Information, Calcul et Communication (partie programmation) : Fonctions

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Vidéos, transparents et quiz

www.coursera.org/learn/initiation-programmation-cpp/

☞ Semaine 4

(jusqu'à "Méthodologie")

Jamila Sam
& Jean-Cédric Chappelier

Notion de réutilisabilité

Un bon langage de programmation doit donc fournir des moyens pour permettre la **réutilisation** de portions de programmes.

☞ les **fonctions**

Pourquoi ne jamais dupliquer du code (copier/coller) :

- cela rend la **mise à jour** de ce programme plus **difficile** : reporter chaque modification de `P` dans chacune des copies de `P`
- cela **réduit** fortement **la compréhension** du programme résultant
- cela augmente inutilement la taille du programme

Jamila Sam
& Jean-Cédric Chappelier

Fonction (en programmation)

fonction = portion de programme réutilisable ou importante en soi

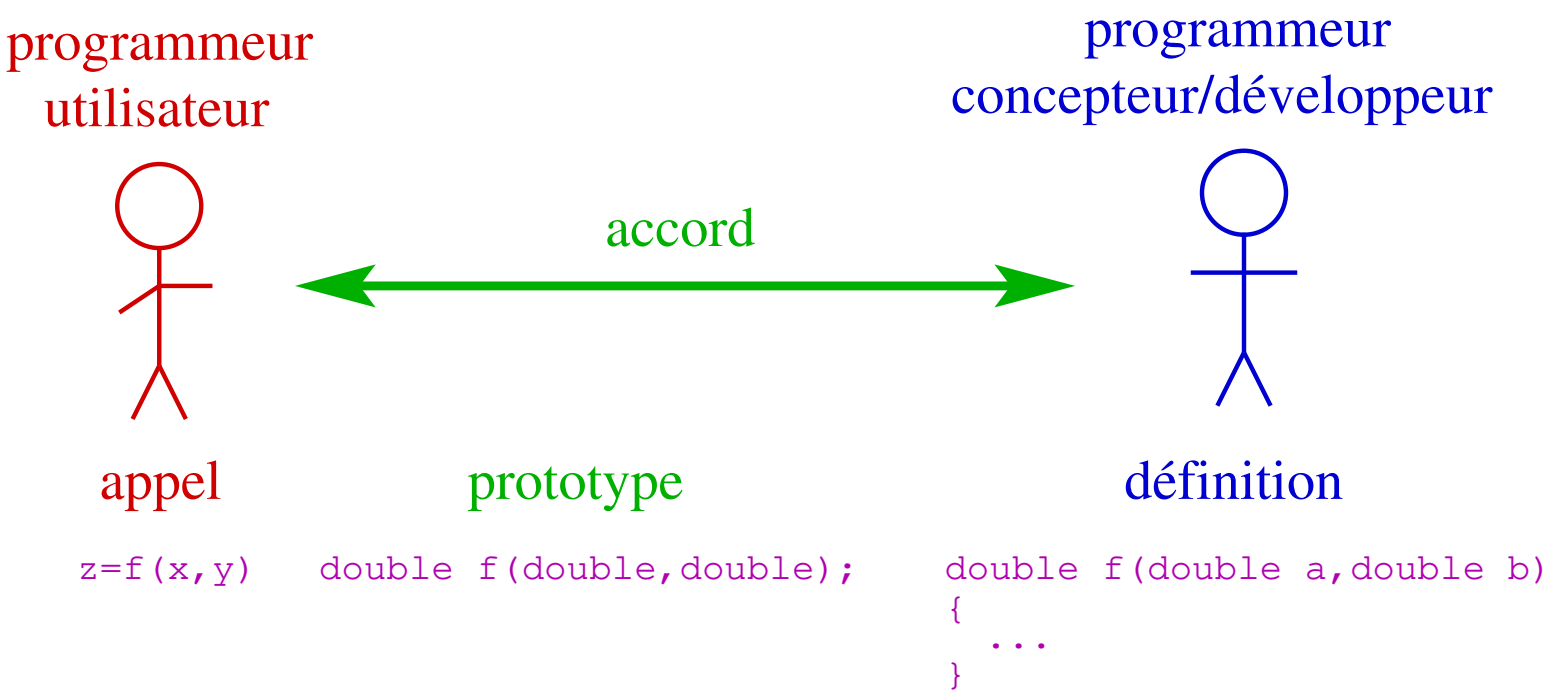
Plus précisément, une fonction est un objet logiciel caractérisé par :

- un corps : le programme à réutiliser/mettre en évidence et qui a justifié la création de la fonction ;
- un nom : référence à l'objet « *fonction* » lui-même, indiquée lors de sa création ;
- des paramètres : (les « *entrées* », on dit aussi « *arguments* ») ensemble de références à des objets définis à l'extérieur de la fonction dont les valeurs sont potentiellement utilisées dans le corps de la fonction ;
- un type et une valeur de retour (la « *sortie* ») : le type est indiqué dans le prototype et la valeur est indiquée dans le corps par la commande `return`.

Jamila Sam
& Jean-Cédric Chappelier

Les « 3 facettes » d'une fonction

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



Exemple complet

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

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;
}

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

prototype

appel

définition

Prototypage (2)

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).

Prototypage par la définition

Si la définition d'une fonction est faite **avant** son utilisation, cette définition peut également servir de prototype.
Dans ce cas, le prototypage est fait en même temps que la définition.

