Un livre de Wikilivres.

Conseils de codage en C

Une version à jour et éditable de ce livre est disponible sur Wikilivres, une bibliothèque de livres pédagogiques, à l'URL : http://fr.wikibooks.org/wiki/Conseils_de_codage_en_C

==

==

==

Lisibilité des sources

Les conseils suivants rendent les logiciels plus maintenables.

Leur respect a pour but de :

- faciliter la lecture des codes sources écrits en C
- diminuer l'effort nécessaire pour diagnostiquer les déficiences ou causes de défaillance, ou pour identifier les parties à modifier.

Cartouche d'entête (c_lis_1)

Chaque fichier source (.c, .h) et chaque fonction C doivent être précédés

d’un cartouche qui pourra contenir :

- Le nom de l'unité ou de la fonction.
- Son rôle et ses points de programmation critiques.
- Pour une fonction, la description des arguments et des codes retours.
- La date, l’auteur et le rôle de chaque modification.
- Sa licence et droits d'utilisation
- La mention de documents de conception ou de référence

Justification

Le cartouche de l’unité permet de faciliter sa compréhension d'un point de vue général. Les cartouches de fonctions permettent de comprendre leur rôle et conditions d'appel sans se plonger dans le code source . Ces entêtes sont indispensables à la gestion de la configuration et à la maintenance.

Exemple

```
/*****************************************************
Nom ......... : limites.c
Role ........ : Affiche les caractéristiques numériques
                pour les types du langage C
Auteur ...... : Thierry46
Version ..... : V1.1 du 10/3/2008
Licence ..... : GPL

Compilation :
gcc -Wall -pedantic -std=c99 -o limites.exe limites.c
Pour exécuter, tapez : ./limites.exe
*****************************************************/
```

Outils

- La plupart des environnements de développement bâtissent un squelette de cartouche d'entête lors de la création d'un nouveau fichier
- Pour les projets importants, développés en équipe, les versions des fichiers sources pourront être gérées par un logiciel spécialisé comme CVS, subversion...
- L'utilisation d'un outil de gestion de la documentation comme Doxygen permet, à l'aide d'une syntaxe particulière dans les commentaires, de générer à partir des sources la documentation de programmation du projet.

Indentation des instructions (c_lis_2)

Les instructions imbriquées sont indentées à l’aide de tabulations ou d'espaces de façon homogène dans tous les fichiers sources du projet.

Justification

La maintenance et la relecture des sources sont facilités par une présentation homogène.

Exemple (extrait)

```
        //...
        // Teste si les valeurs lues sont les valeurs attendues.
        for (i=0; i<(int)nbInt; i++)
        {
                (void)printf("%d, ", valIntLu[i]);
                if ( valIntLu[i] != valInt[i] )
                {
                        (void)printf("Au lieu de %d\n", valInt[i]);
                        exit(EXIT_FAILURE);
                }
        }
```

Outils

- L'outil UNIX **indent** permet d'améliorer la présentation des fichiers sources écrits en C. De nombreuses options permettent de paramètrer le style de ses sorties. Elles peuvent être passées sur la ligne de commande ou mieux dans un fichier de configuration spécifique.

Indentation des commentaires (c_lis_3)

Les commentaires suivent l'indentation des instructions de code.

Justification

Augmente la lisibilité.

Une instruction par ligne (c_lis_4)

Ne pas écrire plus d'une instruction par ligne.

Justification

Augmente la lisibilité.

Décomposer les instructions longues (c_lis_5)

Coder des lignes simples.

Justification

Il est quelquefois tentant de coder des lignes longues :

- Dans le domaine numérique, le programmeur écrit des formules compliquée juxtaposant de nombreuses fonctions et opérateurs.
- Le langage C permet de mélanger tests, appel de fonctions, opérations.

Le code parait, à première vue, plus compact, plus efficace. Il faut cependant décomposer les lignes longues en plusieurs, en utilisant des variables intermédiaires. Les compilateurs récents se chargeront d'optimiser efficacement le code pour vous. Le résultat sera plus facile à comprendre par d'autres. Les problèmes de priorité pourraient être atténués.

Constantes symboliques et macros en majuscule (c_lis_6)

Les noms des constantes symboliques et des macros instructions doivent être écrits en majuscule.

Justification

Dans un fichier source, permet des distinguer rapidement les constantes des variables.

Exemple

```
#define LONGUEUR_NOM_FICHIER 15
#define STR(s) #s
#define XSTR(s) STR(s)
```

Pas de lignes trop longues (c_lis_7)

Les instructions ne doivent pas dépasser la colonne 80. Les instructions longues doivent être placées sur plusieurs lignes.

Justification

Les lignes courtes seront plus faciles à comprendre par une personnes chargée de maintenir le logiciel.

Exemples

Mauvais :

```
#define sonne(N) { int n; for (n=N; n>0; n--) fputc(0x7, stderr);}

   (void)printf("\nProgramme %s, fichier source "__FILE__"\nCompile le "__DATE__" a "__TI
```

Meilleur :

```
#define sonne(N) \
{ \
   int n; \
   for (n=N; n>0; n--) fputc(0x7, stderr); \
}
//...
   (void)printf("\nProgramme %s, fichier source "__FILE__
      "\nCompile le "__DATE__" a "__TIME__"\n",
      argv[0]);
```

Nommage des identificateurs (c_lis_8)

Vous devez donner des noms significatifs aux identificateurs.

- Évitez les noms trop courts sans signification fonctionnelle : v, vv, vvv...
- Ne différenciez pas deux identificateurs uniquement en changeant la casse de certains caractères : Fichier, fichier.

- Ne différenciez pas deux identificateurs uniquement par un nombre trop restreint de lettres.
- N'utilisez pas de caractère souligné en tête ou en fin d'identificateur.
- Utiliser une règle de nommage cohérente dans le projet : mot séparés par des soulignés (fichier_parametres_calcul) ou mots collés commençant par des majuscules sauf première lettre (fichierParametresCalcul).

