éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

UN CAS CONCRET

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

Ce document présente plusieurs thèmes abordables en NSI au travers du cas concret d'une contribution apportée au logiciel TexStudio au cours même de l'élaboration de la présente collection de documents.

1. Présentation du problème

1.1. Situation générale

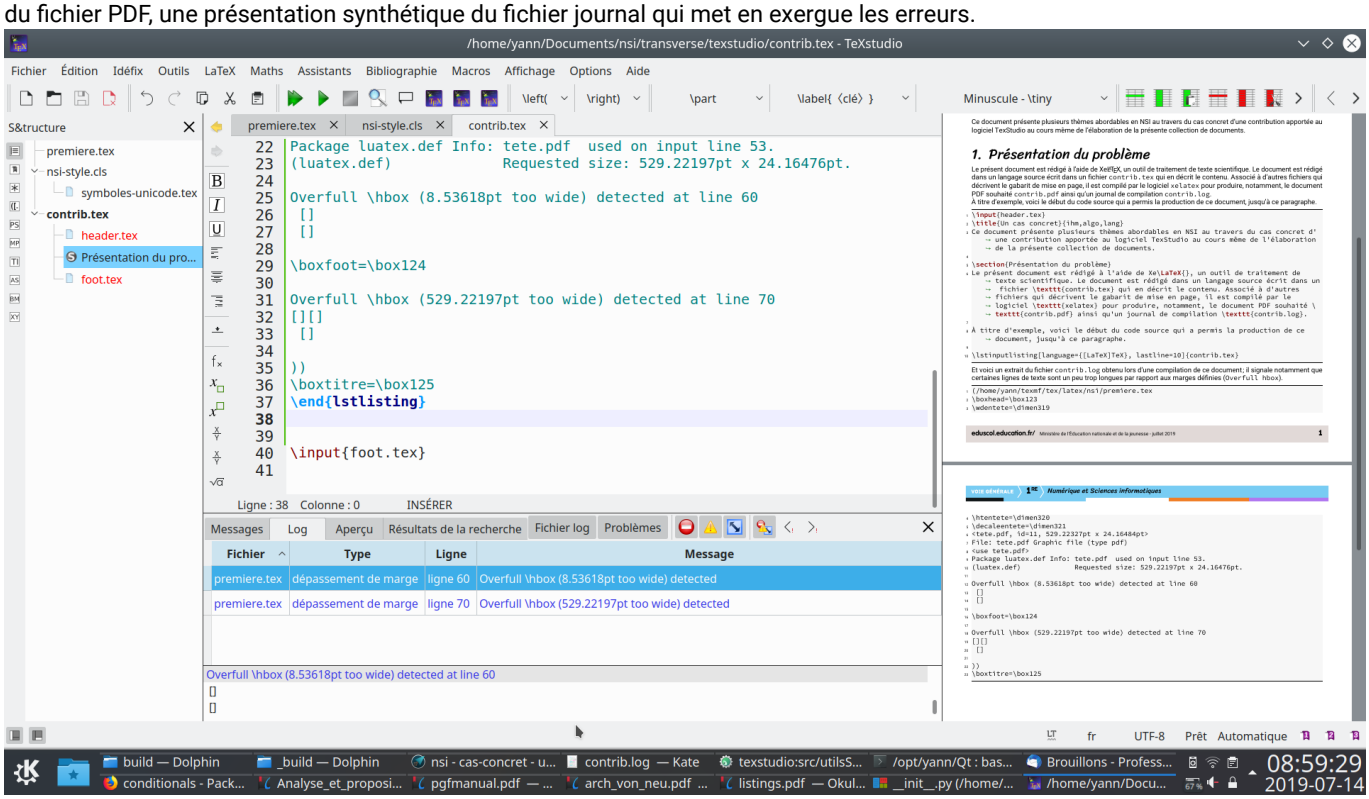
Le présent document est rédigé à l'aide de XeLaTeX, un outil de traitement de texte scientifique. Le document est rédigé dans un langage source écrit dans un fichier `contrib.tex` qui en décrit le contenu. Associé à d'autres fichiers qui décrivent le gabarit de mise en page, il est compilé par le logiciel `xelatex` pour produire, notamment, le document PDF souhaité `contrib.pdf` ainsi qu'un journal de compilation `contrib.log`.
À titre d'exemple, voici le début du code source qui a permis la production de ce document, y compris la commande qui réalise cette inclusion.

```
\input{header.tex}
\title{Un cas concret}{ihm,lang,arch}
Ce document présente plusieurs thèmes abordables en NSI au travers du cas concret d'
    ↪ une contribution apportée au logiciel TexStudio au cours même de l'élaboration
    ↪ de la présente collection de documents.

\section{Présentation du problème}
\subsection{Situation générale}
Le présent document est rédigé à l'aide de Xe\LaTeX{}, un outil de traitement de texte
    ↪  scientifique. Le document est rédigé dans un langage source écrit dans un
    ↪ fichier \>contrib.tex> qui en décrit le contenu. Associé à d'autres fichiers qui
    ↪  décrivent le gabarit de mise en page, il est compilé par le logiciel \>xelatex>
    ↪  pour produire, notamment, le document PDF souhaité \>contrib.pdf> ainsi qu'un
    ↪ journal de compilation \>contrib.log>.

À titre d'exemple, voici le début du code source qui a permis la production de ce
    ↪ document, y compris la commande qui réalise cette inclusion.

\lstinputlisting[language={[LaTeX]TeX}, lastline=11]{contrib.tex}
```

