UNIVERSITÉ PARIS DESCARTES

IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

T04 – Boucle `for`
T11 – Boucle `while`
T14 – Boucle `do ... while`
T18 – Choix des boucles
T19 – Ruptures de séquence
T25 – Type énuméré
T26 – Type pointeur
T27 – Tableau statique
T30 – Chaîne de caractères

Cours n° 2

Les fondamentaux de la programmation impérative

Les structures de contrôle (suite)

Sommaire

Rôles des itérations/boucles

Les instructions de contrôle de type itératif permettent d'écrire un programme qui exécute plusieurs fois le même bloc d'instructions

Elles présentent le risque (comme d'autres structures de contrôle*) d'écrire un programme dont le temps d'exécution peut poser un problème au sens d'un temps trop long, voire « infini »

Trois instructions de type itératif sont définies en C
- à itérations bornées (boucle `for`), le nombre d'itérations est connu
- à itérations non bornées (boucles `while` ... et `do` ... `while`), le nombre d'itérations n'est pas connu

Notion différée* – Fonctions et récursivité

Boucle `for` – Boucle à itérations bornées (1/5)

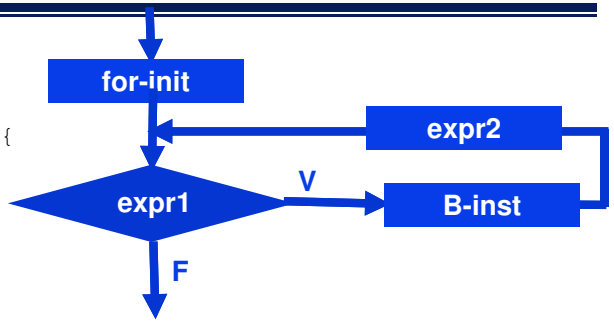
for ([<for-init>] ; [<expr>] ; [<expr>]) <bloc-inst>

<for-init>	les instructions (d'initialisation) exécutées avant l'exécution de la boucle
<expr>	expression (condition), si l'expression est vraie, la boucle exécute son corps (bloc-inst) sinon elle s'arrête
<expr>	les instructions exécutées à la fin de chaque itération (après exécution de bloc-inst)
<bloc-inst>	le boc des instructions exécutées à chaque itération
Remarque	La BNF montre toute l'élaboration de la boucle `for`
Usuellement	la boucle `for` permet un traitement itératif contrôlé par un mécanisme de variable d'itération (compteur) et une condition d'arrêt

Boucle `for` – Contrôle d'exécution (2/5)

Contrôle d'exécution

```
inst1;
for(for-init; expr1; expr2){
  B-inst;
}
inst2;
```



Flux d'exécution

inst1;
for-init
α si (expr1)
  B-inst;
  expr2
  aller en α
finSi
inst3;

Remarques

Les 3 premiers éléments de la boucle sont optionnels

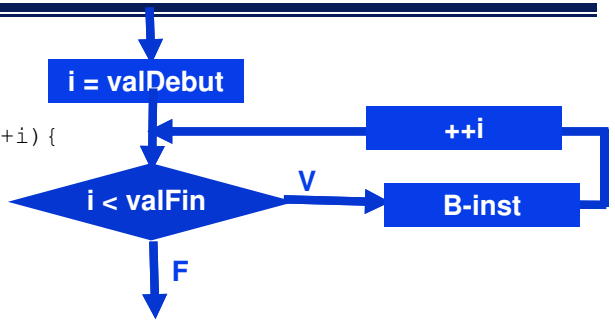
Sans la première expression (expr1),
une constante ≠ 0 est considérée,
la condition est toujours vraie

```
for ( ; ; ); // est ainsi une boucle infinie
```


Boucle `for` – Forme usuelle avec variable d'itération (3/5)

Contrôle d'exécution

```
inst1;
for(i=valDebut; i<valFin; ++i){
  B-inst;
}
inst3;
```



Flux d'exécution

inst1;
i ← valDebut
α si (i < valFin)
  B-inst;
  ++i;
  aller en α
finSi
inst3;

Remarques

(i) Ne jamais modifier la variable d'itération i (compteur) dans le corps de la boucle

(ii) Le nombre d'itérations est borné
dans notre cas de boucle où `i < valFin`,
si `ValFin` est strictement supérieur à `valDebut`,
le nombre d'itérations est `valFin-valDebut`
sinon aucune itération

Boucle `for` – Pseudo-code (4/5)

