

Information, Calcul, Communication (partie programmation) : Fonctions (1)

Jean-Cédric Chappelier

Faculté I&C

Objectifs du cours d'aujourd'hui

- ▶ Premiers rappels sur les fonctions en C++ :
 - ▶ notion de fonction : prototype, appel, définition
 - ▶ passages d'arguments : par valeur / par référence
- ▶ Etude de cas / Questions

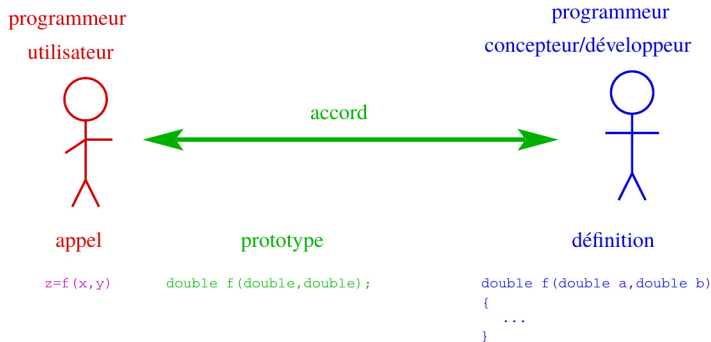
NOTE : vous avez DEUX semaines pour bien travailler les fonctions (décalage avec le MOOC)

Rappel du calendrier

	MOOC	décalage / MOOC	exercices prog. 1h45 Jeudi 8-10	cours prog. 45 min. Jeudi 10-11
1 10.09.26	--	-1	prise en main	Bienvenue/Introduction
2 17.09.26	1. variables	0	variables / expressions	variables / expressions
3 24.09.26	2. if	0	if – switch	if – switch
4 01.10.26	3. for/while	0	for / while	for / while
5 08.10.26	4. fonctions	0	fonctions (1)	fonctions (1)
6 15.10.26		1	fonctions (2)	fonctions (2)
- 22.10.26				
7 29.10.26	5. tableaux (vector)	1	vector	vector
8 05.11.26	6. string + struct	1	array / string	array / string
9 12.11.26		2	structures	structures
10 19.11.26	7. pointeurs	2	pointeurs	pointeurs
11 26.11.26		-	entrées/sorties	entrées/sorties
12 03.12.26		-	erreurs / exceptions	erreurs / exceptions
13 10.12.26		-	révisions	théorie : sécurité
14 17.12.26	8. étude de cas	-	révisions	Révisions

Les « 3 facettes » d'une fonction

- Résumé / Contrat (« **prototype** »)
- Création / Construction (« **définition** »)
- Utilisation (« **appel** »)



Exemple

```
#include <iostream>
using namespace std;
```

prototype

```
double moyenne(double nombre_1, double nombre_2);
```

```
int main()
{
    double note1(0.0), note2(0.0);
    cout << "Entrez vos deux notes : " << endl;
    cin >> note1 >> note2;
    cout << "Votre moyenne est : "
        << moyenne(note1, note2) << endl;
    return 0;
}
```

appel

```
double moyenne(double x, double y)
{
    return (x + y) / 2.0;
}
```

définition



Prototypage : remarque



Dans les prototypes des fonctions, *les identificateurs des paramètres sont optionnels*. En fait, ils ne servent qu'à rendre le prototype plus lisible.

Dans l'exemple précédent, la fonction `moyenne` peut donc également être prototypée par :

```
double moyenne(double, double);
```

Conseil : Écrivez cependant les noms des paramètres dans le prototypage des fonctions et **choisissez des noms pertinents**.

Cela augmente la lisibilité de votre code (et donc facilite sa maintenance).

Évaluation d'un appel de fonction

Pour une fonction définie par

$$\text{typeR } f(\text{type1 } x1, \text{type2 } x2, \dots, \text{typeN } xN) \{ \dots \}$$

l'**évaluation** de l'appel

$$f(\text{arg1}, \text{arg2}, \dots, \text{argN})$$

s'effectue de la façon suivante :

1. les **expressions** $\text{arg1}, \text{arg2}, \dots, \text{argN}$ sont évaluées (dans un ordre quelconque !)
2. les valeurs correspondantes sont **affectées** aux paramètres $x1, x2, \dots, xN$ de la fonction f
(variables locales au corps de f)

Concrètement, ces deux premières étapes reviennent à faire :

$$x1 = \text{arg1}, x2 = \text{arg2}, \dots, xN = \text{argN}$$

3. le programme correspondant au corps de la fonction f est exécuté
4. l'expression suivant la première commande **return** est évaluée et retournée comme résultat de de l'appel.
5. cette valeur remplace l'expression de l'appel, c'est-à-dire l'expression $f(\text{arg1}, \text{arg2}, \dots, \text{argN})$

Évaluation d'un appel de fonction (2)

Les étapes ① et ② n'ont bien sûr pas lieu pour une fonction sans argument.

