Plan

Fonctions en paramètre de fonctions

- On peut passer une fonction `g` en paramètre d'une fonction `f`. On passe un pointeur sur la fonction :

```
f(...,&g,...)
```

- Le type d'un paramètre fonction est un pointeur `pf` dont le type est déclaré par la syntaxe :

```
typeRes (*pf) (type1, type2, ..., typen)
```

- Dans le corps de la fonction ayant ce pointeur comme paramètre, on peut faire un appel en déréférencant le pointeur :

```
*pf(exp1,exp2,...,expn)
```

Fonctions en paramètre de fonctions

```c
#include <stdio.h>

int double(int a) {
    return 2*a;
}

int carre(int a) {
    return a * a;
}

static void affiche(int a, int (*pf)(int)) {
    printf("%d.\n", (*pf)(a));
}

int main(void) {
    affiche(5, &double);
    affiche(5, &carre);
    return 0;
}
```

Fonctions d'arité variable

On peut faire des fonctions à nombre et types variables d'arguments (ex. `printf`). Le dernier paramètre de leur signature est `...`

Définition

```
typeRes nomFonction (type1 nom1, type2 nom2, ...)
```

- Au moins un paramètre identifié doit permettre de déterminer le nombre et le type des autres.
- Les fonctions `va_start()`, `va_arg()` et `va_end()` de la librairie `stdarg.h` permettent d'accéder aux paramètres effectifs dans le corps de la fonction.

Exemple de fonction d'arité variable

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

void affiche_suite(int nb, ...) {
    va_list ap;
    va_start(ap, nb);
    while (nb > 0) {
        int n = va_arg(ap, int);
        printf("%d.\n", n);
        nb--;
    }
    va_end(ap);
}

int main(void) {
    affiche_suite(10,10,20,30,40,50,60,70,80,90,100);
    return 0;
}
```

Les pré-conditions

Pour contrôler le bon usage des fonctions, on peut leur adjoindre des **pré-conditions** :

- L'idée est de permettre de vérifier si les paramètres effectifs respectent bien les conditions normales d'utilisation de la fonction.

Exemple

Soit la spécification de fonction suivante :

Algo : divise
Données : a et b des entiers naturels ; b est différent de 0
Résultat : vrai si b divise a, faux sinon

Sa signature C ne permet pas de traduire son domaine de définition.

```
_Bool divise(int a, int b) {
    return a % b == 0;
}
```

La directive assert

La macrofonction `assert` de la librairie `assert.h` permet de compléter la définition de la fonction de tests **d'expressions booléennes** qui s'ils s'avèrent faux provoquent l'arrêt de la fonction avec un message d'erreur et la génération d'une image mémoire.

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

_Bool divise(int a, int b) {
    assert (a>=0 && b>=0);
    assert (a>=0 && b>0);
    return a % b == 0;
}

int main() {
    int n,d;
    printf("Saisir deux nombres : ");
    scanf("%d %d",&n,&d);
    if(divise(n,d)) printf("%d divise %d\n",d,n);
    else printf("%d ne divise pas %d\n",d,n);
    return 0;
}
```

Exemples d'exécution

Exécutions :

```
> ./divisibilite
Saisir deux nombres : 6 2
2 divise 6

> ./divisibilite
Saisir deux nombres : 2 0
Assertion failed: (b != 0), function divise, file divisibilite.c, line 6.
Abort trap: 6

> ./divisibilite
Saisir deux nombres : -3 4
Assertion failed: (a>=0 && b>=0), function divise, file divisibilite.c, line 5.
Abort trap: 6
```

Ces tests utiles en phase de mise au point peuvent ensuite être supprimés en ajoutant l'option `-DNDEBUG` lors de la compilation.

```
>  gcc -Wall -DNDEBUG divisibilite.c -o divisibilite
> ./divisibilite
Saisir deux nombres : -3 4
4 ne divise pas -3
```