Programmation C/C++

Syntaxe de base du C
Généralités sur la Compilation dans Visual Studio

Nicolas Gazères

Dassault Systèmes
DS Research, Life Sciences
ngs@3ds.com

Qu'est-ce que la Compilation ?

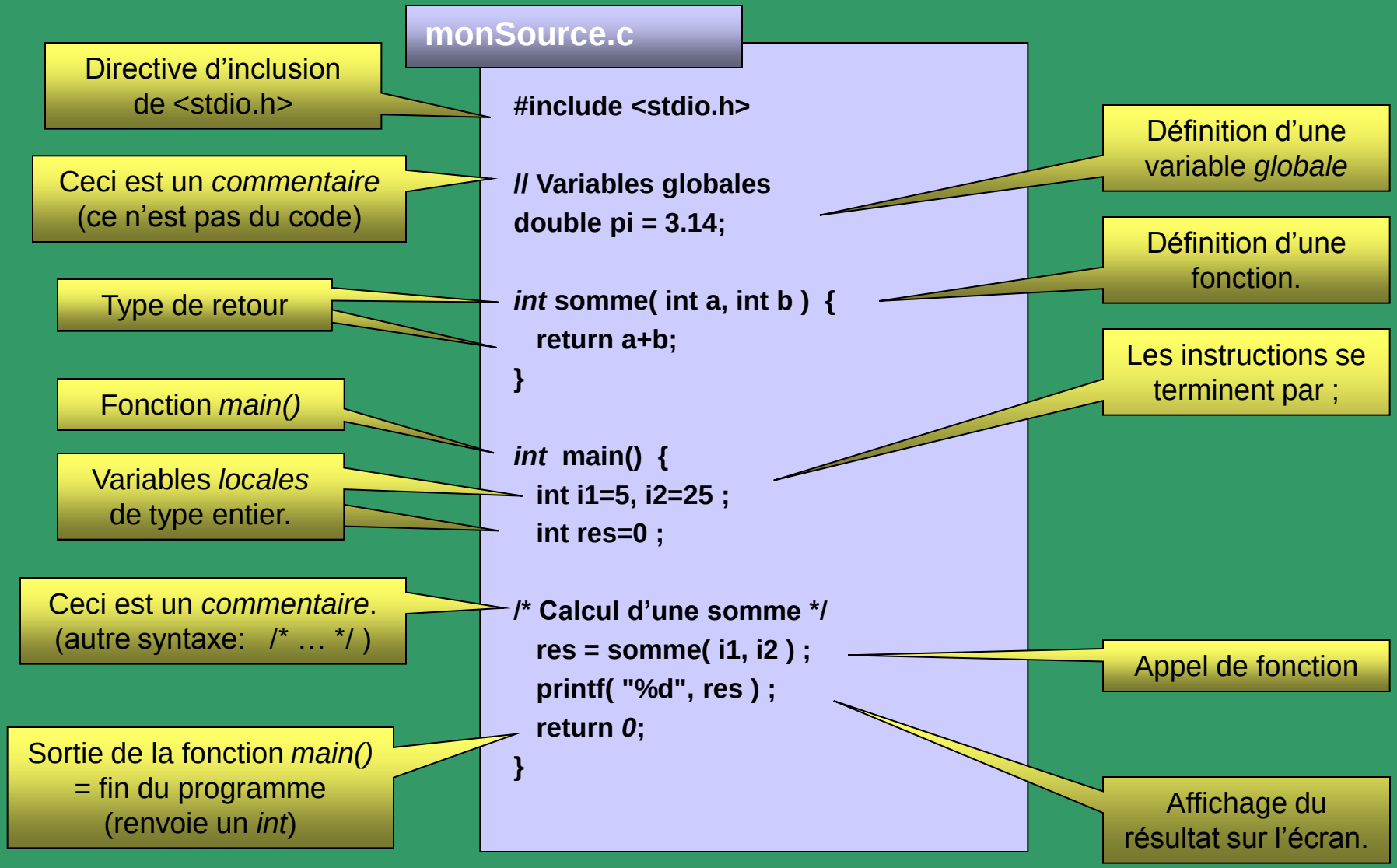
- Qu'est-ce qu'un ***fichier source*** *?*
  - C'est le type fondamental de fichiers que fabrique un humain quand il souhaite programmer.
  - Il exprime la logique du programme.
  - Il est écrit dans un langage intelligible pour l'humain,
    - … mais pas pour la machine.
  - Sur disque, c'est en général un fichier d'extension ".c" (pour le langage C).

