Cours d'introduction à la programmation (en C++)
Fonctions

Jean-Cédric Chappelier
Jamila Sam
Vincent Lepetit

Faculté I&C

Notion de réutilisabilité

Pour l'instant : un programme est une séquence d'instructions

☞ mais sans **partage** des parties importantes ou utilisées plusieurs fois

Si une tâche, par exemple :

```
do {
  cout << "Entrez le nombre de points du joueur : ";
  cin >> nb;
} while ((nb < 0) or (nb > 100));
```

doit être exécutée à *plusieurs* endroits dans un plus gros programme

☞ recopie ? **NON !**

Bonne pratique : Ne *jamais dupliquer* de code en programmant :
Jamais de « copier-coller » !

☞ Ce que vous voudriez recopier doit être mis dans une **fonction**

Notion de réutilisabilité (2)

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

Cela rend le programme

- inutilement long
- difficile à comprendre
- difficile à **maintenir** :
  reporter chaque modification dans *chacune* des copies

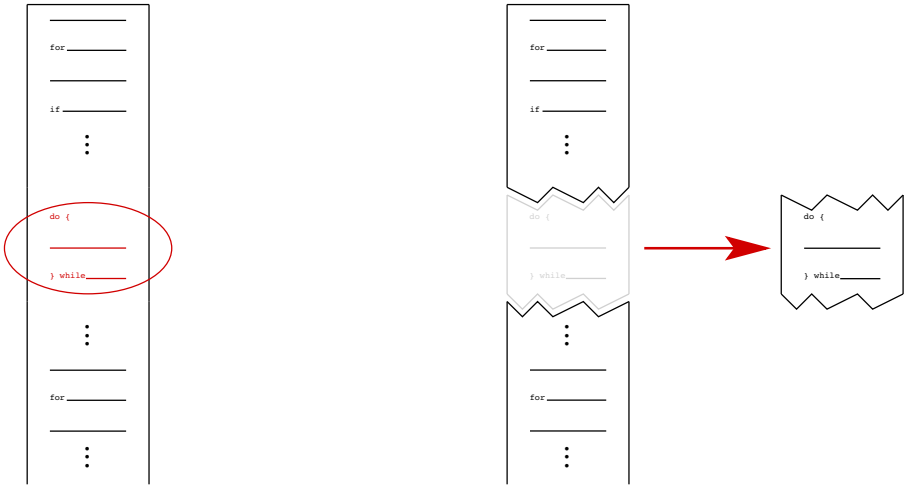
Tout bon langage de programmation fournit donc des moyens pour permettre la **réutilisation** de portions de programmes.

☞ les **fonctions**

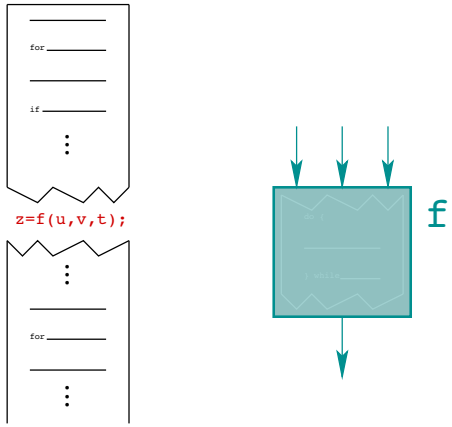
Exemple de fonction

```
int score (double points, double temps_jeu)
{
  int le_score(0);
  if (temps_jeu != 0.0) {
    le_score = 1000 * points / temps_jeu;
  }
  return le_score;
}
```

Notion de réutilisabilité : illustration



Notion de réutilisabilité : illustration



Fonction (en programmation)

fonction = portion de programme réutilisable ou importante en soi

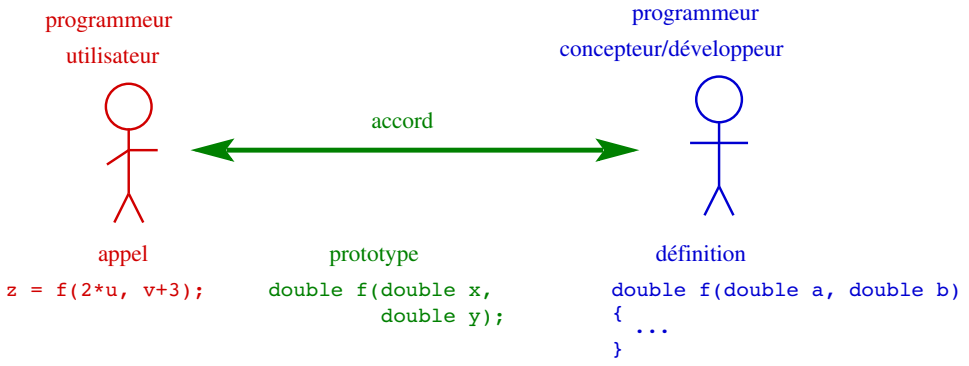
Plus précisément, une fonction est un objet logiciel caractérisé par :

