ISIE4 – C++ & C++ embarqué

Premiers pas en C++

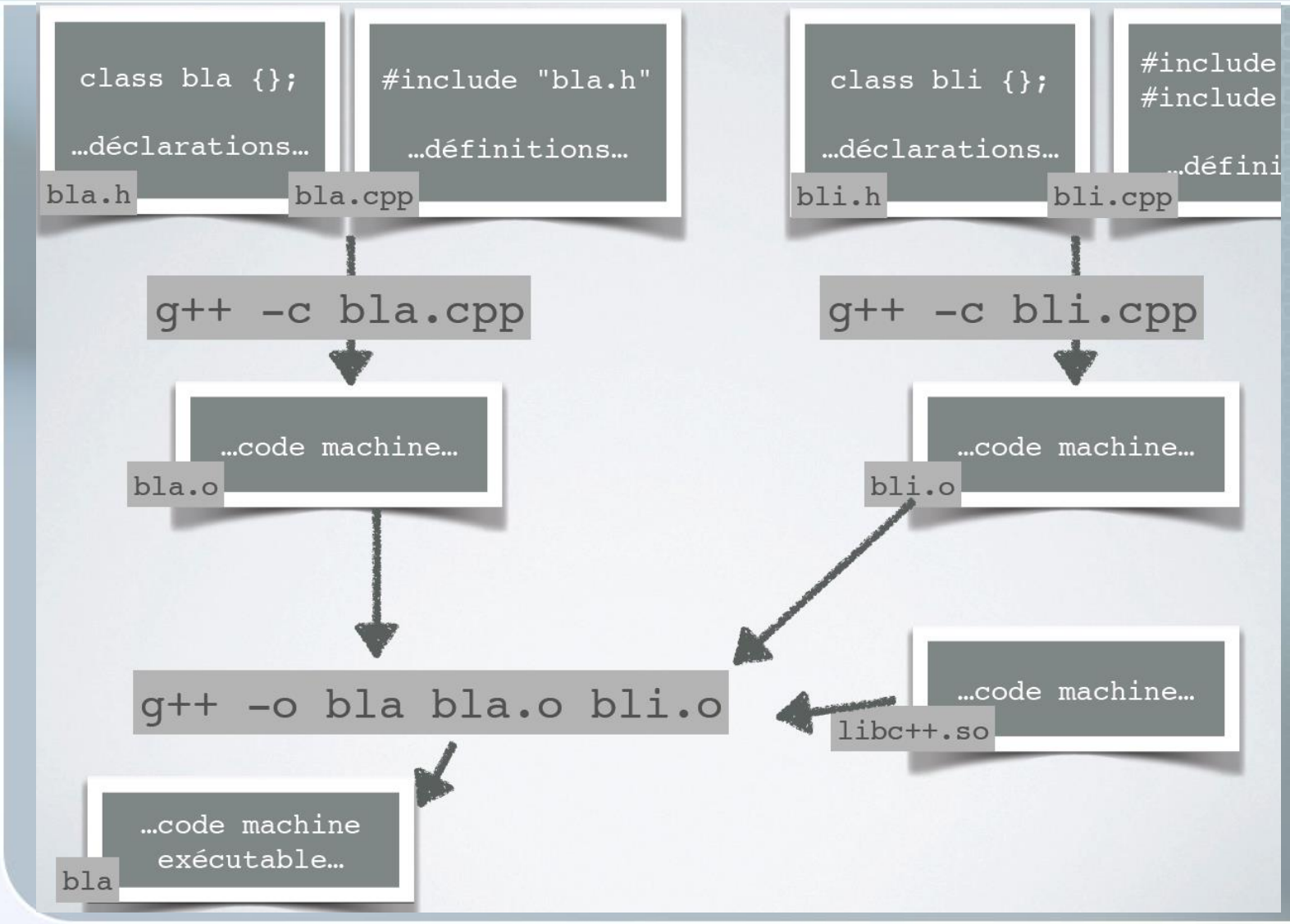
La compilation

- Utiliser de préférence le compilateur GCC, sur une plateforme GNU/Linux
  - GCC polymorphe (multiples langages supportés)
  - GCC supporte tous les standard
    - Base du dernier standard
    - ***GCC 14 min pour support C++23 complet***
  - GCC Utilisable en compilation croisée
    - Cibles ARM notamment
- Utilisez l'IDE / L'outil / La plateforme qui vous convient
  - Y compris les outils en ligne
  - Pour moi : cmake, GCC, GNU/Linux
    - Mais avec Visual Studio Code comme IDE

