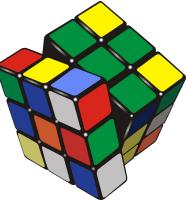
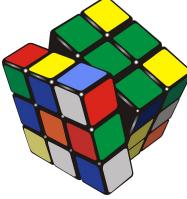


Chapitre 4

Fonctions



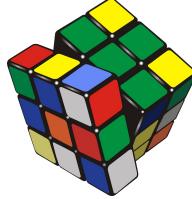
1. Introduction



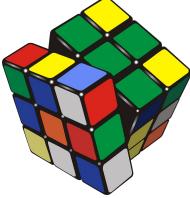
Pourquoi utiliser des fonctions ?

- Quand un programme dépasse quelques pages de texte, il est pratique de le **décomposer** en parties
 - relativement **indépendantes**
 - dont on peut **comprendre** le rôle sans avoir à examiner l'ensemble du code
- La **programmation procédurale** permet un premier pas dans ce sens grâce à la notion de fonction, i.e.
 - un **bloc d'instructions** auquel on donne un **nom**
 - que l'on peut **appeler** depuis un autre point du code
 - éventuellement **plusieurs fois**
 - éventuellement en lui fournissant des **paramètres**

HE^{VD} Fonctions mathématiques vs fonctions IG C++



- En **mathématiques**, une fonction
 - possède des **arguments** dont on fournit la valeur à l'appel `sqrt(x)`
 - fournit un **résultat scalaire** désigné simplement par son appel
`sqrt(x)` désigne le résultat, que l'on peut utiliser `y + 2 * sqrt(x)`
- En **C++**, une fonction peut
 - **modifier** les valeurs de certains **paramètres** transmis
 - réaliser une **action** autre qu'un simple calcul
 - fournir un **résultat non scalaire** (string, structures, objets)
 - fournir une valeur résultat que **l'on n'utilise pas**
 - **ne pas fournir de résultat** du tout
dans certains langages (Pascal, Ada), on parle alors de **procédure**



Exemple - la fonction pow

- Nous avons déjà utilisé des fonctions. Par exemple, `<cmath>` nous fournit la fonction `pow` qui prend deux paramètres : la base et l'exposant
- Le code suivant affiche x^n et y^n pour $x=2$, $y=\frac{1}{2}$ et $n=10$, en appelant deux fois la fonction `pow`

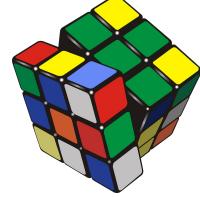
```
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    double n = 10.0;
    double x = 2.0, y = 0.5;

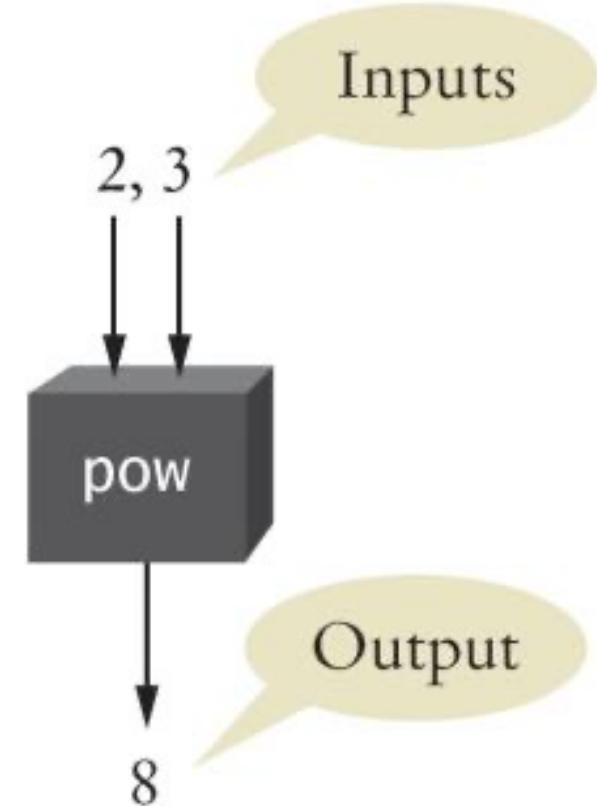
    cout << pow(x, n) << endl;
    cout << pow(y, n) << endl;
}
```

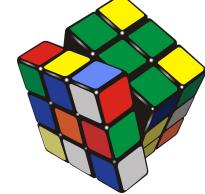
1024
0.000976562

Une boîte noire



- Nous n'avons **pas besoin de savoir comment** le calcul de la puissance est effectué
- Nous pouvons considérer cette fonction comme une **boîte noire** dont le concepteur nous **garantit** que
 - si nous lui fournissons **en entrée** les paramètres **base et exposant**
 - elle nous fournit **en retour** la valeur **base^{exposant}**





function

pow[C90](#) [C99](#) [C++98](#) [C++11](#) [?](#)

<cmath> <ctgmath>

```
double pow (double base      , double exponent);
float pow (float base       , float exponent);
long double pow (long double base, long double exponent);
double pow (Type1 base      , Type2 exponent);           // additional overloads
```

Raise to powerReturns *base* raised to the power *exponent*: $\text{base}^{\text{exponent}}$ [C99](#) [C++98](#) [C++11](#) [?](#)

Additional overloads are provided in this header (<cmath>) for other combinations of arithmetic types (Type1 and Type2): These overloads effectively cast its arguments to `double` before calculations, except if at least one of the arguments is of type `long double` (in which case both are casted to `long double` instead).

This function is also overloaded in <complex> and <valarray> (see [complex pow](#) and [valarray pow](#)).

Parameters**base**

Base value.

exponent

Exponent value.

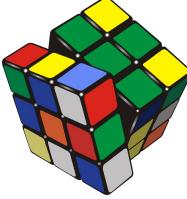
Return ValueThe result of raising *base* to the power *exponent*.

If the *base* is finite negative and the *exponent* is finite but not an integer value, it causes a *domain error*.

If both *base* and *exponent* are zero, it may also cause a *domain error* on certain implementations.

If *base* is zero and *exponent* is negative, it may cause a *domain error* or a *pole error* (or none, depending on the library implementation).

The function may also cause a *range error* if the result is too great or too small to be represented by a value of the return type.



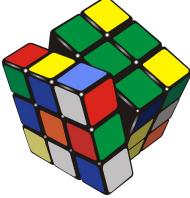
Notre propre fonction puissance

- Réalisons nous-mêmes une fonction puissance simplifiée en ne considérant que des exposants entiers ≥ 0 .
- Le code permettant de calculer $\text{val}=2^{10}$ est assez simple

The diagram illustrates the components of a power calculation program:

- valeurs d'entrée**: Labels the variable declarations `double base = 2.0;` and `int exposant = 10;`.
- résultat à calculer**: Labels the variable declaration `double val = 1;`.
- instructions effectuant le calcul**: Labels the loop body `for (int i = 0; i < exposant; ++i) { val *= base; }`.

- Il faut maintenant le mettre en forme pour qu'il soit **appelable** via `puissance(2.0, 10)`



