Guide de programmation BTS SN

Sommaire

Thierry Vaira - <tvaira@free.fr> - version v0.3 - 23/08/2021 - btssn-lasalle84.github.io

1. Le métier de programmeur

D'après Bjarne Stroustrup, le métier de programmeur consiste à écrire des programmes qui :

- donnent des résultats corrects
- sont simples
- sont efficaces

Bjarne Stroustrup : "L'ordre donné ici est très important : peu importe qu'un programme soit rapide si ses résultats sont faux. De même, un programme correct et efficace peut être si compliqué et mal écrit qu'il faudra le jeter ou le récrire complètement pour en produire une nouvelle version. N'oubliez pas que les programmes utiles seront toujours modifiés pour répondre à de nouveaux besoins."

> Règle n°4 : Les algorithmes élégants comportent plus d'erreurs que ceux qui sont plus simples, et ils sont plus difficiles à appliquer. Utilisez des algorithmes simples ainsi que des structures de données simples.
>
> — Rob Pike (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google)

Cette règle n°4 est une des instances de la philosophie de conception KISS (*Keep it Simple, Stupid* dans le sens de « Ne complique pas les choses ») ou Principe KISS, dont la ligne directrice de conception préconise de rechercher la simplicité dans la conception et que toute complexité non nécessaire devrait être évitée.

Le principe KISS se retrouve dans la philosophie d'Unix. Il apparait aussi dans le Zen de Python sous la forme « Préfère... le simple au complexe [et]... le complexe au compliqué ».

L'expression ironique inverse (devise shadok) : « Pourquoi faire simple quand on peut faire compliqué ? » est assimilable au principe KISS.

Écrire de "bons programmes", c'est **écrire des programmes corrects, simples et efficaces**.

Il faut accepter ces principes pour devenir des professionnels. En termes pratiques, cela signifie que on ne peut pas se contenter d'aligner du code jusqu'à ce qu'il ait l'air de fonctionner : on doit se soucier de sa structure. Paradoxalement, le fait de s'intéresser à la structure et à la "qualité du code" est souvent le moyen le plus facile de faire fonctionner un programme.

En programmation, les principes à respecter s'expriment par des **règles de codage** et des **bonnes pratiques**.

Ces principes permettront de présenter du code clair, simple et fonctionnel. Cela aidera à la compréhension, à la relecture, au contrôle visuel, à la maintenance, au travail collaboratif, aux évolutions et améliorations. Des activités classiques du développement logiciel !

2. Définitions

2.1. Bonnes pratiques et règles de codage

Bonnes pratiques

L'expression « bonnes pratiques » (*best practice* ou *good practice*) désigne, dans un milieu professionnel donné, un ensemble de comportements qui font consensus et qui sont considérés comme indispensables par la plupart des professionnels du domaine. Ils sont souvent établis dans le cadre d'une démarche de qualité.

Règles de codage

Les règles de codage sont un ensemble de règles à suivre pour uniformiser les pratiques de développement logiciel, diffuser les bonnes pratiques de développement et éviter les erreurs de développement "classiques" au sein d'un groupe de développeurs. Les règles de codage s'articulent autour de plusieurs thèmes, les plus courants étant : le nommage et l'organisation des fichiers du code source, le style d'indentation, les conventions de nommage, les commentaires et documentation du code source, recommandations sur la déclaration des variables, sur l'écriture des instructions, des structures de contrôle et l'usage des parenthèses dans les expressions … Les règles de codage participent à la qualité logicielle.

Quelques règles de codage célèbres :

- GNU coding standards écrit par Richard Stallman
- Linux kernel coding style écrit par Linus Torvalds
- Microsoft Naming Guidelines
- Code Conventions for the Java Programming Language
- Style Guide for Python Code

> First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.
>
> — Linus Torvalds (créateur du noyau Linux et de git)

Par exemple, la première règle de codage extraite de Linux kernel coding style :

1) Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

> Learn the rules like a pro, so you can break them like an artist.
>
> — Pablo Picasso

2.2. Notations typographiques

Une notation est très utilisée en programmation informatique car il est souvent nécessaire de nommer lisiblement des concepts compliqués avec une suite de lettres sans espace ni ponctuation.

