Méthodes informatiques pour physiciens

introduction à C++ et
résolution de problèmes de physique par ordinateur

Leçon # 3 : Les fonctions

Alessandro Bravar Alessandro.Bravar@unige.ch
tél.: 96210 bureau: EP 206

assistants

Johanna Gramling Johanna.Gramling@unige.ch
tél.: 96368 bureau: EP 202A

Mark Rayner Mark.Rayner@unige.ch
tél.: 96263 bureau: EP 219

http://dpnc.unige.ch/~bravar/C++2015/L3
pour les notes du cours, les exemples, les corrigés, …

Plan du jour #3

Récapitulatif et corrigé de la leçon #2

Structure du programme

texte Micheloud et Rieder chap. 7

Notion de fonction et ses différents éléments

Déclaration et définition des fonctions

Passage de paramètres à une fonction, valeur renvoyée

Références

Passage de paramètres à une fonction par référence

Surcharge de fonctions

Récursivité

Compilation séparée

Directives des prétraitement (les #...)

Récapitulatif de la leçon #2

Exécutions conditionnelles et branchements

`if`
`if . . . else`

Opérateurs relationnels

`<, <=, >, >=, ==, !=`

Opérateurs logiques

`&&, ||, !`

Itérations et boucles

`while`
`do . . . while`
`for`

Sauts

`break`
`continue`

Structure du programme

Mots clés : environ 60 *instructions* C++
nous en avons déjà vu la moitié et une dizaine en détail
(pour déclarer le type des données et contrôler le déroulement du programme)

Données et variables : types natives : **`int`**, **`double`**, **`char`**, **`bool`**, …
types dérivés : tableaux, références, pointeurs

Instructions : `int a = 5;` `cout << ...;` `return;`

Expressions : `a + b, a < b, a < 0 && b > 5, ...,`

Structures de contrôle : conditions : **`if`**, **`else`**, …
boucles : **`for`**, **`while`**, **`break`**, …

Programmation structurée : décomposition du problème en plusieurs parties plus simples (sous-programmes) ou fonctions,
utilisation des bibliothèques C++ et
des bibliothèques *externes*

Programmation Orientée Objet : abstraction du problème avec des structures autonomes contenant ses propres données et les fonctions qui agissent sur ces données, i.e. la classe

Fonctions

Les programmes sont beaucoup plus volumineux que ceux que nous avons étudiés jusqu' ici. Pour en faciliter et en améliorer la gestion, les problèmes (programmes) complexes sont décomposés en sous-programmes plus simples appelés fonctions.

Une fonction est un sous-programme qui agit sur des données et retourne une valeur.
Ces sous-programmes peuvent être développés, compilés, testés, etc. séparément.
En particulier, ils peuvent être réutilisés plusieurs fois dans le même programme ou dans d'autres programmes (voir p.ex. la fonction `sin`).
Une fonction doit réaliser une action bien définie et compréhensible.

Tout programme C++ comprend au moins une fonction: la fonction principale **`main`**(), qui s'exécute automatiquement (elle est appelée par le système d'exploitation) et qui peut en appeler d'autres.

Chaque fonction doit être déclaré avant sont utilisation, un peu comme les variables.
Chaque fonction porte un nom qui l'identifie dans le programme et est accessible partout dans le programme.
Un fois un nom choisi, ce nom est associé pour toujours à la fonction et il ne peut plus être utilisé pour d'autre fonctions ou variables.
Le nom des fonctions suivit les même règles que les nomes des variables.

Il existe deux catégories de fonctions: les fonctions prédéfinies qui sont intégrées au compilateur C++ et les fonctions définies par l'utilisateur (ou une autre personne).

Exemple utilisant une fonction

Ce programme utilise une fonction pour calculer le volume d'une sphère.

```
#include <iostream>
#include <cmath>      //bibliotheque mathematique

using namespace std;

//declaration de la fonction volume
double volume(double x);

int main() {
  double rayon;
  cout << "Entrez le rayon de la sphere : ";
  cin >> rayon;

//calcul du volume avec la fonction volume
  double vol = volume(rayon);
  cout << "Le volume est: " << vol << endl;

  return 0;
}

//corps de la fonction volume
double volume(double rayon) {
  double x = 4./3. * M_PI * pow(rayon,3.);
  //on peut aussi ecrire return 4./3. * M_PI * pow(rayon,3.);
  return x;
}
```

voir Sphere.cpp

Fonctions (2)

