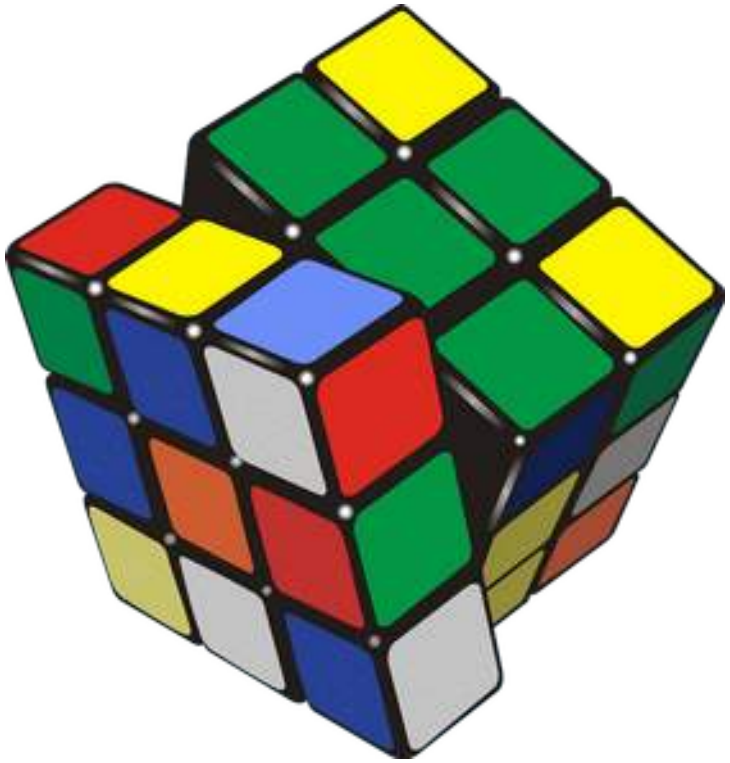


Chapitre 4
Fonctions

Buts du chapitre

- Apprendre à structurer le code d'un programme en sous-programmes effectuant des tâches séparées
- Comprendre le passage de paramètres par valeur, par référence ou par référence constante, et savoir choisir le bon mode de passage en fonction du problème
- Comprendre la notion de référence au-delà de son utilisation pour le passage de paramètres
- Savoir retourner une valeur ou une référence comme résultat d'un appel de fonction

Buts du chapitre

- Distinguer déclaration et définition d'une fonction, ce qui permet d'organiser le code en plusieurs fichiers compilés séparément
- Voir que l'on peut surcharger une fonction en utilisant le même nom avec une liste de paramètres différente
- Comprendre la visibilité et la durée de vie des variables selon qu'elles sont locales, globales ou statiques

Plan du chapitre 4

1. Introduction [5-14]
2. Les paramètres [15-30]
   a. Passage par valeur, par référence ou par référence constante [16-24]
   b. Types des paramètres lors des appels [25-30]
3. `return` [31-40]
4. Prototypes et compilation séparée [41-49]
5. Variables locales, globales et statiques [50-59]
6. Surcharge de fonctions [60-69]
7. `assert` [70-73]
8. Eléments pour une bonne conception du code [74-86]
9. Résumé [87-89]

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

Fonctions mathématiques vs fonctions 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

