OUTILS DE DEVELOPPEMENT

Plan

1. Mise en forme des sources
2. Structure d'un programme
3. Profileur
4. Traçage, temps
5. Débogueur
6. Gestion des versions
7. Fichiers multiples

Mise en forme des sources

L'utilitaire `cb` ou l'utilitaire `indent` ajustent espaces et indentations.

```
#include <stdio.h>

  long fact(long n) { if (n == 0) return 1;/* car 0! = 1 */
  return n*fact(n-1);/* recursif*/}

  void main() {
  printf("Fact(%ld) = %ld\n",5,fact(5));
  printf("Fact(%ld) = %ld\n",10,fact(10));}
```

devient

```
#include <stdio.h>

  long
  fact (long n)
  {
    if (n == 0)
      return 1;                    /* car 0! = 1 */
    return n * fact (n - 1);       /* recursif */
  }

  void
  main ()
  {
    printf ("Fact(%ld) = %ld\n", 5, fact (5));
    printf ("Fact(%ld) = %ld\n", 10, fact (10));
  }
```

De nombreuses options existent, en vue d'obtenir un aspect formalisé.

`cb` a les options

- `-s` pour utiliser le style défini par Kernighan et Ritchie
- `-j` pour reformer (joindre) des lignes éclatées
- `-l`*longueur* pour éclater des lignes trop longues

`indent` par défaut écrit dans le style GNU. Quelques options

- **`-br`** accolades en fin de ligne;
- **`-bl`** accolades sur nouvelle ligne;
- **`-ce`** un **`else`** sur même ligne;
- **`-nce`** un **`else`** sur nouvelle ligne;
- **`-pcs`** un espace entre nom de fonction et la parenthèse (**`-npcs`** n'en met pas)

Structure d'un programme

Un tel utilitaire affiche

- l'arbre des appels des fonctions;
- le type des objets;
- la localisation de l'appel dans le fichier.

Ce n'est pas un "navigateur".
Exemple: utilitaires `cflow` et `calls`.

Profileur

Utilitaire pour

- analyser les performances et localiser les points de passage fréquents (goulots d'étranglement);
- déterminer la fréquence et interdépendance des appels.

Exemple: utilitaire `prof` ou `gprof`.

En trois phases (le programme exemple est `profiler.c`).

- Compilation avec une option spécifique

  ```
  gcc -o profiler -pg profiler.c
  ```

- Exécution du programme

  ```
  profiler
  ```

  Ceci crée un fichier binaire de données, de nom `gnom.out` par défaut.

- Lancement du programme de profilage (`gprof`)

  ```
  gprof profiler gmon.out > profiler.sortie
  ```

  La sortie est volumineuse, il est recommandé de la rediriger.

Programme exemple (`profiler.c`)

```
int prod (int x, int y) {
  return x * y;
}

int fact (int x) {
  return (x == 0) ? 1 : prod (x, fact (x - 1));
}

int fibo (int x) {
  if (x <= 1)
    return x;
  return fibo(x-1)+fibo(x-2);
}

void boucler () {
  int j;
  for (j = 0; j < 1000000; j++);
}

main ()
{
  int i;
  int y = 0;
  boucler ();
  for (i = 1; i <= 900; i++)
    y += fact (i);
  fibo(30);
}
```

Première partie de la sortie, un profil plat.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 68.03      1.00     1.00                             mcount
 25.85      1.38     0.38        1 380000.00 380000.00  fibo
  2.72      1.42     0.04   405450     0.10     0.10  prod
  2.04      1.45     0.03      900    33.33    77.78  fact
  1.36      1.47     0.02        1 20000.00 20000.00  boucler
  0.00      1.47     0.00        1     0.00 470000.00  main
```

`time`
: est la proportion du temps total pris par la fonction (mcount compte).

`cumulative seconds`
: somme partielle du temps par les fonctions au-dessus dans la liste.

`self seconds`
: temps pris par la fonction elle-même.

`calls`
: nombre d'appels de cette fonction

`self us/call`
: temps moyen (en microsecondes) par appel de la fonction

`total us/call`
: temps moyen (en microsecondes) par appel de la fonction et de ses descendantes.

Le graphe des appels `call graph` est plus explicite.

