Programmation « orientée système »
Langage C – Introduction (2/3)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- structures de contrôle
- fonctions

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Structures de contrôle

C (comme la plupart des langages de programmation) permet la représentation d'enchaînements d'instructions complexes grâce aux **structures de contrôle**

☞ permet de modifier l'ordre linéaire d'exécution d'un programme.

☞ faire exécuter à la machine des tâches de façon répétitive, ou en fonction de certaines conditions (ou les deux).

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Retour à notre premier exemple de programme en C

```c
#include <stdio.h>
#include <math.h>
int main(void) {
  double b     = 0.0;
  double c     = 0.0;
  double delta = 0.0;

  printf("Entrez b : "); scanf("%lf", &b);
  printf("Entrez c : "); scanf("%lf", &c);
  delta = b*b - 4*c;
  if (delta < 0.0) {
    printf("pas de solutions reelles\n");
  } else if (delta == 0.0) {
    printf("une solution unique : %f\n", -b/2.0);
  } else {
    printf("deux solutions : %f et %f\n",
           (-b-sqrt(delta))/2.0,
           (-b+sqrt(delta))/2.0);
  }
 return 0;
}
```

données
traitements
structures de contrôle

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Les structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

Si $\Delta = 0$
  $x \leftarrow -\frac{b}{2}$
Sinon
  $x \leftarrow \frac{-b-\sqrt{\Delta}}{2}$, $y \leftarrow \frac{-b+\sqrt{\Delta}}{2}$

les boucles conditionnelles : *tant que ...*

Tant que réponse non valide
  poser la question

les itérations : *pour ... allant de ... à ...*

$$x = \sum_{i=1}^{5} \frac{1}{i^2}$$

$x \leftarrow 0$
Pour i de 1 à 5
  $x \leftarrow x + \frac{1}{i^2}$

Note : on peut toujours faire des itérations en utilisant des boucles, mais conceptuellement (et syntaxiquement aussi dans certains langages) il y a une différence.

Instructions composées : les blocs

En C, les instructions peuvent être **regroupées** en **blocs**.

Les blocs sont identifiés par des **délimiteurs explicites** de début et de fin de bloc : `{ }`

Exemple de bloc :

```
{
  int i ;
  double x ;
  printf("Valeurs pour i et x : \n");
  scanf("%d", &i);
  scanf("%d", &x);
  printf("Vous avez entré : i=%d, x=%d\n", i, x);
}
```


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Blocs et portée

presque comme en Java

En C, les blocs ont une grande autonomie.

Que ce soient des corps de fonctions ou non, ils peuvent contenir leurs propres déclarations / initialisations de variables :

```
if (x != 0.0) {
  double y = 0.0;
  /* ... */
  y = 5.0 / x;
  /* ... */
}
/* à partir d'ici on ne peut plus utiliser cet y */
```

(contrairement à Java) Un sous-bloc peut redéfinir une variable de même nom. Elle a pour portée ce bloc et masque la variable du bloc contenant.

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Notion de portée : local/global

différent de Java !

- les variables déclarées à l'intérieur d'un bloc sont appelées **variables locales** (au bloc). Elles ne sont accessibles qu'à l'intérieur du bloc.
- les variables déclarées en dehors de tout bloc (même du bloc `main(void){}`) seront appelées **variables globales** (au programme). Elles sont accessibles dans l'ensemble du programme.

Conseil : Ne jamais utiliser de variables globales (sauf peut être pour certaines constantes).
☞ effets de bords, contrôle ingérable

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Portée

différent de Java !

Les références à des objets externes au bloc courant sont autorisées.

En cas d'ambiguité dans le nom, c'est-à-dire de plusieurs variables (de blocs différents) portant le même nom, les règles utilisées en C sont les suivantes (*résolution à la portée la plus proche*) :

l'objet interne est systématiquement choisi (**portée locale**),
ou sinon celle du bloc le plus directement supérieur.

Conseil : **à éviter au maximum** d'utiliser plusieurs fois le même nom de variable (sauf peut être pour des variables locales aux boucles : `i`, `j`, etc..)

