C++ avec du style
Patrons de bonnes pratiques et exemples d'erreurs récurrentes

Jérôme Dossogne

Année de parution 2009

Un grand merci à Roel Wuyts dont les enseignements furent en partie source de motivation à la conception de ce recueil.

Un grand merci aux personnes suivantes qui ont contribuées significativement à accroitre la qualité de ce document : Geeraerts Gilles, Markowitch Olivier, Martelange Dorian, Pluquet Frédéric, Roggeman Yves . . .

Table des matières

Liste des codes

1 Introduction

Les remarques qui vont suivre sont là pour vous orienter durant vos expériences de programmation. Il s'agit donc ici de recommandations, de lignes de conduites et conseils plus que d'affirmations dont on peut démontrer la véracité dans toutes les situations imaginables et possibles.

2 Remarques

2.1 Remarques générales

- Pensez à écrire vos commentaires et à tester votre code (sa compilation et son exécution) tout au long de votre travail.
- Nombres d'erreurs peuvent être signalées par le compilateur. Ainsi, l'usage des paramètres suivants fournis au compilateur est fortement recommandé :

```
g++ -Wall -pedantic -Weffc++ -Wextra -Wstrict-null-sentinel -std=c++0x -o
    monProgramme.exe main.cpp fichier01.cpp fichier02.cpp ...
```

- Si votre code ne compile pas, n'hésitez pas à vous faire une *checklist* pour vérifier vos accolades, les ;, ...
- Le compilateur est votre ami. La compilation de vos projets devrait se déroule sans affichage d'avertissement (« warning-free »).

2.2 Indentation/conventions d'écriture

- Lisibilité : Indentation : Consistance de la taille de l'indentation 001

Code 1 Lisibilité : Indentation : Consistance de la taille de l'indentation

```
//suggestion
if(a == 0)
    while(ilFaitBeauCeSoir)
        for(int i = 0; i < 12439; i++)
            cout << "youpiiiiiiiiiiiiie";
//plutôt que
if(a == 0)
            while(ilFaitBeauCeSoir)
              for(int i = 0; i < 12439; i++)
                                cout << "youpiiiiiiiiiiiiie";
```

Remarque(s) sur le code : **Code 1**

– la forme du code est une voie couramment utilisée afin de communiquer des informations au lecteur (i.e. vous même et vos collaborateurs) sur la structure de votre code, son sens, ... Être consistant, et donc faire usage des mêmes conventions partout dans votre code permet de transmettre cette information et donc d'accroitre la lisibilité de votre code, c'est-à-dire l'aisance qu'une personne aura à le lire. Veuillez donc appliquer la même indentation aux morceaux de code qui ont la même sutrcture.

- Lisibilité : Indentation : Alignement d'accolades 002

  Remarque(s) sur le code : **Code 2**

  – l'accolade ouvrante doit être alignée avec l'accolade fermante lors d'une indentation que nous appellerons ici « indentation classique » [1]. Il existe de nombreuses formes d'indentation. Dans la deuxième suggestion [2], nous gardons le fait que l'accolade fermante est alignée avec l'instruction qui a ouvert le bloc (c'est-à-dire le `if` ou le `else` dans notre exemple) tout en économisant une ligne dans notre code. Certaines personnes préfèrent donc cette présentation tout aussi acceptée pour la dite raison.

- Lisibilité : Indentation : Trompeuse 010

  Remarque(s) sur le code : **Code 3**

  – l'indentation est une façon très utile, à l'instar des commentaires, de communiquer des informations supplémentaires sur le sens du code qui est présenté au lecteur. Ainsi, on aligne le `if` et le `else` correspondant l'un avec l'autre. En aligner deux qui n'ont aucun lien est trompeur et il est de ce fait fortement déconseillé de rédiger un tel code.

1. Style « Allman »
2. Dont le style s'approche des conventions style « Kernighan and Ritchie », « 1TBS » ou encore style « BSD KNF »

Code 2 Lisibilité : Indentation : Alignement d'accolades

```
//suggestion
if(lInformatiqueCEstGenial)
{
    bool quEstCeQuIlFautPasDire = true;
    cout << "Je continue mes études (mais bon ... j'en pense pas moins ...)";
}
else
{
    bool quEstCeQuIlFautPasDire = false;
    cout << "Viva la revolucion";
}
//plutôt que
if(lInformatiqueCEstGenial){
    bool quEstCeQuIlFautPasDire = true;
    cout << "Je continue mes études (mais bon ... j'en pense pas moins ...)";
}
else{
    bool quEstCeQuIlFautPasDire = false;
    cout << "Viva la revolucion";
}
//suggestion
if(lInformatiqueCEstGenial){
    bool quEstCeQuIlFautPasDire = true;
    cout << "Je continue mes études (mais bon ... j'en pense pas moins ...)";
} else {
    bool quEstCeQuIlFautPasDire = false;
    cout << "Viva la revolucion";
}
//suggestion
if (lInformatiqueCEstGenial) {
    bool quEstCeQuIlFautPasDire = true;
    cout << "Je continue mes études (mais bon ... j'en pense pas moins ...)";
} else {
    bool quEstCeQuIlFautPasDire = false;
    cout << "Viva la revolucion";
}
//plutôt que
if(lInformatiqueCEstGenial)
    {
    bool quEstCeQuIlFautPasDire = true;
    cout << "Je continue mes études (mais bon ... j'en pense pas moins ...)";
}
else
{
    bool quEstCeQuIlFautPasDire = false;
    cout << "Viva la revolucion";
        }
```

