Programme :

•Historique du langage
•Typologie des langages informatiques, comparaison avec Matlab, JAVA
•Mots-clés et opérateurs du C++ : les ajouts au C
•Types de base C et ajouts du C++
•Les « littéraux » en C et C++
•Le type bool en C++
•Les identifieurs (symboles) en C/C++
•Comparaison entre les notions de « définition » et de « déclaration »
•La définition des variables, leur portée, la résolution de leur portée
•Le préprocesseur C/C++, l'organisation modulaire des fichiers
•La définition et déclaration des tableaux, des constantes et des chaînes de caractères
•Les opérateurs logiques et mathématiques
•Les structures de contrôle (if, while, for, do, switch)
•Les énumérations
•La conversion des types en C, et en C++
•Fonctions et procédures en C et C++
•En C++ : fonction « en ligne », arguments par défaut et surcharge des noms de fonctions
•Les pointeurs, indirection, lien avec les tableaux, leur « arithmétique »
•Les types complexes (struct, union). Les alias de types
•Les références
•Le passage des arguments en C, en C++. Les valeurs de retour des fonctions
•Allocation dynamique de la mémoire et sa gestion

Le langage C++ : historique (1)

D'après « Langage C++, le standard ANSI/ISO expliqué, J. Charbonnel, Dunod, 1997 » :

En 1980, le langage C est le plus utilisé dans le monde. Il allie flexibilité, efficacité, disponibilité et portabilité.

Un étudiant de l'Université de Cambridge, Bjarne Stroustrup décide d'incorporer la notion de classes au C. Il s'agit d'abord d'un préprocesseur utilisé en amont du compilateur : *C with classes*. [1]

Le nouveau langage incorpore des influences positives venant de Simula, Pascal, Modula 2, Ada, Smalltalk, Algol.

En 1983, Stroustrup décide d'implémenter une version à l'aide des technologies traditionnelles des compilateurs. Il s'appelle *Cfront*, il génère du C pour obtenir un maximum de portabilité, mais ce n'est pas un préprocesseur. [2]

ONERA

Notes :

[1] : un préprocesseur génère du code source (texte) à partir du code source (texte) de l'utilisateur. Le langage C/C++ dispose d'un préprocesseur très puissant (c'est lui qui comprend les directives `#include`, `#define`, `#if`, etc.).

[2] : le processus d'obtention des exécutables est alors :

code source C++ => préprocesseur C => Cfront => compilateur C => exécutable

Le langage C++ : historique (2)

La première version officielle du langage est publiée par les laboratoires Bell Labs d'AT&T en 1986, elle est numérotée 1.0. [1]

Elle est suivie des versions 1.1 et 1.2.

La version 2.0 sort en juin 1989, elle apporte l'héritage multiple, les classes abstraites, les méthodes statiques, les méthodes constantes.

La version 2.1 corrige certains bugs. La version 3.0 ajoute la notion de templates et la version 4.0 la gestion des exceptions. [2]

A partir de cette date, le nombre d'utilisateurs explose. Plusieurs sociétés proposent des compilateurs C++. L'implémentation libre G++ de GNU apparaît.

ONERA

Notes :

[1] : Bjarne Stroustrup, The C++ programming language, 1986 – Addison-Wesley.

[2] : M.A. Ellis et Bjarne Stroustrup, The annotated C++ Reference Manual, 1990 – Addison-Wesley

Le langage C++ : historique (3)

Dès 1988, il apparaît nécessaire de normaliser le langage.

Avec l'appui de HP, AT&T, DEC et IBM, Bjarne Stroustrup s'y attelle. La première réunion du comité de normalisation a lieu en décembre 1989 à Washington.

En 1998, le travail est terminé et approuvé :

« ISO/IEC International Standard 14882 – Programming language C++, 1998 »

En 2003, une nouvelle version de la norme :

« ISO/IEC International Standard 14882 – Programming language C++, 2003 »

ONERA

Recensement des utilisateurs dans le monde :

1979 : 1

1980 : 16

1981 : 38

1983 : 85

1985 : 500

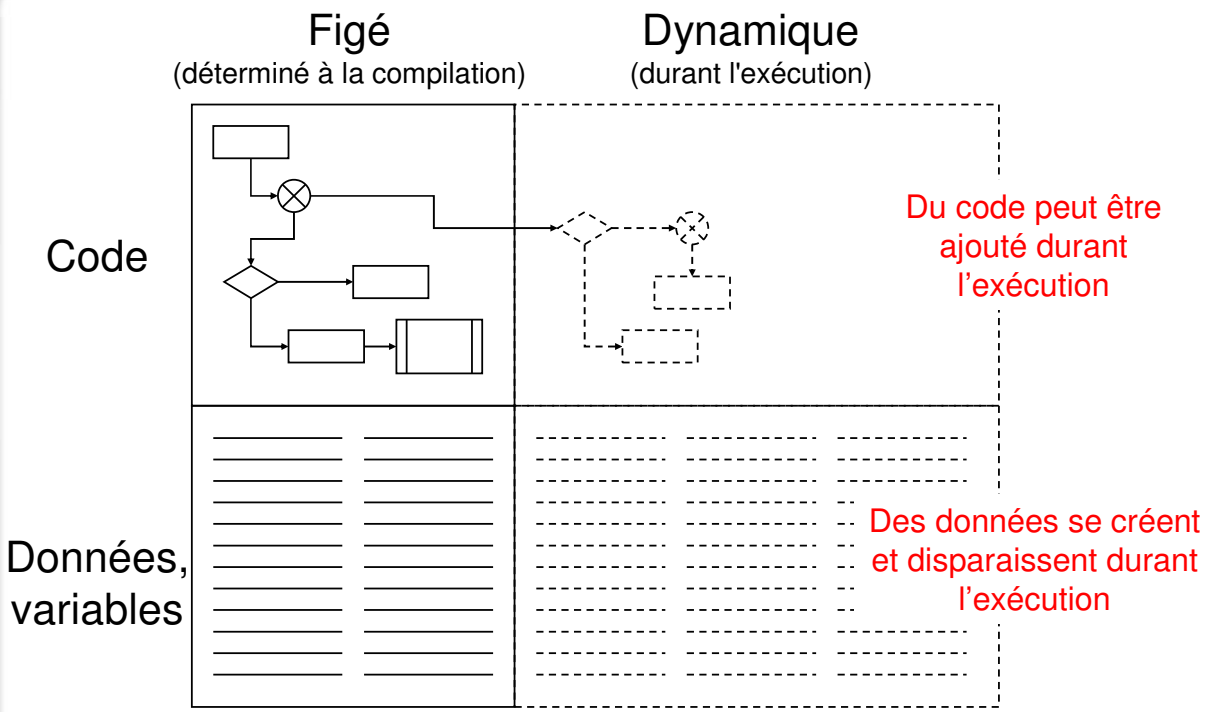
1987 : 4 000

1989 : 50 000

1991 : 400 000

Par la suite, le décompte devient impossible (en 1992, Borland fait état de 500 000 compilateurs en service).

Typologie des langages informatiques (1)



ONERA

Page 5

Explication :

La grande différence entre les langages compilés et les langages interprétés vient de la capacité ou non à étendre le code durant l'exécution du programme. Le principe d'ajouter du code durant l'exécution n'est pas difficile à comprendre : il suffit de penser à une fonction qui crée d'autres fonctions quand on l'exécute.

Typologie des langages : typage

On peut aussi distinguer les langages selon leur façon de gérer le **typage** des variables :

Typage statique

Les types sont fixés à la **compilation**.

➔ Déclaration de toutes les variables et de leur type avant leur utilisation.

Typage dynamique

Un langage dans lequel les types sont découverts à **l'exécution**.

➔ Détermination du type d'une variable la première fois qu'on lui assigne une valeur.

ONERA

Précisions :

En règle générale : les langages interprétés sont à typage dynamique, alors que les langages compilés sont à typage statique.

Typologie des langages informatiques (2)

	Code Figé	Données Figées	Code Dynamique	Données Dynamiques
Fortran- >F77	X	X		
Lisp, Smalltalk			X	X
Fortran F90	X	X		X
C / C++	X	X		X
Java	X	X		X
Python, Matlab	X		X	X

ONERA

Page 7

Précisions :

-Fortran (jusqu'à F77) : pas d'allocation dynamique

-Lisp et Smalltalk sont des langages interprétés, tout est dynamique et du code peut générer du code

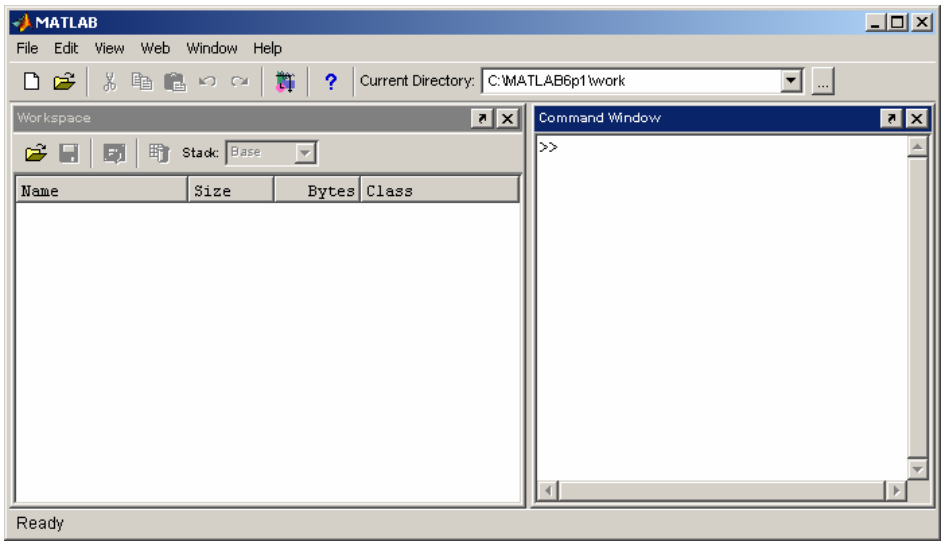
-Matlab et Python sont aussi interprétés, mais il n'est pas standard de rajouter du code durant l'exécution (définition de fonctions « lambda » en Python)

-Fortran 90, C et C++ sont des langages compilés où l'allocation dynamique est possible

-En Java, l'allocation dynamique est la règle générale, sauf pour les types de base

L'exemple de Matlab (1)

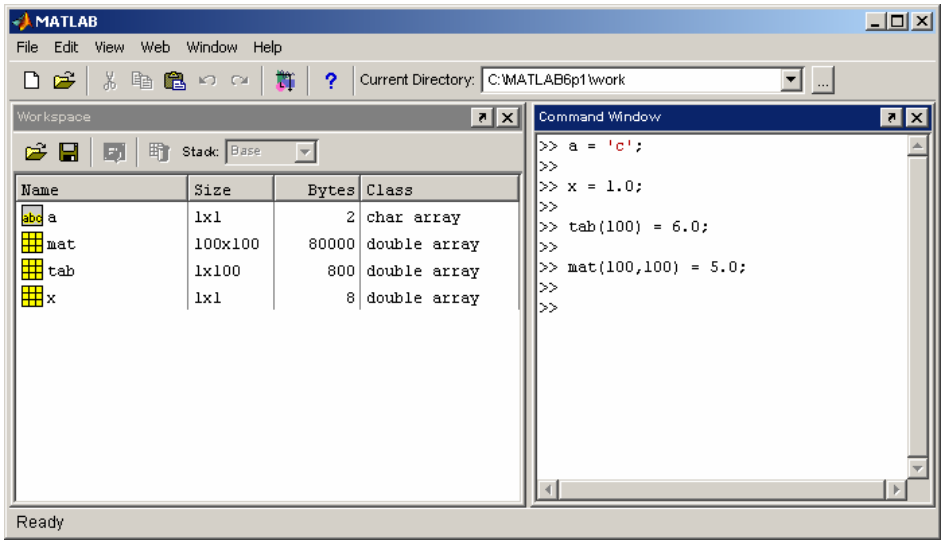
Au lancement du programme, l'espace mémoire est vide :



ONERA

L'exemple de Matlab (2)

Puis on peut ajouter des variables, elles sont allouées dynamiquement et l'espace de travail se remplit :

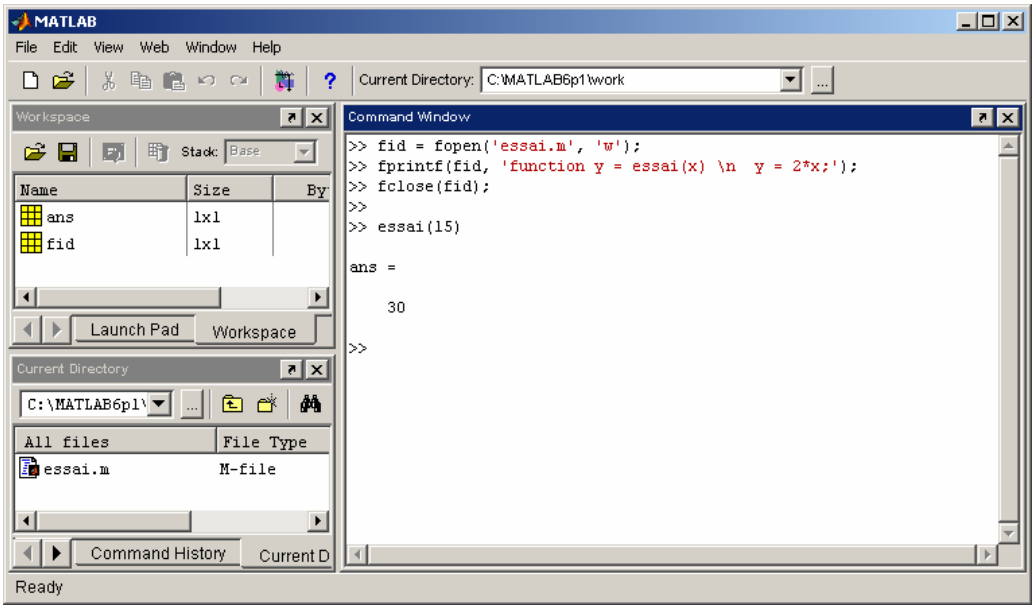


On peut détruire des variables avec la commande « `clear` ».

ONERA

L'exemple de Matlab (3)

On peut créer un fichier qui contient le code d'une nouvelle fonction et l'utiliser immédiatement :



ONERA

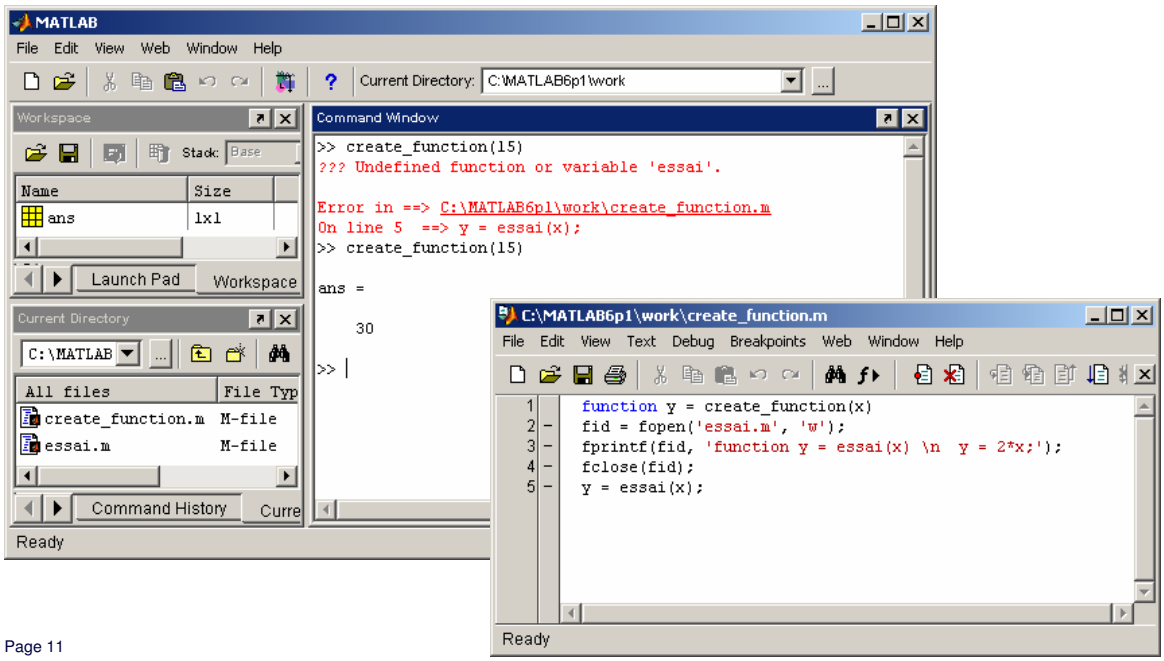
Un extrait de la documentation MATLAB :

What Happens When You Call a Function

When you call a function M-file from either the command line or from within another M-file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear it using the clear function, or until you quit MATLAB.

L'exemple de Matlab (4)

On peut écrire une fonction qui fabrique cette nouvelle fonction, mais MATLAB n'est pas un pur langage interprété et on ne peut utiliser tout de suite la nouvelle fonction :



Précision :

Comme le montre la trace, il faut appeler deux fois la fonction « `create_function` ». Le premier appel retourne une erreur, mais crée le fichier « `essai.m` », ce qui permet au deuxième appel de fonctionner.

Dans le même ordre d'idées, il ne semble pas possible de créer une fonction en utilisant la console de commande (on obtient un message du genre « `??? Strings passed to EVAL cannot contain function declarations` »).

Typologie des langages : passage des paramètres

Une chose importante à connaître est le mode utilisé par chaque langage pour passer les paramètres aux fonctions et procédures.

Passage par valeur

La fonction ne connaît pas la variable de départ, seulement sa valeur.

La variable ne peut être modifiée par la fonction/procédure.

Comment alors écrire une fonction qui modifie des variables ? [1]

Passage par référence

La variable peut être modifiée par la fonction/procédure.