Notre propre fonction puissance

- Mis sous la forme d'une fonction, ce code devient

type de la
valeur de retour nom
paramètres

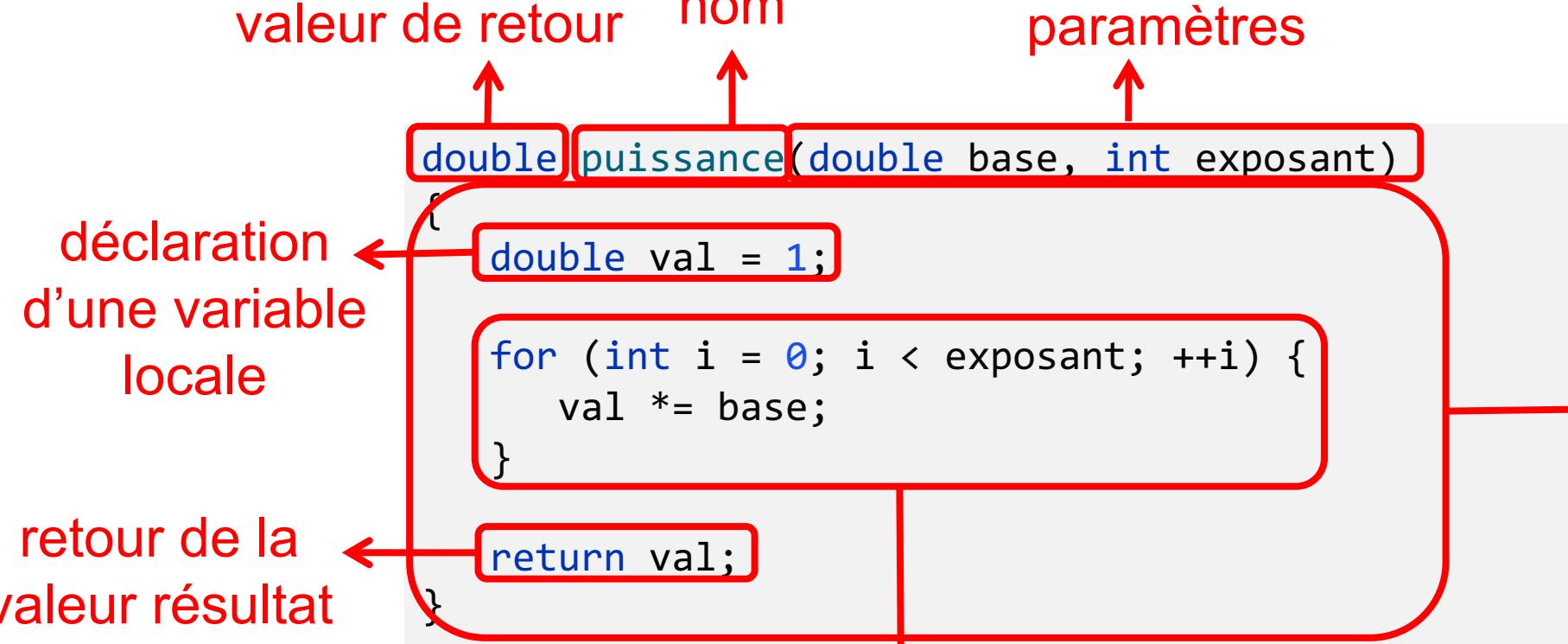
```
double puissance(double base, int exposant)
```

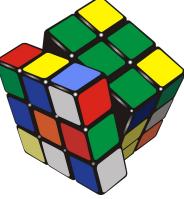
déclaration
d'une variable
locale

retour de la
valeur résultat

corps de
la
fonction

instructions





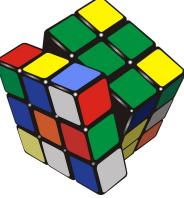
La fonction main()

- Notons que depuis notre tout premier programme, nous écrivons une fonction : la fonction `main`

```
int main() {  
    ...  
    return EXIT_SUCCESS;  
}
```

- Elle ne prend pas de **paramètres** et retourne une valeur entière
Elle garantit de **retourner EXIT_SUCCESS** (qui correspond à 0)
si tout se passe bien, et une valeur non nulle sinon.
- Il en existe aussi une version avec des paramètres d'entrée (voir PRG2)

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



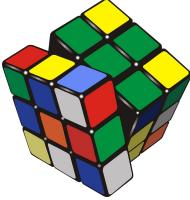
void

- Une fonction peut aussi ne pas fournir de valeur en retour
- On l'indique en utilisant le type **void** comme type de retour

```
void afficher(int val, int width) {
    cout << "|" << setw(width)
        << val << "|" << endl;
}

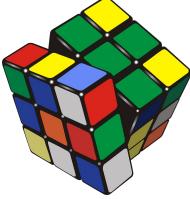
int main() {
    afficher(1, 6);
    afficher(2, 6);
    return EXIT_SUCCESS;
}
```

	1
	2



2. Passage des paramètres

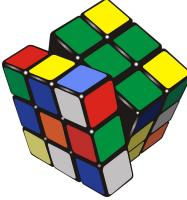
- a) par valeur
- b) par référence
- c) par adresse



Passage des paramètres

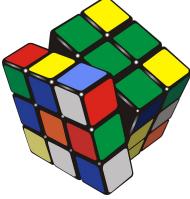
- Les paramètres permettent de **transmettre des informations entre le code appelant la fonction et la fonction elle-même**
- On peut **transmettre ces paramètres**
 - **par valeur** copie une variable du code appelant dans une variable de la fonction
 - **par référence** accès à une variable du code appelant depuis la fonction
 - **par adresse** accès à une variable du code appelant depuis la fonction par son adresse

NB : dans les deux derniers cas, un passage **constant** est possible



2. Passage des paramètres

- a) ... par valeur
- b) ... par référence
- c) ... par adresse



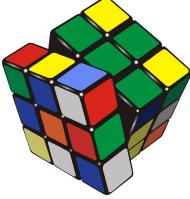
Passage par valeur

```
void echanger(int a, int b) {
    int t;
    cout << "debut echange: a = " << a << " , b = " << b << endl;
    t = a; a = b; b = t;
    cout << "fin echange : a = " << a << " , b = " << b << endl;
}

int main() {
    int c = 3, d = 5;
    cout << "avant echange: c = " << c << " , d = " << d << endl;
    echanger(c, d);
    cout << "apres echange: c = " << c << " , d = " << d << endl;
}
```

- Qu'affiche ce code ?

```
avant echange: c = 3 , d = 5
debut echange: a = 3 , b = 5
fin echange : a = 5 , b = 3
apres echange: c = 3 , d = 5
```

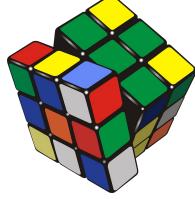


Passage par valeur

- Dans ce code, les paramètres a et b sont **passés par valeur**

```
void echanger(int a, int b) {...}
```

- a et b sont des **variables locales** de la fonction **echanger**, initialisées à l'appel de la fonction en **copiant** les valeurs transmises par le code appelant
- Ces valeurs et leurs types sont éventuellement **convertis** à l'appel en suivant les mêmes règles que pour l'opérateur d'affectation
- Modifier** les valeurs de a ou b **dans la fonction** n'a **aucun effet** sur les valeurs dans le code appelant
- Le passage par valeur permet uniquement de transmettre des paramètres **en entrée, pas en sortie !**

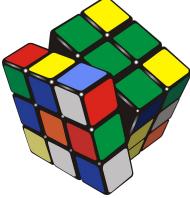


- Le passage par valeur requiert **l'évaluation des valeurs transmises** pour les affecter aux variables des paramètres
 - Note : l'ordre d'évaluation n'est pas spécifié par le standard C++
Le code suivant, par exemple, peut donc avoir un **comportement différent d'un compilateur à l'autre**

```
void f(int a, int b) {  
    cout << a << b;  
}  
  
int main() {  
    int i = 2;  
    f(i++, i++);  
}
```

g++ : affiche warning : opération sur
'i' peut être undefined à la compilation
et 32 à l'exécution

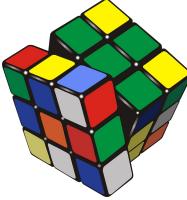
clang : affiche warning: multiple
unsequenced modifications to 'i'
[-Wunsequenced] à la compilation et 23
à l'exécution



Passage par valeur constante

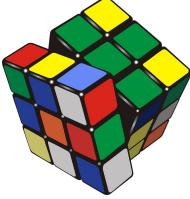
- Comme pour les variables, on peut utiliser le mot `const` pour indiquer qu'un paramètre n'est pas modifié par le code de la fonction
- Note : pour les paramètres passés **par valeur**, cela ne change rien du point de vue de l'appelant – donc il n'est pas utile d'utiliser `const` dans ce cas

```
void f(const int a, const int b) {  
    a = a * 2; // ne compile pas!!  
  
    cout << a << ' ' << b << endl;  
}
```



2. Passage des paramètres

- a) ... par valeur
- b) ... par référence
- c) ... par adresse



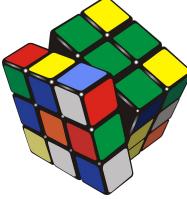
Passage par référence

```
void echanger(int& a, int& b) {
    int t;
    cout << "debut echange: a = " << a << " , b = " << b << endl;
    t = a; a = b; b = t;
    cout << "fin echange : a = " << a << " , b = " << b << endl;
}

int main() {
    int c = 3, d = 5;
    cout << "avant echange: c = " << c << " , d = " << d << endl;
    echanger(c, d);
    cout << "apres echange: c = " << c << " , d = " << d << endl;
    return EXIT_SUCCESS;
}
```