Convention de nommage du code source

- Le code source dans un fichier d'extension (ou postfixé)
  - ***.cpp*** .C .cxx .cc .cp .c++
- Les fichiers d'entête (ne contiennent que des déclarations)
  - ***.hpp*** .H .hxx .hh ***.h***

Rappel…


class bla {};
…déclarations…
bla.h
#include "bla.h"
…définitions…
bla.cpp
class bli {};
…déclarations…
bli.h
#include
#include
…défini
bli.cpp
g++ -c bla.cpp
g++ -c bli.cpp
…code machine…
bla.o
…code machine…
bli.o
g++ -o bla bla.o bli.o
…code machine…
libc++.so
…code machine
exécutable…
bla

Options de compilation « conseillées » (gcc)

- -On : Activer les optimisations
  - -Og : En phase de développement
  - -O1, -O2, -O3 ou -Os pour la taille
  - Cf https://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/Optimize-Options.html
- -std=C++23
  - Utilisation de la norme C++23
- -g : Génération des informations de debug (pour utiliser GDB)
- -Wall, -Wextra, -Wpedantic

Les opérateurs du C++
Priorité 1

::	portée	std::cout

Les opérateurs du C++
Priorité 2

()	appel de fonction	f(3)
()	initialisation membres	A::A(int x) : s(4) {}
[]	indice tableau	t[4]
->	accès par pointeur	pIndividu->nom
.	accès par objet	individu.nom
++	post-incrémentation	i++
--	post-décrémentation	i--
const_cast	coercition	const_cast<dest>(src)
dynamic_cast	coercition	dynamic_cast<dest>(src)
static_cast	coercition	static_cast<dest>(src)
reinterpret_cast	coertition	reinterpret_cast<dest>(src)
typeid	RTTI	typeid(individu)

Les opérateurs du C++
Priorité 3

! ou not	non logique	!true
~ ou compl	complément b.àb.	x = ~0x1f
++	pré-incrémentation	++x
--	pré-décrémentation	--x
-	moins unaire	a = -b;
+	plus unaire	c = +14;
*	déréférencement	(*pIndividu).nom
&	adresse	&individu
new	allocation	new int;
new []	allocation tableau	new int[56];
delete	désallocation	delete pIndividu;
delete []	désallocation tableau	delete [] ptIndividu;
(T)	coercition	(int)13.4
sizeof	taille représentation	sizeof(int), sizeof(i)

Les opérateurs du C++
Priorités 4, 5 et 6

->*	sélecteur (pointeur)	p->*membre
.*	sélecteur (objet)	o.*membre
*	multiplication	a*b
/	division	a/b
%	reste modulo	a%b
+	addition	a+b
-	soustraction	a-b

Les opérateurs du C++
Priorités 7,8 et 9

<<	décalage gauche b.àb.	1<<17
>>	décalage droit b.àb.	a>>4

<	inférieur strict	if (a<b)
<=	inférieur	if (a<=b)
>	supérieur strict	if (a>b)
>=	supérieur	if (a>=b)

== ou eq	comparaison égalité	if (a==b)
!= ou not_eq	comparaison différence	if (a!=b)

Les opérateurs du C++
Priorités 10, 11, 12 et 13

`& ou bitand`	et b.àb.	`0x12 & 0x4;`
`^ ou xor`	ou-exclusif b.àb.	`0x12 ^ 0x4;`
`| ou or`	ou b.àb.	`0x12 | 0x4;`
`&& ou and`	et logique	`if (a and b)`

Les opérateurs du C++
Priorités 14, 15, 17 /!\ et 18 /!\

`|| ou or`	ou logique	`if (pluie || neige)`
`?:`	expression conditionnelle	`(a>b)?a:b`
`throw`	levée d'exception	`throw Ex("incendie")`
`,`	évaluation séquentielle	`a=3,b=7,f(4)`

Les opérateurs du C++
Priorité 16

=	affectation	a = 13
+=	incrémentation et affectation	a += 12
-=	décrémentation et affectation	a -= 12
*=	multiplication et affectation	a *= 2
/=	division et affectation	a /= 15
%=	reste modulo et affectation	a %= 5
&= ou	et b.àb. et affectation	x &= 0xff
^= ou	ou-exclusif b.àb. et affectation	x ^= 0xae
|= ou or_eq	ou b.àb. et affectation	x |= 0xae
<<=	décalage gauche et affectation	i <<= 2
>>=	décalage droit et affectation	i >>= 5