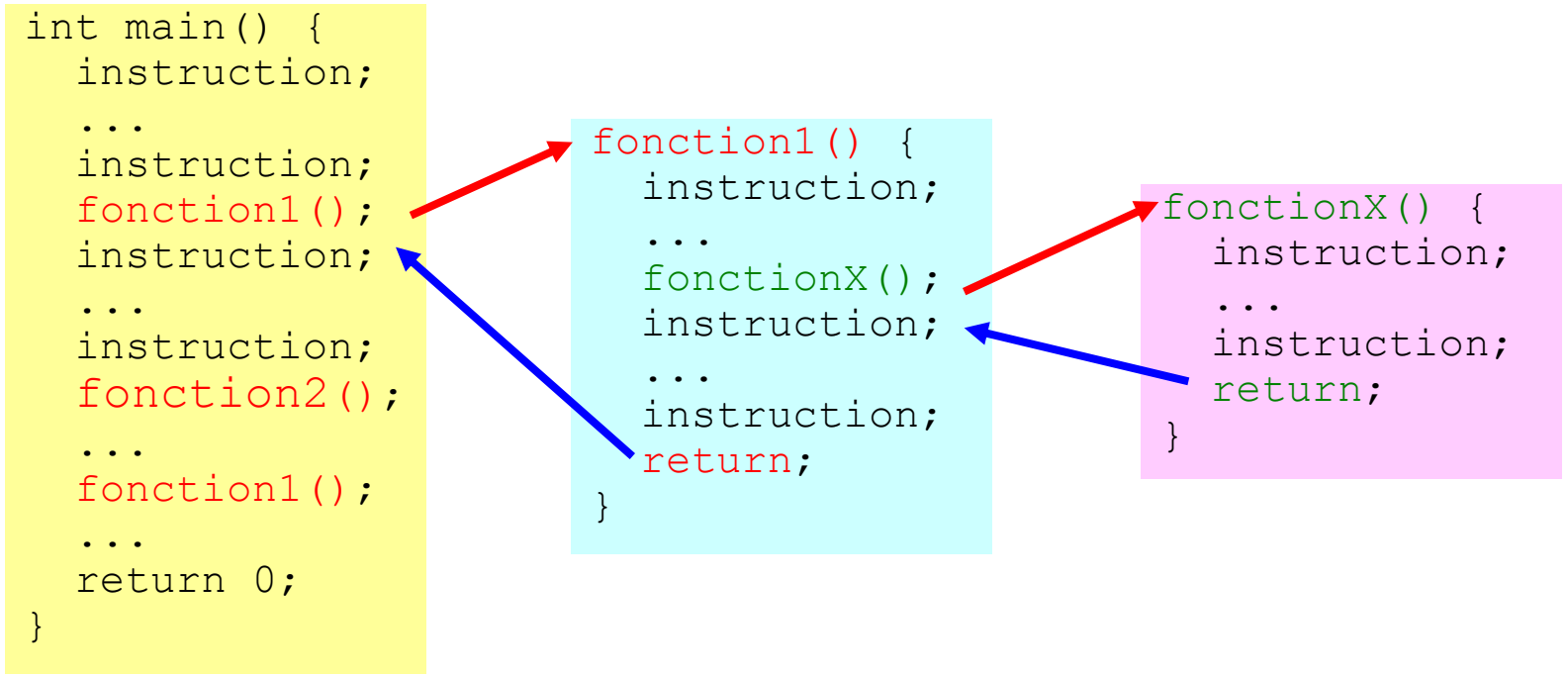
Quand le programme rencontre un appel à une fonction :

1) il se place directement au début de la fonction,

2) exécute les instructions dans cette fonction,

3) après l'exécution, le programme revient à l'instruction qui suit l'appel de la fonction.

Une fonction peut en appeler des autres.



Fonctions prédéfinies

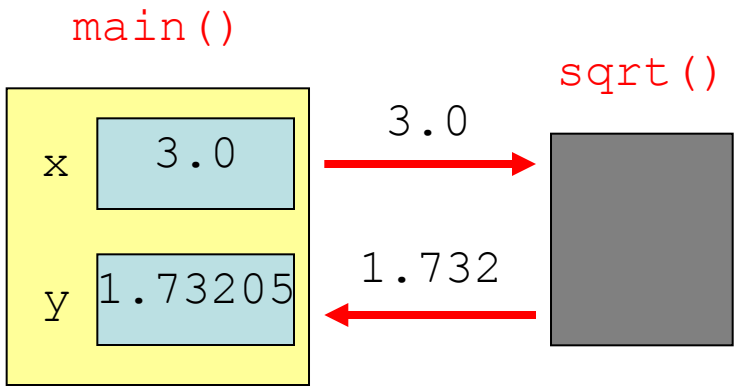
La *bibliothèque C++ standard* constitue un ensemble de fonctions prédéfinies auxquelles on accède via des fichiers d'en-tête (pour connaître toutes les fonctions de C++, il faut consulter un manuel C++).

Nous avons déjà utilisé les fonctions mathématiques définies dans `<cmath>`.
p. ex. `sqrt()`, la fonction racine carrée :

```
#include <cmath>  //definition de la fonction sqrt
int main() {
   double x = 3.;    double y;
   y = sqrt(x);      //appel de la fonction sqrt
   cout << y;
   return 0;
}
```

voir RacineCarree.cpp

Les variables `x` et `y` sont déclarées dans `main()`.
La valeur de `x` (`3.`) est passée à la fonction `sqrt()`, qui renvoie la valeur `1.73205` à `main()` où elle est affectée à la variable `y`.
La boîte qui représente la fonction `sqrt()` apparaît en grisé, parce que son fonctionnement interne est *masqué*.



Déclaration des fonctions

Pour être exploitable dans le programme, une fonction doit être d'abord déclarée puis définie. Toute fonction doit être déclarée avant son utilisation, comme avec les variables. La déclaration doit apparaître en dehors des autres fonctions (y compris la fonction `main`)

La déclaration d'une fonction est désignée sous le nom de prototype de la fonction.
La déclaration se compose du nom de la fonction, du type de la valeur renvoyée par la fonction et d'une liste de paramètres passée à la fonction :

```
type_renvoyé nom_de_la_fonction (liste de paramètres);
```

p. ex. :

```
double volume(double cotea, double coteb, double cotec);
```

Une fonction renvoie toujours une valeur ou résultat (`int`, `double`, …). S'il n'y a pas de valeur renvoyée, le type renvoyé est le type `void`. Dans la liste de paramètres (ou signature de la fonction) on doit toujours définir le type des données passées à la fonction. S'il n'y a pas de paramètres, la liste est vide mais les parenthèses ( ) sont toujours présentes. La déclaration se termine avec un `;` .

Les noms des variables dans la liste de paramètres ne sont pas obligatoires ; des noms bien choisis peuvent aider à comprendre ou deviner le fonctionnement de la fonction, p. ex. :

```
double volumeCylindre(double, double);
```

ou

```
double volumeCylindre(double rayon, double hauteur);
```

Définition des fonctions

La définition d'une fonction est la description de celle-ci (corps de la fonction).
Le corps est un bloc d'instructions délimitées par des accolades `{ }` .

