Base IV

Programmation séquentielle en C, 2020-2021

Orestis Malaspinas (A401), ISC, HEPIA

2020-10-07

Inspirés des slides de F. Glück

Les tableaux (1/6)

Généralités

- `C` offre uniquement des tableaux statiques
  - Un tableau est un "bloc" de mémoire contiguë associé à un nom
    - taille fixe déterminée à la déclaration du tableau
    - la taille ne peut pas être changée.
  - Pas d'assignation de tableaux.
  - Un tableau déclaré dans une fonction ou un bloc est détruit à la sortie de celle/celui-ci
    - $\Rightarrow$ Un tableau local à une fonction ne doit **jamais être retourné** (aussi valable pour toute variable locale)!
- Les éléments d'un tableau sont accédés avec `[i]` où `i` est l'index de l'élément.
- Le premier élément du tableau à l'index `0`!
- Lorsqu'un tableau est déclaré, la taille de celui-ci doit toujours être spécifiée, sauf s'il est initialisé lors de sa déclaration.

Les tableaux (2/6)

Exemple

```
float tab1[5]; // tableau de floats à 5 éléments
               // ses valeurs sont indéfinies

int tab2[] = {1, 2, 3}; // tableau de 3 entiers,
                        // taille inférée

int val = tab2[1]; // val vaut 2 à présent

int w = tab1[5]; // index hors des limites du tableau
                 // comportement indéfini!
                 // pas d'erreur du compilateur
```

Les tableaux (3/6)

Itérer sur les éléments d'un tableau

```
int x[10];
for (int i = 0; i < 10; ++i) {
    x[i] = 0;
}
int j = 0;
while (j < 10) {
    x[j] = 1;
    j += 1;
}
int j = 0;
do {
    x[j] = -1;
    j += 1;
} while (j < 9)
```

Les tableaux (4/6)

Les tableaux comme argument

- Un tableau est le pointeur vers sa première case.
- Pas moyen de connaître sa taille: `sizeof()` inutile.
- Toujours spécifier la taille d'un tableau passé en argument.

```
void foo(int tab[]) { // sans taille...
    for (int i = 0; i < ?; ++i) {
        // on sait pas quoi mettre pour ?
        printf("tab[%d] = %d\n", i, tab[i]);
    }
}
// n doit venir avant tab, [n] optionnel
void bar(int n, int tab[n]) {
    for (int i = 0; i < n; ++i) {
        printf("tab[%d] = %d\n", i, tab[i]);
    }
}
```

Les tableaux (5/6)

Quels sont les bugs dans ce code?

```c
#include <stdio.h>

int main(void) {
    char i;
    char a1[] = { 100,200,300,400,500 };
    char a2[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    a2[10] = 42;

    for (i = 0; i < 5; i++) {
        printf("a1[%d] = %d\n", i, a1[i]);
    }

    return 0;
}
```

Les tableaux (6/6)

Quels sont les bugs dans ce code?

```c
#include <stdio.h>

int main(void) {
    char i;
    // 200, .., 500 char overflow
    char a1[] = { 100,200,300,400,500 };
    char a2[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    a2[10] = 42; // [10] out of bounds

    for (i = 0; i < 5; i++) {
        printf("a1[%d] = %d\n", i, a1[i]);
    }

    return 0;
}
```

Introduction à `make`

A quoi ça sert?

- Automatiser le processus de conversion d'un type de fichier à un autre, en *gérant les dépendances*.
- Effectue la conversion des fichiers qui ont changé uniquement.
- Utilisé pour la compilation:
  - Création du code objet à partir des sources.
  - Création de l'exécutable à partir du code objet.
- Tout "gros" projet utilise `make` (pas uniquement en `C`).

Utilisation de `make`

Le programme `make` exécutera la série d'instruction se trouvant dans un `Makefile` (ou `makefile` ou `GNUmakefile`).