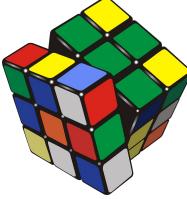
- Qu'affiche ce code ?

```
avant echange: c = 3 , d = 5
debut echange: a = 3 , b = 5
fin echange : a = 5 , b = 3
apres echange: c = 5 , d = 3
```



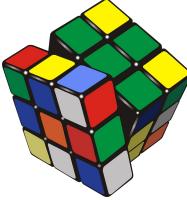
Passage par référence

- Pour pouvoir modifier des variables du code appelant depuis une fonction, il faut utiliser un autre mécanisme : le **passage par référence**
- On l'indique en ajoutant le caractère **&** au **type** du paramètre
- Le paramètre est un **synonyme** de la variable passée par le code appelant, donc modifier ce paramètre **modifie la variable dans le code appelant**
- Le **passage par référence** permet donc d'utiliser les paramètres **en entrée ou en sortie**
- **Attention** : le paramètre effectif (lors de l'appel) doit être **une variable** et doit être **du même type** que celui du paramètre formel



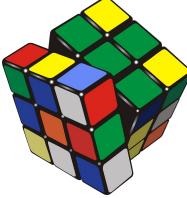
Passage par référence constante

- Comme pour les variables, on peut utiliser le mot `const` pour indiquer qu'un paramètre n'est pas modifié par le code de la fonction
- Pour les paramètres passés par référence, cela a deux impacts majeurs pour l'appelant
 - Il sait que la variable passée par référence ne sera pas modifiée
 - Il ne subit pas les restrictions d'appel (variable et type) du passage par référence → il peut y avoir une conversion implicite si nécessaire



2. Passage des paramètres

- a) ... par valeur
- b) ... par référence
- c) ... par adresse



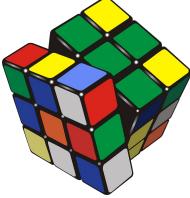
Adresse d'une valeur

- Comme vu au chapitre 2, l'opérateur & retourne l'adresse d'une variable ou d'une constante. Reprenons tout ceci dans le cadre d'une fonction

```
void adresse(int* adr) {  
    cout << " adr : " << adr << endl;  
    cout << "*adr : " << *adr << endl;  
    cout << "&adr : " << &adr << endl;  
}  
  
int main() {  
    int valeur = 3;  
    cout << " valeur : " << valeur << endl;  
    cout << "&valeur : " << &valeur << endl;  
    adresse(&valeur);  
}
```

valeur	:	3
&valeur	:	0x16cf7f568
adr	:	0x16cf7f568
*adr	:	3
&adr	:	0x16cf7f4d8

Note : dans notre fonction, le paramètre adr contient une adresse ... et est lui-même stocké à une adresse ;)



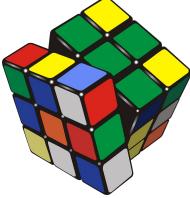
Passage par adresse

```
void echanger(int* a, int* b) {
    int t;
    cout << "debut echange: a = " << *a << " , b = " << *b << endl;
    t = *a; *a = *b; *b = t;
    cout << "fin echange : a = " << *a << " , b = " << *b << endl;
}

int main() {
    int c = 3, d = 5;
    cout << "avant echange: c = " << c << " , d = " << d << endl;
    echanger(&c, &d);
    cout << "apres echange: c = " << c << " , d = " << d << endl;
}
```

- Qu'affiche ce code ?

```
avant echange: c = 3 , d = 5
debut echange: a = 3 , b = 5
fin echange : a = 5 , b = 3
apres echange: c = 5 , d = 3
```



Passage par adresse

Pour mémoire ...

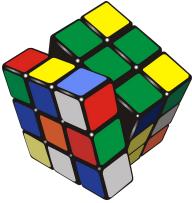
- Pour des raisons de sécurité, une adresse est typée (`int*` , `string*`, ...). Le pointeur doit être du type que la valeur pointée, sous peine d'erreur de compilation

```
int    entier  = 3;
double reel    = 2.7;
int*   ptr_ = &reel;           error: cannot initialize a variable of type
                                'int *' with an rvalue of type 'double *'
```

- Un pointeur peut traiter la valeur pointée comme constante (accès en lecture seule)

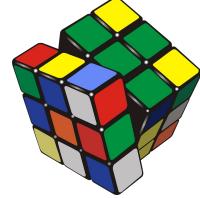
```
const int CSTE      = 2;
        int valeur  = 2;
const int* adr1 = &valeur; // accès en lecture seule sur une variable
        int* adr2 = &valeur; // accès en lecture/écriture sur une variable
const int* adr3 = &CSTE;   // accès en lecture seule sur une constante
        int* adr4 = &CSTE;   // accès en écriture sur une constante?
```

error: cannot initialize a variable of type
'int *' with an rvalue of type 'const int *'



3. Types des paramètres lors des appels

Conversion des types lors du passage par valeur



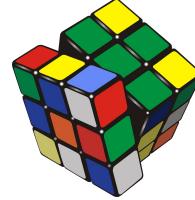
```
void afficher(int p1, char p2) {
    cout << "p1 = " << p1 << " , p2 = " << p2 << endl;
}

int main() {
    const char CAR = 'A';
    afficher(1, 'A');
    afficher(1, 65);
    afficher('A', 67);
    afficher(CAR, CAR + 1);
    return EXIT_SUCCESS;
}
```

p1 = 1 , p2 = A
p1 = 1 , p2 = A
p1 = 65 , p2 = C
p1 = 65 , p2 = B

- Qu'affiche ce code ?

Types des paramètres et passage par référence



- Attention, le passage par référence restreint l'appel de la fonction, qui doit passer une lvalue du type exact spécifié

```
void f(int& p1, char& p2) {
    cout << p1 << p2 << endl;
}
```

```
int main() {
    const int ci = 65;
    const char cc = 'B';
    int vi = 67;
    char vc = 'D';
    f(vi, vc); // Appel valide
    f(vc, vi);
    f(ci, vc);
    f(vi, cc);
    f(67, vc);
    f(vi, 'E');
}
```

error: no matching function for call to 'f'
 note: candidate function not viable: no known conversion from 'char' to 'int &' for 1st argument

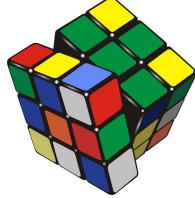
error: no matching function for call to 'f'
 note: candidate function not viable: 1st argument ('const int') would lose const qualifier

error: no matching function for call to 'f'
 note: candidate function not viable: 2nd argument ('const char') would lose const qualifier

error: no matching function for call to 'f'
 note: candidate function not viable: expects an lvalue for 1st argument

error: no matching function for call to 'f'
 note: candidate function not viable: expects an lvalue for 2nd argument

Conversion des types lors du passage par référence constante



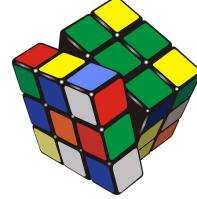
```
void afficher(const int& p1, const char& p2) {
    cout << "p1 = " << p1 << " , p2 = " << p2 << endl;
}

int main() {
    const int CI = 65;
    const char CC = 'B';
    int vi = 67;
    char vc = 'D';

    afficher(vi, vc);      // correct
    afficher(vc, vi);      // correct
    afficher(CI, vc);      // correct
    afficher(vi, CC);      // correct
    afficher(67, vc);      // correct
    afficher(vi, 'E');      // correct
}
```

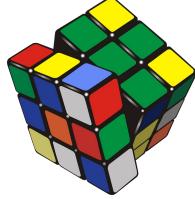
p1 = 67 , p2 = D
p1 = 68 , p2 = C
p1 = 65 , p2 = D
p1 = 67 , p2 = B
p1 = 67 , p2 = D
p1 = 67 , p2 = E

