Optimisation

Le maitre de Linux écouter tu dois ! Trop d'erreurs dans ton code sont présentes. Ce TD à mieux coder t'apprendra. Si la force est avec toi, optimiser ton code facilement tu feras. Deux types d'optimisation tu apprendras : l'algorithmique et la logicielle. Mais attention, le côté obscur de la force, la feignantise, éviter tu dois. Avec elle, des sales notes tu auras et des amis tu perdras. Alors, ton destin (et ton clavier) entre tes mains tu dois prendre ! Que le C soit avec toi.

1 Optimisations algorithmiques

L'objectif de cette section est d'optimiser vos boucles. Certaines actions paraissent anodines mais alourdissent inutilement le code lors de son exécution. Après une présentation des erreurs courantes, vous simplifierez les boucles présentées dans l'exercice. Voyons donc deux erreurs classiques qui sont rédigées dans des boucles.

Les appels de fonction dans une condition d'arrêt de boucle,
tu éviteras.

Supposons dans un premier temps que la variable `i` soit entière et `str` soit une chaine de caractères et considérons le code ci-dessous.

```
1 for (i=0; i<strlen(str); i++)  // Code non optimisé
2   printf ("%c", str[i]);
```

Ce premier code permet d'afficher l'ensemble des caractères de la chaine `str`. Remarquons qu'à chaque tour de boucle, l'appel à la fonction `strlen()` est réalisé alors que cette dernière renvoie toujours le même résultat. Il est donc plus intéressant de sortir cette évaluation de la boucle et de tester sa valeur dans la boucle, comme présenté dans le code ci-dessous.

```
1 int l = strlen(str);           // Code optimisé
2 for (i=0; i<l; i++)
3   printf ("%c", str[i]);
```

Des tests dépendants de l'indice de boucle,
tu te méfieras.

Le code suivant présente également des calculs superflus. En effet, au lieu de tester à chaque tour de boucle si la valeur est paire ou non, il est plus intéressant de réaliser deux boucles : une première pour les indices pairs, une autre pour les indices impairs.

```c
int tab[15] = {0};              // Code non optimisé
for (i=0; i<15; i++)
{
  if (i%2 == 0)
    tab[i]++;
  else
    tab[i]--;
}
```

Exercice 1.

1) Dans un premier temps, simplifiez le code précédent.
2) Pour chacun des codes précédemment rédigés, comptez le nombre exact d'opérations réalisées.
3) Simplifiez chacun des codes suivants en expliquant ce qui ne va pas.
4) Donnez également le nombre d'opérations réalisées avant et après optimisation.

```c
for (i=0; i<2*n; i++)
  printf ("%d ", i+1);
```

```c
for (i=0; i<n; i++)
{
  if (i==0 || i==n)
    printf ("*");
  else
    printf ("-");
}
```

Exercice 2.

Les deux codes suivants peuvent être simplifiés. Une nouvelle fois, expliquer où est l'erreur et optimisez le code. Indice : les erreurs présentes sont plus liées au langage C qu'à de l'algorithmique pure.

```c
void chercheMax (int *tab, int taille, int *max)
{
  int i;
  *max = tab[0];
  for (i=0; i<taille; i++)
    if (tab[i] > *max)
      *max = tab[i];
}
```

```c
char *LireFichier (FILE *pFile, int size)
{
  int i;
  char *data = (char*)malloc(size*sizeof(char));
  for (i=0; i<size; i++)
    fread(&data[i], 1, 1, pFile);
  return data;
}
```

2 Optimisations logicielles

Sans parler d'optimisation, il est nécessaire de maintenir un code clair. Qui sait si un jour vous n'allez pas avoir besoin d'un code qui multiplie deux matrices ? Ou bien si quelqu'un a besoin de vos sources ? Les règles suivantes à partir de maintenant toujours tu respecteras.

Une indentation identique et raisonnable tout au long de ton fichier,
tu respecteras.

Il faut maintenir une indentation **identique** tout au long du même fichier. En effet, les éditeurs n'ont pas forcément le même comportement face à la longueur des tabulations. Si votre code est tantôt composé de tabulations, tantôt d'espaces, il y a de fortes chances pour qu'au final, votre indentation ne ressemble à rien. Choisissez donc tout le temps des espaces ou (exclusif) des tabulations pour régler le problème.

De plus les niveaux d'indentation doivent être **raisonnables**. Il est souvent possible d'éviter des niveaux élevés d'indentation en transformant légèrement votre code comme le montre les deux exemples ci-dessous.

```c
if (argc >= 2)
{
  pfile = fopen(argv[1], "r");
  if (pFile != NULL)
  {
    ...
  }
}
```

```c
if (argc < 2)
  return -1;

pfile = fopen(argv[1], "r");
if (pFile == NULL)
  return -2;

...
```

Que non nuls soient les pointeurs,
tu testeras.