☞ confusion pour le lecteur humain et d'autres programmeurs potentiels, (ou même vous !) en cas de reprise future du code

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

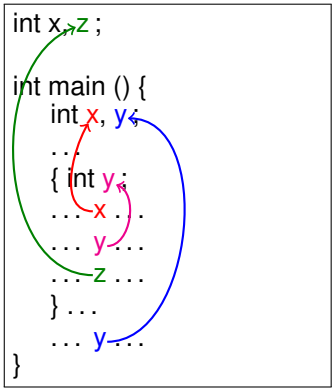
Exemple (à ne pas suivre !)

```c
#include <stdio.h>

int const MAX = 5;

int main(void) {
  int i = 120;

  { int i = 1;
    for (; i < MAX; ++i) {
      printf("%d ", i);
    }
  }
  printf("%d\n", i);
  return 0;
}
```

donne en sortie ➪

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Portée



Branchement conditionnel

comme en Java

Le branchement conditionnel permet d'exécuter des traitements selon certaines conditions.

La syntaxe générale d'un branchement conditionnel est

```
if (condition)
   Instructions 1
else
   Instructions 2
```

La condition est tout d'abord évaluée puis, si le résultat de l'évaluation est vrai alors la séquence d'instructions 1 est exécutée, sinon la séquence d'instructions 2 est exécutée.

Instructions 1 et *Instructions 2* sont soit une **instruction élémentaire**, soit un **bloc d'instructions**.

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Branchement conditionnel (Exemples)

instructions simples :

```
if (x != 0.0)
    printf("%f\n", 1.0/x);
else
    printf("erreur : x est nul.\n");
```

blocs d'instructions :

```
if (x > y) {
   min = y;
   max = x;
} else {
   min = x;
   max = y;
}
```


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Toujours utiliser des blocs

Conseil : **utilisez toujours la syntaxe avec des blocs**, même si vous n'avez qu'une seule instruction.

```
if (x != 0.0) {
  printf("%f\n", 1.0/x);
} else {
  printf("erreur : x est nul.\n");
}
```

- (cela vous évite de retenir 2 syntaxes différentes)
- c'est plus facile pour la maintenance :
  en cas d'ajout ultérieur d'instructions, le bloc est déjà présent
- et surtout ça peut vous éviter bien des soucis :
  cf grave faille de sécurité d'iOS (Apple) en février 2014

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Toujours utiliser des blocs (2)

Le 21 février 2014, Apple sortait iOS 7.0.6 et 6.1.6 pour corriger un grave problème de sécurité des versions précédentes.
Ce problème était du à l'erreur de programmation suivante :

```
static OSStatus SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
              SLBuffer signedParams, uint8_t *signature, UInt16 signatureLen)
{
  // ...
  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
  // ...
}
```

Source : http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Branchement conditionnel (suite)

comme en Java

Le second bloc (`else`) est optionnel :

```
if (x < 0.0) {
  x = -x;
}
y = sqrt(x);
```

(calcule $y = \sqrt{|x|}$)

On peut également enchaîner plusieurs conditions :

```
if (condition 1)
   Instructions 1
else if (condition 2)
   Instructions 2
...
else if (condition N)
   instructions N
else
   instructions N+1
```

Choix multiples

comme en Java

En C on peut écrire de façon plus synthétique l'enchaînement de plusieurs conditions dans le cas où l'on teste différentes valeurs d'une expression

```
switch (i) {
    case 1:
        instructions1;
        break;

    case 12:
        instructions2;
        break;
    ...
    case 36:
        instructionsN;
        break;

    default:
       instructionsN+1;
}
```

Exemple plus complexe

comme en Java

Si on ne mets pas de `break`, l'exécution ne passe pas à la fin du `switch`, mais continue l'exécution des instructions du `case` suivant :

```
switch (a+b) {
 case 0: instruction1; /* executé uniquement lorsque */
   break;              /* (a+b) vaut 0               */
 case 2:
 case 3: instruction2; /* lorsque (a+b) vaut 2 ou 3  */
 case 4:
 case 8: instruction3; /* lorsque (a+b) vaut 2, 3, 4 ou 8 */
   break;
 default: instruction4; /* dans tous les autres cas  */
}
```


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Boucles

comme en Java

Les boucles permettent la mise en œuvre **répétitive** d'un traitement.
La répétition est **contrôlée** par une **condition de continuation**.

La syntaxe générale d'une boucle avec condition de continuation *a priori* (on veut **tester** la condition **avant d'exécuter** les instructions) est :

```
while (condition)
    Instructions
```

Tant que la condition de continuation est vérifiée, les instructions sont exécutées.

Même remarque ici que pour `if` :
Instructions est soit une **instruction élémentaire** (suivie de son `;`), soit un **bloc d'instructions**.
Il est plutôt conseillé de toujours utiliser la syntaxe par bloc.