Le prototype de la fonction et sa définition doivent correspondre, c.-à-d. le même type de la valeur renvoyée et la même liste de paramètres (même nombre et types).

```
type_renvoyé nom_de_la_fonction (liste de paramètres) {
  corps de la fonction;
  return xyz;
}
```

p. ex. :

```
double volume(double cotea, double coteb, double cotec) {
  double v;
  v = cotea * coteb * cotec;
  return v; }
```

La définition de la fonction peut apparaître n'importe où dans le programme après sa définition (en général à la fin du programme) mais en dehors d'autres fonctions, y comprise la fonction `main` (en C++ il n'y a pas d'imbrication de fonctions).

Les variables répertoriées dans la liste des paramètres de la fonction sont appelées arguments. Dans ces exemples, les arguments sont passés par valeur. Cela signifie que leurs valeurs sont affectées aux paramètres correspondant de la fonction. Des variables déclarées dans une fonction sont locales à cette fonction : elles n'existent que pendant l'exécution de la fonction et leur portée est limitée à la fonction.

Déclaration et Définition

```
double volume(double);
```
déclaration

```
int main() {
  double cote;
  cin >> cote;
  cout << volume(cote);
  return 0;
}
```
définition

```
double volume(double cote) {
  double x = cote * cote * cote;
  return x;
}
```

On peut aussi définir la fonction lors de la déclaration.

```
double volume(double cote) {
  double x = cote * cote * cote;
  return x;
}
```
déclaration et définition

```
int main() {
  double cote;
  cin >> cote;
  cout << volume(cote);
  return 0;
}
```

A noter que à la fin d'une déclaration il y a toujours le `;` , tandis que il n'y a pas de `;` à la fin du corps de la fonction. La fonction peut aussi être défini lors de sa déclaration.

Le déclarations sont d'habitude situées dans les fichiers d'en-têtes avec d'autres déclarations, constantes numériques, etc. Les définitions sont contenues dans des fichiers précompilés (fichiers objets). Pendant la compilation, l'éditeur de liens va chercher les définitions de fonctions dans ces fichiers.
P. ex. les fonctions mathématiques de C++ sont déclarées dans `<cmath>` et définies dans `libmath.a` . On utilise la prédirective de compilation `#include` pour inclure ces fichiers d'en-tête, p.ex. `#include <cmath>` .

Exemple : passage par valeur

Ce programme échange deux valeurs dans la fonction swap.

```
#include <iostream>

using namespace std;

void swap(int n1, int n2);

int main() {
  int a = 10;
  int b = 20;
  cout << "avant swap : " << a << "  " << b << endl;
  swap(a, b);
  cout << "apres swap : " << a << "  " << b << endl;
  return 0;
}

void swap(int n1, int n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
  cout << "dans swap : " << n1 << "  " << n2 << endl;
  return;
}
```

a = 10 et b = 20 !
c'est correct ?

la fonction ne renvoie aucune valeur, elle est de type `void` et `return` n'est pas suivi par une valeur à renvoyer (dans ce cas, on peut omettre `return`)

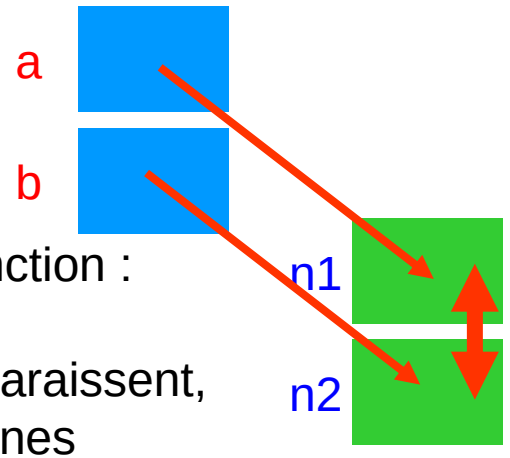
voir Swap.cpp

Exemple : passage par valeur

Ce programme échange deux valeurs dans la fonction swap.

```
#include <iostream>

using namespace std;

void swap(int n1, int n2);

int main() {
  int a = 10;
  int b = 20;
  cout << "avant swap : " << a << "  " << b << endl;
  swap(a, b);
  cout << "apres swap : " << a << "  " << b << endl;
  return 0;
}

void swap(int n1, int n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
  cout << "dans swap : " << n1 << "  " << n2 << endl;
  return;
}
```

On passe des valeurs à la fonction (pas les variables). Les arguments passés à la fonction sont locaux à la fonction : une fois sorti de la fonction, les arguments n1 et n2 disparaissent, donc on retrouve les anciennes valeurs de a et de b.



a = 10 et b = 20 ! c'est correct ?

la fonction ne renvoie aucune valeur, elle est de type `void` et `return` n'est pas suivi par une valeur à renvoyer (dans ce cas, on peut omettre `return`)

voir Swap.cpp

Passage par valeur

Les fonctions ont à la base 2 limitations :

les arguments sont passés par valeur (non modifiables) et
l'instruction return ne peut retourner qu'une seule valeur.

```
int fonct(int var);
```

Dans l'appel de `fonct` , le paramètre `var` est passé par valeur, c.-à-d. que on passe à la fonction la valeur de la variable `var`, mais pas la variable `var`. On peut effectuer des calcul avec la valeur de la variable `var`, mais on ne peut pas modifier la variable `var`, i.e. la valeur enregistrée dans la location mémoire de `var`.
`var` est donc un paramètre en lecture seule (read only).

Pendant l'appel de la fonction, une copie locale du même type de `var` est créée dans la fonction et la valeur de `var` est enregistré dans cette variable locale, un peu comme si on affectait la valeur de `var` à cette variable locale.