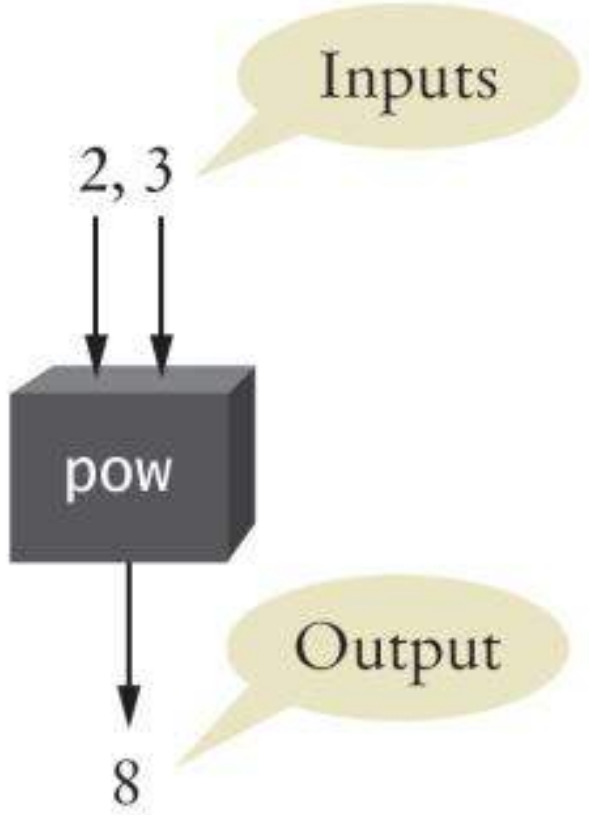
- Le code suivant affiche x^n et y^n pour x=2, y=½ et n=10, en appelant deux fois la fonction `pow`

```
#include <cmath>
#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 $\mathbf{base}^{\mathbf{exposant}}$



cplusplus.com – la fonction pow

function

pow

<cmath> <ctgmath>

C90 | C99 | C++98 | C++11

```
     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 power

Returns *base* raised to the power *exponent*:

$base^{exponent}$

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 Value

The 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 `val`=2^{10} est assez simple

valeurs d'entrée

résultat à calculer

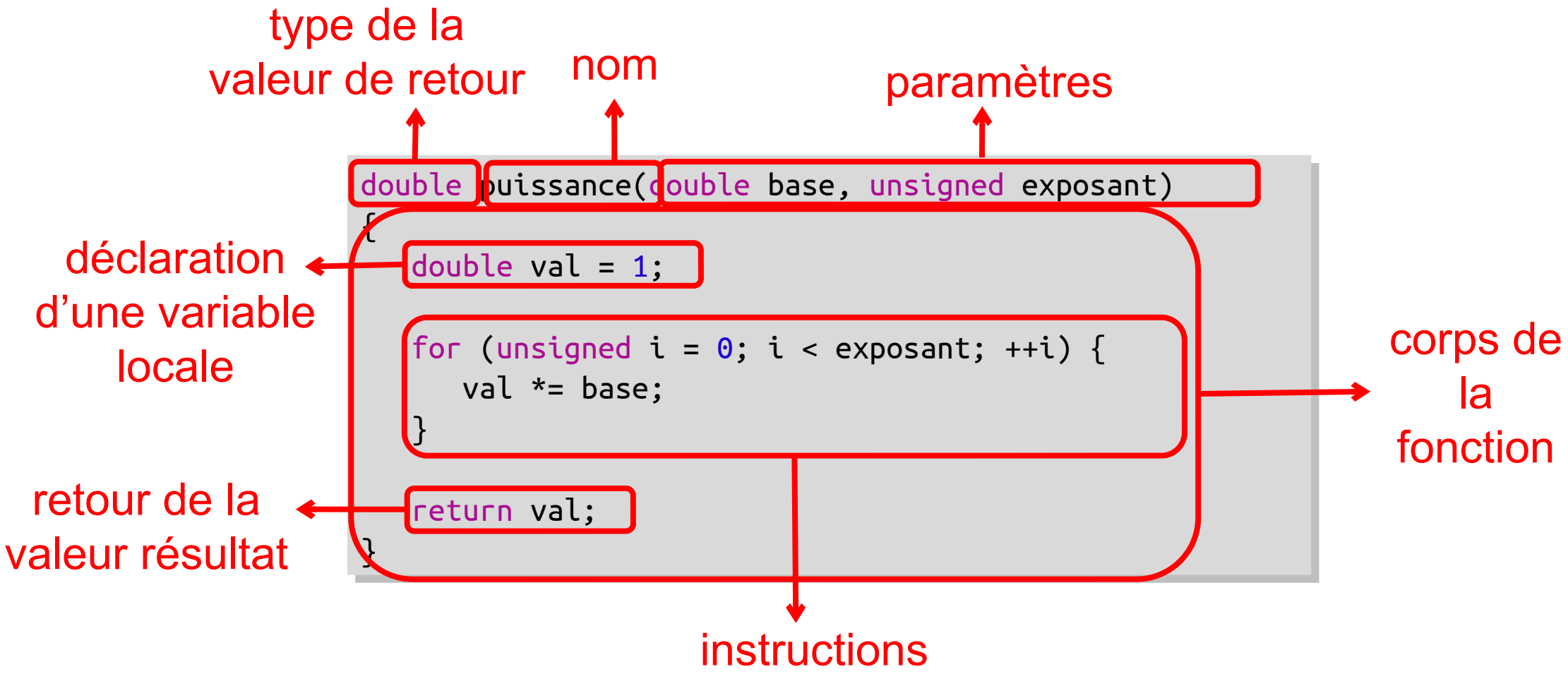
instructions effectuant le calcul

```
double base  = 2.0;
unsigned exposant = 10;

double val = 1;

for (unsigned 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



La fonction `main()`

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

