



UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en C

Cours 2. Types et structures de contrôle avancés

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : `https://upinfo.univ-cotedazur.fr/~obaldellon/`

LICENCE 2 — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- 🍃 Partie I. Makefile
- 🍃 Partie II. Tableaux et chaînes de caractères
- 🍃 Partie III. Types avancés
- 🍃 Partie IV. Structures de contrôle avancées
- 🍃 Partie V. Représentations binaires
- 🍃 Partie VI. Table des matières

- ▶ Pour le TP1, on utilise un script pour compiler nos programmes.

```
#!/bin/bash
echo "Compilation de l'exercice 1"
gcc -Wall -pedantic -ansi exercice1.c -o exercice1
echo "Compilation de l'exercice 2"
gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

compiler.sh

- Problème : c'est inefficace, car on recompile tout à chaque fois.
 - ▶ Pour les gros projets, tout compiler prend beaucoup de temps.
 - ▶ On ne veut compiler que le nécessaire.
 - ▶ On veut aussi s'arrêter en cas d'erreur.
- ▶ Il existe une commande Unix prévue pour cela : « **make** »

- Pour cela créons un fichier nommé **Makefile** et contenant :

```
exercice1: exercice1.c
gcc -Wall -pedantic -ansi exercice1.c -o exercice1

exercice2: exercice2.c
gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

compiler.sh

```
exercice1: exercice1.c
gcc -Wall -pedantic -ansi exercice1.c -o exercice1

exercice2: exercice2.c
gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

Makefile

- ▶ La commande « **make** » regarde le fichier intitulé « Makefile »
 - ▶ le fichier peut aussi s'appeler « makefile »
 - ▶ un autre nom et la commande make ne marche pas.
- ▶ « **make** **exercice1** » fait appel à la règle **exercice1**
 - ▶ si **exercice1** n'existe pas ou s'il est plus ancien que **exercice1.c**
 - ▶ cela signifie qu'il faut (re)compiler pour (re)construire **exercice1**
 - ▶ Pour cela, make lance les commandes associées à la règle **exercice1**

```
olivier@valrose:~ $ ls
exercice1.c  exercice2.c  Makefile
olivier@valrose:~ $ make exercice1
gcc -Wall -pedantic -ansi exercice1.c -o exercice1
olivier@valrose:~ $ ls
exercice1  exercice1.c  exercice2.c  Makefile
olivier@valrose:~ $ make exercice1
make: « exercice1 » est à jour.
```

SHELL

```
all: exercice1 exercice2
exercice1: exercice1.c
    gcc -Wall -pedantic -ansi exercice1.c -o exercice1
exercice2: exercice2.c
    gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

Makefile

- ▶ Pour tout compiler on va ajouter une règle par défaut
 - ▶ « make » (sans argument) fais appel à la règle **all**
 - ▶ Pour cela, il regarde si **exercice1** et **exercice2** existent déjà.
 - ▶ **exercice2** n'existant pas, make cherche une règle pour le construire.
 - ▶ make fait alors appel à la règle **exercice2**

```
olivier@valrose:~ $ ls
exercice1  exercice1.c  exercice2.c  Makefile
olivier@valrose:~ $ make
gcc -Wall -pedantic -ansi exercice2.c -o exercice2
olivier@valrose:~ $ make
make: rien à faire pour « all ».
```

SHELL

```
all: exercice1 exercice2
```

Makefile

```
exercice1: exercice1.c
```


```
→ gcc -Wall -pedantic -ansi exercice1.c -o exercice1
```

```
exercice2: exercice2.c
```

```
→ gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

- ▶ Un Makefile permet de faire énormément de chose!
 - ▶ Makefile ne recompile que si nécessaire.
 - ▶ On verra comment améliorer le Makefile les prochaines semaines.



- ▶ Les espaces en début de ligne (→) doivent être des tabulations ()
- ▶ Astuce : « make -B » permet de tout reconstruire

```
olivier@valrose:~ $ ls
exercice1  exercice1.c  exercice2  exercice2.c  Makefile
olivier@valrose:~ $ make
make: rien à faire pour « all ».
olivier@valrose:~ $ make -B
gcc -Wall -pedantic -ansi exercice1.c -o exercice1
gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