Pour changer la valeur du paramètre ou renvoyer plusieurs valeurs,
on utilise le passage des variables par référence à la fonction

```
int fonct(int &var);
```

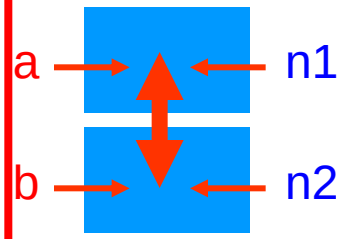
Pour passer de paramètres par référence il faut modifier seulement la liste de paramètres de la fonction (définition et déclaration) en ajoutant une esperluette `&` comme ci-dessus.

Le corps de la fonction ne change pas et l'appel à la fonction non plus.

Exemple: passage par référence

Ce programme échange aussi deux valeurs dans la fonction `swap`, mais le variables sont passées à la fonction `swap` par référence.

```
#include <iostream>

using namespace std;

void swap(int &n1, int &n2);

int main() {
  int a = 10;
  int b = 20;
  cout << "avant swap : " << a << "  " << b << endl;
  swap(a, b);
  cout << "apres swap : " << a << "  " << b << endl;
  return 0;
}

void swap(int &n1, int &n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
  cout << "dans swap : " << n1 << "  " << n2 << endl;
  return; }
```

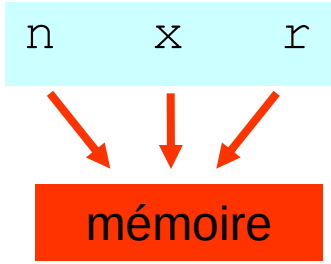


a = 20 et b = 10 ! c'est ça que vous vous attandez ?

voir Swap_ref.cpp et comparez avec Swap.cpp

Références

Une référence est un synonyme (*alias*) d'une autre variable.
On peut faire référence à la même variable (même location mémoire) avec plusieurs identificateurs avec des noms différents : ils doivent être du même type que la variable de départ et ils auront toujours la même valeur (parce que ils accèdent à la même location mémoire !).
On dit que le références sont de types dérivés.



La référence est déclarée en ajoutant une esperluette `&(ampersand)` au nome du nouveau identificateur. La liaison entre la variable et le nouveau identificateur doit être établi au moment de la déclaration de la référence.

voir Reference.cpp

```
int n = 33;
int &r = n;    //r est une reference a n
cout << n << " " << r << endl;  //memes valeurs
--n;
cout << n << " "<< r << endl;
r *= 2;
cout << n << " " << r << endl;
cout << &n << " " << &r << endl;    //adresse memoire
```

Les deux identificateurs `n` et `r` font référence à la même variable. Ils sont des noms symboliques pour le même emplacement mémoire, donc ils ont la même valeur.
Comme une variable `const`, une référence doit être initialisée dès sa déclaration.

Passage par référence

Pour remédier aux limitations du passage par valeur, on peut forcer une liaison entre les variables passés à la fonction et les variables locales de la fonction réservées pour recevoir ces valeurs (référence). Dans ce cas, la fonction agit directement sur les variables (même location mémoire) et pas sur leurs valeurs. Ube copie locale n'est pas crée. Le paramètre passé par référence est en mode lecture-écriture (read-write). Toutes les modifications de la variable dans la fonction seront effectuées sur la location mémoire de la variable en question, donc on change aussi sa valeur de manière définitive.

Pour passer des arguments par référence, on ajoute une esperluette (`&`) entre le type et le nom de la variable ou variables dans la déclaration et la définition de la fonction.

p. ex. : une fonction qui calcule les racines du polynôme $ax^2 + bx + c = 0$

déclaration de la fonction `racine2` :

voir RacinesPolynome_v2.cpp

```
int racine2(double a, double b, double c, double &x1, double &x2);
```

appel de la fonction `racine2` dans main (pas de `&` ici !)

```
double a = 5., b = 8., c = -3.;
double x1, x2;
racine2 (a, b, c, x1, x2);
```

les arguments a, b, c, sont passés par valeur, donc pas modifiables les arguments x1, x2 sont passés par référence, donc on peut les modifier dans la fonction, i.e. Affecter des nouvelles valeurs à x1 x2.

définition de la fonction `racine2` :

(mais le corps ne change pas)

```
int racine2(double a, double b, double c, double &x1, double &x2) {...}
```

Quand utiliser les références ?

Il est préférable de passer un paramètre par référence dans les trois situations suivantes :

1. Si la fonction doit changer les valeurs des paramètres passés, comme dans le cas de `swap()`, les paramètres doivent être passés par référence.

2. Si le paramètre occupe beaucoup d'espace mémoire (p.ex. un long tableau), vous le passerez par référence pour éviter sa duplication. Cela permet aussi à la fonction de changer la valeur du paramètre.

3. Si la fonction doit renvoyer plusieurs valeurs.

Si vous ne désirez pas que la fonction puisse changer le contenu du paramètre, vous pouvez utiliser le passage par référence constante. Ce dernier fonctionne comme le passage par référence, mais la fonction ne peut pas changer la valeur du paramètre(s).

```
void func(int x, int &y, const int &z)
```

* Arguments par défaut

En général, quand on appelle une fonction, il faut passer tous les arguments à la fonction comme spécifié dans la déclaration de la fonction.
Le nombre d'arguments d'une fonction peut varier au moment de l'exécution, si on fournit des valeurs par défaut aux arguments de la fonction pendant la déclaration de la fonction. Si on ne spécifie pas la valeur des ces arguments, la valeur par défaut sera utilisée. Par contre la définition de la fonction ne change pas (elle ne sait pas si on a utilisé des valeurs par défaut ou non).

