CLEAN CODE

Les professeurs d'informatique de la section

Informatique de gestion

HENALLUX – Catégorie économique

« N'importe quel idiot est capable d'écrire du code qu'un ordinateur pourra comprendre.
Un bon programmeur, lui, écrit du code que des humains pourront comprendre. »
M. Fowler

« Quand vous écrivez du code, faites-le toujours en pensant que celui qui devra le modifier est un psychopathe qui sait exactement où vous habitez. »
M. Golding

Introduction

La plupart des entreprises et des groupes de programmation possèdent un « code de programmation » (coding standards) reprenant diverses règles portant sur la manière dont le code doit être conçu et présenté.

En tant que programmeurs, vous allez sans aucun doute développer un style de programmation personnel (vous avez probablement déjà commencé à le faire). Ce style s'est construit peu à peu au fil des habitudes que vous avez prises, soit personnellement, soit en vous inspirant des exemples donnés par vos professeurs. Mais la construction d'un style de programmation est une tâche sans fin : il est toujours bon de remettre ses habitudes en question et de voir s'il n'y a pas lieu de les améliorer. Et la meilleure manière de le faire, c'est en observant, avec un esprit critique, ce que d'autres font.

Si vous participez tôt ou tard à un projet de programmation en groupe (en entreprise ou au sein d'une organisation moins formelle), vous devrez vous adapter au style choisi pour faciliter les échanges entre programmeurs. Concrètement, cela signifie que, même si vous n'êtes pas d'accord avec les conventions choisies, vous devrez être capables de les respecter. De même, les demandes des utilisateurs de vos programmes ne correspondront pas toujours à ce que vous auriez pu imaginer.

Ce document vise plusieurs buts :

- proposer des conseils en matière de présentation du code (le but est que vous vous posiez des questions telles que « Serait-ce une bonne chose pour moi d'adopter cette convention ? » en imaginant que ce code devra être relu beaucoup plus tard par vous-même ou une tierce personne) ;
- souligner l'importance de la présentation du code (tant pour le dossier de programmation que pour vos futurs travaux, à l'école, dans vos loisirs, ou dans votre futur job) ;
- faire le point sur la manière dont un projet de programmation peut être pensé et organisé de manière modulaire (programmation multi-fichiers, fichiers d'en-tête) ou en classes interagissant entre elles.

Deux règles d'or

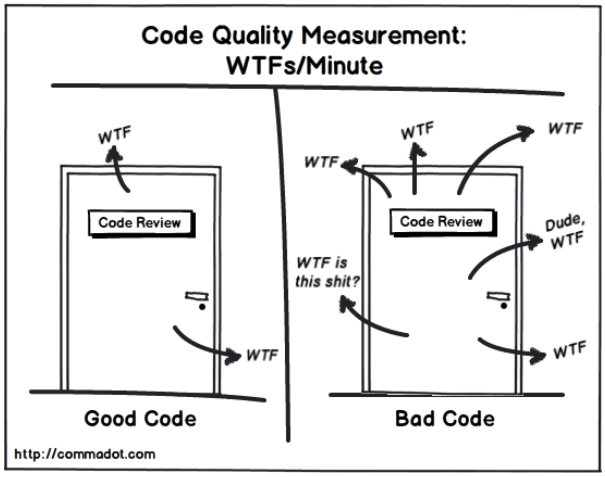
Pourquoi perdre du temps à établir des règles au sujet de la manière dont il faut écrire et présenter les programmes ?

Un programme n'est que très rarement un ouvrage définitivement terminé. Bien souvent, il faut retravailler le code, encore et encore. On peut revenir sur un code déjà écrit pour de multiples raisons :

- pour le déboguer, en partant en chasse contre les erreurs ;
- pour le modifier parce que les désirs de l'utilisateur ont changé ;
- pour l'améliorer, le rendre plus efficace ;
- pour l'étendre, en ajoutant de nouvelles fonctionnalités ;
- pour le revisiter après quelques années de développement (rétroingénerie), l'analyser en vue d'une refonte plus structurée, plus actualisée ou dans un autre langage;
- ou pour comprendre comment fonctionne un programme écrit par une autre personne.

Ces tâches sont si fréquentes qu'un adage dit qu'un programmeur passe seulement 5% de son temps de travail à *écrire* du code, et les 95% restants, à *lire* du code. Cela souligne l'extrême importance que revêt la présentation du code : plus un code est élégant, judicieusement structuré, présenté de manière cohérente, agréable et facile à lire et plus ces tâches seront simplifiées !

Si vous prenez le temps d'écrire vos programmes correctement, tous ceux qui auront à les (re)lire vous en seront reconnaissants. Cela inclut vos professeurs, vos futurs collègues, ceux qui voudront s'inspirer de vos écrits et vous-mêmes, quand vous voudrez reprendre un code plusieurs mois après l'avoir écrit ! Vous y gagnerez également en efficacité : dans un programme propre (« clean code »), les bugs ont beaucoup plus de mal à se cacher !



La plupart des conseils donnés dans les pages suivantes découlent de deux principes fondamentaux ou règles d'or.

Première règle d'or : écrivez votre programme avant tout pour ceux qui vont devoir le lire (le comprendre, le corriger, le modifier, le maintenir, le coter).
Cela signifie entre autres :

- prenez bien soin d'indenter et d'aérer votre code pour le rendre agréable à lire ;
- choisissez des noms (variables, fonctions/méthodes…) qui expriment clairement vos intentions ;
- découpez votre code en parties bien pensées (modules, fonctions/méthodes, groupes d'instructions).