La majorité des langages informatiques utilisent des **fichiers texte** (ASCII ou Unicode) et n'acceptent généralement que des caractères encodés en ASCII standard (7 bits).

Certains caractères (`=`, `+`, `-`, `.` ...) sont réservés par le langage et ne peuvent donc pas être utilisés comme identifiant.

2.2.1. Camel case

Le Camel case (littéralement « casse de chameau ») est une notation consistant à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale (majuscule) la première lettre de chaque mot. Ces noms doivent en effet être constitués de lettres et de chiffres sans espace.

Origine de Camel case

La notation *camel case* semble avoir été pratiquée en premier par les Écossais pour écrire leurs noms de famille comme le clan MacLeod. Au XXe siècle, elle a été utilisée par des marques ou entreprises, par exemple CinemaScope dans les années 1950. Cette façon d'écrire est aussi devenue une mode, le marketing ayant généré de nombreuses marques de cette forme, comme MasterCard, PlayStation, iPhone, etc. Dans les années 1970 cette notation est adoptée pour écrire les noms des variables, et fonctions dans de nombreux langages de programmation informatique.

Il existe deux variantes concernant la casse de la première lettre :

- en *lower camel case*, elle est en minuscule
- en *upper camel case* (ou *Pascal case*), elle est en majuscule

Exemple en Javascript

```
var body = document.getElementsByTagName("body");
var myFirstParagraph = document.createElement("p");
var helloWorld = document.createTextNode("Hello, world!");
myFirstParagraph.appendChild(helloWorld);
body.item(0).appendChild(myFirstParagraph);
```

La langue anglaise est plus propice à l'emploi de *camel case* que la française, où l'adjonction de particules (de, à ...) rend ces mots plus longs.

2.2.2. Pascal case

En *Pascal case* (ou *upper camel case*), la première lettre de l'identificateur et la première lettre de chaque mot concaténé sont en majuscules.

2.2.3. Snake case

Le Snake case est une convention typographique en informatique consistant à écrire des ensembles de mots, généralement, en minuscules en les séparant par des tirets bas _ (underscore). Cette convention s'oppose par exemple au Camel case qui consiste à mettre en majuscule les premières lettres de chaque mot.

Cette convention est conseillée dans certains langages de programmation :

- En Python, pour les noms de variables, de fonctions et de méthodes
- En Ruby, pour les noms de méthodes et de variables.
- En Rust, pour les noms de variables, méthodes, fonctions, modules ...

Exemples :

- "nom de variable" devient `nom_de_variable`
- "NomDeVariableUpperCamelCase" devient `nom_de_variable_upper_camel_case`
- "Variable" devient `variable`
- "variable" devient `variable` (pas de changement)

2.2.4. Screaming snake case

Le *Screaming snake case* est une variante du *snake case* qui consiste à écrire ces ensembles de mots en les séparant par des tirets bas `_` (underscore), mais en **majuscules**.

Elle est surtout utilisée pour écrire des **constantes**, en Ruby et en Python par exemple. C'est une convention originaire du langage C, dans lequel les constantes sont le plus souvent définies en tant que macros, et cette convention s'applique alors aussi aux autres types de macros. Java l'utilise abondamment pour les énumérations informelles (sans enum).

Cela donne `JE_SUIS_UNE_CONSTANTE`.

2.3. Notations informatiques

2.3.1. Notation hongroise

La notation hongroise est, en programmation informatique, une convention de nommage des variables et des fonctions qui met en avant soit leur usage, soit leur type.