```
int main() {
    …
    return 0;
}
```

- Elle ne prend pas de paramètres et retourne une valeur entière
  Elle garantit de retourner 0 si tout se passe bien, et une valeur non nulle sinon. Pour cette fonction (uniquement), la norme n'impose pas d'avoir explicitement une instruction `return`

- 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, unsigned width) {
   cout << "|" << setw(width)
        << val << "|" << endl;
}

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

```
|     1|
|     2|
```

2. Les paramètres

2a. Passage par valeur, par référence, ou par référence constante

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 référence constante : la fonction garantit de ne pas modifier la variable

Passage par valeur : exemple

```
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 : fonctionnement

- 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 !

Évaluation des paramètres passés par valeur

- 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 << endl;
}

int main() {
   int i = 2;
   f(i++, i++);
}
```

Sous gcc 8.1.0 :

- warning à la compilation : "operation on 'i' may be undefined"
- affiche " 3 2" comme résultat

Passage par référence : exemple

```
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;
}
```

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

- Qu'affiche ce code ?

Passage par référence : fonctionnement

- 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
- 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
- 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

Note : les références en dehors des fonctions

- Notons que le concept de références ne se limite pas au passage de paramètres de fonctions
  On peut écrire `int& b = a;` et **b** devient un synonyme de **a**

```
int main() {
   int  a = 5;
   int& b = a;
   cout << a << " " << b << endl;
   b = 3;
   cout << a << " " << b << endl;
}
```

```
5 5
3 3
```

- Attention, une référence doit toujours être initialisée.
  - Initialisation spécifie quelle est la variable synonyme
  - Affectation modifie le contenu de la variable synonyme

2b. 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);
}
```

```
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 variable du type spécifié

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

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

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

// Quels appels sont valides ?

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);  //
   afficher(vc, vi);  //
   afficher(CI, vc);  //  // Quels appels sont valides ?
   afficher(vi, CC);  //
   afficher(67, vc);  //
   afficher(vi, 'E'); //
}
```

```
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
  - Pour signaler à l'utilisateur de la fonction qu'elle ne modifiera pas le paramètre

✖
```
void afficher(string message) { … }
```
✖
```
void afficher(string& message) { … }
```
✔
```
void afficher(const string& message) { … }
```

3. return

Valeur de retour

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

```
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);
}
```

Valeur de retour

- 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` : `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` `return 0;` est utilisé implicitement.

Retour par référence

- 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 (PRG2)

```
int& plusPetit(int& a, int& b) {
    return a < b ? a : b;
}

int main() {
    int a = 5,
        b = 7;
    int c = plusPetit(a, b);

    plusPetit(a, b) = 10;

    cout << "a = " << a
         << ", b = " << b
         << ", c = " << c
         << endl;
}
```

```
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;
```

- Il est possible de la placer à gauche d'une affectation parce qu'elle retourne le type `char&`

Retour par référence – application

- Nous avons vu qu'il est possible d'enchaîner l'opérateur d'affectation = car il retourne la valeur affectée
- On peut donc écrire

```
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)));
```

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

- Qu'en est-il de `constexpr` (chapitre 2) appliqué aux retours de fonctions ?
- Pour initialiser une `constexpr` sur la base de la valeur de retour d'une fonction, celle-ci doit obligatoirement retourner une valeur constexpr et être évaluable à la compilation

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

int main() {
   int a = 1, b = 2;
   constexpr int C1 = f(a, b); // Erreur
   constexpr int C2 = f(1, 2); // Erreur
   constexpr int C3 = g(a, b); // Erreur
   constexpr int C4 = g(1, 2); // OK
}
```

Une fonction `constexpr` permet au compilateur d'évaluer, à la compilation déjà, la valeur retournée par la fonction et ainsi d'insérer cette valeur directement dans le code exécutable produit ☾ optimisation !