Seconde règle d'or : le principe du « lieu de modification unique ».
Ce principe repose sur l'idée suivante. Certaines parties de votre programme devront tôt ou tard être modifiées ; faites en sorte que ces modifications puissent se faire en changeant le moins de lignes de code possible. Quelques exemples d'application :

- définissez des constantes symboliques (taille de tableau) au lieu d'utiliser des valeurs numériques : en cas de changement, vous n'aurez qu'une seule ligne à modifier ;
- plutôt que de copier/coller un bout de programme utilisé plusieurs fois, faites-en une fonction : en cas de changement, vous ne devrez modifier qu'une seule fonction plutôt que plusieurs copies;
- isoler les parties d'impression et d'obtention en cas de changement de l'origine des données (écran, fichiers, bases de données, ...) - en informatique, c'est l'approche dite en couches

Structure de ce document

Ce document est divisé en deux parties.

Dans la première (Écriture du code) sont présentés divers conseils relatifs à la manière de construire et présenter votre code. On y traite de sujets allant de l'indentation au choix des noms (variables, fonctions, méthodes entre autres) en passant par la conception des fonctions/méthodes et l'organisation de vos fichiers sources. Le but est double : vous suggérer comment présenter un code lisible, et vous inciter à choisir des règles de programmation auxquelles vous vous tiendrez (style cohérent).

La seconde partie (Modularité en C) traite de la manière de décomposer votre code en plusieurs fichiers et de construire vos fichiers d'en-tête (« fichiers .h »). Contrairement à la première partie, qui peut s'appliquer à d'autres langages de programmation, celle-ci est plus spécifique au C.

Partie 1 : Écriture du code

Les suggestions proposées ci-dessous se basent à la fois sur le bon sens et sur l'expérience de divers programmeurs et auteurs qui ont mûrement réfléchi au sujet.

1. Présentation du code

Indentation

Indentez le code correctement et de manière cohérente. C'est une absolue nécessité pour le rendre lisible. Les environnements de développement ou IDE (dont Visual Studio, NetBeans, Eclipse) peuvent le faire pour vous !

Longueur des lignes

Évitez absolument toute ligne faisant plus de 79 caractères de long (problème à l'impression et difficulté de lecture).

Utilisation des lignes vides

Agrémentez votre code de lignes vides pour l'aérer. Utilisez par exemple une ligne vide pour séparer les déclarations de variables des instructions, ou encore pour séparer les instructions en groupes correspondent à une même action.
Une utilisation judicieuse des lignes vides vous permet de découper votre code en « blocs » qui peuvent correspondre à la notion de paragraphe utilisée en Principes de Programmation (paragraphe « initialisation », paragraphe « avant l'entrée dans un bloc logique », paragraphe « après avoir lu tout un bloc logique », etc.).
Par contre, supprimez les lignes vides inutiles!

Placement des accolades

Choisissez un style pour le placement des accolades (dans la définition des fonctions/méthodes, dans les conditionnelles, dans les répétitives et dans la définition des types structures et unions).

Voici les deux styles les plus utilisés.

Version 1	`if (montant > 1000) {`
	`  ristourne = .2 *`
	`  montant;`
	`  pointsFidélité += 3;`
	`}`
		`if (prix > 1000) {`
		`  ristourne = 200;`
		`} else if (prix > 500){`
		`  ristourne = 20;`
		`} else {`
		`  ristourne = 0;`
		`}`
			`while (ptr) {`
			`  printf(…, ptr->info);`
			`  ptr = ptr->suivant;`
			`}`

Version 2	`if (montant > 1000)`
	`{`
	`  ristourne = .2 *`
	`  montant;`
	`  pointsFidélité += 3;`
	`}`
		`if (prix > 1000)`
		`{`
		`  ristourne = 200;`
		`}`
		`else`
		`  if (prix > 500)`
		`  {`
		`  ristourne = 20;`
		`  }`
		`  else`
		`  {`
		`  ristourne = 0;`
		`  }`
			`while (ptr)`
			`{`
			`  printf(…, ptr->info);`
			`  ptr = ptr->suivant;`
			`}`

Utilisation des espaces

Choisissez où mettre et ne pas mettre d'espace et tenez-vous à ce choix.
Voici quelques règles standards :

- ne jamais placer plusieurs espaces de suite (sauf pour créer une indentation ; et encore… préférer les tabulations);
- utiliser une espace si un mot-clef est suivi d'une parenthèse (par exemple, entre un if / while et la condition qui suit, ou entre un for et la parenthèse qui le suit);
- par contre, ne pas placer d'espace entre les noms de fonctions/méthodes et les parenthèses qui les suivent (règle valable à la fois pour les prototypes et pour les appels);
- placer des espaces autour des opérateurs binaires (= de l'affectation, +, -, &&, == et les autres), à l'exception de . et -> (qui s'utilisent sans espace);
- ne pas placer d'espace entre les opérateurs unaires (-, !, ~ et les autres) et leur argument.

Utilisation des commentaires

Si les noms (des variables, des fonctions/méthodes et des arguments) sont parlants, il est inutile d'alourdir le code en y ajoutant des commentaires. Dans ces cas-là, aucun[1] commentaire n'est nécessaire (trop de commentaires rendent le programme difficile à lire).

Par contre, on pourra ajouter un commentaire pertinent pour expliquer une astuce algorithmique ou un choix spécifique d'implémentation. Un commentaire pourra aussi servir de titre à un « bloc de code logique » / « paragraphe » (au sens de Principes de Programmation), mais seulement si c'est nécessaire. Inutile d'intituler explicitement « Initialisation » tous les paragraphes d'initialisation par exemple, surtout si ceux-ci ne comportent que deux ou trois

[1] Certains auteurs considèrent que, lorsqu'il est nécessaire de commenter un code, cela indique que celui-ci est mauvais (noms mal choisis, mauvaise découpe du code…) et doit donc être revu.

lignes. Dans tous les cas, il faut éviter à tout prix les commentaires sans intérêts qui se contentent de paraphraser le code, comme le suivant :

```
i++;        // J'incrémente i.
```

Formatez vos commentaires de manière cohérente (choisissez un format puis tenez-vous-y). Par exemple :

`/* sur une ligne */`	`/* * Sur une ligne, important */`	`/* * Commentaire qui * s'étend sur * plusieurs lignes */`

Si vous placez des commentaires, ne négligez en aucun cas l'orthographe ni la tournure des phrases écrites. N'utilisez pas d'abréviation que seuls vous comprenez.