SHELL

- 🍃 Partie I. Makefile
- 🍃 Partie II. Tableaux et chaînes de caractères
- 🍃 Partie III. Types avancés
- 🍃 Partie IV. Structures de contrôle avancées
- 🍃 Partie V. Représentations binaires
- 🍃 Partie VI. Table des matières

- ▶ Le C possède des tableaux.
 - ▶ On accède aux éléments comme en Python
 - ▶ On peut les déclarer sans les initialiser (**Attention!**)

```
int i;
int somme;
int un_tableau[4];
int un_autre_tableau[] = {10, 11, 12, 13};

/* On initialise le premier tableau */
for (i=0; i<4; i++) {
    un_tableau[i] = i;
}

/* On somme les valeurs des deux tableaux*/
somme = 0;
for (i=0; i<4; i++) {
    somme += un_tableau[i] + un_autre_tableau[i];
}
```

*.C

- ▶ Un tableau en C est typé et a une taille fixe.
 - ▶ Impossible d'y ajouter un élément.
 - ▶ Impossible de mélanger les types.
- ▶ Pour déclarer un tableau il faut connaître la taille à la compilation.

```
/* tableau initialisé de taille 3 */  
int tableau[] = {11, 22, 33};  
  
/* tableau non initialisé de taille 10 */  
int tab[10];
```

*.C

- ▶ Le code suivant ne fonctionne pas :

```
int n=10;  
int tab[n]; /*      /\ Erreur /\      */
```

*.C

- ▶ Et si on ne connaît pas la taille ?
 - ▶ par défaut les variables sont créées sur la pile ;
 - ▶ les variables de la pile ont une taille connue à la compilation ;
 - ▶ pour déclarer une variable sur le tas, il faut des pointeurs.

- ▶ Les tableaux C ne connaissent pas leur taille.
 - ▶ pas de fonction pour calculer la longueur (pas de fonction `len`)
 - ▶ Il faut systématiquement la transmettre avec le tableau.

```
void initialiser(int tableau[], int longueur) {  
    int i;  
    for (i=0; i<longueur; i++) {  
        tableau[i] = 25;  
    }  
}  
  
void afficher(int tableau[], int longueur) {  
    int i;  
    for (i=0; i<longueur; i++) {  
        printf("%d ",tableau[i]);  
    }  
    printf("\n");  
}  
  
int main(void) {  
    int tab[10];  
    initialiser(tab, 10);  
    afficher(tab,10);  
    return 0;  
}
```

*.C

- ▶ En C une chaîne de caractères est un tableau de `char`.
- ▶ La taille d'une chaîne n'est pas connu à priori.
 - ▶ il faut la parcourir jusqu'à tomber sur le caractère `'\0'`.

```
/* Trois déclarations équivalentes */  
char s1[] = "Bonjour";  
char s2[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'}  
char s3[] = { 66, 111, 110, 106, 111, 117, 114, 0}
```

*.C

- ▶ On peut aussi déclarer avec la notation « pointeur ».
 - ▶ Dans ce cas la chaîne n'est pas modifiable;

```
char *s1 = "Bonjour";
```

*.C

- ▶ Comment connaître la longueur d'une chaîne.

```
int longueur(char chaine[]) {  
    int i = 0;  
    while (chaine[i] != '\0') {  
        i++;  
    }  
    return i;  
}
```

*.C

- ▶ Écrire des fonctions pour faire les calculs suivants :
 - Concaténation de deux chaînes
 - Ajout d'un élément à la fin d'un tableau
 - Calculer le miroir d'une chaîne.
 - Transformer un entier en chaîne de caractères : 123 → "123".

- ▶ Au risque de vous décevoir c'est impossible...
 - ▶ ... avec les outils vus durant ces deux premières semaines.

- ▶ Il faudrait être capable de **créer dynamiquement** de nouvelles données ;
 - ▶ ce sera l'objet du prochain cours :
 - ▶ la pile, le tas et les pointeurs.