- un corps : la portion de programme à réutiliser ou mettre en évidence, qui a justifié la création de la fonction ;
- un nom : par lequel on désignera cette fonction ;
- des paramètres : (les « *entrées* », on les appelle aussi « *arguments* ») ensemble de variables extérieures à la fonction dont le corps dépend pour fonctionner ;
- un type et une valeur de retour : (la « *sortie* ») ce que la fonction renvoie au reste du programme

L'utilisation de la fonction dans une autre partie du programme se nomme un appel de la fonction.

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

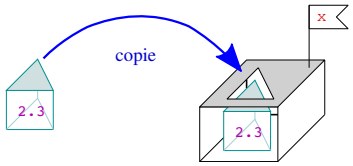
Évaluation d'un appel de fonction

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

Que se passe-t-il lors de l'appel suivant :

```
z = moyenne( 1.5 + 0.8, 3.4 * 1.25 );
```

1. évaluation des expressions passées en arguments :
   `1.5 + 0.8` ⟶ `2.3`
   `3.4 * 1.25` ⟶ `4.25`
2. affectation des paramètres :
   `x = 2.3`
   `y = 4.25`
3. exécution du corps de la fonction :
   rien dans ce cas (corps réduit au simple `return`)
4. évaluation de la valeur de retour (expression derrière `return`)
   `(x + y) / 2.0` ⟶ `3.275`
5. replacement de l'expression de l'appel par la valeur retournée :
   `z = 3.275;`



Évaluation d'un appel de fonction (résumé)

L'évaluation de l'**appel**

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

d'une fonction définie par

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

s'effectue de la façon suivante :

1. les *expressions* *arg1*, *arg2*, ..., *argN* passées en argument sont évaluées
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` rencontrée est évaluée...
5. ...et retournée comme résultat de de l'appel :
   cette valeur remplace l'expression de l'appel, i.e. l'expression

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

Évaluation d'un appel de fonction (résumé)

L'évaluation de l'**appel** d'une fonction s'effectue de la façon suivante :

1. les *expressions* passées en argument sont évaluées
2. les valeurs correspondantes sont **affectées** aux paramètres de la fonction
3. le corps de la fonction est exécuté
4. l'expression suivant la première commande `return` rencontrée est évaluée...
5. ...et retournée comme résultat de de l'appel :
   cette valeur remplace l'expression de l'appel

Les étapes 1 et 2 n'ont pas lieu pour une fonction sans arguments.

Les étapes 4 et 5 n'ont 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).

Appel : autre exemple

Une fonction peut appeler une autre fonction.
Il faut simplement respecter la règle d'or : avoir prototypé la fonction avant l'appel

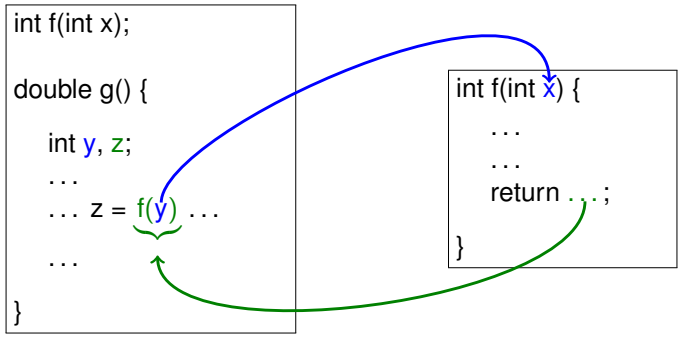
```
int score (double points, double temps_jeu);
void affiche_score (double points, double temps_jeu);

void affiche_score(int joueur, double points, double temps)
{
  cout << "  Joueur " << joueur
       << score(points, temps) << " points" << endl;
}

int score (double points, double temps_jeu)
{ // ... comme avant ...
}
```