2. Choix des noms

Bien choisir le nom d'une variable, d'une fonction/méthode ou d'une constante est une étape très importante dans la conception du programme, car c'est via les noms que vous transmettez aux lecteurs ce que vous avez en tête. Un nom bien choisi…

- sera plus facile à retenir lors de l'écriture (et de la lecture) du programme ;
- rendra plus aisée la recherche de bugs et les futures modifications ;
- permettra au lecteur (professeur, collègue, ou vous d'ici quelques mois) de s'y retrouver plus facilement.

Il ne faut pas hésiter à passer quelques secondes à réfléchir au choix d'un nom. Et si, plus tard, vous trouvez un nom plus adéquat, prenez la peine d'éditer votre programme pour remplacer le précédent !

Cohérence des noms (langue)

Utilisez des noms soit en anglais, soit en français et tenez-vous à ce choix tout au long du même programme. Évitez aussi de combiner un mot en français et un mot en anglais au sein d'un même nom, c'est-à-dire évitez le franglais !

Cohérence des noms (casse)

Choisissez une convention sur l'utilisation des majuscules et des minuscules et tenez-vous-y ! Suivant les langages employés, des conventions différentes peuvent être prises : en Java, les noms des méthodes commencent par une minuscule, en C# par une majuscule, en C aucune règle à ce sujet. Néanmoins, voici une proposition que nous vous conseillons pour le langage C (autres langages : cf. cours concernés) :

- constantes / énumérations : tout en majuscules (`TAILLEMAX`, `TAILLE_MAX`, `LONGUEUR_MAX`)

- macros : tout en majuscules
- variables / paramètres de fonctions : initiale en minuscule (`nomJeu`, `indAbonné`)
- fonctions : initiale en minuscule (`rechercheAbonné`, `fraisTotaux`)
- alias/synonymes de types : initiale en majuscule (`Cellule`).

Construction des noms composés

Il existe plusieurs conventions. Choisissez-en une et tenez-vous-y ! Quelques exemples :

- PascalCase : `MontantFacture`, `PrixCarburant`, `TailleEnMètres`
- camelCase : `montantFacture`, `prixCarburant`, `tailleEnMètres`
- caractère souligné (moins utilisé) : `montant_facture`, `prix_carburant`, `taille_en_mètres`, `TAILLE_MAX`.

Noms significatifs

Un nom bien choisi rend la plupart des commentaires inutiles : si vous devez ajouter un commentaire pour expliquer ce que la variable représente ou ce que la fonction/méthode représente, son nom est sans doute mal choisi.
En même temps, il est inutile de surcharger les noms d'informations superflues (excès inverse). Si une variable a une brève portée (maximum 5 lignes), elle peut porter un nom assez court (du moins, si elle n'est pas significative). Par exemple, une variable qui n'est utilisée qu'au sein des 2 ou 3 lignes d'une boucle peut très bien s'appeler `i`, `j` ou `k` : inutile de la baptiser `indiceElementDansTableauPrincipal` (ça réduirait la lisibilité du code) !
De manière générale, plus une variable a une longue durée de vie (c'est-à-dire plus une variable est utilisée dans une grande partie du programme) et plus son nom devrait être explicite / long. Une variable utilisée tout au long du programme ne devrait pas s'appeler `i` ou `ptr`, mais devrait porter un nom plus parlant.

Noms des variables/fonctions/méthodes en rapport avec leur finalité

Les suggestions qui suivent permettent d'harmoniser les noms choisis pour les différentes composantes du programme. Le but global est de rendre le code aussi lisible que possible. Par exemple, si une fonction/méthode porte comme nom un verbe, son prototype peut se lire quasiment comme une phrase : `afficheTableMultiplication(int base)`.

- **Noms des variables / arguments de fonctions(méthodes)**
  Utilisez un nom commun (pas un verbe ni un adjectif).
- **Noms des variables / arguments de fonctions « booléens »**
  Utilisez un nom ou un adjectif qui indique clairement ce que « vrai » et « faux » signifient. Par exemple : `nameFound` ou `aucuneErreur` au lieu de `name` ou `erreur`.

- **Noms des variables de type tableau / pointeur**
  Si vous choisissez une convention (par exemple faire commencer les noms des tableaux par `tab` et les noms des pointeurs par `p` ou `ptr`), tenez-vous-y tout au long du programme.
- **Noms des procédures (fonctions qui accomplissent une action)**
  Utilisez un verbe (infinitif ou conjugué) décrivant l'action que la fonction accomplit. Par exemple : `afficherListeAbonne` au lieu de `affichage`.
- **Noms des fonctions (fonctions qui renvoient un résultat)**
  Utilisez un nom qui décrit la valeur retournée. Par exemple : `totalFacture` au lieu de `calculerPrix`.
  En effet, lire
  ```
  if (totalFacture(facture) > 1000)
      imprimerFacturePapier(facture);
  else
      envoyerFactureÉlectronique(facture);
  ```
  se lira comme une phrase française.
- **Noms des fonctions « booléennes »**
  Utilisez un nom ou adjectif qui décrit ce qu'une valeur de retour « vrai » signifie. Par exemple : `codeCorrect` au lieu de `verificationCode` (un tel nom doit permettre de savoir ce que « vrai » et « faux » signifient, sans hésitation), `estVide` au lieu de `determinerNombreElement`.
- **Noms des types (structures / union)**
  Utilisez un nom au singulier décrivant le type d'objet représenté.
- **Noms des attributs d'une structure (ou union)**
  Même conseil que pour les variables. Inutile de rappeler le nom de la structure dans chaque attribut : les attributs d'une structure `artiste` pourront être simplement `nom`, `prenom` et `age` plutôt que `nomArtiste`, `prenomArtiste` et `ageArtiste` !