Types des paramètres lors du passage par référence constante



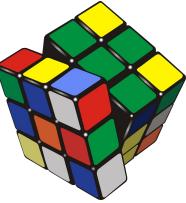
- Avec le **passage par référence constante**, on peut transmettre sans copie
 - des **variables** du même type
 - des **constantes** du même type
 - des **constantes littérales** du même type
- On peut également transmettre, au prix d'une conversion et copie dans une variable temporaire cachée
 - des **variables de type convertible**
 - des **constantes de type convertible**
 - des **constantes littérales de type convertible**
 - et de manière générale, **toute expression de type convertible**

Usage du passage par référence constante

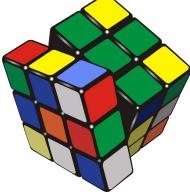


- Pourquoi passer par référence constante plutôt que par valeur ?
 - Pour les types simples, cela n'a pas vraiment d'intérêt
 - Pour les types composés (string, tableaux, classes créées par vous) on économise une copie (si pas de conversion) qui peut être coûteuse
- Pourquoi passer par référence constante plutôt que par référence simple ?
 - Pour éviter les restrictions d'appel du passage par référence

```
✗ void afficher(string message) { ... }  
✗ void afficher(string& message) { ... }  
✓ void afficher(const string& message) { ... }
```



4. return



Valeur de retour

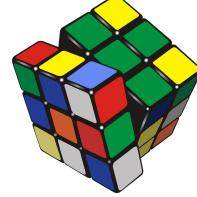
- Le mot-clé `return` fournit le résultat à l'appelant via la syntaxe

```
return EXPRESSION;
```

- Il interrompt l'exécution de la fonction de manière similaire à `break` pour une boucle
- Si nécessaire, EXPRESSION est convertie dans le type annoncé comme résultat de la fonction

```
bool estImpair(int valeur) {  
    // implicitement  
    return valeur % 2;  
}
```

```
bool estImpair(int valeur) {  
    // explicitement  
    return bool(valeur % 2);  
}
```



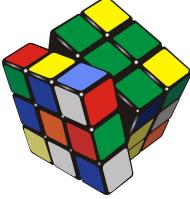
- Les instructions placées après un `return` ne sont jamais exécutées

```
bool estImpair(int valeur) {  
    return valeur % 2;  
    ...  
    cout << "inutile";  
    ...  
}
```

- Plusieurs `return` peuvent être présents en général soumis à des conditions

À utiliser intelligemment

```
int valeur(char car) {  
    switch (car) {  
        case '0': return 0;  
        ...  
        case '9': return 9;  
        default : return -1;  
    } // switch  
}
```

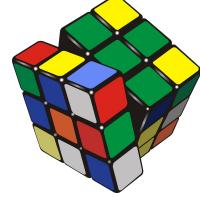


Valeur de retour

- On peut aussi utiliser `return` sans EXPRESSION selon la syntaxe

```
return;
```

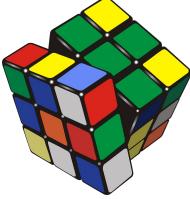
- L'exécution est interrompue et aucune valeur n'est renvoyée
- Ce comportement est correct si le type de retour est `void`, sinon le comportement est indéfini
- Si l'on atteint l'accolade } de fin de corps de fonction sans avoir rencontré de `return`, c'est ce `return` sans EXPRESSION qui est utilisé implicitement
- Une exception à cette règle : pour la fonction principale `main`, la ligne `return 0;` est ajoutée implicitement en fin de fonction



- Notons que l'on peut **retourner une référence**
- Cela permet d'utiliser ce résultat **à gauche** d'une affectation
- Ne pas retourner de référence à une **variable locale**, mais uniquement
 - à une variable globale
 - à une référence passée en paramètre
 - à une variable allouée dynamiquement

```
int& plusPetit(int& a, int& b) {  
    return a < b ? a : b;  
}  
  
int main() {  
    int a = 5;  
    int b = 7;  
    int c = plusPetit(a, b);  
    int& d = plusPetit(a, b);  
    plusPetit(a, b) = 12;  
    d = 10;  
  
    cout << "a = " << a  
        << ", b = " << b  
        << ", c = " << c;  
}
```

a = 10, b = 7, c = 5



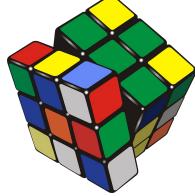
Retour par référence – application

- La méthode `at(size_t pos)` de la classe `string` admet ce type de comportement

```
string s("Hello");
char c = s.at(0);           // at() à droite
s.at(0) = (char) tolower(c); // at() à gauche
cout << s << endl;
```

hello

- Il est possible de la placer à gauche d'une affectation parce qu'elle retourne le type `char&`
- Notons que si `s` était une `const string`, `s.at(0)` retournerait alors une `const char&`



- Nous avons vu qu'il est possible d'**enchaîner** l'opérateur d'affectation = car il **retourne la lvalue** dans laquelle on a affecté
- On peut donner la même valeur à a, b, c et d en écrivant

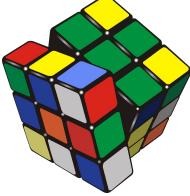
```
a = b = c = d = 5;
```

- Cela fonctionne parce que l'affectation **s'évalue de droite à gauche**
Cette ligne est équivalente à

```
a = (b = (c = (d = 5)));
```

- Par contre, si l'on force à évaluer dans l'ordre contraire, seule la variable a sera modifiée et égale à **5**

```
((a = b) = c) = d) = 5;
```



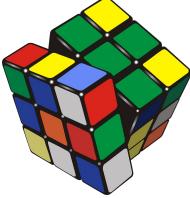
Retour par référence – application

- Par contre, les opérateurs de flux **s'évaluent de gauche à droite**
Les deux lignes suivantes sont équivalentes

```
cout << "Hello" << ',' << " World!" << endl;
```

```
((cout << "Hello") << ',') << " World!") << endl;
```

- L'enchaînement d'opérations de flux est possible car l'opérateur << **retourne une référence** vers son opérande gauche, ici le flux cout (qui est un « objet », voir chapitre 7)



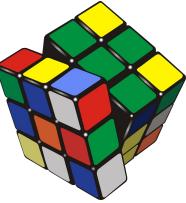
Valeur de retour et constexpr

- On peut initialiser une variable `constexpr` avec la valeur de retour d'une fonction, à condition que celle-ci **retourne une valeur déclarée `constexpr` et soit évaluable à la compilation**
- Cela permet d'effectuer cet appel à la compilation et d'en **stocker le résultat dans le code** du programme → exécution plus rapide
- En revanche, les paramètres ne sont jamais déclarés `constexpr`

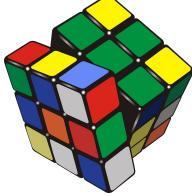
```
constexpr int f(int a, int b) { return a + b; }
    int g(int a, int b) { return a + b; }

int main() {
    int a = 1, b = 2;
    constexpr int c1 = f(1, 2); // OK
    constexpr int c2 = f(a, b); // ne compile pas
    constexpr int c3 = g(1, 2); // ne compile pas
    constexpr int c4 = g(a, b); // ne compile pas
}
```

```
error: constexpr variable
'c2' must be initialized by
a constant expression
error: constexpr variable
'c3' must be initialized by
a constant expression
error: constexpr variable
'c4' must be initialized by
a constant expression
```

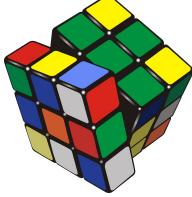


5. Récursivité



- Parfois des problèmes sont plus simples à exprimer de manière **récursive**
- Exemple : $\text{factoriel}(n)$ ou $n!$
 - $5! = 1 \times 2 \times 3 \times 4 \times 5$
 - $6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \quad \Rightarrow 6! = 5! \times 6$
 - $n! = (n - 1)! \times n$
avec $n \geq 0$

```
int facto(int n) {  
  
    // cas trivial  
    if (n <= 1)  
        return 1;  
  
    // appel récursif  
    return n * facto( n - 1 );  
}
```



- Il est essentiel de respecter des conditions d'implémentation

- Une fonction récursive s' **appelle elle-même**

```
return n * facto(...);
```

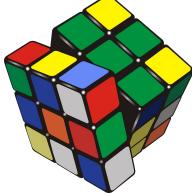
- Une **condition d'arrêt** qui stoppe les appels récursifs et ici retourne une valeur

```
if (n <= 1)  
    return 1;
```

- Les paramètres tendent **vers la condition d'arrêt**

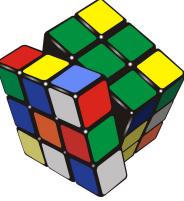
```
return n * facto( n - 1 );
```

Une fonction récursive est en général **très courte**



- Il est souvent préférable d'opter pour **implémentation itérative** lorsque celle-ci est possible sans être trop compliquée ;)

```
int facto(int n) {  
  
    int r = 1;  
  
    for (; n > 1; --n) {  
        r *= n;  
    }  
  
    return r;  
}
```



Récursivité

- Un autre exemple très connu est la **suite de Fibonacci**

n	0	1	2	3	4	5	6	7	8	9	10	11	12	...
f(n)	0	1	1	2	3	5	8	13	21	34	55	89	144	...