Comment alors distinguer entre celles qui modifient la variable et les autres ? [2]

ONERA

Précisions :

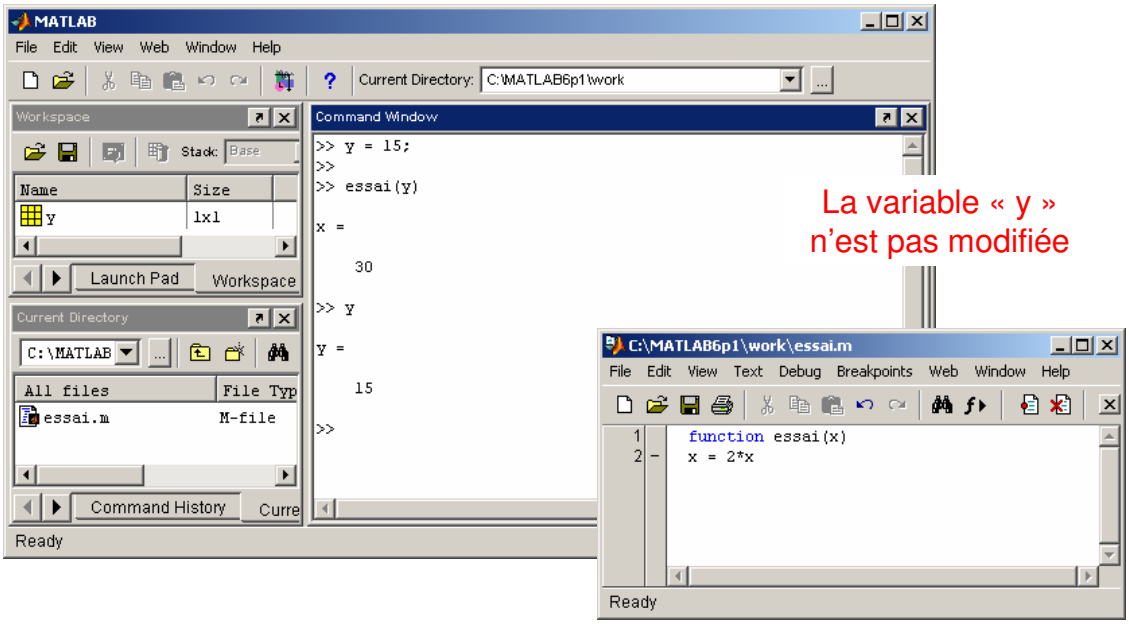
Une « procédure » est une fonction qui ne retourne pas de valeur. Certains langages font cette distinction, mais pas le C/C++.

[1] réponse du C/C++ : en passant la valeur de l'adresse de la variable (c'est-à-dire ce que l'on appelle un pointeur).

[2] réponse du C/C++ : en utilisant le mot-clé `const` pour signifier que la référence ne sera pas modifiée par la fonction.

Passage des paramètres : MATLAB

MATLAB utilise le passage par valeur :



ONERA

Un extrait de la documentation MATLAB :

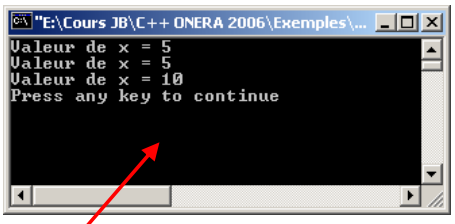
How MATLAB Passes Function Arguments

From the programmer's perspective, MATLAB appears to pass all function arguments by value. Actually, however, MATLAB passes by value only those arguments that a function modifies. If a function does not alter an argument but simply uses it in a computation, MATLAB passes the argument by reference to optimize memory use.

Passage des paramètres : langage C

Le C utilise aussi le passage par valeur :

```
#include <stdio.h>

void f(double y) {
     y = 2 * y;
}

void g(double *y) {
     *y = 2 * *y;
}

void main()
{
     double x = 5;
     printf("Valeur de x = %g\n", x);

     f(x);
     printf("Valeur de x = %g\n", x);

     g(&x);
     printf("Valeur de x = %g\n", x);
}
```



ONERA

Page 14

Explications :

- La première fonction « `f` » reçoit la valeur de la variable, elle copie cette valeur dans une variable qui est locale à la fonction « `f` ». C'est cette valeur locale qui est multipliée par `2`. Cela ne sert à rien, car la portée de cette variable est locale, justement.

- La deuxième fonction « `g` », reçoit aussi une valeur, mais c'est une adresse. La fonction utilise l'opérateur « `*` » qui signifie « la variable se trouvant à cette adresse » pour travailler sur la variable et modifier efficacement la valeur.

Nous verrons que la passage par référence du C++ simplifie grandement les choses.

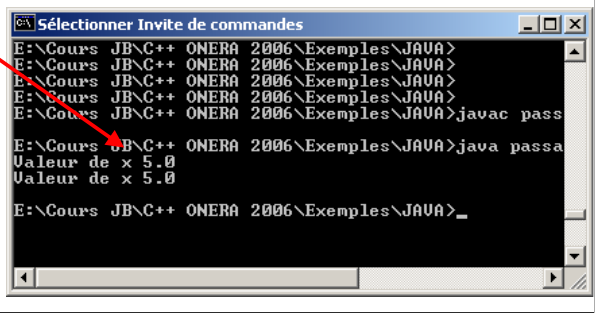
Exemple : `Exple_01_passageC`

Passage des paramètres : JAVA

En JAVA, ça dépend. Les types de base sont passés par valeurs :

```
public class passage
{
  static void f(double y) {
    y = 2 * y;
  }

  public static void main(String[] args) {
    double x = 5.0;
    System.out.println("Valeur de x " + x);

    f(x);
    System.out.println("Valeur de x " + x);
  }
};
```

Mais tous les autres types sont passés en référence.



Précisions :

Il existe en JAVA la notion de « classes d'enveloppe des types de base » ou *wrappers*.

Par exemple « `Double` » est la classe enveloppe du type primitif « `double` ». Quoiqu'il en soit, après 2h de recherche, je n'ai pas trouvé comment réaliser une fonction qui modifie la valeur d'une variable simple. Si quelqu'un veut bien chercher …

Passage des paramètres : récapitulation

	Passage par valeur	Passage par référence
MATLAB	OUI	Le langage l'utilise "dans le dos" du programmeur
Fortran, Lisp, Smalltalk, Python	NON	OUI
C	OUI => on passe les adresses des variables pour les modifier	NON
C++	oui, par défaut	possible, au choix du programmeur
Java	oui, types simples "dans le dos" du programmeur	oui, objets "dans le dos" du programmeur

ONERA

Conclusion :

Le C++ permet à l'utilisateur de choisir entre le passage par valeur et le passage par référence. Choix que ne permet aucun autre langage de la liste.

Le langage C++ : récapitulation

Le langage C++ est un langage **compilé** :

⇒ le code est figé et ne peut être étendu durant l'exécution

⇒ en contre partie, gain énorme en performances

Les données peuvent être **statiques** ou allouées **dynamiquement** selon les choix du programmeur :

⇒ statiques : gain de performance

⇒ allouées dynamiquement : gain de place mémoire

Le langage C++ permet au programmeur de choisir entre le passage par **valeur** ou le passage par **référence**.

ONERA

Précisions :

De tous les langages, le C++ offre le plus de possibilités au programmeur. A part les contraintes liées à tout langage compilé, il n'impose aucun choix préconçu.

C/C++ : premier programme (1)


```c
/* Une fonction
   aimable en C */

#include <stdio.h>

void main(void)
{
  printf("Hello\n");
}
```

```cpp
// Une fonction
// aimable en C++

#include <iostream>

void main()
{
  std::cout << "Hello" << std::endl;
}
```

Commentaires C : `/* sur plusieurs lignes */`

Commentaires C++ : `// sur une ligne`

ONERA

Exercice :

- créer deux projets `Exple_02_mainC` et `Exple_03_mainCPP`
- ces projets comprennent chacun un programme principal
- « main.c » pour le code C
- « main.cpp » pour le code C++

C/C++ : premier programme (2)

Tout ce qui était possible en C, l'est aussi en C++ :

```
/* les commentaires
   a la mode C */

#include <iostream>
#include <cstdio>                    // les librairies C

int main()
{
  std::cout << "Hello" << std::endl;
  printf ("Hello bis\n");            // les instructions C
  return 0;
}
```

En général, on considère que le programme principal « `main` » doit retourner « **0** » s'il s'est correctement exécuté.

ONERA

Exercice :

- poursuivre le code des exemples `Exple_02_mainC` et `Exple_03_mainCPP` en créant un nouveau projet : `Exple_04_mainCPP2`

C : mots-clés

Le langage C dispose de 33 mot-clés :

`asm`	`auto`	`break`	`case`
`char`	`const`	`continue`	`default`
`do`	`double`	`else`	`enum`
`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`
`return`	`short`	`signed`	`sizeof`
`static`	`struct`	`switch`	`typedef`
`union`	`unsigned`	`void`	`volatile`
`while`			

Auxquels on ajoute :

- des opérateurs unaires : `-` `~` `!` `*` `&` `sizeof` `+` `++` `--`
- des opérateurs binaires : `.` `->` `*` `/` `%` `+` `-` `<<` `>>` `<` `>` `<=` `>=` `==` `!=` `&` `|` `^` `&&` `||`
- des « ponctuations » : `[ ]` `( )` `{ }` `,` `:` `=` `;` `...` `#`

ONERA

C++ : nouveaux mots-clés

Le C++ introduit 41 nouveaux mot-clés qui s'ajoutent aux 33 du langage C :

bool	catch	class	const_cast
delete	dynamic_cast	explicit	export
false	friend	inline	mutable
namespace	new	operator	private
protected	public	static_cast	reinterpret_cast
template	this	throw	true
try	typeid	typename	using
virtual	wchar_t	and(&&)	and_eq(&=)
bitand(&)	bitor(|)	compl(~)	not(!)
not_eq(!=)	or(||)	or_eq(|=)	xor(^)
xor_eq(^=)			

=> Incompatibilités inévitables avec le langage C

```
char *friend = "Jeremie" ;    // illegal
void delete(void *ptr);       // illegal
int  new;                     // illegal
```

ONERA

Page 21

Sur l'exemple de la planche, voici les erreurs produites par VisualC++ 6.0 :

```
warning C4518: 'friend ' : storage-class or type specifier(s) unexpected here; ignored
error C2059: syntax error : '='
warning C4091: '' : ignored on left of 'void' when no variable is declared
error C2143: syntax error : missing ';' before 'delete'
error C2143: syntax error : missing ';' before 'delete'
warning C4091: '' : ignored on left of 'int' when no variable is declared
error C2143: syntax error : missing ';' before 'new'
error C2143: syntax error : missing ';' before 'new'
```

D'où l'intérêt d'un bon éditeur (avec au minimum coloration syntaxique) pour faire apparaître ces mots-clés.

Les mots-clés du langage C : `asm auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while`

Exemple : `Exple_05_motscles`

C/C++ : opérateurs (1)

Opérateur	Signification	Arité	Associativité
`: :`	*Scope resolution* Résolution de portée	binaire	non
`: :`	*Global* Portée globale	unaire	non
`[ ]`	*Array subscript* Manipulation de tableaux	binaire	gauche à droite
`( )`	*Function call* Appel de fonction	variable	gauche à droite
`.`	*Member selection (object)* Sélection de membre	binaire	gauche à droite
`->`	*Member selection (pointer)* Sélection de membre	binaire	gauche à droite
`++`	*Postfix increment* Incrément (suffixe)	unaire	non
`--`	*Postfix decrement* Décrément (suffixe)	unaire	non

ONERA

Page 22

Précisions :

- L'opérateur `: :` permet de lever les ambiguïtés entre des variables de même nom
- Le choix de l'opérateur de sélection dépend de l'entité à laquelle il s'applique
- Il existe des opérateurs d'incrémentation et de décrémentation préfixés

C/C++ : opérateurs (2)

Opérateur	Signification	Arité	Associativité
`new`	*Allocate object* Allocation dynamique	variable	non
`delete`	*Deallocate object* Libération mémoire	unaire ou binaire	non
`delete[ ]`	*Deallocate object* Libération mémoire (tableaux)	unaire ou binaire	non
`++`	*Prefix increment* Incrément (préfixe)	unaire	non
`--`	*Prefix decrement* Décrément (préfixe)	unaire	non
`*`	*Dereference* Valeur du pointeur	unaire	non
`&`	*Address-of* Adresse de	unaire	non
`-`	*Arithmetic negation (unary)* Négation unaire	unaire	non

ONERA

Précisions :

- Les opérateurs **`new`** et **`delete`** ont une grande importance en C++
- Les opérateurs **`*`** et **`&`** sont inverses l'un de l'autre
- L'opérateur de négation unaire est différent de l'opérateur de négation binaire

C/C++ : opérateurs (3)

Opérateur	Signification	Arité	Associativité
`!`	*Logical NOT* Négation logique	unaire	non
`~`	*Bitwise complement* Complément à un	unaire	non
`sizeof`	*Size of object or type* Taille d'une variable ou d'un type	unaire	non
`typeid( )`	*Type name* Nom d'un type (chaîne de caractères)	unaire	non
`(`*`type`*`)`	*Type cast (conversion)* Conversion de type (transtypage)	binaire	droite à gauche
`.*`	*Apply pointer to class member (objects)* Sélection de membre et déréférencement	binaire	gauche à droite
`->*`	*Dereference pointer to class member* Sélection de membre et déréférencement	binaire	gauche à droite
`*`	*Multiplication* Multiplication	binaire	gauche à droite

ONERA

Précisions :

- L'opérateur **`sizeof`** retourne une valeur du type **`size_t`**
- L'opérateur **`typeid`** retourne une valeur du type **`const type_info&`**
- Ne pas confondre la multiplication **`*`** et l'opérateur d'indirection **`*`**

C/C++ : opérateurs (4)

Opérateur	Signification	Arité	Associativité
/	*Division* Division	binaire	gauche à droite
%	*Remainder (modulus)* Modulo	binaire	gauche à droite
+	*Addition* Addition	binaire	gauche à droite
–	*Subtraction* Soustraction	binaire	gauche à droite
<<	*Left shift* Décalage à gauche	binaire	gauche à droite
>>	*Right shift* Décalage à droite	binaire	gauche à droite
<	*Less than* Inférieur à	binaire	gauche à droite
>	*Greater than* Supérieur à	binaire	gauche à droite

ONERA

Précisions :

• Les opérateurs **<** et **>** retournent un booléen (bool)

C/C++ : opérateurs (5)

Opérateur	Signification	Arité	Associativité
<=	*Less than or equal to* Inférieur ou égal à	binaire	gauche à droite
>=	*Greater than or equal to* Supérieur ou égal à	binaire	gauche à droite
==	*Equality* Egalité	binaire	gauche à droite
!=	*Inequality* Inégalité	binaire	gauche à droite
&	*Bitwise AND* ET binaire (bits)	binaire	gauche à droite
^	*Bitwise exclusive OR* OU exclusif binaire (bits)	binaire	gauche à droite
|	*Bitwise OR* OU binaire (bits)	binaire	gauche à droite
&&	*Logical AND* ET logique (bool)	binaire	gauche à droite

ONERA

Précisions :

• Les opérateurs de comparaison retournent un booléen (bool)

• Ne pas confondre les opérateurs logiques et ceux qui travaillent sur des champs de bits

C/C++ : opérateurs (6)

Opérateur	Signification	Arité	Associativité
`||`	*Logical OR* OU logique (bool)	binaire	gauche à droite
`e1?e2:e3`	*Conditional* Condition if-else	ternaire	droite à gauche
`=`	*Assignment* Affectation	binaire	droite à gauche
`*=`	*Multiplication assignment* Multiplication et affectation	binaire	droite à gauche
`/=`	*Division assignment* Division et affectation	binaire	droite à gauche
`%=`	*Modulus assignment* Modulo et affectation	binaire	droite à gauche
`+=`	*Addition assignment* Addition et affectation	binaire	droite à gauche
`-=`	*Subtraction assignment* Soustraction et affectation	binaire	droite à gauche

ONERA

Précisions :

• L'opérateur conditionnel ternaire permet d'écrire des expressions plus concises qu'avec if-else

C/C++ : opérateurs (7)

Opérateur	Signification	Arité	Associativité
<<=	*Left-shift assignment* Décalage à gauche et affectation	binaire	droite à gauche
>>=	*Right-shift assignment* Décalage à droite et affectation	binaire	droite à gauche
&=	*Bitwise AND assignment* ET binaire et affectation	binaire	droite à gauche
|=	*Bitwise inclusive OR assignment* OU binaire et affectation	binaire	droite à gauche
^=	*Bitwise exclusive OR assignment* OU exclusif binaire et affectation	binaire	droite à gauche
,	*Comma* Virgule	binaire	gauche à droite

ONERA

Précisions :

• L'opérateur virgule sert à évaluer deux ou plusieurs expressions, là où une seule est autorisée

• Il ne faut pas le confondre avec le signe de ponctuation virgule

Exercice : trouver les N erreurs de ce programme (`Exple_06_Nerreurs`)

```
struct MaStruct
{
  MaStruct *this;    // pointeur sur moi-même
  MaStruct *friend;  // pointeur sur le suivant
};

typedef MaStruct* try;
void using(MaStruct* s)  // fonction récursive
{
  using(s->this);
  using(s->friend);
}
void inline(MaStruct* new)  // Affichage
{
  MaStruct *public = new;
  cout << "INLINE\n" << public;
}
```

C : types de base

Types de base : **char, int, float, double**
+ des modificateurs optionnels : **long, short, unsigned, signed**

```
char c, ch;
c  = 'a';
ch = '\n';

int i, j;
i = 10;

long int ii;
long jj;
ii = 12345;
jj = 888888L;

int m = 1000;
long n = 999999L;
```

```
short int mm;
short nn;
short pp = 100;

unsigned int i1;
unsigned long j1;
unsigned short k1;
i1 = 345U;
j1 = 888888UL;

signed int i2;
signed long j2;
signed short k2;
```

```
float x1, x2;
float x3 = 0.1F;

double y;
long double z;

x1 = 0.45;
x2 =-3.5e12F;
y  = 0.45;
z  = 0.99e-2L;
```

void : pour spécifier qu'il n'y a pas de type

ONERA

Page 29

Important :

En C, on met toutes les déclarations de variables au début des fonctions. On ne peut mélanger déclarations et affectations dans n'importe quel ordre comme dans l'exemple ci-dessus. En C++, c'est possible, les variables peuvent être déclarées n'importe où dans le code.