Les étapes ④ et ⑤ n'ont bien sûr pas lieu pour une fonction sans valeur de retour (`void`).

L'étape ② n'a pas lieu lors d'un passage par référence (voir plus loin).



Portée / Appel



```
int x, z;  
  
int main () {  
    int x, y;  
    ...  
    { int y;  
      ... x ...  
      ... y ...  
      ... z ...  
    } ...  
    ... y ...  
}
```

```
int f(int x);  
int z;  
int main () {  
    int x, y;  
    ...  
    ... f(y) ...  
}
```

```
int f(int x) {  
    int y;  
    ...  
    ... x ...  
    ... y ...  
    ... z ...  
}
```

Le passage des arguments

On distingue en général 2 types de passages d'arguments :

passage par valeur :

La variable locale associée à un argument passé par valeur correspond à une **copie** de l'argument (c'est-à-dire un objet distinct mais de même valeur littérale).

*Les modifications effectuées à l'intérieur de la fonction **ne sont donc pas répercutées** à l'extérieur de la fonction.*

passage par référence :

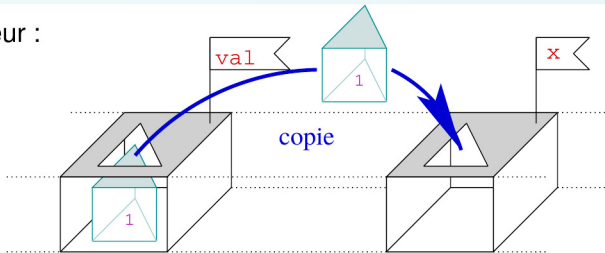
La variable locale associée à un argument passé par référence correspond à une **référence** sur l'objet associé à l'argument lors de l'appel.

Une modification qui est effectuée à l'intérieur de la fonction peut alors se répercuter à l'extérieur de la fonction.

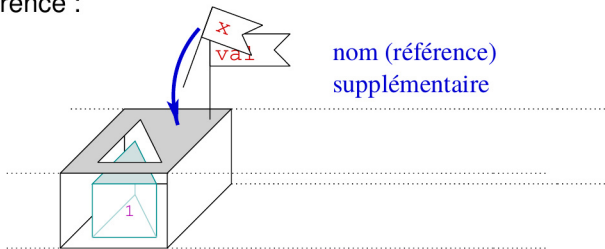
Le passage par référence peut être explicitement sélectionné en définissant le type des paramètres de la fonction comme étant des références (identifiées par le symbole **&**, par exemple `double& x`).

Passages d'argument : schéma

Passage par valeur :



Passage par référence :





Optimisation (1)



On souhaite parfois **éviter la copie locale** faite par un passage par valeur.

On utilise alors pour cela un **passage par référence**.

Mais comme il s'agit d'une optimisation et non pas d'un vrai passage par référence, on n'autorisera pas la fonction à modifier ses arguments en **protégeant la référence** par le mot `const`.

Exemple :

```
double moyenne (const double& x, double const& y) {  
    return (x+y) / 2.0 ;  
}
```

Conseil : **Pour les types avancés** (à venir), utilisez toujours `const` dans vos passages d'arguments sauf si vous voulez **vraiment** modifier la variable passée (par référence).

Note : On **ne** passe **pas** les types de base (`int`, `double`, `char`, `bool`) par référence constante

(mais vous ne connaissez pour le moment aucun autre type)



C++11

Optimisation (2) :

données temporaires

 $f(\underline{3})$

Dans le cours sur les *variables*, nous avons souligné l'existence de données temporaires, non nommées.

C++11 permet une meilleure utilisation de ces données temporaires et introduit la notion de **déplacement**.

Dans le cas d'un **passage par valeur**, le compilateur peut éviter la copie de données temporaire et simplement les *déplacer* (= gestion intelligente du « nom », sans copie physique de la valeur).

Pour le **passage par référence**, on peut introduire explicitement le passage de références vers des données temporaires (« *rvalue reference* ») avec le signe **&&** :

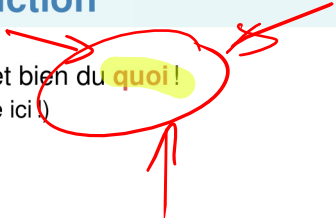
```
typeR f(type1&& nom);
```

Mais cela est très spécifique et sort du cadre d'un cours d'introduction.
Nous n'en reparlerons qu'un peu, au niveau avancé, lors de la surcharge des opérateurs au second semestre.

