C si facile

Prof d'info

Lycée Thiers

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution
  - Opérateurs arithmétiques et bouléens

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution
  - Opérateurs arithmétiques et bouléens
  - Expressions conditionnelles

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution
  - Opérateurs arithmétiques et bouléens
  - Expressions conditionnelles
  - Boucle `while`

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution
  - Opérateurs arithmétiques et bouléens
  - Expressions conditionnelles
  - Boucle `while`
  - Déclarations et appels de fonctions simples.

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution
  - Opérateurs arithmétiques et bouléens
  - Expressions conditionnelles
  - Boucle `while`
  - Déclarations et appels de fonctions simples.
  - Déclaration de tableaux statiques à une dimension.

À propos

- Ces transparents résument en quelques lignes de code le minimum à connaître pour s'en sortir en C dans les TD de début d'année.
- Bien qu'il existe souvent plusieurs syntaxes équivalentes pour une même instruction, on se limite à une seule.
- Thèmes abordés :
  - Variables : déclaration, affectation, affichage, saisie au clavier
  - Compilation de projet en un fichier, exécution
  - Opérateurs arithmétiques et bouléens
  - Expressions conditionnelles
  - Boucle `while`
  - Déclarations et appels de fonctions simples.
  - Déclaration de tableaux statiques à une dimension.
  - Assertions

Squelette

```c
#include <stdio.h>// charger bibli. d'entrées-sorties

int main(void){
   /*
     votre code ici
    */
   return 0; // on renvoie un entier (car 'int main')
}
```

Squelette

```c
#include <stdio.h>// charger bibli. d'entrées-sorties

int main(void){
   /*
     votre code ici
    */
   return 0; // on renvoie un entier (car 'int main')
}
```

- Commentaires en une ligne `// mon commentaire`. ET
  Commentaires sur plusieurs lignes `/* mes commentaires */`

Squelette

```c
#include <stdio.h>// charger bibli. d'entrées-sorties

int main(void){
   /*
     votre code ici
    */
   return 0; // on renvoie un entier (car 'int main')
}
```

- Commentaires en une ligne `// mon commentaire`. ET Commentaires sur plusieurs lignes `/* mes commentaires */`
- La fonction `main` renvoie un entier (en général). Cet entier est 0 par défaut (tout s'est bien passé). Autres valeurs renvoyées : code pour signaler un problème.

Squelette

```c
#include <stdio.h>// charger bibli. d'entrées-sorties

int main(void){
   /*
     votre code ici
    */
   return 0; // on renvoie un entier (car 'int main')
}
```

- Commentaires en une ligne `// mon commentaire`. ET Commentaires sur plusieurs lignes `/* mes commentaires */`
- La fonction `main` renvoie un entier (en général). Cet entier est 0 par défaut (tout s'est bien passé). Autres valeurs renvoyées : code pour signaler un problème.
- Une et une seule fonction **main** par programme (en cas de code sur plusieurs fichiers, un seul contient un **main**).

Déclaration, initialisation

Ouvrir un éditeur de texte comme **emacs ou gedit** (Linux) ou **blocnote** (Windows) MAIS PAS UN TRAITEMENT DE TEXTE.

```c
// dans un fichier exemple.c
#include <stdio.h>// entrées-sorties

int main(void){
  int i; //i déclaré mais de valeur non connue (déconseillé)
  float x = 15.3;//déclaration et initialisation en 1 ligne
  i = 10;// i a maintenant une valeur
  x = 3.6; // changer la valeur de x
  return 0;
}
```

- On déclare une variable en déclarant d'abord le type puis le nom. Point-virgule `;` à la fin de chaque instruction.
- On peut initialiser immédiatement (cf `x`) ou attendre un peu (cf `i`).

Compilation

Ouvrir un terminal :

- ♡ Compilation avec affichage des *warning* (mises en garde) (option `-Wall`) :

  ```
  gcc -Wall exemple.c
  ```

  Fichier exécutable produit : `a.out` (nom par défaut)
- ♡ Si on veut fixer le nom du fichier produit :

  ```
  gcc -Wall initialiser.c -o toto
  ```
- Exécution.

```
$ ./toto
```

- Récupérer la valeur renvoyée par le **main** :

```
$ echo $?
0
```

Cette commande récupère en fait le retour de la dernière commande passée dans le terminal.