- En revanche, une fonction ne peut pas avoir de paramètres déclarés avec `constexpr`

Références, `constexpr` et *lvalue / rvalue*

- En C++, on définit les concepts de lvalue et de rvalue, où les lettres **l** et **r** signifient gauche (*left*) et droite (*right*) respectivement (d'une affectation)
- Une lvalue désigne tout ce qui peut apparaître à gauche d'une affectation
  - variable ou référence
- Une rvalue désigne tout ce qui ne peut pas apparaître à gauche, mais bien à droite d'une affectation
  - constante, constante littérale, expression, référence constante, etc.

4. Prototypes et compilation séparée

Prototypes

- 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

```
Error: Use of undeclared
identifier 'f'
```

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

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

- Une solution simple consiste à définir la fonction avant de l'utiliser

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

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

Prototypes

La solution propre consiste à séparer

- la déclaration de la fonction, son prototype, qui exprime l'interface qu'elle propose à ses appelants
- 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;
}
```

Prototypes

- Dans le prototype d'une fonction, les noms des paramètres sont facultatifs
  - Il peuvent être omis ou différer des noms utilisés dans la définition …
    mauvaise pratique !
  - sert 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
```

Compilation séparée

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 inclure 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 une et une seule fois
  - 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

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

- Pour ne déclarer un prototype qu'une seule fois, il faut
  - ne l'écrire que dans un seul fichier d'en-tête
  - n'inclure ce fichier d'en-tête qu'une seule fois
    - ❖ mais cela peut être compliqué si un projet comporte beaucoup de fichiers source, voire impossible s'il y a des références circulaires
- 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

maFonction.h

```
#ifndef MAFONCTION_H
#define MAFONCTION_H

int f(int val);

#endif
```

5. 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 ?

extern

- 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;
```

static

- 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)

static

- 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
```

static

- 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` (voir le cours d'ASD, et pour le langage C le cours de PRG2)

6. Surcharge de fonctions

Surcharge de fonctions

- Surcharger une fonction, c'est utiliser le même nom pour des fonctions que l'on distingue uniquement par leurs paramètres
- La signature (ou profil) d'une fonction correspond aux caractéristiques de ses paramètres
  - Leur nombre
  - Le type respectif de chacun d'eux
- Le compilateur choisira la fonction à utiliser selon les paramètres effectifs (de l'appel) par rapport aux paramètres formels (du prototype) des fonctions candidates

Surcharge de fonctions

- Les éléments suivants **ne permettent pas** de différencier deux surcharges
  - Le type de retour
  - Les valeurs par défaut des paramètres
  - La présence d'un `const` pour un paramètre passé par valeur
- Si deux fonctions de même nom ne diffèrent que par l'un ou plusieurs éléments ci-dessus, le compilateur signale une erreur à la déclaration de la deuxième fonction

Surcharge de fonctions

- Par contre, on peut surcharger une fonction
  en faisant varier le type de passage d'un paramètre
  - par valeur (constante ou pas, indistinctement)
  - par référence
  - par référence constante
- Mais cela peut poser problème lors de l'appel de ces fonctions
  Seule la paire référence/référence constante sera utile en pratique

Choix de la fonction surchargée

Correspondance exacte

```
void f(int);      // f1
void f(double);   // f2

int    i;
double d;

f(i);     // f
f(d);     // f
f(2.35); // f
```

Quelle fonction ?

Promotion numérique

```
void f(int);      // f1
void f(double);   // f2

char  c;
float y;

f(c);    // f
f(y);    // f
f('e'); // f
```

Quelle fonction ?