Méthodologie pour construire une fonction

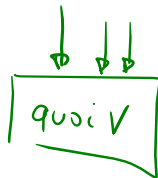
- ① clairement identifier ce que **doit faire** la fonction

☞ ne pas se préoccuper ici du *comment*, mais bel et bien du **quoi!**
(ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)



Méthodologie pour construire une fonction

- ① clairement identifier ce que **doit faire** la fonction
 - ☞ ne pas se préoccuper ici du *comment*, mais bel et bien du **quoi** !
(ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)
- ② que doit recevoir la fonction pour faire cela ?
 - ☞ identifie les **arguments** de la fonction

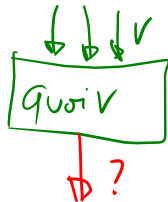


Méthodologie pour construire une fonction

- ① clairement identifier ce que **doit faire** la fonction
 - ☞ ne pas se préoccuper ici du **comment**, mais bel et bien du **quoi** !
(ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)
- ② que doit recevoir la fonction pour faire cela ?
 - ☞ identifie les **arguments** de la fonction
- ③ pour chaque argument : doit-il être modifié par la fonction ?
(si oui ☞ passage par référence)
Optionnel : se demander si cela a un sens de donner une valeur par défaut au paramètre correspondant
- ④ que doit « retourner » la fonction ☞ type de retour
Se poser ici la question (pour une fonction nommée **f**) :
est-ce que cela a un sens d'écrire :

`z = f(...);`

Si oui ☞ le type de **z** est le type de retour de **f**
Si non ☞ le type de retour de **f** est **void**



Méthodologie pour construire une fonction

- ① clairement identifier ce que **doit faire** la fonction
 - 👉 ne pas se préoccuper ici du **comment**, mais bel et bien du **quoi** !
(ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)
- ② que doit recevoir la fonction pour faire cela ?
 - 👉 identifie les **arguments** de la fonction
- ③ pour chaque argument : doit-il être modifié par la fonction ?
(si oui 👉 passage par référence)
Optionnel : se demander si cela a un sens de donner une valeur par défaut au paramètre correspondant
- ④ que doit « retourner » la fonction 👉 type de retour
Se poser ici la question (pour une fonction nommée **f**) :
est-ce que cela a un sens d'écrire :
$$z = f(\dots);$$

Si oui 👉 le type de **z** est le type de retour de **f**
Si non 👉 le type de retour de **f** est **void**
- ⑤ (maintenant, et seulement maintenant) Se préoccuper du **comment** :
c'est-à-dire comment faire ce que doit faire la fonction ?
c'est-à-dire écrire le corps de la fonction

Etude de cas

extrait d'un ancien devoir du MOOC : « *Sommes et produits* »

On cherche les 20 premiers nombres entiers plus grands que 10 dont la somme des chiffres est égale au produit de ces mêmes chiffres.

Par exemple 123 est un tel nombre : $1 + 2 + 3 = 1 \times 2 \times 3$.

Voici les 5 premiers nombres que vous devriez trouver : 22, 123, 132, 213 et 231.

Notes :

- ▶ $n \% 10$ donne le chiffre le plus à droite de n .
Par exemple, $178 \% 10$ donne 8.
- ▶ La division *entière* de n par 10, supprime ce chiffre.
Par exemple, $178 / 10$ donne 17.

Somme (123)

n	resultat	
	0	
	3	+3
12	5	+2
1	6	+1
0		

Approche « montante » / « par petits bouts »

① Écrire une fonction qui retourne la **somme** des chiffres d'un nombre passé en paramètre.

Par exemple, retourner 7 quand 223 est passé en paramètre ($2 + 2 + 3 = 7$).

② **TESTER** sa fonction !!

③ Écrire ensuite une fonction qui retourne le **produit** des chiffres d'un nombre passé en paramètre.

Par exemple, cette fonction devra retourner 24 quand 423 est passé en paramètre ($4 \times 2 \times 3 = 24$).

④ Utiliser ensuite ces deux fonctions pour écrire une fonction qui **teste** si la somme des chiffres d'un nombre entier passé en paramètre est égale au produit de ces mêmes chiffres.

Par exemple, `somme_produit_egaux(12)` devra renvoyer `false`,
`somme_produit_egaux(123)` devra renvoyer `true`.

⑤ Utiliser enfin cette fonction pour afficher les 20 nombres recherchés.

Approche « descendante »

Supposer que toutes les fonctions dont on a besoin existent.
Mais bien les spécifier (prototype) au fur et à mesure de nos besoins.

- ① Écrire la boucle de recherche des nombres en question
 - ☞ suppose l'existence de `bool somme_produit_egaux(int);`
- ② Écrire `somme_produit_egaux`
 - ☞ suppose l'existence de `int somme_chiffres(int);`
et `int produit_chiffres(int);`
- ③ Écrire `somme_chiffres`
- ④ Écrire `produit_chiffres`
- ⑤ Se rendre compte d'une partie commune : **JAMAIS de copié-collé**
⇒ faire une fonction supplémentaire pour la partie commune
- ⑥ Penser à écrire des **tests** !