Code 3 Lisibilité : Indentation : Trompeuse

```
//suggestion
if(b == c)
{
    if(c == a)
        ;
    else
        ;
}
//plutôt que
if(b == c)
{
    if(c == a)
        ;
else
    ;
}
//suggestion
switch(x)
{
    case 1:
        if(c == 2)
            cout << "cas numéro 1";
        cout << "plouf plouf";
        break;
    case 2:
        cout << "cas numéro 2";
        break;
}
//suggestion
switch(x)
{
case 1:
    if(c == 2)
        cout << "cas numéro 1";
    cout << "plouf plouf";
break;
case 2:
    cout << "cas numéro 2";
break;
}
//plutôt que
switch(x)
{
    case 1:
        if(c == 2)
            cout << "cas numéro 1";
    cout << "plouf plouf";
    break;
    case 2:
        cout << "cas numéro 2";
break;
}
```

  - il est d'usage de convenir que deux instructions alignées l'une sur l'autre appartienne au même bloc, dépendent des mêmes conditions, ...
  - la troisième suggestion rappelle au lecteur la sémantique du « switch » en indentant chaque « case » de « 0 » de même que le « default » puisque ceux-ci sont équivalents aux `if ... else if ... else`
- Fonctionnement : Compilation : Nombre impair d'accolades 003

Code 4 Fonctionnement : Compilation : Nombre impair d'accolades

```
//suggestion
if(monVoisinMeFaitCoucou)
{
    if(!ilFaitBeau)
    {
        ;
    }
}
//plutôt que
if(monVoisinMeFaitCoucou)
{
    if(!ilFaitBeau)
    {
        ;
}
```

Remarque(s) sur le code : **Code 4**

  - une erreur fréquente, lorsqu'on se fait ses premières armes aux côtés d'un compilateur, est d'oublier que toute accolade « { » a forcement sa réciproque « } » qui ferme le bloque qu'elle a ouvert. Si votre code présente un nombre impair d'accolades, c'est qu'il y a forcement une erreur quelque part. Ce genre d'erreur est bien sûr détecté à la compilation, attendez-vous à entendre g++ vous crier dessus.
- Lisibilité : Convention : Consistance sur la nomenclature, langue etc. 016

Code 5 Lisibilité : Convention : Consistance sur la nomenclature, langue etc.

```
//suggestion
int monEquipe, tonEquipe, compteTour, sauvegardeDuScore, maCuisine,
    monPantalonAMoiQueJAdoreTellementQuIlEstTropJoli;
void affichageMessageDeBienvenue();
//plutôt que
int monEquipe, yourTeam, compteTour, saveScore, mijnKeuken,
    monPantalonAMoiQueJAdoreTellementQuIlEstTropJoli;
void displayMessageDeWelkom();
//suggestion
int maMaison, monProgrammeQuiTueTout;
//plutôt que
int maMaison, mon_Programme_Qui_Tue_Tout;
//suggestion
int monAppart, maFenetre, maChambre;
//plutôt que
int MonAppart, mafenetre, m4cH4mBr3;
```

Remarque(s) sur le code : **Code 5**

  - éviter de mélanger les langues
  - rester consistant sur votre usage des séparateurs de mots
  - rester consistant sur votre usage de la casse (première lettre en majuscule ou non, ...)
- Lisibilité : Indentation : Pas d'espace superflus 032

Code 6 Lisibilité : Indentation : Pas d'espace superflus

```
//suggestion
while(x > 0) //il était une fois ... d'ailleurs il faut savoir que ... mais ...
{
    ;
}
//plutôt que
while(x > 0)//il était une fois ...
            //d'ailleurs il faut savoir que ...
            //mais pas avant le petit déjeuner ...
            //et ils vécurent heureux et eurent beaucoup de lignes de code
{
    ;
}
```

Remarque(s) sur le code : **Code 6**

- le code source est avant tout fait pour être travaillé sur ordinateur. Dans ce cas, autant dissocier le contenu de l'affichage. Si celui qui lit souhaite avoir un retour à la ligne automatique, il suffit de lui laisser faire le réglage en question dans son éditeur de texte. Ainsi, chaque lecteur peut disposer d'une présentation et d'un affichage qui lui convient. Par contre, si vous forcez le retour à la ligne en formatant le texte vous même, alors seuls ceux qui le souhaitaient seront heureux, tous les autres devront subir cette présentation qui éventuellement les dérange.
- dans de rare cas, l'éditeur qui vous est imposé (s'il y en a un) ne vous permet pas de personnaliser l'affichage ou l'impression de votre code. Il est évident que dans ces cas, ce qui prime est de pouvoir accomplir vos tâches.