3. Déclarations de variables

Une déclaration par ligne

Déclarez une variable par ligne (et profitez-en pour l'initialiser si nécessaire ; inutile de tomber dans le travers de mettre tout au zéro du type correspondant) : cela permet non seulement d'ajouter plus facilement un commentaire si nécessaire mais cela rend également le code plus lisible !

Où les déclarer ?

Au début de la fonction si elles sont utilisées partout à l'intérieur de celle-ci. Au début d'un bloc d'instructions (dans un `if`, `while` ou `switch`) si elles ne sont nécessaires que dans le bloc. Pour rappel, un bloc (en C, Java, C#) est un ensemble d'instructions placées entre accolades.

Utilisation de variables réelles

Préférez le type `double` au type `float`. Attention lors des tests d'égalité entre valeurs réelles : vu les approximations effectuées pour stocker ces nombres en mémoire, ne testez pas l'égalité pure (`x == y`) mais autorisez une certaine imprécision (`abs(x-y) <= epsilon` où epsilon est un nombre très petit, par exemple 0.00000001).

Utilisation de variables intermédiaires

Si cela rend le code plus clair et plus aisé à lire, et surtout si cela permet d'éviter de calculer plusieurs fois la même chose ou de faire appel plusieurs fois à un élément d'une collection, introduisez des variables intermédiaires (voir exemples ci-dessous). Le coût d'une variable intermédiaire (un emplacement mémoire) étant quasiment insignifiant, il serait dommage de s'en passer !

… devient …	

`if (level + 1 >= …)`
`  … level + 1 …`
	`int levelUp = level + 1;`
	`if (levelUp >= …)`
	`  … levelUp …`

`ristourne = .1 *`
`tabClient[i].valFacture;`
`if (tabClient[i].valFacture > 1000)`
`  ristourne = .2 *`
`tabClient[i].valFacture;`
`printf(… tabClient[i].valFacture …);`
	`double montant =`
	`tabClient[i].valFacture;`
	`ristourne = .1 * montant;`
	`if (montant > 1000)`
	`  ristourne = .2 * montant;`
	`printf(… montant …);`

4. Gestion dynamique de la mémoire

Préférez une structure dynamique quand vous ne connaissez pas le nombre d'éléments à priori excepté le cas où des variables de type statique suffisent. Exemple : le programme a besoin de 5 entiers.

En sortant, remettez la mémoire dans l'état où vous l'avez trouvée. Assurez-vous que votre programme libère bien tout l'espace mémoire alloué de manière dynamique avant de se terminer.

En langage C, seulement si nécessaire, si une variable automatique (c'est-à-dire une variable locale « normale ») suffit, inutile de la remplacer par un pointeur dont vous gérez l'allocation de mémoire explicitement !

5. Instructions

Une instruction par ligne

N'écrivez pas plusieurs instructions à la suite l'une de l'autre sur la même ligne : une ligne par instruction. Cela permet non seulement d'ajouter plus facilement un commentaire si nécessaire mais cela garde également le code plus lisible !

Écriture des conditions (if, while, for)

Lorsque c'est possible, évitez les conditions négatives (qui sont généralement plus difficiles à lire). Par exemple, réécrivez la condition `( !(tabNombres[i] > 10) )` en `(tabNombres[i] <= 10)`.

Accolades facultatives dans les if/while/for

Si le corps d'une conditionnelle ou d'une répétitive ne comporte qu'une seule instruction, celle-ci ne doit pas forcément être encadrée d'accolades. Malgré tout, la plupart des auteurs conseillent d'écrire ces accolades : cela permet d'éviter les erreurs lorsqu'on ajoute d'autres instructions par la suite.

D'un autre côté, si vous ne voulez absolument pas mettre d'accolades inutiles, choisissez alors d'écrire l'instruction sur la même ligne que la condition :

```
if (code == codeRecherche) nombre++;
```

Affectations embarquées

Les affectations embarquées (++ et --) sont généralement à éviter, sauf dans les cas les plus simples (comme `i++` ou `nbJoursRestants--`).
Par exemple,

```
while (indiceDebut++ <= -- indiceFin) {...}
```

est peut-être plus difficile à décrypter que la répétitive suivante :

```
while ( indiceDebut <= indiceFin) {
            indiceDebut ++;
            indiceFin--;
            ...
 }
```

Utilisation des for

Évitez les répétitives « for » si celles-ci deviennent trop complexes à lire. C'est généralement le cas quand on tente de placer trop de choses (trop de conditions complexes ou trop d'actions) dans le `for`. Dans tous ces cas, employez plutôt un `while`.

Quelques cas où le « for » est conseillé :

- répéter quelque chose *n* fois : `for (i = 0 ; i < n ; i++)`
- parcourir un tableau : `for (i = 0 ; i < TAILLE ; i++)`
- parcourir une chaîne de caractères, une liste chaînée, une collection, ...

Écriture des instructions switch

Si vous avez un cas « default », placez-le en dernière position. N'oubliez pas les `break` nécessaires à la fin de chaque cas. Dans la foulée, mettez-en également un à la fin du dernier cas (sauf éventuellement s'il s'agit de « default ») ; cela permettra d'éviter les erreurs si vous changez l'ordre des cas existants ou si vous en ajoutez de nouveaux par la suite. Si vous omettez intentionnellement un `break` (pour que l'exécution continue dans le cas suivant), ajoutez un commentaire pour le signaler explicitement.