Justification

Facilite la lisibilité et la maintenabilité.

Cohérence des identificateurs (c_lis_9)

Dès lors qu'un identificateur de variable est défini pour une entité significative, il faut utiliser ce même identificateur quelle que soit l'unité de code considérée.

Justification

Facilite l'analyse du programme.

Arguments du programme principal (c_lis_10)

Les arguments du programme principal sont standards : `int main(int argc, char *argv[])`

Justification

Améliore la lisibilité du code. argv[0] représente le nom de lancement du programme et peut être utilisé pour les messages d'erreur. La récupération des arguments du programme pourra se faire avec les fonctions getopt(), getsubopt().

Limiter l'utilisation des opérateurs ++ et -- (c_lis_11)

Les opérateurs ++ et -- ne sont autorisés que pour les indices de boucles et les pointeurs, à condition qu'aucun autre opérateur n'apparaisse et que seule l'utilisation post-fixée soit utilisée.

Justification

Facilite l'analyse et la maintenance par des programmeurs habitués à d'autres langages.

Exemple

- Correct : `for (i=0; i<n; i++) {//....`
- Moins bon : `t[i++]=f(i++);`

Limiter l'utilisation de l'opérateur de test ternaire (c_lis_12)

L'utilisation de l'expression conditionnelle ? : est interdite en dehors des macros.

Justification

Facilite l'analyse et la maintenance par des programmeurs habitués à d'autres langages.

Ne pas économiser les accolades (c_lis_13)

Utilisez des accolades {} autour d'une ou des instructions d'un bloc. Ce bloc peut faire partie d'une structure de contrôle comme **if** - **else** - **for** - **do** - **while**.

Justification

Rend le code plus lisible et évite les erreurs lors de l'ajout d'une instruction dans un bloc qui n'en contient qu'une seule.

Exemple

Mauvais :

```
for (i=0; i<n; i++)
   table[i] = 1;
```

Meilleur :

```
for (i=0; i<n; i++)
{
   table[i] = 1;
}
```

Exemple d'erreur provoquée par la violation de ce conseil :

```
if (test == 2)
{
   for (i=0; i<n; i++)
      table[i] = 1;
      table2[i] = 2;
   }

/* Suite... */
```

L'instruction `table2[i] = 2;` donne faussement l'impression de faire partie de la boucle **for**, ce qui est renforcé par l'indentation et l'accolade fermante du **if**.

Si table2 était dimensionné à n, les indices acceptables iraient de 0 à n-1 inclus. Lors de l'exécution de l'instruction `table2[i] = 2;`, i vaudrait la valeur n, ce qui provoquerait l'écriture d'une valeur hors de l'espace mémoire réservé pour le tableau table2.

Il s'en suivra :

- Le débordement hors de l'espace utilisateur qui provoquera une violation mémoire : Sanction rapide par arrêt du programme exception SIGSEGV.
- L'écrasement du contenu d'un autre pointeur situé juste après l'espace table2 qui lorsqu'il sera utilisé plus tard (100 lignes plus bas ?) provoquera une autre catastrophe : Cette erreur peut être difficile à détecter sans l'utilisation d'un outil qualité d'analyse dynamique.
- Ces erreurs peuvent arriver aléatoirement selon le système, le compilateur utilisé ou le chargement du

programme en mémoire.

Bien aligner les accolades (c_lis_14)

Une accolade fermante se trouve toujours à la verticale de l'accolade ouvrante correspondante.

Justification

Permet de mieux repérer les blocs d'instructions conditionnelles, les corps de boucles surtout en cas de structures imbriquées.

Exemple

```c
// Compilation : gcc -Wall -pedantic -std=c99 -o essai.exe essai.c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   const int nbLigne = 3;
   const int nbCol = 4;

   int table[nbLigne][nbCol];
   for(int ligne = 0; ligne < nbLigne; ligne++)
   {
      for(int col = 0; col < nbCol; col++)
      {
         table[ligne][col] = ligne + col;
      }
   }

   // Impression
   for(int ligne = 0; ligne < nbLigne; ligne++)
   {
      for(int col = 0; col < nbCol; col++)
      {
         (void)printf("table[%d][%d] = %d,",
                      ligne, col, table[ligne][col]);
      }
      (void)puts("");
   }
   return EXIT_SUCCESS;
}
```

Éviter d'utiliser les opérateurs d'affectation spécifique au C (c_lis_15)

L'utilisation de l'assignation composée (`+=, -=, *=, %=, /=`) et de l'affectation multiple (`v1 = v2 = v3;`) est déconseillée.

Justification

Facilite l'analyse et la maintenance par des programmeurs habitués à d'autres langages.

Ne pas déclarer plusieurs variables dans la même instruction

(c_lis_16)

Il faut déclarer chaque variable séparément et non les unes à la suite des autres, séparées par des virgules.

Justification

La déclaration de plusieurs variables dans une même instruction peut provoquer des erreurs de type pour les variables.

Une déclaration séparée permet aussi de décrire dans un commentaire en bout de ligne le rôle de chaque variable.

Exemple

```
// Declaration incorrecte :
// nomFicSortie est de type char au lieu de char * et ne peut donc être initialisé à NULL
char *nomFicEntree = NULL, nomFicSortie = NULL;

// Declaration correcte mais pas lisible
char *nomFicEntree = NULL, *nomFicSortie = NULL;

// Meilleur
char *nomFicEntree = NULL; // Fichier de parametres du calcul
char *nomFicSortie = NULL; // Fichier pour les points calcules
```

Seuils pour les métriques déterminant la facilité d'analyse (c_lis_17)

Les métriques suivantes influent sur la facilité d'analyse d'un programme (ISO/IEC 9126) :

- VG : nombre cyclomatique : nombre de chemins linéairement indépendants dans un graphe connexe g, V(g) = A - N + 1 avec A : nombre de graphe entre les nœuds du graphe et N : nombre de nœuds du graphe.
- STMT : Nombre d'instructions exécutables entre les accolades de début et de fin de la fonction.
- FCOM : Fréquence des commentaires : F_COM = (BCOM+BCOB) / STMT, avec BCOM = Nombre de blocs de commentaires dans la fonction, BCOB : nombre de blocs de commentaires avant la fonction.
- AVGS : Taille moyenne des instructions, calculée à partir du nombre d'opérateur et d'opérande distincts.