Par exemple :

- la variable booléenne `danger` est préfixée par un `b` pour indiquer un type booléen : `bDanger`
- la variable indexant un `client` sera préfixée par un `idx` pour indiquer son usage : `idxClient`

On distingue en principe deux notations hongroises :

- *Apps* qui préfixe le nom des variables de manière à indiquer son utilisation
- *Systems* qui préfixe le nom des variables de manière à indiquer son type

Cette notation est notamment utilisée par **Microsoft** qui a introduit deux coutumes pour sa notation *Systems* : utiliser trois caractères au lieu d'un pour qualifier une variable ou une constante, et commencer chaque nom par une majuscule au sein d'un mot comportant plusieurs noms.

Lien : Naming Guidelines chez Microsoft©

2.3.2. Sigil

Un sigil est le premier caractère d'un identificateur en langage Perl. Il est non alphanumérique et dénote son type. Perl 6 introduit des *sigils* secondaires appelés *twigils*.

Exemple en Perl

```
$a   # variable de type scalaire
@a   # variable de type tableau
%a   # variable de type hash
\&a  # fonction
```

Le fait que le type fasse partie de la syntaxe comporte un avantage considérable : le type est ainsi indissociable du nom. C'est une forme d'autodocumentation.

2.4. Variable métasyntaxique

Ce terme fait partie du jargon informatique.

En programmation informatique, une variable métasyntaxique est une variable générique (`var` pour variable serait l'exemple type). Ces variables sont utilisées dans les exemples pour se concentrer sur le fond plutôt que sur la forme.

Leurs noms sont choisis pour être tacitement reconnus comme tel par les administrateurs et les programmeurs. Le mot `toto` est l'exemple le plus parlant. L'utilisation des variables métasyntaxiques permet de libérer le programmeur de la recherche d'un nom de variable logique adéquat au sujet étudié.

Les plus courants :

- `toto` : traditionnellement, la première variable métasyntaxique d'un programme ou d'une fonction s'appellera `toto`. Il est possible de créer autant de variantes de toto qu'il y a de voyelles : `tata`, `titi`, `tete`, `tutu`, `tyty`.
- `foo` : historiquement `fu`, pour *fucked up*, ou peut-être *forward observation officer*, connus pendant la Seconde Guerre mondiale notamment pour les inscriptions laissées derrière les lignes ennemies *foo was here* ; selon une autre interprétation, il s'agirait de l'acronyme de *File Or Object*. Utilisé en langage C comme nom de fonction.
- `bar`, suite de `foo` : foobar est alors l'acronyme de *fucked up beyond all recognition / repair*. Utilisé en langage C comme deuxième nom de fonction.
- `i`, `j` et `k` : en Fortran, premier langage scientifique de très large utilisation, la commodité d'utiliser les variables I, J, K… (en fait, toute variable commençant par une lettre de I à N) sans avoir à les déclarer ni à préciser qu'elles étaient entières a contribué à la popularité du langage : c'était elles qu'on utilisait le plus souvent comme indices de boucle.
- 42, valeur métasyntaxique, réponse à la grande question sur la vie, l'univers et le reste.

Bien évidemment, les variables métasyntaxiques ne doivent jamais apparaître dans un projet logiciel professionnel.

2.5. Environnements de développement intégré

Un environnement de développement intégré (IDE ou EDI en français) est un ensemble d'outils qui permet d'augmenter la productivité des programmeurs qui conçoivent des logiciels. En aucun cas, ils doivent être considérés comme des outils indispensables ou obligatoires.

> Le marteau ne fait pas l'architecte !

Un EDI comporte généralement un **éditeur de texte** destiné à la programmation, des fonctions qui permettent, par pression sur un bouton, de démarrer le **compilateur** ou l'**éditeur de liens**, d'**exécuter** le programme ainsi qu'un **débogueur** en ligne, de consulter la **documentation**, etc ... Certains environnements sont dédiés à un langage de programmation (IDLE pour Python par exemple), un *framework* (Qt Creator pour Qt par exemple) ou une architecture (Xcode pour Mac OS X et iOS par exemple) en particulier.

