# Plan

# Instructions

Un programme impératif est constitué d'une séquence d'instructions exécutées les unes après les autres.

## Définition

Une instruction correspond à une étape atomique dans le programme.
↪ toute instruction s'exécute complètement.

Une instruction C est une **expression** terminée par un point-virgule.

```
1 int a, c;
2 a=3;
3 a+5;
4 c=a+10;
```

# Bloc d'instructions

Toute séquence d'instructions doit être regroupée dans un **bloc** :

## Définition

Un bloc d'instructions est constitué d'une séquence d'instructions délimitées par des accolades

```
{ instr1; instr2; ...; instrn;}
```

Les blocs d'instruction peuvent être :

- imbriqués : `{ instr1; { instr2; instr3;} }`
- successifs : `{ instr1; instr2; } { instr3;instr4;}`

## Remarque

Le bloc d'instructions de la fonction `main` définit les instructions du programme.

# Instructions complexes

En général, les expressions mises en séquence d'instruction sont celles ayant un **effet de bord** sur l'environnement d'exécution :

- une déclaration de variable ;
- une affectation ;
- un retour de fonction (cf. instruction `return`) ;
- l'appel à une fonction d'entrée-sortie ;
- ...

Enfin, un jeu de **structures de contrôle** permet la création d'instructions complexes se rapprochant du langage algorithmique :

- les conditionnelles
- les itératives

# Plan

# Les entrées-sorties

Comme pour tout langage de programmation il est souhaitable de pouvoir interagir avec le programme :

- saisir des valeurs au clavier
- afficher des valeurs à l'écran

En C, les fonctionnalités d'entrée-sortie standards sont définies dans le fichier `stdio.h`.

```
1  #include <stdio.h>
2
3  int main() {
4      ...
5      return 0;
6  }
```

# Instruction d'affichage

## Définition

`printf("chaîne de contrôle", exp_1, ..., exp_n)`

- `printf` est le nom de la fonction d'écriture formatée sur la sortie standard (par défaut l'écran), fonction de la librairie C `stdio.h`
- `"chaîne de contrôle"` est une chaîne de caractères contenant le **texte à afficher** entrecoupé de $n$ **spécifications de format** %d, %f, %c , %s ... une pour chacune des `exp_i`
- $n \geq 0$ expressions dont on veut afficher la valeur

```
1  #include <stdio.h>
2  int main() {
3     printf("Evaluation d'un calcul :\n");
4     printf("\tla valeur de %d + %d est %d\n",2,3,2+3);
5     return 0;
6  }
```

# Fonctionnement de la fonction printf

Les spécifications de format spécifient :

- comment doit être affiché chaque expression `exp_i`
- le type attendu de chaque `exp_i`

Schéma d'exécution de `printf` :

1. Construction de la chaîne de caractères à afficher à partir de la chaîne de contrôle
   - par remplacement de chaque spécification de format par une séquence de caractères représentant la valeur de l'expression associée `exp_i`
   - les autres caractères de la chaîne de contrôle restent inchangés
2. Appel à une routine système d'entrée/sortie permettant l'écriture de la chaîne sur la sortie standard

# Remplacement des spécifications de format

- À une spécification de format est associé :
  - un type de donnée
  - une notation comme suite de caractères des valeurs de ce type

  *Exemple : à `%d` est associé :*
  - *type :* `int`
  - *notation : séquence de caractères numériques évent. précédée d'un -*

- Soit un couple *(spéc. format, expression)* le remplacement consiste à :
  1. évaluer l'expression $\Rightarrow$ valeur du type de l'expression
  2. si besoin conversion implicite de cette valeur en une valeur du type du format
  3. transformation de cette dernière valeur en une suite de caractères correspondant à la notation associée au format

# Les principales spécifications de format

| **format** | **paramètre convertit en** | **notation à l'affichage** |
|---|---|---|
| %d | int | suite de chiffres (avec signe) |
| %hd | short | suite de chiffres (avec signe) |
| %ld | long | suite de chiffres (avec signe) |
| %u | unsigned int | suite de chiffres |
| %hu | unsigned short | suite de chiffres |
| %lu | unsigned long | suite de chiffres |
| %f | float | notation décimale ou scientifique |
| %lf | double | notation décimale ou scientifique |
| %c | unsigned char | caractère |
| %s | char* | chaîne de caractères |
| %p | void * (adresse mémoire) | valeur hexadécimale de l'adresse |

Différentes options permettent de préciser ces formats (nombre de chiffres, affichage en octal...)