Et voici un extrait du fichier `contrib.log` obtenu lors d'une compilation de ce document; il signale notamment que certaines lignes de texte sont un peu trop longues par rapport aux marges définies (`Overfull \hbox`).

```
(/home/yann/texmf/tex/latex/nsi/premiere.tex
\boxhead=\box123
\wdentete=\dimen319
\htentete=\dimen320
\decaleentete=\dimen321
<tete.pdf, id=11, 529.22327pt x 24.16484pt>
File: tete.pdf Graphic file (type pdf)
<use tete.pdf>
Package luatex.def Info: tete.pdf  used on input line 53.
(luatex.def)             Requested size: 529.22197pt x 24.16476pt.

Overfull \hbox (8.53618pt too wide) detected at line 60
 []
 []

\boxfoot=\box124

Overfull \hbox (529.22197pt too wide) detected at line 70
[][]
 []

))
\boxtitre=\box125
```

Bien entendu, il serait fastidieux de devoir lire à la main le fichier `contrib.log` pour relever les erreurs. Il est fréquent de s'appuyer sur un environnement de développement. Par exemple, le logiciel TeXStudio donne après compilation du document, outre un aperçu du fichier PDF, une présentation synthétique du fichier journal qui met en exergue les erreurs.



1.2. TeXStudio ne s'y retrouve plus

Par commodité, j'ai configuré mon installation XeLaTeX pour que les fichiers intermédiaires, dont le journal, soient écrits dans un sous-dossier `./_build`. Ainsi, le fichier journal est à `./_build/contrib.log`. TeXStudio peut être configuré pour aller lire le fichier journal à cet endroit.
Mais par ailleurs, pour pouvoir produire en une fois l'ensemble des documents ressources pour NSI et les rassembler dans un même dossier, l'équipe de rédaction a mis en place un système de compilation centralisé, qui n'utilise pas ce sous-dossier `._build`. Pour peu qu'on ait lancé une compilation centralisée, il y a donc aussi un fichier journal `./contrib.log`.
Le problème est alors que TeXStudio, ayant trouvé ce fichier journal, ne lit pas l'autre, même quand ce serait pertinent. Il en résulte que si j'introduis une erreur dans `contrib.tex` et que je compile, TeXStudio ne montre pas l'erreur car elle est journalisée dans `./_build/contrib.log` et non dans `./contrib.log`.

2. Chasse au bug

TeXStudio est un logiciel libre dont le code source est disponible à https://github.com/texstudio-org/texstudio.
Il m'a donc été possible d'inspecter ce code source pour localiser l'origine de ce comportement non souhaité.

2.1. Jeu de piste

Il n'est pas aisé de modifier un programme qu'on n'a pas soi-même écrit. C'est cependant possible s'il a été rédigé avec soin, comme on va le voir ici.
L'élément de configuration de TeXStudio qui permet de régler le sous-dossier où aller chercher le fichier journal porte les mentions « Chemins de recherche additionnels » et « Fichier log ». Une recherche sur ces chaines dans le fichier de la traduction française permet d'obtenir la version anglaise de ces phrases ainsi que les lignes du code de l'interface utilisateur où elles sont utilisées.
Il s'agit d'un fichier XML qui donne la structure de l'interface graphique et dont voici un extrait pertinent.

