

LE LANGAGE C++ MASTER 1 UN C ÉTENDU ?

Jean-Baptiste.Yunes@univ-paris-diderot.fr

U.F.R. d'Informatique

Université Paris Diderot - Paris 7

2019–2020

LA COMPILATION

gcc un compilateur polymorphe...

- gcc permet de compiler plusieurs langages (sauf exception) : C, C++, Objective-C, Fortran, Java et Ada
- gcc supporte les tous standards
- gcc est la base du dernier standard

Convention de nommage du code source :

- un code source C++ prend place dans un fichier d'extension `.cpp` `.C` `.cxx` `.cc` `.cp`
`.c++`
- un fichier d'entête (ne contenant que des déclarations sauf exception cf. templates `.tcc`) est d'extension `.hpp` `.H` `.hxx` `.hh`

La compilation s'effectue en invoquant le compilateur sur le fichier source :

```
g++ -c fichier.cpp
```

cela génère un fichier objet

```
fichier.o
```

l'édition de lien peut être faite en invoquant le compilateur :

```
g++ -o fichierExécutable fichier.o
```

ces étapes peuvent être agrémentées d'options...

```
class bla {};  
...déclarations...
```

bla.h

```
#include "bla.h"  
...définitions...
```

bla.cpp

```
g++ -c bla.cpp
```

```
...code machine...
```

bla.o

```
g++ -o bla bla.o bli.o
```

```
...code machine  
exécutable...
```

bla

```
class bli {};  
...déclarations...
```

bli.h

```
#include  
#include  
...défini
```

bli.cpp

```
g++ -c bli.cpp
```

```
...code machine...
```

bli.o

```
...code machine...
```

libc++.so

on peut forcer le langage par l'intermédiaire de l'option `-x c++`

- gcc supporte `c++98`, `c++03`, `c++11`, `c++14`, `c++17` (plus l'expérimental `c++2a`). L'option `--std=c++98` permet de sélectionner explicitement le standard (et non les extensions GNU)
- l'option `-pedantic` (ou `-pedantic-errors`) permet d'obtenir toutes les erreurs spécifiées du standard (en tant qu'erreurs et non avertissements)

LES MOTS RÉSERVÉS

Les mots réservés du C++

`alignas alignof` and `eq asm auto bitand bitor bool`
`break case catch char char16_t char32_t class compl`
`const constexpr const_cast continue decltype default`
`delete do double dynamic_cast else enum explicit`
`export extern false float for friend goto if inline`
`int long mutable namespace new noexcept not not_eq`
`nullptr operator or or_eq private protected public`
`register reinterpret_cast return short signed sizeof`
`static static_assert static_cast struct switch`
`template this thread_local throw true try typedef`
`typeid typename union unsigned using virtual void`
`volatile wchar_t while xor xor_eq`

en rouge les mots réservés normalisés par le C++11

LES OPÉRATEURS

- Les opérateurs du C++ (priorité 1) :

::	portée	std::cout
----	--------	-----------

- **Les opérateurs du C++ (priorité 2) :**

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

- Les opérateurs du C++ (priorité 3) :

! ou not	non logique	!true
~ ou compl	complément b.àb.	x = ~0x1f
++	pré-incrémentation	++x
--	pré-décrémententation	--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++ (priorité 4) :

<code>->*</code>	sélecteur (pointeur)	<code>p->*membre</code>
<code>.*</code>	sélecteur (objet)	<code>o.*membre</code>

- Les opérateurs du C++ (priorité 5) :

*	multiplication	$a*b$
/	division	a/b
%	reste modulo	$a\%b$

- Les opérateurs du C++ (priorité 6) :

+	addition	a+b
-	soustraction	a-b

- Les opérateurs du C++ (priorité 7) :

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

- Les opérateurs du C++ (priorité 8) :

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

- Les opérateurs du C++ (priorité 9) :

<code>==</code> ou <code>eq</code>	comparaison égalité	<code>if (a==b)</code>
<code>!=</code> ou <code>not_eq</code>	comparaison différence	<code>if (a!=b)</code>

- Les opérateurs du C++ (priorité 10) :

<code>& ou bitand</code>	et b.àb.	<code>0x12 & 0x4;</code>
------------------------------	----------	------------------------------

- Les opérateurs du C++ (priorité 11) :

<code>^</code> ou <code>xor</code>	ou-exclusif b.àb.	<code>0x12 ^ 0x4;</code>
------------------------------------	-------------------	--------------------------

- Les opérateurs du C++ (priorité 12) :

ou or	ou b.à.b.	0x12 0x4 ;
-------	-----------	--------------

- Les opérateurs du C++ (priorité 13) :

lazy evaluation

<code>&& ou and</code>	et logique	<code>if (a and b)</code>
--------------------------------	------------	---------------------------