Exercice

Exercice

Écrire un code qui soulève une erreur (d'exécution, pas de compilation) avant le `return 0` du main. Compiler, exécuter et récupérer la valeur renvoyée par le programme. Vaut-elle zéro ?

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.
- Chaque famille regroupe plusieurs types qui différent par leurs tailles et leurs aspects signés ou non signés.

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.
- Chaque famille regroupe plusieurs types qui différent par leurs tailles et leurs aspects signés ou non signés.
- La liste des tailles données ci-dessous est non exhaustive :

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.
- Chaque famille regroupe plusieurs types qui différent par leurs tailles et leurs aspects signés ou non signés.
- La liste des tailles données ci-dessous est non exhaustive :
  - Entiers. Plusieurs tailles différente. On se contente ici de `int` (signés) et `unsigned int` (non signés, au comportement cyclique). Au moins 2 octets (16 bits) très souvent sur 4.
    Pour de gros entiers, préferer `long` et `unsigned long`. Taille au moins 4 octets souvent sur 8.

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.
- Chaque famille regroupe plusieurs types qui différent par leurs tailles et leurs aspects signés ou non signés.
- La liste des tailles données ci-dessous est non exhaustive :
  - Entiers. Plusieurs tailles différente. On se contente ici de `int` (signés) et `unsigned int` (non signés, au comportement cyclique). Au moins 2 octets (16 bits) très souvent sur 4.
    Pour de gros entiers, préferer `long` et `unsigned long`. Taille au moins 4 octets souvent sur 8.
  - Caractères. `char`. C'est la plus petite unité adressable de la machine. Souvent sur un octet. La taille des autres types en est un multiple.

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.
- Chaque famille regroupe plusieurs types qui différent par leurs tailles et leurs aspects signés ou non signés.
- La liste des tailles données ci-dessous est non exhaustive :
  - Entiers. Plusieurs tailles différente. On se contente ici de `int` (signés) et `unsigned int` (non signés, au comportement cyclique). Au moins 2 octets (16 bits) très souvent sur 4.
    Pour de gros entiers, préferer `long` et `unsigned long`. Taille au moins 4 octets souvent sur 8.
  - Caractères. `char`. C'est la plus petite unité adressable de la machine. Souvent sur un octet. La taille des autres types en est un multiple.
  - Flottants : `float` (en général 4 octets) et `double` (en général 8 octet) et leurs versions non signées.

Types de base

- Il y a trois familles de types de base : caractères, entiers et flottants.
- Chaque famille regroupe plusieurs types qui différent par leurs tailles et leurs aspects signés ou non signés.
- La liste des tailles données ci-dessous est non exhaustive :
  - Entiers. Plusieurs tailles différente. On se contente ici de `int` (signés) et `unsigned int` (non signés, au comportement cyclique). Au moins 2 octets (16 bits) très souvent sur 4.
    Pour de gros entiers, préferer `long` et `unsigned long`. Taille au moins 4 octets souvent sur 8.
  - Caractères. `char`. C'est la plus petite unité adressable de la machine. Souvent sur un octet. La taille des autres types en est un multiple.
  - Flottants : `float` (en général 4 octets) et `double` (en général 8 octet) et leurs versions non signées.
- La taille des types dépend de la machine et de l'implémentation. Les bonnes infos sont dans les fichiers **limits.h** et **float.h**

Tableaux, chaînes de caractères

- Dans ce document, nous abordons brièvement la notion de tableau *statique*. Nous ne parlons pas des tableaux *dynamiques* (ou VLA) conformément à l'usage en MP2I.

Tableaux, chaînes de caractères

- Dans ce document, nous abordons brièvement la notion de tableau *statique*. Nous ne parlons pas des tableaux *dynamiques* (ou VLA) conformément à l'usage en MP2I.
- Il n'y a pas de type chaîne de caractères comme le `string` de `Python`. Les chaînes de caractères sont représentées

Tableaux, chaînes de caractères

- Dans ce document, nous abordons brièvement la notion de tableau *statique*. Nous ne parlons pas des tableaux *dynamiques* (ou VLA) conformément à l'usage en MP2I.
- Il n'y a pas de type chaîne de caractères comme le `string` de `Python`. Les chaînes de caractères sont représentées
  - soit par le type `char *` ou *constantes chaînes* : pointeur sur un caractère, en général le contenu est non modifiable.