Choix de la fonction surchargée

Il est ici impossible de trancher, des conversions équivalentes menant à plusieurs prototypes

```
void f(int,    double); // f1
void f(double, int);    // f2

int    i, j;
double d;
char   c;

f(i, d);  // f1
f(c, d);  // f1 c -> int
f(d, i);  // f2
f(i, j);  // appel ambigu
```

Quelle fonction ?

Erreur de compilation

Pour l'appelant, les deux prototypes sont équivalents

```
void f(int    n = 0, double x = 0); // f1
void f(double x = 0, int    p = 0); // f2

int i; double d;

f(i, d);  // f1
f(d, i);  // f2
f(i);     // f1
f(d);     // f2
f();      // appel ambigu
```

Quelle fonction ?

Erreur de compilation

Choix de la fonction surchargée

- Par contre, on peut distinguer entre référence et référence constante

```
void f(int&);         // f1
void f(const int&);   // f2

int n = 3; const int p = 5; float x;

                                  // Quelle fonction est appelée ?
f(n);   // appelle f1
f(p);   // appelle f2
f(2);   // appelle f2, après copie éventuelle de 2
        // dans un entier temporaire dont la référence
        // est transmise à f.
f(x);   // appelle f2, après conversion de la valeur
        // de x en un entier temporaire dont la
        // référence est transmise à f
```

Choix de la fonction surchargée

- Il ne faut donc jamais surcharger passage par valeur et passage par référence

```
void f(int&);          // f1
void f(int);           // f2

int n = 3; const int P = 5; float x;

f(P);   // appelle f2
f(2);   // appelle f2                    // Quelle fonction est appelée ?
f(x);   // appelle f2
f(n);   // appel ambigu, ne compile pas.
        // même s'il est possible de compiler les
        // deux déclarations de f, aucun paramètre
        // effectif ne permet d'appeler f1 sans ambiguité
```

Règles de choix de la fonction surchargée

- Le compilateur recherche la meilleure correspondance, en testant dans l'ordre
  - Correspondance exacte - les types sont identiques
  - Correspondance avec conversion simple
    - variable transformée en constante (pas l'inverse)
    - tableau transformé en pointeur (ou l'inverse)
  - Correspondance avec promotion numérique
    - `bool`, `char` ou `short` => `int`
    - `float` => `double`
  - Correspondance avec conversion de type (ajustement de type ou conversion dégradante) toutes celles acceptées par l'opérateur d'affectation
- Le compilateur s'arrête au premier niveau de correspondance trouvé
- Il y a ambigüité si plusieurs prototypes correspondent à ce niveau

La «meilleure» conversion implicite ?

- Notons qu'en C++, contrairement à Java,
  la notion de « meilleure » conversion implicite n'existe pas

```
void f(short n) {cout << "Appel de f(short n)" << endl;}
void f(long  n) {cout << "Appel de f(long n)" << endl;}

int main() {
   int n = 1;
   f(n);       // Erreur à la compilation. Appel ambigu
}
```

- L'appel à `f(n)` est ambigu car les 2 fonctions sont potentiellement candidates
- La conversion dégradante `int -> short` est considérée au même niveau
  que l'ajustement de type `int -> long`

8. assert

assert

- La gestion des erreurs peut être traitée et rapportée par une valeur de retour en C, voire par une exception en C++ (chap. 9)
- Lorsque les conditions ne sont pas remplies pour un code et qu'il devient nécessaire de stopper son exécution, C et C++ mettent à disposition `assert`

```
assert(expression)
```

- La macro `assert` évalue `expression` ce paramètre étant traité comme un booléen (0 et différent de 0)
  - `false` écrit un message dans cerr et termine le programme avec un `abort` (chap. 9)
  - `true` ne fait rien

assert

```
#include <cassert>
```

```
char intToHexa(int value) {

   // arrêt si pas convertible
   assert(value >= 0 and value <= 15);

   if (value < 10)       // '0' - '9'
      return char('0' + value);
   else                  // 'a' - 'f'
      return char('a' + value - 10);
}
```

```
intToHexa(16);
```

```
Assertion failed:
(value >= 0 and value <= 15),
function intToHexa, file
/Users/main.cpp, line 10.
```

Quel caractère retourner en cas d'erreur ?