- Les opérateurs du C++ (priorité 14) :

lazy evaluation

<code> </code> ou <code>or</code>	ou logique	<code>if (pluie neige)</code>
------------------------------------	------------	----------------------------------

- Les opérateurs du C++ (priorité 15) :

? :	expression conditionnelle	$(a > b) ? a : b$
-----	---------------------------	---------------------

- Les opérateurs du C++ (priorité 16) :

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

- Les opérateurs du C++ (priorité 17) :

<code>throw</code>	levée d'exception	<code>throw Ex("incendie")</code>
--------------------	-------------------	-----------------------------------

- Les opérateurs du C++ (priorité 18) :

,	évaluation séquentielle	<code>a=3 , b=7 , f (4)</code>
---	-------------------------	----------------------------------

- Les priorités permettent d'interpréter les expressions multi-opérateurs :

$$a+b*c$$

- Certains opérateurs sont gauches :

$$a+b+c+d \iff ((a+b)+c)+d)$$

- D'autres opèrent à droite :

$$a=b=c=d \iff (a=(b=(c=d)))$$

- Les commentaires :
 - multiligne

```
/*  
    cette suite d'instructions est la plus  
    importante du programme  
*/  
i = i+1;  
j = i-1;
```

- uniligne

```
i = i+1; // et un de plus!
```

LES VARIABLES

- 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

- Quelques rappels sur les variables :
- 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** *simplification!*
 - 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**

- La **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

- La **définition** d'une variable est :
 - une **déclaration** (agrémentée ou non d'un qualifieur : auto, static, ...)
 - associée à une **allocation** mémoire
 - le tout éventuellement suivi d'une **initialisation**

Attention, auto en c++11 sert à autre chose...

Définition = Déclaration + Allocation [+ Initialisation]

- Les 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!)

- En C++ :
 - une déclaration :

```
extern int i;
```

- une définition :

```
int i;
```

- une définition avec initialisation :

```
int i=4;
```

- une affectation :

```
i=4;
```

- Attention :

```
int i;  
i=4; // ceci n'est pas une initialisation
```

- En C++ :
 - syntaxe fonctionnelle pour l'initialisation

```
int i(4);
```

qui s'interprète exactement comme

```
int i=4;
```

- peut être utilisé pour n'importe quel type (on verra plus tard d'autres cas intéressants) :

```
double pi(3.1415);  
char c('x');
```


- En C++11, il existe trois syntaxes (!?) pour l'initialisation :

```
int i=4;
```

```
int i(4);
```

```
int i{4};
```

- On étudiera plus tard les différences induites par ces initialisations

La forme préférée en C++11 est {}

```
double pi(3.1415), angle;
```

```
angle = pi/2;
```

```
...
```

```
char c('x');
```

```
c += 2;
```

```
...
```

```
for (int i=0; i<10; i++) {
```

```
...
```

```
}
```

définitions

instruction

définition

instruction

définition

- pour `for` : la définition a la portée de la boucle...
Attention c'était anciennement faux pour les compilateurs ©Microsoft!

**LES CONSTANTES
EXPRESSIONS CONSTANTES
ET
VARIABLES CONSTANTES**

- Les **expressions constantes** (c++11)
 - une expression constante doit être définie avec le mot-clé `constexpr`

```
constexpr double pi = 3.1415926;  
constexpr double pi_sur_2 = pi/2;
```

- les valeurs des variables de ce genre sont **nécessairement** déterminées à la compilation

- de telles variables peuvent être utilisées partout où des constantes sont attendues :

```
constexpr int taille = 30;
...
double tableau[taille];
...
switch (something) {
    case taille:
        ...
}
```

- ces variables sont non modifiables...

sont à utiliser de préférence en lieu et place de #define

- Les **types constants** : `const type`
- interdit l'usage en tant que l-value, i.e. les opérateurs qui modifient la valeur de la variable du type considéré (ex: `=`, `++`)
- initialisation obligatoire
- le reste est identique au type non-constant


```
const double avogadro = 6.022E23;  
const double racineDeTrois = 1.732;  
avogadro = 34; // interdit
```

```
const double *pointeurSurConstante = &avogadro;  
pointeurSurConstante = &racineDeTrois;  
*pointeurSurConstante = 12; // interdit
```

```
double rapport = 24, longueur = 4.5;  
double *const pointeurRapport = &rapport;  
*pointeurRapport = 89;  
pointeurRapport = &longueur; // interdit
```

```
const double * const pConstanteDAvogadro = &avogadro;  
pConstanteDAvogadro = &racineDeTrois; // interdit  
*pConstanteDAvogadro = 28.9; // interdit
```