- La ***compilation*** est la transformation par la machine d'un *fichier source* (ou plusieurs) en un *fichier objet* (ou plusieurs), puis en un *fichier exécutable*.
  - On utilise pour cela (1) un *compilateur* et (2) un *éditeur de liens* (*linker).*

- Qu'est-ce qu'un ***fichier exécutable*** *?*
  - C'est le seul type de fichier compréhensible par la machine.
    - Il n'est en général plus compréhensible par l'humain.
  - C'est la traduction du (ou des) fichier(s) source(s)
    - il exprime exactement la même logique, dans un format différent.
  - Il spécifie un *point d'entrée*:
    - C'est l'endroit précis où commence l'exécution du programme.
  - Un fichier exécutable peut être lancé plusieurs fois simultanément:
    - Chaque exécution individuelle porte le nom de *process.*

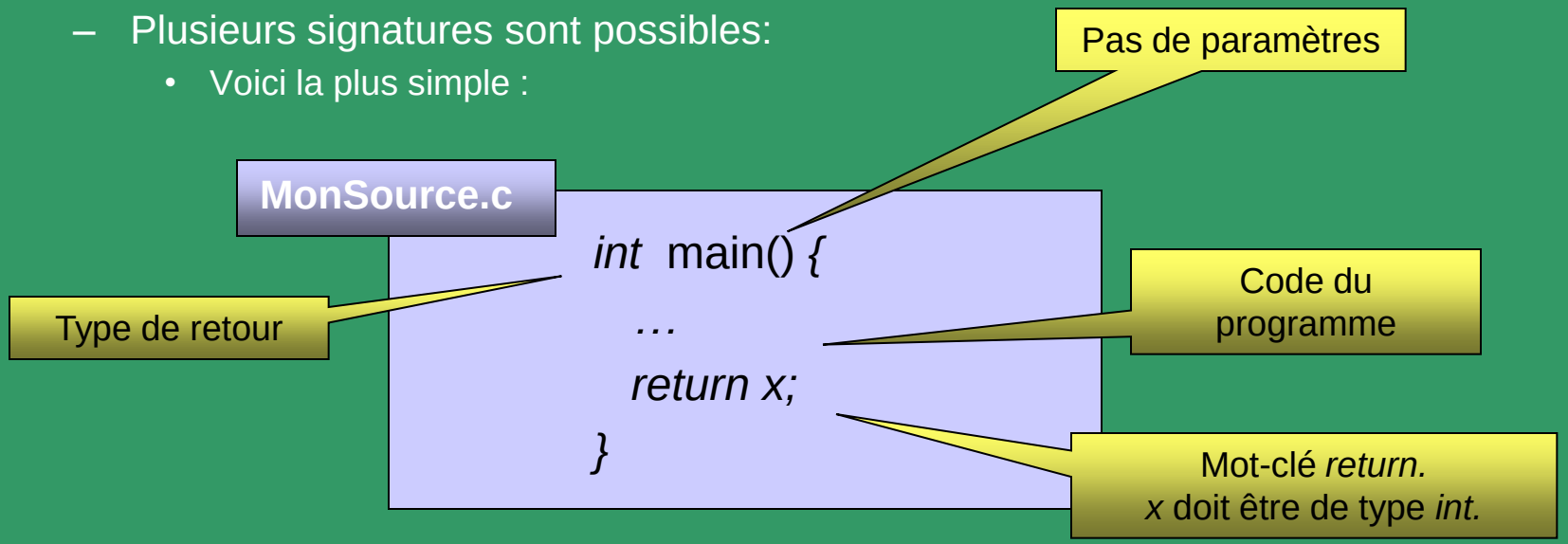
Fichier «source»

