Information, Calcul et Communication (partie programmation) :

Cours de programmation (C++)
Fonctions (2))

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Vidéos, Quiz et transparents

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

☞ Semaine 4

EPFL

Objectifs du cours d'aujourd'hui

- Suite des rappels sur les fonctions en C++ :
  - rappel méthodologie
  - cas particuliers
  - surcharge
  - valeurs par défaut
  - Etude de cas (suite)
- Fonctions récursives

EPFL

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

EPFL

Effets de bord

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 objects extérieurs à la fonction, sont usuellement appelés des **effets de bord**.

Exemple :

```
int memoire(0);
int carre(int a);

main () {
  int j;
  ...
  j = carre(5);
  ...
}

int carre(int a) {
   // memorise la derniere valeur utilisee
   memoire = a;
   return a * a;
}
```


Effets de bord

Évitez absolument les effets de bords : c'est-à-dire de faire dans une fonction des modifications sur des objets extérieurs à la fonctions.

Bonne solution :

```
int carre(int argument);

main ()
{
  int memoire(0);
  int j;
  ...
  memoire = 5;
  j = carre(memoire);
  ...
}

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


La surcharge des fonctions

En C++, *les types des paramètres font partie intégrante de la définition d'une fonction*.

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 paramètres
c'est-à-dire des fonctions correspondant à des traitements de même nature mais s'appliquant à des entités de types différents.

La surcharge des 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

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 de valeur 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);

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

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

L'exécution de ce programme produit l'affichage :

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

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 fonctions avec plusieurs arguments ayant des valeurs par défaut, les arguments omis doivent être les derniers et omis **dans l'ordre** de la liste des arguments.

  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

Etude de cas

- Méthodologie de développement : tests et debugging

Fonctions récursives

Principe de l'approche récursive :

> *ramener le problème à résoudre à un sous-problème*, version simplifiée *du problème d'origine*.

Attention ! Pour que la résolution récursive soit correcte, il faut une

condition de terminaison

sinon, on risque une boucle infinie.

Exemple : Les tours de Hanoï

Jeu des tours de Hanoï :
déplacer une colonne de disques d'un pilier à un autre

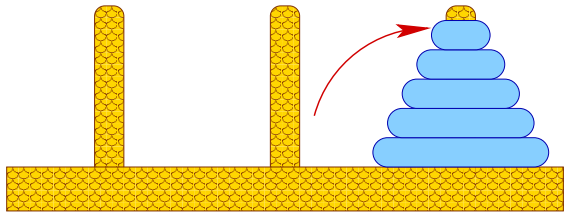
- en utilisant un seul pilier de transition (c'est-à-dire 3 piliers en tout)
- en ne posant un disque que sur le sol ou sur un disque plus grand.

Les tours de Hanoï (2)

Idée : si je sais le faire pour une pile de n disques,
je sais le faire pour $n+1$ disques

Démonstration :

- Déplacer les n disques du haut sur le pilier de transition (en utilisant la méthode que je connais par hypothèse)
- Mettre le dernier disque sur le pilier destination
- Redéplacer la tour de n disques du pilier de transition au pilier destination (en utilisant le pilier initial comme transition).



Les tours de Hanoï (3)

