

## Chapitre 2 : Notions d'Algorithmique et de Programmation en C

12 septembre 2023

# Langage C

Langage utilisé : C

- Langage **bas-niveau**
- créé dans les années 1970 ; populaire depuis longtemps
- langage **impératif**

# Langage C

Paradigmes de programmation :

- **Impératif** : opérations = *séquences d'instructions* modifiant l'état d'un programme
- **Fonctionnel** : calculs = évaluations de fonctions mathématiques

# Variables et Types

Données stockées en mémoire représentées par des **variables**

Une variable est caractérisée par :

- son **identifiant** (nom donné dans le programme)
- son **type**
- sa **valeur**

# Variables et Types

Pour affecter une valeur à une variable en C, il faut :

- **déclarer** la variable en indiquant son type : `int v;`
- lui **assigner** sa valeur : `v = 5;`

Possible de faire les deux simultanément : `int v = 5;`

# Types Élémentaires en C

- ❶ `int` pour les **nombres entiers**
- ❷ `float` pour les « nombres à virgule flottante » (ou, par abus de langage, nombres **flottants**)
- ❸ `bool` pour les **booléens** (en important `stdbool.h`) :  
`true` et `false` (sans `stdbool.h` : 0 et 1)
- ❹ `char` pour les **caractères**

# Types Elémentaires en C

Le langage C est un langage à **typage statique**, c'est-à-dire que le type d'une variable est connu au moment de la compilation et doit être explicité par le programmeur. Cela permet notamment de détecter les erreurs de type au moment de la compilation, avant l'exécution du code.

# Fonctions

- TP01 : instructions tapées dans fonction `main`
- Possible de créer d'autres fonctions, appelées dans le programme

Intérêt : réutiliser un même bloc d'instructions plusieurs fois



# Fonctions

## Définition

Définition La **définition** d'une fonction est la donnée de :

- sa **signature** (type de retour, nombre et types de ses paramètres en entrée)
- son **corps** (bloc d'instructions)

# Syntaxe des Fonctions

La syntaxe d'une fonction en C est la suivante :

```
type_R  nom_fonction (type_1  arg_1 , ... )  
  {  
    // [bloc d'instruction]  
    return valeur_R ;  
  }
```

# Syntaxe des Fonctions

- `valeur_R` correspond à la valeur de retour et `type_R` à son type ;
- `nom_fonction` correspond à l'identificateur de la fonction ;
- `arg_1, ...arg_N` correspondent aux paramètres en entrée de la fonction, et `type_1, ...type_N` à leurs types.

## Exemple : PGCD

```
int pgcd(int a, int b)
{
    int r;
    while (b != 0){
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

# Signature

La signature de la fonction (aussi appelée type, par abus de langage) est indiquée dans l'*En-tête* : la fonction prend en entrée un couple d'entiers et renvoie un entier. Elle peut se noter  $\text{int} \times \text{int} \rightarrow \text{int}$ .

# Remarques

- Possibilité d'avoir des `return` ailleurs qu'à la fin de la fonction (attention à en avoir à chaque chemin emprunté par la fonction : graphe du flot de contrôle)
- Fonctions qui ne renvoient rien : type `void`

# Exercice

Exercice : créer un programme qui dans lequel on définit une fonction pgcd, puis on l'appelle et l'affiche sur des variables ayant comme valeurs 35 et 21.

# Exercice

```
// int pgcd(...) {}  
  
int main() {  
    int c = 35;  
    int d = 21;  
    int e = pgcd(c,d); // appel de pgcd  
    printf("%d", e);  
    return 0;  
}
```



# Vocabulaire

- pgcd est **appelée** dans la fonction main
- a et b sont les **paramètres** de la fonction pgcd
- c et d sont les **arguments** sur lesquels s'applique la fonction pgcd.

# Portée des Variables

## Définition

Définition En informatique, la **portée** d'un identifiant correspond à la partie d'un programme au sein de laquelle l'identifiant existe et peut être utilisé.

- variables **locales** : déclarée à l'intérieur d'une fonction, existe seulement pendant l'appel de la fonction
- variables **globales** : déclarées à l'extérieur d'une fonction, existent dans tout le programme

# Portée des Variables

```
#include <stdio.h>
```

```
void auxiliaire(){  
    int x = 50;  
}
```

```
int main(){  
    auxiliaire();  
    printf("%d",x);  
}
```

# Portée des Variables

Message d'erreur : la variable x appelée à la ligne  
`printf("%d",x);` n'est pas déclarée, car le x de la fonction  
auxiliaire n'existe plus une fois l'appel à la fonction  
auxiliaire terminé.

# Portée des Variables

```
#include <stdio.h>
```

```
int x;
```

```
void  auxiliaire(){  
    x = 50;  
}
```

```
int  main(){  
    auxiliaire();  
    printf("%d",x);  
}
```

# Passage en Valeur

Que se passe-t-il si l'on crée une fonction auxiliaire qui prend un paramètre en entrée et modifie sa valeur, et que l'on appelle cette fonction sur un argument donné ?

# Passage en Valeur