Exemple d'exécutable construit à partir d'un seul source



Points d'entrée et point de sortie d'un exécutable

- Le ***point d'entrée*** de l'exécutable:
  - C'est la fonction *main(), pour un programme C*
  - Plusieurs signatures sont possibles:
    - Voici la plus simple :



  - Cette signature ne permet pas de récupérer les paramètres passés au programme lors de son lancement (*mais voir cours prochain*).
- Le ***point de sortie*** de l'exécutable
  - … est le plus souvent une instruction *return* .
  - Dans notre exemple, *x* doit être de type *int*
    - pour correspondre à la déclaration de *main().*

Les variables

- Définition
  - Une ***variable*** est un symbole qui, à tout instant, possède une *valeur* particulière, d'un *type* donné
    - Cette valeur pouvant changer au cours du temps.
  - Un ***type*** est défini par
    - l'ensemble des valeurs possibles pour une variable donnée
    - L'ensemble des opérations possibles sur cette variable.
  - En C/C++, le type d'une variable est entièrement déterminé par sa déclaration.
    - Dès qu'elle déclarée, une variable donnée ne peut plus changer de type.
      - On verra qu'on peut convertir la *valeur* de la variable vers un autre type, mais il ne s'agit plus de la variable *initiale*, ie. la valeur du nouveau type est alors contenue dans une *autre* variable (du bon type).
- En C "pur", les variables doivent obligatoirement être déclarées en tête de fonction.
  - En C++, cette contrainte est relaxée:
    → la variable doit juste être déclarée avant sa première utilisation
- Les noms de variables:
  - Les mots-clés du langage (*for*, *if*, *else*, …) sont réservés et ne peuvent pas servir de nom de variable.
    → il y a erreur de compilation en cas de violation de la règle.

Le bit, l'octet, les types scalaires

- Le bit (*bi*nary uni*t*): vaut 0 ou 1.
- L'octet: un assemblage de 8 bits.
  - peut coder 256 valeurs en base 2 (*0-255*).
- En C existent un certain nombre de types scalaires
  - Qui diffèrent par leur taille en octets
    - *short, int, long, __int64 (long long* sous Unix*)*
  - Existent en version signée/non-signée
    - signed int/unsigned int, …
- Sur les architectures *Intel / Windows* :

char	1 octet	short	2 octets
int/long	4 octets	__int64	8 octets

- Le langage C impose la contrainte suivante (valable quelle que soit l'architecture):

$$\text{short} \leq \text{int} \leq \text{long} \leq \text{\_\_int64}$$

Les types réels (flottants)

- Une variété de types réels est disponible:
  - Flottant simple précision: *float x ;*
  - Flottant double précision: *double y ;*
  - Flottant précision étendue: *long double z ;*

- Une valeur numérique littérale est par défaut de type *double* .

```
3.47*(x-0.97)     // version double
```

- Pour spécifier explicitement qu'un littéral est
  - en *simple précision*, ajouter le suffixe *f* ou *F (ie. 3.47F)*
  - en *précision étendue*, ajouter le suffixe *l* ou *L (ie. 3.47L)*

```
3.47F*(x-0.97F)     // version simple précision
3.47L*(x-0.97L)     // version précision étendue
```

Conversion de type *explicite* (*cast*)

- Utilité
  - Lorsque le programmeur estime que le résultat d'un calcul peut être contenu – *sans perte d'information* – dans une variable d'un type « plus petit » que celui de l'expression,
  - il utilise une conversion de type explicite.
    - également appelée *cast* (ou *type cast*, ou *type coercion…)*

- Exemple

```
double x=2.0, y=4.0 ;
float  f1 = x*y;              // Perte d'info a priori → warning du compilateur
float  f2 = (float) (x*y);    // On sait que le résultat tient dans un float
                              // → Cast → plus de warning.
float f3 = (float) x;         // (***)
```

  - (***): il est important de comprendre que:
    - Ce n'est pas la variable castée *x* qui passe du type *double* au type *float*…
    - …mais plutôt que le résultat de l'expression (ici *x*) est manipulé dans une unité de stockage *temporaire* de type *float*, non-nommée et de durée de vie très courte.