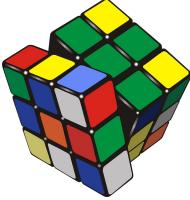
Ou plus simplement

$$\begin{aligned}f(0) &= 0 \text{ et } f(1) = 1 \\f(n) &= f(n-2) + f(n-1)\end{aligned}$$

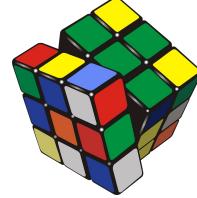
```
int fibo(int n) {
    // cas trivial
    if (n < 2)
        return n;

    // appels récursifs
    return fibo(n-2) + fibo(n-1);
}
```

- Ce sujet sera repris en détails dans le cours d'ASD : Algorithme et Structures de Données



6. Prototypes et compilation séparée

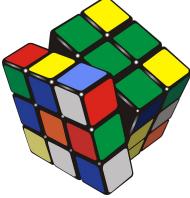


- Tout comme on ne peut utiliser une variable avant de l'avoir déclarée, on ne **peut pas appeler une fonction avant de l'avoir déclarée**
- Le code suivant **ne compile pas**
- Une solution **simple** consiste à **définir la fonction avant de l'utiliser**

```
int main() {  
    int a = f(0);  
}  
  
int f(int val) {  
    return val + 42;  
}
```

error: use of undeclared
identifier 'f'

```
int f(int val) {  
    return val + 42;  
}  
  
int main() {  
    int a = f(0);  
}
```



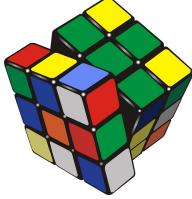
La solution simple ne suffit pas

- La solution « simple » est dans les faits compliquée à mettre en œuvre.
- Le programmeur doit trier les fonctions dans un ordre qui garantit que toute fonction utilisée par un autre soit placée avant celle-ci.
- Ce tri est impossible dans le cas ci-contre où `f()` appelle `g()` et vice-versa.

```
int f(int x) {
    if(x <= 0) return x;
    return g(x-1) + 1;
}

int g(int x) {
    if(x <= 0) return x;
    return f(x / 2) * 2;
}

int main() {
    cout << f(1000) << endl;
}
```



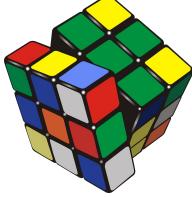
La solution propre consiste à **séparer**

- la **déclaration** de la fonction, son **prototype**, qui exprime l'interface qu'elle propose à ses appellants
- de la **définition** de la fonction, qui indique comment la fonction est mise en œuvre

```
int f(int val);    // déclaration = prototype

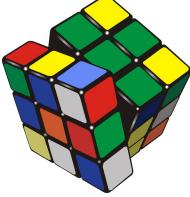
int main() {
    int a = f(0);
}

int f(int val) {    // définition
    return val + 42;
}
```



- Dans le prototype d'une fonction, les **noms des paramètres** sont **facultatifs**
 - Ils peuvent être **omis** ou **différer** des noms utilisés dans la définition ...
:(mauvaise pratique !
 - Ils servent uniquement à **documenter** le code ce qui est essentiel
- Les deux lignes ci-contre sont donc équivalentes

```
int f(int val1, double val2);  
int f(int, double);
```
- Ce qui importe pour le compilateur, c'est uniquement
 - le **type** des paramètres
 - leur **ordre**



Valeurs par défaut des paramètres

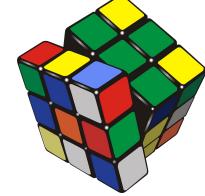
- Le prototype – mais pas la définition – peut également préciser des valeurs par défaut pour certains paramètres
- Cela permet d'appeler la fonction avec une liste de paramètres réduite, en omettant un ou plusieurs des paramètres *les plus à droite*

```
void f(int i1, int i2 = 42, double d = 0.);

void f(int i1, int i2, double d) {
    cout << i1 << " " << i2 << " " << d << endl;
}

int main() {
    f(10, 20, 30);
    f(10, 20);
    f(10);
}
```

10 20 30
10 20 0
10 42 0



La **séparation** entre **prototype** et **définition** des fonctions permet de **découper** le code en plusieurs fichiers

```
int f(int val);

int main() {
    int a = f(0);
}

int f(int val) {
    return val + 42;
}
```

maFonction.h

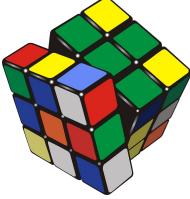
```
int f(int val);
```

main.cpp

```
#include "maFonction.h"
int main() {
    int a = f(0);
}
```

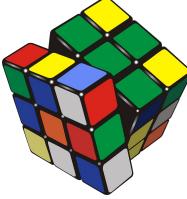
maFonction.cpp

```
#include "maFonction.h"
int f(int val) {
    return val + 42;
}
```



Compilation séparée

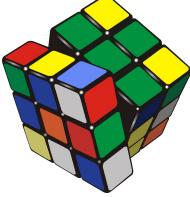
- Les déclarations de **prototypes** sont situées dans des **fichiers d'en-tête** (header, d'où l'extension **.h** ou **.hpp**)
- On peut **include** ces fichiers avec la directive de compilation **#include**, suivie du nom de fichier entre **""**, ce qui les distingue des en-têtes de la bibliothèque standard que l'on entoure de **<>**
- Les **définitions** de fonctions, y compris de la fonction principale **main**, sont situées dans des fichiers **.cpp**
- Chaque fichier **.cpp** inclut les fichiers d'en-têtes déclarant les fonctions qu'il
 - **utilise**
 - **définit**



Compilation séparée

La compilation s'effectue **en deux étapes**

- Le compilateur compile chaque fichier .cpp séparément pour créer un **module objet** (.o ou .obj)
 - toutes les fonctions utilisées par un fichier .cpp doivent être **déclarées avant d'être utilisées**
 - les fonctions utilisées par un fichier n'ont **pas besoin d'y être définies**
- L'éditeur de liens regroupe ces modules objets ainsi que ceux de la bibliothèque standard dont il a besoin
 - toutes les fonctions utilisées **doivent être définies**
 - **une et une seule fois**



#include imbriqués

- Les fichiers d'en-tête peuvent eux-mêmes inclure d'autres fichiers d'en-tête.
Par exemple, si votre en-tête déclare la fonction

```
void afficher(const std::string& s);
```

votre fichier doit contenir avant cette déclaration la ligne

```
#include <string>
```

qui déclare le type `std::string` avant que vous ne l'utilisiez

- Mais comment éviter les boucles d'inclusion infinies ?

header1.h

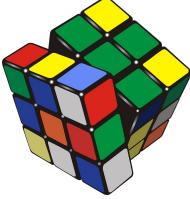
```
#include "header2.h"
```

...

header2.h

```
#include "header1.h"
```

...