Les variables

- Rappels & définitions
  - Pour bien savoir de quoi on parle…
- Déclaration
- Définition
- Usages
- Subtilité (1)
- Syntaxe fonctionnelle
  - Pour l'initialisation

Rappels et définitions © B. STROUSTRUP

- Un type est un ensemble de valeurs et un ensemble d'opérations associées
- Un objet est un espace mémoire contenant une valeur d'un type donné
- Une valeur est un mot binaire interprété dans un type donné
- Une variable est un objet nommé
- Une déclaration associe un nom à un objet
- Une définition est une déclaration associant un espace mémoire à l'objet déclaré

Rappels

- Une ***variable*** est un contenant informatique nommé sur lequel des opérations de consultation et modification du contenu sont définies
  - La lecture produit une ***r-value***
  - L'écriture se fait par l'intermédiaire de la ***l-value***
- Le nom est appelé identificateur de variable.
  - En C++ on parle aussi de ***référence***

Déclaration d'une variable

- consiste en la déclaration d'une ***référence***
  - pour en contrôler sa portée/visibilité
- Nécessite un ***type***
  - pour en contrôler l'usage
- ***Déclaration = Référence + Type***

Définition d'une variable

- Déclaration
  - Associée ou non à un qualifieur
    - auto, static…
- Associée à une ***allocation mémoire***
- Le tout, éventuellement suivi d'une ***initialisation***

- ***Définition = Déclaration + Allocation [+ Initialisation]***

Usages d'une variable

- La consultation s'effectue en utilisant en r-value la référence
- La modification s'effectue en utilisant en l-value la référence : instruction d'affectation
- Attention
  - Initialisation et affectation sont deux opérations distinctes (***il ne peut jamais y avoir qu'une seule initialisation pour une variable donnée!***)

Pour synthétiser tout ça… (subtilité 1)

- Une déclaration

  extern int i;
- Une définition

  int i;
- Une définition AVEC initialisation

  int i=42;
- Une affectation

  i=666;
- Attention

  int i;

  i=42; // Pas une initialisation !

Initialisation
Syntaxe fonctionnelle

- *int i(42);*
  - S'interprète ***exactement*** comme

    int i = 42;
- Syntaxe utilisable avec n'importe quel type

  double pi(3.1415);

  char Car('A');

  …

Initialisation en C++ (>11)