```
#include <stdio.h>
```

```
void auxiliaire(int x){  
    x = x + 10;  
}
```

```
int main(){  
    int y = 50;  
    auxiliaire(y);  
    printf("%d",y);  
}
```

# Passage en Valeur

Lorsqu'on appelle une fonction de paramètre  $x$  sur un argument  $y$  :

- Une variable temporaire  $x$  est créée, à laquelle on affecte la valeur de l'argument  $y$  ;
- Les instructions de la fonction sont appliquées sur la variable  $x$  ;
- La variable  $x$  cesse d'exister une fois l'appel terminé.



# Exemples

PGCD :

- a et b sont créées ;
- a et b prennent les valeurs des arguments c et d (35 et 21) ;
- La variable a vaut 7 à la fin des calculs, et est renvoyée par la fonction ;
- La valeur 7 est affectée à e ;
- a et b disparaissent

# Exemples

Programme précédent :

- $x$  est créée ;
- $x$  prend la valeur de l'argument  $y$  (50) ;
- $x$  prend la valeur  $x+10$  et vaut 50 ;
- $x$  disparaît ;  $y$  est inchangée

# Exemples

Cette procédure s'appelle le **passage par valeur**. Comme on l'a vu, **la fonction appelée ne peut pas modifier directement la valeur des arguments**, les opérations étant faites sur une copie de ceux-ci.

Une autre syntaxe permet à une fonction de modifier la valeur de ses arguments. On parle alors de **passage par adresse** : c'est ce que l'on a fait avec la fonction `scanf`, où les arguments étaient notés `&x` au lieu de `x`.

# Instructions Conditionnelles

Instructions conditionnelles : exécutées seulement lorsqu'une condition (booléens) est vérifiée

Syntaxe des conditions : cf TP 01

# Boucles

Boucles : exécutées à plusieurs reprises tant qu'une condition reste vérifiée

Condition : peut dépendre d'une variable dont on fera varier la valeur dans la boucle

Il existe deux types de boucles en C : les boucles non-bornées (`while`) et les boucles bornées (`for`)

# Boucles While

Les boucles `while` ("tant que" en anglais) permettent de répéter un bloc d'instructions tant qu'une condition est vérifiée. A chaque itération de la boucle, le programme ré-évalue la condition, lance une nouvelle itération si elle est vérifiée et sort de la boucle sinon.

# Syntaxe des Boucles While


La syntaxe d'une boucle `while` est la suivante :

```
while (condition)
{
    [bloc d'instructions]
}
```

# Boucles While

La syntaxe d'une boucle `while` est la suivante :

## Attention



Rien ne garantit a priori que la condition va cesser d'être vérifiée à un moment, et qu'on va sortir de la boucle. Si ce n'est pas le cas, la boucle sera infinie et le programme ne s'arrêtera pas sans intervention de l'utilisateur.



# Boucles While

Le programme ci-dessous a une boucle infinie :

```
#include <stdio.h>

int main(){

    int x = 5;
    while (x>0){
        printf("%d \n", x);
    }
}
```

# Incrémentation et Compteurs

**Incrémentation** : augmentation d'une unité de la valeur d'une variable

Syntaxe possible : `variable = variable + 1` ou `variable++`

**Décrémentation** : `variable--`

Incrémentation/Décrémentation d'autre chose qu'une unité : `+=` et `-=` (exemple : `variable += 2`)

# Incrémentation et Compteurs

Incrémentations : utilisées dans boucles avec **compteur** qui varie entre une valeur initiale et une valeur finale

Exemple :

```
#include <stdio.h>
```

```
int main(){
```

```
    int compteur = 10;
```

```
    while (compteur<=20){
```

```
        printf("%d \n", compteur);
```

```
        compteur++;
```

```
    }
```

```
}
```

# Boucles For

L'utilisation d'une boucle avec un compteur dont la valeur varie est fréquente en informatique. Elles ont trois caractéristiques :

- L'**initialisation** de la valeur d'une variable compteur ;
- La **condition** de la boucle while, indiquant la valeur maximale (ou minimale) à atteindre pour cette valeur ;
- L'**incrément** (ou la décrémentation) après l'exécution des autres instructions à l'intérieur de la boucle.

# Boucles For

Les **boucles** for permettent d'avoir une écriture plus condensée pour ce type de boucles avec compteur. Leur syntaxe est la suivante :

```
#include <stdio.h>
```

```
int main(){
```

```
    for (initialisation ; condition ; incrementa  
        [instructions]  
    }
```

```
}
```

# Boucles For

Exemple : programme équivalent au programme précédent, utilisant la boucle for :

```
#include <stdio.h>

int main(){

    int x;
    for (x = 10 ; x<=20 ; x++){
        printf("%d \n",x);
    }
}
```