- 🍃 Partie I. Makefile
- 🍃 Partie II. Tableaux et chaînes de caractères
- 🍃 **Partie III. Types avancés**
- 🍃 Partie IV. Structures de contrôle avancées
- 🍃 Partie V. Représentations binaires
- 🍃 Partie VI. Table des matières

- ▶ Un `enum` permet de créer un type avec un nombre fini de valeurs

```
enum Feu {  
    vert,  
    orange,  
    rouge  
};
```

*.c

- ▶ Usage :

```
enum Feu {vert, orange, rouge};  
  
int main(void) {  
    enum Feu x = vert;  
    if (x == orange) {  
        printf("J'accélère !!!");  
        x = rouge;  
    }  
    return 0;  
}
```

*.c

```
enum Feu {  
    vert,  
    orange,  
    rouge  
};
```

*.C

- ▶ Comment fonctionnent les énumérations?
 - ▶ C associe un entier (int) à chacune des valeurs (en commençant par 0).
 - ▶ vert **est** l'entier 0, orange l'entier 1 et rouge l'entier 2
- ▶ Remarque en C la couche d'abstraction est toujours très fine.
 - ▶ C'est une des forces (et faiblesses) de ce langage.
- ▶ On peut choisir nous même les entiers associés.

```
enum Carte {Carreau = 1, Pique = -5, Trefle = 5, Coeur};  
enum Operateur {Plus='+', Moins='-', Fois='*', Div = '/'};
```

*.C

- ▶ Une structure permet de combiner plusieurs données en un seul type.

```
struct date {  
    short jour;  
    short mois;  
    short année;  
};
```

*.c

- ▶ on déclare une nouvelle date avec le mot clé **struct**

```
struct date aujourd_hui;  
aujourd_hui.jour = 9;  
aujourd_hui.mois = 11;  
aujourd_hui.année = 2023;
```

*.c

- ▶ on peut initialiser les champs dès la déclaration.

```
struct date aujourd_hui = {9, 11, 2023};
```

*.c

- ▶ chaque champs s'utilise comme une variable.

- ▶ On peut créer des structures contenant des structures.

```
struct date {                               /* Déclarations multiples */
    short jour, mois, année; /* sur une ligne */
};

struct etudiant {
    int numero;
    struct date naissance;
    unsigned char notes_semestre[3];
};

int main(void) {
    struct date d = { 9, 11, 2002 };
    struct etudiant alice;
    alice.numero = 2188234;
    alice.naissance = d;
    alice.notes_semestre[0] = 6;
    alice.notes_semestre[1] = 2;
    alice.notes_semestre[2] = 7;
    return 0;
}
```

*.c

- ▶ Remarque : il est interdit d'écrire `alice.notes_semestre = {6, 2, 7};`
 - ▶ la syntaxe `= {6, 2, 7}` ne marche que lors des déclarations;
 - ▶ `{6, 2, 7}` n'est pas un tableau!

- ▶ Alice a raté son semestre : doublons ses notes

```
void doubler_note(struct etudiant kevin) {  
    int i;  
    for (i=0; i<3; i++) {  
        kevin.notes_semestre[i] *= 2;  
    }  
}  
  
int main(void) {  
    /* [...] À compléter, déclaration et initialisation des variables */  
    alice.notes_semestre[0] = 0;  
    alice.notes_semestre[1] = 7;  
    alice.notes_semestre[2] = 2;  
    doubler_note(alice); /* Les notes n'ont pas changée... */  
}
```

*.c

- Zut! Ça ne marche pas!
 - ▶ Le paramètre d'une fonction est toujours une variable locale.
 - ▶ Les tableaux sont l'exceptions : on peut les modifier via une fonction.
- Comment modifier une structure avec une fonction (sans pointeurs)?
 - ▶ On modifie la fonction pour qu'elle renvoie kevin
 - ▶ On réaffecte la variable : `alice = doubler_note(alice)`

```
struct rgb { unsigned char red, green, blue; } ;
```

*.C

- ▶ On souhaite représenter une couleur de deux façons :
 - ▶ soit un triplet RGB ;
 - ▶ Soit une chaîne de caractère au format hexa.

```
struct rgb bleu = {0, 0, 255};  
char rouge[] = "#FF0000";
```

*.C

- ▶ Comment définir une variable qui tantôt vaut bleu, tantôt vaut rouge ?
 - ▶ Le C ne permet pas le polymorphisme comme en Python.
 - ▶ une variable ne peut se voir affecter qu'un type
- ▶ Les **unions** permettent de déclarer des variables pouvant recevoir **consécutivement** des données de plusieurs types.

```
union couleur {  
    struct rgb triplet;  
    char hexa[8];  
};
```