Dans l'exemple suivant, 3 parmi le 5 paramètres sont initialisés pendant la déclaration de la fonction ; cette fonction peut être appelée avec 2, 3, 4 ou 5 arguments :

```
void func(int a, int b, int c=4, int d=7, int e=3);
```

Tous les arguments facultatifs doivent être listés en dernier.

voir Polynome.cpp et VolumeCylindre.cpp

Ensuit, on peut appeler la fonction `func` de différentes façons :

`x = func(1,2);` dans ce cas, les valeurs par défaut seront attribuées aux variables c, d et e soit a = 1, b = 2, c = 4, d = 7, e = 3

`x = func(1,2,3);` dans ce cas, les valeurs par défaut seront attribuées aux variables d et e soit a = 1, b = 2, c = 3, d = 7, e = 3

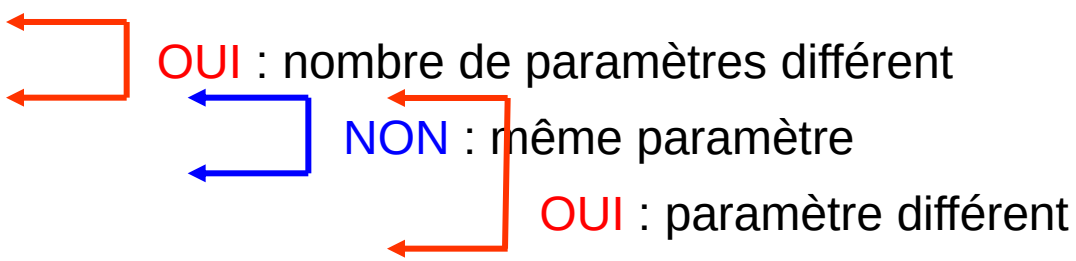
`x = func(1,2,3,4,5);` aucune valeur par défaut ne sera utilisé (tous les arguments ont été spécifiés)

Surcharge de fonctions

Parfois on voudrait utiliser le même nom pour des fonctions qui font le même traitement sur des paramètres de type différents ou nombre de paramètres différents. Cette fonctionnalité est la surcharge de fonctions.

Une fonction surchargée doit différer au niveau de la liste des paramètres, c.-à-d. de leur type et/ou de leur nombre. Le type de la valeur renvoyée peut être identique ou non pour une fonction surchargée, mais modifier seulement le type renvoyé ne suffit pas à surcharger une fonction. Pour modifier le type renvoyé, on doit également modifier la signature (type et/ou nombre de paramètres).

```
int fonction(int, int);
int fonction(int);
double fonction(int);
double fonction(double);
```



Entre deux fonctions surchargées, le compilateur déterminera celle qui doit être appelée en fonction du type et/ou du nombre de paramètres passés à la fonction.
P. ex. on peut utiliser le même nom pour ces trois fonctions différentes :

```
int cubeInt(int);
float cubeFloat(float);
double cubeDouble(double);
```

→

```
int cube(int);
float cube(float);
double cube(double);
```

voir Surcharge.cpp

* Récursivité

Il est possible d'utiliser la valeur renvoyée par une fonction comme paramètre d'une autre fonction (il ne s'agit pas d'imbrication de fonctions !),

p. ex. `resultat = triple(carre(cube(nombre)))`

En C++ une fonction peut aussi appeler soi-même. Cette fonctionnalité s'appelle la récursivité. P. ex. si les données de départ doivent être traitées de la même façon que les données d'arrivée, on peut utiliser la récursivité.
Bien que « élégante » l'utilisation de la récursivité peut ralentir l'exécution du programme et le faire tourner en boucle. Dans ce cas, mieux utiliser une boucle.

P. ex. : pour calculer la factorielle de `n`, `n!` on peut procéder de deux façons différentes :

par itération: `n! = 1 · 2 · 3 · ... · n`

par récursivité:

```
0! = 1
n! = n · (n-1)!
```

Calculez la factorielle de `n` avec une boucle `for` (itération)
et par récursivité avec une fonction !

La fonction `fact` appel soi-même tant que `n > 0`. A chaque nouveau appel on « décrément » `n` de `1`.

```
int fact(int n) {
  if (n==0) return 1;
  else return (n*fact(n-1));
}
```

voir Factorielle_rec.cpp

* Fonctions en ligne

Chaque appel à une fonction implique plusieurs opérations et peut ralentir l'exécution du programme (stockage des variables locales et courantes, stockage de l'emplacement où revenir dans le programme principal après le retour de la fonction, etc.).
On peut éviter tout cela en déclarant la fonction comme `inline`. Cette opération indique au compilateur de remplacer explicitement chaque appel de la fonction par le corps de la fonction. En réalité, c'est le compilateur qui décide si étendre la fonction ou non (optimisation du code). En général, seules les fonctions très courtes seront compilées en ligne.

P. ex.

```
inline int cube(int x) {
  return x*x*x;
}

int main() {
  int n = 3;
  cout << cube(n) << endl;
  return 0;
}
```

sera compilé comme s'il s'agissait de

```
int main() {
  int n = 3;
  cout << n*n*n << endl;
  return 0;
}
```

Mais il ne faut pas exagérer, car le programme peut devenir très long (chaque appel à une fonction est remplacé par le corps de cette fonction) s'il y a beaucoup d'appels à la même fonction.

* Compilation séparée

Un programme consiste d'une unité principale, la fonction `main`, et de plusieurs fonctions et classes. Pour faciliter et améliorer la gestion, la portabilité, la réutilisation, etc. du programme on décompose le programme en plusieurs fichier :

- un fichier pour la fonction main `main.cpp`

- un fichiers pour chaque fonction (ou classe) contenant la définition de la fonction (selon les taches accomplis plusieurs fonctions sont regroupées dans le même fichier), p.ex. `nomfonction.cpp`

- un fichier en-tête contenant la déclaration (ou déclaration) de la fonction, des constantes, etc., p.ex. `nomfonction.h`

Pendant la première phase de la (pré-)compilation, le compilateur inclut les fichier en-tête (`#include`) dans les fichier `.cpp` .
Puis toutes les fichier `.cpp` du projet sont compilés; la compilation génère des fichiers objet `.o` pour chaque fichier `.cpp` .