Seuils pour les métriques :

- VG : de 1 à 20
- STMT : de 1 à 100
- FCOM : de 0,2 à 1,2
- AVGS : de 2 à 10

Justification

- VG trop élevée montre qu'une fonction est trop complexe et mériterait d'être décomposée en plusieurs ou réanalysée.
- STMT trop élevée indique que la fonction est trop longue. Il faudrait la diviser ou peut-être appeler des fonctions.
- FCOM : Habituellement les programmes ne sont pas assez commenté : FCOM faible. Cependant un programme avec trop de commentaires peut aussi montrer des problèmes (trop de ruses d'un expert qui

aurait dû écrire plus simplement ???).
- AVGS : Les instructions trop longues sont difficiles à comprendre et peuvent provoquer des problèmes de priorité des opérateurs, de conversion de type involontaires.

Recherche des erreurs

L'application de ces conseils facilite la recherche des erreurs et permet d'en éviter certaines.

Des identificateurs plus parlants (c_rec_1)

Utilisez des identificateurs parlants pour les variables. Le nom d'une variable devra rappeler son rôle ou son utilité.

Justification

Des noms de variable trop courts et souvent sans signification (a, n, x...) ne permettent pas aux relecteurs de faire la relation entre l'identificateur informatique et la réalité représentée. La recherche des variables est alors difficile et les risques d'avoir deux identificateurs dont les portées se superposent sont augmentés.

Exemple

Pour désigner un numéro de maille.

- Mauvais : n
- Bon : numMaille

Sortie de boucle (c_rec_2)

La sortie d'une boucle (do, while, for) doit se faire par la condition de test.

Justification

Un algorithme bien conçu ne doit pas nécessiter de sortie prématurée dans le corps de boucle. La programmation structurée permet d'éviter le style spaghetti des premiers programmeurs. Ce mode de programmation ancien rendait le suivi du déroulement difficile à l'aide d'un débugueur. Il fallait mettre des points d'arrêt sur chaque sortie de boucle potentiel. La logique du programme était très difficile à appréhender par la personne chargée d'effectuer des modifications.

Les instructions goto, continue, break (hors switch) sont donc vivement déconseillées.

Exemple

```
for (int i = indiceMax; table[i] != NULL; i++)
{
```

```
   free(table[i]);
}
```

for pour contrôler les boucles (c_rec_3)

L'instruction d'une boucle for ne doit contenir aucune instruction vide. De plus, chacun de ses membres doit porter sur au moins une variable commune.

Justification

L'instruction **for** doit permettre le contrôle total d'une itération. Si une instruction est vide, cela veut dire que la boucle aurait pu être écrite sous une autre forme (**do** ou **while**).

Mettre des parenthèses dans les expressions (c_rec_4)

Les expressions arithmétiques et logiques doivent être placées entre parenthèses.

Justification

Cette règle évite les erreurs d'interprétation dues aux règles d'associativité du langage C. Son non respect peut conduire à des erreurs dans le codage des expressions mathématique ou de test.

Exemple

On veut tester si n est pair : n & 1 effectue un ET logique de bit entre n et 1, mais l'opérateur relationnel == est prioritaire par rapport à &, l'expression (n & 1 == 0) sera donc toujours fausse.

- Mauvais : if (n & 1 == 0) ...
- Bon : if ((n & 1) == 0) ...

Mettre des parenthèses autour des paramètres des macros (c_rec_5)

Les macros doivent être écrites avec des parenthèses autour de leurs paramètres.

Justification

Lors du remplacement des paramètres formels par les paramètres effectifs, il peut y avoir des problèmes de priorité des opérateurs dans les expressions résultats.

Exemple

Mauvais : `#define double(a) 2 * a` :
L'appel double(2+1) sera étendu en 2 * 2 +1 : vaudra 5 au lieu de 6.

Meilleur : `#define double(a) 2 * (a)` :
L'appel double(2+1) sera étendu en 2 * (2 +1) : vaudra 6

Ne pas écrire de macro de plus de 5 instructions (c_rec_6)

Justification

Au-delà de 5 instructions, le déroulement est assimilable à un traitement, et doit faire partie intégrante d'une fonction. De plus, les macros instructions ne permettent pas un contrôle strict de leurs paramètres.

Si l'utilisation d'une fonction entraîne une dégradation des performances, il faut recourir à l'inlining. Le mot clé C99 inline répond à ce problème.

Pas d'opérateurs unaire à l'appel des macros (c_rec_7)

Ne pas placer d'opérateurs unaire (++, --) dans les paramètres d'appel des macros.

Justification

Le résultat est bien souvent imprévisible par suite des effets de bord, surtout en cas de répétition du paramètre dans la définition de la macro.

Éviter les problèmes d'inclusion multiple (c_rec_8)

On protégera chaque fichier d'en-tête .h des inclusions multiples.

Justification

Lorsqu'un fichier d'entête (.h) est inclus plusieurs fois directement, un test en son début permet d'éviter les erreurs de compilation dues à des déclarations répétées :

Exemple

Pour le fichier *graphique.h*:

```
#ifndef GRAPHIQUE_H
#define GRAPHIQUE_H
   // corps du fichier d'en-tête
   // ...
#endif
```

Mélange d'opérateur arithmétique et relationnel (c_rec_9)

Le calcul d'une expression complexe au sein d'une condition est à éviter.

Justification

Facilite l'analyse des sources, évite de superposer les erreurs de priorité des opérateurs arithmétiques à ceux des opérateurs relationnels.

Prototypes de fonction (c_rec_10)

Les fonctions doivent être prototypées, une fonction ou une variable ne doit pas avoir de type par défaut.