```
index % time    self  children    called     name
                0.00    0.47       1/1           _start [2]
[1]    100.0    0.00    0.47       1         main [1]
                0.38    0.00       1/1           fibo [3]
                0.03    0.04     900/900         fact [4]
                0.02    0.00       1/1           boucler [6]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.47                 _start [2]
                0.00    0.47       1/1           main [1]
-----------------------------------------------
                             2692536             fibo [3]
                0.38    0.00       1/1           main [1]
[3]     80.9    0.38    0.00       1+2692536 fibo [3]
                             2692536             fibo [3]
-----------------------------------------------
                              405450             fact [4]
                0.03    0.04     900/900         main [1]
[4]     14.9    0.03    0.04     900+405450  fact [4]
                0.04    0.00  405450/405450      prod [5]
                              405450             fact [4]
-----------------------------------------------
                0.04    0.00  405450/405450      fact [4]
[5]      8.5    0.04    0.00  405450         prod [5]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[6]      4.3    0.02    0.00       1         boucler [6]
-----------------------------------------------
```

Les fonction des lignes au-dessus de l'entrée d'une fonction sont *appelantes*, et en dessous les fonctions appelées. Par exemple, **`fact`** est appelée par **`fact`** et **`main`**, et appelle **`fact`** et **`prod`**. **`<spontaneous>`** signifie que le parent n'a pas pu être déterminé.

Pour la fonction elle-même,

`time`
: proportion du temps pris par la fonction et ses enfants. Ainsi, les 100 pourcents de `main` se partagent en 80.9 (`fibo`) + 14.9 (`fact`) + 4.3 (`boucler`)

`self`
: temps total pris par cette fonction

`children`
: temps contribué par les enfants (p.ex `fact` prend 0.03 et `prod` 0.04)

`called`
: nombre d'appels de cette fonction (suivi du nombre d'appels récursif s'il y a lieu). `fibo` n'a été appelée qu'une fois, et `fact` 900 fois.

Pour les parents de la fonction,

`self`
: est la contribution de la fonction au temps total du parent: dans [3], `fibo`.

`children`
: la contribution des enfants de la fonction au temps total du parent.

`called`
: le nombre de fois que ce parent a appelé la fonction, et le nombre total d'appels (appels récursifs exclus). Ainsi, dans [5], `prod` a été appelé 405450 fois par `fact`, sur un total de 405450 appels; donc n'a été appelé que par `fact`.

Pour les enfants de la fonction,

`self`
: est le temps contribué par cet enfant à la fonction: `prod` a contribué 0.04 sec à `fact`.

`children`
: le temps contribué par les enfants des enfants: dans [1], l'enfant de `fact` (à savoir `prod`) a contribué 0.04 sec.

`called`
: le nombre d'appel de cet enfant par la fonction, comparé au nombre total d'appels.

Traçage, temps

Le traçage d'un programme permet de suivre diverses fonctions ou variables à travers l'exécution d'un programme. C'est une version allégée d'un débogueur.

Exemples

`ctrace` prend en option le nom des fonctions à tracer (resp. à ne pas tracer), et un fichier *source*, et crée un autre fichier source.

```
ctrace -f fact profiler.c > profiler_t.c
```

Le nouveau fichier est compilé et exécuté.

`strace` et `truss` affichent les appels système exécutés durant le fonctionnement d'un programme.

`time` lance la commande donnée en argument, et fournit trois résultats sur la sortie erreur standard:

- le temps total écoulé (`real`);
- le temps passé à exécuter le programme (`user`);
- le temps pris par le noyau (`sys`)

Par exemple,

```
time profiler
```

donne :

```
1.48 user 0.01 system 0:01.49 elapsed
```

Débogueur

Un *débogueur* est un utilitaire d'aide pour la recherche d'erreurs de programmation.

Il permet de dérouler un programme pas à pas, et d'inspecter les valeurs de variables (et de la pile) au cours de l'exécution.
La notion principale est le *point d'arrêt*. Il désigne un endroit du programme source où l'exécution peut être suspendue en vue de l'inspection.

Utilitaires :

- `gdb` et sa version X-Window appelée `xxgdb`.
- `dbx` et sa version X-Window `xdbx`.

Avant utilisation, compiler avec option `-g`.