3. Guide de programmation en BTS SN (LaSalle Avignon)

Ce guide de programmation intègre les règles de codage, bonnes pratiques et conventions de nommage à appliquer pour les travaux pratiques, les mini-projets et les projets réalisés en BTS SN (LaSalle Avignon). Elle concerne essentiellement le développement en C / C++.

Le C++ est une extension du langage C qui a apporté notamment le concept de programmation orientée objet (POO). La programmation orientée objet introduit les concepts de classe, d'attributs (des variables membres) et de méthodes (des fonctions membres).

3.1. Travaux Pratiques

Les Travaux Pratiques ont pour but d'établir ou de renforcer vos compétences pratiques. Vous pouvez penser que vous comprenez tout ce que vous lisez ou tout ce que vous a dit votre enseignant mais la répétition et la pratique sont nécessaires pour développer des compétences en programmation.

Ceci est comparable au sport ou à la musique ou à tout autre métier demandant un long entraînement pour acquérir l'habileté nécessaire. Imaginez quelqu'un qui voudrait disputer une compétition dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.

3.2. Mini-projets et projets

Les activités de mini-projet et de projet ont pour objectifs :

- d'analyser l'expression d'un besoin client
- d'organiser et respecter la planification d'un projet

- de travailler en équipe
- de contribuer à la modélisation de tout ou partie d'un module logiciel et/ou matériel
- de réaliser la conception détaillée d'un module logiciel et/ou matériel
- de tester et valider un module logiciel et/ou matériel
- de proposer des corrections ou des améliorations
- de documenter une réalisation logicielle et/ou matérielle
- d'assurer la traçabilité

3.3. Règles de codage et conventions de nommage

Ces règles de codage et conventions de nommage ont pour objectifs de rendre les programmes plus clairs et lisibles et de faciliter le travail collaboratif.

Il est interdit d'utiliser des caractères de contrôle (comme l'espace) ou des caractères étendus (comme le `é`). Seuls les caractères du jeu ASCII standard sur 7 bits sont autorisés (`man ascii`).

3.3.1. Les répertoires

Les répertoires définissent la structure d'un projet logiciel.

Les noms de répertoires sont des **noms principaux** (surtout pas un verbe) suffisamment éloquents (ils doivent synthétiser ce qu'ils contiennent).

Exemple de structure classique pour un projet logiciel sous GNU / Linux :

- `bin` contient le(s) exécutable(s)
- `doc` contient la documentation
- `include` contient les fichiers de déclaration (les fichiers d'en-tête ou *header*)
- `lib` contient les bibliothèques
- `src` contient les fichiers sources
- `tests` contient les tests unitaires

3.3.2. Les fichiers

Un nom de fichier est un **nom principal** (surtout pas un verbe) suffisamment éloquent (il doit synthétiser ce qu'il contient).

Exemples :

- les fichiers exécutable et sources sont en *Camel Case* : `racineCarre.exe`, `racineCarre.cpp`, `main.cpp`, ...
- les classes sont en *Pascal Case* : `LampeDePoche.h` et `LampeDePoche.cpp` pour la classe `LampeDePoche`

L'utilisation des minuscules et du `-` (tiret haut) est acceptée : `racine-carre.exe`

3.3.3. Les variables et attributs

Un nom de variable (ou d'attribut) est un **nom principal** (surtout pas un verbe) suffisamment éloquent, éventuellement complété par :

- une caractéristique d'organisation ou d'usage
- un qualificatif ou d'autres noms

On utilisera la convention *camel case*.

Exemples : `distance`, `distanceMax`, `consigneCourante`, `etatBoutonGaucheSouris`, ...

Certaines abréviations sont admises quand elles sont d'usage courant : `nbre` (ou `nb`), `max`, `min`, ... Les lettres `i`, `j` et `k` utilisées seules sont usuellement admises pour les indices de boucles.