Justification

Facilite l'analyse. Permet un meilleur contrôle de la correspondance arguments - paramètres. La norme C99 interdit de transgresser cette règle.

Outils

Le contrôle du peut être effectué en utilisant le compilateur C avec options les plus strictes ou un outils qualité comme lint ou splint.

Les variables ne doivent pas se masquer (c_rec_11)

Une variable ne doit pas en masquer une autre. Cela se produit lorsque deux variables ont le même nom dans des blocs imbriqués.

Justification

Lorsque plusieurs entités sont désignées par le même nom et ne peuvent être distinguées que par leur position par rapport aux structures de contrôle, cela réduit la lisibilité du code et favorise l'apparition d'erreurs, en particulier lors de la maintenance du source.

Exemple à ne pas suivre

Dans l'extrait de source suivant, les variables i se masquent.

```
// Première déclaration de i
int i = 0;
// ...
{
   // Deuxième déclaration de i
   int i = 0;
   // ...
   i++;
   // ...
}
```

Si la deuxième déclaration de i est supprimée, alors il y a un risque pour que l'incrémentation ne le soit pas, ce serait alors la variable déclarée hors du bloc qui serait utilisée, ce qui ne serait pas le comportement souhaité. Le code resterait alors valide et aucun problème ne serait détectée.

Pas de suppression de l'attribut const (c_rec_12)

Justification

Selon le compilateur, il est possible que les données const soient stockées dans une zone accessible en lecture, mais pas l'écriture.

Limiter l'utilisation des pointeurs (c_rec_13)

Éviter l'utilisation des pointeurs. Préférer l'utilisation des indices de tableau.

Justification

L'utilisation abusive de pointeurs est la source de graves dysfonctionnements très difficiles à détecter. Les programmeurs habitués à d'autres langages sont souvent perdus face à certaines subtilités du C. Pour les parcours de tableaux, il est préférable d'utiliser des indices (tab[i]) au lieu du déréférencement par pointeur.

Utiliser le mot-clé const (c_rec_14)

Utiliser le mot-clé ANSI const pour définir :

- Une variable non modifiable.
- Un argument non modifiable par une fonction.

Justification

Rejette, dès la phase de compilation, certaines modifications abusives.

Affectations et identificateurs de tableau (c_rec_15)

Être prudent lors de l'utilisation des identicateurs de tableau et des pointeurs sur chaîne constante.

Justification

Les types int a[] et int* a sont complètement différents, même si, une fois déclarés, leur usage parait identique. Le programmeur doit être particulièrement vigilant : par exemple, beaucoup d'éditeurs de liens confondraient un int* a et un int a[] définis dans deux modules (.o) différents. Ceci est susceptible de provoquer une erreur fatale.

Exemple

Voici une illustration de la différence entre ces deux types [C FAQ] :

```
char a[] = "hello";
char* p = "hello";
```

La variable " a " occupe 6 octets dans l'espace de la mémoire dynamique. Cette zone sera désallouée lorsque la variable sortira de son espace de validité. La variable p occupe 4 octets (taille courante d'un pointeur). Elle est un pointeur qui référence une région de la mémoire non modifiable. Une nouvelle valeur peut être affectée à "p", mais pas à "a". En fait, un bon compilateur devrait imposer ici le type const char*.

Facilité de modification

Un programme est souvent amené à être modifié, adapté, réutilisé par celui qui l'a écrit ou par un autre, sur une longue durée (correction d'erreur, évolution du logiciel, réutilisation dans une autre application...).

Ces conseils visent à réduire les efforts nécessaires lors d'évolutions.

Éviter les constantes littérale (c_mod_1)

Les constantes chaîne, numérique doivent être représentées par des symboles. Les constantes littérales sont interdites.

Justification

Les constantes littérales dupliquées et réparties dans le code source rendent les modifications plus difficiles : risque d'altération, d'oubli d'une occurrence. Il faut les regrouper sous formes de constantes symboliques (définition du pré-processeurs) en tête de fichier source ou dans des fichiers inclus.

Exemple

- Mauvais : `perimetre = 3.14159 * diametre;`
- Meilleur : `#define PI 3.14159 ... perimetre = PI * diametre;`

Écrire des commentaires fonctionnels (c_mod_2)

Les commentaires ne doivent pas paraphraser le code, ils doivent expliquer la logique de traitement et justifier les choix effectués (pourquoi tel algorithme, telle structure de données, ...).

Justification

La personne, qui relira le code source en vue de sa maintenance, aura besoin de comprendre les grandes étapes de l'algorithme, les fonctions réalisées par la suite d'instructions du langage.

Exemple

Mauvais :

```
// Incrémentation de i
i = i + 1
```

Meilleur :

```
// Passage au fichier suivant
i = i + 1
```

Pas d'éléments inutiles ou non initialisés (c_mod_3)

Toute fonction définie doit être utilisée et toute variable déclarée doit être initialisée.

Justification

Évite la présence de code mort qui nuit à la maintenance.

La valeur par défaut d'une variable dépend de facteurs qui peuvent échapper au programmeur. Cette règle évite les fonctionnements aléatoires de certains programmes.

Outils

Les compilateurs et les outils de vérifications statiques comme splint émettent des avertissements en cas de problème. Certains compilateurs possèdent des options pour générer tous les avertissements sous forme d'erreur, ce qui permet d'interrompre une chaîne de compilation afin de corriger le code.

Pas de duplication de code source (c_mod_4)

Ne pas dupliquer de grandes parties de code dans plusieurs fichiers source.

Justification

- La duplication de source oblige à intervenir à plusieurs endroits pour corriger une erreur.
- Rend le code source moins maintenable.
- Lire plusieurs fois le même code ralentit sa compréhension.

Les parties dupliquées doivent être fusionnées en une seule. Le code commun, éventuellement paramétré, deviendra une fonction. Si des contraintes de performance l'impose, l'inlining pourra être utilisé.

Outils

La recherche des parties dupliquées pourra se faire avec un outil de recherche de chaînes de caractères : sous UNIX **grep** ou à l'aide d'un éditeur s'il permet la recherche de texte dans plusieurs fichiers.