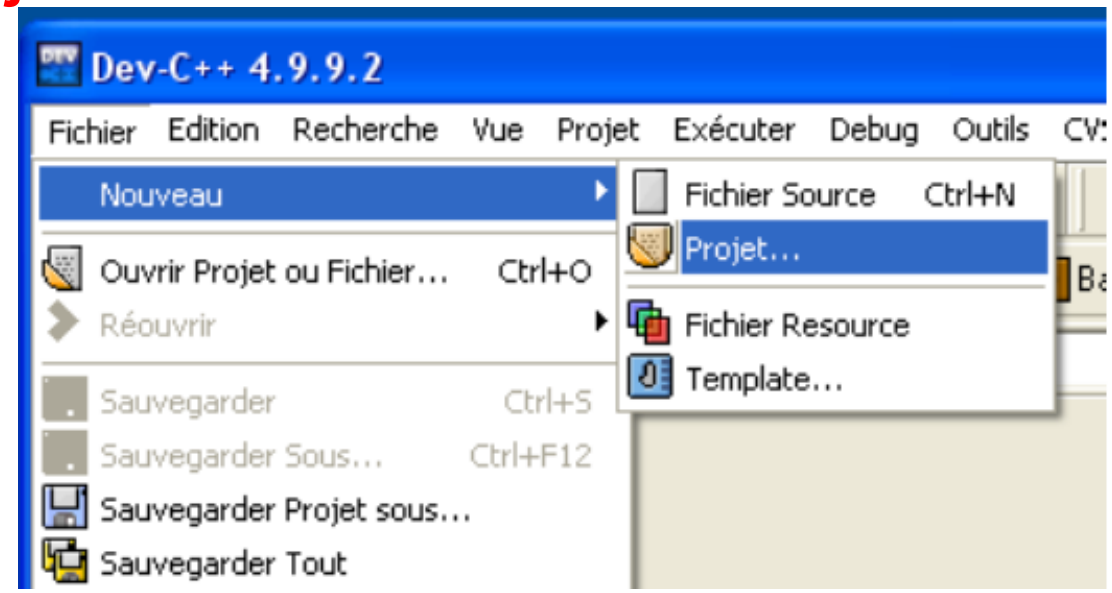
A la fin, l'éditeur de liens génère les liaisons entre les différentes fonctions (fichiers) et génère le fichier exécutable `.exe` .

Pour gérer la compilation d'un programme composé par plusieurs fichiers on utilise des scripts `Makefile` (on pourrait bien taper toutes les instructions sur le terminal).

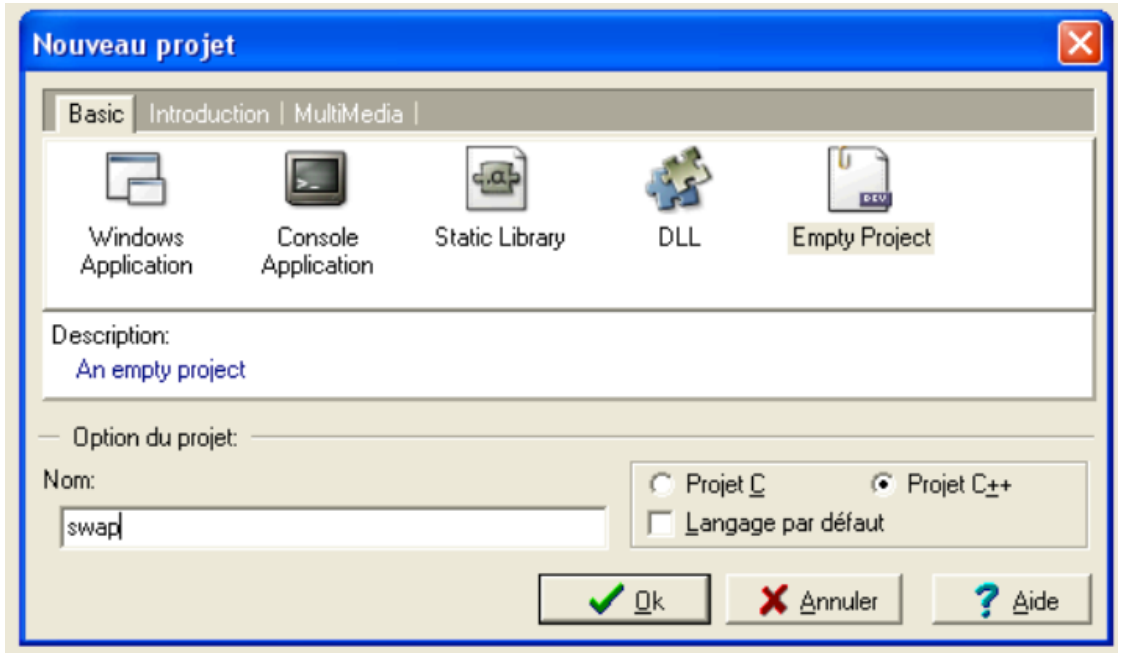
Le Dev-CPP inclut cette fonctionnalité sous forme de projet qui génère la script Makefile (`Makefile.win`) et compile le programme.