Le pseudo-code de la boucle `for`
correspond aux itérations bornées et à l'utilisation d'un compteur

Pseudo-code (forme avec variable d'itération)

pour i allant de 1 à n par pas de p faire
** bloc-inst**
finFaire

Le pas est omis s'il vaut 1

pour i allant de 1 à n faire
** bloc-inst**
finFaire

Le pas peut être négatif

pour i allant de n à 1 par pas de -1 faire
** bloc-inst**
finFaire

Remarque
Boucle la plus adaptée lorsque le nombre d'itérations est connu

Boucle `for` – Exemple (5/5)

Calculer la moyenne des 30 premiers entiers positifs

```
/* @file Cours2-Exo01.c */
#include "stdafx.h"

int main()
{
    int i;
    int nb;
    float cumul;
    for (i=1, nb=30, cumul=0.0; i<=nb; ++i) {
        cumul+=i; // eq. cumul=cumul+i;
    }
    printf("Moyenne des %d premiers entiers >0 %f\n", nb,
            cumul/nb);

    system("pause"); return 0;
}
```

Tests unitaires ?

`i=1, nb=30, i<=nb ,++i`
arrêt de boucle vérifié

Moyenne des 30 premiers entiers positifs 15.500000

Boucles `for` imbriquées (1/2)

Une instruction de boucle peut être une boucle

Exemple :

```
pour i allant de 1 à 10 faire
    pour j allant de 0 à i-1 faire
        écrire('O')
    finFaire
    écrire('K')
finFaire
```

Affichage attendu ?

```
Dans la boucle (i), pour i=1
   La boucle interne (j) varie de 0 à 0,
   son bloc est exécuté 1 fois
     ("O" est affiché)
   Arrêt de la boucle (j)
   "K" est affiché
la boucle (i) continue de s'exécuter,
i=2
...
```


Boucles `for` imbriquées – Exemple (2/2)

```c
/* @file Cours2-Exo02.c */
#include "stdafx.h"

int main() {
   int i,j;
   for(i=1; i<=10; ++i) {
      for(j=0; j<i; ++j)
         printf("%c", 'O');
      printf("%c\n", 'K');
   }

   system("pause"); return 0;
}
```

```
OK
OOK
OOOK
OOOOK
OOOOOK
OOOOOOK
OOOOOOOK
OOOOOOOOK
OOOOOOOOOK
OOOOOOOOOOK
```


Boucle `while` – Itérations non bornées (1/3)

Permet un traitement itératif contrôlé par une condition d'itération a priori

`while (<expr>) <bloc-inst>`

Pseudo-code

tantQue (condition) faire
**   bloc-inst**
finFaire

Remarque :

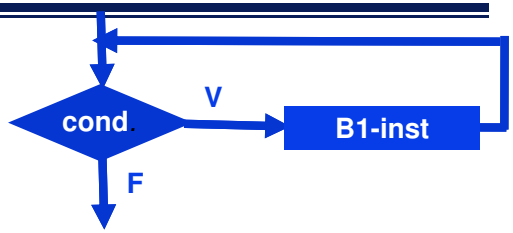
Boucle utilisée lorsque le nombre d'itérations n'est pas connu

L'exécution du bloc d'instruction est contrôlée par la condition
Si cette condition n'est pas satisfaite initialement, le bloc n'est pas exécuté

Boucle `while` – Contrôle d'exécution (2/3)

Contrôle d'exécution

```
     inst1;
1    while (condition) {
α      instB1;
       instB2;
     }
     inst3;
```



Flux d'exécution

inst1;
1. si (condition est vraie) faire
   // continuer en séquence (en α)
   instB1; instB2; // fin de bloc
   aller en 1.
   fin faire
   sinon inst3;

Remarque :

Une instruction (au moins) de la boucle doit avoir un effet de bord sur (modifier) la condition et tendre vers la condition d'arrêt

Boucle `while` – Exemple (3/3)

Et si `i` *était initialisé à 0 avant la boucle ?*

Déterminer le premier nombre entier n tel que la somme de 1 à n dépasse strictement 1000

L'entier n tel que 1+2+...+n > 1000 est : 45

```
/* @file Cours2-Exo03.c */
#include "stdafx.h"

int main() {
    int i, som;
    som=0;  1
    i=1;
    while (som <= 1000)  2
        som+=i; // eq. som=som+i;
        i=i+1;                          3
    }
    printf("n tel que 1+2+...+n > 1000 est : %d\n", i);

    system("pause"); return 0;
}
```