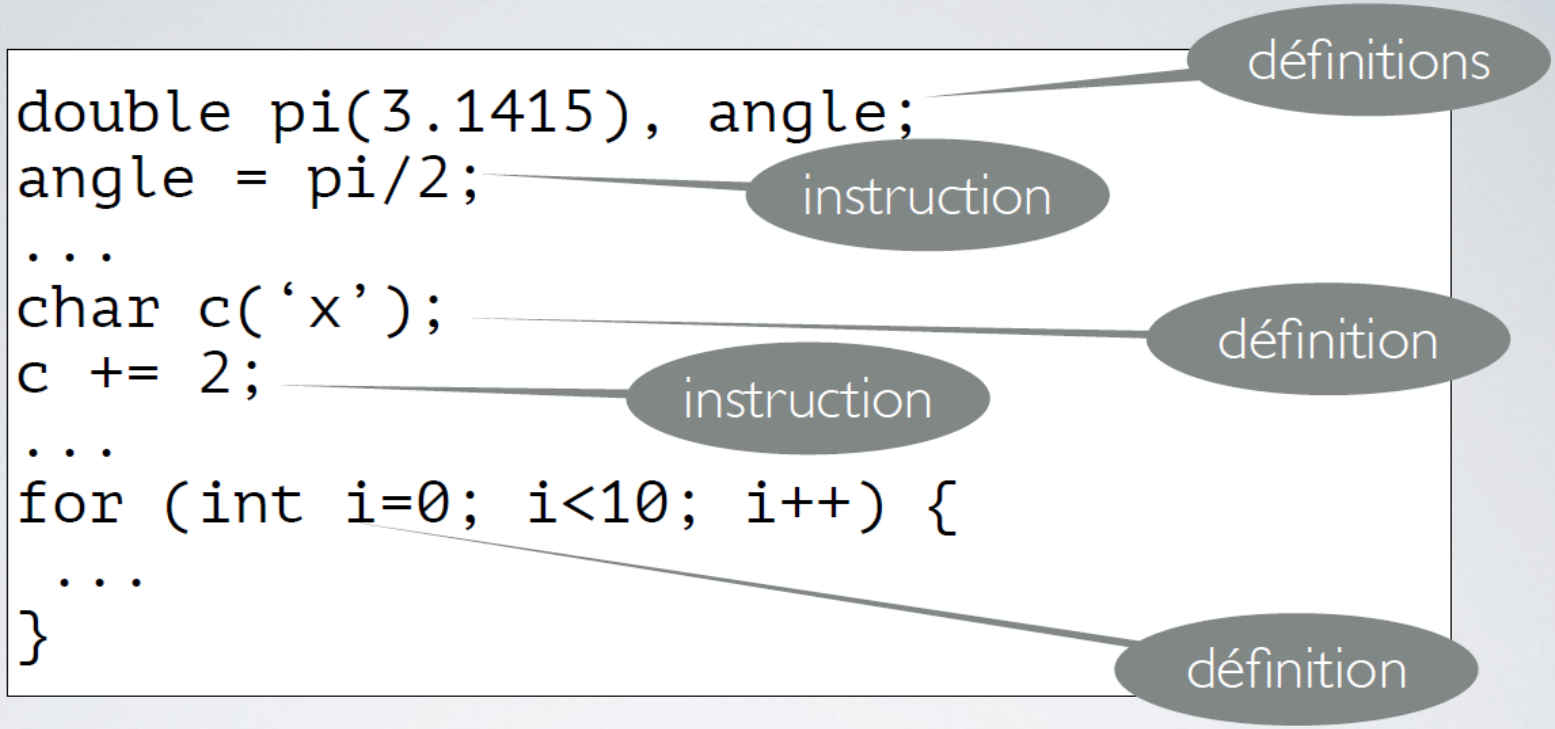
- Trois syntaxes
- Classique (C flavour)

  int i=42;
- Fonctionnelle (C++ flavour)

  int i(42);
- Fonctionnelle V2 (C++11 flavour)

  int i{42};
  - Forme préférée depuis le C++11
    - Même si pas (encore) assez utilisée en dehors de l'initialisation d'un champ d'objet

Re synthèse…



- Boucle for
  - La définition a la portée de la boucle
  - /!\ « anciens » compilateur Microsoft

Au passage…
Les types fondamentaux du C++

- On retrouve l'ensemble des types fondamentaux du C
  - Avec des extensions (long long par exemple)
  - Types stdint (C99)
    - uint_8_t, int16_t…
- Type booléen : bool
  - true / false
- Le type ***auto*** (depuis le C++11)
  - N'est pas un type !
  - Ne pas en avoir peur, à utiliser sans modération, mais avec ***discernement*** !

Autour des constantes…

- Types constants
- Objets et variables constants
- Expressions constantes

Types constants
const *type*

- Interdisent l'usage en tant que l-value
  - Avec les opérateurs qui modifient la valeur de la variable (=, ++, etc.)
- Initialisation obligatoire
  - Car affectation interdite
- Identiques aux types non-constants pour le reste
- Définition de ***deux*** types différents pour chaque type initial
  - exemple pour int
    - const int : type constant
    - int : type non constant

Objets et variables constants

- Variables et objets définis à partir d'un type constant
  - const int Nb{42};
  - const uint8_t CR{0x0D};
  - const std::string_view Msg{"Hello World!"};
  - …
- Fonctionne avec pointeurs et références

Expressions constantes ***constexpr***

- A la fois simple et très compliqué
- Très puissant en termes d'optimisation des performances
  - Transfert (dès que possible) des calculs et opérations du run-time vers le compile-time
- Ne pas chercher à tout comprendre en une seule fois
  - Retenir quelques réflexes et mécanismes de base et très simples

Types constants dans les prototypes

```
char *strcpy(char *dst,const char *src);
```

```
char *s = new char[100];
const char *t = “Bonjour“;
strcpy(s,t);
strcpy(t,s); // interdit
char *u = new char[100];
strcpy(u,s);
```