* Création d'un projet avec Dev-CPP

1. Lancez Dev-CPP et choisissez Nouveau Projet .

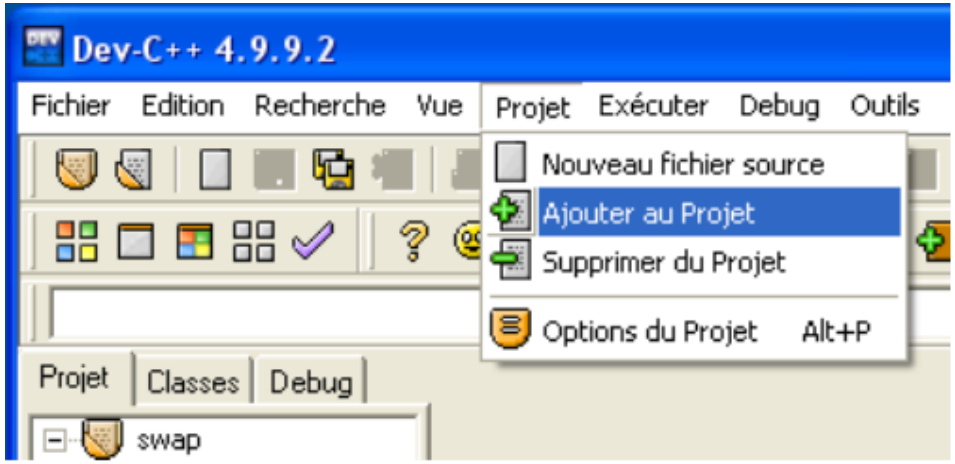


2. Choisissez un projet vide (Empty Project)
et sauvegardez-le
(choisissez un nom pour le projet).

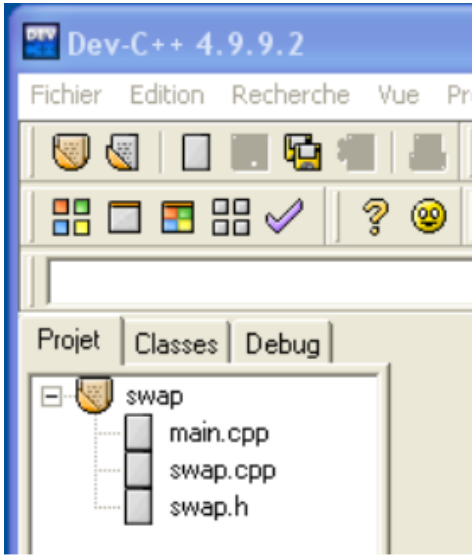


Un fichier `.dev` sera crée.
Le fichier `.dev` contient toutes les informations nécessaires pour gérer le projet.

3. Maintenant il faut ajouter les fichiers au projet.
On commence par le module principal `main.cpp` et on ajout un fichier pour chaque fonction (`.cpp`) et les fichier en-tête (`.h`)



P.ex. le projet `swap`



4. Compilez-le et exécutez le programme

voir swap, surcharge, fibonacci, pascal

* Directives de prétraitement

Le compilateur C/C++ contient un préprocesseur capable d'inclusion de fichiers, de compilation conditionnelle et de substitution de «macros». Les directives de prétraitement commencent par le symbole #. Ce sont des outils permettant d'avoir une meilleure vue d'ensemble du programme ou de grands projets.
Il ne s'agit pas des programmation C++, mais de manipulation du texte.

```
#include <nom du fichier>
```

Permet d'inclure le contenu entier du fichier spécifié; on utilise cette directive avec le fichier en-têtes (header file) contenant des déclaration de classes, fonctions, etc.

```
#define identificateur symbole
```

Cette directive provoque le remplacement par le préprocesseur de toutes les occurrences suivantes l'`identificateur` par la séquence `symbole`.

Ces constantes sont globales par rapport au fichier.

```
#define LONGEUR 80
var = LONGEUR * 20 → var = 80 * 20
```

Il est conseillé d'utiliser `const` au lieu de `#define` pour déclarer les constantes du programme, car les noms ne sont pas typés et ne suivent pas les règles de portée.

```
#if expressionConstante
  . . .
#endif
```

ou

```
#if expression Constante1
  . . .
#elif expressionConstante2
  . . .
#else
  . . .
#endif
```

Si l'`expressionConstante` est vrai (donc différente de 0) les instructions ou directives placées dans cette structure seront compilés.

```
#ifdef identificateur
  . . .
#endif
```

ou

```
#ifndef idenficateur
   . . .
#endif
```

Si l'identificateur est vrai, il a bien été défini avec `#define` (cas `#ifdef`) ou n'a pas été défini (cas `#ifndef`) les instructions ou directives seront compilés.

En général toutes le fichers en-tête sont construits sur ce modèle pour éviter plusieurs inclusions d'un même fichier.

Mots-clés du langage C++

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual
const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

mots-clés vus pendant la première leçon

mots-clés rencontrés la semaine passée

mots-clés rencontrés aujourd'hui

Résumé

Ce qu'il faut retenir / savoir faire à la fin de cette leçon :

- Comment créer et utiliser des fonctions
  - déclaration d'une fonction
  - définition d'une fonction
  - appel d'une fonction
- Passage de paramètres aux fonctions par valeur et référence
- Les références (`&`)

Mémorisez le programme suivant !
Il contient tous les élément importants vus aujourd'hui dans l'utilisation d'une fonction.

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

//prototype de la fonction racine2
int racine2(double a, double b, double c, double &x1, double &x2);

int main() {
  double a = 0., b = 0., c = 0.;
  cout << "Quelle sont les coefficients de l'equation ?\n";
  cin >> a; cin >> b; cin >> c;
  double x1, x2;
  int flag = racine2(a, b, c, x1, x2);   //calcul des racines
  if (flag < 0)
    cout << "L'equation n'a pas des solutions reelles ! " << endl;
  else if (flag == 0)
    cout << "Les solutions sont x1 = " << x1 << " et x2 = " << x2 << endl;
  else if (flag == 1) {
    cout << "a est nul, il s'agit d'une equation de 1ere degree !\n";
    cout << "La solution est " << -c/b << endl; }
  return 0;}

