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

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
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

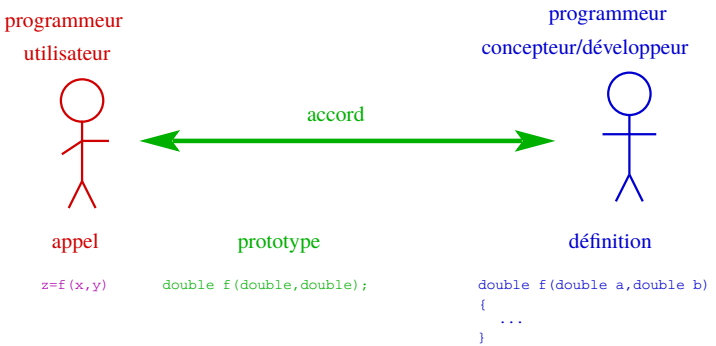
NOTE : vous avez DEUX semaines pour bien travailler les fonctions (décalage avec le MOOC)

Rappel du calendrier

		MOOC	décalage / MOOC	exercices prog. 1h45 Jeudi 9-11	cours prog. 45 min. Jeudi 11-12
1	23.09.21	--	-1	prise en main	Bienvenue/Introduction
2	30.09.21	1. variables	0	variables / expressions	variables / expressions
3	07.10.21	2. if	0	if – switch	if – switch
4	14.10.21	3. for/while	0	for / while	for / while
5	21.10.21	4. fonctions	0	fonctions (1)	fonctions (1)
6	28.10.21		1	fonctions (2)	fonctions (2)
7	04.11.21	5. tableaux (vector)	1	vector	vector
8	11.11.21	6. string + struct	1	array / string	array / string
9	18.11.21		2	structures	structures
10	25.11.21	7. pointeurs	2	pointeurs	pointeurs
11	02.12.21		-	entrées/sorties	entrées/sorties
12	09.12.21		-	erreurs / exceptions	erreurs / exceptions
13	16.12.21		-	révisions	théorie : sécurité
14	23.12.21	8. étude de cas	-	**Examen final (2h45)**	
					(ne sont pas sur le MOOC)

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

Jean-Cédric Chappelier
& Jamila Sam

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

typeR f(type1 x1, type2 x2, ..., typeN xN) { ... }

l'**évaluation** de l'appel

f(arg1, arg2, ..., argN)

s'effectue de la façon suivante :

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

   Concrètement, ces deux premières étapes reviennent à faire :
   `x1 = arg1`, `x2 = arg2`, ..., `xN = 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(arg1, arg2, ..., argN)`

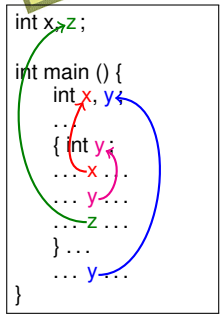
Évaluation d'un appel de fonction (2)

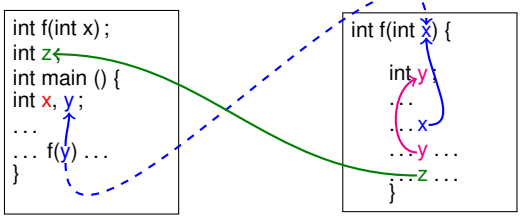
Les étapes ① et ② n'ont bien sûr pas lieu pour une fonction sans argument.

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





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 :

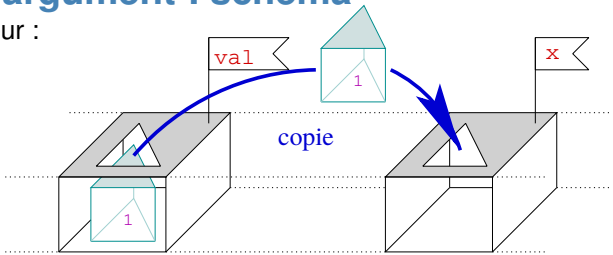
La variable locale associée à un argument passé par référence correspond à une **référence** sur l'objet associé à l'argument lors de l'appel.

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

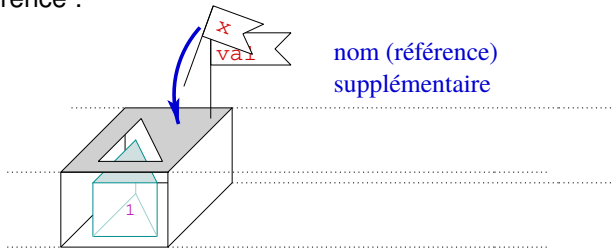
Le passage par référence peut être explicitement sélectionné en définissant le type des paramètres de la fonction comme étant des références (identifiées par le symbole `&`, par exemple `double& x`).

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 : utilisez toujours `const` dans vos passages d'arguments sauf si vous voulez **vraiment** modifier la variable passée (par référence).

Jean-Cédric Chappelier
& Jamila Sam

Optimisation (2) : données temporaires

Dans le cours sur les *variables*, nous avions 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 !)

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 à 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`

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 : $1+2+3=1\times2\times3$.

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`.

comment faire ?