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Boucles (2)

comme en Java

La syntaxe générale d'une boucle avec condition de continuation *a posteriori* (on veut **exécuter** les instructions au moins une fois **avant de tester** la condition) est :

```
do
    Instructions
while (condition);
```

Les instructions sont exécutées **jusqu'à ce que** la condition de continuation soit fausse (et au moins une fois au départ, indépendemment de la valeur de la condition).

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

L'itération `for`

comme en Java

Les itérations permettent l'**application itérative** d'un traitement, contrôlée par une **initialisation**, une **condition de continuation**, et une opération de **mise à jour** de certaines variables.
Syntaxe :

```
for (initialisation; condition;  mise_à_jour)
    Instructions
```

Même remarque ici que pour `if` et `while` :
Instructions est soit une **instruction élémentaire** (suivie de son `;`), soit un **bloc d'instructions**.
Il est plutôt conseillé de toujours utiliser la syntaxe par bloc.

Note : Une boucle `for` (sans `continue` !) est équivalente à la boucle `while` suivante :

```
{ initialisation;
   while (condition) {
       Instructions
       mise_à_jour;
} }
```

L'itération `for` : Exemple

Remarque : si plusieurs instructions d'initialisation ou de mise à jour sont nécessaires, elles sont séparées par des **virgules**. Elles sont **exécutées de la gauche vers la droite**.

Exemple :

```
int i, s;
for (i = 0, s = 0; i < 5; s += i, ++i) {
  printf("%d, %d\n", i, s);
}
```

affichera

```
0, 0
1, 0
2, 1
3, 3
4, 6
```

Sauts : `break` et `continue`

comme en Java

C fournit deux instructions prédéfinies, `break` et `continue`, permettant de contrôler de façon plus fine le déroulement d'une boucle.

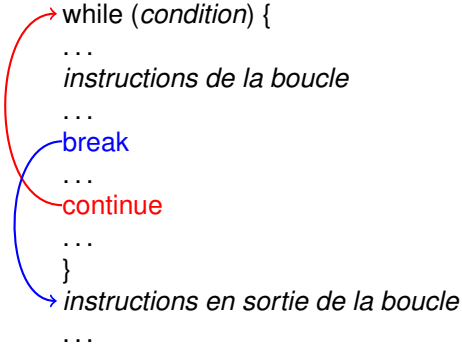
- Si l'instruction `break` est exécutée au sein du bloc intérieur de la boucle, l'exécution de la boucle est interrompue (quelque soit l'état de la condition de contrôle) ;
- Si l'instruction `continue` est exécutée au sein du bloc intérieur de la boucle, l'exécution du bloc est interrompue et la condition de continuation est évaluée pour déterminer si l'exécution de la boucle doit être poursuivie.
  Dans le cas d'un `for` la partie mise à jour est également effectuée (avant l'évaluation de la condition).

Conseil : En toute rigueur on n'aurait **pas besoin** de ces instructions, et tout bon programmeur évite de les utiliser.

> Pour la petite histoire, un bug lié à une mauvaise utilisation de `break;` a conduit à l'effondrement du réseau téléphonique longue distance d'AT&T, le 15 janvier 1990. Plus de 16'000 usagers ont perdu l'usage de leur téléphone pendant près de 9 heures. 70'000'000 d'appels ont été perdus.
> [P. Van der Linden, *Expert C Programming*, 1994.]

Instructions `break` et `continue`

comme en Java



Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Instruction `break` : exemple

Exemple d'utilisation de `break`
Une **mauvaise** (!) façon de simuler une boucle avec condition d'arrêt

```
while (1) {
  Instruction 1;
  /* ... */
  if (condition d arrêt)
      break;
}
autres instructions;
```

pas de variable locale

☞ Question : quelle est la bonne façon d'écrire le code ci-dessus ?

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Instruction `continue` : exemple

Exemple d'utilisation de `continue`

```
int i;
...
i = 0;
while (i < 100) {
  ++i;
  if ((i % 2) == 0) continue;
  /* la suite n'est exécutée que
     pour les entiers ... */
  Instructions;
  ...
}
suite;
```

On suppose que `i` n'est pas modifié

☞ Question : quelle est une meilleure façon d'écrire le code ci-dessus ?