Précisions :

Par défaut, une constante :

- entière : est de type « `int` »
- réelle : est de type « `double` » (ainsi l'instruction « `x1 = 0.45;` » avec `x1` nombre réel simple précision peut occasionner un *warning '=' : truncation from 'const double' to 'float'*).

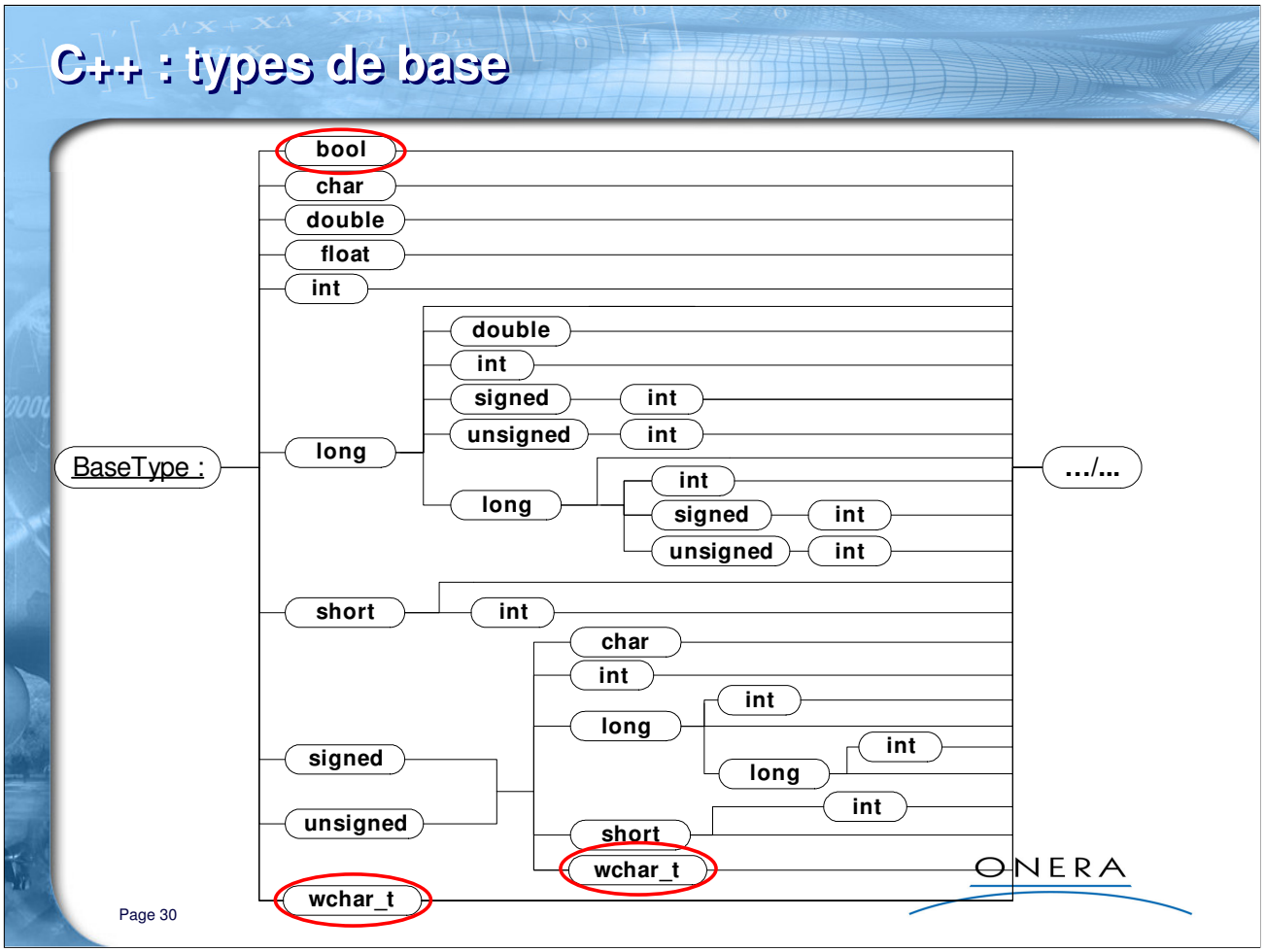
Les lettres « `L` », « `U` » et « `F` » permettent de préciser le caractère « `long` », « `unsigned` » ou « `float` » d'une constante..

A ces types de base, le C++ ajoute deux autres types de base : « `bool` » et « `wchar_t` ».

Il est bien évident, que (contrairement au Fortran) les noms des variables et des constantes sont complétement libres. Certaines lettres ne sont pas considérées par défaut comme entiers ou réels !

La différence entre signés et non signés tient à la plage de valeur que peuvent représenter des variables de ce type. Par exemple un entier signé 16 bits peut représenter des nombres entre -32768 et +32767, alors que non signé la plage de valeur est de 0 à 65535.

Exemple : `Exple_07_typesbaseC`



Précisions :

Nous verrons le type « **`bool`** » en détail plus loin.

En ce qui concerne le type « **`wchar_t`** », il s'agit d'une représentation des caractères généralement sur 16 bits ce qui permet d'utiliser le codage Unicode des caractères à la place du codage ASCII (8 bits). Ce codage étendu est nécessaire pour représenter des alphabets non latins et envisager ce que l'on appelle *l'internationalisation* des programmes.

Il faut utiliser la lettre « `L` » en préfixe pour spécifier qu'un caractère est étendu (= *wide*) :

```
wchar_t  w1 = 'x';
wchar_t  w2 = '\t';
wchar_t  w3 = L'yz';
wchar_t  w4 = L'\x77\x34';
```

Exemple : `Exple_08_typesbaseCPP`

Note importante :

Le C et le C++ n'ont pas de type de base pour représenter les chaînes de caractères (comme en JAVA le type « `string` »). Les chaînes de caractères sont représentées par des tableaux de caractères. Il existe cependant dans la librairie standard de C++ une classe `std::string`, mais il ne s'agit pas d'un type de base, faisant partie du noyau du langage.

C/C++ : implémentation des types de base

La norme ne fixe pas l'implémentation des types de base par les compilateurs, contrairement à JAVA. Il peut donc exister des différences entre les compilateurs.

Ce que la norme impose :

- la taille de `char`, `unsigned char` et `signed char` est de 1 octet
- la taille de `wchar_t` (`signed` / `unsigned`) est supérieure ou égale à celle de `char`
- la taille de `short int` (`signed` / `unsigned`) est supérieure ou égale à celle de `char`
- la taille de `int` (`signed` / `unsigned`) est supérieure ou égale à celle de `short int`
- la taille de `long int` (`signed` / `unsigned`) est supérieure ou égale à celle de `int`
- la taille de `double` est supérieure ou égale à celle de `float`
- la taille de `long double` est supérieure ou égale à celle de `double`

```
                          wchar_t  ≥  char   =    8 bits
long int ≥  int   ≥  short int  ≥  char   =    8 bits
     long double  ≥  double  ≥  float  = Non fixée
                                bool   = Non fixée
```

ONERA

Page 31

Précisions :

En principe, le type « `int` » est de la taille du type natif du processeur utilisé, 32 bits par exemple sur une architecture 32 bits.

Exercice :

- créer un projet `Exple_09_sizeof`

- dans le programme principal, utiliser la méthode « sizeof » qui retourne la taille d'une variable (d'un objet ou d'un type) pour afficher la taille de plusieurs types de base. Par exemple :

```
std::cout << sizeof(int) << std::endl;
```

Reporter ici les tailles :

```
char =                          wchar_t =
unsigned char =                 long =
signed char =                   unsigned long =
short int =                     signed long =
unsigned short int =            float =
signed short int =              double =
int =                           long double =
unsigned int =                  bool =
signed int =
```

C/C++ : littéraux caractères

Les littéraux caractères se notent avec des guillemets simples (« quote » en anglais). Eventuellement précédés de « **L** » pour les caractères longs :

```
char    c1 = 'a';
wchar_t c2 = L'b';
```

Les caractères inaccessibles au clavier peuvent être fournis grâce à leur code en base 16 (précédé de **\x**) ou en base 8 (simplement précédé de ****) :

```
char  c1 = 'a';
char  c2 = L'a';
char  c3 = '\141';
char  c4 = '\x61';
```

toutes représentations du caractère 'a'

Il existe un certain nombre de séquences d'échappement reconnues par le langage C/C++ :

```
\'   \"   \?   \\
\a   \b   \f   \n   \r   \t   \v
```

ONERA

Page 32

Liste des séquences d'échappement :

newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash \	\\
question mark ?	\?
single quote '	\'
double quote "	\"

Pour représenter les guillemets simples (') et le backslash (\), la séquence d'échappement correspondante est obligatoire (\') et (\\) respectivement.

Exercice (facultatif) :

- créer un projet `Exple_10_litteraux`
- s'exercer à définir quelques exemples de littéraux caractères et les afficher avec `std::cout <<`

C/C++ : littéraux entiers

Les littéraux entiers sont des nombres, sans exposant, ni virgule.

- en base décimale (10), les nombres entiers ne peuvent commencer par « **0** »
- en base octale (8), ils doivent commencer par « **0** »
- en base hexadécimale (16), ils doivent commencer par « **0x** » ou « **0X** »
- les nombres peuvent être précédés d'un opérateur unaire (« **–** » ou « **+** »)

```
unsigned int a1 = 67;
unsigned int a2 = 0103;
unsigned int a3 = 0x43;
```

toutes représentations du nombre `67'

Rappel : les suffixes « **U** » et « **L** », en minuscules ou majuscules, seuls ou ensembles peuvent être employés pour préciser si l'entier est non signé ou s'il est long.

ONERA

Exemples :

Quelques exemples variés de constantes entières :

```
long h1 = 12345678L;
long h2 = 12345678l;
unsigned long h3 = 12345678U;
unsigned long h4 = 12345678u;
unsigned long h5 = 12345678UL;
unsigned long h6 = 12345678ul;
unsigned long h7 = 12345678LU;
unsigned long h8 = 12345678lu;
```

Exercice (facultatif) :

- poursuivre le projet précédent `Exple_10_litteraux`
- s'exercer à définir quelques exemples de littéraux entiers et les afficher avec `std::cout <<`

C/C++ : littéraux réels

Les littéraux flottants sont des nombres réels avec un exposant et/ou une virgule. Les nombres peuvent être précédés d'un signe, opérateur unaire (« **–** » ou « **+** ») :

```
float x1, x2;
float x3 = 0.1F;

double y;
long double z;

x1 =  0.45;
x2 = -3.5e12F;
y  =  0.45;
z  =  0.99e-2L;
```

Rappel : les suffixes « **F** » et « **L** », en minuscules ou majuscules, permettent de préciser respectivement que la constante est simple précision (« `float` ») ou quadruple precision (« `long double` »). S'il n'y a aucun suffixe, il s'agit d'une constante double précision (« `double` »).

ONERA

Exercice (facultatif) :

- poursuivre le projet précédent `Exple_10_litteraux`
- s'exercer à définir quelques exemples de littéraux réels et les afficher avec `std::cout <<`

C/C++ : littéraux chaînes de caractères

En C et C++, les littéraux chaînes de caractères sont définis comme des tableaux de n caractères du type « **`const char`** ».

En C et C++, les chaînes de caractères se terminent par le caractère zéro (**`\0`**) qui compte dans le nombre de caractères. C'est ce que l'on appelle les chaines de caractères « *null terminated* » par opposition aux chaînes de la classe **`string`** de la librairie standard C++ que nous verrons plus tard.

```
const char s1[5] = "abcd";    // OK
const char s2[5] = "abcde";   // error: array bounds overflow
s1[0] = 'R';                  // error: l-value specifies
                              // const object

char s3[5] = "abcd";          // OK
s3[0] = 'R';                  // OK
```

ONERA

Précisions :

Il ne faut pas confondre les littéraux, les constantes et les variables :

- littéraux : `"abcd"` ou `'R'`
- constantes : `const int I`, `const char S[5]`
- variables : `int I`, `char S[5]`

Exercice (facultatif) :

- poursuivre le projet précédent `Exple_10_litteraux`
- s'exercer à définir quelques exemples de littéraux chaînes et les afficher avec `std::cout <<`

C++ : le type bool

Contrairement à C, le langage C++ dispose d'un type booléen, le type **bool**, dont les valeurs sont **true** et **false**. Les conditions du `if`, `while`, `do`, `for` et de l'expression conditionnelle **?** **:** sont des expressions de type `bool`. Les opérateurs relationnels et logiques travaillent sur ce nouveau type.

```
int i, j, k;
bool b;

b = (i>j);      // Bien
k = (j>i);      // A éviter

while (b)       // Bien
{
  // ...
}

if (k)          // A éviter
  ...
```

Pour des raisons de compatibilité avec le C, les conversions implicites entre booléens et types arithmétiques sont conservées.

CONSEIL : les éviter

ONERA

Précisions :

En C, le type booléen n'existe pas. On représente les valeurs « vrai » et « faux » par des entiers. Le zéro signifie « faux » et n'importe quel nombre différent de zéro signifie « vrai ».

Le C++ définit deux « littéraux » booléens : **true** et **false**.

Exercice :

- en le compilant, regardez ce que produit le code suivant (`Exple_10_litteraux`) :

```
bool b1 = 0;
bool b2 = 4;
bool b3 = false;
bool b4 = true;
std::cout << b1 << "\t" << b2 << "\t"
          << b3 << "\t" << b4 << std::endl;
```

C/C++ : les identifieurs

Les identifieurs ou **symboles** permettent de nommer :

- des classes (`class`), des structures (`struct`) ou des unions (`union`)
- des variables, des constantes, ou des instances de classe
- des types énumérés (`enum`)
- des attributs de classes
- des fonctions ou des méthodes de classes
- des alias de types (`typedef`)
- des labels (`goto`), des macros (`#define`) ou des paramètres de macros

Un symbole C/C++ est formé d'une suite de lettres (**`a-z A-Z`**), de chiffres (**`0-9`**) ou de soulignés (*underscore* _). Le premier caractère d'un symbole est obligatoirement une lettre ou un souligné :

```
int   H1B4_8;
float __abc_1_2;
char  x1x2x3;
```

ONERA

Notes :

- le C/C++ est sensible à la casse minuscules/majuscules des caractères (`filename` et `FileName` sont deux symboles différents).
- les symboles ne peuvent être des mots-clés du langage (`int` et `float` par exemple)
- il est déconseillé d'utiliser des symboles commençant par deux soulignés (__) ou un souligné suivi d'une lettre majuscule (cf. la norme C++).

Ce que dit la norme C++ :

17.4.3.1.2 Global names

Certain sets of names and function signatures are always reserved to the implementation:

— Each name that contains a double underscore (_ _) or begins with an underscore followed by an uppercase letter (2.11) is reserved to the implementation for any use.

— Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

Exemple : `Exple_11_symboles`

C/C++ : définitions et déclarations (1)

Définir une variable, une classe ou une fonction c'est lui donner un type, des attributs ou un corps de fonction. La définition est unique.

Tous les symboles utilisés pour compiler un fichier doivent être **déclarés**. La déclaration peut consister à définir le symbole en question ou à rappeler des symboles ailleurs.

```
// CI-DESSOUS, DES DEFINITIONS, SAUF UNE DECLARATION
int a;                          // definit a
extern const int b = 1;         // definit b
int f(int x) { return x+a; }    // definit f et definit x

struct S { int a; int b; };     // definit S, S::a, et S::b

struct T                        // definit T
{
   int t;           // definit l'attribut d'instance T::t
   static int a;    // declare l'attribut de classe T::a
   T(): t(0) { }    // definit un constructeur de T
};
```


A suivre …

ONERA

Notes :

Il est possible d'arriver à distinguer les variables qui portent le même nom (ici il y a par exemple de multiples définitions de la variables « `a` »). grâce à l'opérateur de résolution de portée (`::`) ou aux opérateurs de sélection de membres (« `.` », « `->` », « `.*` », « `->*` »).

(cf exemple planche suivante)

Exemple : `Exple_12_definitiondeclaration`

C/C++ : définitions et déclarations (2)

… suite et fin

```
int T::a = 1;              // definit T::a

enum { haut, bas };        // definit haut et bas
namespace MyN { int a; } // definit MyN et MyN::a
namespace AliasN = MyN;   // definit AliasN

T uneInstanceT;            // definit uneInstanceT


// CI-DESSOUS, DES DECLARATIONS
extern int a;             // declare a
extern const int b;       // declare b
int f(int);               // declare f
struct S;                 // declare S
typedef int Int;          // declare Int
extern T unAutreT;        // declare unAutreT
using MyN::a;             // declare MyN::d
```

ONERA

Exemples de manipulations des variables « a » : `Exple_12_definitiondeclaration`

```
::a = 10;
std::cout << "Valeur de    ::a = " << ::a << std::endl;

S s1;
s1.a = 20;
std::cout << "Valeur de   s1.a = " << s1.a << std::endl;

S* s2 = new S();
s2->a = 30;
std::cout << "Valeur de  s2->a = " << s2->a << std::endl;
delete s2;

T::a = 40;
std::cout << "Valeur de   T::a = " << T::a << std::endl;

uneInstanceT.a = 50;
std::cout << "Valeur de   T::a = " << T::a << std::endl;

MyN::a = 60;
std::cout << "Valeur de MyN::a = " << MyN::a << std::endl;

AliasN::a = 70;
std::cout << "Valeur de MyN::a = " << MyN::a << std::endl;
```

C++ : définition des variables

La définition d'une variable se compose au minimum d'un nom (symbole) du type de données et du nom (symbole) de l'identifiant :

```
type identifiant;
```

On peut optionnellement donner une valeur à la variable :

```
type identifiant [= valeur];
```

On peut définir d'un coup plusieurs variables du même type en les séparant par une virgule :

```
type ident1 [= val1], ident2 [= val2], … identN [= valN];
```

On peut les déclarer pratiquement n'importe où (en se limitant ainsi aux stricts endroits nécessaires), ce qui est une grande différence avec le C (cf. planche suivante).