# Instruction de saisie clavier

## Définition

`scanf("chaîne de contrôle", adr_1, ..., adr_n)`

- `scanf` est le nom de la fonction de lecture formatée depuis l'entrée standard (par défaut le clavier)
- `"chaîne de contrôle"` est une chaîne de caractères contenant la chaîne à lire entrecoupée de **spécifications de format** des $n$ données à lire et stocker aux adresses mémoires `adr_i`
- $n \geq 1$ adresses de variables déclarées que l'on veut affecter

```
#include <stdio.h>
int main() {
  int a;
  printf("Veuillez saisir un entier : ");
  scanf("%d", &a);
  printf("Vous avez saisi la valeur %d\n",a);
  return 0;
}
```

# Tampons d'entrée/sortie

Pour éviter de trop nombreuses opérations physiques d'accès aux périphériques, des tampons d'entrée/sortie sont gérés par le système d'exploitation.

- Le système gère le remplissage du tampon d'entrée standard à partir du clavier
  - les caractères tapés au clavier ne sont introduits dans le tampon que lors d'un appui sur la touche <Entrée>
- La fonction `scanf` lit les caractères dans ce tampon
  - La lecture est **bloquante** : si aucun caractère à lire dans le tampon l'exécution du programme est bloquée jusqu'au remplissage du tampon

# Fonctionnement de la fonction scanf

On traite itérativement chaque élément de la chaîne de contrôle :

- Les caractères "simples" (pas les spécifications de format) doivent être lus tels quels sur l'entrée standard
- Pour les spécifications de format :
  1. la **plus longue séquence de caractères correspondant à ce format** est lue sur l'entrée standard,
  2. convertit en une valeur du type associé au format,
  3. cette valeur est alors stockée à l'adresse de la variable correspondante.

Exemple : `scanf("%d,%c",&i,&c)` cherche à lire un entier relatif puis une virgule et un caractère.

# Lecture/affectation d'une donnée

Cas d'erreurs

- La saisie s'interrompt (mais le programme continue) dès qu'un élément de la chaîne de contrôle n'est pas satisfait (c'est à-dire si les caractères sur l'entrée standard ne correspondent pas au caractère ou format de donnée attendu) $\Rightarrow$ les données restantes ne sont pas affectées.
- Si le type du format n'est pas compatible avec le type de la variable un débordement peut avoir lieu $\Rightarrow$ modification inattendue de la mémoire pouvant provoquer l'interruption du programme

Remarque : le format associé aux nombres (entier et réels) accepte qu'un nombre quelconque de caractères séparateurs préfixent ce nombre :

- tabulation
- retour à la ligne (touche <Entrée>)
- espace

Exemple : "  3.5" est une séquence de 9 caractères (6 espaces) qui peut être lue comme un flottant.

# Exemples de saisies

## exemple.c

```c
#include <stdio.h>
int main() {
    int i;
    float f;
    printf("Saisissez un entier et un reel\n");
    scanf("%d%f",&i,&f);
    printf("i vaut %d, f vaut %f\n",i,f);
    return 0;
}
```

Saisissez un entier et un réel
35 23.45e-4
i vaut 35, f vaut 0.002345

Saisissez un entier et un réel

23
2.3
i vaut 23, f vaut 2.300000

Saisissez un entier et un réel
val 23 2.3
i vaut 0, f vaut 0.000000

Saisissez un entier et un réel
2.3
i vaut 2, f vaut 0.300000

Saisissez un entier et un réel
.3 2.4
i vaut 0, f vaut 0.000000

# Plan

# Types de structures alternatives

Les conditionnelles sont des structures de contrôle permettant d'exécuter une séquence d'instructions sous condition d'une expression booléenne.