Principe du calcul incrémental
Une variable `som` *// somme demandée*
`som` *initialisée à* `0` *// élément neutre de l'addition*
À chaque itération : `i` *est additionnée à* `som`
`i` *est incrémenté de* `1`

{1, 2 ,3} : arrêt de boucle vérifié

Test
On sait que 1+2+...+n= n.(n+1)/2
Vérifier :
n.(n+1)/2 > 1000 et n.(n-1)/2 <= 1000

Boucle `do` – Itérations non bornées (1/3)

Permet un traitement itératif contrôlé par une condition d'itération a postériori

`do` <bloc-inst> **`while`** `(`<expr>`)`

Pseudo-code
faire
bloc-inst
tantQue (condition)

Remarque :

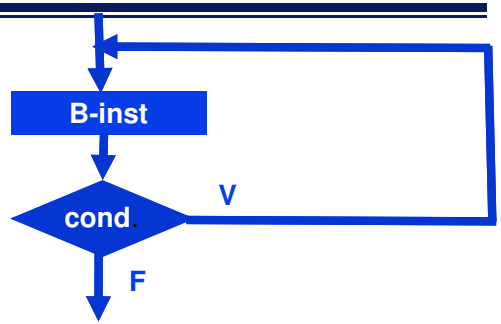
Boucle adaptée lorsque le nombre d'itérations n'est pas connu

Le bloc d'instruction est au moins exécuté une fois sans contrôle les autres itérations sont contrôlées par la condition

Boucle `do` – Contrôle d'exécution (2/3)

Contrôle d'exécution

```
    inst1;
    do {
1       B-inst1;
        B-inst2;
    } while (condition);
    inst3;
```



Remarque :

Une instruction (au moins) de la boucle doit avoir un effet de bord sur la condition (la modifier) et tendre vers la condition d'arrêt

Flux d'exécution

```
    inst1;
1   B-inst1; B-inst2; // fin de bloc
    si (condition est vraie)
        aller en 1
    sinon inst3;
```


Boucle `do` – Exemple (3/3)

En combien d'années serez-vous millionnaire ? En partant d'un capital nul et en investissant chaque année 1000 euros à 57% d'intérêt.

```
/* @file Cours2-Exo04.c */
#include "stdafx.h"

int main() {
    double capital;
    short nbAnnees;
    capital=0.0;   1
    nbAnnees=0;
    do {
        capital=capital+1000.;  2
        capital=(1+0.57)*capital;
        nbAnnees++;
    } while (capital<1000000.);  3
    printf("Millionnaire dans %d ans!!!\n", nbAnnees);

    system("pause"); return 0;
}
```

```
Millionnaire dans 14 ans!!!
```

Reste à trouver l'investissement…

1, 2 ,3 : arrêt de boucle vérifié

Tests unitaires ?

Flux d'exécution – (`for` et `while`) & (`do` et `while`)

D'après les flux d'exécution, la boucle `for`

```
for (<initialisation_s>; <condition>; <mise_s_à_jour>)
    <blocB>
```

peut s 'écrire sous la forme d'une boucle `while`

```
<initialisation_s>;
while (condition)
    <blocB>
    <mise_s_à_jour>
}
```

La boucle `do`

```
do
    <blocB>
while (<condition>)
```

peut s 'écrire sous la forme d'une boucle `while`

```
<blocB>
while (<condition>)
    <blocB>
```


Choix des boucles

La boucle `for` est à choisir pour les itérations bornées

Elle est conceptuellement élaborée, son écriture est compacte et ergonomique dans la localisation
(1) des initialisations d'entrée dans la boucle (au moins du compteur)
(2) de la condition d'itération
(3) de la mise à jour de fin d'itération (du compteur)
Le bloc `for` contient uniquement les instructions de traitement

La boucle `for` peut également être utilisées pour les itérations non bornées (cf. Transparent 34)

Les boucles `while` et `do` utilisées pour les itérations non bornées doivent être conçues par le programmeur avec la même rigueur
(1) des initialisations nécessaires avant l'entrée dans la boucle
(2) la mise à jour de fin d'itération dans le bloc instruction
(3) la mise à jour de fin d'itération doit faire converger la condition vers la condition d'arrêt de boucle
Le bloc instructions contient imbriquées : les instructions de traitement et de mise à jour de fin d'itération

Ruptures de séquence (1/2)