#define, #ifdef, #ifndef, #else, #endif

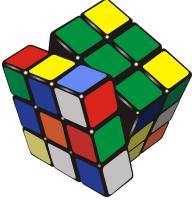
- Solution : utiliser des **directives du préprocesseur**
 - **#ifndef** inclut ce qui suit jusqu'à la directive **#endif** si le symbole n'est pas défini
 - **#define** définit le symbole qui suit
- La première fois que le fichier est inclus, le symbole n'est pas défini et tout le texte entre **#ifndef** et **#endif** est inclus.
- S'il est inclus d'autres fois, le symbole est déjà défini et le texte entre **#ifndef** et **#endif** est ignoré
- Attention, le symbole doit être unique. L'IDE vous suggère normalement un nom basé sur le nom du fichier d'en-tête et éventuellement de votre projet

maFonction.h

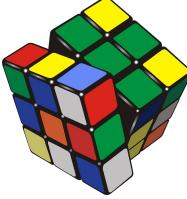
```
#ifndef MAFONCTION_H
#define MAFONCTION_H

int f(int val);

#endif
```

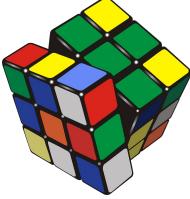


7. Variables locales, globales et statiques



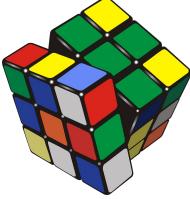
Variables locales

- Comme pour tout bloc d'instructions,
on peut déclarer des variables dans le corps d'une fonction
- Ces **variables locales**
 - ne sont **visibles** que depuis l'intérieur de la fonction
 - **cachent** éventuellement des variables de même nom déclarées ailleurs
 - sont **créées automatiquement**, à chaque fois que l'on appelle la fonction
 - **disparaissent automatiquement**, à chaque fois que l'on sort de la fonction
- Ces propriétés nous aident à faire de nos fonctions des **boîtes noires**



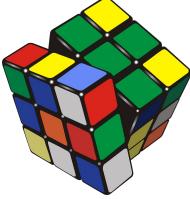
Variables globales

- Il est également possible de déclarer des variables en dehors de tout bloc de toute fonction
- Ces variables sont appelées globales et elles...
 - sont visibles depuis tout le code figurant après la déclaration de la variable globale
 - peuvent être cachées par une variable locale du même nom
 - sont créées statiquement, une seule fois en début de programme et sont initialisées à zéro (0, 0., false, '\0') par défaut
 - ne disparaissent qu'à la fin de l'exécution du programme



Variables globales

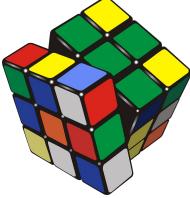
- L'utilisation de variables globales est en général une **mauvaise pratique**
- En effet, une variable globale
 - peut être modifiée par toutes les fonctions
 - ce qui rend sa valeur difficile à prédire
 - et est contraire au concept de boîte noire
- La librairie standard définit cependant quelques variables globales, telles que les « flux » nommés `cin`, `cout`, `cerr`, ... car
 - il n'y a qu'un seul de ces flux dans un programme
 - il doit être accessible depuis toutes les fonctions



Variables globales et compilation séparée

Où placer une déclaration de variable globale en compilation séparée ?

- Dans un fichier .cpp
 - mais alors elle n'est pas visible depuis les autres fichiers .cpp
- Dans un fichier d'en-tête
 - mais alors elle est déclarée plusieurs fois si cet en-tête est inclus par plusieurs fichiers .cpp, ce que refuse l'éditeur de liens, qui n'accepte pas ces symboles dupliqués
- Comment résoudre cette contradiction ?



- La solution nous est fournie par le mot clé **extern** qui indique qu'une **variable globale** est déclarée ailleurs
- Le mécanisme est identique à celui des prototypes pour les fonctions

```
int variableGlobale = 5;

int main() {
    int a = variableGlobale;
}
```

module.h

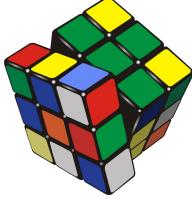
```
extern int variableGlobale;
```

main.cpp

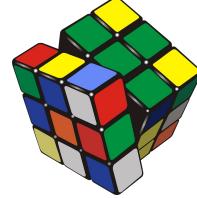
```
#include "module.h"
int main() {
    int a = variableGlobale;
}
```

module.cpp

```
#include "module.h"
int variableGlobale = 5;
```



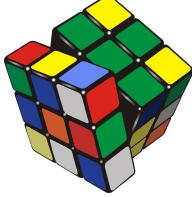
- Le mot clé **static** permet de créer un troisième type de variables, **hybride entre locales et globales**
- Une variable locale statique dans une fonction
 - **visibilité** identique à une variable **locale**
 - ❖ donc visible uniquement **depuis l'intérieur de la fonction**
 - **durée de vie** identique à une variable **globale**
 - ❖ est **créée statiquement** au début du programme
 - ❖ ne disparaît pas en sortie de fonction (sa valeur est mémorisée)



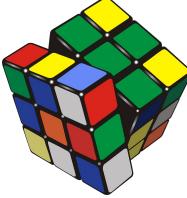
- Une variable locale statique permet par exemple de **compter le nombre d'appels à une fonction**
- Comme les variables globales, une variable statique est **initialisée à zéro par défaut**

```
void f();  
  
void f() {  
    static int compteur;  
    compteur++;  
    cout << "appel #" << compteur << endl;  
}  
  
int main() {  
    for (int i = 0; i < 5; ++i) {  
        f();  
    }  
}
```

appel #1
appel #2
appel #3
appel #4
appel #5



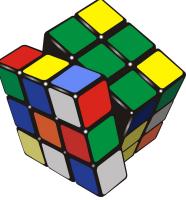
- Notons que le mot clé **static** peut aussi **qualifier une variable globale**
- Dans ce cas, il indique que cette variable n'est **visible que depuis le fichier .cpp** qui la contient. Il est impossible d'y accéder depuis ailleurs, même en utilisant le mot clé **extern**
- Par contre, dans le fichier .cpp où elle est déclarée, elle **se comporte comme toute autre variable globale**



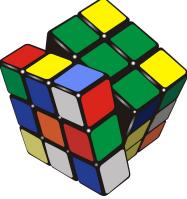
Méthodes d'allocation

Pour être complet, signalons que C++ dispose de 3 manières d'allouer de la mémoire pour y stocker des données

- **Automatique**
 - pour les variables locales
 - existent pendant la durée d'exécution du bloc où elles sont déclarées.
- **Statique**
 - pour les variables globales, statiques, et les constantes littérales
 - existent pendant toute la durée du programme
- **Dynamique**
 - créées et effacées explicitement par le programmeur avec les instructions `new` et `delete` (chap. 15 – PRG1)



8. Eléments pour une bonne conception du code

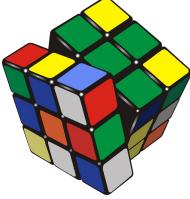


Eviter la duplication de code

- Quand un code est répété presque à l'identique, il faut sans doute le remplacer par une fonction

```
int heures;
do {
    cout << "Entrez un entier entre 0 et 23: ";
    cin >> heures;
} while (heures < 0 || heures > 23);
```

```
int minutes;
do {
    cout << "Entrez un entier entre 0 et 59: ";
    cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

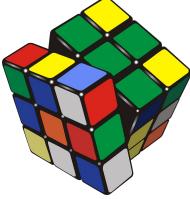


Eviter la duplication de code

- Si l'on occulte les différences, l'essentiel du code des 2 boucles est identique, et les parties occultées devront être des paramètres ou valeurs de retour de la fonction

```
int heures;
do {
    cout << "Entrez un entier entre 0 et __: ";
    cin >> _____;
} while (_____ < 0 || _____ > __);
```

```
int minutes;
do {
    cout << "Entrez un entier entre 0 et __: ";
    cin >> _____;
} while (_____ < 0 || _____ > __);
```

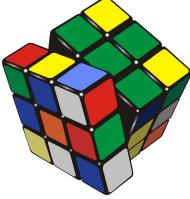


Eviter la duplication de code

- Par ailleurs, on note que 23 et 59 sont des **entrées** des blocs, tandis que heures et minutes en sont des **sorties**

```
int heures;
do {
    cout << "Entrez un entier entre 0 et 23: ";
    cin >> heures;
} while (heures < 0 || heures > 23);
```

```
int minutes;
do {
    cout << "Entrez un entier entre 0 et 59: ";
    cin >> minutes;
} while (minutes < 0 || minutes > 59);
```



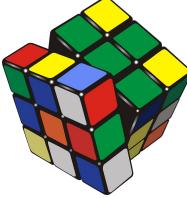
Eviter la duplication de code

- On obtient donc la fonction

```
int lireUnEntierJusquA(int maxVal) {  
    int input;  
    do {  
        cout << "Entrez un entier entre 0 et " << maxVal << ":";  
        cin >> input;  
    } while (input < 0 || input > maxVal);  
    return input;  
}
```

- Avec les appels correspondants