Si vous voulez déclarer une variable locale pour un des cas du `switch`, encadrez simplement le bloc d'instructions correspondant d'accolades.

Préférez un `switch` à de nombreux `if`/`else` quand c'est possible. Dans ce cas-là, considérez aussi l'utilisation d'un type énuméré[2] pour obtenir des conditions plus parlantes (voir l'exemple suivant).

… devient …	

`if (codeErreur == 0)`
`  printf("Pas d'erreur");`
`else if (codeErreur == 1)`
`  printf("Erreur : prix négatif !");`
`else if (codeErreur == 2)`
`  printf("Erreur : quantité négative !");`
`else`
`  printf("Code erreur inconnu.");`
	`typedef enum erreur Erreur;`
	`enum erreur {`
	`PASDERREUR, PRIXNEGATIF, QTNEGATIVE`
	`};`
	
	`Erreur codeErreur;`
	`...`
	`switch (codeErreur)`
	`{`
	`  case PASDERREUR :`
	`  printf("Pas d'erreur");`
	`  break;`
	`  case PRIXNEGATIF :`
	`  printf("Erreur : prix négatif !");`
	`  break;`
	`  case QTNEGATIVE :`
	`  printf("Erreur : qté négative !");`
	`  break;`

[2] Ou de constantes symboliques éventuellement.

	default :
	printf("Code erreur inconnu.");
	}

if (prix < 0)
return 1;
if (quantite < 0)
return 2;
prixTotal = prix * quantite;
return 0;
	if (prix < 0)
	return **PRIXNEGATIF**;
	if (quantite < 0)
	return **QTNEGATIVE**;
	prixTotal = prix * quantite;
	return **PASDERREUR**;

6. Fonctions

Découpe en fonctions/méthodes

Chaque fonction/méthode doit correspondre à une tâche facilement identifiable (et identifiée par le nom). En tant que programmeur, vous devez être capable de décrire précisément ce que fait une fonction ou une méthode par une phrase simple. Si ce n'est pas possible, l'organisation doit sans doute être revue. Même si un bout de code n'est utilisé qu'une seule fois, l'organiser sous la forme d'une fonction ou méthode rend le programme plus lisible : au minimum, celui permettra de lui donner un « titre » explicite.
Exemples :

... imprimerEnteteFacture()
... imprimerLignesArticle(...);
... imprimerTotalFacture(...);
	...deplacerPion(...)
	if (estSurCaseOccupee(...))
	{
	...prendrePion();
	}

Longueur des fonctions/méthodes

Une fonction/une méthode ne devrait jamais faire plus d'une trentaine de lignes (un écran). Si une fonction/méthode est plus longue, il serait sans doute bon de la diviser en plusieurs fonctions/méthodes.

Copier/coller ? Pas dans mon code

Ne dupliquez jamais du code (copier/coller est votre ennemi !) Si un même code est utilisé à deux endroits, faites-en un module / une fonction /une méthode !

Organisation des fonctions/méthodes

Au sein d'un module, regroupez les fonctions/méthodes par thème (une fois encore, pour faciliter la lecture et la compréhension du code). Dans chaque section, ordonnez-les de la plus générale (celle qui utilise les autres) à la plus particulière (qui est appelée par les autres). Ainsi,

en lisant le code du début à la fin, on rencontre d'abord les fonctions de haut niveau avant de s'enfoncer de plus en plus loin dans les détails.

Écriture des prototypes de fonctions ou les signatures des méthodes

En C, les prototypes des fonctions peuvent être écrits sans donner de noms aux arguments (le mot-clef est « peuvent » : ce n'est pas une obligation). Ceci dit, lorsque vous présentez le prototype d'une fonction, y inclure des arguments dont les noms sont significatifs peuvent grandement améliorer la lisibilité du code.
N'hésitez donc pas à nommer les arguments, même si la syntaxe du C ne l'exige pas. Par exemple, le prototype

```
void copieChaine (char *, char *);
```

nécessite l'ajout d'un commentaire pour expliquer le rôle que joue chacun des arguments, alors que

```
void copieChaine (char * destination, char * source);
```

ne demande aucune explication supplémentaire.

7. Définitions de types

Utilisation des alias / synonymes

Créez un alias / synonyme (via `typedef`) pour chaque type « struct » et « union ». Et, tant qu'à faire, placer le `typedef` avant la définition du type ; cela vous permettra d'utiliser l'alias dans la définition du type (voir exemple ci-dessous). Dans le cas de fichiers d'en-tête, cela vous permettra aussi de rendre l'alias publique mais de garder la définition du type privée (voir la seconde partie de ce document).

… devient …	

`typedef struct cellule`
`{`
`  int info;`
`  struct cellule * pSuivant;`
`} Cellule;`
	`typedef struct cellule`
	`Cellule;`
	
	`struct cellule`
	`{`
	`  int info;`
	`  Cellule * pSuivant;`
	`};`

Séparation des définitions de types et déclarations de variables

Le C permet de déclarer une variable tout en définissant un type. Pour plus de clarté et pour mieux organiser votre code, ne le faites pas (voir exemple ci-dessous).

… devient …	

`typedef struct artiste Artiste;`
 	
`struct artiste`
`{`
`  char nom[30];`
`  int age;`
`  char chansons[100][30];`
`} monArtistePrefere;`
	`typedef struct artiste Artiste;`
	
	`struct artiste`
	`{`
	`  char nom[30];`
	`  int age;`
	`  char chansons[100][30];`
	`};`
	
	`Artiste monArtistePrefere;`

Déclarations des attributs d'un type structuré

Déclarez un attribut par ligne : c'est plus clair, plus facile à éditer, et ça permet d'ajouter éventuellement des descriptions (les raisons sont similaires à celles qui justifient qu'on ne déclare qu'une seule variable locale par ligne).
Organisez les déclarations des attributs dans un ordre bien pensé. Entre autres, regroupez les attributs qui sont apparentés (par exemple, dans le cas d'une cellule d'une liste chaînée ou d'un arbre, toutes les informations ensemble puis tous les pointeurs de chaînage).