ONERA

C vs C++ : définition des variables

1) en C++, pas nécessairement regroupées en début de bloc
2) leur portée s'arrête à la fin du bloc (accolade fermante `}`)
3) on peut initialiser avec des valeurs obtenues par calcul
4) on peut déclarer dans les parenthèses des `if`, `for`, `while`, `do`

En C

```c
int main(void)
{
  int i, n, m;
  float x;
  scanf("%d", &n);
  m = n;

  for (i=0; i<m; i++)
  {
    /* ... */
  }
  return 0;
}
```

En C++

```c
int main()
{
  int n;
  std::cin >> n;
  const int m = n;

  for (int i=0; i<m; i++)
  {
    float x;
    // ...
  }
  return 0;
}
```

ONERA

Page 41

Précisions :

- dans le code C, les variables sont valables dans tout le code du « `main` »

- dans le code C++ :

  - la variable « `n` » et la constante sont valables dans tout le « `main` »
  - la variable « `i` » est seulement valable dans le « `for` »
  - la variable « `x` » est seulement valable dans le bloc interne au « `for` »

C++ : variables « globales »

Il est possible de définir/déclarer des variables dans un fichier « cpp », c'est-à-dire dans ce que l'on appelle **une unité de compilation**, en dehors de toute fonction, classe ou méthode :

```
int globale;
static int locale;
```

fichier1.cpp

```
int globale;          // Erreur de link
extern int globale;   // OK
int locale;           // OK, mais pas locale …
```

fichier2.cpp

- « **static** » : la variable est locale au fichier
- « **extern** » : la variable est définie ailleurs, il s'agit d'une déclaration

ONERA

Précisions :

- la variable « `globale` » du *fichier1* est globale au programme, on ne peut pas la redéfinir ailleurs (erreur de link). En revanche, on peut la manipuler ailleurs en la déclarant « `extern` ».

- la variable « `locale` » du *fichier1* est locale à ce fichier (mot-clé « `static` »). Elle n'est connue que de lui. On peut donc ailleurs définir une variable globale qui porte le même nom (mais cela est dangereux et embrouillé !).

Exemple : `Exple_13_portee`

C++ : opérateur de résolution de portée

L'opérateur de résolution de portée (`::`) permet d'accéder à ces variables dites « globales ou « locales » à un fichier.

C'est un opérateur unaire. On doit obligatoirement s'en servir quand la variable est **masquée** par une variable qui porte le même nom, à l'intérieur du code d'une fonction par exemple :

```
int i;
void f() {
  int i;
  ::i = 1;
  i = 2;
}
```

Dans la pratique :

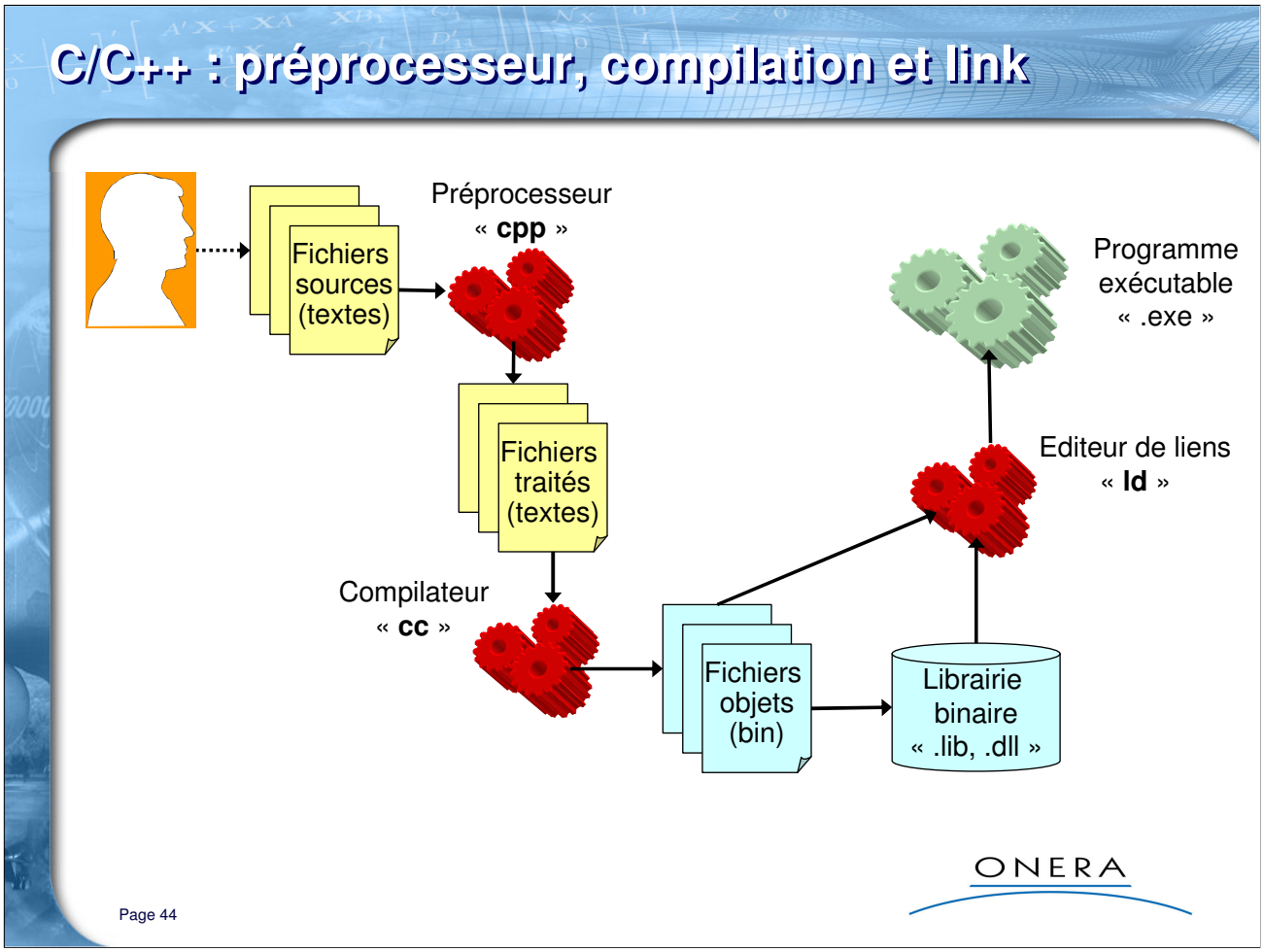
- on évitera les variables globales, autant que possible ;
- on évitera de masquer les noms des variables, quand il y a un risque d'ambiguïté, c'est une source d'erreurs ;
- mais parfois, c'est inévitable, surtout quand on réutilise du code de quelqu'un d'autre.

ONERA

Exercice : `Exple_14_resolutionportee`

- devinez, sans le tester dans un compilateur ce que produit le code suivant :

```
#include <iostream>

int YY = 1;

void main()
{
  YY = 2;
  int YY = 3;
  std::cout << "  YY = " << YY << std::endl;
  std::cout << "::YY = " << ::YY << std::endl;
  {
    YY = 4;
    int YY = 5;
    ::YY = 6;
    std::cout << "  YY = " << YY << std::endl;
    std::cout << "::YY = " << ::YY << std::endl;
  }
  std::cout << "  YY = " << YY << std::endl;
  std::cout << "::YY = " << ::YY << std::endl;
}
```



Précisions :

La fabrication d'un exécutable comprend 3 phases :

- la phase de préprocessing (les fichiers sources « .h », « .c », « .cpp » écrits par le développeur sont traités pour produire de nouveaux fichiers textes) qui produit des fichiers textes appelés « unités de compilation »

- la phase de compilation qui transforme les unités de compilation en code binaire (fichiers « objets »)

- l'assemblage des fichiers objets en librairies ou en codes exécutables

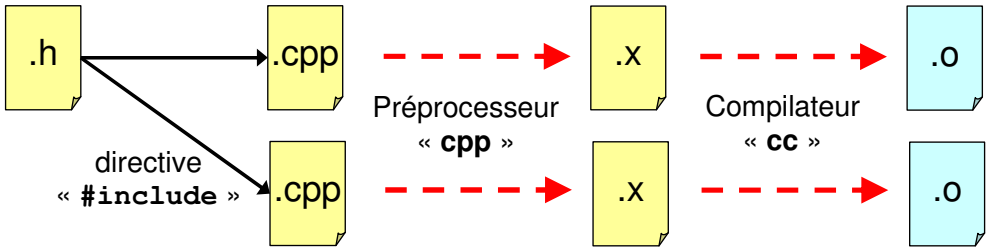
C/C++ : organisation modulaire des fichiers

On organise généralement les fichiers sources de façon modulaire, en assemblant des éléments communs interagissant dans un même fichier source et en séparant les éléments qui interagissent peu.

=> ainsi les fichiers sont compilables séparément

Un fichier source (extensions « .cc » ou « .cpp ») contient par exemple toutes les méthodes d'une classe d'objets ou tous les algorithmes réalisant une certaine tâche.

Des fichiers d'en-tête (extension « .h ») comprennent des déclarations communes à plusieurs fichiers sources.



ONERA

Précisions :

- Il n'y a pas d'obligation de donner des extensions « .h » ou « .cpp » aux fichiers sources. Comme d'habitude en C++, il n'y a rien d'obligatoire, il s'agit juste d'un usage et de suffixes de fichiers que les environnements de développement arrivent à reconnaître automatiquement.

- Il faut imaginer que les fichiers inclus sont « copiés-collés » tels quels dans le fichier source où se trouve la directive `#include`. C'est le préprocesseur qui se charge de ce travail.

- Des fichiers d'en-têtes peuvent être inclus dans d'autres fichiers d'en-tête.

C/C++ : le préprocesseur (1)

Le préprocesseur manipule le code source au niveau textuel.

Une tâche importante confiée au préprocesseur est de se prémunir contre les inclusions multiples de fichiers d'en-tête ou contre les inclusions réciproques. C'est une tâche qui est confiée au préprocesseur grâce aux deux directives **`#define`** et **`#ifdef/#ifndef`**.

Tous les fichiers en-tête "`.h`"

```
/* Fichier : module.h
   Auteur  : moi-meme
   Date    : 1/1/2000 */

#ifndef MODULE_H
#define MODULE_H
...
/* Ne rien mettre avant le #ifndef
   et rien apres le #endif      */
...
#endif
```

Tous les fichiers "`.h`" et "`.cpp`"

Chaque inclusion est soumise à un test par le préprocesseur

```
#include "module.h"
...
```

ONERA

Précisions :

Pourquoi se prémunir contre les inclusions réciproques ?

- Parce que c'est un processus sans fin : « `A.h` » inclut « `B.h` » qui inclut « `A.h` », etc.

- Algorithme :

  - si la macro « `MODULE_H` » n'est pas définie :
    - la définir
    - traiter le contenu du fichier
  - sinon : ne rien faire

- le nom de la macro est quelconque, mais il vaut mieux utiliser un nom en rapport avec le nom du fichier (on peut mettre des underscores, on met généralement « `_H` » à la fin).

=> C'EST UN REFLEXE A ACQUERIR SYSTEMATIQUEMENT POUR TOUT .H

Exemple : `Exple_15_macros`

C/C++ : le préprocesseur (2)

Les directives du préprocesseur commencent toutes par un dièse (**#**) et sont placées en début de ligne.

- la directive **#include** permet d'inclure un fichier :

```
#include "fichier"   // inclusion d'un fichier depuis le
                     // repertoire courant de l'utilisateur

#include <fichier>   // inclusion d'un fichier systeme
                     // repertoires include additionnels
```

- la directive **#define** définit des macros, par exemple :

```
#define MAX_S 100

#define norm(x,y)   sqrt((x)*(x)+(y)*(y))
```

→ A chaque fois que le préprocesseur rencontre « **MAX_S** » ou qu'il rencontre « **norm(,)** », il remplace par le texte correspondant.

ONERA

Précisions :

- l'inclusion avec les guillemets concerne généralement les fichiers écrits par le programmeur, l'inclusion avec des chevrons (<>) concerne les librairies (C/C++, STL).

- une « macro » est un alias : il faut imaginer qu'à chaque fois que le préprocesseur rencontre la macro il la remplace par le texte correspondant. Une macro ne définit pas des constantes ou des fonctions au sens informatique; il s'agit on le répète d'un traitement purement textuel.

Exemple : `Exple_15_macros`

C/C++ : le préprocesseur (3)

On peut définir des zones de compilation conditionnelle :

- directive **#ifdef / #ifndef / #else / #endif**
- directive **#if / #elif / #endif**

```
#ifdef WINDOWS_VERSION
#include <conio.h>
      const int I;
      void calcul();
#endif

#if MAX_S == 100
      // Faire quelque chose
#elif MAX_S == 99
      // Faire autre chose
#else
      // Faire une chose par defaut
#endif
```

Il faut toujours mettre **#endif** à la fin.

ONERA

Précisions :

- `#ifdef` : si la macro est définie
- `#ifndef` : si la macro n'est pas définie
- `#if` : si la condition est remplie
- `#elif` : sinon, si la condition est remplie

C/C++ : le préprocesseur (4)

Les macros sont sources d'erreurs et peuvent être difficiles à manipuler. En particulier les parenthèses autour des arguments des macros sont absolument nécessaires :

```
#define MUL(x,y) (x*y)        // ERREUR
#define MUL(x,y) ((x)*(y))    // OK
```

Toujours penser en termes de remplacement texte à texte, ne jamais croire qu'il s'agit d'une fonction :

```
#define norm(x,y) sqrt((x)*(x) + (y)*(y))
double x = 0.0;
double y = 1.0;
std::cout << norm(++x, y) << std::endl;
```

⟶ Affiche racine de 3 au lieu de racine de 2 !
et `x` vaut 2 car elle a été incrémentée 2 fois

ONERA

Exercice :

- « Expansez » sur le papier les macros « `MUL` »
- et essayez de deviner le résultat du calcul : `MUL(3+5, 1+1)`

Explication :

- l'opérateur « `++` » préfixé signifie : j'incrémente la variable, puis je fais le calcul dans lequel elle est impliquée
- l'opérateur « `++` » suffixé signifie : je fais le calcul avec la variable et ensuite je l'incrémente
- `norm(++x, y)` est expansé en `sqrt((++x)*(++x) + (y)*(y))`

Exemple : `Exple_15_macros`

C/C++ : le préprocesseur (5)

Conclusion : essayer de limiter l'utilisation du préprocesseur à la compilation conditionnelle. Cependant, dans certains cas, les macros sont irremplaçables.

Elles sont remplacées en C++ par des moyens plus sûrs :

- les fonctions en ligne (*inline*) ;
- les patrons (*templates*).

Lors d'une utilisation avancée du C/C++ on peut rencontrer :

- d'autres directives : **`#error`**, **`#line`**, **`#pragma`**, **`#undef`**
- un autre opérateur : **`##`**
- des macros prédéfinies : `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__`, `__cplusplus`

ONERA

Précisions :

- `#error` : affiche un message d'erreur à la compilation
- `#line` : agit sur le numéro de la ligne qui est compilée
- `#pragma` : dépend de l'implémentation du compilateur
- `#undef` : contraire de `#define`
- `##` : permet de concaténer un argument de macro

C/C++ : déclarations/définitions de tableaux

Indiquer la dimension lors de la déclaration de la variable. On accède aux éléments via une notation entre crochets : [ ].

Attention : en C/C++, la numérotation des éléments commence à zéro.

Un tableau de 100 nombres réels :

```
float tab1[100];
```

Les tableaux multidimensionnels sont des tableaux de tableaux. Ainsi par exemple, voici un tableau de tableaux de 50 nombres entiers :

```
int tab2[20][50];

tab1[5]   = 1.5e6;
tab1[100] = 0.0;  /* ERREUR A L'EXECUTION */

tab2[4]; /* 5eme element de tab2, c'est
            lui-meme un tableau */
```

ONERA

C++ : expressions constantes

Une expression constante est une expression qui peut être évaluée à la compilation :

```
const int MAX = 100;  // expression non constante en C
                      // expression constante en C++
```

Les constantes C++ se rapprochent des symboles définis avec **#define** (que l'on évitera d'utiliser désormais pour définir des constantes) :

En C

```
#define M 1000
const int N = 100;

char tabm1[M];      // OK
char tabm2[M+1];    // OK
char tabn[N];       // illégal
char tabnm[M*N];    // illégal
```

En C++

```
const int M = 1000;
const int N = 100;

char tabm1[M];      // OK
char tabm2[M+1];    // OK
char tabn[N];       // OK
char tabnm[M*N];    // OK
```

ONERA

Précisions :

C'est évident, mais ça va mieux en le disant : une expression constante ne peut pas être modifiée durant l'exécution.

Il ne faut pas s'y tromper les calculs « `M+1` » ou « `M*N` » ne sont pas faits à l'exécution, mais durant la compilation.

Exemple : `Exple_16_constantesC` et `Exple_17_constantesCPP`

C/C++ : déclarations de tableaux et initialisation

Ce mode de déclaration de tableaux permet de déclarer des tableaux dont la taille est connue **à la compilation**.

Pour les tableaux dont la taille n'est connue que durant l'exécution, il faut utiliser **l'allocation dynamique** :

```
int n = 10;
double tab3[n];  // error : expected constant expression
```

On peut **initialiser** les valeurs d'un tableau avec la syntaxe suivante :

```
int t1[]     = {1, 2, 3};     /* dim. implicite : 3 */
double t2[5] = {4.0, 5.0};    /* dimension 5, mais
                                 initialisation de
                                 t2[0] et t2[1]     */
int t3[2][3] = {10, 20, 30, 40, 50, 60};
int t4[2][3] = {{10, 20, 30}, {40, 50, 60}};
```

ONERA

Premier exemple d'allocation dynamique :

```
int n = 10;
double* tab = new double[n];
// Bla ... bla
delete[] tab;
```

Exemple : `Exple_18_tableaux`

C/C++ : déclarations de chaînes de caractères

Les chaînes de caractères sont des tableaux de caractères à une dimension. Une chaîne se termine au premier caractère "null" (`'\0'`) rencontré. Il n'y a donc pas de limite théorique de longueur.