On distingue 3 types d'instructions conditionnelles en C :

- le **si simple** qui n'exécute une séquence d'instruction que si la condition est remplie
- le **si sinon** qui selon que la condition est vraie ou fausse exécute une séquence ou une autre.
- le **choix multiple** qui selon une expression entière e continue l'exécution à partir de l'instruction étiquetée par la valeur $v_e$ de cette expression.

# **if...else** et **if** simple

## Définition

```
if (exp)
   instr1
else
   instr2
```

- *exp* est une expression booléenne ou implicit. convertie en `_Bool` !
- $instr_1$ et $instr_2$ sont :
  - une instruction
  - ou un bloc d'instructions

## Remarque

Le `else` est facultatif, la forme la plus simple est donc :

```
if (exp) instr
```

# Exemple: if ... else

## Exemple

Déterminer si un entier est pair ou impair

```c
#include <stdio.h>

int main() {
   int a;
   a = 17;
   if (a%2 == 0)
      printf("%d est pair\n", a);
   else
      printf("%d est impair\n", a);
   return 0;
}
```

# Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolades pour définir un bloc d'instructions :

```
1  ...
2     if (a > b){
3         max = a;
4         min = b;
5     }
6     else
7         max = b;
8         min = a; // ATTENTION: n'appartient pas au else
9
10    printf("le min est %d, le max est %d\n",min, max);
11 ...
```

## Exemple

Si a vaut 5 et b vaut 2, ce code affichera :

```
le min est 5, le max est 5
```

# Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolades pour définir un bloc d'instructions :

```
1  ...
2     if (a > b)
3         max = a;
4         min = b;
5     else {
6         max = b;
7         min = a; // grace aux accolades appartient au else
8     }
9     printf("le min est %d, le max est %d\n",min, max);
10 ...
```

mais l'oubli des accolades sur le bloc du `if` génère une erreur de compilation car le `else` se retrouve isolé.

# Structures de contrôle : erreur classique

Avec les accolades, tout va bien :

```
1  if (a > b) {
2      max = a;
3      min = b;
4  }
5  else {
6      max = b;
7      min = a;
8  }
9  printf("le min est %d, le max est %d\n",min, max);
```

On peut toujours mettre des accolades même quand on a une seule instruction à faire :

```
1  if (a < 0) {
2      a = -a;
3  }
```

# `if...else` imbriqués : l'indentation n'est pas interprétée !

```
if(exp1)
   inst1
   if(exp2)
        instr2
   else
        instr3
```

est identique à

```
if(exp1)
   inst1
if(exp2)
        instr2
else
    instr3
```

et signifie

**si** $exp_1$ **alors**
  $inst1$
**finsi ;**
**si** $exp_2$ **alors**
  $instr_2$
**sinon**
  $instr_3$
**finsi**

pour exprimer

**si** $exp_1$ **alors**
  $inst1$ **;**
  **si** $exp_2$ **alors**
    $instr_2$
  **sinon**
    $instr_3$
  **finsi**
**finsi**

on doit créer un bloc

```
if(exp1)
{
   inst1
   if(exp2)
        instr2
   else
        instr3
}
```

# `if...else` imbriqués : faire des blocs si besoin

```
if(exp1)
    if(exp2)
        instr2
    else
        instr3
```

est identique à

```
if(exp1)
    if(exp2)
        instr2
else
    instr3
```

et signifie

**si** $exp_1$ **alors**
  **si** $exp_2$ **alors**
    $instr_2$
  **finsi**
**sinon**
    $instr_3$
**finsi**

où est $instr_1$ ?

pour exprimer

**si** $exp_1$ **alors**
  **si** $exp_2$ **alors**
    $instr_2$
  **sinon**
    $instr_3$
  **finsi**
**finsi**

on doit créer un bloc

```
if(exp1) {
    if(exp2)
        instr2
    else
        instr3
}
```

# Conseils pour les `if...else` imbriqués

**1- Observer l'indentation automatique faite par les éditeurs de code qui montre les portions de code "incluses" !**

l'indentation automatique écrira

```
if(exp1)
    if(exp2)
        instr2
else
    instr3
```

et non ça qui est fallacieux

```
if(exp1)
    if(exp2)
        instr2
    else
        instr3
```