Vous le savez, un segfault survient lorsqu'on tente d'accéder à une zone mémoire non allouée. Nombre de ces erreurs lors de l'exécution peuvent facilement être évitées en vérifiant que la zone mémoire à laquelle on tente d'accéder est une adresse valide. Cela se traduit plus simplement par le test d'un pointeur afin de vérifier si ce dernier ne possède pas la valeur NULL. Ce genre d'évènement peut survenir la plupart du temps à la suite des actions suivantes :

— ouverture d'un fichier qui a échouée ;
— argument de fonction qui est nul ;
— allocation mémoire qui a échouée.

Le premier cas est le plus classique et survient lorsque l'on ouvre un fichier dont le chemin est inconnu. Il faut donc systématiquement tester le pointeur après l'appel à la fonction `fopen()`. Le second cas survient particulièrement lors de l'accès aux arguments de la ligne de commande. Il suffit donc de tester que ces derniers sont présents en nombre suffisant à l'aide de la valeur `argc`. Notons également que vous devez systématiquement tester la validité d'un pointeur lors de tout appel de fonction renvoyant une adresse. Enfin les derniers cas ne survient que très rarement et vous êtes (pour le moment) dispensés de vérifier que les allocations se passent bien.

Un pointeur toujours à NULL, mon jeune padawan,
tu initialiseras.

LA règle ci-dessus permet effectivement d'éviter un certain nombre d'erreurs. Elle vient en complément des deux autres et est particulièrement simple à mettre en place. Son coût est en effet négligeable puisqu'aucun test ne dépend d'elle. Elle permettra également de rendre valide la règle précédente. En effet, si on déclare un pointeur sans l'initialiser à NULL, il prendra la valeur présente en mémoire lors de sa déclaration, qui a de grandes chances d'être non nulle. Le test de ce pointeur dans une portion ultérieure de code sera donc inutile et un segfault sera certainement levé.

Exercice 3.

Récupérez le fichier *CodePasBeau.c* sur Moodle. Modifiez-le afin de le rendre compatible avec les consignes de sécurité qui vous ont été données au travers de cette section.

Les Warnings de GCC avec patience,
tu corrigeras.

Vous le savez déjà le compilateur GCC accepte des options pour modifier le comportement de la compilation. Vous connaissez entre autres l'option `-o` qui permet de renommer l'exécutable, l'option `-Wall` qui permet d'afficher l'ensemble des avertissements de compilation, ou encore les options `-lm` ou `-lSDL` permettant de linker des librairies système.

Rappelons dans un premier temps que **la correction des avertissement n'est pas facultative** ! En effet, nombre d'entre eux débouchent sur une erreur de segmentation lors de l'exécution. À l'avenir, n'oubliez donc pas de les corriger avant de lancer votre exécutable.

Exercice 4.

Récupérez le fichier *TropDeWarnings.c* sur Moodle et compilez-le (n'oubliez pas l'option `-Wall`). Corrigez la série de Warning apparaissant dans votre console.

L'option d'optimisation `-O3` de GCC pour du récursif terminal,
tu utiliseras.

Voici maintenant une nouvelle option de GCC qui vous permettra d'optimiser votre code. Il s'agit de l'option `-Ox` (la lettre O) suivie immédiatement et sans espace d'un autre caractère remplaçant la lettre `x`. Suivant la valeur de ce caractère, GCC optimise le code de différentes façons.

— `-O0` : pas d'optimisation ;
— `-O1` : diminue la taille de l'exécutable et la temps d'exécution ;
— `-O2` : mieux que `-O1` ;
— `-O3` : mieux que `-O2` ;
— `-Os` : diminue seulement la taille de l'exécutable.

L'option qui nous intéresse particulièrement est l'option `-O3`. En effet, c'est la seule qui permette d'obtenir vraiment un exécutable utilisant le récursif terminal. Notons enfin que si cette option parait idéale, elle ralentit le temps de compilation, ce qui peut devenir un handicap pour un code conséquent. Pensez simplement que votre système d'exploitation est codé en C et qu'il faut plusieurs heures pour le compiler...

Exercice 5.
Récupérez votre code permettant de réaliser la somme des entiers de 1 à `n` de façon récursive. Nous allons vérifier si ce code est capable de calculer un résultat pour des grandes valeurs de `n`.

1) Votre code **récursif simple** fonctionne-t-il pour `n` = 10^2 ? Pour `n` = 10^4 ? Pour `n` = 10^6 ?
2) Mêmes questions pour le code **récursif terminal**.
3) Lorsqu'il a donné une solution, a-t-il donné la bonne réponse ? Si non, pourquoi ?

3 Conclusion

Au bout de ce TP tu es arrivé et la force en toi tu as enfin trouvée. La sagesse et la patience maintenant tu as acquis. Ton autonomie en C tu as finalement obtenue [1]. Bravo cher Padawan !

Exercice 6.
Pour conclure listez les 7 règles de bonne conduite en vrai français afin de vous en souvenir. Et bien évidemment, tout ceci n'était pas qu'un simple exercice, il s'agit de vraies nouvelles règles à suivre à l'avenir !

1. Marre j'en ai eu d'écrire comme Yoda !