//definition de la fonction racine2
int racine2(double a, double b, double c, double &x1, double &x2) {
  if (a == 0.) return 1;
  double delta = b*b - 4.*a*c;
  if (delta < 0.) return -1;
  x1 = (-b - sqrt(delta) ) / (2.*a);
  x2 = (-b + sqrt(delta) ) / (2.*a);
  return 0;}
```

voir RacinesPolynome_v2.cpp

Exercices – série 3

Questions

1. Quelle est la différence entre la déclaration et la définition d'une fonction ?

2. Quelle est la différence entre le passage par valeur et le passage par référence ?

3. Quel peut être l'avantage d'un passage par référence constante ?

4. Qu'entend-on par « paramètre en lecture seule / en lecture-écriture » ?

5. Qu'entend-on par « récursivité » ?

6. A quoi servent les directives des prétraitement ?

Trouvez l'erreur !

Quelles sont les erreurs dans les programmes suivants ?

1.
```
double maFonction(double x);
int main() {
  double a=10., b;
  b = maFonction(double a);
  cout << a << " " << b << endl;
  return 0;
}
void maFonction(double x) {
  return (4*x); }
```

2.
```
void maFonction(double x);
int main() {
  double a, b;
  b = maFonction(double);
  cout << a << " " << b << endl;
  return 0;
}
void maFonction(double x); {
  return x*x; }
```

Exercices

1. Calculez le $\log_{10}(x)$ avec la fonction mathématique `log10` définie dans la bibliothèque mathématique de C++.

2. Ecrivez une fonction qui renvoie le maximum de deux nombres entiers:

```
int max(int n1, int n2);
```

3. Ecrivez une fonction qui calcule la surface d'une sphère et une fonction qui calcule le volume de la même sphère (p. 6).
(pour la valeur de π utilisez M_PI défini dans `<cmath>`)

```
double surface(double rayon);
double volume(double rayon);
```

4. Ecrivez une fonction qui convertit les degrés Fahrenheit en Kelvin.

```
double fahrenToKelvin(double temp);
```

5. Ecrivez une fonction qui utilise des paramètres par défaut (voir Polynome.cpp , p. 19)

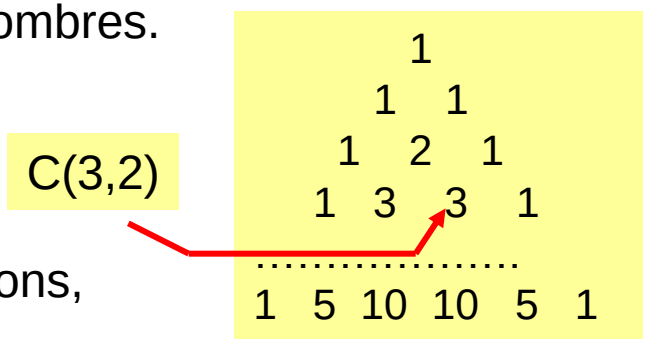
6. Développez l'exemple de surcharge à la page 20.

7. Calculez la factorielle de n par récursivité (p. 21).
   Calculez la suite de Fibonacci (1, 1, 2, 3, 5, 8, …) par récursivité :

$$F(0) = 0,\ F(1) = 1,\ F(n) = F(n-1) + F(n-2)\quad n \geq 2$$

8. Le *Triangle de Pascal* est un tableau triangulaire de nombres.
Chaque nombre représente l'une des combinaisons

```
C(n,k) = n! / k! (n-k)!
```



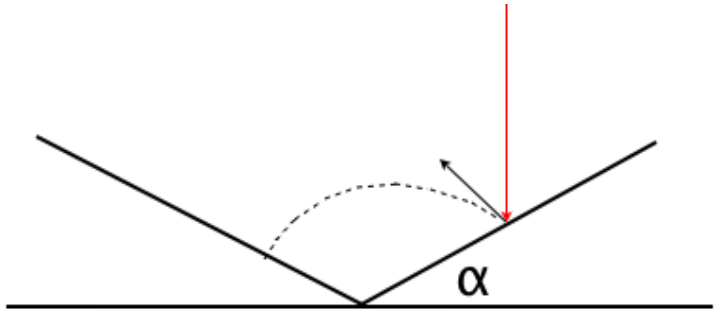
Ecrivez un programme qui utilise la fonction `fact`
pour la factorielle et la fonction `comb` pour les combinaisons,
et qui imprime le *Triangle* jusqu'à la ligne n.
Utilisez des méthodes différentes pour calculer C(n, k) (*Triangle de Pascal*),
p.ex. avec la fonction de permutation P(n, k) : C(n, k) = P(n, k) / k!

9. Développez le programme à la page 16 (définition de références).

10. Ecrivez et étudiez le programme à la page 13 (Swap.cpp).
Est-ce que la fonction fait ce que vous voulez ?
Quelle est la différence avec le programme à la page 15 (Swap_ref.cpp) ?

Problème

Plans inclinés : une balle est lâchée d'une hauteur h au-dessus d'un plan incliné.
Le dispositif est tel que la balle rebondit entre deux plans symétriques
(cf. schéma, soit $\alpha = 30^0$).
Le rebond est parfaitement élastique et la balle ne se propage que dans le potentiel gravitationnel constant ($g = 9.8\ m/s^2$).
Ecrivez un programme qui calcule la position de la balle après n rebonds.
Utilisez plusieurs fonctions pour étudier les rebonds (p. ex. pour le calcul de l'angle de rebond, le point d'intersection, etc.)
A chaque rebond affichez la position initiale et finale, la vitesse initiale et finale et le temps entre les deux rebonds. Répétez l'exercice si la balle perd une fraction constante ε de son énergie à chaque rebond.



Premier Control Continu

12 Mars 2015
10h15 – 12h15

Salle 202 Science I (ici)

Vous pouvez travailler avec vos portables !

Vous pouvez utiliser toutes les notes du cours (incl. les corrigées 2015), des textes C++, …

interdit : e-mail, téléphone, des recherches sur la toile, facebook …