- Lisibilité : Éviter les lignes vides inutiles 034

Code 7 Lisibilité : Éviter les lignes vides inutiles

```
//suggestion
if(joueurAGagne == true)
    cout << "Le joueur a gagné";
//plutôt que
if(joueurAGagne == true)



    cout << "Le joueur a gagné";
```

Remarque(s) sur le code : **Code 7**

- en général, en vue d'améliorer la lisibilité, on a plutôt tendance à vouloir limiter le nombre de ligne. Placer des lignes vides après la ligne d'un « if », d'un « for », d'un « while » ... n'apporte rien en clarté, au contraire. Par contre, il est bien sûr admis de placer des lignes vides améliorant la clarté (après vos déclarations de variables, entre deux boucles for, entre les grandes étapes d'une séquence ...)

2.3 Variables

- Fonctionnement : Variables : Problème d'initialisation 004
  Remarque(s) sur le code : **Code 8**
  - lors de la déclaration d'une variable, la valeur de la zone mémoire accessible via cette variable est inconnue. De ce fait, à moins qu'il s'agisse de l'objectif avoué par le programmeur, se baser sur la valeur de cette variable pour un test, une instruction, une opération, ... est généralement un *bug*. Il

Code 8 Fonctionnement : Variables : Problème d'initialisation

```
//suggestion
bool joueurHumainGagne = true;
while(joueurHumainGagne)
{
    ;
}
//plutôt que
bool joueurHumainGagne;
while(joueurHumainGagne)
{
    ;
}
```

est donc recommandé d'éviter et de détecter ces pratiques ou de penser à initialiser systématiquement ses variables et ceci dans la déclaration plutôt que dans le code qui suit, si pour autant le code ainsi résultant est correct bien sûr.

- Lisibilité : Variables : Nom de variable dénué de sens 005

Code 9 Lisibilité : Variables : Nom de variable dénué de sens

```
//suggestion
int choixMenu, scoreMonEquipe, scoreEquipeAdverse;
//plutôt que
int a, b,c;
//suggestion
int nombreDeVictoires, totalDesPoints;
//plutôt que
int victoires, points; //victoires représente le nombre total de victoires et
    points représente le total des points
```

Remarque(s) sur le code : **Code 9**

- ayez comme objectif la conception du code le plus facilement lisible, maintenable et *debuggable*. De nombreux algorithmes vous offriront, de par leur nature, une complexité et un défi cérébral suffisant. Il est donc inutile d'ajouter de la difficulté à votre code en retirant tout sens aux noms de vos variables. Accorder un nom porteur de sens aux variables permet en effet d'augmenter la lisibilité de votre code (ceci étant un des importants objectifs visés par l'usage de language de haut niveau), d'accroitre sa qualité.
- si le nom de votre variable est bien choisi, il n'a pas besoin d'être commenté.

- Maintenance-Lisibilité : Constantes : Définissez des constantes pour une meilleur lisibilité/maintenance du code 006

  Remarque(s) sur le code : **Code 10**
  - lorsque par convention, vous définissez qu'une constante représente à vos yeux une couleur, un type de choix (« le match se déroule à domicile », « le match se déroule à l'extérieur », « sortir du menu », ...), il peut être intéressant de définir des constantes permettant de remplacer dans votre code cette convention par le sens qui y est associé. Ainsi,
    - le « niveau » du code est plus élevé,
    - si un jour, vous décidez que votre convention change, vous pouvez changer celle-ci en ne modifiant votre code qu'à un endroit : là où votre convention est définie. De plus si tel n'était pas le cas, vous devriez le changer à chaque endroit en différenciant bien les endroits où, ici, le sens de la constante « 10 » représente la couleur « rouge » et non réellement la valeur « 10 » tel que c'est le cas dans l'instruction `totalDePoints = 10*nombreDeProjets;`. Ce qui fait qu'un remplacement automatisé via votre éditeur de texte peut être très risqué, ce dernier n'étant pas capable de faire la

Code 10 Maintenance-Lisibilité : Constantes : Définissez des constantes pour une meilleur lisibilité/maintenance du code

```
//suggestion
const int rouge = 10;
if(a==rouge)
    cout << "Attention, vous avez déjà un carton rouge";
c = rouge;
totalDePoints = 10*nombreDeProjets;
//plutôt que
if(a==10)
    cout << "Attention, vous avez déjà un carton rouge";
c = 10;
totalDePoints = 10*nombreDeProjets;
```

différence.
  – pensez à faire appel aux `enum` lorsque la situation s'y prête.
- Exécution-Lisibilité : Variables : Surplus de variables 025

Code 11 Exécution-Lisibilité : Variables : Surplus de variables

```
//suggestion
int a, n;
cin >> n;
cin >> a;
if(n == 0)
{
    cout << a + 1;
}
else
{
    cout << a + 2;
}
//plutôt que
int a, b, n;
cin >> n;
if(n == 0)
{
    cin >> a;
    cout << a + 1;
}
else
{
    cin >> b;
    cout << b + 2;
}
```