```
gcc -g -o fact fact.c
```

Commandes principales (réalisées par des boutons):

`file` : charge le fichier exécutable;
`list` : montre le fichier source;
`break`: place des points d'arrêt (break point);
`run` : lance le programme. Il s'arrête au premier point d'arrêt;
`next` : poursuit jusqu'au point d'arrêt suivant;
`print` : imprime la valeur de l'expression sélectionnée;
`print*`: imprime la valeur à l'adresse;
`stack` : affiche la pile d'exécution;
`up, down` : pour naviguer dans la pile et examiner les environnements.

```
(xxgdb) No symbol table is loaded.  Use the "file" command.
file fact
(xxgdb) break 6
Breakpoint 1 at 0x8048477: file fact.c, line 6.
(xxgdb) break 14
Breakpoint 2 at 0x80484a7: file fact.c, line 14.
(xxgdb) print fact
$1 = {long int (long int)} 0x8048474 <fact>
(xxgdb) print *fact
$2 = {long int (long int)} 0x8048474 <fact>
info stack
#0  fact (n=3) at fact.c:6
#1  0x804848e in fact (n=4) at fact.c:8
#2  0x804848e in fact (n=5) at fact.c:8
#3  0x80484ae in main () at fact.c:14
#4  0x804842b in ___crt_dummy__ ()
```

Gestion des versions

Un *gestionnaire de versions* doit permettre

- de gérer plusieurs versions d'un même document;
- de garantir l'intégrité des versions;
- de fusionner des versions.

Utilitaires :

- SCCS (Source Code Control System)
- RCS (Revision Control System)

Les deux sont semblables. RCS est plus récent. Il

- conserve et restitue les multiples versions d'un texte. RCS sauvegarde toutes les versions antérieures de manière efficace. Un changement ne détruit plus une version précédente. Les versions sont accessibles selon différents critères, et notamment le numéro de version.
- maintient un historique complet des changements. En plus du texte des modifications, RCS conserve l'auteur, la date et l'heure, et un message (à fournir par l'auteur) résumant les changements.
- gère les conflits d'accès. Lorsque plusieurs programmeurs veulent modifier la même version, RCS émet un message et empêche la modification.

Les acteurs

Le système RCS opère au moyen de deux commandes.

`ci` pour “check-in” enregistre une nouvelle version d’un fichier;

`co` pour “check-out” récupère une version d’un fichier.

La gestion des versions est faite dans un fichier créé soit explicitement, soit implicitement au premier enregistrement.

Pour séparer le fichier des versions du document, il est sage de le placer dans un répertoire séparé. RCS le cherche par défaut dans le répertoire `./RCS`.

```
$ more txt.c            <-- Premier fichier
#include <stdio.h>

main(int argc, char* argv[])
{
  printf("Bonjour !\n");
}
$
```

Enregistrement

```
$ ci txt.c               <-- Premier enregistrement
RCS/txt.c,v  <--  txt.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> Source du programme "Hello, world"
>> .
initial revision: 1.1
done
$ ls -l                  <-- le fichier txt.c a disparu !
total 8
drwxr-xr-x   2 jb       users        1024 Nov 23 12:28 RCS
-rwxr-xr-x   1 jb       users        3926 Nov 23 12:27 txt
$
```

`RCS/txt.c,v` est le fichier de conservation des versions.

Il est créé au premier enregistrement.

Il demande un message d'identification.

Le fichier enregistré est supprimé.

Extraction

```
$ co txt.c              <-- extraction de txt.c
RCS/txt.c,v  -->  txt.c
revision 1.1
done
$ ls -l
total 9
drwxr-xr-x   2 jb       users        1024 Nov 23 12:28 RCS
-rwxr-xr-x   1 jb       users        3926 Nov 23 12:27 txt
-r--r--r--   1 jb       users          78 Nov 23 12:36 txt.c
$
```

`co` pour "check-out" extrait le codument du fichier des versions.
Par défaut, la version la plus récente.
Sans option, le fichier extrait est *uniquement consultable* et non modifiable.
Avec l'option `-l` pour "locked", le fichier est *verrouillé* à son extraction. Seul le verrouilleur peut le modifier et enregistrer une nouvelle version.