Tableaux, chaînes de caractères

- Dans ce document, nous abordons brièvement la notion de tableau *statique*. Nous ne parlons pas des tableaux *dynamiques* (ou VLA) conformément à l'usage en MP2I.
- Il n'y a pas de type chaîne de caractères comme le `string` de `Python`. Les chaînes de caractères sont représentées
  - soit par le type `char *` ou *constantes chaînes* : pointeur sur un caractère, en général le contenu est non modifiable.
  - Soit par le type `char[]` ou *tableau de caractères*. Contenu modifiable.

Types structurés

Définis plus tard

Affichage

```c
// fichier hello.c
#include <stdio.h>// charger les entrées-sorties

int main(void){
   int i = 10;
   printf("i=%d\n", i);// affichage d'un entier %d
   printf("%f\n", 3.14);//afficher un flottant %f
   printf("coucou\n"); //afficher directement une chaîne
   char * salut = "hello";// une constante chaîne
   printf("%s\n",salut); // afficher une chaîne %s
   return 0; }
```

Exemple d'appel d'affichages avec les différents *spécifieurs de formats* : `%d,%f,%s` qui indiquent la nature de l'objet à afficher.

Affichage

- ♡ Ne pas oublier de charger la bibliothèques d'entrées-sorties `<stdio.h>` .

Affichage

- ♡ Ne pas oublier de charger la bibliothèques d'entrées-sorties `<stdio.h>`.
- `\n` pour passer à la ligne dans l'affichage.

Affichage

- ♡ Ne pas oublier de charger la bibliothèques d'entrées-sorties `<stdio.h>`.
- `\n` pour passer à la ligne dans l'affichage.
- Et toujours le `return 0` du **main**.

Compilation puis exécution

Dans un terminal, entrer

```
$ gcc -Wall hello.c -o hello # compilation
$ ./hello # exécution
i=10
3.140000
coucou
hello
```

- On exécute le programme produit en tapant `./hello`

Compilation puis exécution

Dans un terminal, entrer

```
$ gcc -Wall hello.c -o hello # compilation
$ ./hello # exécution
i=10
3.140000
coucou
hello
```

- On exécute le programme produit en tapant `./hello`
- Noter les commentaires en bash : `# un commentaire`

Tableau de spécificateurs de formats

Symbole	Type	Impression comme
%d ou **%i**	**int**	entier relatif
%u	**uint**	entier naturel non signé
%o	**int**	entier exprimé en octal
%x	**int**	entier exprimé en héxadécimal
%c	**char**	caractère
%s	**char *** et **char t[]**	chaîne de caractères
%f	**double**	flottants et doubles en notation décimale
%e	**double**	flottant en notation scientifique
%p	adresses	

A connaître `%d %f %s %p` ♡.

Saisie

```c
#include <stdio.h>// charger les entrées-sorties

int main(void){
  int i,j;// déclaration de deux entiers
  printf("saisir la valeur de i");
  scanf("%d",&i);
  printf("saisir la valeur de j");
  scanf("%d",&j);
  printf("vous avez saisi %d et %d\n",i,j);
  return 0; }
```

- `scanf` prend au minimum deux paramètres : un *spécifieur de format* (`%d,%f,%s...`) et une *adresse* (celle de la variable qu'on veut renseigner). ♡

Saisie

```c
#include <stdio.h>// charger les entrées-sorties

int main(void){
  int i,j;// déclaration de deux entiers
  printf("saisir la valeur de i");
  scanf("%d",&i);
  printf("saisir la valeur de j");
  scanf("%d",&j);
  printf("vous avez saisi %d et %d\n",i,j);
  return 0; }
```

- `scanf` prend au minimum deux paramètres : un *spécifieur de format* (`%d,%f,%s...`) et une *adresse* (celle de la variable qu'on veut renseigner). ♡
- La saisie de chaînes de caractères doit se faire avec prudence (voir plus tard)

Saisie de chaînes de caractères

```
1  int main(void){
2     char s[10];
3     printf("saisir votre nom\n");
4     scanf("%s",s);
5     printf("Bonjour %s\n",s);
6     return 0;
7  }
8
```

- Pour les chaînes de caractères, tout *séparateur* (comme un espace) interrompt la lecture. On peut préciser qu'on veut ou ne veut pas certains caractères.