Le "null" est automatiquement ajouté pour les constantes, dans les autres cas, il faut le gérer *à la main*.

La taille déclarée d'une chaîne de caractères doit toujours être supérieure de 1 à ce que l'on désire y stocker.

```
char chaine[100];

char texte[] = "Hello\n";    // Tableau de 7 caractères

char s[5];
s[0] = 'a';
s[1] = 'b';
s[2] = 'c';
s[3] = '\0';
```

ONERA

Page 54

Précision :

1) La manipulation « avancée » des chaînes de caractères (et des tableaux) nécessite de comprendre et de maîtriser le mécanisme des pointeurs.

2) Comme pour les tableaux, il faut utiliser l'allocation dynamique (que nous verrons plus tard) si la chaîne de caractères est de taille variable (du moins indéterminée à la compilation).

Fonctions de la librairie C :

La libraire standard du C propose plusieurs fonctions de manipulations des chaînes de caractères :

```
#include <cstring>
char* strcat (char* s1, char* s2);  // concatène la chaine s2
                                    // a la fin de s1 et retourne s1
int strcmp(char* s1, char* s2);     // Compare selon l'ordre lexicographique
char* strcpy(char* s1, const char* s2); // Copie s2 dans s1 et retourne s1
int strlen(char* s);                // Retourne la taille, non compris
                                    // le caractère null de fin
```

Exemple : `Exple_19_chaines`

C/C++ : opérateurs logiques et mathématiques (1)

Opérateurs « multiplicatifs » :

Opérateur	Signification	Arité	Associativité
*	*Multiplication* Multiplication	binaire	gauche à droite
/	*Division* Division	binaire	gauche à droite
%	*Remainder (modulus)* Modulo	binaire	gauche à droite

Précisions :

- la division est la division entière si les deux arguments sont entiers
- le modulo ne travaille que sur des arguments entiers

Page 55

ONERA

Précision :

Si le deuxième argument de `/` ou de `%` est zéro, le comportement est indéfini; sinon la valeur de `(a/b)*b + a%b` est égale à `a`.

Précision très importante :

Il ne faut pas confondre l'opérateur `*` de multiplication avec l'opérateur `*` d'indirection. Le premier est binaire, le second unaire et il ne font franchement pas la même chose (cf. partie « pointeurs »).

Exemple : `Exple_20_operateurs`

C/C++ : opérateurs logiques et mathématiques (2)

Opérateurs « additifs » :

Opérateur	Signification	Arité	Associativité
+	*Arithmetic addition (unary)* « addition » unaire	unaire	non
–	*Arithmetic negation (unary)* Négation unaire	unaire	non
+	*Addition* Addition	binaire	gauche à droite
–	*Subtraction* Soustraction	binaire	gauche à droite

Précisions :

Il est clair que « l'addition » unaire ne fait rien pour les types simples (nombres entiers et réels), mais grâce à la *surcharge des opérateurs*, on peut donner un sens à cet opérateur sur des types (= classes) complexes.

ONERA

C/C++ : opérateurs logiques et mathématiques (3)

Opérateurs de « décalage » :

Opérateur	Signification	Arité	Associativité
<<	*Left shift* Décalage à gauche	binaire	gauche à droite
>>	*Right shift* Décalage à droite	binaire	gauche à droite

Précisions :

Ces opérateurs sont assez peu employés dans leur contexte « C » (décalages de bits), mais ils prennent une importance considérable en C++ dans la manipulation des flux d'entrées-sorties (y compris lecture de fichiers et formatages de toutes sortes).

ONERA

Exemple :

```
int b = 100;  // 1100100 en code binaire

std::cout << "     b = " << b << std::endl;
std::cout << "(b<<1) = " << (b<<1) << std::endl; // 200 = 11001000
std::cout << "(b<<2) = " << (b<<2) << std::endl; // 400 = 110010000

std::cout << "(b>>1) = " << (b>>1) << std::endl; // 50 = 110010
std::cout << "(b>>2) = " << (b>>2) << std::endl; // 25 =  11001
std::cout << "(b>>3) = " << (b>>3) << std::endl; // 12 =   1100
```

Exemple : `Exple_20_operateurs`

C/C++ : opérateurs logiques et mathématiques (4)

Opérateurs « relationnels » :

Opérateur	Signification	Arité	Associativité
<	*Less than* Inférieur à	binaire	gauche à droite
>	*Greater than* Supérieur à	binaire	gauche à droite
<=	*Less than or equal to* Inférieur ou égal à	binaire	gauche à droite
>=	*Greater than or equal to* Supérieur ou égal à	binaire	gauche à droite

Précisions :

En C++, ces opérateurs retournent un type **`bool`** (valeurs **`true`** ou **`false`**), contrairement **au C** où ils retournent un type **`int`** (valeurs **`0`** ou **`1`**).

ONERA

Note :

Attention à l'associativité :

« `a<b<c` » ne signifie pas « `a` plus petit que `b` et `b` plus petit que `c` »

mais : « le résultat de `(a<b)` plus petit que `c` » :

```
a<b<c      EQUIVALENT A       (a<b)<c
           DIFFERENT DE     (a<b)&&(b<c)
```

C/C++ : opérateurs logiques et mathématiques (5)

Opérateurs « d'équalité » :

Opérateur	Signification	Arité	Associativité
==	*Equality* Egalité	binaire	gauche à droite
!=	*Inequality* Inégalité	binaire	gauche à droite

Précisions :

En C++, ces opérateurs retournent un type **`bool`** (valeurs **`true`** ou **`false`**), contrairement **au C** où ils retournent un type **`int`** (valeurs **`0`** ou **`1`**).

Erreur classique : ne pas confondre l'opérateur d'égalité (==) avec l'opérateur d'affectation (=), cf exemple.

ONERA

Note :

Les programmeurs C/C++ ont souvent l'habitude de faire des affectations « à la volée ») dans un test logique. Par exemple en C pour ouvrir un fichier et le tester :

```
FILE* f;
if ( (f = fopen("NomFichier.xxx", "r")) != NULL)
{
   // Lire le contenu du fichier
}
```

La fonction « `fopen` » retourne « `NULL` », c'est-à-dire zéro (`0`).

On peut aussi rencontrer directement :

```
FILE* f;
if ( !(f = fopen("NomFichier.xxx", "r")) )
{
   // Lire le contenu du fichier
}
```

C/C++ : opérateurs logiques et mathématiques (6)

Opérateurs « sur les bits » :

Opérateur	Signification	Arité	Associativité
&	*Bitwise AND* ET binaire (bits)	binaire	gauche à droite
^	*Bitwise exclusive OR* OU exclusif binaire (bits)	binaire	gauche à droite
|	*Bitwise OR* OU binaire (bits)	binaire	gauche à droite

Précisions :

Ces opérateurs travaillent sur les bits, comme les décalages (**<<** et **>>**).

Ils sont assez peu employés (surtout dans un contexte scientifique). Attention à ne pas les confondre avec les opérateurs logiques.

ONERA

Exemple :

```
a = 14;  // 00001110 en code binaire
b = 100; // 01100100 en code binaire

std::cout << (a & b) << std::endl;  // 4   = 00000100
std::cout << (a | b) << std::endl;  // 110 = 01101110
std::cout << (a ^ b) << std::endl;  // 106 = 01101010
```

Exemple : `Exple_20_operateurs`

C/C++ : opérateurs logiques et mathématiques (7)

Opérateurs « logiques » :

Opérateur	Signification	Arité	Associativité
`&&`	*Logical AND* ET logique (bool)	binaire	gauche à droite
`||`	*Logical OR* OU logique (bool)	binaire	gauche à droite
`e1?e2:e3`	*Conditional* Condition if-else	ternaire	droite à gauche

Précisions :

Ces opérateurs travaillent sur le type **`bool`** (si ce n'est pas le cas, il y aura éventuellement une conversion implicite) et retournent un **`bool`**.

Le dernier opérateur signifie : « si `e1` est vraie alors faire `e2`, sinon faire `e3` »

ONERA

Notes :

Il vaut mieux user, voire abuser de paranthèses autour des expressions :

```
(a || b) && (c || (d && e))
```

Exemple :

L'opérateur « conditionnel » est utilisé pour réduire le code d'un *if-then-else* :

```
int max(int a, int b)
{
  return (a > b ? a : b);
}
```

Exemple : `Exple_20_operateurs`

C/C++ : opérateurs logiques et mathématiques (8)

Opérateurs « d'affectation » :

Opérateur	Signification	Arité	Associativité
=	*Assignment* Affectation	binaire	droite à gauche
*=	Multiplication et affectation	binaire	droite à gauche
/=	Division et affectation	binaire	droite à gauche
%=	Modulo et affectation	binaire	droite à gauche
+=	Addition et affectation	binaire	droite à gauche
−=	Soustraction et affectation	binaire	droite à gauche
<<=	Décalage à gauche et affectation	binaire	droite à gauche
>>=	Décalage à droite et affectation	binaire	droite à gauche
&=	ET binaire et affectation	binaire	droite à gauche
|=	OU binaire et affectation	binaire	droite à gauche
^=	OU exclusif binaire et affectation	binaire	droite à gauche

ONERA

Exemple :

Ces opérateurs sont des raccourcis:

```
int i;
i += 10;    // equivalent a : i = i + 10;
```

Exemple : `Exple_20_operateurs`

C/C++ : les structures de contrôle (1)

La structure conditionnelle « `if` » :

```
if (test)              // forme 1
     operation;

if (test)              // forme 2
     operation1;
else
     operation2;
```

Précisions :

- les parenthèses autour de `test` sont nécessaires
- `test` est en principe un **`bool`**, mais toute valeur non nulle est considérée comme vraie

ONERA

Précisions : `Exple_21_structurescontrole`

Si les opérations à exécuter sont multiples, il faut ajouter des accolades pour constituer un bloc :

```
if (a > 0)
    appel1(a);
    appel2(a);
else                  // error : illegal else without matching if
    appel3(a);
```

Le `else` se rattache toujours au `if` le plus proche qui n'a pas de partie `else` :

```
if (i == j)     /* premier if sans else  */
  if (k == j)   /* deuxieme if avec else */
  {
    . . .
  }
  else          /* else du deuxieme if   */
    . . .

if (i == j)     /* premier if avec else  */
{
  if (k == j)   /* deuxieme if sans else */
    . . .
}
else            /* else du premier if    */
  . . .
```

C/C++ : les structures de contrôle (2)

La boucle « `for` » :

```
for (initialisation; test; iteration)
    operation;
```

- `initialisation` est une instruction (ou un bloc d'instructions) exécutée avant le premier parcours
- `test` est une expression qui, si elle est vraie, détermine la fin de la boucle
- `iteration` est une instruction (ou un bloc d'instructions) qui est effectuée à chaque boucle

Forme « habituelle » en C++ :

```
for (int i=0; i < n; i++)
{
  // operations
}
```

« Boucle infinie » :

```
for (;;)
{
  // operations
}
```

ONERA

Page 64

Note :

En C++, on peut définir la variable « `i` » dans le `for`, ce que l'on ne peut faire en C. La portée de la variable est celle du `for`.

Algorithme équivalent :

```
initialisation;
debut :
  if (! test) goto fin;
  operation;
  iteration;
  goto debut;
fin :
```

Exemple : `Exple_21_structurescontrole`

C/C++ : les structures de contrôle (3)

Le « `while` » :

```
while (test)
    operation;
```

Mêmes précisions que pour le « if » :

- les parenthèses autour de `test` sont nécessaires
- `test` est en principe un **`bool`**, mais toute valeur non nulle est considérée comme vraie
- il faut mettre des accolades s'il y a plusieurs instructions.

ONERA

Exemple :

```
int somme=0, n=0;
while (somme < 1000)
{
   somme = somme + 2 * n / (5 + n);
   n = n +1;
}
```

Exemple : `Exple_21_structurescontrole`

C/C++ : les structures de contrôle (4)

Le « do-while » :

```
do
    operation;
while (test);
```

Mêmes précisions que pour le « `if` » et que pour le « `while` ».

Le « `do-while` » est semblable au « `while` », sauf sur un point : les instructions qui sont mentionnées dans « `operation` » sont exécutées au moins une fois.

ONERA

Exemple :

```
int p = n = 1;
do {
  p = p * n;
  n = n + 1;
}
while (n != 10);
```

Exemple : `Exple_21_structurescontrole`

C/C++ : les structures de contrôle (5)

Le branchement conditionnel « `switch / case` » :

```
switch (valeur)
{
   case cas1:
     operations1;
     break;            // Facultatif
   …
   case casN:
     operationsN;
     break;            // Facultatif

   default:            // Facultatif
     operationsDefaut;
}
```

Utile pour éviter les « `if – else` » successifs.

ONERA

Notes :

Cette structure de contrôle s'apparente à un « goto » : le programme saute jusqu'au cas dont la valeur correspond. Cela signifie que toutes les instructions après le saut sont exécutées, y compris celles des cas qui suivent.

C'est la raison d'être du « break », facultatif, qui permet d'éviter ce comportement et de sortir de la strcuture conditionnelle.

Exercice : `Exple_21_structurescontrole`

Selon les valeurs de « `k` » supposé être du type `int`, déterminez quelles instructions seront appelées :

```
switch (k) {
  case 1:
  case 2:
    operationA;
  case 3:
    operationB;
    break;
  case 4:
    operationC;
  default:
    operationD;
}
```

C/C++ : les structures de contrôle (6)

Les ruptures de séquence :

```
continue;
```

Saute à la fin du bloc d'instructions pour un `for`, `do` ou `while`

```
break;
```

Permet de sortir du `for`, `do` ou `while` et de continuer après

```
return;
```

```
return valeur;
```

Quitte la fonction en cours

ONERA

Exemple :

```
for (i=0, j=0; ((i<n)&&(i<j)); i++, j=i%3)
{
  if (j > 3)
  {
    . . .
    continue;
  }
  else if (i+j > 100)
    break;
  . . .
}
```

Le saut GOTO :

Le « goto » existe en C et C++. Il est à proscrire, surtout en C++ où il est incompatible avec la gestion des exceptions et où il pose des problèmes avec la portée des variables.

Néanmoins, il peut être indispensable dans certains cas précis :

```
goto etiquette;
```

où `etiquette` est une ligne marquée de la façon suivante :

```
etiquette:
```

C : les énumérations

Une énumération en C est un type particulier qui associe des **symboles** constants à des **entiers** :

```
enum jour { lun, mar, mer, jeu, ven, sam, dim };
```

Une énumération définit un nouveau type "`enum jour`" que l'on peut utiliser pour déclarer des variables :

```
enum jour j = lun;
```

La numérotation est automatique et commence à zéro, ainsi on aurait pu écrire à la place :

```
enum jour j = 0;    // equivalent a "lun"
```

ONERA

La numérotation :

L'utilisateur peut affecter des numéros manuellement, le compilateur complétant :

```
enum poisson {carpe=1, truite=0, requin, raie};
```

Définition de variables :

On peut avec la même syntaxe, définir des variables lors de la définition de l'énumération :

```
enum jour { lun, mar, mer, jeu, ven, sam, dim } j1, j2, j3;
```

Exemple : `Exple_22_enumC`

C++ : les énumérations

En C++, une énumération est un véritable type, différent du type **int**. Un type énuméré ne peut prendre de valeur numérique :

```
enum jour { lun, mar, mer, jeu, ven, sam, dim };
enum sign { neg, pos, nul };

enum jour j = 1;      // légal en C, pas en C++
enum sign s = lun;    // légal en C, pas en C++
```

Mais, pour des raisons de compatibilité, la conversion implicite de **enum** vers **int** est conservée :

```
int i = mer;     // conversion implicite enum -> int
                 // i prend la valeur 2

void f(jour j);      // On peut alors exploiter
void f(sign s);      // sans problème la surcharge
void f(int i);       // des noms de fonctions
```

ONERA

Exemple : `Exple_23_enumCPP`

C : conversions de types

Dans certains cas, le compilateur peut être amené à effectuer certaines conversions lors des évaluations pour obtenir un exécutable correct :

```
int i, j, k;
float x;

x = i;
j = k+x;
```

On peut laisser faire certaines conversions implicites, mais il faut mieux apprendre à les maîtriser soi-même.

Pour forcer une conversion : `(type) expression;`

```
int i, j, k;
float x;

x = (float) i;
j = k + (int) x;   /* On peut utiliser aussi 'floor'
                      et 'ceil' pour transformer un
                      float en entier */
```

ONERA

Précision :

On appelle **transtypages** ces opérations de conversions de types. Ceux **implicites**, sont réalisés par le compilateur, les **explicites** sont réalisés par le programmeur. En anglais on parle aussi de **cast**.

C++ : conversions de types

C++ possède une nouvelle syntaxe pour le *transtypage* explicite, bien que l'opérateur *cast* du langage C soit toujours valable, et même indispensable dans certains cas :

En C

```
int i=1;
float x=3.14;

i = (int) x;
x = (float) i;
```

En C++

```
int i=1;
float x=3.14;

i = int(x);          // Nouvelle syntaxe
x = float(i);

i = (int) x;         // Toujours valable
x = (float) i;

ptr1 = char*(ptr2); // INCORRECT
ptr1 = (char*)ptr2; // L'ancien opérateur
                    // est ici obligatoire
```

ONERA

Page 72

Note :

L'opération de **cast** a beaucoup de défauts, notamment celui d'utiliser les parenthèses, très exploitées dans le langage, ce qui rend l'opération de transtypage difficilement repérable dans un code.