Remarque(s) sur le code : **Code 11**
  – bien que la variable `b` soit utilisée, elle est superflue. Dans le cas présent, en mettant en évidence l'instruction `cin >> a;`, on évite la démultiplication de variables.
- Exécution : Variable : Inutile/Inutilisée 027
  Remarque(s) sur le code : **Code 12**
  – dans notre exemple, la variable « b » n'est pas utilisée
  – dans notre exemple, la variable « c » est inutile

Code 12 Exécution : Variable : Inutile/Inutilisée

```
//suggestion
int main()
{
    int a;
    cin >> a;
    cout >> a + 1;
    return 0;
}
//plutôt que
int main()
{
    int a, b, c;
    cin >> a;
    c = 0;
    cout >> a + 1;
    return 0;
}
```

  - inutile de préciser qu'il est déconseillé de créer des variables qui ne sont pas utilisées ou totalement inutiles (sur lesquelles on opère sans que le résultat n'ait d'influence sur l'exécution du programme hormis le temps perdu sur l'opération)
  - l'utilisation des paramètres « -Wall -pedantic -Wextra » fournit au compilateur g++ permet d'améliorer la qualité de son code et d'éviter certains *bugs*. Entre autre, il vous avertira à la compilation si une variable déclarée est restée non utilisée.

- Efficacité : Exécution : Réutilisation/Calcul multiple d'une même valeur 035

Code 13 Efficacité : Exécution : Réutilisation/Calcul multiple d'une même valeur

```
//suggestion
solution[0] = fonctionLourde(0);
solution[1] = fonctionLourde(1);
for(int i = 1; i < n - 1 ; i++)
{
    solution[i+1] = fonctionLourde(i+1);
    if(solution[i-1] <= 2050 or solution[i+1] >= 6549876)
        solution[i] = -1;
}
//plutôt que
solution[0] = fonctionLourde(0);
for(int i = 1; i < n ; i++)
{
    if(fonctionLourde(i-1) > 2050 && fonctionLourde(i+1) < 6549876)
        solution[i] = fonctionLourde(i);
    else
        solution[i] = -1;
}
```

Remarque(s) sur le code : **Code 13**

  - dans la suggestion que nous vous proposons, pour chaque valeur de `i`, `fonctionLourde(i)` n'est appelée qu'une seule fois. Dans l'exemple à éviter, on recalcule plusieurs fois la même valeur. Certes, le deuxième code peut paraître, parfois, plus clair, plus lisible. Mais s'il s'agit d'une fonction lourde, mieux vaut éviter d'y faire appel inutilement. Il s'agit donc ici d'un choix, d'un équilibre entre la

sauvegarde d'une valeur et le temps de calcul. Par contre, dans le cas où les valeurs sont de toute façon sauvegardées, comme c'est le cas dans notre exemple, alors inutile de recalculer une valeur que nous connaissons déjà.

2.4 Commentaires

- Lisibilité : Commentaires : Manque de commentaires 017
  Remarque(s) :
  – un code parfaitement rédigé, avec des noms de variables, de fonctions... bien choisi aura moins de commentaires car devrait moins compenser. Il n'en sera pas dénué pour autant.
- Lisibilité : Commentaires : Emplacement mal choisi 019

Code 14 Lisibilité : Commentaires : Emplacement mal choisi

```
//suggestion
int x; // x est la variable servant à l'entrée pour les choix de l'utilisateur
    dans notre menu
cin >> x;
while(x != 0)
{
    if(x == 1);
    else if (x == 2);
}
//plutôt que
int x;
cin >> x;
while(x != 0) // x est la variable servant à l'entrée pour les choix de l'
    utilisateur dans notre menu
{
    if(x == 1); // x est la variable servant à l'entrée pour les choix de l'
    utilisateur dans notre menu
    else if (x == 2); // x est la variable servant à l'entrée pour les choix de
    l'utilisateur dans notre menu
}
```

Remarque(s) sur le code : **Code 14**
– bien placer ses commentaires peut éviter de les dupliquer car on sait où retrouver l'information si on a besoin d'un rappel sur le sens d'une instruction
– si le nom de la variable avait été plus judicieusement choisi, le commentaire aurait été inutile. D'où l'importance de bien choisir les noms de ses variables
– il est courant de mettre les commentaires avant le bloc qui les concerne. Cela permet d'éviter au lecteur de de passer du temps à comprendre le bloc commenté pour, une fois l'avoir compris, se rendre compte qu'il y avait une explication le concernant
– toute ligne, instruction, commentaire dupliqué est une ligne qu'il faut mettre à jour plusieurs fois et donc à des endroits différents lors de la maintenance de notre code. La duplication augmente le risque d'avoir des commentaires qui ne sont plus à jour[3]. Il est donc impératif d'être très très prudent dès que l'on choisit de faire un « copier/coller »

- Lisibilité : Commentaires : Superflus 021
  Remarque(s) sur le code : **Code 15**
  – les commentaires ont pour objectif d'ajouter de la valeur au code, d'accroitre sa lisibilité, d'aider à la compréhension, ...
- Lisibilité : Commentaires : Lié 038
  Remarque(s) sur le code : **Code 16**