5 opérateurs sans surprise♡

Les opérateurs arithmétiques ne devraient pas surprendre les utilisateurs de Python :

- l'addition `+`

5 opérateurs sans surprise♡

Les opérateurs arithmétiques ne devraient pas surprendre les utilisateurs de Python :

- l'addition `+`
- la soustraction `-`

5 opérateurs sans surprise♡

Les opérateurs arithmétiques ne devraient pas surprendre les utilisateurs de Python :

- l'addition `+`
- la soustraction `-`
- la multiplication `*`

5 opérateurs sans surprise♡

Les opérateurs arithmétiques ne devraient pas surprendre les utilisateurs de Python :

- l'addition `+`
- la soustraction `-`
- la multiplication `*`
- la division `/` (euclidienne ou flottante)

5 opérateurs sans surprise♡

Les opérateurs arithmétiques ne devraient pas surprendre les utilisateurs de Python :

- l'addition `+`
- la soustraction `-`
- la multiplication `*`
- la division `/` (euclidienne ou flottante)
- le modulo `%`.

5 opérateurs sans surprise♡

Les opérateurs arithmétiques ne devraient pas surprendre les utilisateurs de Python :

- l'addition `+`
- la soustraction `-`
- la multiplication `*`
- la division `/` (euclidienne ou flottante)
- le modulo `%`.
- Pas d'opérateur d'exponentiation.

Pas de type bouléen importé par défaut

- En **C**, il n'existe pas de type bouléen importé par défaut.

Pas de type bouléen importé par défaut

- En **C**, il n'existe pas de type bouléen importé par défaut.
- A la place, la valeur 0 représente le bouléen `false` et toute autre valeur, `true`.

Pas de type bouléen importé par défaut

- En **C**, il n'existe pas de type bouléen importé par défaut.
- A la place, la valeur 0 représente le bouléen `false` et toute autre valeur, `true`.
- Or, ce n'est pas l'esprit du programme d'info en CPGE. Une bonne façon de mettre des bouléens dans un programme **C** est alors d'importer la bibliothèque `#include <stdbool.h>` .♡

Opérateurs bouléens ♡

Les voici :

Symbole	Signification
&&	ET
||	OU
!	NON

```c
#include <stdio.h>// charger les entrées-sorties
#include <stdbool.h>// bouléens

int main(void){
   bool b = 1==1;
   bool c = !b;
   printf("b est %d, c est %d\n",b,c);
   return 0;}
```

Après compilation puis exécution :

```
b est 1, c est 0
```

Opérateurs bit-à-bit

Pour information

- Opérateurs de comparaison bit-à-bit `&`, `|`, $\wedge$ (Et bit-à-bit, Ou bit-à-bit, XOR bit-à-bit)

Opérateurs bit-à-bit

Pour information

- Opérateurs de comparaison bit-à-bit `&`, `|`, $\wedge$ (Et bit-à-bit, Ou bit-à-bit, XOR bit-à-bit)
- Et également des opérateurs de décalage `>>`, `<<` : division/multiplication par une puissance de 2.

Les comparateurs ♡

On donne le tableau suivant déjà connu des utilisateurs de **Python** :

Symbole	Signification
==	est égal à
$>$	est strictement supérieur à
$<$	est strictement inférieur à
$>=$	est supérieur ou égal à
$<=$	est inférieur ou égal à
$!=$	est différent de

Comme en **Python**, le symbole `=` n'est pas un opérateur de comparaison mais d'affectation.

Expression conditionnelle

```
if (condition réalisée) {
    liste instructions
}

else {
      autre série instructions
}
```

- La condition est indiquée entre parenthèse

Expression conditionnelle

```
if (condition réalisée) {
    liste instructions
}

else {
      autre série instructions
}
```

- La condition est indiquée entre parenthèse
- Les blocs sont écrits entre accolades.

Expression conditionnelle

```
if (condition réalisée) {
    liste instructions
}

else {
      autre série instructions
}
```

- La condition est indiquée entre parenthèse
- Les blocs sont écrits entre accolades.
- L'indentation ne fait pas sens en C contrairement à `Python`

Si alors sinon ♡

Détection et affichage de la parité d'un entier saisi

```
// parité d'un entier
int n;
printf("entrer un nombre entier : ");
scanf("%i",&n);
if (n%2==0)
  {
    printf("votre nombre est pair\n");
  }
else
  {
    printf("votre nombre est impair\n");
  }
```

Sinon si ♡

`else if`

Le `elif` de **Python** s'écrit `else if` en **C** et se place entre `if` et `else` :

```c
//Indication du nombre de racines d'un trinôme du 2nd degré

double delta;
printf("entrer la valeur du discriminant : ");
scanf("%lf",&delta);
if (delta==0)
{
printf("une racine double\n");}
else if (delta > 0)
{
   printf("deux racines réelles\n");}
else
{
printf("pas de racine réelle\n");}
```