Code 1 – configdialog.ui

```
959 <item row="0" column="1">
960   <widget class="QLineEdit" name="lineEditPathLog"/>
961 </item>
    ....
996 <item row="0" column="0">
997  <widget class="QLabel" name="label_57">
998   <property name="text">
999    <string>Log File</string>
1000  </property>
1001 </widget>
1002 </item>
```

Même sans connaitre finement ni le programme ni le langage dans lequel il est écrit, on en déduit que l'élément où l'on saisit le chemin est appelé `lineEditPathLog`. Une nouvelle recherche sur ce terme fournit l'endroit du code source qui gère l'enregistrement de cette option :

Code 2 – configmanager.cpp

```
651 registerOption("Tools/Log Paths", &BuildManager::additionalLogPaths, "", &pseudoDialog
      ↪ ->lineEditPathLog);
```

À nouveau, l'attention portée par les auteurs du code au nom des variables permet d'inférer que c'est probablement `additionalLogPaths` ↪ qui contient les chemins additionnels pour trouver les fichiers journaux.
Une nouvelle recherche sur ce nom de variable permet d'identifier deux fonctions, l'une pour savoir s'il existe un fichier journal, l'autre pour le lire, qui utilisent pour localiser le fichier à l'aide de ces chemins additionnels une fonction nommée `findFile`.

2.2. Analyse et correction de l'algorithme

Ayant identifié la portion de code pertinente, nous allons pouvoir analyser ce qui ne va pas.

Code 3 – adapté de buildmanager.cpp

```
2098 QString BuildManager::findFile(const QString &defaultName, const QStringList &
     ↪ searchPaths)
{
  //TODO: merge with findResourceFile
  QFileInfo base(defaultName);
  if (base.exists()) return defaultName;
  if (searchPaths.isEmpty()) return "";

  foreach (QString p, searchPaths) {
    QFileInfo fi;
    if (p.startsWith('/') || p.startsWith("\\\\") || (p.length() > 2 && p[1] == ':' &&
    ↪  (p[2] == '\\' || p[2] == '/'))) {
      fi = QFileInfo(QDir(p), base.fileName());
    } else {
      // ?? seems a bit weird: if p is not an absolute path, then interpret p as
    ↪ directory
      // e.g. default = /my/filename.tex
      //      p = foo
      // -->  /my/foo/filename.tex
      // TODO: do we want/use this anywere or can it be removed?
      QString absPath = base.absolutePath() + "/";
      QString baseName = "/" + base.fileName();
      fi = QFileInfo(absPath + p + baseName);
    }
    if (fi.exists()) return fi.absoluteFilePath();
  }
  return "";
}
```

Cette fonction est écrite en C++ et s'appuie sur la bibliothèque Qt, qui sert notamment à gérer l'interface graphique mais dont on n'utilise ici que des objets `QFileInfo` et `QDir`, qui servent à gérer les fichiers, ainsi que `QString` et `QStringList`. On peut envisager la même fonction en Python en s'appuyant également sur la bibliothèque Qt. Dans notre exemple il suffit d'importer `from PyQt5.QtCore import QFileInfo, QDir` : en effet, en Python `QString` et `QStringList` ne sont pas utiles puisque le langage Python fournit déjà des objets chaines et listes équivalents.

```
def findFile(defaultName, searchPaths) :
    base = QFileInfo(defaultName)
    if base.exists() : return defaultName
    if searchPaths == [] : return ""

    for p in searchPaths :
        if p.startswith('/') or p.startswith(r"\\") or (len(p) > 2 and p[1] == ":" and
    ↪  (p[2] == "\\" or p[2] == "/")) :
            fi = QFileInfo(QDir(p), base.fileName())
        else :
            absPath = base.absolutePath() + "/"
            baseName = "/" + base.fileName()
            fi = QFileInfo(absPath + p + baseName)
```

```
        if fi.exists() : return fi.absoluteFilePath()
    return ""
```

Nonobstant la gestion particulière des chemins relatifs, qui n'est pas en cause dans notre problème, on constate que l'algorithme mis en œuvre est très simple : on sélectionne le premier fichier candidat qui existe réellement, en commençant par le dossier de base puis en explorant les chemins additionnels dans l'ordre où ils sont enregistrés. Ceci explique entièrement le comportement observé.