- On a la garantie que les caractères de la chaîne source ne seront pas modifiés lors d’un appel à la fonction
- Celui qui implémente la fonction ne peut modifier les caractères de l’argument constant

Fonctions *constexpr*

- Rappel : constantes d'expression via constexpr
  - ~~Équivalent à #define~~
- L'écriture de ces expressions pourrait être simplifiée par l'emploi de fonctions
- Les fonctions constexpr sont l'équivalent propre des macro-fonctions définies par #define
  - Et plus encore…

• Old C

```
#define PI 3.1415926
#define hypothenuse(x,y) \
                    (sqrt((x)*(x)+(y)*(y)))
```

• Old C++

```
#define hypothenuse(x,y) \
                    (sqrt((x)*(x)+(y)*(y)))
const double PI = 3.1415926
```

• C++11

```
constexpr double divideBy2(const double v) {
  return v/2.0;
}
constexpr double PI = 3.1415926;
constexpr double PI_SUR_2 = divideBy2(PI);
```

constexpr vs const

- Pour des variables constantes
- Peut simplement être utilisé en remplacement des #define (macros) du langage C
  - L'expression reste typée → sécurité
- Exemples
  - constexpr uint16_t SIZE{1024};
  - constexpr std::string_view MESSAGE{"Hello World!"};
  - …

Remarques

- Une fonction constexpr peut-être évaluée à la compilation lorsqu'elle est appelée avec des expressions constantes
  - Sinon elle se comporte comme une fonction ordinaire!
- Pour cela, elle ne peut comporter de boucles (pas de for, while, etc) - entre autres.
  - Cette fonction doit pouvoir être évaluée à la compilation! (évolution du C++14)

Le prototypage en C++

- Obligatoire
  - C++ bien plus strict que le C
- Déclaration d'une fonction appelée f, prenant 3 arguments respectivement un entier, un flottant et un entier, et renvoyant une valeur entière :

  int f(int i1,float f1,int i2);
- déclaration d'une fonction ne prenant PAS d'argument (et renvoyant un entier) :

  int f();
  - attention car en C c'est int f(void) car int f() signifie fonction à nombre d'arguments indéfini!
    - Fonction variadique

Valeurs par défaut

```
constexpr int RADIAN=0;
constexpr int DEGRE=1;
constexpr int GRADE=2;

double sinus(double angle,
             const int unite=RADIAN);

void main() {
  double angle=3.14, s;
  s = sinus(angle);
  double angleEnDegres = 90, sd;
  sd = sinus(angleEnDegres,DEGRE);
}
```

Valeurs par défaut (suite)

- Niveau 0 du polymorphisme
- Ne peuvent être arbitrairement mélangées. La liste des arguments d'une fonction est sécable en deux parties (éventuellement vides)
  - D'abord les arguments sans valeur par défaut (à l'appel ils doivent tous être spécifiés)
  - Ensuite ceux avec valeur par défaut (à l'appel on peut spécifier des valeurs pour les premiers d'entre eux et laisser le compilateur compléter la spécification à l'aide des valeurs par défaut)

```
void f(int a,int b,int c=1,int d=2,int e=3);
void main() {
    f(10,11,13); // équiv. f(10,11,13,2,3);
}
```

La surcharge

- Comment résoudre élégamment le problème suivant :
  - Écrire une fonction permettant d'additionner deux entiers, puis une autre deux flottants...
- En langage C:

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

```
int   add1(int a   , int b)   { return a+b; }
float add2(float a, float b) { return a+b; }
```

La surcharge (suite)

- En C++:

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

- Pas de problème de conflit de nom, le compilateur est suffisamment malin pour deviner quoi faire avec :

```
int neuf = add(4,5);
float f = add(4.6f,99.23f);
char c = 12; int i = add(c,14); // promotion char→int
```

Attention : la détermination de la fonction adéquate n'utilise que la **signature** des fonctions

Signature : identificateur + liste des types des arguments
Prototype : Signature + type de la valeur de retour

Passage de paramètres

- Deux modes de passage d'arguments
  - Par valeur
  - Par référence
- C'est tout ?
  - Oui
- Et le passage par pointeurs (par adresse) ?
  - Un pointeur est un type
    - On passe la ***valeur*** du pointeur
      - C'est un passage par valeur

Transmission par valeur pass-by-value

- création d'une variable locale à la fonction initialisée avec la valeur transmise
  - ***création d'une copie locale***