Limiter le nombre des variables externes ou globales (c_mod_5)

L'utilisation de variables externes ou globales doit être limité au maximum.

Justification

Les variables globales peuvent être modifiées par plusieurs unités de compilation, le contrôle des modifications est donc très délicat, les effets de bord sont possibles (modifications non désirées d'une variable ou d'une zone mémoire par du code lié). Un trop grand nombre de variables externes dénote une mauvaise analyse des interfaces entre unités de compilation ou une mauvaise gestion de la portée des variables.

Le mot-clé *static* en C permet, entre autres, de limiter la portées d'une variable globale ou d'une fonction à son seul fichier source. Il garantit que ces fonctions ou variables ne seront pas utilisées, par erreur, par d'autres unités. Les fonctions locales pourraient être masquées ou masquer d'autres fonctions de même nom, mais définies ailleurs dans le projet ou dans une bibliothèque liée. Le comportement du programme pourrait alors dépendre de l'ordre dans lequel les objets et bibliothèques sont présentés à l'édition de lien. Le respect de cette règle allégera l'édition de lien et évitera certains effets de bord.

Outils

- Recherche dans les fichiers sources des mots clés *extern*, des déclarations de variables hors des fonctions.
- L'utilitaire Unix **nm** et ses options -g et -o permet de connaître les symboles externes dans les résultats de compilation (modules objets .o, bibliothèques .a, .so).

Ne pas réinventer la roue (c_mod_6)

Utiliser les fonctions et bibliothèques écrites par d'autres programmeurs, principalement dans :

- les bibliothèques standards du langage.
- les bibliothèques de calcul numérique reconnues (LAPACK, LINPACK, BLAS).
- tout autre développement suffisamment fiable.

Justification

Le respect de ce conseil limite le nombre de lignes de code à maintenir ainsi que les efforts nécessaires aux tests du logiciel.

Limiter le nombre de sortie (c_mod_7)

Une fonction ne devrait comporter qu'un seul point de sortie.

Justification

Diminue la complexité du code et facilite l'analyse du déroulement du programme en mise au point.

Contrôle

Rechercher des instructions *return* et *exit* à l'aide d'un utilitaire comme **grep**.

Évitez le type union (c_mod_8)

L’utilisation du type union est déconseillé, sauf pour manipuler des champs de bits.

Justification

Améliore la portabilité.

Pas de code dans les .h (c_mod_9)

Ne jamais écrire de fonctions ou d'instructions dans les fichiers d’entête .h (excepté les macros instructions).

Justification

Écrire le corps de fonction dans un fichier .h et l’inclure dans plusieurs sources dupliquerait ce code et pourrait provoquer des erreurs due aux symboles définis plusieurs fois. Plutôt que d’utiliser le pré-processeur, il faut répartir le code source dans différents fichiers .c qui seront compilés séparément et réunit lors de l’édition de lien.

Seuils pour les métriques déterminant la facilité de modification (c_mod_10)

Les métriques suivantes influent sur la facilité de modification d'un programme (ISO/IEC 9126) :

- AVGS : Taille moyenne des instructions, calculée à partir du nombre d'opérateurs et d'opérandes distincts.
- LEVL : Nombre maximal d'imbrications des structures de contrôle (for, while, if...) de la fonction plus un.

- VOCF : Fréquence d'utilisation du vocabulaire dans un composant, défini par la formule : F_VOC = (N1+N2)/ (n1+n2) avec :
  - N1 est le nombre d'occurrence des opérateurs,
  - N2 est le nombre d'occurrence des opérandes,
  - n1 est le nombre opérateurs distincts,
  - n2 est le nombre opérandes distincts.
- CALL : Nombre d'appels distincts : Nombre total d'appels de fonctions effectués dans la fonction analysée.

Seuils pour les métriques :

- AVGS : de 2 à 10
- LEVL : de 1 à 5
- VOCF : de 1 à 4
- CALL : de 0 à 7

Justification

- AVGS : Les instructions trop longues sont difficiles à modifier. Elles peuvent provoquer des problèmes de priorité des opérateurs, des conversions de type involontaires.
- LEVL : Un trop grand nombre d'imbrication de structures de contrôle pourra être une source d'erreur en cas de modification d'un programme. Il faut déplacer certains niveaux dans une fonction.
- VOCF : Un vocabulaire qui se répète trop pourra être dû à des copiés-collés abusifs de parties de code ou à un programmeur à jeu d'instruction réduit. Les modifications à apporter devront alors s'appliquer à de nombreux endroits. Une solution consiste à factoriser le code à l'aide de fonction paramétrées ou à réutiliser du code externe développé et validé (bibliothèques spécifiques).
- CALL : Une valeur trop élevée montre une mauvaise hiérarchisation dans la fonction. Un nombre élevé de fonctions appelée et de paramètres augmente le nombre de combinaisons possibles dans le déroulement du programme. Il faut limiter le nombre d'appel au besoin en découpant la fonction.

Robustesse des programmes

L'application de ces conseils suivants rendent les logiciels plus robustes : plus tolérants aux fautes.

Compilation stricte et outils qualité (c_rob_1)

Il faut compiler en utilisant les options les plus strictes, qui demandent au compilateur d'indiquer tous les avertissements et toutes les erreurs.

Il faut utiliser des outils de qualité qui permettent d'effectuer encore plus de contrôle.

Il faudra ensuite comprendre les messages puis corriger les sources afin d'éliminer tous ces avertissements.

Justification

Cette pratique permet d'obtenir des logiciels plus robustes et plus portables

Outils

- gcc avec les options -Wall -Werror -pedantic
- Outils de contrôle statique (syntaxique) comme splint, lint, proLint...
- Outil de contrôle dynamique (gestion des ressources à l'exécution) comme leaks, purify.

switch et clause default (c_rob_2)

Les instructions de choix multiple : (*switch, case*...) doivent comporter une clause pour les valeurs non testées individuellement (*default*).