*.C

```
union couleur {  
    struct rgb triplet;  
    char hexa[8];  
};  
  
union couleur x;
```

*.c

- ▶ On a alors l'équivalent de deux variables `x.triplet` et `x.hexa`.
 - ▶ mais on ne peut pas les utiliser en même temps.
 - ▶ Quand on affecte `x.triplet`, on ne doit plus utiliser `x.hexa`
 - ▶ Quand on affecte `x.hexa`, on ne doit plus utiliser `x.triplet`
- ▶ Avantage par rapport au structure :
 - taille des champs
 - ▶ le champs `triplet` fait 3 octets;
 - ▶ le champs `hexa` fait 8 octets;
 - taille des types *(le calcul est parfois plus compliqué pour des raisons d'alignement)*
 - ▶ union : 8 octets = $\max(3, 8)$
 - ▶ structure : 11 octets = $3 + 8$

*.C

```
union couleur {
    struct rgb tuple;
    char hexa[8];
};

int main(void) {
    int i;
    union couleur teinte;
    struct rgb bleu = {0, 0, 255};
    char rouge[] = "#FF0000";

    teinte.tuple = bleu;
    afficher_rgb(teinte.tuple)
    /* je n'ai pas le droit d'utiliser teinte.hexa */

    for (i=0; i<8; i++) teinte.hexa[i] = rouge[i];
    printf("%s\n", teinte.hexa);
    /* je n'ai plus le droit d'utiliser teinte.tuple */

    return 0;
}
```

```
(0,0,255)
#FF0000
```

stdout

- ▶ Les mots-clés (**enum**, **union**, ...) rendent les déclarations fastidieuses.
- ▶ On peut donner des alias de type avec **typedef**.
 - ▶ Cela permet de donner des noms plus lisibles
 - ▶ et de rajouter un peu de sémantique.
 - ▶ Ces changements sont cosmétiques et ne crée pas de nouveaux types.

```
/* Je remplace « unsigned char » par « octet » */  
typedef unsigned char octet;  
  
struct rgb {  
    octet r;  
    octet g;  
    octet b;  
};  
/* Je remplace « struct rgb » par « couleur » */  
typedef struct rgb couleur ;  
  
int main(void) {  
    couleur blanc = {255, 255, 255};  
    octet x = 200;  
    return 0;  
}
```

*.C

- ▶ On peut même définir l'alias lors de la création du type.

```
/* « struct rgb » → « couleur » */  
typedef struct rgb {  
    unsigned char r;  
    unsigned char g;  
    unsigned char b;  
} couleur;  
  
/* « enum feu » → « signalisation » */  
typedef enum feu {vert, orange, rouge} signalisation;  
  
/* « union lumiere » → « ampoule » */  
typedef union lumiere {  
    signalisation s;  
    couleur rgb;  
} ampoule;
```

*.c

- 🍃 Partie I. Makefile
- 🍃 Partie II. Tableaux et chaînes de caractères
- 🍃 Partie III. Types avancés
- 🍃 **Partie IV. Structures de contrôle avancées**
- 🍃 Partie V. Représentations binaires
- 🍃 Partie VI. Table des matières

► En C, le **for**, le **while** et le **if** sont suivies d'une seule instruction qui peut se présenter sous trois formes.

- Un bloc contenant une ou plusieurs instructions (recommandée)

```
while (i < 10) {  
    i++;  
}
```

*.C

- Une seule instruction (sans accolades) : déconseillé.

```
while (i < 10)  
    i++; /* l'indentation est purement cosmétique */
```

*.C

- une seule instruction, mais vide (noté avec un simple ;))

```
while (i++ < 10);
```

*.C

- ▶ Les blocs permettent de regrouper plusieurs instructions en une.
- ▶ Ils permettent aussi de définir des variables avec une portée locale.

```
int main(void) {  
    int i=3;  
    int j=4;  
    {  
        int i=10;  
        i++;  
        j++;  
        printf("%d %d\n",i,j); /* affiche « 11 5 »*/  
    }  
    printf("%d %d\n",i,j); /* affiche « 3 5 »*/  
    return 0;  
}
```