3.3.4. Les constantes

Les identificateurs des constantes sont des **noms principaux** (en *Screaming snake case*) qui doivent être composés de capitales (majuscules) et de tirets bas (_).

Les variables qualifiées `const` peuvent être écrites comme des variables en camel case.

Une constante en Screaming snake case ne doit jamais commencé par un tiret bas (_) ou deux tirets bas (__) car cette notation est réservée par le système.

3.3.5. Les fonctions et méthodes

Un nom de fonction est construit à l'aide d'un **verbe** (surtout pas un nom), et éventuellement d'éléments supplémentaires comme :

- une quantité
- un complément d'objet
- un adjectif représentatif d'un état

Le verbe peut être au présent de l'indicatif ou à l'infinitif.

On utilisera la convention *camel case*.

Exemples : `ajouter()`, `sauverValeur()`, `estPresent()`, `estVide()`, `remplirReservoir()`, ...

Les noms des fonctions retournant un booléen (`bool`) sont généralement préfixées du verbe être (ou avoir) au présent : `estPlein()`

3.3.6. Les classes

Un nom de classe est un **nom principal** (surtout pas un verbe) suffisamment éloquent (il représente un concept), éventuellement complété par :

- une caractéristique d'organisation ou d'usage
- un qualificatif ou d'autres noms

On utilise la convention *Pascal case*.

Exemples : `Point`, `PointCouleur`, `CommunicationBluetooth`, ...

3.3.7. Les commentaires

On écrit des commentaires pour décrire ce que le programme est supposé faire.

On utilise seulement les commentaires pour fournir des informations utiles impossibles à exprimer directement dans le code.

Il faut distinguer deux types de commentaires :

- Les commentaires "internes" ou "privés" : ces commentaires sont destinés aux développeurs et restent dans le code source. Ils ne seront donc pas extraits pour alimenter une documentation.

Des commentaires pour les développeurs

```
// Un commentaire interne ou privé (mais utile) sur une seule ligne

/*
 * Un commentaire interne ou privé (mais utile) sur plusieurs lignes
 */
```

- Les commentaires "externes" ou "publics" : ces commentaires sont destinés à la documentation et seront donc extraits (par l'utilisation de tags préfixés par `@` ici) par un logiciel de documentation automatique comme Doxygen.

Des commentaires pour Doxygen

```
/**
 * @def NB
 * @brief Définit le nombre 42 !
*/
#define NB 42 //!< Un nombre NB

/**
 * @class       Exemple exemple.h "exemple.h"
 * @brief       La déclaration de la classe Exemple
 * @details     La classe \c Exemple permet de montrer l'utilisation des \em tags \b
Doxygen
 * @author      Thierry vaira <tvaira@free.fr>
 * @version     0.1
 * @date        2020
 */
class Exemple
{
    private:
        int a; //!< a est ...
};

/**
 * @brief Accesseur de l'attribut a
 * @return a la valeur de l'attribut a
 * @retval int la valeur de l'attribut a
 */
int Exemple::getA() const
{
    return a;
}

/**
 * @brief Une fonction ...
 * @param a ...
 */
void foo(int a);
```

Voir aussi : Documentation du code avec Doxygen

3.4. Bonnes pratiques

Ces bonnes pratiques augmentent les chances d'obtenir un programme qui fonctionne correctement.

3.4.1. Les fichiers

- Les fichiers d'en-tête (*header*) d'extension `.h`, `.hh` ou `.hpp` ne contiennent que des déclarations et

aucune définition.