- 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

- En C++ :
- les constantes sont des variables (contraintes car non modifiables) qui peuvent être employées partout où des littéraux constants peuvent l'être :

```
const int DIMENSION = 100;  
  
int tableau[DIMENSION];  
for (int i=0; i<DIMENSION; i++) {  
    ...  
}
```


- `constexpr` vs `const` ?
- les `constexpr` sont des variables constantes déterminées à la compilation
- les `const` sont des variables constantes dont la valeur n'est pas nécessairement déterminée/connue à la compilation mais parfois à l'exécution



LES TABLEAUX

- En C++11 il est recommandé d'utiliser le type `vector` au lieu des tableaux hérités du C :

```
#include <vector>
vector<int> monTableau = { 1, 2, 3, 4, 5 };
monTableau[2] += 10;
cout << monTableau[3] << endl;
```

- on bénéficie des initialisations avec des séquences, etc.

- on peut itérer sur les éléments d'un `vector` avec la boucle sur les séquences :

```
#include <vector>
vector<int> monTableau = { 1, 2, 3, 4, 5 };
for (int i : monTableau) {
    ...
}
```

- **Attention : les `vector` C++ ne lèvent pas d'exception lors d'accès logiquement erronés via `[]`. Les accès via `.at()` sont par contre sûrs.**

L'ALLOCATION DYNAMIQUE

L'allocation dynamique en C++ new :

- *new type*

```
int *pi = new int;  
*pi = 12; // affectation!!!!
```

- *new type(valeur initiale)*

```
int *pi = new int(12); // initialisation
```

- *new type[dimension]*

```
int *t = new int[12]; // tableau!?
```

- pas d'initialisation de tableaux à l'allocation dynamique

Attention : ne pas employer `new` et `malloc/calloc` en même temps. On fait du C ou du C++.

L'allocation dynamique en C++ delete :

- `delete adresse`

```
int *pi = new int;  
*pi = 12; // affectation!!!!  
delete pi;
```

- `delete [] adresse`

```
int *t = new int[12]; // tableau!?  
delete [] t;
```

Attention : ne pas employer **delete** et **free** en même temps.
On fait du C ou du C++.

LE PROTOTYPAGE

Le prototypage en C++

- obligatoire! C++ est 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'arguments (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!

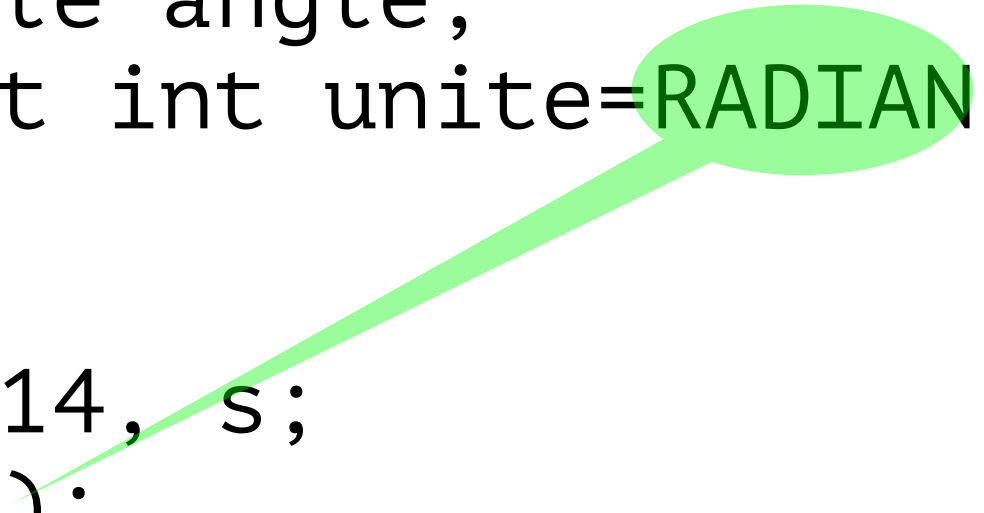
C++ est un langage fortement typé

Les **valeurs par défaut** en argument (sont à placer dans les déclarations)

```
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);
}
```



Les valeurs par défaut en argument

- un **polymorphisme** très pauvre
- 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 il 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 glu C/C++

- pour déclarer une fonction C à utiliser dans le monde C++, il faut la déclarer comme telle (le nommage des liens, les transmissions d'arguments peuvent être (et sont généralement) différents d'un monde à l'autre) :

```
extern "C" int f(int);
extern "C" {
    int g(int);
    void h(float);
}

...
    f(4);
    h(5.4f);
...

```

LA SURCHARGE

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

mais il faut de la mémoire ou une bonne documentation....