Boucle `while` ♡

- Syntaxe

```
1       while ( Condition ){
2        // Instructions à répéter
3   }
4
```

Boucle `while` ♡

- Syntaxe

```
1       while ( Condition ){
2        // Instructions à répéter
3   }
4
```

- Voici comment afficher les carrés des chiffres de 0 à 9 :

```
1    int cpt = 0;//compteur
2    while (cpt<10){
3       printf("%d\n",cpt*cpt);
4       cpt = cpt+1;
5    }
6
```

Boucle `while` ♡

Attention aux boucles infinies !

```
1  int cpt = 0;//compteur
2  while (cpt<10){
3     printf("%d\n",cpt*cpt);
4  }
```

Pour arrêter : CTRL + C

Exemple boucle **while** ♡

- Voici une boucle qui demande d'entrer un nombre jusqu'à ce que l'utilisateur saisisse 10 :

```
1     int nb =0 ;
2     while (nb != 10)//condition entre parenthèses
3       {
4          printf("entrer un nb entier :");
5          scanf ("%d", &nb);
6       }
```

Exemple boucle **while** ♡

- Voici une boucle qui demande d'entrer un nombre jusqu'à ce que l'utilisateur saisisse 10 :

```
1     int nb =0 ;
2     while (nb != 10)//condition entre parenthèses
3       {
4          printf("entrer un nb entier :");
5          scanf ("%d", &nb);
6       }
```

- Le programme entre au moins une fois dans la boucle. Voici une trace d'exécution :

```
entrer un nb entier :12
entrer un nb entier :10
```

Programmation modulaire

- Un gros programme peut et doit être découpé en plusieurs *modules* (sous-parties du programme).

Programmation modulaire

- Un gros programme peut et doit être découpé en plusieurs *modules* (sous-parties du programme).
- Cela permet :

Programmation modulaire

- Un gros programme peut et doit être découpé en plusieurs *modules* (sous-parties du programme).
- Cela permet :
  - d'améliorer la lisibilité (en gros : une idée = 1 module)

Programmation modulaire

- Un gros programme peut et doit être découpé en plusieurs *modules* (sous-parties du programme).
- Cela permet :
  - d'améliorer la lisibilité (en gros : une idée = 1 module)
  - d'éviter les séquences d'instructions répétitives, et cela d'autant plus facilement que la notion d'arguments permet de paramétrer certains modules.

Programmation modulaire

- Un gros programme peut et doit être découpé en plusieurs *modules* (sous-parties du programme).
- Cela permet :
  - d'améliorer la lisibilité (en gros : une idée = 1 module)
  - d'éviter les séquences d'instructions répétitives, et cela d'autant plus facilement que la notion d'arguments permet de paramétrer certains modules.
  - le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois. La *compilation séparée* est ici bien pratique pour ne compiler qu'une partie du programme indépendamment des autres.

Programmation modulaire en C

Fonctions et procédures

- Il n'existe qu'un seul type de module en C : les fonctions et procédures

- Il n'existe qu'un seul type de module en C : les fonctions et procédures
- Les arguments sont transmis *par valeur* en C aux fonctions.
  Il semble donc impossible de modifier un argument transmis mais en réalité certaines valeurs (pointeurs et tableaux) sont en fait des adresses : cela fournit un moyen détourné de modifier l'agument transmis.

Programmation modulaire en C

Fonctions et procédures

- Il n'existe qu'un seul type de module en C : les fonctions et procédures
- Les arguments sont transmis *par valeur* en C aux fonctions.
  Il semble donc impossible de modifier un argument transmis mais en réalité certaines valeurs (pointeurs et tableaux) sont en fait des adresses : cela fournit un moyen détourné de modifier l'agument transmis.
- La compilation séparée se fait par fichier source (alors qu'en Fortran par exemple, elle se fait par module).

Déclaration de fonction♡

```
1  type_de_retour nomFonction(parametres){
2    /* corps de la fonction :
3       Insérez vos instructions ici */
4     }
5
```

Déclaration de fonction♡

```
1  type_de_retour nomFonction(parametres){
2    /* corps de la fonction :
3       Insérez vos instructions ici */
4     }
5
```

- avant le nom de la fonction, on indique son type de retour (comme `int`, `bool`, `float`, `char *`...)