3. suite à un oubli par exemple

Code 15 Lisibilité : Commentaires : Superflus

```
//suggestion
if(x == 0)
//plutôt que
if(x == 0) // si x vaut zero
//suggestion
while(x == 0)
//plutôt que
while(x == 0) // tant que x vaut zero, on repete le corps de la boucle
```

Code 16 Lisibilité : Commentaires : Lié

```
//suggestion
//suggestion dépendante des circonstances

//plutôt que
//on déclare des variables
int x, y;
//pour blablabla
cin >> x;
while(x > 0 && y < -18 && x < 18*42) //tant qu'on fait du profit
{
    //pendant que blablabla
}
//car nous savons que ...
```

  - lorsque vous rédigez vos commentaires, réfléchissez aux besoins et attentes du lecteur. Pour être clairs et utilisables, il peut être intéressant de s'assurer qu'ils puissent être lus indépendemment les uns des autres. Les commentaires ne sont pas forcement tous lus. Tant que le code est clair, le lecteur s'épargne souvent la lecture de ces derniers. Si l'on observe le deuxième code, on constate un inconvénient. Comme l'explication de l'algorithme n'est pas unifiée, on peut parfois se retrouver face à des commentaires qui ne sont que des bouts de phrases. Ainsi, si comme nous venons de le supposer, le lecteur a sauté les commentaires du début, ce dernier risque de ne pas comprendre le sens du commentaire lié à un groupe d'instructions puisque ce sens est lié aux commentaires précédents.

- Lisibilité : Commentaires : Présentation de fonction erronée 039
  Remarque(s) sur le code : **Code 17**
  - cette fonction affiche le contenu d'un vecteur passé en paramètre. Contrairement à ce qui est dit dans le commentaire du deuxième code, elle n'affiche pas systématiquement les nombres de Catalan. Ce n'est que suite au contenu de notre fonction « main » que de telles valeurs se trouvent dans le vecteur. Ainsi, en glissant un tel commentaire avant notre fonction, on trompe le lecteur sur l'objectif de la fonction puisque celle-ci peut être utilisée dans d'autres situations que ce cas particulier, d'autres programmes, etc.
  - de plus, dans le cas présent, le nom de la fonction documente suffisamment sur son comportement. Il semble donc, à première vue, qu'il soit inutile ici de documenter plus le but de la fonction.
- Exécution : Commentaires : Supposition d'un comportement en dehors de la fonction 040
  Remarque(s) sur le code : **Code 18**
  - veuillez comparer les commentaires décrivant les fonctions dans le deuxième code. On constate qu'un gros défaut apparait. Pour la fonction « void catalan(int solution[], int taille) », il est supposé que le vecteur « solution » sera fourni en paramètre en étant initialisé à zéro. Pourtant rien n'indique à l'utilisateur de la fonction Catalan qu'il est supposé faire cela dans sa fonction appelante [4]. Ainsi, si dans l'énoncé il vous est demandé une fonction effectuant l'opération « Placer les taille premiers nombres de Catalan dans le vecteur solution », rien ne vous permet de croire que les utilisateurs de

4. Indication que nous pourrions fournir, par exemple, via le commentaire décrivant le comportement de la fonction

Code 17 Lisibilité : Commentaires : Présentation de fonction erronée

```
//suggestion
int main()
{
    /*du code*/
    catalan(solution, taille);
    affichageVecteurDEntier(solution, taille);
    return 0;
}
void affichageVecteurDEntier(int vecteurAAfficher[], int taille)
{
    /*du code*/
}
//plutôt que
int main()
{
    /*du code*/
    catalan(solution, taille);
    affichageVecteurDEntier(solution, taille);
    return 0;
}
//Fonction qui affiche les nombres de Catalan du vecteur solution
void affichageVecteurDEntier(int vecteurAAfficher[], int taille)
{
    /*du code*/
}
```

votre fonction initialiseront le vecteur passé en paramètre et ce, encore moins si vous ne documentez pas votre fonction en indiquant que tel doit être le cas !
- l'erreur dans le second commentaire est encore plus flagrante. La fonction sert à afficher les éléments d'un vecteur. Si on fait appel à cette fonction sur un vecteur contenant les nombres de Catalan, alors oui, elle affichera les nombres de Catalan, mais il s'agit d'un cas particulier sur lequel le code de cette fonction n'a aucune prise.

2.5 Fonctions/Instructions

- Exécution : Instruction : Instruction inutile 009
  Remarque(s) sur le code : **Code 19**
  - si une instruction n'a aucun effet, supprimez la. Vous diminuez ainsi le nombre de lignes, et donc aussi le nombre de lignes à maintenir, à *débugger*, ...
- Exécution - Lisibilité - Maintenance : Instruction : Code commun/dupliqué 011
  Remarque(s) sur le code : **Code 20**
  - à l'instant où vous effectuez un « copier/coller » ... arrêtez-vous. Il s'agit souvent d'une mauvaise pratique. De façon non exhaustive, on peut dire que :
    - soit il est vraiment requis qu'une opération s'effectue deux fois, mais avec des paramètres différents. Et alors, dans ce cas, il est conseillé de rédiger une fonction à laquelle vous transmettrez ces paramètres/valeurs,
    - soit, comme c'est le cas ici, nous sommes en face d'un code qui ne dépend pas vraiment des conditions testées dans le `if` puisque quel que soit le résultat du test, l'instruction est exécutée. Dans ce cas, on peut le sortir du branchement.
- Lisibilité : Instructions : Consistance dans la logique du code 018
  Remarque(s) sur le code : **Code 21**