La surcharge de fonctions

- Écrire une fonction permettant d'additionner deux entiers, puis une autre deux flottants. 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

La directive `inline` :

- permet d'économiser l'appel de fonction...
mais peut être ignorée par le compilateur...
donc inutile d'y prêter (trop) attention...

```
inline int f(int a,int b) { ... }
```

même rôle que la macro-définition mais
avec vérification des types et sans effet
secondaire.

PORTABILITÉ C/C++

- Incompatibilités C/C++ :
 - types structurés. En C, struct type est le nom du type défini, en C++ type est le nom du type défini par struct type {};
 - en C++ les types imbriqués le sont vraiment! En C, ils sont aplatis :

```
struct Vehicule {
  struct Moteur {
  };
};
Vehicule k;
Vehicule::Moteur m;
```

En C++

```
struct Vehicule {
  struct Moteur {
  };
};
struct Vehicule k;
struct Moteur m;
```

En C

```
struct Vehicule {
};
struct Moteur {
};
struct Vehicule k;
struct Moteur m;
```

En C



Utilisation de l'opérateur de portée ::

LES PORTÉES DES NOMS

- L'opérateur `::` pour contrecarrer l'effet du masquage :

```
// une variable globale
int v;

int f(int v) {
    v = 3;
    ::v = 3;
    return 12;
}
```

un paramètre (var. locale)

accès à la variable locale

accès à la variable globale

En réalité l'opérateur permet d'accéder à un nom dans un contexte donné (espace de noms)

pas de contexte (mais opérateur) = contexte global

- Les espaces de noms en C++ :
- les conflits de noms sont inévitables mais 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 = "C5H9N04";  
}  
chimie.hpp
```

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

- Usage du namespace via l'opérateur de portée ::

```
#include <iostream>

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

- Le compilateur retrouve ses petits s'il n'y a pas d'ambiguïté via la mise à plat des espaces de noms :

```
#include <iostream>

using namespace std;

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

- 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;
```


LE PASSAGE DE PARAMÈTRES

- Le C++ possède essentiellement deux modes de passage d'arguments
 - La transmission par valeur (à la C)
 - La transmission par référence (à la `var` du Pascal ou `in out` d'ADA)

Attention : le langage définit plusieurs autres passages mais il ne s'agit que de variantes des modes par valeur et par référence.

- **La transmission par valeur *pass-by-value***
 - création d'une variable locale à la fonction initialisée avec la valeur transmise (création d'une copie)

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

int f(int i, int j) {
    cout << &i << ', ' << &j << endl;
    return i +=j;
}

int main() {
    int a{3}, b{12};
    cout << &a << ', ' << &b << endl;
    cout << a << ', ' << b << endl;
    f(a,b);
    cout << a << ', ' << b << endl;
    return 0;
}
```

```
[Ciboulette] ./test
0x7fff5ac3f978,0x7fff5ac3f974
3,12
0x7fff5ac3f91c,0x7fff5ac3f918
3,12
[Ciboulette]
```


- **La transmission par valeur** *pass-by-value*
- **Attention** : le passage par pointeurs n'existe pas... Les pointeurs sont des types. Passer un pointeur comme paramètre à l'appel ne constitue que la transmission de sa valeur (si on utilise un passage par valeur).

- **La transmission par référence pass-by-reference**
- pour éviter la copie de variables volumineuses, il est possible de transmettre la variable elle-même et non sa valeur
- la variable peut donc être modifiée durant l'appel

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

void echangeNous(int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}

int main() {
    int a{123}, b{321};
    cout << a << ', ' << b << endl;
    echangeNous(a,b);
    cout << a << ', ' << b << endl;
    return 0;
}
```

```
[Ciboulette] ./test3
123,321
321,123
[Ciboulette]
```


- **La transmission par référence constante**
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

```

#include <iostream>
using namespace std;

struct MonType {
    int t[10000];
};

void f(MonType v) { // copie
    cout << &v << endl;
}

void g(const MonType &v) { // pas de copie
    cout << &v << endl;
    // v.t[10] = 123; // interdit!
}

int main() {
    MonType a;
    cout << &a << endl;
    f(a);
    g(a);
    return 0;
}

```

```

[Ciboulette] ./test2
0x7fff54c57c08
0x7fff54c44360
0x7fff54c57c08
[Ciboulette]

```

- Attention, en ce qui concerne les copies :
 - les compilateurs C++ sont explicitement autorisés à utiliser des optimisations pour les éviter
 - RVO : Return Value Optimization
 - CE : Copy Elision
- Consultez la documentation du compilateur si nécessaire!

LES FONCTIONS CONSTEXPR

- On a déjà vu la possibilité de définir des constantes d'expression via `constexpr`
 - équivalent propre de constantes définies par `#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`

- Old C

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

- Old C++

```
#define hypotenuse(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);
```


- 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 vers C++14)