Déclaration de fonction♡

```
1  type_de_retour nomFonction(parametres){
2    /* corps de la fonction :
3       Insérez vos instructions ici */
4     }
5
```

- avant le nom de la fonction, on indique son type de retour (comme `int`, `bool`, `float`, `char *`...)
- Le nom de chaque paramètre est précédé de son type.

Déclaration de fonction♡

```
1  type_de_retour nomFonction(parametres){
2    /* corps de la fonction :
3       Insérez vos instructions ici */
4     }
5
```

- avant le nom de la fonction, on indique son type de retour (comme `int`, `bool`, `float`, `char *`...)
- Le nom de chaque paramètre est précédé de son type.
- Le *corps* de la fonction est entre accolades.

Déclaration de fonction♡

```
1  type_de_retour nomFonction(parametres){
2    /* corps de la fonction :
3       Insérez vos instructions ici */
4     }
5
```

- avant le nom de la fonction, on indique son type de retour (comme `int`, `bool`, `float`, `char *`...)
- Le nom de chaque paramètre est précédé de son type.
- Le *corps* de la fonction est entre accolades.
- La partie avant l'accolade est appelée *prototype de la fonction*

Déclaration de fonction♡

```
1  int triple(int nombre){// ex avec un paramètre
2      return 3 * nombre;}
3
4  void salut(){
5      printf("coucou\n");
6   }
7
8  int mult(int i, int j){// ex avec 2 paramètres
9      return i*j;
10     }
11
```

Déclaration de fonction♡

```
1  int triple(int nombre){// ex avec un paramètre
2      return 3 * nombre;}
3
4  void salut(){
5     printf("coucou\n");
6   }
7
8  int mult(int i, int j){// ex avec 2 paramètres
9     return i*j;
10    }
11
```

- le mot `return` est obligatoire si la valeur de retour n'est pas `void`. La valeur retournée doit être conforme au type annoncé.

Déclaration de fonction♡

```
1  int triple(int nombre){// ex avec un paramètre
2      return 3 * nombre;}
3
4  void salut(){
5      printf("coucou\n");
6   }
7
8  int mult(int i, int j){// ex avec 2 paramètres
9      return i*j;
10     }
11
```

- le mot `return` est obligatoire si la valeur de retour n'est pas `void`. La valeur retournée doit être conforme au type annoncé.
- Si la fonction ne retourne rien, on dit que c'est une *procédure*. Dans ce cas, le type de retour est `void`.

Appel de fonction

```
1  // fichier appel.c
2  int main(void){
3     int i = 3;
4     // 2 spécifieurs %d :
5     printf("triple(%d) = %d\n",i,triple(i));
6     salut();
7     return 0; }
8
```

Appel de fonction

```
1  // fichier appel.c
2  int main(void){
3     int i = 3;
4     // 2 spécifieurs %d :
5     printf("triple(%d) = %d\n",i,triple(i));
6     salut();
7     return 0; }
8
```

- Le prototype de la fonction doit être placé *avant* son premier appel. Mais il est possible d'écrire le corps de la fonction *après* un ou plusieurs appels (cf plus tard).
  Les prototypes de `triple` et `salut` sont placés avant la fonction `main` qui peut ainsi appeler ces fonctions.

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé.

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé.
- On l'utilise pour déclarer qu'une fonction ne renvoie pas de valeur : `void f(int x)`

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé.
- On l'utilise pour déclarer qu'une fonction ne renvoie pas de valeur : `void f(int x)`
- Pour les fonctions sans argument, préférer `int f(void)`, (qui signifie explicitement que `f` ne prend pas d'argument) à `int f()` (qui indique que le nombre d'arguments de `f` n'est pas connu).

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé.
- On l'utilise pour déclarer qu'une fonction ne renvoie pas de valeur : `void f(int x)`
- Pour les fonctions sans argument, préférer `int f(void)`, (qui signifie explicitement que `f` ne prend pas d'argument) à `int f()` (qui indique que le nombre d'arguments de `f` n'est pas connu).
- Enfin, si `void` n'est pas un type, `void *` désigne un pointeur vers une valeur dont on ne connaît pas le type.
  Les règles de typage de C permettent d'utiliser une valeur de type `void *` là où une valeur d'un certain type de pointeur est attendu : `int *x = malloc(sizeof(int))` (malloc renvoie un `void*`).