Code 18 Exécution : Commentaires : Supposition d'un comportement en dehors de la fonction

```
//suggestion
int main()
{
    /*du code*/
    int solution[taille] = {0};
    catalan(solution, taille);
    return 0;
}
//Place les taille premiers nombres de Catalan dans le vecteur solution,
    initialisé à zéro pour chacune de ses composantes
void catalan(int solution[], int taille)
{
    solution[0] = 1;
    for(int n = 1; n < taille; n++)
        for(int j = 0; j <= n - 1; j++)
            solution[n] = solution[n] + solution[j] * solution[n - 1 - j];
}

void affichageVecteurDEntier(int solution[], int taille)
{
    for(int j = 0; j < n ; j++)
        cout << solution[j];
}
//plutôt que
int main()
{
    /*du code*/
    int solution[taille] = {0};
    catalan(solution, taille);
    return 0;
}
//Place les taille premiers nombres de Catalan dans le vecteur solution
void catalan(int solution[], int taille)
{
    solution[0] = 1;
    for(int n = 1; n < taille; n++)
        for(int j = 0; j <= n - 1; j++)
            solution[n] = solution[n] + solution[j] * solution[n - 1 - j];
}
//Affiche les nombres de Catalan
void affichageVecteurDEntier(int solution[], int taille)
{
    for(int j = 0; j < n ; j++)
        cout << solution[j];
}
```

Code 19 Exécution : Instruction : Instruction inutile

```
//suggestion
//rien du tout
//plutôt que
if(b == c)
    a +=  b - c;
```

Code 20 Exécution - Lisibilité - Maintenance : Instruction : Code commun/dupliqué

```
//suggestion
if(leMatchEstJoueADomicile)
    cin >> b >> c;
else
    cin >> c >> b;
a = b + c;
//plutôt que
if(leMatchEstJoueADomicile)
{
    cin >> b >> c;
    a = b + c;
}
else
{
    cin >> c >> b;
    a = b + c;
}
```

Code 21 Lisibilité : Instructions : Consistance dans la logique du code

```
//suggestion
a = a + b;
c = c + b;
cout << a << b << c;
//plutôt que
a = a + b;
cout << a << b << c = c + b;
```

– les étapes de calculs sont plus évidentes.

- Exécution : Fonction : Type de retour 020

Code 22 Exécution : Fonction : Type de retour

```
//suggestion
int main()
{
    //votre code
    return 0;
}
//plutôt que
int main()
{
    //votre code
}
//suggestion
int maFonctionAMoiQuiRenvoitCinq()
{
    /*du code*/
    return 5;
}
//plutôt que
int maFonctionAMoiQuiRenvoitCinq()
{
    /*du code*/
}
//suggestion
bool faitIlBeauTousLesJoursEnBelgique()
{
    /*du code*/
    return false;
}
//plutôt que
bool faitIlBeauTousLesJoursEnBelgique()
{
    /*du code*/
}
```

Remarque(s) sur le code : **Code 22**

– le `int` de `int main()` est l'indication que la fonction dont le nom est `main` renvoie une valeur avec comme type de retour un entier. Par conséquent, il est logique que celle-ci en renvoie un.
– vérifiez que vous renvoyez une valeur si le type de votre fonction est différent de `void`
– remarquez que le compilateur peut vous afficher des « warnings » dans ce cas.

- Lisibilité : Instruction : Un seul `return` et bien placé par fonction 031

Remarque(s) sur le code : **Code 23**

– le return, dans la plupart des cas, sera votre dernière instruction avant de fermer la fonction. Cela accroit la lisibilité en fixant le point de sortie de votre programme. En plaçant des return un peu partout, non seulement vous la baissez mais comme on le voit sur l'exemple ci-dessus, la condition `(x == 0)` est testée plusieurs fois par boucle.

- Lisibilité : Maintenance : *Debug* : Fonctions 033

Remarque(s) sur le code : **Code 24**

– si un bout de code représentant une fonction, a un sens bien défini au point de mériter un commentaire disant « ici, je fais ça sur les variables x, y ,z » alors … pourquoi ne pas en faire directement une

Code 23 Lisibilité : Instruction : Un seul `return` et bien placé par fonction

```
//suggestion
int main()
{
    int a,b;
    if(a > 0)
        b = 5;
    else
        b = 6;
    return b;
}
//plutôt que
int main()
{
    int a;
    if(a > 0)
        return 5;
    else
        return 6;
}
//suggestion
int main()
{
    int x;
    cin >> x;
    while(x != 0)
    {
        if(x == 1)
            cout << "A l'aventure compagnon ..." << endl;
        cin >> x;
    }
    return 0;
}
//plutôt que
int main()
{
    int x;
    cin >> x;
    while(x != 0)
    {
        if(x == 1)
            cout << "A l'aventure compagnon ..." << endl;
        cin >> x;
        if(x == 0)
            return 0;
    }
}
```