**2- Systématiquement créer des blocs qui permettent de bien visualiser le code "inclus" !**

la version équivalente au code du dessus :

```
if(exp1) { if(exp2) { instr2 } } else { instr3 }
```

l'autre parenthésage possible

```
if(exp1) { if(exp2) { instr2 } else { instr3 } }
```

# Exemple: if...else

## Exemple

Déterminer le maximum de 3 entiers

```
1  #include <stdio.h>
2  int main() {
3     int  a, b, c, max;
4     scanf("%d%d%d",&a,&b,&c);
5     if (a > b) {
6        if(a > c)
7           max = a;
8        else
9           max = c;
10    }
11    else if (c > b)
12       max = c;
13    else
14       max = b;
15    printf("le max est %d\n", max);
16    return 0;
17 }
```

# Conditionnelle à choix multiple : `switch`

Lorsqu'on a besoin de faire un choix parmi plus de 2 possibilités, on peut

- utiliser la conditionnelle `if .. else`

```
1  unsigned int a;
2  scanf("%d", &a);
3  if (a==1)
4      printf("a=1\n");
5  else if (a==2)
6      printf("a=2\n");
7  else if (a==3)
8      printf("a=3\n");
9  else
10     printf("a>3\n");
```

- faire une conditionnelle à choix multiple

# Conditionnelle à choix multiple : `switch`

## Choix multiple `switch`

```
switch (exp) {
  case cst1:
       instr1; break;
  ...
  case cstn:
       instrn; break;
  default:
       instrdefault;
}
```

- **exp** est une expression à valeur entière
- **cst1, ..., cstn** sont des constantes entières
- **instr1, ..., instrn, instrdefault** sont des séquences d'instructions terminées par un `break;`

Les instructions sont exécutées en fonction de la valeur **exp**

Si la valeur de **exp** est une des constantes **cst1,...,cstn** on exécute la suite d'intructions correspondante sinon on exécute **instrdefault**.

# Exemple : `switch`

```
1  unsigned int a;
2  scanf("%d",&a);
3  switch (a){
4    case 1:
5        printf("a=1\n"); break;
6    case 2:
7        printf("a=2\n"); break;
8    case 3:
9        printf("a=3\n"); break;
10   default:
11       printf("a>3\n"); break;
12 }
```

L'affichage sera

- a=1 si la valeur de a est 1
- a=2 si la valeur de a est 2
- a=3 si la valeur de a est 3
- a>3 si la valeur de a est différent de 1, 2 ou 3

# Exemple : `switch`

On peut grouper les cas, ne pas mettre le `break`, ni le `default`.

```c
#include <stdio.h>
int main() {
  unsigned char n;
  printf("Saisir un nombre de 1 a 10\n");
  scanf("%d",&n);
  switch (n) {
    case 2 :
    case 4 :
    case 8 :
      printf("c'est une puissance de 2\n");
    case 6 :
    case 10 :
      printf("il est pair\n");
      break :
    case 1 : case 3 : case 5 : case 7 : case 9 :
      printf("il est impair\n");
  }
  return 0;
}
```

# Plan

# Types de structures à boucle

Les itératives sont des structures de contrôle permettant d'exécuter une séquence d'instructions plusieurs fois.

On distingue 2 types d'instructions itératives :

- la boucle **pour** qui permet de répéter un nombre de fois déterminé a priori une séquence.
- la boucle **tant que** qui permet de répéter une séquence tant qu'une condition reste vérifiée

# Répétition d'instructions : la boucle pour

Intérêt : répéter un nombre de fois donné une même suite d'instructions.

## Exemple

**calcul de la somme des entiers entre 1 et 10**

```
s := 0 ;
pour i de 1 à 10 (par pas de 1) faire
    s := s+i ;
fin pour ;
```

En C, les boucles **pour** peuvent être réalisées avec l'instruction `for`

# Utilisation de l'instruction `for` pour réaliser un **pour**

## Définition

**for** (`exp1 ; exp2 ; exp3`)
`instr`

- `exp1` est une expression quelconque évaluée une seule fois au début de la boucle. Elle permet d'initialiser l'indice de boucle.
- `exp2` est une expression booléenne évaluée au début de chaque tour de boucle. Elle définit la poursuite/l'arrêt de la boucle.
- `exp3` est une expression quelconque évaluée à la fin de chaque tour de boucle. Elle permet d'incrémenter l'indice de boucle.