Les opérateurs

- Comparaisons

Test d'égalité	==	Non-égalité	!=
Inférieur strict	<	Inférieur ou égal	<=
Supérieur strict	>	Supérieur ou égal	>=

- Affectation
  - C'est l'opérateur =

```
a = b+c*d ;
```

  - ie. La valeur de l'expression *b+c*d* est stockée dans *a*.
- Opérations classiques

Addition	+	Soustraction	-
Produit	*	Division et modulo	/, %
ET	&&	OU	||

- Opérateurs combinés: +=, -=, *=, /=

```
a += b ;
```

équivalent à

```
a = a+b ;
```

Opérateur d'incrémentation (++) et de décrémentation (--)

- Principe
  - L'opérateur ++ incrémente la variable de 1.
  - L'opérateur -- décrémente la variable de 1.
- Exemple

```
int x = 1;
x++;    // postfixé
        // → x vaut maintenant 2
--x;    // préfixé
        // → x vaut maintenant 1 à nouveau
```

- Les opérateurs ++ et -- peuvent être préfixés ou postfixés.
  - *Postfixé:* « *x*++ » renvoie « le *x* d'avant »
  - *Préfixé:* « ++*x* » renvoie « le *x* d'après » (ie. le x d'avant plus un)
  - Dans les deux cas, *x* est incrémenté de 1.
- *Lequel est le plus performant (ie. rapide) ?*
  - « *x*++ » est équivalent à « *y=x, x=x+1, return y* »
  - « ++*x* » est équivalent à « *x=x+1, return x* » ← *celui-là !*

Priorité des opérateurs

- Exemple

```
if ( ! a==b ) {
   …
}
```

- Deux interprétations possibles
  - Si == est prioritaire sur !

```
if ( ! (a==b) ) {
   …
}
```

  - Si ! est prioritaire sur ==

```
if ( (!a)==b ) {
   …
}
```

La réponse est donnée par le tableau suivant…

Priorité des opérateurs

Priorité	Opérateur	Description	Associativité

1	`()`
	`[]`
	`.  ->`
	`++  --`
		Parenthèses d'appel de fonction
		Crochets d'accès à un tableau par indice
		Accès à un membre de structure depuis un objet / depuis un pointeur
		Incrément/décrément postfixé
			de gauche à droite

2	`++  --`
	`+  -`
	`!  ~`
	`(`*`type`*`)`
	`*  &  sizeof`
		Incrément/décrément préfixé
		Plus/moins unaire
		Négation logique / complément bit-à-bit
		Cast (conversion de *type* explicite)
		Déréférencement / Prise de l'addresse / Détermination de la taille en octets
			de droite à gauche

3	`*  /  %`	Multiplication/division/modulo	de gauche à droite
4	`+  -`	Addition/soustraction (binaire)	de gauche à droite
5	`<<  >>`	Décalage à gauche bit-à-bit/décalage à droite bit-à-bit	de gauche à droite
6	`<  <=  >  >=`	inférieur à / inférieur ou égal à / supérieur à / supérieur ou égal à	de gauche à droite
7	`==  !=`	est égal à / est différent de	de gauche à droite
8	`&`	ET bit-à-bit (Bitwise AND)	de gauche à droite
9	`^`	OU exclusif bit-à-bit (Bitwise exclusive OR)	de gauche à droite
10	`|`	OU simple bit-à-bit (Bitwise inclusive OR)	de gauche à droite
11	`&&`	ET logique	de gauche à droite
12	`||`	OU logique	de gauche à droite
13	`?:`	Opérateur conditionnel ternaire	de droite à gauche

14	`=`
	`+=  -=  *=  /=`
	`%=  &=  ^=  |=`
	`<<=  >>=`
		Affectation
		Affectation avec addition / soustraction / multiplication / division
		Affectation avec modulo / ET bit-à-bit / OU exclusif bit-à-bit / OU simple bit-à-bit
		Affectation avec décalage à gauche bit-à-bit / avec décalage à droite bit-à-bit
			de droite à gauche