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Les structures de contrôle

les branchements conditionnels : *si ... alors ...*

```
if (condition)
    instructions
.............................
if (condition 1)
    instructions 1
...
else if (condition N)
    instructions N
else
    instructions N+1
```

```
switch (expression) {
   case valeur:
       instructions;
       break;
   ...
   default:
       instructions;
}
```

les boucles conditionnelles : *tant que ...*

```
while (condition)
Instructions
```

```
do Instructions while
(condition);
```

les itérations : *pour ... allant de ... à ...*

```
for (initialisation ; condition ; increment)
    instructions
```

les sauts : `break;` et `continue;`

Note : `instructions` représente 1 instruction élémentaire ou un bloc.
`instructions;` représente une suite d'instructions élémentaires.

Réutilisabilité

Pour l'instant : programme = séquence d'instructions, simples ou composées

☞ mais pas de partage des parties importantes ou utilisées plusieurs fois

Conseil : Ne **jamais** faire **de copier/coller** en programmant.

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

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

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Fonction (en programmation)

presque comme en Java

fonction = portion de programme réutilisable
$\simeq$ méthode en Java (sauf que les fonctions ne sont pas encapsulées dans des objets)

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

un nom référence à l'objet « *fonction* » lui-même, indiquée lors de sa création ;

un corps le programme à réutiliser qui a justifié la création de la fonction ;

des arguments (les « *entrées* ») 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`.

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple de fonction

```
int diviseur(int nombre)
{
  if (0 == (nombre % 2)) return 2;

  const double borne_max = sqrt((double) nombre);
  for (int i = 3; i <= borne_max; i += 2) {
    if (0 == (nombre % i)) return i;
  }

  return nombre;
}
```


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Prototypage

différent de Java !

Le **prototypage** est la déclaration de la fonction sans en définir le corps.

Il sert à définir pour le reste du programme quels sont le nom, les arguments et le type de la valeur de sortie de la fonction.

Toute fonction doit être **prototypée avant d'être utilisée**.

Syntaxe :

type nom (liste d'arguments) ;

où *type* est le type de la valeur renvoyée par la fonction, *nom* son nom et *liste d'arguments* la liste de ses arguments, de la forme :

$$type_1\ id\_arg_1,\ \ldots,\ type_N\ id\_arg_N$$

Cette liste peut être vide si la fonction n'a pas besoin d'arguments.

Exemples de prototypes :

```
double moyenne(double x, double y);
int nb_au_hasard(void);
```

Prototypage (2)

Dans les prototypes des fonctions, *les identificateurs des arguments 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 arguments dans le prototypage des fonctions et **choisissez des noms pertinents**.
Cela augmente la lisibilité de votre code (et donc facilite sa maintenance).

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Définition des fonctions

comme en Java

La **définition** d'une fonction sert à définir (! !) ce que fait la fonction.
C'est donc la spécification de son **corps**.

Contrairement au prototypage, il n'est pas nécessaire d'avoir défini une fonction avant de l'utiliser. La définition d'une fonction peut être faite n'importe où ailleurs (mais après son prototypage !).

Syntaxe :

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


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Corps de fonction

comme en Java

Le corps de la fonction est donc un **bloc** (au sens vu dans le cours précédent).

En plus des variables qui lui sont propres, ce bloc peut également utiliser les variables arguments de la fonction :
les *arguments* d'une fonction constituent des *références internes au corps* de la fonction.

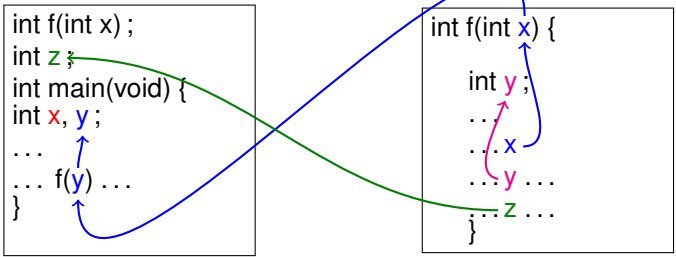
La valeur retournée par la fonction est indiquée par l'instruction :

```
return expression;
```

où *expression* est une expression du type retourné par la fonction (*type* dans l'exemple précédent), parfois réduite à une seule variable ou même une valeur littérale.