Instructions de rupture de séquence : `break et continue`

`break`

- Exclusivement dans une boucle ou une clause de sélection (case)
- Interrompt le flux d'exécution dans la boucle (la plus interne) ou l'alternative de sélection en provoquant un saut vers l'instruction suivant la structure de contrôle

`continue`

- Exclusivement dans une boucle
- Interrompt le flux d'exécution des instructions du bloc en provoquant un transfert d'exécution à l'itération suivante de la boucle
  Dans le cas d'une boucle for : mise à jour des variables d'itération et ré-évaluation de la condition d'arrêt

Bonne Pratique

Utiliser avec discernement `break` et `continue` (e.g., cas de recherche)

Ruptures de séquence (2/2)

Exemples d'utilisation

```
while (true) {  //boucle infinie
  instB1;
  …
  if (condition)
     break;
  instB2;
  …
}
inst3;
```

Code spaghetti ?
Utile dans des algorithmes de recherche

```
int borne=20;
for (int x=0; x<borne; ++x) {
   if (x%2)
      continue;
   // Traitements des entiers impairs
   instB1;
   …
}
```

Retour sur les types de données

les types énumérés

et les pointeurs

avant de passer aux tableaux et à leur traitement

Type énuméré – enum

Un type énuméré est un sous-type des entiers
Sa définition indique la liste des valeurs (constantes d'énumération) qu'une variable de ce type peut prendre

Déclaration d'un type énuméré

```
enum Color{BLUE, WHITE, RED};
```

Les constantes d'énumération ont des valeurs entières (à partir de 0) sauf si elles sont redéfinies

```
enum Color {BLUE, WHITE=3, RED, PINK=9};
// BLUE vaut 0, WHITE vaut 3, RED 4 et  PINK 9
```

Déclaration et initialisation de variable

```
enum Color c; // le type est enum Color
c=BLUE;
```

Permet de déclarer une constante (d'énumération) et de lui affecter une valeur

```
enum {N=5};
```


Type pointeur – Opérations élémentaires

Exemple pour une adresse mémoire sur 4 octets (processeur 32 bits) et un short sur 2 octets

Tout pointeur
(1) est une variable destinée à contenir une adresse mémoire
(2) est associé à un type d'objet

Déclaration du pointeur

L1 `short* pt_s;`

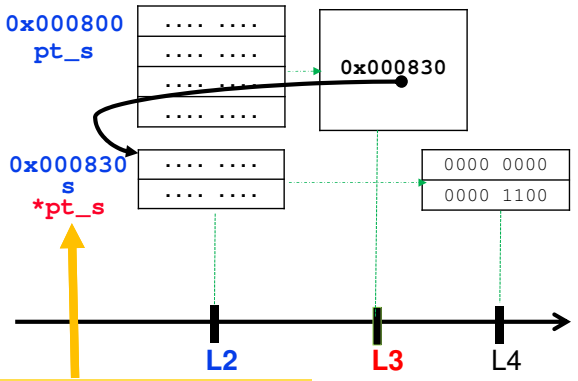
L2 `short s;`

(t1) Initialisation du pointeur

L3 `pt_s=&s;`

Déréférencement

L4 `*pt_s=12;`



*s et *pt_s sont deux noms qui accède au même emplacement mémoire*

Question
`short m;`
`m=*pt_s;`
Que vaut m ?

Tableaux statiques – Déclaration (1/3)

Un tableau est une collection d'objets tous du même type

Déclaration

```
short tab[5]; // un tableau de 5 éléments entiers (codés sur 2 octets)
```

Bonne Pratique **Utiliser une constante pour définir la taille du tableau**

```
const N=5; // Constantes non autorisées en C90
```

On utilisera une variable d'énumération de valeur 5 (cf. T22)

```
enum {N=5};

short tab[N];
```

Représentation en mémoire (? : état de la mémoire)

tab	?	?	?	?	?
	0	1	2	3	4

Index du tableau : {0, .., 4}
accès au 4ème élément par `tab[3]`

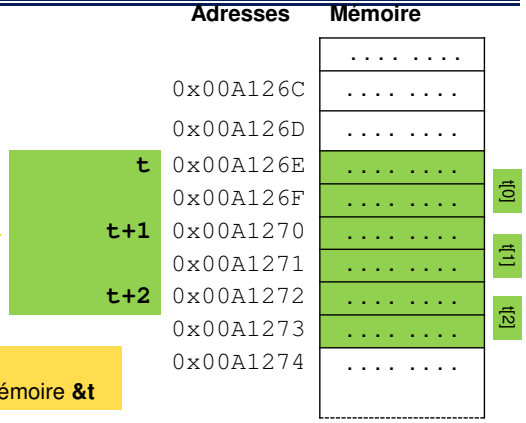
Schématiquement en mémoire

cf. TP
Affichage de tab (pointeur) et des 3 éléments du tableau (tab[i] pour i allant de 0 à 2) et de leur adresse (tab, tab+1, tab+2)