- `instr` est une instruction ou un bloc d'instructions

# L'instruction `for` pour un **pour**

## Définition

**for** (`exp1 ; exp2 ; exp3`)
`instr`

- `exp1` est une expression quelconque évaluée une seule fois au début de la boucle (souvent une affectation)

On se sert de exp1 pour initialiser la variable de boucle.

## Exemple

**exp1** est remplacée par `int i=1` (ou `i=1` si `i` est déjà déclaré).

# L'instruction `for` pour un **pour**

## Définition

**for** (`exp1 ; exp2 ; exp3`)
`instr`

- `exp2` est une expression booléenne

Si `exp2` est :

- vrai : la boucle `for` continue et on exécute `instr`
- faux : on sort de la boucle `for` sans exécuter `instr`

## Exemple

**exp2** est remplacée par `i<11` ou `i<=10`

# L'instruction `for` pour un **pour**

## Définition

**for** (`exp1 ; exp2 ; exp3`)
`instr`

- `exp3` est une expression quelconque évaluée à chaque tour de boucle (après exécution de `instr` mais avant réévaluation de `exp2`) ;
- on passe à la valeur suivante de l'indice de boucle.

## Exemple

**exp3** est remplacée par `i=i+1` ou `i++`

# L'instruction `for` pour un **pour** : retour à l'exemple

## Exemple

calculer la somme des entiers entre 1 et 10

```
1  #include <stdio.h>
2
3  int main() {
4     int i,s;
5     s = 0;
6     for (i=1 ; i<11 ; i++) // i++ est equivalent a i=i+1
7        s = s+i;
8     printf("La somme vaut %d\n",s);
9     return 0;
10 }
```

ce programme affiche : La somme vaut 55

# L'instruction `for` : le schéma général d'exécution

## Définition

**for** (`exp1 ; exp2 ; exp3`)
`instr`

1. évaluation de `exp1`
2. évaluation de `exp2` :
   - si `exp2` est faux `sortie de boucle`
   - si `exp2` est vrai :
     3. évaluation de `instr`
     4. évaluation de `exp3`

   on recommence en 2

# Du pour d'algorithmique au `for` du C

## pour haut

pour i de *a* à *b* par pas de *k* faire
    instr
fin pour ;

```
1 for(int i=a ; i<=b ; i=i+k) {
2     instr
3 }
```

## pour bas

pour i de *b* bas *a* par pas de *k* faire
    instr
fin pour ;

```
1 for(int i=b ; i>=a ; i=i-k) {
2     instr
3 }
```

# L'instruction `for` : exemple de pour descendant

## Exemple

afficher les entiers entre 10 et 1 qui sont multiples de 2 ou de 3

```c
#include <stdio.h>

int main(){
   for (int i=10;i>0;i--) {
      if (i%2==0 || i%3==0)
         printf("%d ",i);
   }
   printf("\n");
   return 0;
}
```

Ce programme affiche : 10 9 8 6 4 3 2

**Remarque** : on peut déclarer la variable de boucle dans `exp1` ; dans ce cas, on ne peut plus y accéder après la boucle.

# L'instruction `for` : ce qu'il ne faut pas faire

Attention aux boucles qui ne se terminent jamais ! ! !
les boucles infinies ...

```
1   int i,s;
2   s=0;
3   for (i=1 ; i<11 ; s=s+i)
4       ...
```

→ la variable de boucle n'est pas incrémentée

```
1   int i;
2   for (i=1 ; i!=10 ; i+=2)
3       ...
```

→ la condition d'arrêt de la boucle n'est jamais atteinte

# Répétition d'instructions : la boucle tant que

Intérêt : répéter une instruction tant qu'une condition est vérifiée

## Exemple

calculer la somme des entiers entre 1 et 10

```
s := 0 ;
i := 1 ;
tant que i<11 faire
    s := s+i ;
    i := i+1 ;
fin tant que ;
```