NB : Le message écrit dans `cerr` dépend de l'implémentation

assert

- Souvent utilisée en phase de développement, cette macro peut être désactivée si une macro nommée NDEBUG a été définie

```
#define NDEBUG
...
#include <cassert>
```

- `assert` est donc mise à disposition pour gérer des erreurs de programmation et non des erreurs d'exécution

9. Eléments pour une bonne conception du code

Éviter 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 __: ";
    cin >> ______ ;
} while (______ < 0 || ______  > __);


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

Éviter 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);
```

Éviter 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

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

Éviter la duplication de code

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

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

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

Approche descendante – raffinement

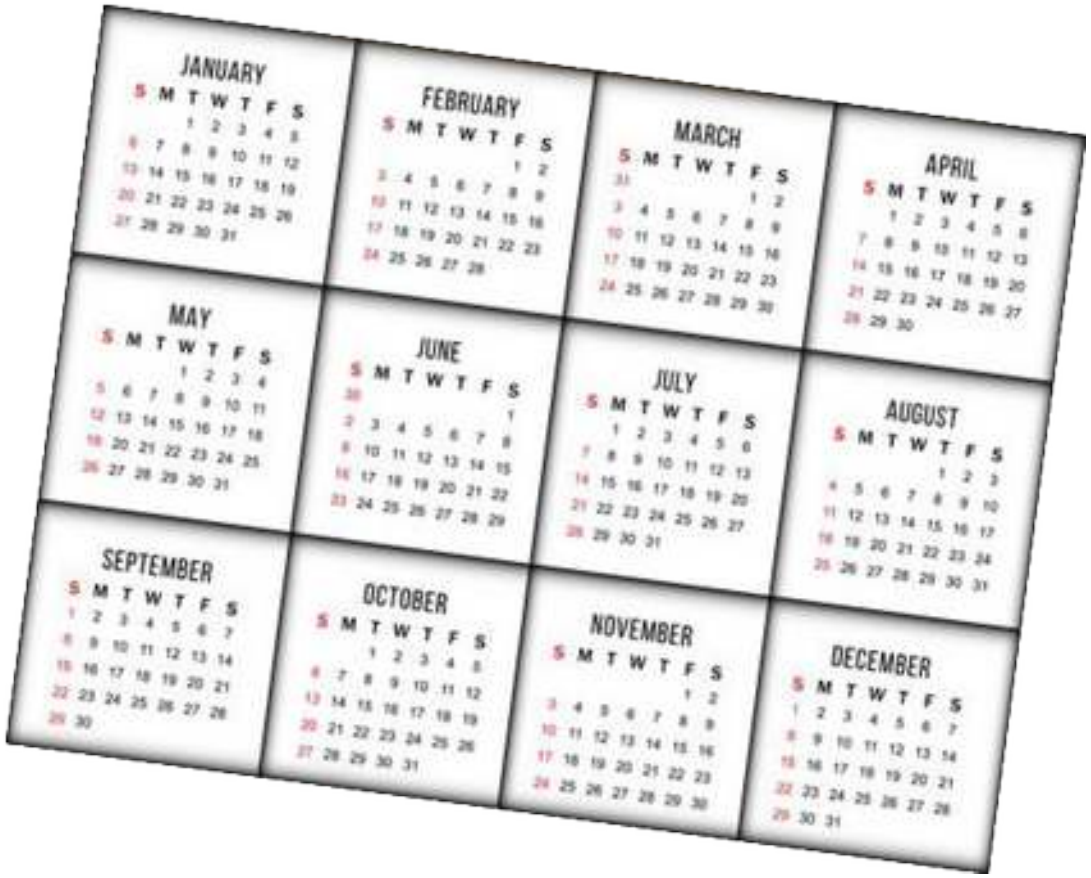
Décomposer un problème complexe
en tâches plus simples
(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)
(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)
(et pour les tâches où c'est nécessaire,
les décomposer en tâches plus simples)
(et pour 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
```