- Les fichiers sources C/C++ d'extension `.c`, `.cpp` ou `.cc` ne contiennent que des définitions et aucune déclaration. Pour accéder aux déclarations dont ils ont besoin, on utilise la directive de préprocesseur (ou de pré-compilation) `#include`
- Les fichiers sources d'extension `.c`, `.cpp` ou `.cc` doivent pouvoir se compiler séparément avec : `g++ -c <fichier>.cpp`
- On n'utilise jamais de chemins absolus dans une directive `#include`. Les chemins sont précisés avec l'option `I<chemin>` à la compilation.
- En projet, les fichiers possèderont un en-tête de documentation (précisant au moins sa description, l'auteur et la version).
- Les fichiers d'en-tête (*header*) d'extension `.h`, `.hh` ou `.hpp` doivent être protégés contre le risque d'inclusion multiple.

Protection contre l'inclusion multiple d'un fichier `outils.h`

```
#ifndef OUTILS_H
#define OUTILS_H

// Dans un fichier .h, on peut :

// inclure d'autres fichiers .h de déclarations (seulement si c'est nécessaire)

// déclarer ses propres constantes avec #define

// déclarer des variables définies à l'extérieur avec extern

// déclarer des définitions de type avec typedef

// déclarer des énumérations avec enum

// déclarer des structures de données avec struct (généralement une seule par fichier)

// déclarer des classes avec class (généralement une seule par fichier)

// déclarer des fonctions par leur prototype
// (plusieurs si elles sont liées par un même concept)

#endif /* OUTILS_H */
```

3.4.2. Formatage

- On indente avec des **espaces** (3 ou 4). L'indentation est obligatoire (notamment après une accolade ouvrante `{`).
- On sépare les identifiants d'une expression avec un espace : `int a = 0;`
- On ne sépare les traitements que par **une et une seule ligne vide**.
- L'accolade ouvrante `{` d'un bloc est placée après un saut de ligne.

3.4.3. Variables

- Les noms de variable sont des noms !
- Toutes les variables doivent être définies avec une valeur initiale.
- Les variables globales sont déconseillées (risque d'effet de bord).
- Deux nombres flottants ne doivent pas être comparés pour une stricte égalité (`==`).
- On ne teste pas l'égalité d'un booléen à `true` ou à `false`, on écrit ceci : `if (estCher) ⋯` ou `if (!estCher) ⋯`
- On utilise des variables booléennes à chaque fois que cela est possible.
- En projet, les attributs possèderont une information de documentation.

3.4.4. Constantes

- On n'écrit pas de valeurs « en dur » (comme `10` ou `3.14`) dans une expression car ce sont des constantes.

3.4.5. Fonctions

- Les noms de fonction sont des verbes !
- La fonction réalise une **seule action logique** et bien évidemment celle qui correspond à son nom.
- On limitera la taille des fonctions à une valeur comprise entre **10 à 15 lignes maximum** (accolades exclues).
- On évitera le plus possible d'utiliser des saisies et des affichages dans les fonctions (à l'exception des fonctions dont c'est la responsabilité) pour permettre notamment leur ré-utilisation.
- L'ordre de définition des paramètres doit respecter la règle suivante : `nomFonction(parametrePrincipal, listeParametres)` où `parametrePrincipal` est la donnée principale sur laquelle porte la fonction, la `listeParametres` ne comportant que des données secondaires, nécessaires à la réalisation du traitement réalisé par la fonction.
- Les fonctions qui ne reçoivent aucun paramètre et qui ne retourne aucun résultat (comme `void foo()`) sont très suspectes !
- En C++, les paramètres (ou arguments) d'une fonction (ou d'une méthode) sont passées par **référence** (une **référence constante** si la fonction ne doit pas le modifier).
- Les méthodes doivent être déclarées `const` si elles ne modifient aucun attribut de la classe.
- En projet, les méthodes possèderont une information de documentation.

3.4.6. Structures de contrôle

- On met les blocs `if` et `else` entre accolades (idem pour les autres blocs comme `for`, `while`,). Le bloc `else` est obligatoire même s'il ne contient aucune instruction. Cela est valable aussi pour `default` dans un `switch`.
- On n'utilise pas l'instruction `break` (sauf éventuellement dans un `switch`) pour sortir d'une structure de contrôle (`for`, `while`, `if`, ...). Les boucles infinies du type `while(1)` et `for(;;)` sont

fortement déconseillées.