15	`,`	Virgule (séparation d'expressions)	de gauche à droite

Les promotions de types

- Le principe général: 2 règles
  - 1/ Les arguments sont convertis dans le type le plus riche figurant dans l'expression, avant application de l'opérateur.
  - 2/ Les types entiers « courts » (*char, short, enum, bitfields*) sont automatiquement convertis en *int* avant d'être pris en compte dans une expression.
    - Et respectivement pour les types non-signés:
      → *unsigned char* et *unsigned short* sont convertis en *unsigned int*.
- A la différence du *cast*, les promotions de types sont *implicites*.
- Les règles complètes sont variées et subtiles.
  - *Cf. guide de référence du C ou le Web.*
- Exemples

```
float y = 1.0/2 ;      // → 0.5F (conversion int→double, puis double→float)
double y = 1.0/2 ;     // → 0.5 (double)
double z = 1/2.0 ;     // → 0.5 (double)
float x = 1/2 ;        // → 0.0F !!!!
```

Les caractères en C

- Les caractères
  - Peuvent être spécifiés littéralement par le caractère *entre guillemets*:
    - Exemple: ‘a‘ est la constante de type *char* représentant le *a*.
  - Le format de codage est l’ASCII ()
    - Tous les caractères US ont un code entre 0x00 et 0xFF.
    - Il n’y a pas d’*Unicode* en C !
    - *http://www.commentcamarche.net/contents/base/ascii.php3*

- Exemples
  - Lettres et chiffres: ‘a’, ‘b’, …’z’, ‘A’, ‘B’, …’Z’, ‘0’, ‘1’, …‘9’
  - Symboles: ‘&’, ‘#’, ‘(‘,’[‘, ‘^’, ‘’, …
  - Ponctuation: ‘!’, ‘?’, ‘:’, ‘;’, …
  - Caractères spéciaux
    - → un backslash, suivi d’un caractère, le tout entre apostrophes
      - Le guillemet: ‘\"’
      - L'apostrophe: ‘\’’
      - Le passage à la ligne: ‘\n’
      - La tabulation: ‘\t’
      - le caractère nul: ‘\0’
        - » Attention: le caractère nul ‘\0’ n’est pas le caractère zéro ‘0’ !

Les chaînes de caractères en C

Présentation simplifiée (voir cours sur tableaux)

- Une ***chaîne de caractères*** contient
  - une séquence de caractères (*char*),
  - qui se termine obligatoirement par le caractère *nul*.
    - Le compilateur le rajoute automatiquement et implicitement.

- Une chaîne de caractères *littérale* est une séquence de caractères débutant et finissant par des guillemets:
  - Exemples de chaînes de caractères *littérales*

```
… "toto"  …;                    // Donne «toto»
… ""  … ;                       // La chaîne vide
… "Le guillemet est \""  …; // Donne «Le guillemet est "»
```

Les structures de contrôle (1)

- Les tests (*if-else*)
  - *condition* est une expression dont la valeur est comparée à zéro.

- La valeur de vérité de *condition* est
  - *false* si l'expression vaut 0
  - *true* si elle vaut autre chose que 0

```
if ( condition ) {
   // code si true…
} else {
   // code si false…
}
```

```
if ( condition1 ) {
    // action 1 …
} else if ( condition2 ) {
    // action 2 …
} else {
    // action 3 …
}
```

Action3 est le bloc *else* de la condition2.

Les structures de contrôle (2)

- Les boucles *while* (« tant que »)

Le bloc peut ne pas être exécuté.

```
while ( condition ) {
  // code …
}
```

```
do {
  // code …
} while ( condition ) ;
```

Le bloc est toujours exécuté au moins une fois.