Il suffit de modifier cette fonction pour parcourir systématiquement tous les fichiers candidats et retenir celui qui a été le plus récemment modifié : c'est très probablement celui qui résulte de la dernière compilation par `xelatex` et c'est donc celui qui nous intéresse.

La fonction `findFile` est peut-être utilisée dans d'autres endroits du programme que ceux que nous avons analysés; il ne faut pas modifier la validité ni le comportement de ces autres parties du programme. Pour cela, on ajoute à `findFile` un paramètre `mostRecent` facultatif et dont la valeur par défaut lui conserve son comportement antérieur.

```
def findFile(defaultName, searchPaths, mostRecent = False) :
    base = QFileInfo(defaultName)
    mr = None
    if base.exists() :
        if mostRecent :
            mr = QFileInfo(base)
        else :
            return defaultName

    for p in searchPaths :
        if p.startswith('/') or p.startswith(r"\\") or (len(p) > 2 and p[1] == ":" and
    ↪  (p[2] == "\\" or p[2] == "/")) :
            fi = QFileInfo(QDir(p), base.fileName())
        else :
            absPath = base.absolutePath() + "/"
            baseName = "/" + base.fileName()
            fi = QFileInfo(absPath + p + baseName)
        if fi.exists() :
            if mostRecent :
                if mr is None or mr.lastModified() < fi.lastModified() :
                    mr = fi
            else :
                return fi.absoluteFilePath()
    if mostRecent and mr is not None :
        return mr.absoluteFilePath()
    else :
        return ""
```

2.3. Activité expérimentale

Sans prendre la peine de télécharger tout le code source de TeXStudio ni de faire du C++, il est possible faire ces modifications sur la version Python de la fonction et de les tester à l'aide d'un dispositif ad hoc. Pour simplifier, on peut créer un fichier `exemple.log` dans le dossier temporaire `/tmp` et un autre dans `/tmp/_build`. Sous Linux, la commande système `touch` permet de changer la date de modification d'un fichier. On peut alors tester, en fonction des dates des fichiers, quels sont les noms de fichier renvoyés par `findFile("/tmp/exemple.log", ["./_build"], False)` et `findFile("/tmp/exemple.log", ["./_build"], True)`.

3. Difficulté supplémentaire en C++

Les modifications faites dans la version Python sont aisées. Pour modifier TeXStudio, il faut faire la même chose en C++. On est alors confronté à la nécessité de gérer la mémoire nous-même, tandis qu'en Python, la durée de vie des objets successivement désignés par `mr` est gérée automatiquement.

En C++, le caractère facultatif du nouveau paramètre `mostRecent` ainsi que sa valeur par défaut sont déclarés dans une autre partie du programme, le fichier d'en-tête. Outre cette différence cosmétique, on doit cette fois créer (**new**) et détruire (**delete**) explicitement les objets désignés par `mr`. Négliger de détruire les objets conduirait à une perte de mémoire progressive au cours de l'exécution du programme, par l'occupation continue des objets devenus inutiles.

```
QString BuildManager::findFile(const QString &defaultName, const QStringList &
    ↪ searchPaths, bool mostRecent)
{
  QFileInfo base(defaultName);
  QFileInfo* mr = nullptr;
  if (base.exists()) {
    if (mostRecent)
      mr = new QFileInfo(base);
    else
      return defaultName;
  }

  foreach (QString p, searchPaths) {
    QFileInfo fi;
    if (p.startsWith('/') || p.startsWith("\\\\") || (p.length() > 2 && p[1] == ':' &&
    ↪  (p[2] == '\\' || p[2] == '/'))) {
      fi = QFileInfo(QDir(p), base.fileName());
    } else {
      QString absPath = base.absolutePath() + "/";
      QString baseName = "/" + base.fileName();
      fi = QFileInfo(absPath + p + baseName);
    }
    if (fi.exists()) {
      if (mostRecent) {
        if (mr == nullptr || mr->lastModified() < fi.lastModified()) {
          if (mr != nullptr)
            delete mr;
          mr = new QFileInfo(fi);
        }
      } else
        return fi.absoluteFilePath();
    }
  }
  if (mostRecent && mr != nullptr) {
    QString result = mr->absoluteFilePath();
    delete mr;
    return result;
  } else {
    return "";
  }
}
```