonction.
- Exemple:

```
void show_text(void) { // second void optionnel
    printf("Aucun argument et pas de retour.\n");
    return; // optionnel
}
```

```
void show_text_again() { // c'est pareil
    printf("Aucun argument et pas de retour.\n");
}
```

Prototypes de fonctions

- Le prototype donne la **signature** de la fonction, avant qu'on connaisse son implémentation.
- L'appel d'une fonction doit être fait **après** la déclaration du prototype.

```
int max(int a, int b); // prototype

int max(int a, int b) { // implémentation
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Les fonctions (5/7)

Arguments de fonctions

- Les arguments d'une fonction sont toujours passés **par copie**.
- Les arguments d'une fonction ne peuvent **jamais** être modifiés.

```
void set_to_two(int a) { // a: nouvelle variable
    // valeur de a est une copie de x
    // lorsque la fonction est appelée, ici -1

    a = 2; // la valeur de a est fixée à 2
} // a est détruite

int main() {
    int x = -1;
    set_to_two(x); // -1 est passé en argument
    // x vaudra toujours -1 ici
}
```

Arguments de fonctions: pointeurs

- Pour modifier un variable, il faut passer son **adresse mémoire**.
- L'adresse d'une variable, `x`, est accédé par `&x`.
- Un **pointeur** vers une variable entière a le type, `int *x`.
- La syntaxe `*x` sert à **déréférencer** le pointeur (à accéder à la mémoire pointée).

Les fonctions (7/7)

Exemple

```
void set_to_two(int *a) {  
    // a contient une copie de l'adresse de la  
    // variable passée en argument  
  
    *a = 2; // on accède à la valeur pointée par a,  
           // et on lui assigne 2  
} // le pointeur est détruit, pas la valeur pointée    int x = -1;  
    set_to_two(&x); // l'adresse de x est passée  
    // x vaudra 2 ici  
}
```

Quiz: Les fonctions

Quiz: Les fonctions

La fonction `main()` (1/2)

Généralités

- Point d'entrée du programme.
- Retourne le code d'erreur du programme:
 - 0: tout s'est bien passé.
 - Pas zéro: problème.
- La valeur de retour peut être lue par le shell qui a exécuté le programme.
- `EXIT_SUCCESS` et `EXIT_FAILURE` (de `stdlib.h`) sont des valeurs de retour **portables** de programmes C.

La fonction main() (2/2)

Exemple

```
int main() {  
    // ...  
    if (error)  
        return EXIT_FAILURE;  
    else  
        return EXIT_SUCCESS;  
}
```

- Le code d'erreur est lu dans le shell avec \$?

```
$ ./prog  
$ echo $?  
0 # tout s'est bien passé par exemple  
$ if [ $? -eq 0 ]; then echo "OK" ; else echo "ERROR"; fi  
ERROR # si tout s'est mal passé
```