Justification

Permet de traiter les cas non prévus explicitement en détectant les cas dégradés dus à des valeurs inattendues.

Ces valeurs peuvent provenir :

- de modifications non contrôlées d'une autre partie du code ;
- d'un mauvais interfaçage de la fonction ;
- d'une valeur non prévue reçue ou lue ;
- pour un appel système, à la compilation et de l'exécution sur un système non prévu.

Exemple (extrait)

```
//...
  while ((optc = getopt_long (argc, argv, "htvm", longopts, (int *) 0)) != EOF)
    {
      switch (optc)
        {
        case 'v':
          v = 1;
          break;
        case 'h':
          h = 1;
          break;
        case 'm':
          m = 1;
          break;
        default:
          unknown = 1;
          break;
        }
    }
```

Outils

- Les outils de vérification statique, comme **splint**, émettent un warning lorsqu'une clause *default* est oubliée.
- Un comptage des mots clés des mots switch et default dans un éditeur de texte ou avec les outils **grep** et **wc** d'UNIX.

Test des codes retours (c_rob_3)

Les codes de retour des fonctions et surtout des appels systèmes doivent être testés.

Justification

Les codes retournés par les fonctions peuvent signaler :

- des problèmes dans leur déroulement : validité des paramètres, erreurs de calcul...
- des erreurs dues à l'environnement du logiciel : non - conformité de l'arborescence des fichiers, droits d'accès, ressources non disponibles...

Ignorer ces problèmes peut conduire aux pires catastrophes.

Une bonne habitude consiste à caster en void tous les appels à des fonctions dont on souhaite ignorer les codes retour. Cette pratique facilite les contrôle avec les outils qualité.

Exemple

```
//...
        hFile = fopen(NOM_FIC, "r");
        if (hFile == NULL)
        {
                perror("Erreur");
                (void)fprintf(stderr,
                        "Impossible d'ouvrir %s en lecture\n",
                        NOM_FIC);
                exit(EXIT_FAILURE);
        }
```

Outils

Les outils de contrôle statiques comme **splint** émettent un warning lorsqu'un codes retour de fonction n'est pas testé ou explicitement ignoré.

Gestion des ressources (c_rob_4)

Toute ressource allouée (mémoire, fichier ouvert, outil de synchronisation, ...) doit obligatoirement être libérée quoi qu'il se passe, erreur ou non.

Justification

Une ressource non libérée après utilisation peut n'être libérée qu'à la fin de l'application. Mais auparavant, celle-ci limite les ressources disponibles du système et si le code est appelé plusieurs fois, une erreur de ressource peut se produire (pas assez de mémoire, ...).

Outils

Un certain nombre d'outils permet de détecter les fuites mémoires et de voir l'occupation des ressources.

Mais avant tout, il faut avoir un code de gestion des ressources correct :

- Tester le code de retour de la fonction d'allocation, et ne pas exécuter le reste du code en cas d'échec,
- Libérer la ressource quoi qu'il se passe, en un seul endroit du code afin d'éviter de libérer plusieurs fois la même ressource, et si possible dans la même fonction que l'instruction d'allocation.
- Mettre à NULL le pointeur sur la structure de contrôle du fichier (file handle) pour éviter la l'utilisation ultérieure d'une ressource invalide.

Exemple

```
//...
        hFile = fopen(NOM_FIC, "r");
        if (hFile == NULL)
        {
                perror("Erreur ouverture fichier");
                (void)fprintf(stderr,
                        "Impossible d'ouvrir %s en lecture\n",
                        NOM_FIC);
                exit(EXIT_FAILURE);
        }
        else
        {
                erreur = traiterFichier(hFile);
                if (fclose(hFile) != 0)
                {
                      perror("Erreur fermeture fichier");
                }
                hFile = NULL;
        }
```

Le traitement du fichier est délégué dans une fonction séparée. Celle-ci pouvant alors retourner un code d'erreur avec l'instruction `return` sans se soucier de fermer le fichier ouvert, vu qu'il sera fermé juste après l'appel à la fonction.

Contrôle des arguments (c_rob_5)

Avant de commencer un traitement, un module doit effectuer un contrôle minimum sur les valeurs de ses arguments (pointeur NULL, valeur hors domaine de calcul, chaîne de caractère correcte).

Justification

Améliore la stabilité et la fiabilité du logiciel. Évite les erreurs provenant de la poursuite du traitement avec des données d'entrée invalides.

Remontée des erreurs (c_rob_6)

Lorsqu'il y a une erreur, le programme ou la fonction doit générer un code spécifique à l'erreur sur la sortie réservée aux erreurs.

Justification

Maintenabilité. Identification rapide des erreurs.

Traitement efficace des retours d'erreurs qui peuvent en étant considérée comme des valeurs de sortie normale entraîner des catastrophes.

Utiliser le mécanisme d'assertion (c_rob_7)

Les assertions sont des tests à placer dans vos programmes. C'est une condition qui doit être obligatoirement vérifiée. Dans le cas contraire le programme s'arrêtera.

Le langage C implémente les assertion à l'aide d'une macro : assert(condition) qui en plus est débrayable.

Justification

Vous devez utiliser ce mécanisme d'assertion pour construire des programmes plus robustes et faciliter la maintenance et la relecture du code.

Lorsque vous concevez une fonction, vous identifiez des invariants : des conditions qui doivent toujours être vérifiées pour pouvoir exécuter le traitement ou fournir des résultats utilisables. Par exemple pour une loi physique : une température doit toujours être supérieure ou égale à 0 Kelvin.

Vous coderez ces conditions sous forme d'assertion. En phase de mise au point, si une assertion n'est pas vérifiée, le programme ne doit pas se poursuivre. Lors de la phase de test, lorsqu'une assertion n'est pas vérifiée, le programmeur doit alors coder un mécanisme permettant de présenter un message d'erreur clair ou un contournement qui permettra de continuer. Les assertions qui n'auront pas été déclenchées en fin de période de test seront laissées dans les sources.