Appel : résumé

L'évaluation de l'appel d'une fonction peut être schématisé de la façon suivante :



Résumé du jargon

« *Appeler la fonction* `f` » = utiliser la fonction `f` : `x = 2 * f(3);`

« *3 est passé en argument* » = (lors d'un appel) la valeur 3 est copiée dans un paramètre de la fonction :

`x = 2 * f(3);`

« *la fonction retourne la valeur de y* » = l'expression de l'appel de la fonction sera remplacée par la valeur retournée

```
 return y;
}
...
x = 2 * f(3);
```

Autres exemples : « `cos(0)` retourne le cosinus de 0 », « `cos(0)` retourne 1 ».

Le passage des arguments (1)

Que peut-on garantir sur la valeur d'une variable passée en argument lors d'un appel de fonction ?

Par exemple :

```
void f(int x) {
  x = x + 1;
  cout << "x=" << x;
}
int main() {
  int val(1);
  f(val);
  cout << " val=" << val << endl;
  return 0;
}
```

Que vaut `val` ? Qu'affiche le programme ?

Le passage des arguments (2)

On distingue 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

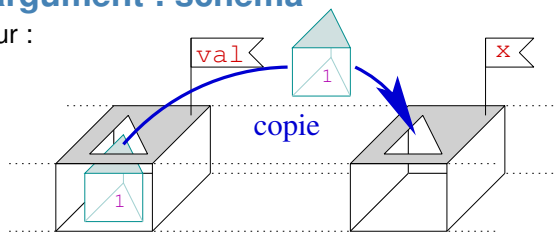
la variable locale associée à un argument passé par référence correspond à une **référence** sur l'objet associé à l'argument lors de l'appel.

☞ Une modification effectuée à l'intérieur de la fonction *se répercute* alors *à l'extérieur* de la fonction.

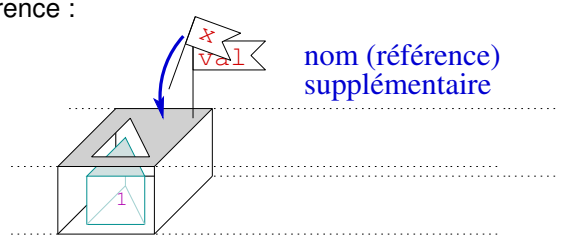
Le passage par référence doit être explicitement indiqué en utilisant le symbole `&` après le type ; par exemple : `double& x`.

Passages d'argument : schéma

Passage par valeur :



Passage par référence :



Exemple de passage par valeur

```
void f(int x) {
  x = x + 1;
  cout << "x=" << x;
}
int main() {
  int val(1);
  f(val);
  cout << " val=" << val << endl;
  return 0;
}
```

L'exécution de ce programme produit l'affichage :

`x=2 val=1`

Ce qui montre que les modifications effectuées à l'intérieur de la fonction `f()` **ne** se répercutent **pas** sur la variable extérieure `val` associée au paramètre `x` et passée par valeur.

Exemple de passage par référence

```
void f(int& x) {
  x = x + 1;
  cout << "x=" << x;
}
int main() {
  int val(1);
  f(val);
  cout << " val=" << val << endl;
  return 0;
}
```

L'exécution de ce programme produit l'affichage :

`x=2 val=2`

Ce qui montre que les modifications effectuées à l'intérieur de la fonction `f()` **se répercutent** sur la variable extérieure `val` associée au paramètre `x` et passée par référence.

Utilisation du passage par référence

Quand utiliser un passage par référence ?

☞ lorsque l'on souhaite modifier une variable

Par exemple :

- pour saisir une valeur :

```
void saisie_entier(int& a_lire);
...
int i(0);
...
saisie_entier(i);
```

Alternative : retourner la valeur :

```
int saisie_entier();
...
i = saisie_entier();
```

- pour « retourner » plusieurs valeurs :

```
void cartesiennes_vers_polaires(double x, double y,
                                double& distance, double& rayon);
```

Alternative : utiliser les structures (futur cours)

- pour « échanger » des variables :

```
void swap(int& i, int& j);
```

Prototypage

Toute fonction doit être *annoncée* avant d'être utilisée : **prototype**

prototype = *déclaration* de la fonction, sans en définir le corps :

- nom
- paramètres
- type de (la valeur de) retour

Syntaxe : $type\ nom\ (\ \overbrace{type_1\ id\_param_1\ ,\ \ldots\ ,\ type_N\ id\_param_N}^{liste\ de\ paramètres}\ )\ ;$