Approche descendante

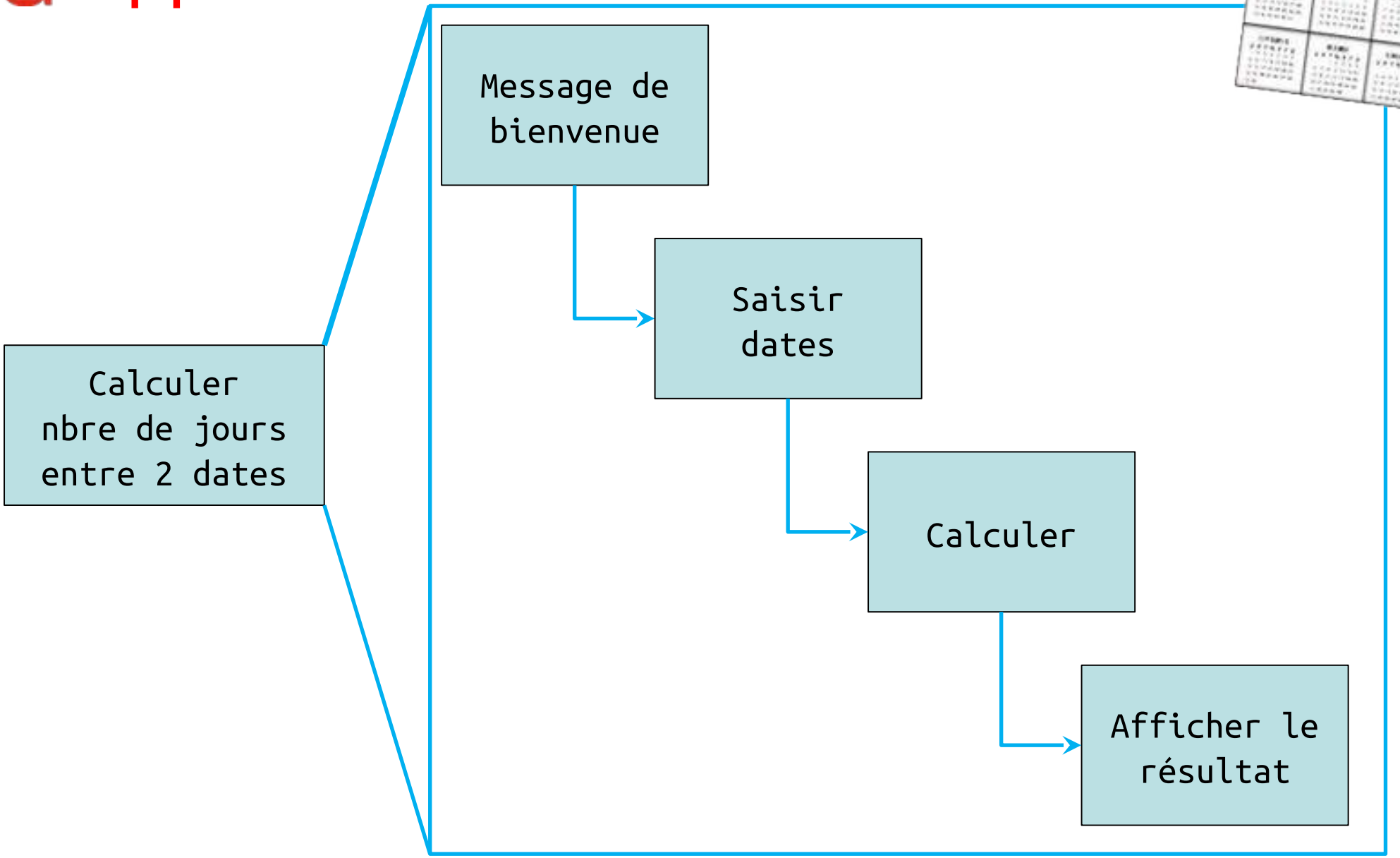
Combien de jours séparent deux dates ?