- Les boucles infinies

Il doit exister au moins une façon de sortir.

```
while ( 1 ) {
  …
  if (…)    // une condition de sortie
    break ;
  …
}
```

Les structures de contrôle (3)

- Les boucles *for*
  - *Syntaxe*

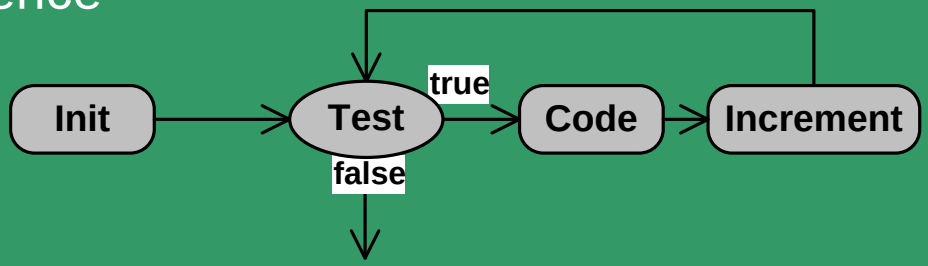
Version avec *for()*

```
for ( init_expr; test_expr; increm_expr ) {
  // code …
}
```

Version equivalente avec *while()*

```
init_expr;
while (test_expr) {
  // code …
  increm_expr ;
}
```

  - Séquence



  - Exemple
    - Calcul direct de factorielle 10 (*10!*).

```
unsigned int factorielle = 1;
for ( int i=1 ;  i<=10 ;  i++  )
   factorielle *= i ;
```

Les fonctions

- Exemples
  - ***somme*** = fonction prenant deux entiers en entrée et renvoyant un entier en sortie.
    - le type de l'expression dans le *return (ici « a+b »)* doit correspondre au type de retour déclaré pour la fonction (*int*).

```
int somme( int a, int b ) {
  return a+b;
}
```

  - ***maFonc*** = fonction prenant un flottant en entrée et ne renvoyant pas de valeur.
    - Dans un tel cas, le type de retour déclaré est *void*.
    - L'intérêt d'une telle fonction réside dans les traitements qu'elle accomplit: on les appelle les ***effets de bord*.**
    - **Recommandation**
      On essaie en général d'éviter la programmation par effets de bord, car les programmes deviennent difficiles à comprendre.

```
void maFonc( float x ) {
  …;
  return ;
}
```

Bloc de code.

- ***Définition*** de fonction
  - La *définition* d'une fonction contient le code complet de la fonction.
  - Une fonction ne peut avoir qu'une seule *définition* dans tout le programme.

Les fonctions: à quoi ça sert ?

- Factoriser une certaine logique de calcul
  - Appliquer le même calcul à des arguments différents
  - Réduire la taille du code
    - mais il y a une pénalité CPU pour *l'appel* de la fonction.
  - Débugger une seule fois le code de la fonction

- Abstraction de plus haut niveau:
  - Une fonction sert à donner un nom à un traitement particulier.
  - Une fonction peut être récursive (s'appeler elle-même)
    - *voir plus loin…*

- Meilleure modularisation du code
  - Un plus grand nombre de petits modules
  - Chaque module a une responsabilité bien isolée.

Fonction récursive

- Une ***fonction récursive*** est une fonction qui peut s'appeler elle-même.
  - Exemple: la factorielle

Factorielle.cpp

```
int  factorielle( int n )
{
   if (n==1)
      return 1;
   return  n*factorielle( n-1 ) ;
}
```

- Il faut au moins une condition d'arrêt dans la fonction.
  - Sinon, boucle infinie.
- Tous les langages ne supportent pas la récursivité
  - Ex. Pascal.

Contrôle de type

- Définition
  - Dans un appel de fonction (par exemple), le *contrôle de type* consiste à vérifier que :
    - le symbole appelé a bien été déclaré au préalable,
    - qu'il y a bien autant d'arguments que de paramètres,
    - que pour chaque paramètre, le type du paramètre attendu est *compatible* avec (ie. plus large que) le type de l'argument passé.
  - En cas d'erreur, la compilation échoue.
- Exemples