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé.
- On l'utilise pour déclarer qu'une fonction ne renvoie pas de valeur : `void f(int x)`
- Pour les fonctions sans argument, préférer `int f(void)`, (qui signifie explicitement que `f` ne prend pas d'argument) à `int f()` (qui indique que le nombre d'arguments de `f` n'est pas connu).
- Enfin, si `void` n'est pas un type, `void *` désigne un pointeur vers une valeur dont on ne connaît pas le type.
  Les règles de typage de C permettent d'utiliser une valeur de type `void *` là où une valeur d'un certain type de pointeur est attendu : `int *x = malloc(sizeof(int))` (malloc renvoie un `void*`).
- **(PI)** Enfin `void *` permet le *polymorphisme*. Exemple : une fonction qui agit sur un tableau dont les éléments sont de type quelconque. Nous n'utilisons pas cette fonctionnalité.

Tableaux statiques à une dimension

- La taille des *tableaux statiques* est connue à la compilation.

Tableaux statiques à une dimension

- La taille des *tableaux statiques* est connue à la compilation.
- Déclaration sous la forme `type nom [taille]` dans lequel `taille` est un nombre (pas une variable, sinon c'est un *tableau à longueur variable* appelé *Variable Length Array (VLA)*

```
1 int t[3]; // déclaration d'un tableau de 3 entiers.
2
```

Tableaux statiques à une dimension

- La taille des *tableaux statiques* est connue à la compilation.
- Déclaration sous la forme `type nom [taille]` dans lequel `taille` est un nombre (pas une variable, sinon c'est un *tableau à longueur variable* appelé *Variable Length Array (VLA)*

```
1 int t[3]; // déclaration d'un tableau de 3 entiers.
2
```

- Pas de VLA en MP2I/MPI!!

```
gcc -Wall -Werror=vla myfile.c # interdire VLA
```

Tableaux statiques à une dimension

- La taille des *tableaux statiques* est connue à la compilation.
- Déclaration sous la forme `type nom [taille]` dans lequel `taille` est un nombre (pas une variable, sinon c'est un *tableau à longueur variable* appelé *Variable Length Array (VLA)*

```
1 int t[3]; // déclaration d'un tableau de 3 entiers.
2
```

- Pas de VLA en MP2I/MPI !!

```
gcc -Wall -Werror=vla myfile.c # interdire VLA
```

- Les cases d'un tableau de taille N sont indicées de 0 à $N-1$. Accès à la i-ème case : `t[i]`.

Tableaux statiques à une dimension

- La taille des *tableaux statiques* est connue à la compilation.
- Déclaration sous la forme `type nom [taille]` dans lequel `taille` est un nombre (pas une variable, sinon c'est un *tableau à longueur variable* appelé *Variable Length Array (VLA)*

```
1 int t[3]; // déclaration d'un tableau de 3 entiers.
2
```

- Pas de VLA en MP2I/MPI!!

```
gcc -Wall -Werror=vla myfile.c # interdire VLA
```

- Les cases d'un tableau de taille N sont indicées de 0 à $N-1$. Accès à la i-ème case : `t[i]`.
- Contrairement à `Python` il n'y a pas de fonction `len` qui donne la longueur du tableau. C'est au programmeur de retenir la taille de son tableau (voir chapitre sur les Bytes).

Ecrire dans un tableau

```
1  int t [2];// t1 a deux éléments
2  t[0] = 6; t[1] = 2;// écrire dans t1
3
4  int t3[] = {1,2,3}// correct : t3 prend une taille de 3
5
6  // La dernière case t2[2] est initialisée à 0 :
7  int t2[3] = {2,6};
8  t2 = {8,9,10} // soulève une erreur de compilation !
9
```

Un tableau n'est pas une **lvalue** ! ! ♡

Remplir puis parcourir un tableau ♡

```c
int main(void){
  int i = 0;// compteur
  int t[5];// tableau de 5 entiers
  while (i<5){// remplir
    t[i] = 2*i;// mettre 2i dans la case i
    i=i+1;
  }
  i=0;//remise à zéro du compteur
  while (i<5){// afficher
    printf("%d,",t[i]);  i=i+1;
  }
  printf("\n");
  return 0; }

```

Remplir puis parcourir un tableau

Après compilation et exécution :

```
$ gcc -Wall tab.c
$ ./a.out
0,2,4,6,8,
```

Initialisation de tableau

- Avec `int t[5]` on déclare un tableau de 5 entiers. Ce qui est dedans est impossible à prévoir.