La question est simple … mais en y réfléchissant

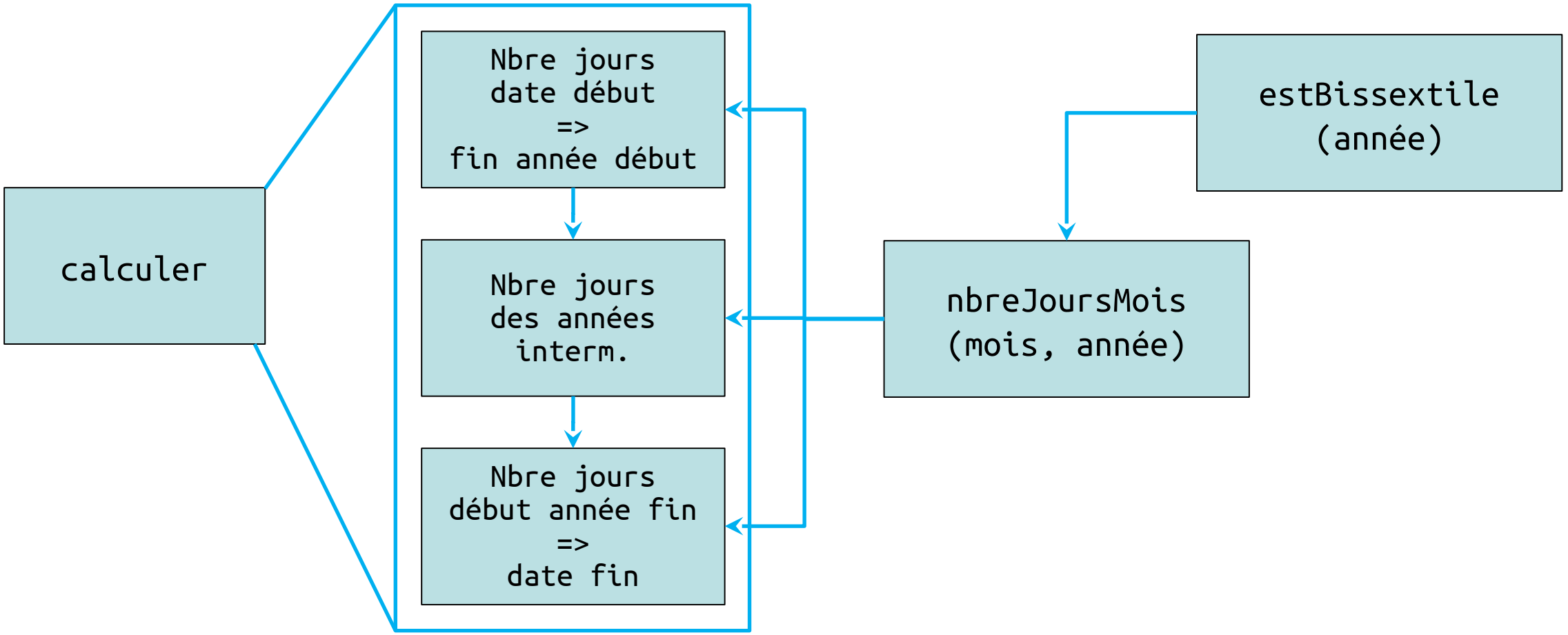
- Tous les mois n'ont pas le même nombre de jours
- Comment identifier une année bissextile
- Les saisies utilisateurs sont multiples
- Comment gérer les dates erronées à la saisie (30 février 2020)
- …

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
- …

Des fonctions de quelle longueur ?

- 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

10. Résumé

Résumé

- Les fonctions permettent de structurer le code.
- On peut leur passer en entrée des paramètres par valeur, par référence ou par référence constante.
- Elles peuvent retourner une valeur ou une référence. Il est aussi possible d'utiliser les paramètres passés par référence en sortie.
- Une fonction doit être déclarée (via un prototype) ou entièrement définie avant d'être utilisée.
- Séparer déclaration et définition des fonctions permet de répartir le code sur plusieurs fichiers compilés séparément.

Résumé

- Les variables peuvent être locales, globales ou statiques
  Cela affecte leur visibilité et leur durée de vie.

- Il est possible de surcharger des fonctions
  i.e. de réutiliser le même nom avec des paramètres différents.

- La conception de la structure d'un code est un art. La factorisation du code (suppression des duplications) et l'approche descendante peuvent nous y aider.