Code 24 Lisibilité : Maintenance : *Debug* : Fonctions

```
//suggestion
effectueLOperationXSurLaVariableY(Y);
//plutôt que
//code de l'opération X sur la variable Y
```

fonction ? Ainsi, si le nom de votre fonction est bien choisi :
- lisibilité : vous y gagnez en clarté, plus besoin d'écrire un commentaire et donc de le lire
- maintenance : plus besoin de mettre à jour, de maintenir vos commentaires
- *débuggage* : si votre code est, sur le moment ou par la suite, appelé à plusieurs endroits, votre code n'est plus dupliqué et ne doit être *débuggé* qu'à un seul endroit.

- Exécution : Performance : Instruction (parfois) inutile 037

Code 25 Exécution : Performance : Instruction (parfois) inutile

```
//suggestion
int signe;
cin >> signe;
if(signe == 1)
{
    maFonction();
    signe = 0;
}
else if(signe == 0)
{
    monAutreFonction();
}
//plutôt que
int signe;
cin >> signe;
if(signe == 1)
{
    maFonction();
}
else if(signe == 0)
{
    monAutreFonction();
}
signe = 0;
```

Remarque(s) sur le code : **Code 25**
- si, comme c'est le cas dans notre exemple, les fonctions utilisées ne modifient pas la valeur de la variable `signe`, alors, dans le deuxième code, on effectue l'assignation sur la variable `signe` dans tous les cas, même si celle-ci valait déjà `0`.

- Lisibilité : Instruction : Interdit (`friend`) 041
  Remarque(s) :
  - pour certains cours, comme dans de nombreuses situations, l'usage du mot-clef `friend` est à proscrire. Faites donc bien attention à respecter les consignes du cours et les conséquences de l'usage de ce mot-clef.

2.6 Affichages

- Exécution : Affichage : Usage de `endl`, ... 007
  Remarque(s) sur le code : **Code 26**
  - vous constaterez rapidement, lors de vos projets, qu'un affichage « lisible » aide au *debuggage*. Ainsi, n'hésitez pas à faire usage de `endl`,`" "`, `setw()`, `setfill()`, ... dans vos `cout`.
- Consignes : Présentation : Feuilles volantes 024
  Remarque(s) :
  - évitez les feuilles volantes pour tout document transmis (projets, examens, etc.). Agrafez les, placez les dans une chemise plastifiée et n'hésitez pas à numéroter chacune de vos feuilles ainsi qu'à indiquer

Code 26 Exécution : Affichage : Usage de `endl`, ...

```
//suggestion
const int nombreDeProjets = 6;
int mesPointsDeProjetsINFOF101[] = {10,8,7,10,2,10};
for(int i = 0; i < nombreDeProjets; i++)
    cout << "Projet numero " << i << " : " << mesPointsDeProjetsINFOF101[i] <<
    endl;
//plutôt que
const int nombreDeProjets = 6;
int mesPointsDeProjetsINFOF101[] = {10,8,7,10,2,10};
for(int i = 0; i < nombreDeProjets; i++)
    cout << "Projet numero " << i << " : " << mesPointsDeProjetsINFOF101[i];
```

le nombre total de feuilles remises

2.7 Branchement conditionnels et boucles

- Lisibilité : Instruction : Interdit (`break`, `goto`, `continue`) 012
  Remarque(s) sur le code : **Code 27**
  - il est fortement déconseillé d'avoir un `break` dans son code. La plupart du temps, il est possible de rédiger son code plus proprement sans faire usage de cette instruction. Dans l'exemple fourni, on voit mieux à quelle condition on sort de la boucle.
  - pour les mêmes raisons que le `break`, il est recommandé d'éviter les `continue`.
- Lisibilité : Instruction : Interdit (`while(true)`, `while(1)`, `while(1*(18%2)+3-(29%3))`) 026
  Remarque(s) sur le code : **Code 28**
  - les `while(true)` sont à éviter [5].
- Lisibilité : Instruction : `if(x)`, `while(x)` 028
  Remarque(s) sur le code : **Code 29**
  - bien que ces exemples soient corrects, il vaut mieux écrire la condition explicitement. Notez que dans certains cas, certains programmeurs apprécient de faire la distinction entre les tests numériques et la manipulation de pointeur. Ainsi, la précédente remarque ne serait, dans ce contexte, pas d'application dans les cas tels que `while(p) p = p->svt;` avec `p`, un pointeur.
- Exécution : Lisibilité : Test inutile 036
  Remarque(s) sur le code : **Code 30**
  - le troisième `if` ne sert à rien, si on est dans le deuxième `else` c'est que `a` est strictement positif.
- Lisibilité : Instruction : Usage du `for` 013
  Remarque(s) sur le code : **Code 31**
  - l'usage d'une boucle `for` non paramétrée indique clairement un mauvais usage de la boucle.
- Exécution : Instruction : `if;else if;` / `switch` plutôt que `if` / `if` 014
  Remarque(s) sur le code : **Code 32**
  - dans les premiers cas nous avons au plus trois tests, dans le dernier nous avons exactement trois tests quelque soit la valeur de `c`. Face à un tel code, la sémantique n'étant pas la même, il importe de se demander quel est l'effet recherché.
- Exécution : Instruction : Test multiple de la même condition 015
  Remarque(s) sur le code : **Code 33**
  - on voit directement que la même condition `(choixMenu == aDomicile)` est testée à de multiples endroits sans pour autant que la variable `choixMenu` soit modifiée entre temps. Cela devrait nous mettre la puce à l'oreille et nous pousser à nous assurer que notre code est bien rédigé de façon optimale. Ce n'est pas toujours évident et peut dépendre de l'intention du programmeur.
- Exécution : Instruction : Confusion entre = et == 022