Initialisation de tableau

- Avec `int t[5]` on déclare un tableau de 5 entiers. Ce qui est dedans est impossible à prévoir.
- Pour déclarer le contenu d'un tableau à sa création, on peut utiliser une syntaxe énumérative :

```
1  int t[5] = { 1 , 5 , 45 , 3 , 9 };
2
```

Initialisation de tableau

- Avec `int t[5]` on déclare un tableau de 5 entiers. Ce qui est dedans est impossible à prévoir.
- Pour déclarer le contenu d'un tableau à sa création, on peut utiliser une syntaxe énumérative :

```
1  int t[5] = { 1 , 5 , 45 , 3 , 9 };
2
```

- Si on écrit `int t[5]=`$\{10, 20\}$, les éléments 2,3,4 sont mis par défaut à 0.

Danger

```
1    int tableau[1]; // déclare un tableau d'un entier */
2    tableau[10] = 5; //l'élément 10 n'existe pas
3    printf("%d\n", tableau[10]);
4
```

Danger

```
1    int tableau[1]; // déclare un tableau d'un entier */
2    tableau[10] = 5; //l'élément 10 n'existe pas
3    printf("%d\n", tableau[10]);
4
```

- Ce programme peut parfois compiler (le compilateur peut ne pas détecter le dépassement de capacité) et même s'exécuter.

Danger

```
1    int tableau[1]; // déclare un tableau d'un entier */
2    tableau[10] = 5; //l'élément 10 n'existe pas
3    printf("%d\n", tableau[10]);
4
```

- Ce programme peut parfois compiler (le compilateur peut ne pas détecter le dépassement de capacité) et même s'exécuter.
- Ce que fait alors le processus n'est pas prévu par la norme (comportement indéfini). Il peut afficher 5, s'arrêter en signalant une erreur, ou, plus grave, corrompre une autre partie de la mémoire du processus, rendant très difficile à prévoir son comportement.

Danger

```c
int tableau[1]; // déclare un tableau d'un entier */
tableau[10] = 5; //l'élément 10 n'existe pas
printf("%d\n", tableau[10]);

```

- Ce programme peut parfois compiler (le compilateur peut ne pas détecter le dépassement de capacité) et même s'exécuter.
- Ce que fait alors le processus n'est pas prévu par la norme (comportement indéfini). Il peut afficher 5, s'arrêter en signalant une erreur, ou, plus grave, corrompre une autre partie de la mémoire du processus, rendant très difficile à prévoir son comportement.
- Faire très attention aux dépassements de capacité !

Passer un tableau en paramètre

En C, lorsqu'on passe un tableau en argument d'une fonction, il est prudent de fournir sa taille. Ce n'est pas obligatoire mais cela permet d'éviter d'écrire où il ne faut pas.

```c
void affiche(int t[], int nb_elmts){
  //afficher contenu de t
  int i=0;
  while (i<nb_elmts){
    printf("t[%d]=%d  \n",i,t[i]);
    i=i+1;
  }
  printf("\n");
}

void modifier(int t[], int p, int x){
  //modifier contenu de t en position p
  t[p] = x;
}
```

Exemple d'appel

```
1  int main(void){
2    int t[5]={10,21,-23};
3    affiche(t,3);
4
5    printf("modifions t[1] en 100\n");
6    modifier(t,1,100);
7    affiche(t,3);
8
9    return 0;
10   }
11
```

Exemple d'exécution

Après compilation/exécution

```
t[0]=10 t[1]=21 t[2]=-23
modifions t[1] en 100
t[0]=10 t[1]=100 t[2]=-23
```

Assertions

- Pour diverses raisons on peut souhaiter interrompre un programme si une condition n'est pas réalisée (ex : impossibilité d'allouer de la mémoire). Les *assertions* sont alors utiles.

Assertions

- Pour diverses raisons on peut souhaiter interrompre un programme si une condition n'est pas réalisée (ex : impossibilité d'allouer de la mémoire). Les *assertions* sont alors utiles.
- Exemple d'assertion :

```
1 #include <assert.h>// Entête pour utiliser une assertion
2
3 int main(void) {
4     int i=0;// on veut imposer i>1
5     assert( i > 1 );// si i<=1 : arrêt du pgm
6     printf( "Le programme peut se poursuivre normalement"
      );
7     return 0;
8 }
9
```

Assertions

- Trace d'exécution :

```
a.out: assertion2.c:8: main: Assertion ‘i > 1’ failed.
Abandon (core dumped}
```