Performances

Ces conseils permettent d'obtenir des programmes plus rapides ou utilisant seulement les ressources nécessaires.

Éviter d'éparpiller les entrées-sorties (c_perf_1)

Il faut éviter d'effectuer des entrées sorties dans une partie du programme qui réalise des calculs de façon intensive. Il faut regrouper ces opérations d'échange dans une partie consacrée à l'initialisation ou à l'écriture et la communication des résultats.

Justification

L'appel répété aux fonctions du système est pénalisant pour les performances. Les accès aux moyens de stockage de masse (disques) ou au réseau sont beaucoup plus longs que les échanges avec les registres ou la mémoire vive.

Utiliser l'algorithme le plus rapide (c_perf_2)

Pour effectuer une même opération, il peut exister différents algorithmes ayant un nombre d'opérations différent. Il faut employer le plus adapté pour obtenir de meilleures performances en temps de réponse.

Justification

Les instructions issues de la compilation des fichiers sources sont exécutées par un processeur, mais pas de façon simultanée. Certaines opérations prennent plus de temps que d'autres.

Exemple

- Pour trier un grand nombre d'information, utiliser un algorithme de tri rapide de type Quick Sort comme qsort de la librairie C standard. plutôt qu'un tri bulle.
- Pour des calculs mathématiques (matrices, algèbre linéaire), utiliser des bibliothèques spécialisées.

Sécurité

L'application des conseils suivants rendent les logiciels plus sécurisés : moins vulnérables aux attaques.

Pas d'affichage direct (c_sec_1)

Il faut éviter d'afficher une chaîne provenant de l'utilisateur (argument de programme, de fonction, contenu d'un fichier) en l'employant comme premier argument des fonctions `printf`.

Justification

Améliorer la sécurité en évitant le plantage du programme.

Exemple

Le mauvais exemple :

```
/* prog.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    if (argc > 1)
    {
        printf(argv[1]); // afficher le premier argument
    }
    return EXIT_SUCCESS;
}
```

Que va-t'il se passer si l'on tape la commande suivante :

```
prog "Exemple : %s"
```

L'argument pour le %s n'étant pas fourni explicitement, le programme risque d'afficher le contenu de la mémoire à une adresse indéfinie, voire retourner une erreur de segmentation.

Le bon exemple :

```c
/* prog.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    if (argc > 1)
    {
        (void)printf("%s", argv[1]); // afficher le premier argument
    }
    return EXIT_SUCCESS;
}
```

Si l'on tape la commande suivante :

```
prog "Exemple : %s"
```

Le programme affiche littéralement :

```
Exemple : %s
```

Cacher les mots de passe (c_sec_2)

Lorsqu'un programme a besoin d'un mot de passe saisi par l'utilisateur, que ce soit par l'interface graphique, ou par une console texte, la saisie ne doit pas être faite en clair.

Il faut :

- soit afficher des étoiles ou points (ou tout caractère fixé) lors de la frappe,
- soit ne rien afficher du tout.

Justification

Améliorer la sécurité en évitant que les mots de passe d'un utilisateur ne soient visibles aux autres (poste public, lieu de travail, ...), et éviter une opération intempestive de presse papier contenant le mot de passe (envoi d'une copie d'écran par exemple).

Outil

Certaines librairies graphiques possèdent des champs de saisie spéciaux pour les mots de passe, d'autres librairies possèdent des fonctions de saisie de mots de passe sur la console.

Pour la saisie par console, si aucune fonction spéciale existe, il faut désactiver l'affichage de la saisie tant que l'utilisateur n'a pas fini sa saisie.

Programmes plus exploitables

Ces quelques conseils permettent de réaliser des programmes plus faciles à utiliser, plus faciles à porter sur une autre architecture.

Conseils de codage en C

Utiliser la norme la plus récente (c_exp_1)

Lors d'un nouveau développement, il vaut parfois mieux utiliser une norme récente du langage tel que le C99 .

Justification

Le codage est plus efficace lors de l'utilisation des nouvelles fonctionnalités introduite par la norme (mot-clé restrict, type de taille fixe, tableau de taille variable (VLA), boolean...).

Le programme sera plus facile à reconstruire et à porter sur une autre machine dans le futur.

Tout chemin d'accès figé est interdit (c_exp_2)

Les chemins d'accès aux fichiers ne doivent pas être écrits en dur dans les fichiers sources.

Justification

Un programme doit être paramétrable sans avoir à modifier ses fichiers sources et le recompiler.

Les chemins d'accès aux ressources doivent être récupérés depuis l'environnement ou depuis des fichiers de configuration : de préférence dans un format auto-décrit et éditable : comme XML.

Code retour du programme (c_exp_3)

Un programme doit toujours renvoyer la valeur 0 (EXIT_SUCCESS de stdlib.h) en cas de fin correcte ou un code erreur en cas d'incident (généralement EXIT_FAILURE de stdlib.h).

Justification

- Ce comportement respecte le standard de fonctionnement d'UNIX et peut s'appliquer à d'autres système (MS-DOS/Windows).
- Permet d'utiliser un programme dans une chaîne de traitement par lot et d'arrêter le déroulement en cas d'erreur.

Des messages d'erreur utiles (c_exp_4)

Les messages d'erreur émis par les logiciels doivent aider les utilisateurs :

- L'utilisateur doit pouvoir les comprendre : par leur langue, par leur vocabulaire.
- Ils doivent lui permettre de corriger le problème ou de contacter un support technique.
- Ils doivent proposer éventuellement un moyen de contournement, une cause probable.
- Leur contenu doit différer selon qu'ils sont destinés à l'utilisateur ou à la personne qui assure la maintenance.
- Ils doivent permettre par un moyen (fichier de log par exemple) de fournir à la maintenance le plus possible d'informations : version du programme, date, nom de l'utilisateur, plate forme, nom de fonction,

numéro de ligne, message système, valeurs ayant provoqué l'erreur…

Justification

Améliore l'ergonomie du logiciel.