8. Commandes du préprocesseur

Constantes numériques

Conformément au principe de « lieu unique de modification », définissez une constante symbolique pour toutes les valeurs numériques qui pourraient être modifiées dans le futur. Par exemple, pour la taille d'un tableau, n'utilisez pas une valeur numérique dans votre code mais définissez plutôt une constante symbolique comme NB_MAX_ARTISTES.

Par contre, si vous devez effectuer un calcul du style `gainAnnuel = gainMensuel * 12` ou encore `surface = 2 * PI * rayon`, il n'est pas obligatoire de définir une constante symbolique pour le nombre de mois dans une année ou la constante 2 dans la formule de la surface (il est très peu probable que ces valeurs changent dans le futur).

Ceci dit, même si une valeur ne risque pas de changer, il peut être utile de lui donner un nom (via une constante symbolique ou une constante locale) si cela permet d'exprimer de manière plus claire ce qu'elle représente. Par exemple : `largeur = 16./9. * hauteur` gagnerait en clarté si on définissait `#define FORMAT_ECRAN = 16./9.` pour expliciter de quoi il s'agit.

Définitions de constantes symboliques

La directive `#define` (en langage C) n'est pas la seule manière de définir des constantes symboliques.

Deux autres options : définir un type énuméré (mieux pour des groupes de constantes) ou encore définir une constante avec le mot réservé `const` (pour des constantes à utilité plus locale par exemple).

Prudence dans l'écriture des macros fonctionnelles

Encadrez chaque variable par des parenthèses pour éviter les mauvaises interprétations. Si la macro est une expression, encadrez le tout avec des parenthèses. Évitez les macros trop complexes (pensez plutôt à les remplacer par des fonctions dans ce cas-là).

9. Fichiers sources (attention : nous parlons du langage C)

Structure des fichiers sources

Organisez tous les fichiers selon la même structure. Par exemple :

- `#include` pour les bibliothèques système (c'est-à-dire les `#include` avec < >);
- `#include` pour les bibliothèques locales (c'est-à-dire les `#include` avec " ");
- définitions de constantes symboliques;
- définitions de types;
- définitions des fonctions ou prototypes.

Dans le cas du fichier / programme principal, le point (5) est généralement décomposé en :

- prototypes des fonctions propres au programme principal;
- définition de la fonction « main »;
- définitions des fonctions propres au programme principal.

Fichiers d'en-tête

Dans les fichiers d'en-tête, on ne retrouve que les points de (1) à (4), vu que les définitions des fonctions se trouvent dans les fichiers .c associés (voir seconde partie de ce document). Les fichiers d'en-tête ne devraient contenir que les définitions de constantes, les définitions d'alias / synonymes, les définitions de types et les prototypes de fonctions qui doivent être visibles de l'extérieur, pas les fonctions auxiliaires « privées ».

10. Opérateur sizeof

Soient les instructions

```
int * p ;
...
p = (int *) malloc (sizeof (*p)) ;
```

Utilisez sizeof (variable) plutôt que sizeof (type) car le lieu de modification sera unique en cas de changement de type de variable.

11. Instructions break et continue

Ces instructions sont à utiliser avec parcimonie. Le cours théorique a montré leur utilité.
A savoir que le break dans une boucle permet d'arrêter cette répétitive dans des cas bien précis au lieu de placer un test d'arrêt plus complet et une alternative.

L'exemple suivant présente une boucle destinée à stocker des entiers lus dans un tableau en arrêtant la lecture dès que la valeur 0 est rencontrée.

<table>
<tr>
<td>

```
for (i = 0;
    i < 100 && (i >= 1 && tableau[i-1]);
    i++)
{
          ...
          scanf ("%d",&tableau[i]);
          if ( tableau[i] != 0)
          {
                    ...
          }
}
```

</td>
<td>

```
i = 0;
do
{
    scanf("%d", &tableau[i]);
    i++;
}
while (i < 100 &&
         tableau[i-1] != 0);
```

</td>
</tr>
<tr>
<td>devient</td>
<td>

```
for (i = 0 ; i < 100 ; i++)
{
  ...
   scanf("%d",&tableau[i]);
   if ( tableau[i] == 0)
           break;
   ...
}
```

</td>
</tr>
</table>

L'instruction continue permet de rendre plus lisible le fait d'éviter une partie du traitement.

```
for (i = 0; i < 100 ; i++)
{
   ...
   if ( tableau[i] > 0)
   {
      ...
   }
}
```

```
for (i = 0; i < 100; i++)
{
          ...
          if ( tableau[i] <= 0) continue;
        // traitement en cas d'élément positif
          ...
}
```

Partie 2 : Modularité en C

Un programme C peut s'étendre sur plus d'un fichier (c'est ce qu'on nomme la programmation multi-fichiers). Un des fichiers contiendra le code du programme principal (la fonction main) alors que les autres, appelés « modules », contiendront la définition des diverses fonctions utilisées (directement ou indirectement) par le programme principal. C'est au sein de ces modules que se trouvera la quasi-totalité du code, le programme principal se réduisant alors à peu de choses près à une série d'appels de fonctions.

Note. En C, le mot « module » désigne généralement un fichier annexe (une bibliothèque) regroupant une série de définitions, de fonctions et autres. Par exemple, `string.h` et `ctype.h` sont des modules (ou bibliothèques).