```
const int heures = lireUnEntierJusquA(23);  
const int minutes = lireUnEntierJusquA(59);
```

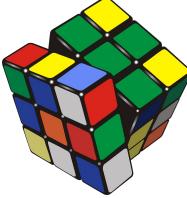


Eviter la duplication de code

- Mais on peut sans doute rendre cette fonction encore **plus réutilisable**

```
int lireUnEntierEntre(int minValue, int maxValue) {
    int input;
    do {
        cout << "Entrez un entier entre " << minValue << " et " << maxValue << ": ";
        cin >> input;
    } while (input < minValue || input > maxValue);
    return input;
}
```

```
const int heures = lireUnEntierEntre(0, 23);
const int minutes = lireUnEntierEntre(0, 59);
const int mois = lireUnEntierEntre(1, 12);
```



Approche descendante - raffinement

Décomposer un problème complexe
en tâches plus simples

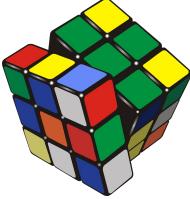
(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)

(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)

(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)

(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)

Jusqu'à ce que les tâches soient réellement simples à mettre en œuvre



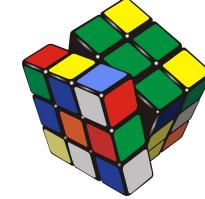
Approche descendante

Combien de jours séparent deux dates ?

Calculer
nbre de jours
entre 2 dates

JANUARY	FEBRUARY	MARCH	APRIL
S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28	S M T W T F S 31 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
MAY	JUNE	JULY	AUGUST
S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	S M T W T F S 30 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
SEPTEMBER	OCTOBER	NOVEMBER	DECEMBER
S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	S M T W T F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Approche descendante



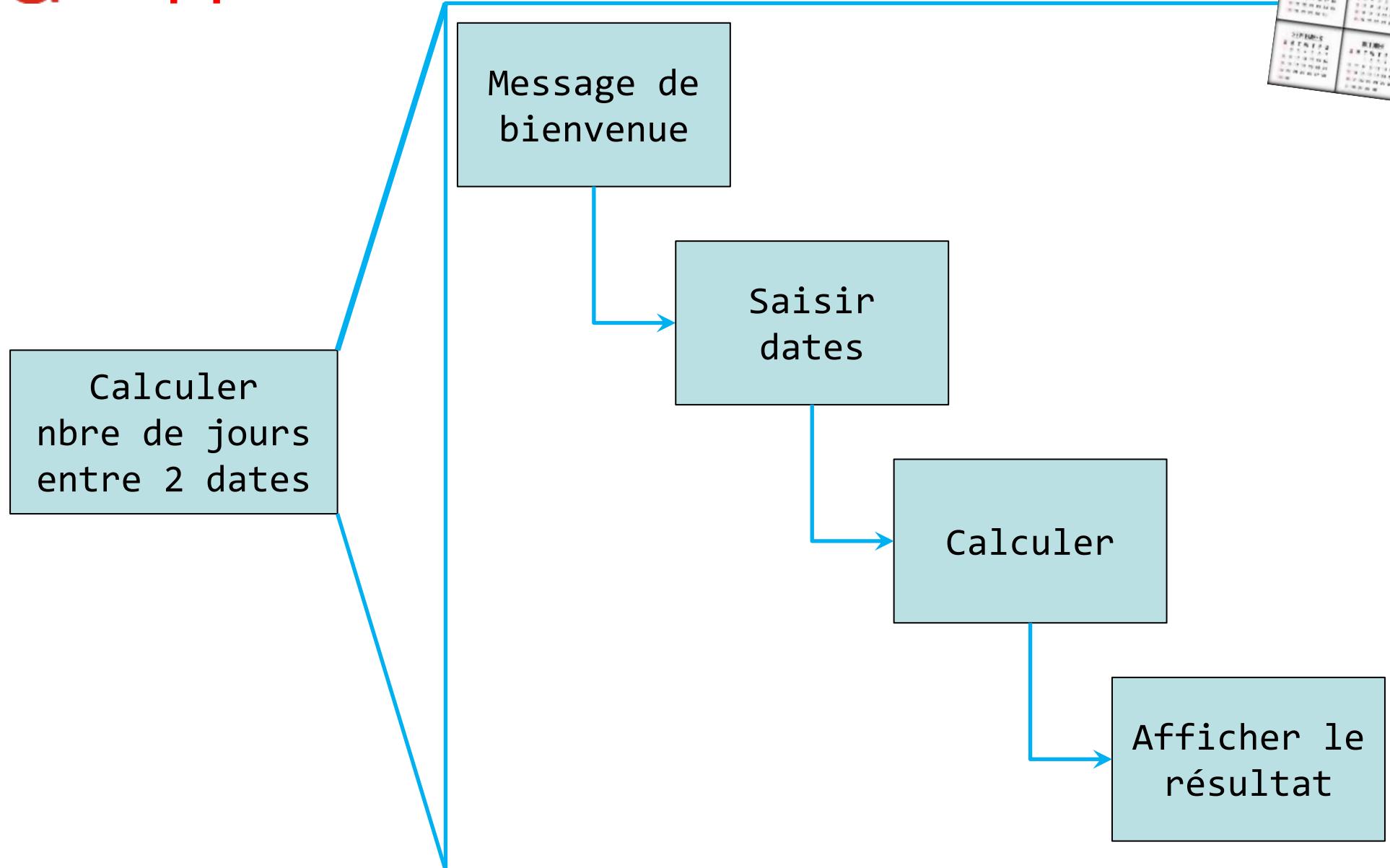
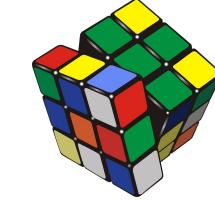
Combien de jours séparent deux dates ?

La question est simple ... mais en y réfléchissant

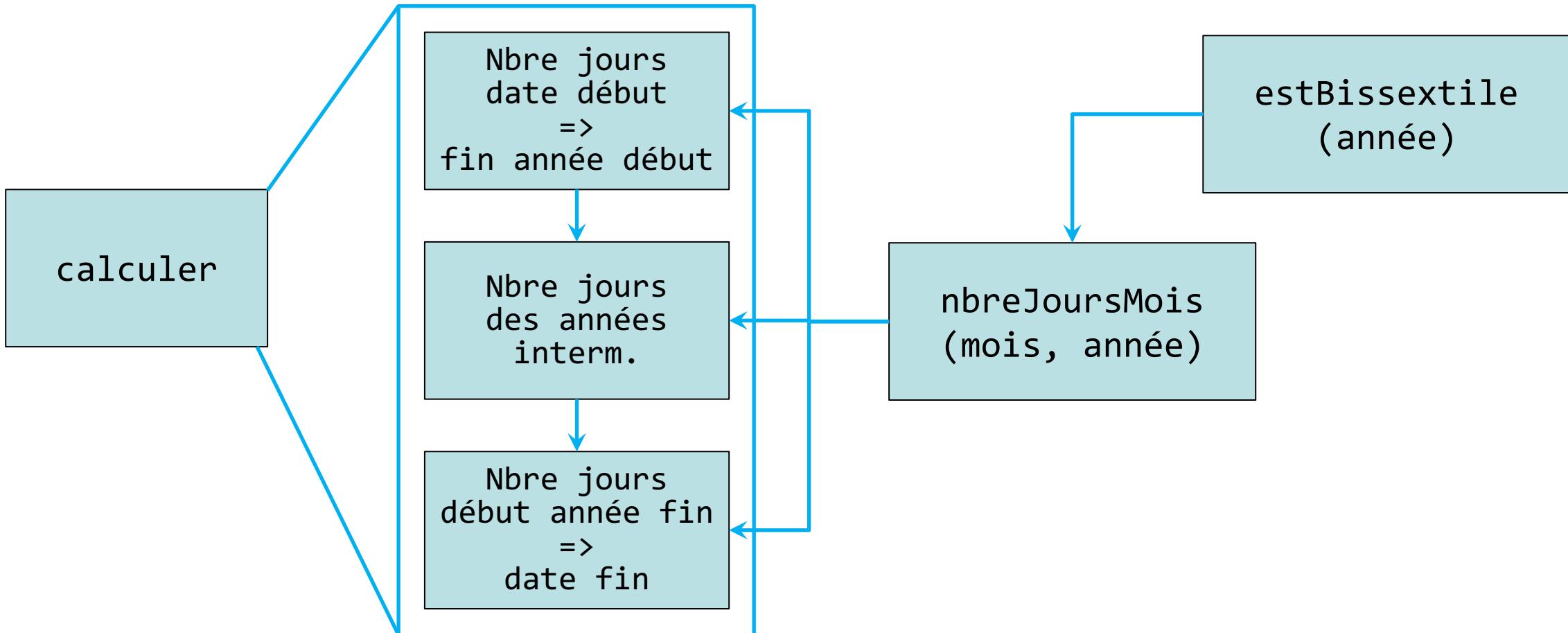
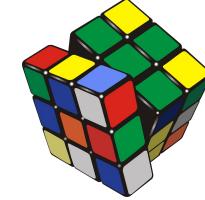
- Tous les mois n'ont pas le même nombre de jours
- Comment identifier une année bissextile
- Les saisies utilisateurs sont multiples
- Comment gérer les dates erronées à la saisie (30 février 2020)
- ...

Bien poser un algorithme et identifier les fonctions nécessaires est essentiel

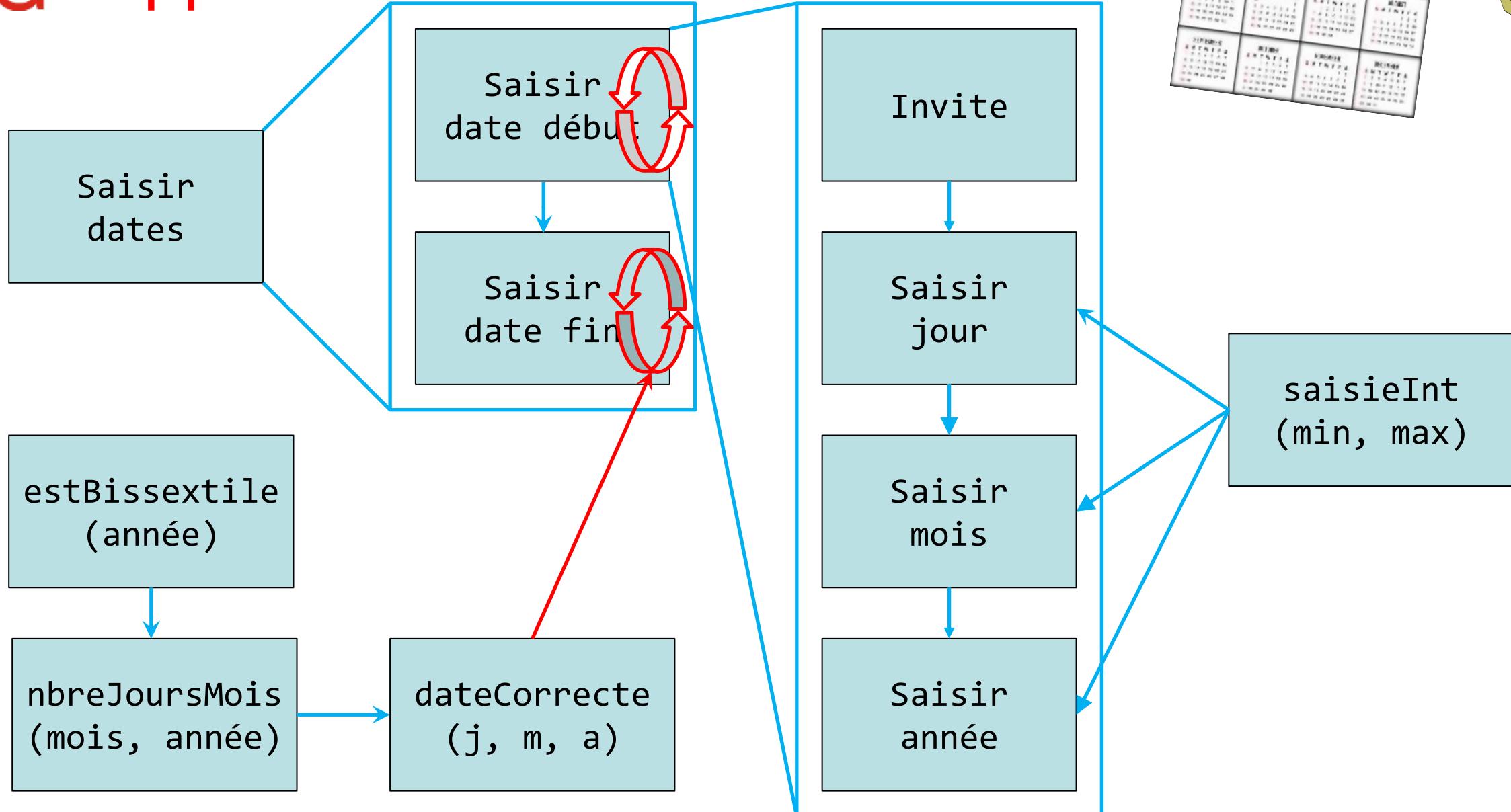
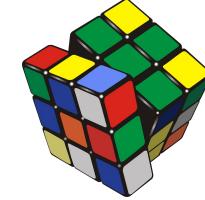
Approche descendante

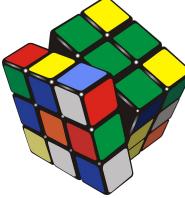


Approche descendante



Approche descendante

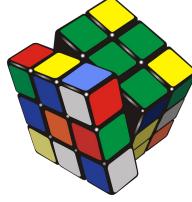