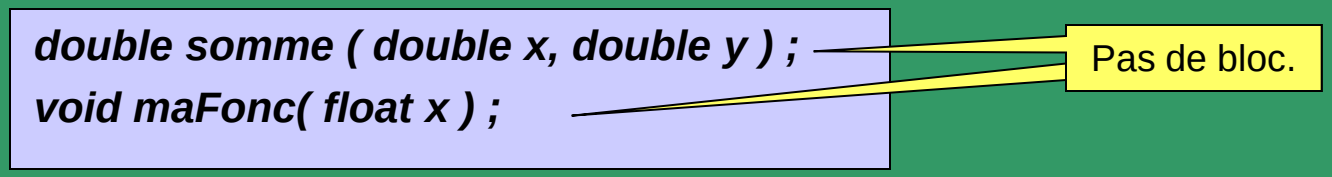
monSource.cpp

```
int  factorielle( int n )  { … }

int maFonction() {
  factorielle( 3.5 );         // KO: argument d'un type trop large
  factorielle( 3, "xyz" );    // KO: un argument de plus qu'attendu
  factrielle( 10 );           // KO: faute de frappe dans le nom de fonction
  factorielle( 10 );          // OK
  factorielle( 'a' );         // OK: conversion de type implicite
}
```

Déclaration de fonction

- Exemples de déclaration:



- ***Déclaration*** de fonction
  - Une déclaration de fonction contient uniquement la *signature* de la fonction.
    - type de retour, nom de la fonction et paramètres d'entrée (avec leur type)
  - A la différence de la ***définition***, une ***déclaration*** de fonction n'est pas suivie du bloc de code de la fonction:
    - La déclaration est la signature suivie d'un *point-virgule* (;)
  - Une même fonction peut être déclarée un nombre arbitraire de fois dans un même source, et dans un programme.

Déclaration de fonction
à quoi ça sert ?

- Une déclaration de fonction indique:
  - Qu'il existe *quelque part* une fonction portant ce *nom*
    - écrite par le programmeur lui-même ou bien fournie par une librairie
  - qui prend des paramètres d'entrée de tel et tel *type*
  - qui renvoie une valeur de tel *type*…

- Une déclaration permet donc de pouvoir réaliser dès la compilation, pour chaque appel de fonction, un *contrôle de type complet* :
  - Vérifier la cohérence entre
    - les *types* attendus par la fonction et
    - les *types* des arguments effectivement passés par l'appelant.
  - Prévenir le programmeur d'éventuelles perte d'information dans la récupération de la valeur de retour.
    - ie. si le *type* d'accueil est « plus petit » que le *type* de retour déclaré

- Ce contrôle de type complet est possible même si la *définition* de la fonction (ie. son *code*) n'est *pas disponible*.
  - Cas d'un éditeur de logiciel qui voudrait protéger son savoir-faire
  - Ce mécanisme offre un gain de temps énorme en compilation (cf. cours *modularisation*)

Déclaration de fonction

Recommandations pratiques

- Toute fonction doit être déclarée avant son utilisation.
  - Si la fonction n'a pas été déclarée avant d'être appelée, le compilateur renvoie une erreur parce qu'on lui demande de prendre ne compte un ***symbole*** qu'il ne connait pas.

Surcharge des noms de fonction *en C*

- Attention
  - En C, on ne peut pas définir deux fonctions *de même nom* qui diffèreraient
    - uniquement par le type de retour, ou
    - uniquement par la liste des arguments.
- Exemples

monSource.cpp

```
int  maFonction( int n ) { … }

// KO: valeur de retour d'un type différent
double maFonction( int n ) { … }

// KO: listes d'arguments différentes
double maFonction( double d ) { … }
double maFonction( int n, char c ) { … }
```

Les entrées/sorties simples en *C*

- Sortie à l'écran
  - fonction *printf()*, déclarée dans *stdio.h*.
- Affichage d'une chaine de caractères littérale

```
printf( "J'irais bien à la plage\n" );
```

\n représente le caractère de passage à la ligne

- Version avec paramètres:
  - chaîne de format,
  - puis, les paramètres (en nombre variable).

```
int monInt = 3;
float monFloat = 0.95f ;
printf( "Le flottant numero %d est %f\n", monInt, monFloat ):
```

Donne à l'écran

```
Le flottant numero 3 est 0.95
```

Entrées/sorties: codes de formatage

Types numériques	
%d	signed int
%u	unsigned int
%ld	signed long
%lu	unsigned long
%f	float
%e	float (scientific)
%lf	double
%le	double (scientific)

Autres types scalaires	
%c	caractère seul
%s	chaîne de caractères
%x	hexa (tout type)
%p	pointeur

Fin