Les opérateurs **`static_cast`**, **`reinterpret_cast`**, **`const_cast`** et **`dynamic_cast`** ont été introduits afin de rendre plus explicites et moins sournois les transtypages :

Exemple d'utilisation de : `static_cast<T>(exp)`

```
long n = 12345L;
int k = static_cast<int>(n);
enum Direction {haut, bas, droite, gauche};
Direction d = static_cast<enum Direction>(2);
```

Exemple : `Exple_24_transtypages`

C : fonctions et procédures (1)

Un programme en langage C est constitué uniquement, en dehors des définitions et déclarations de types, par des fonctions/procédures.
Y compris le programme principal qui s'appelle **`main`**.

```
type_resultat  nom_fonction (type1 par1, ..., typeN parN)
```

```
// Déclarations
// dites "Prototypes"
void f1 (void);
void f2 (int i);
void f3 (int i, double x);
void f4 ();
int  f5 (void);
int  f6 (int i);
int  f7 (int i, double x);
int  f8 ();
```

```
// Définition
// "Corps de la fonction"
int f6 (int i)
{
  /* calcule le ième
     nombre premier */
}
```

ONERA

Page 73

Précisions :

La différence entre fonctions et procédures n'existe pas en C/C++. Pour moi une procédure (subroutine en Fortran) est une fonction qui ne retourne pas de valeur.

- `f1` : fonction qui ne prend pas d'argument et ne retourne rien, « procédure »
- `f2` : fonction qui prend un argument entier et ne retourne rien, « procédure »
- `f3` : fonction qui prend deux arguments et ne retourne rien, « procédure »
- `f4` : fonction qui prend un nombre quelconque d'arguments (en C seulement), « procédure »
- `f5` : fonction qui ne prend pas d'argument et retourne un entier
- `f6` : fonction qui prend un argument entier et retourne un entier
- `f7` : fonction qui prend deux arguments et retourne un entier
- `f8` : fonction qui prend un nombre quelconque d'arguments et retourne un entier

Fonctions internes :

En C/C++, on ne peut définir une fonction à l'intérieur d'une autre.

C : fonctions et procédures (2)

```c
void reset (double t[], int n)
{
  int i;
  for (i=0; i<n; i++)
    t[i]=0.0;
}


/* PROGRAMME PRINCIPAL */
int main(void)
{
  double tab[100];

  reset(tab, 100);
  ...
  reset(tab, 50);
  ...
  return 0;  /* OK */
}
```

```c
int maximum (int i, int j)
{
  if (i>j)
    return i;
  else
    return j;
}

/* PROGRAMME PRINCIPAL */
int main(void)
{
  int i=10;
  int j=0, k1, k2;
  k1 = maximum(i, j);
  ...
  k2 = maximum(k1, 100);
  ...
  return 0;  /* OK */
}
```

ONERA

Valeur de retour :

Le mot-clé « `return` » suivi de la valeur à retourner, ou seul, s'il n'y a pas de valeur de retour, permet de sortir de la fonction. On peut l'utiliser avec ou sans parenthèses :

`return val;` OU `return(val);`

Il peut y avoir plusieurs « `return` » dans la fonction, mais il faut que toutes les branches conditionnelles sortent avec une valeur du type attendu :

```c
double racine(double x)
{
  if (x > 0)
    return sqrt(x);
  else
    std::cout << "ERREUR";      // warning 'racine' : not all
}                                // control paths return a value
```

Précisions :

• « `reset` » est une procédure. Elle modifie ses arguments (ici les valeurs du tableau « `t` » et ne retourne pas de valeur.

• « `maximum` » est une fonction. Elle ne modifie pas ses arguments et retourne une valeur.

Exemple : `Exple_25_fonctionsprocedures`

C : fonctions et procédures (3)

En C, tous les arguments sont passés valeur.

La fonction ne travaille pas directement sur l'argument réel, mais sur une *copie locale* à la fonction, définie implicitement par le compilateur et initialisée à la valeur de l'argument réel.

```
void permute (int i, int j)
{
  int k = j;
  j = i;
  i = k;
}
```

Est-ce que ça marche à votre avis ?

ONERA

Précisions :

La règle « *En C, tous les arguments sont passés par valeur* » n'admet pas de dérogation en langage C. Ce que l'on appelle « passage par pointeur » n'échappe pas à cette règle (cf. planche suivante).

C : fonctions et procédures (4)

Pour modifier une variable, il faut alors passer son adresse.

On travaille alors sur la valeur de l'adresse, pas sur la valeur de la variable.

```
void reset (double t[], int n)
{
  /* ... */
}
```

On passe l'adresse du début du tableau `t[]`.

```
void permute (int *i, int *j)
{
  int k = *j;
  *j = *i;
  *i = k;
}
```

On passe les adresses de `i` et de `j` et on accède à leurs valeurs par l'opérateur `*`.

ONERA

Les pointeurs :

Un pointeur c'est une adresse (un nombre 32 bits par exemple, sur une machine dont l'architecture interne est 32 bits).

Ni plus, ni moins, **un nombre**.

C : fonctions et procédures (5)

```
void permute (int *i, int *j)
{
  int k = *j;
  *j = *i;
  *i = k;
}


int main(void)
{
  int i = 1, j = 2, k = 3;

  permute(&i, &j);
  permute(&j, &k);

  return 0;
}
```

L'opérateur `&` appliqué à un nom de variable retourne son adresse.

Quelles sont les valeurs de `i`, `j` et de `k` à la fin ?

ONERA

Page 77

Les pointeurs :

Démonstration qu'un pointeur est une adresse :

```
double t[10];
t[0] = t[1] = 100.0;
std::cout << "Adresse de t    = " << t << std::endl;
std::cout << "Adresse de t[0] = " << &(t[0]) << std::endl;
std::cout << "Adresse de t[1] = " << &(t[1]) << std::endl;
```

Ce programme produit :

```
Adresse de t    = 0012FF2C
Adresse de t[0] = 0012FF2C
Adresse de t[1] = 0012FF34
```

Exemple : `Exple_25_fonctionsprocedures`

C vs C++ : prototypes de fonctions

Une différence entre le C et le C++ concerne les prototypes des fonctions qui sont déclarées sans paramètre :

- en C : la fonction peut être appelée avec n'importe quel argument ;
- en C++ : la fonction n'a effectivement pas de paramètres.

En C

```
int f();   /* paramètres
              quelconques */

f();            /* Légal */
f(1);           /* Légal */
f(1, 2);        /* Légal */

int g(void);    /* pas de
                   paramètre */
g();            /* Légal */
g(1);           /* ILLEGAL */
```

En C++

```
int f(); // pas de paramètre

f();            // Légal
f(1);           // ILLEGAL
f(1, 2);        // ILLEGAL

int g(void);    // pas de
                // paramètre
g();            // Légal
g(1);           // ILLEGAL
```

ONERA

Page 78

Rappel :

1) Toute fonction doit avoir été déclarée (pas forcément définie) au moment de son utilisation.

2) Il peut y avoir plusieurs déclarations, il n'y a qu'une définition.

Nombre d'arguments quelconques :

En C, on peut manipuler des fonctions avec un nombre quelconque d'arguments. On fait appel à des macros (`va_start`, `va_arg`, etc.). Cette notation est à éviter en C++ qui propose au développeur deux techniques plus sures : surcharge des noms de fonctions et arguments par défaut.

La manipulation de ces macros est d'une complexité extrême, ne faisant pas partie du « noyau » du langage, mais de librairies annexes, elles ne sont pas normalisées et dépendent des compilateurs.

C++ : arguments par défaut

En C++, il est possible de définir des arguments par défaut pour une fonction. Ces arguments doivent être à la fin de la liste ou ne doivent être suivis que d'autres paramètres ayant aussi une valeur par défaut :

```
void f (double x, int i=0);                    // Légal
void g (double x, int i=0, char *s="");        // Légal
void h (double x=0.0, int i);                  // ILLEGAL
```

Lors de l'appel de la fonction, l'argument optionnel doit être le dernier de la liste (tous les suivants éventuels ayant aussi été omis) :

```
g(3.14159, 10, "Jacques");   // Légal
g(3.14159, 10);              // Légal
g(3.14159);                  // Légal
g(3.14159, "Jacques");       // ILLEGAL
```

ONERA

Page 79

Note :

Les arguments par défaut sont très utiles :

```
double MethodeSimpson (Function f,
                       double a, double b,
                       double dx = 0.1);
```

Où « `Function` » est un type « pointeur vers une fonction qui prend un double et rend un double » :

```
typedef double (*Function)(double);
```

L'utilisateur peut appeler :

```
integrale = MethodeSimpson(sin, 0, 1);
```

aussi bien que :

```
integrale = MethodeSimpson(sin, 0, 1, 0.01);
```

Exemple : `Exple_26_defautArguments`

Exercice facultatif : `Exple_27_methodeSimpson`

Programmer la méthode de Simpson et la tester sur quelques exemples. Pour mémoire : l'intégrale entre x et x+dx de la fonction f(x) est évaluée par la méthode de Simpson :

$$I = (1/6\ f(x) + 2/3\ f(x+dx/2) + 1/6\ f(x+dx)) * dx$$

Vous aurez probablement besoin d'inclure la librairie standard du C : `#include <cmath>`

C++ : fonctions « en ligne » (1)

Les fonctions en ligne (**inline**) sont expansées à chaque appel par le compilateur qui incorpore à l'endroit de l'appel les instructions du corps de la fonction sans générer les séquences habituelles d'entrées-sorties :

```
inline double norm (double x, double y)
{
  return sqrt(x*x + y*y);
}
```

Les fonctions en ligne sont intéressantes pour les petites fonctions.

```
#define NORM(x, y) sqrt((x)*(x)+(y)*(y))

NORM(++z1, z2);  // Problème
```

Elles remplacent les macros du préprocesseur C, les effets de bord en moins.

ONERA

Précisions :

Les appels avec la macro ne donnent pas le bon résultat, ils sont en tout cas différents de la valeur attendue, correcte avec la fonction en ligne. On rappelle pour mémoire que « `++z1` » signifie que l'on incrémente la variable, puis qu'on l'utilise.

Exercice : `Exple_28_methodesInline`

Essayer de comprendre pourquoi :

```
double z1 = 2;
double z2 = 4;

// AFFICHE 5 (racine de 25)
std::cout << norm(++z1, z2) << std::endl;
z1 = 2;
z2 = 4;

// AFFICHE 5.2915 (racine de 28)
std::cout << NORM(++z1, z2) << std::endl;
```

C++ : fonctions « en ligne » (2)

La portée d'une fonction en ligne est limitée au fichier dans lequel elle est définie.

Pour partager une fonction en ligne entre plusieurs fichiers sources, on doit la définir dans un fichier en-tête commun "`.h`".

Le mot-clé **`inline`** n'est qu'indicatif pour le compilateur, celui-ci peut en fait refuser de mettre en ligne certaines fonctions :

- fonctions récursives ;
- fonctions avec des **`switch`**, des **`goto`** ;
- fonctions trop volumineuses ;
- fonctions dont l'adresse doit être utilisée ;
- etc.

ONERA

Exercice : `Exple_28_methodesInline`

1) Ecrire une fonction en ligne qui soit équivalente à la macro :

```
#define ABS(x) ((x)>0 ? (x) : (-x))
```

On utilisera l'opérateur conditionnel : `(cond_bool)?(val_si_True):(val_si_False)`

2) Comparer le comportement de la macro et de la fonction en ligne :

- au niveau du typage : `int abs(int i)`, `double abs(double d)`
- des effets de bord : `abs(i++)`, `abs(x++)`, ...

Note :

Il y a un avantage à la macro « `ABS` » : elle marche pour tous les types (entiers, réels, …) pour lesquels les opérateurs de comparaison (>) avec un entier et de négation unaire (−) existent.

La fonction en ligne, elle, ne marche que pour la signature prévue. Pour avoir le même avantage de généricité, nous verrons qu'il faut utiliser les "*templates*" ou fonctions génériques.

C++ : surcharge des noms de fonctions (1)

Le C++ offre la possibilité d'avoir plusieurs fonctions de même nom. Les fonctions doivent se différencier par le nombre et/ou le type de leurs arguments :

En C

```
int ipower(int i, int j)
{
  /* i^j = i*i*i* ... *i; */
}

double fpower(double x,
              double y)
{
  /* x^y = e^(y*Log(x)); */
}

... = ipower(2, 16);
... = fpower(3.14, -0.5);
```

En C++

```
int power(int i, int j)
{
  ...
}

double power(double x,
             double y)
{
  ...
}

... = power(2, 16);
... = power(3.14, -0.5);
```

ONERA

C++ : surcharge des noms de fonctions (2)

La signature d'une fonction est l'ensemble des informations qui participent à la résolution de la surcharge de la fonction.

Le type de retour n'en fait pas partie :

```
void f(int i);
int f(int i);       // ILLEGAL
```

Des alias de types ne sont pas considérés comme des types distincts :

```
typedef int entier;
void g(int i);
void g(entier e);   // ILLEGAL

void h(char *s);
void h(char s[]);   // ILLEGAL
```

ONERA

Note :

Surcharge des noms de fonctions, *arguments par défaut* et *conversions implicites/explicites* sont des notions qui interviennent toutes dans le choix que fait le compilateur de la fonction à appliquer. Ce choix est effectué par l'*algorithme de résolutions des surcharges* (cf. planches suivantes).

C++ : algorithme de résolution des surcharges (1)

1) Le compilateur cherche une fonction dont les types concordent parfaitement :

```
void f(int i);
void f(double d);

f(1);          // appelle f(int)
f(3.14);       // appelle f(double)
```

2) Le compilateur recherche les conversions implicites :

- "promotion" des entiers : `char -> short -> int -> long`
- "promotion" des réels : `float -> double`

```
void f(int i);
void f(double d);

f('x');        // appelle f(int)
f(1.0F);       // appelle f(double)
```

ONERA

C++ : algorithme de résolution des surcharges (2)

3) Le compilateur recherche à appliquer d'autres conversions standards, qui comportent un risque (`int -> float`, `unsigned long -> int`)

```
void f(char *c);
void f(int i);

f(1.2345);    // appelle f(int)  !!!!
```

4) Le compilateur recherche des transtypages définis par l'utilisateur

5) Le compilateur recherche une fonction de même nom avec un nombre indéterminé de paramètres :

```
void f(char *c);
void f(double d, ...);

f(1.2345, 3.14159); // appelle f(double, ...)
```

ONERA

C++ : algorithme de résolution des surcharges (3)

6) Sinon, ERREUR

Il y a erreur aussi si plusieurs fonctions concordent :

```
void f(long l);
void f(float x);

f(1.0F);        // OK
f(12345L);      // OK
f(1.0);         // ERREUR
f(12345);       // ERREUR
```

Des erreurs similaires peuvent provenir de l'utilisation de paramètres par défaut :

```
void f(int a, int b, int c=3);
void f(int d, int e);

f(1, 2);        // ERREUR
```

ONERA

Exercice : Arguments par défaut et surcharge de noms de fonctions

Le programme suivant est-il compilable ? Si oui, ou en supposant que l'on met en commentaire les lignes fautives, que fait-il ? Donner des justifications.

```
double f(int i=0, double x=0.0)  {return (double)i * x;}
double f(int i)  {return 10.0*i;}
double f(double x)  {return 100.0*x;}
int main()
{
  std::cout << f(5, 2.0) << std::endl;
  std::cout << f(5) << std::endl;
  std::cout << f(2.0) << std::endl;
  std::cout << f() << std::endl;
  return 0;
}
```

Exercice : `Exple_29_surchargeFonctions`

C++ : surcharge ou arguments par défaut ?

Conseil : utiliser les arguments par défaut quand l'algorithme est le même pour toutes les variantes de la fonction ou quand une valeur évidente par défaut existe.

```
// Calcul de la matrice de passage
// 6 fonctions trigonométriques, 16 multiplications, ...
DMatrix &euler_to_matrix(double psi,
                         double theta, double phi);
```

Version 1

```
// On ajoute un argument par défaut (angle de roulis nul)
// Sans modifier l'algorithme
DMatrix &euler_to_matrix(double psi,
                         double theta, double phi=0.0);
```

Version 2

```
// On fait une deuxième version simplifiée de la fonction
// 4 fonctions trigonométriques, 4 multiplications, ...
DMatrix &euler_to_matrix(double psi,
                         double theta, double phi);
DMatrix &euler_to_matrix(double psi, double theta);
```

Version 3

ONERA

Page 87

Précisions :

- « `DMatrix` » est un type (= classe) que le programmeur a défini pour représenter des matrices réelles
- « `euler_to_matrix` » est la fonction de calcul de la matrice de passage à partir de trois angles :
  - psi : angle de lacet
  - theta : angle de tangage
  - phi : angle de roulis

C/C++ : les pointeurs (1)

Tous les éléments d'un programme sont stockés en mémoire : variables, constantes, mais aussi fonctions, instances (de classes), etc. Leur position en mémoire est une adresse.

Les pointeurs sont des variables dont la valeur est une adresse.

On dit que le pointeur « pointe » vers l'élément en question.

Fondamentaux à comprendre :

- changer la valeur d'un pointeur : c'est pointer vers autre chose
- on doit donner le type de l'objet pointé :

```
int* pi;     // pointeur vers entier
double** t; // pointeur vers pointeur (de double)
```

- l'opérateur « adresse de » (**&**) permet d'obtenir l'adresse d'un élément
- l'opérateur d'indirection (*****) permet d'accéder à l'élément pointé
- il existe une valeur spéciale : le pointeur « nul », dénoté **NULL**

ONERA

Précisions :

Tous les pointeurs ont la même taille quel que soit l'objet pointé. Par exemple sur une machine 32 bits, ils ont tous cette taille.

Il faut apprendre à lire les opérateurs adresse et d'indirection :

- **&** : « adresse de … »
- ***** : « valeur pointée par le pointeur … »

Initialisation des pointeurs :

C'est une bonne pratique d'initialiser les pointeurs dès leur création. Le compilateur définit le pointeur nul par une macro « NULL ». Cela permet au programmeur (et au compilateur) de tester la validité d'un pointeur avant son utilisation.