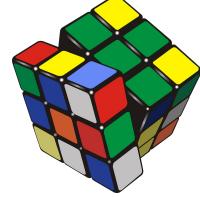
Approche descendante

Cette analyse nous a permis d'identifier quelques fonctions

- `saisirDate` retourne (ou rend par paramètres) une date
- `calculer` retourne le nombre de jours entre deux dates
- `afficher` afficher le nombre de jours dans un format choisi
- `saisieInt` saisi un entier dans un intervalle choisi
- `nbreJoursMois` retourne le nombre de jours pour un mois et une année
- `estBissextile` retourne un booléen indiquant si l'année est bissextile
- `dateCorrecte` retourne un booléen indiquant si la date est correcte
- ...



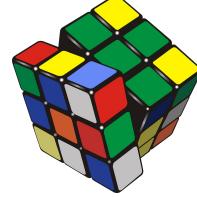
- L'approche descendante pose une question: jusqu'à quel niveau de détail faut-il décomposer les tâches ? **Quelle doit être la longueur de nos fonctions ?**
- Des fonctions **trop longues** sont difficiles à comprendre, voire à lire, si elles ne tiennent pas sur un écran.
- Des fonctions **trop courtes** augmentent sensiblement le travail additionnel de structuration, car chaque fonction doit être
 - conçue pour être réutilisable
 - déclarée
 - définie et codée
 - testée
 - commentée



<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

- [F.1](#) choisir un nom de fonction aussi pertinent que possible
- [F.2](#) une fonction doit faire une opération logique simple
- [F.3](#) une fonction doit être courte et simple
- [F.8](#) préférer des fonctions « pures » (résultat prévisible, pas d'effet de bord, pas d'affichage, ...)
- [F.10](#) si une opération peut être réutilisée, nommer là
- [F.16](#) passer les paramètres d'entrées de la manière la plus économique possible

```
void f1(const string& s);    // OK: pass by reference to const; always cheap
void f2(string s);          // bad: potentially expensive
void f3(int x);             // OK: Unbeatable
void f4(const int& x);      // bad: overhead on access in f4()
```



- F.17 passer les paramètres d'entrée-sortie par référence non-constante

```
void update(Record& r); // assume that update writes to r
```

- F.20 pour les valeurs de sortie, préférer un « return »
- F.60 préférer « T* » à « T& » quand « no argument » est une option possible
(T* peut valeur nullptr alors qu'il n'est pas possible d'avoir une référence à nulle)
- F.43 Ne jamais retourner un pointeur ou une référence à un objet local

```
int* f() {  
    int fx = 9;  
    return &fx;  
}
```

```
int& f() {  
    int fx = 9;  
    return fx;  
}
```

- F.56 éviter les conditions imbriquées inutiles