En C, les boucles **tant que** se font avec l'instruction `while`

# L'instruction `while`

## Définition

**while** (`exp`)
  `instr`

- `exp` est une expression booléenne contrôlant la poursuite de la boucle
- `instr` est une instruction ou un bloc d'instructions qui doit agir sur la valeur de `exp` pour qu'une sortie de boucle soit possible

# L'instruction `while` : schéma d'exécution

## Définition

**while** (`exp`)
  `instr`

1. évaluation de `exp` :
   - si `exp` est faux `sortie de boucle`
   - si `exp` est vrai :
     2. évaluation de `instr`

   on recommence en 1

# L'instruction `while` : exemple 1 - simulation d'un pour

## Exemple

calculer la somme des entiers entre 1 et 10

```
1  #include <stdio.h>
2
3  int main(){
4     int i,s;
5     s = 0;
6     i = 1;
7     while (i<11) {
8        s = s+i;
9        i = i+1;
10    }
11    printf("la somme est %d\n",s);
12    return 0;
13 }
```

ce programme affiche : la somme est 55

# L'instruction `while` : ex. avec nb. d'itérations non connu

## Exemple

calcul de la plus petite puissance de 2 supérieure à un entier

```
1  #include <stdio.h>
2
3  int main(){
4     int a, p = 1;
5     printf("Tapez un entier");
6     scanf("%d",&a);
7     while (p<a)
8        p = 2*p;
9     printf("%d est la plus petite puiss. de 2 sup. a %d",p,a);
10    return 0;
11 }
```

Pour 27, il affiche : 32 est la plus petite puiss. de 2 sup. a 27.

Mais il peut ne pas s'arrêter si la plus petite puissance de 2 supérieure à l'entier saisi est supérieure au plus grand entier codable !

# L'instruction `do while`

Parfois, il est souhaitable d'exécuter le corps de boucle avant la condition de boucle (`instr` avant `exp`).

Dans ce cas, on peut utiliser l'instruction `do ... while`

## Définition

```
do {
    instr
} while (exp);
```

- `instr` et `exp` sont identiques à ceux utilisés dans la boucle `while` classique

# L'instruction `do{} while` : schéma d'exécution

## Définition

```
do {
   instr
} while (exp);
```

1. évaluation de `instr`
2. évaluation de `exp` :
   - si `exp` est faux `sortie de boucle`
   - si `exp` est vrai on recommence en 1

# L'instruction do{} while : exemple

## Exemple

Le jeu du trouve mon code secret !

```
1  #include <stdio.h>
2
3  int main() {
4      int secret = 135;
5      int rep;
6      do {
7          printf("Ta proposition : ");
8          scanf("%d",&rep);
9          if(rep < secret) printf("\tplus grand !\n");
10         if(rep > secret)printf("\tplus petit !\n");
11     }
12     while (rep!=secret);
13     printf("Bravo, tu as trouve !\n");
14     return 0;
15 }
```

# boucle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolades pour définir un bloc d'instruction :

```
1  ...
2    int i,s;
3    s = 0;
4    i = 1;
5    while (i<11)
6        s = s+i;
7        i++; // n'appartient pas a la boucle
8
9    printf("la somme est %d\n",s);
10 ...
```

Le programme ne s'arrête pas $\Rightarrow$ boucle infinie !

# boucle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolades pour définir un bloc d'instruction :

```
1  ...
2     int i,s;
3     s = 0;
4     i = 1;
5     while (i<11) {
6         s = s+i;
7         i++; // grace aux accolades appartient a la boucle
8     }
9     printf("la somme est %d\n",s);
10 ...
```

# Récapitulatif

- les variables ont un type (ex : int, float...)
- on peut calculer grâce aux opérateurs (+,*,%,...)
- on modifie un programme par des affectations (ex : `a=6` )
- faiblement typé → conversions de type implicite
- on affiche les valeurs à l'écran avec `printf`
- on saisit les valeurs au clavier avec `scanf`
- on peut écrire des algorithmes avec les structures de contrôle classiques : `if else`, `switch`, `for`, `while`, `do...while`