Transmission par référence pass-by-reference

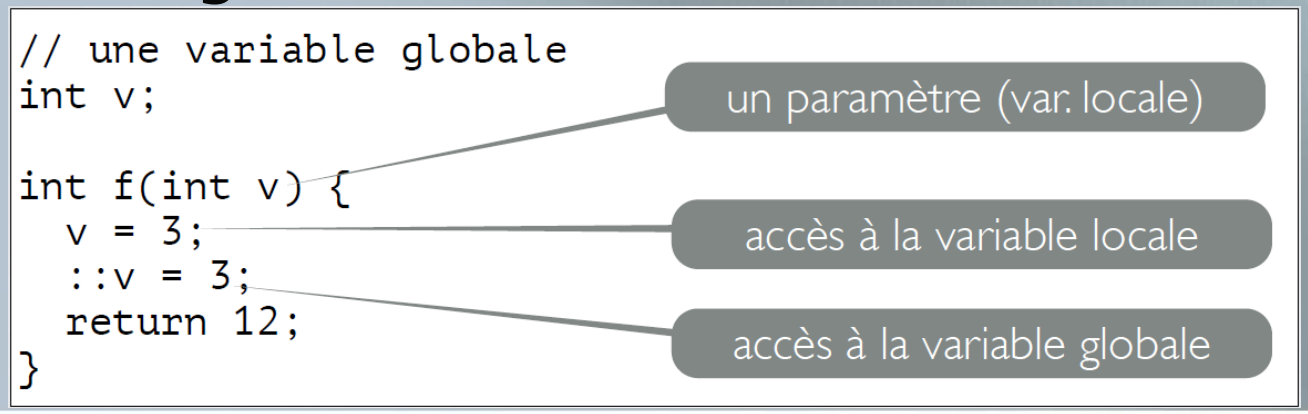
- Pour éviter la copie de variables (volumineuses ou non), il est possible de transmettre la variable elle-même et non sa valeur
- La variable peut donc être modifiée durant l'appel
- Il est recommandé en C++ de n'utiliser ***que*** le passage de paramètres par référence (ou référence constante)
  - Surtout quand il s'agit de travailler avec des objets

Transmission par référence pass-by-const-reference

- Pour éviter la copie de variables volumineuses, il est possible de transmettre la variable elle-même et non sa valeur
- Pour protéger la variable de modifications non souhaitée dans la fonction

Portées des noms

- Opérateur :: pour contrecarrer l'effet du masquage
  - En réalité l'opérateur permet d'accéder à un nom dans un contexte donné (espace de noms)
  - pas de contexte (mais opérateur) = contexte global



Espace de nommage

- Les conflits de noms sont inévitables
  - Comment limiter leur impact ?
- En encapsulant les déclarations dans des espaces de noms...

```
namespace math {
 double pi = 3.1415926;
 double e = 2.718;
}
```
maths.hpp

```
namespace chimie {
 char *e = “C5H9NO4“;
}
```
chimie.hpp

```
#include “math.hpp“
#include “chimie.hpp“

float quatreVingtDixDegresEnRadians = math::pi / 2;
char *acideGlutaminique = chimie::e;
```
ailleurs.cpp

Utilisation

- Via l'opérateur de portée ::

```
#include <iostream>

int main() {
   std::cout << "Bonjour tout le monde" << std::endl;
   return 0;
}
```

- Avec *using* (s'il n'y a pas d'ambiguïté)

```
#include <iostream>

using namespace std;

int main() {
   cout << "Bonjour tout le monde" << endl;
   return 0;
}
```

Trois dernières choses là-dessus…

- Il existe un espace de noms (par défaut) sans nom
- On peut imbriquer les espaces de noms

```
namespace one {
 namespace two {
  const int zero = 0;
 }
}
one::two::zero;
```

- On peut définir des alias d'espace de noms

```
namespace MonEspace { const int zero = 0; }
namespace MySpace = MonEspace;

MonEspace::zero; // équiv. MySpace::zero;
```

Résumé

- Conventions nommage des fichiers
- Opérateurs
- ***Variables***
  - Déclaration, définition, etc.
- ***Autour des constantes***
  - Types, variables et expressions
- ***Prototypage***
  - Valeurs par défaut, surcharge…
- Portée des noms
- ***Passage de paramètres***
  - Par valeur, par référence

FIN