5. les boucles dites « n +1/2 », vues dans certains cours d'algorithmique, sont considérées par certains comme faisant exception à la règle

Code 27 Lisibilité : Instruction : Interdit (`break`, `goto`, `continue`)

```
//suggestion
while(a > 10 && a < 20)
    cout << "ok";
//plutôt que
while(a > 10)
{
    if (a > 20)
        break;
    else
        cout << "ok";
}
//suggestion
do
{
    cout << "Veuillez entrer une valeur";
    cin >> b;
    if (b > 0)
    {
        a = a + b;
        cout << "ok";
    }
}
while(a > 10);
//plutôt que
do
{
    cout << "Veuillez entrer une valeur";
    cin >> b;
    if (b <= 0)
        continue;
    else
    {
        a = a + b;
        cout << "ok";
    }
}
while(a > 10);
```

Code 28 Lisibilité : Instruction : Interdit (`while(true)`, `while(1)`, `while(1*(18%2)+3-(29%3))`)

```
//suggestion
do
{
    cin >> j;
    do
    {
        cin >> i;
    }
    while(i != 0)
}
while(j != 0);
//plutôt que
while(1)
{
    cin >> j;
    while(true)
    {
        cin >> i;
        if(i == 0)
            break;
        if(i == 42)
            continue;
    }
    if(j == 0)
        break;
}
```

Code 29 Lisibilité : Instruction : `if(x)`, `while(x)`

```
//suggestion
if(x != 0);
//plutôt que
if(x);
//suggestion
while(x != 0);
//plutôt que
while(x);
```

Code 30 Exécution : Lisibilité : Test inutile

```
//suggestion
if(a < 0)
    ;
else if (a == 0)
    ;
else
    ;
//plutôt que
if(a < 0)
    ;
else if (a == 0)
    ;
else if (a > 0)
    ;
```

Code 31 Lisibilité : Instruction : Usage du `for`

```
//suggestion
for(int i = 0; i < 10; i++)
    ;
//plutôt que
int i;
i = 0;
while(i<10)
    i = i + 1;
//et proscrire
int i = 0;
for(;;)
{
    if(i >= 10)
        break;
    i++;
}
```

Code 32 Exécution : Instruction : `if;else if;` / `switch` plutôt que `if` / `if`

```
//suggestion
int c;
cin >> c;
if(c == 0)
    cout << "haha";
else if(c==1)
    cout << "hihi";
else
    cout << "hoho";
//suggestion
int c;
cin >> c;
switch (c)
{
    case 0:
        cout << "haha";
        break;
    case 1:
        cout << "hihi";
        break;
    default:
        cout << "hoho";
}
//plutôt que
int c;
cin >> c;
if(c == 0)
    cout << "haha";
if(c == 1)
    cout << "hihi";
if(c != 0 && c != 1)
    cout << "hoho";
```

Code 33 Exécution : Instruction : Test multiple de la même condition

```
//suggestion
//suggestion dépendante du code exécuté
//plutôt que
if(choixMenu == aDomicile);
else if (choixMenu == aLExterieur);

if(scoreNotreEquipe < 0 || choixMenu == aDomicile)
    cout << "Erreur";
else if (choixMenu == aDomicile);
```

Code 34 Exécution : Instruction : Confusion entre = et ==

```
//suggestion
if(a == b);
//plutôt que
if(a = b);
//ou alors, si vraiment c'était ce qui était voulu
//suggestion
bool a, b;
// ... du code
a = b;
if(a == true);
```

Remarque(s) sur le code : **Code 34**

– les exemples fournis ont une sémantique différente. En général, un `if(a = b)` n'est pas ce que l'on pensait faire, il y a juste eu confusion entre l'opérateur d'assignation et l'opérateur de comparaison. `if(a = b)` assigne la valeur de `b` à `a` et effectue le test du `if` sur la valeur renvoyée par l'opérateur d'assignation alors que `if(a == b)` fait appel à l'opérateur de comparaison entre `a` et `b`. Il importe de se demander ce que l'on souhaite réellement.

- *Debug* : Tests : Faire des tests [023]

  Remarque(s) :

  – ne pas lésiner sur les tests que vous appliquez à votre programme. Testez bien les cas particuliers prévus ou sous-entendus par l'énoncé/la problématique à laquelle vous êtes confrontés et ce dès que possible.