3.4.7. Classes

- Les noms de classe sont des noms (de concept) ! N'oubliez pas qu'une classe est la définition d'un nouveau **type**.
- Concentrer vos efforts en premier sur l'identification des caractéristiques ou propriétés des classes (c'est-à-dire le nom et le type des attributs)
- Créer des objets à partir d'une classe se nomme l'instanciation (ou instancier). Une instance de classe est donc un objet.
- Utilisez par défaut le nom de la classe en minuscule pour le nom de l'objet que vous créez : `LampeDePoche lampeDePoche;` (`laLampeDePoche` et `maLampeDePoche` sont corrects aussi)
- En C++, les mots clés `struct` et `class` sont équivalents à l'exception que les membres d'une structure `struct` sont considérées comme `public` par défaut et `private` pour une classe `class`.
- Le mot clé `struct` sera réservé pour déclarer seulement des structures de données (qui ne contiennent pas de méthodes, même si cela n'est pas interdit par le langage C++)
- Les classes qui ne contiennent aucun ou un seul attribut sont très suspectes !
- En projet, les classes possèderont un en-tête de documentation (précisant au moins sa description, l'auteur et la version).

3.4.8. Commentaires

- On commente le moins possible ! C'est possible en respectant les règles de codage et les bonnes pratiques. Il faut savoir qu'en pratique les commentaires sont difficilement maintenus à jour par les développeurs.
- On ne laisse jamais des fragments de code en commentaires.

3.4.9. Programme principal

- Le minimum attendu : on doit pouvoir fabriquer un exécutable et le lancer !
- On doit fournir les règles de fabrication dans un fichier de type `Makefile` ou équivalent
- Le programme doit énoncer ce qu'il fait et faire ce qu'il énonce !
- On limitera évidemment la taille de la fonction `main()` à une valeur comprise entre **10 à 15 lignes maximum** (accolades exclues).
- En projet, le programme principal possèdera un en-tête de documentation (précisant au moins sa description, l'auteur et la version).

La première ligne d'un programme doit énoncer simplement ce que le programme est censé faire et pas ce que nous avons voulu qu'il fasse ! Prenez donc l'habitude de mettre ce type de commentaire au début d'un programme.

3.4.10. Débogage (*release* vs *debug*)

Le débogage est un processus de diagnostic, de localisation et d'élimination des erreurs des programmes informatiques. Le débogage permet d'obtenir des programmes qui donnent des résultats corrects.

On utilise un débugueur (de l'anglais *debugger*) ou débogueur (de la francisation bogue).

> Un débogueur est un logiciel qui aide un développeur à analyser les *bugs* (bogues) d'un programme. Pour cela, il permet d'exécuter le programme en pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme ...
>
> Le programme à débuguer est exécuté à travers le débogueur et s'exécute normalement. Le débogueur offre alors au programmeur la possibilité d'observer et de contrôler l'exécution du programme.
>
> GNU Debugger, également appelé `gdb`, est le débogueur standard du projet GNU. Il est portable sur de nombreux systèmes type Unix et fonctionne pour plusieurs langages de programmation, comme le C et le C++. Il fut écrit par Richard Stallman en 1988. `gdb` est un logiciel libre, distribué sous la licence GNU GPL. L'interface de `gdb` est une simple ligne de commande, mais il existe des applications qui lui offrent une interface graphique beaucoup plus conviviale.
>
> Pour débuguer un programme avec `gdb`, il faut l'avoir compilé avec l'option `-g` de `g++`.
>
> Notons également que `gdb` est souvent invoqué en arrière-plan par les environnements de développement intégré (IDE).

Il est aussi possible (mais déconseillé si l'utilisation d'un débogueur est possible) d'utiliser de simples affichages pendant l'exécution du programme.