A la déclaration d'un tableau de 3 entiers de type `short`

```
short t[3];
```

Zone contigüe en mémoire des 3 éléments du tableau (arithmétique des pointeurs)



t = &t = &t[0]
t est l'adresse du tableau implanté en mémoire **&t**

t est également l'adresse du 1er élément du tableau **&t[0]**
t+1 est l'adresse du 2ème élément du tableau **&t[1]**
t+2 est l'adresse du 3ème élément du tableau **&t[2]**
(arithmétique des pointeurs)

.... état (aléatoire) de la mémoire

Tableaux statiques – Exemple (3/3)

```c
/* @file Cours2-Exo07.c */
#include "stdafx.h"

int main() {
    int i;
    enum {N=5}; // constante (d'enumeration) initialisee à 5
    int t1[N]={7, 2, 3, 6, 1}; // t1 initialisé par liste
    int t2[N];

    printf("Tableau t1 : ");
    for (i=0; i<N; ++i)
        printf("%d ", t1[i]);

    for (i=0; i<N; ++i) // t2 initialisé par boucle
        t2[i]=i+1;

    printf("\nTableau t2 : ");
    for (i=0; i<N; ++i)
        printf("%d ", t2[i]);

    system("pause"); return 0;
}
```

```
Tableau t1 : 7 2 3 6 1
Tableau t2 : 1 2 3 4 5
```


Chaînes de caractères – Littéral (1/3)

Représentation interne d'une chaîne de caractères littérale

```
printf("Hello, world!");
```

H	e	l	l	o	,		W	o	r	l	d	!	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Taille utile (13 octets)

Taille occupée en mémoire (14 octets)

Une chaîne est stockée dans un tableau de caractères

Attention

'a' est différent de `"a"`
'a' est un caractère stocké en mémoire sur un caractère
`"a"` est une chaîne de caractère stocké sur 2 caractères ('a' et '\0')

Chaînes de caractères – `strlen()` et `sizeof()` (2/3)

Une chaîne de caractères est stockée dans un tableau de caractères et est initialisée par un littéral chaîne

```
char s1[80] = "Hello!";
char s2[] = "Hello, you!";
```

A savoir

`strlen(...)` fonction de la bibliothèque `<string.h>`
`strlen(s1)` longueur utile de la chaîne s1 (sans le délimiteur '\0')

`sizeof(...)` opérateur de C
donne la taille mémoire (en octets) d'une variable/type

`strlen(s1)` vaut 6
`strlen(s2)` vaut 11

`sizeof(s1)` vaut 80
`sizeof(s2)` vaut 12
`sizeof(int)` vaut 4

Chaînes de caractères – Exemple (3/3)

```
strlen(s1)=6
sizeof(s1)=80
strlen(s2)=11
sizeof(s2)=12
```

```c
/* @file Cours2-Exo07.c */
#include "stdafx.h"

int main()
{
    char s1[80]="Hello!"; // initialisation par un littéral chaîne
    char s2[] = "Hello, you!";

    printf("strlen(s1)=%d\n", strlen(s1));
    printf("sizeof(s1)=%d\n\n", sizeof(s1));

    printf("strlen(s2)=%d\n", strlen(s2));
    printf("sizeof(s2)=%d\n", sizeof(s2));

    system("pause"); return 0;
}
```

strlen() taille utile d'une chaîne
sizeof() taille mémoire occupée par une variable ou un type

Bilan de ce que vous avez appris en cours

- Le rôle des itérations
- Les trois différents types de boucle en C, leur flux d'exécution et leurs ruptures de séquence
- L'importance de la boucle `for` pour les itérations bornées
- Ce qui doit guider le choix des boucles et les bonnes pratiques pour l'utilisation des boucles non bornées
- L'aide à la conception des boucles en vérifiant le critère d'arrêt
- Le type énuméré et le type pointeur pour l'introduction des tableaux
- La déclaration et la manipulation des tableaux statiques, et des chaînes de caractères ainsi que leur représentation mémoire

Au prochain cours…

La suite des traitements en programmation : les fonctions