*.c

- ▶ On peut séparer plusieurs expressions par une virgule.
 - ▶ La nouvelle expression s'obtient en exécutant toutes les sous-expressions
 - ▶ Sa valeur est le résultat de la dernière expression

```
int main(void) {  
    int t[20];  
    int i;  
    int j;  
    for (i=0, j=19; i < j; i++, j--) {  
        t[i] = j;  
        t[j] = i;  
    }  
    i = (j=9, j++, j++); /* i vaut 10 */  
    return 0;  
}
```

*.C

- ▶ À ne pas reproduire ! C'est affreux !

```
int i=0;  
for(;; printf("%d\n",i),i++<10;);
```

*.C

- ▶ `condition ? expression_1 : expression_2`
 - ▶ Seul opérateur ternaire du langage
 - ▶ Correspond à un `if`
- Les deux affectations suivantes

```
x = (n%2) ? 10 : 20;  
y = (n == 0) ? 43 : (n == -1) ? 52 : 100;
```

*.C

- correspondent à :

```
if (n%2) x=10;  
else x=20;  
  
if (n==0) {  
    y=43;  
} else if (n==-1) {  
    y=52;  
} else {  
    y=100;  
}
```

*.C

- ▶ les **goto** : symboles décadents d'une époque révolue.
 - ▶ parfois utiles pour remplacer les exceptions.
 - ▶ très fortement déconseillés.

```
int main(void) {  
    int i = 0;  
  
debut:  
    printf("%d\n",i);  
    if (i == 10) {  
        goto fin;  
    }  
    i++;  
    goto debut;  
  
fin:  
    return 0;  
}
```

*.C

EDSGER W. DIJKSTRA Letters to the editor : Go To Statement Considered Harmful

Communications of the ACM Volume 11 Issue 3 March 1968 pp 147-148

<https://doi.org/10.1145/362929.362947>

- ▶ L'appel de fonction `exit(0)` permet de quitter le programme. Il faut pour cela importer `stdlib.h`.
- ▶ Le mot clé `return` permet de quitter la fonction.
- ▶ Le mot clé `break` permet de quitter une boucle
- ▶ Le mot clé `continue` permet de ne pas faire la fin du corps de boucle et reprend au début de ce dernier.

```
/* Qu'affiche le code suivant ? */  
i=0;  
while (1) {  
    if (i%2 == 0) {  
        i++;  
        continue;  
    }  
  
    if (i==10)  
        break;  
  
    printf("%d\n",i);  
    i++;  
}
```

*.C

- ▶ L'aiguillage (**switch**) permet de choisir rapidement entre des entiers.
 - ▶ Les étiquettes doivent être calculable à la compilation
 - ▶ Une étiquette ne peut pas contenir de variable.
 - ▶ Si aucune étiquette ne correspond, on n'entre pas dans le **switch**;

```
typedef enum feu {vert, orange, rouge} signalisation;
signalisation suivant(signalisation courant) {
    signalisation prochain;
    switch (courant) {
        case vert:
            prochain = orange ;
            break ; /* Attention à ne pas oublier le break ! */
        case orange:
            prochain = rouge ;
            break ;
        case rouge:
            prochain = vert;
            break ;
    }
    return prochain;
}
```

*.C

- ▶ Le **switch** fonctionne comme un **goto**
 - ▶ Il saute à une étiquette et exécute le code à partir de là;
 - ▶ Le **break** (ou **return**) est donc indispensable;
 - ▶ On peut ajouter une étiquette **default**;

```
typedef enum bool {OUI, NON, EUH} boule;
boule est_pair(char chiffre) {
    switch (chiffre) {
        case '0':
        case '2':
        case '4':
        case '6':
        case '8':
            return OUI;
        case '1':
        case '3':
        case '5':
        case '7':
        case '9':
            return NON;
        default : /* Ce n'est pas une chiffre */
            return EUH;
    }
}
```

*.C

- 🍃 Partie I. Makefile
- 🍃 Partie II. Tableaux et chaînes de caractères
- 🍃 Partie III. Types avancés
- 🍃 Partie IV. Structures de contrôle avancées
- 🍃 **Partie V. Représentations binaires**
- 🍃 Partie VI. Table des matières

► On se donne un entier positif b .