Exemples de prototypes :

```
double moyenne(double x, double y);
int nbHasard();
int score (double points, double temps_jeu);
double sqrt(double x);
```

Prototype – Bonnes pratiques

- Une fonction ne doit faire que ce pour quoi elle est prévue
  Ne pas faire des choses cachées («*effets de bords*») ni modifier de variables extérieures (non passées comme arguments)
- Choisissez des **noms pertinents** pour vos fonctions et vos paramètres

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

  ☞ Il est en particulier très important que le nom représente bien ce que doit faire la fonction
- Commencez toujours par faire le prototype de votre fonction :

  Demandez-vous ce qu'elle doit recevoir et retourner.

Résumé : attention à la syntaxe !

`int a;` : déclaration de *variable* non initialisée

`int a();` : prototype de **fonction** sans paramètre

`int a(5);` : déclaration/initialisation de *variable*

`a(5);` : appel de **fonction** à un argument

Définition des fonctions

La **définition** d'une fonction sert, comme son nom l'indique, à définir ce que fait la fonction :

- spécification du **corps** de la fonction

Syntaxe :

```
type nom ( liste de paramètres )
{
    instructions du corps de la fonction;
    return expression;
}
```

Exemple :

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

Corps de fonction

Le corps de la fonction est donc un **bloc**
dans lequel on peut utiliser les paramètres de la fonction (en plus des variables qui lui sont propres).

La valeur retournée par la fonction est indiquée par l'instruction :

```
return expression;
```

où l'*expression* a le même *type* que celui retourné par la fonction.

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

L'instruction `return` fait deux choses:

- elle précise la valeur qui sera fournie par la fonction en résultat
- elle met fin à l'exécution des instructions de la fonction.

L'expression après `return` est parfois réduite à une seule variable ou même à une valeur littérale, mais ce n'est pas une nécessité.

Remarques sur l'instruction `return` (1/4)

Il est possible de placer *plusieurs* instructions `return` dans une même fonction.

Par exemple, une fonction déterminant le maximum de deux valeurs peut s'écrire avec une instruction `return` :

```
double max2(double a, double b)
{
  double m;
  if (a > b) {
    m = a;
  } else {
    m = b;
  }
  return m;
}
```

ou deux :

```
double max2(double a, double b)
{
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Remarques sur l'instruction `return` (2/4)

Le type de la valeur retournée doit correspondre au type dans l'en-tête :

```
double f() {
  bool b(true);
  ...
  return b; // Erreur : mauvais type
}
```

Remarques sur l'instruction `return` (3/4)

`return` doit être la toute dernière instruction exécutée:

```
double lire() {
  cout << "Entrez un nombre : ";
  double n(0.0);
  cin >> n;
  return n;
  cout << "entré : " << n << endl; // Jamais exécuté
}
```

Remarques sur l'instruction `return` (4/4)

Le compilateur doit être sûr de toujours pouvoir exécuter un `return`:

```
double lire() {
  cout << "Entrez un nombre : ";
  double n(0.0);
  cin >> n;
  if (n > 0.0) {
    return n;
  }
  // Erreur : pas de return si n <= 0 !
}
```

Fonctions sans valeur de retour

Quand une fonction ne doit fournir aucun résultat (on appelle de telles fonctions des « **procédures** ») :

☞ définir une fonction **sans valeur de retour**

On utilise alors le type particulier `void` comme type de retour.

Dans ce cas la commande de retour `return` est optionnelle :

- soit on ne place aucun `return` dans le corps de la fonction
- soit on utilise l'instruction `return` sans la faire suivre d'une expression: `return;`

Exemple :

```
void affiche_racine(double a)
{
  if (a < 0.0) {
    return; /* Grâce à ce return, on quitte la fonction avant *
             * de calculer sqrt(a) si a est négatif.          */
  }
  cout << sqrt(a);
  // il n'est pas nécessaire de mettre un return ici
}
```

Fonctions sans paramètre

Il est aussi possible de définir des fonctions **sans paramètre**.
Il suffit, dans le prototype et la définition, d'utiliser une liste de paramètres vide : `()`

Exemple :