Dans une découpe en modules, les fonctions sont regroupées par « thème ». On pourrait par exemple avoir un module reprenant toutes les fonctions (et définitions de types) liées à la gestion d'erreurs, un module gérant tout l'aspect interface avec l'utilisateur et un troisième module regroupant les fonctions plus techniques (travail sur les chaînons d'une liste par exemple).

Pourquoi programmer de façon modulaire ?

Cette découpe en modules a de nombreux avantages et objectifs. En voici quelques-uns.

(1) Un code mieux découpé pour une écriture facilitée.

Un programmeur (ou gestionnaire de projet) décide de la manière dont l'application sera découpée en modules. Une fois cette découpe effectuée, le programmeur peut se concentrer sur l'écriture de chacun des modules les uns après les autres (des tâches plus petites / plus ciblées / plus faciles).
La découpe se traduit en un « plan de travail » pour le programmeur.
Cette découpe est également très utile lorsqu'il s'agit de répartir le travail entre plusieurs programmeurs.

(2) Un code mieux organisé pour une lecture et une maintenance facilitées.

Comme le code de l'application est découpé en plusieurs parties associées chacune à un aspect différent, sa lecture se retrouve facilitée :

- pour vérifier que le code est correct, on peut examiner les modules un par un et les relire morceau par morceau ;
- pour comprendre le code, on peut se servir de la découpe en modules comme d'une table des matières donnant une vue d'ensemble du projet ;

- pour retrouver un bug ou modifier une fonctionnalité du programme, la découpe en modules permet de cibler plus aisément la partie de code à laquelle il faut s'intéresser.

Si un module concerne l'entrée des données via le clavier, si l'utilisateur veut modifier cet encodage (le graphisme par exemple), un seul module est à changer.
Si un module a pour but de sauver/charger les données, en cas de changement de structures de stockage, de nouveau, moins de code est à mettre à jour.

(3) Des bouts de code plus faciles à réutiliser.

La découpe en modules permet de construire des parties de code indépendantes les unes des autres (techniquement, c'est ce qu'on appelle des parties de code « peu couplées » ou « à couplage faible »).
Par exemple, dans le cas d'une application de jeu utilisant deux modules, un module s'occupant de l'interface graphique et un second module gérant tous les calculs propres au jeu, c'est le programme principal qui fera le « lien » entre les deux modules. Ce programme principal utilisera le module de calcul pour connaître la position du personnage et des ennemis et communiquera ces informations au module d'interface graphique qui se chargera d'afficher ce qu'on lui demande. Le module d'interface graphique, s'il est bien construit[3], pourra facilement être réutilisé pour gérer l'affichage dans un autre jeu, avec un personnage, des ennemis et des règles complètement différents.

(4) Un code source découpé en parties pour des (re)compilations plus efficaces.

Comme le code source est réparti en plusieurs modules qui peuvent être compilés indépendamment les uns des autres, lorsqu'un de ces modules est modifié, il suffit de recompiler la partie concernée.
Si le code n'avait pas été divisé en modules, il aurait fallu recompiler l'ensemble à chaque modification.
Chaque environnement de développement se charge automatiquement d'identifier, à chaque lancement du programme, quelles parties ont été modifiées et doivent donc être recompilées.

[3] C'est-à-dire s'il est bien faiblement couplé au module de calcul.

Programmation modulaire en C

En C, chaque module se compose de deux fichiers :

- un fichier de code (fichier .c) qui contient la définition des fonctions (et autres),
- un fichier d'en-tête (fichier .h pour « header ») qui ne reprend que la partie publique, c'est-à-dire ce que le « monde extérieur » (le programme principal et les autres bouts de code qui vont utiliser le module) doit connaître.

Cette distinction entre l'aspect code et l'aspect en-tête est assez similaire à celle qu'on trouve dans un langage orienté objet Java ou C#, où certaines méthodes sont publiques (le « monde extérieur » peut y faire appel) mais où les attributs d'instance et l'implémentation de certaines méthodes sont, eux, privés (phénomène d'encapsulation).

L'idée maîtresse est que le fichier d'en-tête .h doit donner au « monde extérieur » suffisamment d'informations pour qu'il soit possible d'utiliser le module, mais aucune information superflue. Entre autres, le fichier .h doit indiquer **ce que le module peut faire, mais pas comment il le fait**.

Pratiquement, pour plus de clarté dans la découpe du code, les fichiers .c et .h d'un même modèle porteront toujours le même nom (seule l'extension changera). Ainsi, un module d'interface graphique sera codé dans `interface.c` et aura pour fichier d'en-tête `interface.h`.

Si un module extérieur (ou le programme principal) a besoin d'utiliser un module donné (par exemple le module d'interface graphique), il suffira d'ajouter une commande `#include` dans le module appelant. Dans le cas de cet exemple, il s'agira de `#include "interface.h"`. Grâce à cette ligne, le module appelant connaîtra (entre autres) les prototypes de toutes les fonctions que le module interface met à sa disposition et vous permettra d'y faire appel.

interface.h	interface.c

`void afficheMonstre (int coordX, int coordY, int typeMonstre);`
 	
`void afficheScore (int score);`
	`#include "interface.h"`
	
	`void afficheMonstre (int coordX, int coordY, int type)`
	`{ … code … }`
	
	`void afficheScore (int score)`
	`{ … code … }`

Dans le fichier principal (et/ou les autres modules qui veulent utiliser ces fonctions), il suffira d'ajouter la ligne `#include "interface.h"` pour que le prototype des fonctions `afficheMonstre` et `afficheScore` soient connus. Grâce à cela, on pourra faire appel à ces fonctions dans le code du fichier principal.	

Que trouve-t-on dans un fichier d'en-tête ?