Exemple

- Mauvais message : *Can't open file*, puis arrêt brutal du programme.
- Meilleur message : *Accès impossible en écriture au fichier configuration.txt. Vérifiez ses droits d'accès.*

Fichiers temporaires (c_exp_5)

Les fichiers temporaires doivent être générés dans les répertoires prévus à cet effet. Ils doivent être détruits à la fin de l'exécution (qu'il s'agisse d'une terminaison normale ou dégradée).

Justification

Limite la prolifération des fichiers parasites.

Outils

Une analyse des appels systèmes en cours d'exécution du logiciel peut se réaliser avec un utilitaire comme **strace** pour UNIX ou **ktrace** sur Mac. Analyse des modifications du système de fichiers.

Qualité numérique

L'application de ces règles doit permettent d'éviter des erreurs de calcul.

Types homogènes dans les calculs (c_num_1)

Une expression arithmétique ne doit comporter que des éléments du même type.

Justification

Évite les erreurs de calcul dues aux règles de conversions implicites des langages. C'est le programmeur qui doit définir explicitement les conversions de type. Les conséquences du mélange d'opérandes de même type :

- Perte de précision par troncature de la mantisse.
- Perte de la valeur numérique, par exemple : un grand entier long devient un entier court négatif.

Exemple

```
unsigned int u =2;
int n = -2;
n = n / u;
```

Programmation C

Modifier le sommaire

n ne vaut pas -1 à la fin, mais une valeur du genre : 2147483647, ce qui faussera beaucoup les calculs ultérieurs , pour avoir un résultat correct il aurait fallut écrire n = n/(int) u;

Outils

Les compilateurs et les outils de contrôle statique comme splint émettent des avertissements.

Ne pas tester l'égalité de deux réels (c_num_2)

Justification

Améliore la qualité numérique.

a = a +1 dans une boucle peut produire la valeur a = 9.999999999 au lieu de 10.0, une instruction du genre `if (a==10.0)` ne fonctionnera pas comme prévue.

Au lieu de `if ( A == B )`

Écrire `if ( fabs( A - B ) < EPS )` EPS est une valeur très petite qui dépend de la précision de la machine, utiliser si disponible des constantes symboliques définies sur le système.

Outils

Les compilateurs et les outils de contrôle statique comme splint émettent des avertissements.

Limitez les erreurs numériques (c_num_3)

Évitez les phénomènes d'annulation, d'absorption et d'arrondi.

- Annulation: Erreur qui se produit lors de la soustraction de deux valeurs trop proches.
- Absorption : Erreur qui se produit lorsqu'on additionne deux valeurs d'un ordre de grandeur trop différent. Ce phénomène est sensible lorsqu'on réalise l'opération un grand nombre de fois.
- Arrondi : pertes de chiffres significatifs dans la partie décimale du nombre.

Justification

Les logiciels informatiques tentent de manipuler des nombres réels.

L'ensemble mathématique des décimaux, déjà plus restreint, est infini en étendue : il contient les nombres compris entre plus à moins l'infini. L'écart entre deux nombres peut être infiniment petit. Par contre les nombres en machine sont stockés dans un espace mémoire de taille très limité (32, 64, 128 bits...). Les nombres y sont représentés par une mantisse et un exposant codés sur un nombre fini de bits. On pourrait comparer le domaine de représentation des nombres en machine à une tranche de gruyère pleine de trous.

- Les résultats trop grands ou trop petits sont tronqués ou changent de signe.
- Une valeur trop petite peut être ignorée lorsqu'elle est additionnée à un grand nombre.
- La différence entre deux nombres peut être considérée comme nulle alors que ces nombres sont différents.

Il faut parfois réorganiser les expressions numériques et leur ordre de calcul pour éviter ces erreurs.

==

Je m'appelle Thierry.

Je m'intéresse à la qualité des logiciels.

Ma page principale se trouve sur Wikipédia Utilisateur:Thierry46

Ma devise : *Additionnons nos forces, partageons nos connaissances !*

Contributions

- Exercices en langage C
- Programmation C
- Conseils de codage en C

Autres projets

- Sur Wikiversity : v:fr:utilisateur:Thierry46
- Sur Wikipedia : w:fr:utilisateur:Thierry46
- Sur Wikimedia Commons : c:user:Thierry46

==

Conseils de codage en C/Version imprimable/Onglets

Bonjour !

Étant ingénieur informatique, beaucoup de mes contributions concerneront ce domaine, en particulier la programmation et les protocoles. Je créé également quelques gadgets pour MediaWiki.

J'utilise parfois un bot que je suis en train de développer en Java. N'hésitez pas à demander ses services ici ou là.

Wikilivres	Wikipédia	Wikinews	Wikiversity	Wiktionnaire	W
fr en es	fr en es	fr en es	en	fr en es	

Wikisource	Commons	Wikispecies	Meta	MediaWiki	Inc
fr en es	en	en	en	en	

Babel

fr Cet utilisateur a pour **langue maternelle** le **français**.

en-3 This user is able to contribute with an advanced level of English.

es-1 Este usuario puede contribuir con un nivel **básico** de **español**.

fr en es fr en es

b-0 Cet utilisateur refuse la **langue de bois**.

Présentation

France ∇
Je viens de France

Administrateurs
Je fais partie des administrateurs de Wikilivres francophone
(vérifier (https://fr.wikibooks.org/w/index.php?title=Sp%C3%A9cial:Listusers&limit=1&username=Conseils_de_codage_en_C/Version_imprimable) ; voir mon élection).

en en

Bureaucrate
Je fais partie des **bureaucrates** du Wikilivre francophone

Utilisateur SUL ∇
Cet utilisateur a créé un compte global et son compte principal est sur Wikilivres français.

Cet utilisateur contribue à Wikilivres depuis ***3288*** jours.

Wiki-bricoleur
Je bricole des modèles et des gadgets.

Éclair...
ou Éclair...cie en ce moment ?

- Reporter un bug de mediawiki

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Conseils_de_codage_en_C

/Version_imprimable&oldid=441443 »