```
void hanoi(unsigned int n, unsigned int origine,
           unsigned int destination,
           Jeu jeu)
{
  if (n != 0) {
    unsigned int auxiliaire(autre(origine, destination));
    hanoi(n-1, origine, auxiliaire, jeu);
    deplace(jeu[origine], jeu[destination]);
    affiche(jeu);
    hanoi(n-1, auxiliaire, destination, jeu);
  }
}
```

Autre exemple

Calculer la somme des n premiers entiers.

Si je sais le faire pour n, je sais le faire pour $n+1$:

$$S(n+1) = (n+1) + S(n)$$

Condition d'arrêt :

Je sais le faire pour $n = 0$: $S(0) = 0$

Algorithme :

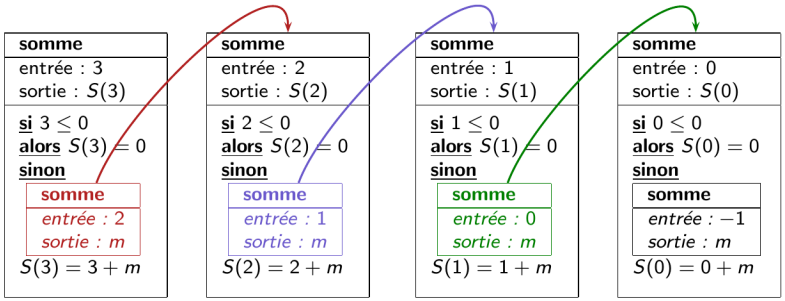
somme
entrée : n
sortie : $S(n)$

si $n \leq 0$
alors $S(n) \leftarrow 0$
sinon

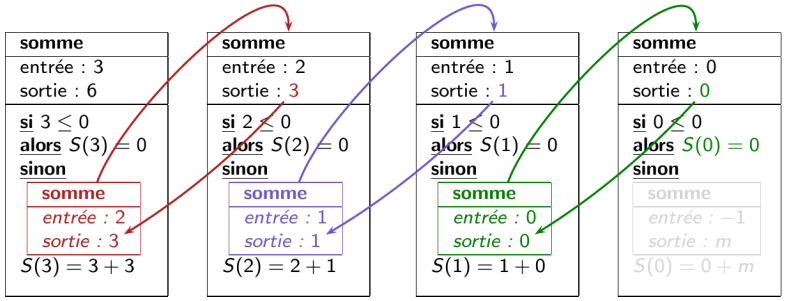
somme
entrée : $n-1$
sortie : m

$S(n) \leftarrow n + m$

Somme récursive : empilement des appels



Somme récursive : dépilement des appels



Somme récursive en C++

```
int somme(int n) {
  if (n <= 0) // condition arret
    return 0;
  else
    return (n + somme(n-1));
}
```

Remarque

Notez qu'il est parfois beaucoup mieux d'écrire la méthode sous une autre forme que la forme récursive.
Si l'on reprend l'exemple de la somme des n premiers entiers :

$$S(n+1) = (n+1) + S(n)$$

mais on a aussi (!) :

$$S(n) = \sum_{i=1}^{n} i$$

On préfèrera alors programmer le calcul de $S(n)$ comme une boucle (solution itérative) :

```
for (int i=1, s=0; i <= n ; i++) {
   s += i;
}
```

plutôt que sous forme récursive !..
ICC : Comparer formellement la vitesse des deux méthodes.

Remarque (2)

...voire utiliser une expression analytique, lorsqu'on en a une !
Par exemple :

$$S(n) = \frac{n\,(n+1)}{2}$$

que l'on pourra directement calculer par :

```
n * (n+1) / 2
```


Note : on peut toujours "*dérécursifier*" une fonction récursive.
C'est-à-dire en trouver un équivalent itératif.
(mais on n'a pas toujours une solution analytique au problème !)

Les fonctions récursives

Le schéma général d'une fonction récursive est donc le suivant :

```
type nom(type1 arg1, type2 arg2, ... ) {
  if (terminaison(arg1, arg2, ...)) {
     ...
  } else {
     type1 z; // si nécessaire pour
     type2 y1; //    des calculs intermédiaires
     type2 y2;
     ...
     z = nom(y1, y2, ...)
     ...
  }
}
```

même nom

Pour conclure

La solution récursive n'est pas toujours la seule solution et n'est pas toujours la plus efficace...
...mais elle est parfois beaucoup **plus simple** et/ou **plus pratique** à mettre en œuvre !
Exemples : tris, traitement de structures de données récursives (e.g., arbres, graphes, ...), ...

Les fonctions

Prototype (à mettre **avant** toute utilisation de la fonction) :

```
type nom ( type1 arg1, ..., typeN argN [ = valN ] );
```

`type` est `void` si la fonction ne retourne aucune valeur.

Définition :

```
type nom ( type1 arg1, ..., typeN argN )
{
   corps
   return value;
}
```

Passage par **valeur** :
`type f(type2 arg);`
`arg` ne peut pas être modifié par `f`

Passage par **référence** :
`type f(type2& arg);`
`arg` peut **être modifié** par `f`

Surcharge (exemple) :

```
void affiche (int arg);
void affiche (double arg);
void affiche (int arg1, int arg2);
```


EPFL

Pour préparer le prochain cours

Vidéos et quiz du MOOC semaine 5 :

- Tableaux : introduction [09 :34]
- Tableaux : déclaration et initialisation des vector [07 :44]
- Tableaux : utilisation des vector [15 :44]
- Tableaux : exemples simples (vector) [06 :53]
- Tableaux : fonctions spécifiques vector [11 :50]

EPFL