Globalement, le fichier d'en-tête devra contenir toutes les informations dont le module appelant pourrait avoir besoin. Cela signifie entre autres :

- les prototypes des fonctions mises à disposition ;
- les définitions de types qui doivent être connues de l'extérieur (il peut s'agir de définitions de structures si le monde extérieur doit pouvoir accéder au contenu ou encore de définitions d'alias / synonymes via `typedef`) ;
- les définitions des constantes symboliques (et/ou types énumérés) qui doivent être connues du monde extérieur ;
- les définitions de macros qui doivent être connues du monde extérieur ;
- et, éventuellement, les #include qui sont nécessaires à tout ce qui est indiqué ci-dessus.

Note. Dans les projets utilisant des variables globales, celles-ci peuvent également être déclarées dans les fichiers .h (ceci dit, votre dossier de programmation ne doit pas utiliser de telles variables).

Deux exemples et quelques remarques

erreur.h	commande.h

`typedef enum erreur Erreur;`
`enum erreur`
`{ PASDERREUR, PRIXNEG,`
`QUANTITENEG };`
 	
`void afficheErreur(int erreur);`
`int estErreur(int code);`
	`#include "erreur.h"`
	
	`typedef struct commande Commande;`
	
	`Erreur ajouteCommande(int num, int prix, int qte);`
	`void afficheCommande(Commande);`
	`Commande rechercheCommande(int num);`

Ce module définit des constantes symboliques (type énuméré) pour les types d'erreurs ainsi que deux fonctions : `afficheErreur` affiche un message d'erreur dont le code est donné et `estErreur` indique si le code donné correspond à une erreur. L'implémentation erreurs.c utilisera certainement stdio.h (pour afficher un message d'erreur) mais erreur.h ne contient pas #include <stdio.h> car seule la partie implémentation (erreurs.c) en a besoin.	Ce module permet de gérer des commandes caractérisées par un numéro, un prix unitaire et une quantité. Il propose trois fonctions : ajouter une commande, afficher une commande et rechercher une commande dont le numéro est connu. commande.h inclut erreurs.h car le type Erreur est utilisé dans le prototype de certaines fonctions.

Dans l'exemple ci-dessus, notez que le fichier erreurs.h définit explicitement les constantes du type Erreur (= `enum erreur`). Cela signifie que le « monde extérieur » peut utiliser les constantes

PASDERREUR, PRIXNEG et QUANTITENEG explicitement. C'est nécessaire, entre autres, à la fonction ajouteCommande de commandes.h vu que celle-ci doit renvoyer un code d'erreur.

Par contre, commande.h ne définit pas explicitement le type Commande (= struct commande). C'est parce que le « monde extérieur » n'a pas besoin de connaître les détails de ce type, qui est défini de manière « privée » à l'intérieur de commande.c.
Dans commande.c, on pourrait par exemple trouver l'une des définitions suivantes (selon que les commandes sont stockées dans un tableau, un fichier, une liste chaînée ou un arbre par exemple). Les choix d'implémentation seront par contre invisibles au monde extérieur : le programme principal ne pourra pas utiliser d'expressions du type commande.numero vu qu'il ne sera pas informé de la manière dont le type est défini.

Exemples de définitions du type struct commande dans commande.c

```
struct commande
{
  int numero;
  int prix;
  int qtte;
};
```

```
struct commande
{
  int numero;
  int prix;
  int qtte;
  Commande * pSuivante;
};
```

```
struct commande
{
  int numero;
  int prix;
  int qtte;
  Commande * pSAG;
  Commande * pSAD;
};
```

Notez que, dans la définition du type struct commande, on peut utiliser directement l'alias / synonyme Commande car commande.c comportera un #include "commande.h" et donc, reprendra automatiquement la ligne typedef struct commande Commande de commande.h.

Comme certains prototypes de commandes.h utilisent le type Erreur, le fichier d'en-tête contient un #include "erreur.h". Cette ligne est nécessaire car, sans elle, on court le risque qu'un module appelant ne connaisse pas la définition du type Erreur.

Par contre, en incluant la ligne #include "erreur.h" dans commande.h, on encourt un autre risque.
Soit le diagramme ci-dessous :

erreur.h	commande.h	commande.c
...	#include "erreur.h"	#include "commande.h" #include "erreur.h"

Risque de double définition :

- la première fois, quand l'appel de commande.h sera effectué : on lira déjà une première fois erreur.h à cause de l'include qui se trouve dans commande.h

- une deuxième fois, par l'appel de erreur.h dans commande.c.

Pour éviter cela, sont ajoutées des « sentinelles » ou « gardes » (en gras dans l'exemple ci-dessous ; le texte en italique correspond au contenu du fichier d'en-tête), qui permettent de s'assurer qu'on ne va tenir compte des définitions de types et de prototypes qu'une seule fois.

#ifndef ERREUR_CHARGE **#define ERREUR_CHARGE** *enum erreur* *{ PASDERREUR,* *PRIXNEG,* *QUANTITENEG };* *typedef enum erreur Erreur;* *void afficheErreur(int erreur);* *int estErreur(int erreur);* **#endif**	La directive #ifndef agit comme une instruction « if » : • « Si la constante symbolique ERREUR_CHARGE n'est pas définie, on la définit (pour indiquer qu'on a déjà pris en compte les définitions de erreurs.h) et on inclut ces définitions. » • « Si, par contre, la constante symbolique ERREUR_CHARGE a déjà été définie (c'est-à-dire si on a déjà lu les définitions de erreurs.h), on passe directement au #endif. »

#ifndef ERREUR_H **#define ERREUR_H** *enum erreur* *{ PASDERREUR,* *PRIXNEG,* *QUANTITENEG };* *typedef enum erreur Erreur;* *void afficheErreur(int erreur);* *int estErreur(int erreur);* **#endif**	Autre manière de procéder