Par exemple, on peut directement écrire :

Conseil : il est cependant préférable de prototyper avant de définir une fonction.
On pourra, par la suite par exemple déplacer la définition de la fonction, ou bien écrire d'autres fonctions utilisant cette fonction, etc...

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, i.e. 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).

Jamila Sam
& Jean-Cédric Chappelier

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 ...
    }
```


EPFL

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 (*i.e.* 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*.

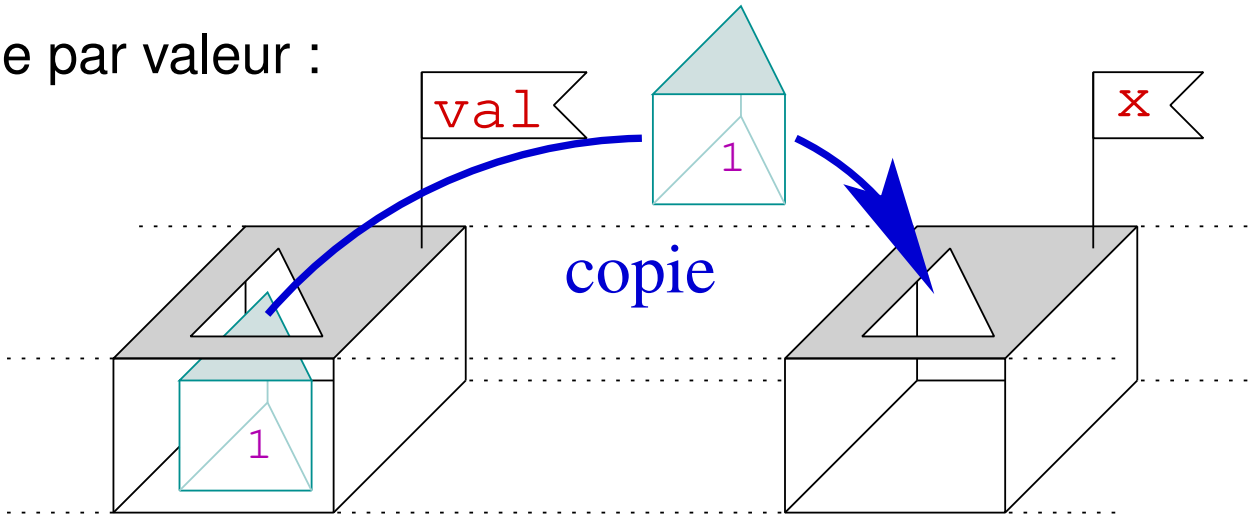
passage par référence :

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.

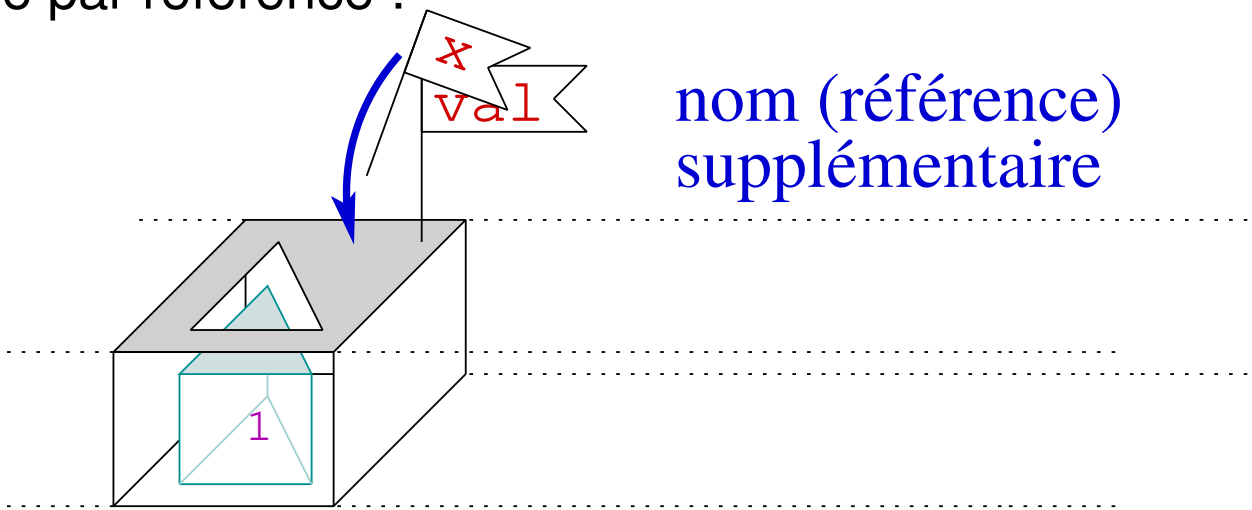
Une modification qui est effectuée à l'intérieur de la fonction peut alors se répercuter à l'extérieur de la fonction.
Le passage par référence peut être explicitement sélectionné en définissant le type des paramètres de la fonction comme étant des références (identifiées par le symbole **&**, par exemple `double& x`).

Passages d'argument : schéma

Passage par valeur :



Passage par référence :



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

Approfondissement

- Optimisation du passage des arguments

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 :

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

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


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.

Etude de cas

- IMC (revisité)

Jamila Sam
& Jean-Cédric Chappelier

Pour préparer le prochain cours

Vidéos et quiz du MOOC semaine 4 (suite) :

- Fonctions : arguments par défaut et surcharge [10 :25]

Jamila Sam
& Jean-Cédric Chappelier