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Portée et fonctions



Exemple complet

```c
#include <stdio.h>

double moyenne (double nombre_1, double nombre_2);

int main(void) {
  double note1, note2;
  printf("Entrez vos deux notes :");
  scanf("%lf %lf", &note1, &note2);
  printf("Votre moyenne est : %f\n",
         moyenne(note1, note2));
  return 0;
}

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


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Fonctions sans argument

presque comme en Java

Il est bien sûr possible de définir des fonctions **sans argument**. Il suffit, dans le prototype et la définition, d'utiliser le mot clé « `void` » en place de la liste d'arguments.

Exemple :

```
int saisie_entier(void) {
  int i;
  printf("entrez un entier: ");
  scanf("%d", &i);
  return i;
}

int main(void) {
  int val = saisie_entier();
  printf("%d", val);
  return 0;
}
```

Fonctions sans argument

presque comme en Java

Remarque :

- en C,
  ```
  Type f();
  ```
  est une « *deprecated feature* » indiquant un « pré-prototype » de `f`, pour lequel le nombre d'arguments n'est pas spécifié (notez bien la différence : cela ne veut pas dire qu'il soit 0 !)

- en C++,
  ```
  Type f();
  ```
  signifie *exactement la même chose* que
  ```
  Type f(void);
  ```
  (et les deux sont acceptés de façon égale).

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Fonctions sans valeur de retour

Il est aussi possible de définir des fonctions **sans valeur de retour**, c'est-à-dire des fonctions qui ne renvoient rien.
(on appelle de telles fonctions des « **procédures** »)

comme en Java

Il suffit, dans le prototype et la définition, d'utiliser le type prédéfini `void` comme type de retour.

Dans ce cas la commande de retour `return` est optionnelle.
Si vous souhaitez néamoins la mettre (par exemple pour forcer la sortie de la procédure), elle se met alors **sans** argument : `return;`

Exemple :

```
void affiche_entiers(int n) {
  for (int i = 0; i < n; ++i) {
    printf("%d\n", i); }
}

int main(void) {
  affiche_entiers(10);
  return 0;
}
```


Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Évaluation d'un appel de fonction

comme en Java

Pour une fonction définie par

`int 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 non spécifié !)
2. les valeurs correspondantes sont **affectées** aux arguments `x1`, `x2`, ..., `xN` de la fonction `f` (variables locales au corps de `f`)
   Concrètement, ces deux 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`*`)`

Exemple

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

pour la fonction `moyenne()` définie précédemment, considérons l'appel suivant :

```
z = moyenne(1.0 + sqrt(24.0), 12 % 5);
```

① évaluation des expressions passées en arguments :
`12 % 5` $\Rightarrow$ `2`
`1.0 + sqrt(24.0)` $\Rightarrow$ `5.89897948557`

② affectation des arguments :
`x = 5.89897948557`
`y = 2`

③ exécution du corps de la fonction : rien dans ce cas (le corps réduit au simple `return`)

④ évaluation de la valeur de retour
`(x + y)/2.0` $\Rightarrow$ `3.94948974278`

⑤ replacement de l'expression de l'appel par sa valeur :
`z = 3.94948974278;`

Le passage des arguments

différent de Java ?

On distingue en général *deux types de passage d'arguments* pour les fonctions :

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.

En Java, il n'y a (essentiellement) que des passages par ...

En C, il n'y a *exclusivement* **que des passages par valeur**.
La simulation du passage par référence se fait en C en passant la valeur d'un pointeur.

Exemple de passage par valeur

```
void f(int x) {
  x = x + 1;
  printf("x=%d", x);
}
int main(void) {
  int val = 1;
  f(val);
  printf(" val=%d\n", val);
  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 `val` associée à l'argument `x` passé par valeur.

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple de passage par référence simulé par pointeurs

Le cours sur les **pointeurs** viendra la semaine prochaine.
Il s'agit ici d'insister sur les effets (de la simulation) du passage par référence.

```
void f(int* x) { /* passage par "reference" */
  *x = *x + 1;
  printf("x=%d", *x);
}
int main(void) {
  int val = 1;
  f(&val);
  printf(" val=%d\n", val);
  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 à l'argument `*x` passé par référence.

La surcharge des fonctions

différent de Java !

Rappel : c'est quoi la surcharge ?

possibilité de définir **plusieurs fonctions de même nom** si ces fonctions *n'ont pas les mêmes listes d'arguments* : nombre ou types d'arguments différents.

en C, il n'y a pas de surchage !

Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Les fonctions

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

```
type nom ( type1 arg1, ..., typeN argN );
```

`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);
f(x)
```

☞ `x` **ne** peut **pas** être modifié par `f`

Passage par **référence** (valeur de pointeur en fait) :

```
type f(type2* arg);
f(&x)
```

☞ `x` peut **être modifié** par `f`