Le `Makefile`

- Contient une liste de *règles* et *dépendances*.
- Règles et dépendances construisent des *cibles*.
- Ici utilisé pour compiler une série de fichiers sources

```
$ gcc -c example.c # + plein d'options..
$ gcc -o example exemple.o # + plein d'options
```

`Makefile`

```
example: example.o
    gcc -o example example.o

exmaple.o: exmaple.c example.h
    gcc -c example.c
```

Terminal

```
$ make
gcc -c example.c
gcc -o example example.o
```

Syntaxe d'un `Makefile` (1/4)

```
example: example.c example.h
	gcc example.c -o example
```

Figure 1 – Un exemple simple de `Makefile`.

Syntaxe d'un `Makefile` (2/4)

cible (target)

```
example: example.c example.h
	gcc example.c -o example
```

Figure 2 – La cible.

Syntaxe d'un `Makefile` (3/4)

dépendances (dependencies)

```
example: example.c example.h
    gcc example.c -o example
```

Figure 3 – Les dépendances.

Syntaxe d'un `Makefile` (4/4)

```
example: example.c example.h
    gcc example.c -o example
```

règle (rule)

Figure 4 – La règle.

Principe de fonctionnement

1. `make` cherche le fichier `Makefile`, `makefile` ou `GNUmakefile` dans le répertoire courant.
2. Par défaut exécute la première cible, ou celle donnée en argument.
3. Décide si une cible doit être régénérée en comparant la date de modification (on recompile que ce qui a été modifié).
4. Regarde si les dépendances doivent être régénérées:
   - Oui: prend la première dépendance comme cible et recommence à 3.
   - Non: exécute la règle.

`make` a un comportement **récursif**.

Exemple avancé

Makefile

```
hello: hello.o main.o
    gcc hello.o main.o -o hello

hello.o: hello.c hello.h
    gcc -Wall -Wextra -c hello.c

main.o: main.c
    gcc -Wall -Wextra -c main.c

clean:
    rm -f *.o hello

rebuild: clean hello
```

Un graph complexe

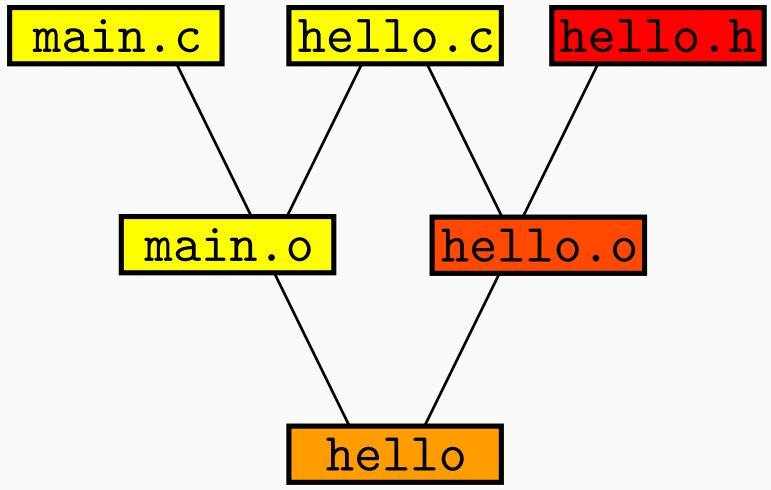

Figure 5 – Makefile complexe.

Factorisation

Ancien `Makefile`

```
hello: hello.o main.o
    gcc hello.o main.o -o hello

hello.o: hello.c hello.h
    gcc -Wall -Wextra -c hello.c

main.o: main.c
    gcc -Wall -Wextra -c main.c

clean:
    rm -f *.o hello

rebuild: clean hello
```

Nouveau `Makefile`

```
CC=gcc -Wall -Wextra

hello: hello.o main.o
    $(CC) $^ -o $@

hello.o: hello.c hello.h
    $(CC) -c $<

main.o: main.c
    $(CC) -c $<

clean:
    rm -f *.o hello

rebuild: clean hello
```

Variables

Variables utilisateur

- Déclaration

```
id = valeur
id = valeur1 valeur2 valeur3
```

- Utilisation

```
$(id)
```

- Déclaration à la ligne de commande

```
make CFLAGS="-O3 -Wall"
```

Variables internes

- `$@` : la cible
- `$^` : la liste des dépendances
- `$<` : la première dépendance
- `$*` : le nom de la cible sans extension