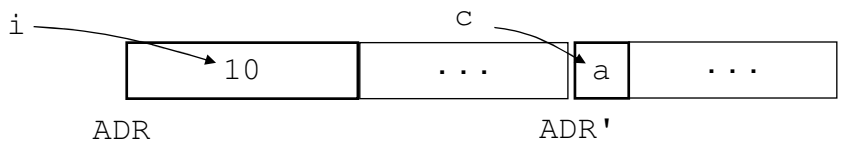
Exemple :

```
int* pi = NULL;
double** t = NULL;
```

C/C++ : les pointeurs (2)

Exercice :

```
int  i = 10;
char c = 'a';
```



Que valent les entités suivantes dans le programme et quelle est leur nature ?

	valeur	nature
i		
&i		
c		
&c		

ONERA

Page 89

Attention, source d'erreur :

Il faut faire attention quand on déclare plusieurs pointeurs à la suite :

```
int *p1, p2;              // p1 est un pointeur, p2 est un entier
int *p1, *p2;             // p1 et p2 sont des pointeurs
int *p1=NULL, *p2=NULL;   // ils sont initialisés
```

Exercice facultatif : `Exple_30_pointeurs`

Déclarer les deux variables de l'exercice et afficher leurs valeurs et leurs adresses.

C/C++ : les pointeurs (3)

Pour déclarer un pointeur : précéder le nom de la variable du caractère ***** :

```
int  *pi;          /* pointeur sur un entier              */
struct point *p;   /* pointeur sur une structure          */
TRUC *t1;          /* pointeur sur un TRUC                */
int **ppi;         /* pointeur sur pointeur vers entier */
double ***pppx     /* ...                                 */
pi = NULL;         /* Prudent de toujours initialiser     */
p = NULL;          /* a la valeur NULL : "synonyme de     */
                      ne pointe sur rien"                  */
```

Déclarer un pointeur ne revient pas à déclarer la variable sur laquelle il pointe.

Il faut que l'adresse pointée corresponde à une variable ou à une constante déjà existante si on veut travailler sur sa valeur. Sinon, il faut faire de l'**ALLOCATION DYNAMIQUE**.

ONERA

Précision :

Il y a donc toujours deux temps à respecter avant d'utiliser un pointeur (comme pour toute variable, d'ailleurs) :

```
int *p1;              // JE DECLARE
p1 = &i;              // JE DONNE UNE VALEUR

int *p2;              // JE DECLARE
p2 = new int();       // JE DONNE UNE VALEUR (VERSION AVEC
                      // ALLOCATION DYNAMIQUE)
// ...
delete p2;            // MAIS IL FAUDRA LIBERER LA MEMOIRE
```

Le danger vient que l'on manipule ici des adresses (et que dans un ordinateur, tout a une adresse : un pixel de l'écran, le disque dur et ses variables, etc.). Un pointeur non initialisé pointe vers n'importe quoi (ce qui provoque des erreurs « aléatoires », non reproductibles, qui plongent le néophyte dans des abimes de perplexité).

C/C++ : les pointeurs (4)

Exercice

Soit les déclarations : `int a, *p;`

Que vous inspirent les opérations d'affectation suivantes ? On se posera (au moins) les questions suivantes : sont-elles cohérentes vis-à-vis des types ? Manipule-t-on des valeurs bien définies ?

```
a  = *p;

*p = &a;

*p = *(&a);

a  = p;

a  = (int)p;
```

ONERA

Réponses : Exple_31_pointeurs

```
a  = *p;

*p = &a;

*p = *(&a);

a  = p;

a  = (int)p;
```

C/C++ : pointeurs, indirection et déréférencement

Exercice Soit le code suivant :

```
int i, j, *pi, *pj;
i = 1;
j = 2;

 pi = &i;
 pj = &j;
*pi = 3;
*pj = *pi;
*pi = 4;
 pj = pi;
*pi = 5;
```

Peut-on faire :
`*pi = 3;`
`*pj = i;`
Pourquoi ?

Faire un graphique représentant l'état de la mémoire à chaque étape.

Quelles sont les lignes qui modifient la valeur des variables `i` et `j` ?

Combien valent `i`, `j`, `pi`, `pj`, `*pi`, `*pj` à la fin ?

ONERA

Corrigé de l'exercice : `Exple_32_pointeurs`

C/C++ : « arithmétique » des pointeurs

Il est possible d'effectuer des opérations arithmétiques avec les pointeurs : addition et soustraction avec un entier, soustraction entre deux pointeurs.

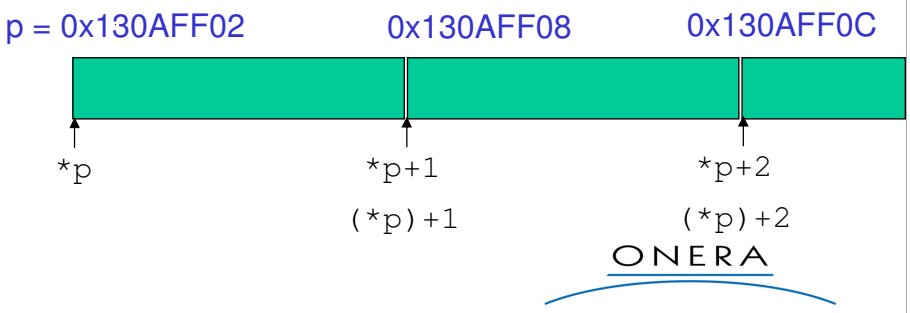
Si « **p** » est un pointeur vers un type « **T** » et « **i** » un entier :

p + i = adresse de **p**, décalée de **i** objets de taille **sizeof(T)**

Quand on fait « **p+1** » ou « **p++** », on se déplace en mémoire à l'adresse de l'objet n°2 de type **T** qui serait contigu à l'objet n°1 pointé par «**p** », s'ils étaient l'un après l'autre en mémoire. Et cela, même si l'objet n°2 n'existe pas !

Exemple :

Objets de type **T**
Sizeof(T) = 6



ONERA

Exemple : `Exple_33_arithmetiquePointeurs`

C/C++ : pointeurs et tableaux (1)

Dans les deux déclarations :

```
int a1[10];
int *a2;
```

`a1` et `a2` sont du même type, pointeur sur un entier. Mais il y a une différence fondamentale :

`a1` est un pointeur **constant**. La déclaration réserve un emplacement mémoire pour 10 entiers, `a1` pointe réellement sur quelque chose et sa valeur ne peut être modifiée :

`a2` est un pointeur **variable**. Aucun emplacement mémoire n'est réservé pour des variables de type `int`, la valeur de `a2` est indéfinie à l'origine, il est donc prudent de l'initialiser à `NULL`.

```
int a1[10];
int *a2;
a2 = NULL;   /* c'est prudent          */
...
a2 = a1;     /* a2 pointe sur les éléments de a1 */
a1 = a2;     /* ILLEGAL, a1 est constant     */
```

ONERA

C/C++ : pointeurs et tableaux (2)

A l'usage, les deux déclarations permettent de manipuler le tableau de valeurs de manières identiques :

```
void reset (double t[],
            int n)
{
  int i;
  for (i=0; i<n; i++)
    t[i]=0.0;
}
```

```
void reset (double *t,
            int n)
{
  int i;
  for (i=0; i<n; i++)
    t[i]=0.0;
}
```

```
int main(void)
{
  double tab[100];
  reset(tab, 100);
  ...
  reset(tab, 50);
  ...
  return 0;   /* TOUT A ETE OK */
}
```


ONERA

C/C++ : pointeurs et chaînes de caractères (1)

On peut déclarer une chaîne de caractères de deux façons différentes :

```
/* Tableaux de
   caractères */

char chaine1[100];
char chaine2[] = "Hello\n";
```

`chaine1` et `chaine2` sont des pointeurs **constants**, on peut modifier le contenu du tableau mais pas son adresse.

```
/* ILLEGAL */
chaine2 = chaine1;
```

```
/* Pointeur vers UN
   caractère */

char *chaine3;
```

`chaine3` est un pointeur **variable**, on peut le diriger vers le début d'une éventuelle chaîne de caractères.

```
/* LEGAL */
chaine3 = chaine1;
```

ONERA

Précisions :

On peut initialiser la variable « `chaine3` » :

```
char* chaine3 = "bonjour";
```

Si on veut avoir un comportement presque égal entre les deux types de déclarations, il faut déclarer des pointeurs constants en utilisant le mot-clé « **`const`** » :

```
char *const chaine3 = "bonjour";
```

Exercice facultatif : `Exple_35_pointeursEtChaines`

Tester les réactions du compilateur sur les exemples ci-dessus. Observer warnings et erreurs.

C/C++ : pointeurs et chaînes de caractères (2)

Mais pour accéder aux valeurs, tout se passe comme s'il s'agissait de tableaux de caractères :

```
char chaine1[100];
char chaine2[] = "Hello\n";
char *chaine3;

chaine3 = chaine2;

chaine1[0] = ...;

... = chaine2[4];   /* Le caractère 'o'  */
... = chaine2[5];   /* Le caractère '\n' */
... = chaine2[6];   /* Le caractère '\0' */

... = chaine3[4];   /* Le caractère 'o'  */
... = chaine3[5];   /* Le caractère '\n' */
... = chaine3[6];   /* Le caractère '\0' */
```

ONERA

C/C++ : définition des structures (struct) (1)

En C, les structures (mots-clés `struct`) permettent de créer de nouveaux types, complexes, en assemblant d'autres types :

```
struct point
{
  double x, y;
  int numero;
};
```

Définit un nouveau type « `struct point` » avec trois champs

On accède aux différents champs en utilisant une notation « pointée » :

```
struct point p1 = {0.0, 0.0, 1}, P[100];
point p2;
p2.numero = ...    ;
p1.x      = ...    ;
...       = P[50].x ;
```

En C++, le « `struct` » est facultatif

ONERA

Page 98

Notes :

1) Il existe d'autres types complexes, les unions, mot-clé « `union` ». Ce sont des types variables, polymorphes, assez peu employés.

2) La syntaxe permet de définir des variables au moment de la définition de la structure :

```
struct point
{
  double x, y;
  int numero;
} p1, p2;      // Un type et deux variables : p1 et p2
```

3) Il n'est même pas obligatoire de donner un nom à la structure :

```
struct {
  double x, y;
  int numero;
} p1, p2;      // Seulement deux variables : p1 et p2
```

Exemple : `Exple_36_struct`

C/C++ : définition des structures (struct) (2)

```
struct point
{
  double x, y;
  int numero;
};

typedef struct point Point;

Point p3, tabp[100], *pp;

p3.numero = ...          ;
...       = tabp[50].x ;
pp        = &tabp[30];
pp->x     = 10.0;
...       = pp->numero;
(&p3)->x  = 20.0;
...       = (&p3)->x;
```

Lorsque la structure est référencée par un pointeur, on utilise une notation « fléchée » pour accéder aux champs à partir du pointeur

ONERA

Exercice : `Exple_37_structEtPointeurs`

- Créer un nouveau projet et définir la classe « `Point` » comme dans l'exemple ci-dessus (mais avec une majuscule, il vaut mieux prendre de bonnes habitudes C++).
- Définir une structure « `Segment` » comportant deux champs `p1` et `p2` qui sont des pointeurs vers le type « `Point` ».
- Définir deux variables `a` et `b` de type « Point », une variable `s` de type « `Segment` » et affecter aux champs de `s` les adresses de `a` et `b`.
- Définir un pointeur `ps` vers la variable `s`. Etablir la syntaxe qui permet de manipuler les adresses de `a` et de `b`, depuis le pointeur `ps`.
- En utilisant la notation pointée et la notation fléchée : vous devez trouver 8 façons différentes de manipuler le « `numero` » du champ `p1` :

  - à partir de `s` ou de son adresse (`&`)

  - à partir de `ps` ou de la valeur pointée par `ps` (indirection `*`)

  - en utilisant le champ `p1` (pointeur) ou l'indirection depuis `p1`

  - …

C/C++ : les alias de noms de type (typedef)

Le mot-clé **typedef** permet de créer des synonymes pour les noms de types existants ou que le programmeur choisit de construire. L'intérêt est de rendre les programmes plus lisibles.

```
typedef struct point Point;    /* Alias de type structure          */

struct point p1;               /* les deux déclarations            */
Point p1;                      /* sont équivalentes                */

typedef int *Ptr_int;          /* Pointeur sur un int              */

typedef double Double10[10];   /* tableau de 10 doubles            */

typedef Double10 *PtrTab10;       /* pointeur sur tableau d'entiers
*/

typedef Ptr_int TabPtr10[10]; /* tableau de 10 pointeurs
                                 sur entiers                       */
```

ONERA

Page 100

Précisions :

En C++, on utilise moins le `typedef` qu'en C (voir pourquoi planche suivante). En fait on ne l'utilise que pour « débrouiller » les choses quand on déclare des types compliqués (*pointeur vers des tableaux de pointeurs vers des fonctions qui prennent en entrée un pointeur vers entier et retournent un double*).

Voir la « règle de l'escargot » pour ce qui est d'apprendre à lire les types compliquées syntaxiquement.

Par exemple, avec les types de la planche ci-dessus, les déclarations suivantes sont équivalentes :

```
            PtrTab10 ptr1;
et          int (*ptr1)[10];

            TabPtr10 tab1;
et          int *tab1[10];
```

Exemple : `Exple_38_typedefC`

C vs C++ : enum, struct et union

En C++, le nom d'une énumération, d'une structure ou d'une union est par lui-même suffisant pour identifier le type. L'ajout des mots clés `enum`, `struct` et `union` est facultatif :

En C

```
enum etype { /*...*/ };
struct stype { /*...*/ };
union utype { /*...*/ };

enum etype e;

typedef struct stype SType;
SType s;

void f(union utype *u);
```

En C++

```
enum EType { /*...*/ };
struct SType { /*...*/ };
union UType { /*...*/ };

EType e;

SType s;

void f(UType *u);
```

ONERA

Exemples : `Exple_38_typedefC` et `Exple_39_typedefCPP`

C++ : les références (1)

Un nouveau type dérivé a été introduit en C++ : le type référence. Si `T` est un type, alors `T&` est le type référence vers `T`.
Une référence est une adresse [1], mais il existe des différences fondamentales avec les pointeurs :

1) une référence doit toujours être initialisée lors de sa création (pas de valeur `NULL`). Elle peut être utilisée comme synonyme d'une variable existante ou pour donner un nom à un objet qui n'en a pas :

```
double x=5.0;
double t[10];

double &ref1; // ERROR : "references must be initialized"
double &ref2  = x;
double &first = t[0];
double &last  = t[9];
```

ONERA

Notes:

[1] Il vaut mieux voir en fait les références commes des « alias » des variables que comme des adresses ou des pointeurs. En Java, où paraît-il n'y a pas de pointeurs, ce que l'on appelle référence est encore différent : il s'agit en fait de pointeurs !

Exemple : `Exple_40_references`

C++ : les références (2)

2) une fois initialisée, une référence ne peut être changée :

```
double x, y;
double &ref = x;

&ref = y;          // ERREUR
&ref = &y;         // ERREUR
```

3) la syntaxe pour manipuler la valeur est la même que s'il s'agissait d'un objet "normal" :

```
int tab[5] = {0, 2, 4, 6, 8};
int &first = tab[0];

first++;           // Incremente tab[0]
... = first / 2;   // Division entière
```

ONERA

Exemple : `Exple_40_references`

C++ : les références (3)

4) Le type référence sur un `void` n'existe pas (contrairement à pointeur sur `void`), il est également impossible de créer un pointeur sur une référence, de créer une référence sur une référence ou un tableau de références :

```
typedef void& voidRef; // ERROR : Illegal use of type void
void *t;               // OK

double &*px = ...;     // ERROR : Pointer to reference is illegal
double *&px = pi;      // OK : Une réference sur un pointeur

double& &ref = ...;    // ERROR : Reference to reference is illegal
double **p;            // OK

typedef double& doubleRef;
doubleRef tab[5] = ...; // ERROR : Arrays of references are illegal
double *tab[5];         // OK
```

ONERA

Exemple : `Exple_40_references`

C++ : les références (4)

Les références permettent de réaliser un nouveau type de passage d'arguments, sans utiliser explicitement de pointeurs. Le gain en lisibilité et facilité d'utilisation est important :

En C

```
void swap(int *i, int *j)
{
  int k = *i;
  *i = *j;
  *j = k;
}

int main()
{
  int a=1, b=2;
  swap(&a, &b);
  ...
}
```

En C++

```
void swap(int &i, int &j)
{
  int k = i;
  i = j;
  j = k;
}

int main()
{
  int a=1, b=2;
  swap(a, b);
  ...
}
```

ONERA

Exemple : `Exple_41_passageReference`

C : passage des arguments

En C, tous les paramètres d'une fonction/procédure sont passés par valeur, on ne peut donc les modifier pour le compte de l'appelant.

Il existe deux cas où il faut (il est utile de) passer un pointeur sur ces paramètres :

1) pour modifier la variable pointée

```
void modifier (int *i)
{
  *i = ...
}

...
int k = 3;
modifier(&k);
...
```

2) si le paramètre à passer est un objet de grande taille (`struct`)

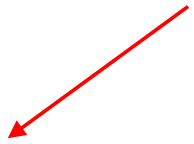
```
double norme_L1 (Point *p)
{
  return abs(p->x) + abs(p->y);
}

void initialiser (Point *p)
{
  p->x = 0.0;
  p->y = 0.0;
  p->numero = 0;
}
```

ONERA

Exemple : `Exple_41_passageReference`

C++ : passage par référence

En C++, on peut passer des références, avec la même efficacité qu'avec les pointeurs mais on peut aussi préciser si la variable référencée va être modifiée ou non (`const`) :



```
struct Point {
  double x, y;
  int numero;
};

double norme_L1 (const Point &p)  // REFERENCE CONSTANTE
{                                 // p ne peut être modifiée
  return abs(p.x) + abs(p.y);
}

void initialiser (Point &p)       // REFERENCE NON CONSTANTE
{                                 // p modifiable
  p.x = p.y = 0.0;
  p.numero = 0;
}
```

ONERA

Exemple : `Exple_41_passageReference`

C : fonctions, valeur de retour (1)

Lorsqu'une fonction retourne une valeur, le programme effectue une copie de cette valeur dans la variable du programme appellant.