- Tout nombre $n \in \mathbb{N}$ peut s'écrire $n = \sum_{i=0}^k c_i \times b^i$, avec $\forall i \in \llbracket 0, k \rrbracket, c_i < b$.
- La liste des $(c_i)_{i \in \llbracket 0, k \rrbracket}$ est alors unique et on écrit $n = \overline{c_k \dots c_1 c_0}_b$

► Exemples :

- $1562 = 1 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 2 \times 10^0 = \overline{1562}_{10}$
- $1562 = 1 \times 7^4 + 0 \times 7^3 + 3 \times 7^2 + 2 \times 7^1 + 0 \times 7^0 = \overline{10320}_7$
- $1562 = 1 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^9 + \dots + 1 \times 2^1 + 0 \times 2^0 = \overline{101000000010}_2$
- $1562 = 10 \times 16^2 + 0 \times 16^1 + 2 \times 16^0 = \overline{A02}_{16}$

► Pour les chiffres plus grands que 10, on utilise les lettres :

- $A = 10, B = 11$, etc.

▶ Comment convertir 1562 en base 7?

- $1562 = 366 \times 7 + 0$

- $366 = 52 \times 7 + 2$

- $52 = 7 \times 7 + 3$

- $7 = 1 \times 7 + 0$

- $1 = 0 \times 7 + 1$

▶ On obtient donc $\overline{10320}_7$

▶ On obtient le quotient et le reste avec les opérations / et %

▶ On s'arrête lorsque le quotient est nul.

► Comment convertir $\overline{10320}_7$ en base 10 ?

- $\overline{1}_7 = 1$

- $\overline{10}_7 = 1 \times 7 + 0 = 7$

- $\overline{103}_7 = 7 \times 7 + 3 = 52$

- $\overline{1032}_7 = 52 \times 7 + 2 = 366$

- $\overline{10320}_7 = 366 \times 7 + 0 = 2562$

- ▶ Le C possède plusieurs opérations binaires bit à bit
 - ▶ La négation ou complément : `~`
 - ▶ Le « et » `&`, le « ou » `|` et le « ou exclusif » `^`
 - ▶ Les décalages à gauche et à droite. : `<<` et `>>`

```
~3      /* ~0000 0011 = 1111 1100*/  
3 & 5   /* 0000 0011 & 0000 0101 = 0000 0001 */  
3 | 5   /* 0000 0011 | 0000 0101 = 0000 0111 */  
3 ^ 5   /* 0000 0011 ^ 0000 0101 = 0000 0110 */  
3 << 5  /* 0000 0011 << 5 = 0110 0000 */  
73 >> 5 /* 0100 1001 >> 5 = 0000 0010 */
```

*.c

- ▶ On définit $-n$ comme l'unique nombre vérifiant $n + (-n) = 0$

- ▶ Supposons que l'on travaille avec un octet (8 bits).
 - $255 = \overline{1111\ 1111}_2$ et donc $255 + 1 = \overline{1\ 0000\ 0000}_2$
 - Mais la retenue finale disparaît (on travaille sur 8 bits).
 - On a donc $255 + 1 = 0$ et par suite $255 = -1$

- ▶ En pratique pour tout nombre n on a : $n + \sim n = \overline{11\dots11}_2$
 - et en particulier $n + \sim n + 1 = \overline{00\dots00}_2$
 - Finalement, on code $-n$ par le nombre $\sim n + 1$

Merci pour votre attention

Questions



Hello world !



Cours 2 — Types et structures de contrôle avancés

Partie I. Makefile

Compilation avec Makefile

Makefile : principes

La compilation avec Makefile en mieux!

Makefile : remarques

Partie II. Tableaux et chaînes de caractères

Tableaux

Tailles des tableaux

Tableaux et fonctions

Chaînes de caractères

Interro surprise!

Partie III. Types avancés

Énumérations : principes

Énumérations : fonctionnement

Structures : définitions

Structures : exemple

Structures : fonctions et modifications

Unions : principe

Unions : usage

Unions : exemple

Définir des alias de types

Usage de `typedef`

Partie IV. Structures de contrôle avancées

Instruction vide

Blocs

Opérateur virgule

Une autre syntaxe pour le `if`

Les goto

Interruptions

Switch

Switch

Partie V. Représentations binaires

Concept de base

Algorithmes de conversion 1/2

Algorithmes de conversion 2/2

Opérateurs binaires

Les nombres négatifs

Partie VI. Table des matières