```
double saisie()
{
  double nb_points(0.0);

  do {
    cout << "Entrez le nombre de points (0-100) : ";
    cin >> nb_points;
  } while ((nb_points < 0.0) or (nb_points > 100.0));

  return nb_points;
}
```

La fonction `main()`

`main` est aussi une fonction avec un nom et un prototype imposés.

Par convention, tout programme C++ doit avoir une fonction `main`, qui est appelée automatiquement quand on exécute le programme.

Cette fonction doit retourner une valeur de type `int`. La valeur 0 indique par convention que le programme s'est bien déroulé.

Les deux seuls prototypes autorisés pour `main` sont :

```
int main();
int main(int argc, char** argv);
```

Seul le premier sera utilisé dans ce cours.

Pour résumer : Méthodologie pour construire une fonction

1. clairement identifier ce que **doit faire** la fonction
   (ce point n'est en fait que conceptuel, on n'écrit aucun code ici !)
   ☞ ne pas se préoccuper ici du *comment*, mais bel et bien du **quoi** !

   Les instructions dans le corps de la fonction dont la finalité n'est pas le calcul de la valeur de retour, ou qui modifient des objets extérieurs à la fonction (non passés en paramètre), sont appelées des « **effets de bord** ».
2. quels **arguments** ?
   ☞ que doit recevoir la fonction pour faire ce qu'elle doit ?
3. passage(s) par valeur(s) / référence(s) ?
   ☞ 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

Pour résumer : Méthodologie pour construire une fonction

4. quel type de retour ?
   ☞ que doit « retourner » la fonction ?
   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* :
   comment faire ce que doit faire la fonction ?
   ☞ c'est-à-dire écrire le corps de la fonction

Arguments par défaut

Lors de son prototypage, une fonction peut donner des **valeurs par défaut** à ses paramètres.
Il n'est alors pas nécessaire de fournir d'argument à ces paramètres lors de l'appel de la fonction.

La syntaxe d'un paramètre avec valeur par défaut est :

$$type\ identificateur = valeur$$

Attention : Les paramètres avec valeur par défaut doivent apparaître **en dernier** dans la liste des paramètres d'une fonction.

Arguments par défaut : exemple

Exemple :

```
void affiche_ligne(char elt, int nb = 5);

int main() {
  affiche_ligne('*');
  affiche_ligne('+', 8);
  return 0;
}

void affiche_ligne(char elt, int nb) {
  for(int i(0); i < nb; ++i) {
    cout << elt;
  }
  cout << endl;
}
```

Résultat :

```
*****
++++++++
```

Lors de l'appel `affiche_ligne('*')`, la valeur par défaut 5 est utilisée ; c'est strictement équivalent à `affiche_ligne('+', 5)`

Lors de l'appel `affiche_ligne('+', 8)`, la valeur explicite 8 est utilisée.

Arguments par défaut : Remarques

- Les arguments par défaut se spécifient dans le **prototype** et non pas dans la définition de la fonction
- Lors de l'appel à une fonction avec plusieurs paramètres ayant des valeurs par défaut, les arguments omis doivent être les derniers et omis **dans l'ordre** de la liste des paramètres.

  Exemple :

  `void f(int i, char c = 'a', double x = 0.0);`

  `f(1)` → correct (vaut `f(1, 'a', 0.0)`)
  `f(1, 'b')` → correct (vaut `f(1, 'b', 0.0)`)
  `f(1, 3.0)` → **incorrect !**
  `f(1, , 3.0)` → **incorrect !**
  `f(1, 'b', 3.0)` → correct

La surcharge de fonctions

En C++, il est de ce fait possible de définir **plusieurs fonctions de même nom** si ces fonctions n'ont pas les mêmes listes de paramètres : nombre ou types de paramètres différents.

Ce mécanisme, appelé **surcharge des fonctions**, est très utile pour écrire des fonctions « *sensibles* » au type de leurs arguments
c'est-à-dire des fonctions correspondant à des traitements de même nature mais s'appliquant à des entités de types différents.

La surcharge de fonctions : exemple

```
void affiche(int x) {
  cout << "entier : " << x << endl;
}
void affiche(double x) {
  cout << "reel : " << x << endl;
}
void affiche(int x1, int x2) {
  cout << "couple : " << x1 << x2 << endl;
}
```

`affiche(1)`, `affiche(1.0)` et `affiche(1,1)` produisent alors des affichages différents.

Remarque :

```
void affiche(int x);
void affiche(int x1, int x2 = 1);
```

est interdit ! ☞ ambiguïté