Cette opération peut-être coûteuse pour les objets de grande taille.

```
int f (...)
{
  int tmp;
  ...
  return tmp;
}

void main(void)
{
  int i;
  i = f(...);   /* copie du resultat */
                /* de f dans i       */
  ...
}
```

Fin de la portée de la variable « tmp »

ONERA

Précisions :

En C/C++, les arguments des fonctions sont passés « par valeur ». Il en est de même pour le retour des fonctions : ce sont les valeurs qui sont retournées. Retourner une valeur signifie qu'on la copie dans la fonction appelante.

C : fonctions, valeur de retour (2)

```
typedef struct point Point;

Point calcul_1 (...)
{
  Point p;
  ...
  return p;  /* RECOPIE
                STRUCTURE */
}

Point *calcul_2 (...)
{
  Point *ptr_p;
  /* ALLOCATION
     DYNAMIQUE A FAIRE */
  ...
  return ptr_p;  /* RECOPIE
                    ADRESSE */
}
```

ATTENTION !!
PIEGE CLASSIQUE !!

Ne pas retourner l'adresse d'une variable locale qui n'a pas d'existence en dehors de la fonction.

```
Point *calcul_3 (...)
{
  Point p;
  ...
  return &p;  /* ERREUR!! */
}
```

ONERA

Page 109

Précisions :

Dans l'exemple « `calcul_1` », la variable « `Point p` » a une portée qui reste locale à la fonction. Une copie est faite lors du retour à l'appelant.

Dans « `calcul_3` », on doit retourner une adresse sur un point. Le problème est de retourner une adresse vers quelle chose de valide. Ici, on ne peut retourner l'adresse de la variable locale (qui n'est pas valable à l'extérieur).

La solution apparaît dans « `calcul_2` », on crée dynamiquement un nouvel objet, dont l'existance va perdurer au délà de la fonction. Et on retourne la valeur de l'adresse de cet objet.

C++ : valeur de retour de type référence

Les références permettent de créer des fonctions dont la valeur est une *lvalue*, c'est-à-dire une entité qui peut se trouver à gauche ("l" pour "left") d'un opérateur d'assignation :

En C

```
int x, y;

int *func(int b)
{
  return (b ? &x : &y);
}

*func(0) = 5; // met 5 dans y
*func(1) = 2; // met 2 dans x
```

En C++

```
int x, y;

int &func(int b)
{
  return (b ? x : y);
}

func(0) = 5; // met 5 dans y
func(1) = 2; // met 2 dans x
```

ONERA

Exemple : `Exple_42_retourReference`

C++ : toujours le piège classique !

En C, le renvoi d'un pointeur sur une variable locale est un piège classique. Ce piège subsiste encore avec le renvoi de référence.

Il y a deux solutions :

- renvoyer une référence sur un argument qui est lui-même référence
- allouer dynamiquement une variable et renvoyer sa référence

```
DMatrix &inverse(DMatrix &m)
{
  // on modifie la matrice
  ...
  return m;
}
```

```
DMatrix &inverse(const DMatrix &m)
{
  // on crée une matrice
  DMatrix *inv = new DMatrix(...);
  ...
  return *inv;
}
```

A l'appellant alors de détruire la référence lorsqu'elle devient inutile !!!

ONERA

C : allocation dynamique (1)

Comment faire si on doit fabriquer un nouveau "`point`" qu'il n'était pas possible de prévoir à la compilation ?

```
struct point
{
  double x, y;
  int numero;
};

typedef struct point Point;

Point *creer_point(double xx, double yy, int num)
{
  Point *p;
  p = (Point*) malloc(sizeof(Point));
  p->x = xx;
  p->y = yy;
  p->numero = num;
  return p;
}
```

C'est ce qu'on appelle un CONSTRUCTEUR

RAPPEL !!
Ne pas tomber dans le piège classique

ONERA

Exemple : `Exple_43_allocationC`

C : allocation dynamique (2)

Les étapes obligatoires de l'allocation dynamique :

- stocker le résultat de l'allocation (**malloc**) dans une variable de type pointeur
- effectuer un typage explicite
- utiliser **sizeof** pour connaître la quantité de mémoire nécessaire
- multiplier cette quantité par le nombre d'objets à allouer

```
/* Chaine de longueur 10                          */
chaine = (char*) malloc(10*sizeof(char));

/* tableau de 100 reels                           */
x1 = (float*) malloc(100*sizeof(float));

/* idem, mais les initialise à la valeur zéro */
x2 = (float*) calloc(100, sizeof(float)); /

/* Objet de type TRUC                             */
t1 = (TRUC*) malloc(sizeof(TRUC));
```

ONERA

C : allocation dynamique (3)

Toujours penser à libérer la mémoire par la commande "**free**" :

```
typedef struct point Point;

void detruire_point(Point *p)
{
  if (p != NULL)
    free(p);
}

void main(void)
{
  Point *p;
  p = creer_point (10.0, 20.0, 1);
  ...
  ...
  free(p);   /* On peut aussi liberer la mémoire
                sans faire de destructeur, juste
                quand on n'en a plus besoin */
}
```

C'est ce qu'on appelle un DESTRUCTEUR

ONERA

Exemple : `Exple_43_allocationC`

C++ : gestion dynamique de la mémoire (1)

Deux nouveaux opérateurs de gestion mémoire, **new** et **delete**, offrent une alternative aux fonctions `malloc` et `free` de la bibliothèque standard C.

Pour allouer un unique objet :

```
new MonType;      // MonType est un type quelconque
```

pour allouer un tableau de n objets :

```
new MonType[n];   // n est une expression arithmétique
```

`new` renvoie un pointeur de type `MonType*` (vers l'objet unique ou vers le premier objet du tableau).

Pour libérer la mémoire :

```
delete adresse;    // détruit l'objet
delete[] adresse; // détruit le tableau
```

ONERA

C++ : gestion dynamique de la mémoire (2)

En C

```
struct point {
  double x, y;
  int numero;
};
typedef struct point Point;

Point *ptr, *tab;

ptr = (Point*)malloc(sizeof(Point));
tab = (Point*)malloc(100*sizeof(Point));

/* ... */

free(ptr);
free(tab);
```

En C++

```
struct Point {
  double x, y;
  int numero;
};


Point *ptr, *tab;

ptr =  new Point;
tab =  new Point[100];

/* ... */

delete ptr;
delete[] tab;
```

ONERA

Exemple : `Exple_44_allocationCPP`

C++ : gestion dynamique de la mémoire (3)

En première approximation, la règle qui dit « *tout ce qui est alloué par* **new[]** *doit être libéré par* **delete[]** *et tout ce qui est alloué par* **new** *doit être libéré par* **delete** » est vraie. Mais la réalité est plus subtile :

```
typedef double DVect[100]; // nouveau type "DVect"

DVect *tab = new DVect;     // FAUX car 'new DVect' est
                            // synonyme de 'new double[100]'
double *tab1 = new DVect;         // CORRECT
double *tab2 = new double[100];   // CORRECT et synonyme
...
delete tab1;              // NON !! (a l'execution)
delete[] tab1;            // OUI car tab1 est un tableau
delete[] tab2;            // OUI aussi
```

ONERA

Exemple : `Exple_44_allocationCPP`

C vs C++ : pointeurs génériques

En C, un pointeur **`void*`** est appelé pointeur générique et ce type peut être converti vers tout type pointeur et réciproquement. En C++, seule existe la conversion dans le sens pointeur d'un type quelconque -> `void*`.

conversion de `T*` vers **`void`**`*` `// autorisée en C et C++`

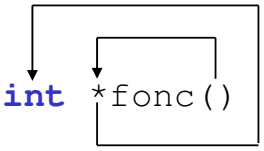
conversion de **`void`**`*` vers `T*` `// autorisée en C, illégale en C++`

```
void *generic;
int *ptr;

generic = ptr;          // autorisée en C et C++
ptr = generic;          // autorisée en C, illégale en C++

ptr = malloc(sizeof(int));  // autorisé en C, illégal en C++
ptr = (int*)malloc(sizeof(int));   // autorisée en C et C++
```


ONERA

C/C++ : une règle de lecture

La **règle de l'escargot** : lire le premier des opérateurs à droite de l'identificateur, puis le premier à gauche, à droite, à gauche, etc., le type de gauche toujours en dernier.



fonction qui retourne un pointeur sur un entier

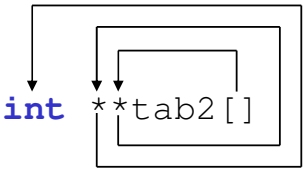


tableau de pointeurs sur des pointeurs d'entiers

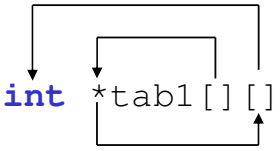
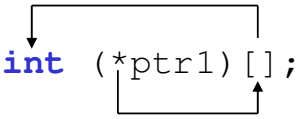


tableau de pointeurs sur des tableaux d'entiers

CONSEIL : utiliser **typedef** pour améliorer la lisibilité



pointeur sur un tableau d'entiers

ONERA

Notes :

Toujours partir de l'élément que l'on définit/déclare.

Prononcez (mentalement ou à haute voix) tout en faisant le parcours. Tenir compte des parenthèses pour faire les parcours des plus internes vers les plus externes. Quand on rencontre :

- `*` : on dit « est un pointeur »
- `[]` : on dit « est un tableau »
- `(args)` : on dit « est une fonction »

Par exemple : **`double`** `(*yt[2])(`**`double`**`)`, on dit « *yt est tableau de 2 pointeurs vers des fonctions prenant en argument un double et retournant un double* ».

```
#include <cmath>
#include <iostream>

double (*yt[2])(double);

void main() {
  yt[0] = &sqrt;
  yt[1] = &exp;
  std::cout << (*yt[0])(2.0) << std::endl;  // affiche 1.41412
  std::cout << (*yt[1])(1.0) << std::endl;  // affiche 2.71828
}
```

C/C++ : récursivité

Récursivité : possibilité pour une procédure de s'appeler elle-même.

Dans une fonction/procédure récursive, il y a toujours :

- une ou des conditions de terminaison ;
- un ou des appels à la fonction/procédure ;

Et parfois :

- une fonction/procédure qui initialise la récursion.

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

La récursivité est utilisée pour les parcours de listes, d'arbres de graphes, ..., pour les tris, ...

ONERA

C/C++ : le programme principal

« en plus »

```
void main(void);
int main(void);
int main(int argc, char *argv[]);
```

sont trois formes d'en-têtes de programme "`main`" autorisées par la quasi-totalité des compilateurs.

Soit un exécutable qui peut être appelé avec des paramètres optionnels :

```
> simu

> simu -f in.dat
```

`argv[0]` désigne le nom du programme :

```
int main(int argc, char* argv[]) {
  if (argc == 1)
    launch("simu.dat");
  else
    if ( (argc == 3) &&
         (strcmp(argv[1],"-f") == 0) )
      launch(argv[2]);
    else {
      printf("syntax : simu [-f in.dat]");
      return 1;
    }
  return 0;
}
```

ONERA

Précisions sur l'exemple :

La forme avec `argc` / `argv` est la forme « professionnelle » du programme principal.

Explication de l'algorithme :

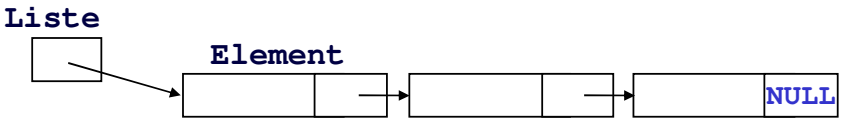
- s'il n'y a pas d'argument (`argc==1`), alors lancer le programme sur le fichier « `simu.dat` »
- s'il y a 2 arguments (`argc==3`), vérifier que l'option « `-f` » est présente et lancer le programme
- sinon produire une erreur

C/C++ : liste chaînée (1)

On distingue :

- les éléments : qui contiennent des données et ont un pointeur vers l'élément suivant (on suppose dans l'exemple qu'il s'agit de deux nombres réels)
- la tête de la liste : pointeur vers le premier élément de la liste

Le dernier élément de la liste pointe vers **NULL** en guise d'élément suivant.




```
struct Element {
  double x, y;    // Données
  Element *next;
};
typedef Element* ListXY;
```

ONERA

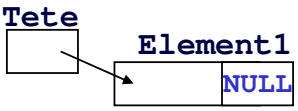
Note :

Ce que l'on va appeler « une liste » par la suite c'est un pointeur vers un élément. C'est ce que dénote l'alias de type : `Element*` ⇔ `ListXY`.

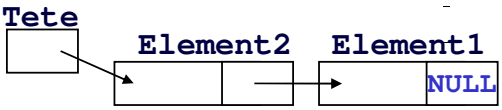
C/C++ : liste chaînée (2)

Processus de création d'une liste :

Au début la liste est vide. Le pointeur contient « NULL ».



On ajoute un élément à une liste vide. La liste pointe maintenant vers cet élément. Comme suivant, l'élément a « NULL ».



On ajoute un élément à une liste non vide. La liste pointe vers ce nouvel élément, il a comme suivant l'ancienne tête de la liste.

ONERA

Exercice : `Exple_45_listechainee`

1) Créer une méthode « `insert` » qui prend en entrée : une liste et deux nombres réels. La méthode ajoute un élement au début de la liste et modifie la liste pour qu'elle pointe vers cet élément. Utiliser le passage par référence.

2) Créer une méthode « `print` » qui affiche le contenu de la liste sans la modifier (utiliser const pour passer une référence constante en paramètre). La méthode affiche les couples de valeurs réelles. Elle doit fonctionner bien sûr si la liste est vide.

3) Créer une méthode « `clear` » qui vide le contenu de la liste. Elle détruit les éléments et transforme la liste passée en argument pour qu'elle soit une liste vide.

Cette dernière méthode est sans doute la plus difficile à faire : il faut effacer progressivement les éléments; mais quand on efface un élément, on risque de perdre l'information « `next` » qui permet de passer à l'élément suivant …

TESTER LE CODE AVEC UN OUTIL COMME **PURIFY**

C/C++ : allocation de matrices (1)

En C/C++, il est possible de créer des matrices dont la taille peut être déteminée à la compilation :

```
const int n = 2;
const int p = 3;

double a[2][3];
double b[n][3];
double c[2][n];
double d[n][p];
```

Il n'est pas possible simplement d'allouer dynamiquement une matrice comme on alloue un vecteur dimensionnel :

```
double *v = new double[n];
```

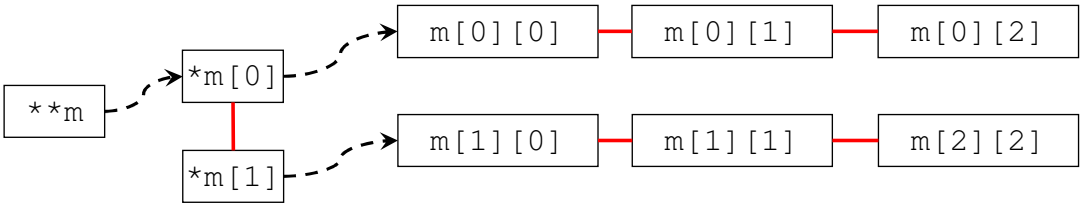
Une matrice est un tableau de tableaux. Comme un tableau est un pointeur, un tableau de tableau est un pointeur de pointeurs :

```
double **m;
```

ONERA

C/C++ : allocation de matrices (2)

Une première façon de procéder consiste à allouer autant de vecteurs que de lignes de la matrice :



Pour une matrice de taille `N` x `P` :

- on alloue un tableau de `N` pointeurs vers des doubles, le premier élément de ce tableau est pointé par `**m`
- par une boucle, on alloue `N` tableaux de `P` nombres réels double précision, les adresses des premiers éléments de chacun de ces tableaux étant pointées par un élément du tableau de pointeurs alloués précédemment

ONERA

Exercice : `Exple_46_matrices`

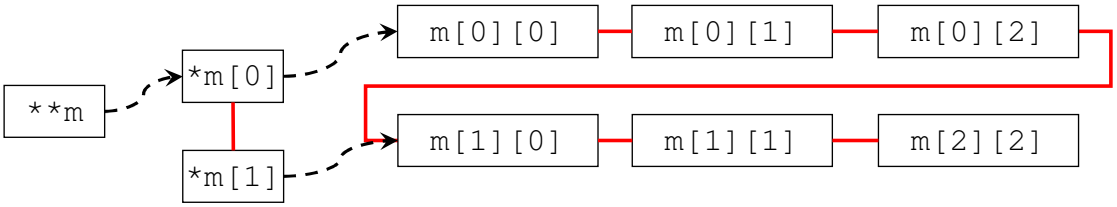
1) Réaliser une fonction qui retourne une matrice allouée dynamiquement selon la solution décrite par la planche : `double** alloc_dmatrix(unsigned int n, unsigned int p)`

2) Tester ces matrices avec l'opérateur []

3) Réaliser une fonction qui libère la mémoire allouée : `void free_dmatrix(double** mat)`

C/C++ : allocation de matrices (3)

Une autre solution (adoptée par *Numerical Recipes* par exemple) consiste à réaliser en mémoire la structure suivante :



Pour une matrice de taille `N` x `P` :

- on alloue un tableau de `N` pointeurs vers des doubles, le premier élément de ce tableau est pointé par `**m`
- on alloue un tableau de `N` x `P` nombres réels **contigüs** double précision, le premier élément de ce tableau est pointé par le premier pointeur du tableau alloué précédemment
- par une boucle, on fait pointer chaque pointeur du premier tableau vers le premier élément de chaque ligne de `P` éléments

ONERA

Exercice : `Exple_46_matrices`

1) Réaliser une fonction qui retourne une matrice allouée dynamiquement selon la solution décrite par la planche : `double** alloc_dmatrix(unsigned int n, unsigned int p)`

2) Tester ces matrices avec l'opérateur []

3) Réaliser une fonction qui libère la mémoire allouée : `void free_dmatrix(double** mat)`