Dans ce cas, on procédera de la manière suivante :

- On doit pouvoir activer et désactiver les affichages de débogage
- On doit afficher des messages concis et précis (par exemple, le nom de la variable et sa valeur)
- On doit afficher le nom du fichier, de la fonction et la ligne qui produit l'affichage de débogage

Exemple pour l'affichage d'un message de débogage

```
// Ce programme permet la saisie d'une année !

#include <iostream> /* pour cin et cout */

#define DEBUG ①

int main()
{
    int annee = 0;

    std::cout << "Entrez une année : ";
    std::cin >> annee;

    #ifdef DEBUG
    std::cout << "[" << __FILE__ << ":" << __FUNCTION__ << "():" << __LINE__    << "]
" << "annee" << " = " << annee << std::endl;
    #endif

    // ...

    return 0;
}
```

① On placera cette ligne en commentaire pour désactiver les messages de débugage et obtenir une version *release*. Il est possible d'obtenir une version *debug* en intégrant l'activation des message directement au moment de la compilation : `g++ -DDEBUG …`

On obtient :

```
$ g++ annee.cpp
$ ./a.out
Entrez une année : 2020
[annee.cpp:main():12] annee = 2020
```

release **vs** *debug*

Une version *release* est une version livrable du logiciel. C'est celle qu'on fournit au client. Elle ne doit contenir aucun affichage de débugage.

Une version *debug* est une version en cours de développement. Elle peut contenir des affichages de débugage.

3.4.11. Divers

- Il ne faut pas confondre les **opérateurs booléens** (`&&`, `||` et `!`) qui traitent des valeurs comme `true` ou `false` et les **opérateurs bit à bit** (`&`, `|` et `~`) qui traitent les valeurs en binaire (composées

de `0` et de `1`).

- Pour limiter la taille des blocs d'instruction à **10 ou 15 lignes maximum** (accolades exclues), on pratique le remaniement de code qui consiste à extraire des fonctions à partir des lignes de code existantes.

Il est indispensable de décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre. Cette approche consistant à décomposer une tâche complexe en une suite d'étapes plus petites (et donc plus facile à gérer) ne s'applique pas uniquement à la programmation et aux ordinateurs. Elle est courante et utile dans la plupart des domaines de l'existence.

> Diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.
>
> — Descartes (mathématicien, physicien et philosophe français) dans le Discours de la méthode

- Concentrer vos efforts sur l'identification et les caractéristiques de vos données (c'est-à-dire le nom et le type de vos variables)

> Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes.
>
> — Rob Pike (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google)

Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! ». Les types sont au centre de la plupart des notions de programmes corrects, et certaines des techniques de construction de programmes les plus efficaces reposent sur la conception et l'utilisation des types.

4. Exemple

Exemple

```
/**
 * @file        estMajeur.cpp
 *
 * @brief       Ce programme assure la saisie de l'age d'une personne et affiche si
celle-ci est majeure
 * @author      Thierry vaira <tvaira@free.fr>
 * @version     0.1
 * @date        2020
 * @attention   Aucune vérification de saisie n'est effectuée
 *
 * @return      int 0 si le programme s'est exécuté avec succès
 *
 */

#include <iostream> /* pour cin et cout */
#include <cstdlib> /* pour EXIT_SUCCESS */

#define AGE_MAJORITE_FRANCE   18

int main()
{
    unsigned int age = 0;

    std::cout << "Entrez un age : ";
    std::cin >> age;

    bool estMajeur = (age >= AGE_MAJORITE_FRANCE);

    if(estMajeur)
    {
        std::cout << "Vous êtes majeur." << std::endl;
    }
    else
    {
    }

    return EXIT_SUCCESS;
}
```

5. Voir aussi

- Documentation du code avec Doxygen
- Rédaction de documents techniques avec Markdown ou Asciidoc
- Visual Studio Code (VSCode) et PlatformIO