```
$ co -l txt.c
RCS/txt.c,v  -->  txt.c
revision 1.1 (locked)
done
$ ls -l
total 9
drwxr-xr-x   2 jb       users        1024 Nov 23 12:38 RCS
-rwxr-xr-x   1 jb       users        3926 Nov 23 12:27 txt
-rw-r--r--   1 jb       users          78 Nov 23 12:38 txt.c
$
```

Nouvelles versions

```
$ more txt.c            <-- modification de txt.c
#include <stdio.h>

main(int argc, char* argv[])
{
  printf("Bonjour, monde !\n");
}
$ ci txt.c              <-- Enregistrement version 2
RCS/txt.c,v  <--  txt.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Change' "Bonjour, monde"
>> .
done
$ co -l txt.c           <-- Extraction version 2
RCS/txt.c,v  -->  txt.c
revision 1.2 (locked)
done
$ ...
$ ci txt.c
RCS/txt.c,v  <--  txt.c
new revision: 1.3; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> Change' "monde" en "tout le monde"
>> .
done
$
```

L'enregistrement d'une nouvelle version s'accompagne

- de l'incrément du numéro de version;
- de l'adjonction d'un texte (à écrire) décrivant les changements apportés.

Récupération et changement d’anciennes versions

```
$ co -l1.2 txt.c             <-- extraction de 1.2
RCS/txt.c,v  -->  txt.c
revision 1.2 (locked)
done
$ more txt.c                 <-- modification
#include <stdio.h>

main(int argc, char* argv[])
{
  if (argc != 2)
    fprintf(stderr,"%s: Usage: %s <qui>\n",argv[0],argv[0]);
  printf("Bonjour,  %s !\n", argv[1]);
}
$ gcc -o Bonjour txt.c
$ Bonjour
Bonjour: Usage: Bonjour <qui>
Bonjour,  (null) !
$ Bonjour Max
Bonjour,  Max !
$ ci txt.c                    <-- Enregistrement
RCS/txt.c,v  <--  txt.c
new revision: 1.2.1.1; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> Nouvelle version, demande le nom sur la
>> ligne de commande
>> .
done
$
```

L’extraction d’une version par son numéro de version.

Le nouveau numéro est construit en ajoutant à l’ancien l’extension `1.1`.

Historique des versions

```
$ rlog txt.c                   <-- Historique des versions
RCS file: RCS/txt.c,v
Working file: txt.c
head: 1.3
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 5;     selected revisions: 5
description:
Source du programme "Hello, world"
----------------------------
revision 1.3
date: 1997/11/23 11:45:41;  author: jb;  state: Exp;  lines: +1 -1
Change' "monde" en "tout le monde"
----------------------------
revision 1.2
date: 1997/11/23 11:42:25;  author: jb;  state: Exp;  lines: +1 -1
branches:  1.2.1;
Change' "Bonjour, monde"
----------------------------
revision 1.1
date: 1997/11/23 11:28:03;  author: jb;  state: Exp;
Initial revision
----------------------------
revision 1.2.1.2
date: 1997/11/23 11:59:34;  author: jb;  state: Exp;  lines: +3 -1
Avec sortie en cas d'erreur
----------------------------
revision 1.2.1.1
date: 1997/11/23 11:54:34;  author: jb;  state: Exp;  lines: +3 -1
Nouvelle version, demande le nom sur la
ligne de commande
==================================================================
```

Chaque version est identifiée par

- son numéro
- la date et l'heure de l'enregistrement
- l'auteur
- l'état (`Exp` par défaut, qui veut dire expérimental)
- le nombre de lignes ajoutées et enlevées
- le texte inclu par l'auteur.

Mots-clé

Des *mots-clé* permettent d'inclure dans le fichier extrait par `co` des informations concernant la version. La plus complète est `$Header$`.

```
$ more txt.c                      <-- quelques revisions plus loin
static char rcsid [] = "$Header$";
#include <stdio.h>

main(int argc, char* argv[])
{
  printf("Bonjour, tout le monde !\n");
}
$
```

Le mot-clé `$Header$` sera repéré par la commande `co` qui lui substitue le véritable en-tête.

```
$ ci txt.c                        <-- enregistrement
RCS/txt.c,v  <--  txt.c
new revision: 1.4; previous revision: 1.3
enter log message, terminated with single '.' or end of file:
>> avec en-tete
>> .
done
$ co -l txt.c                     <--- extraction avec en-tete
RCS/txt.c,v  -->  txt.c
revision 1.4 (locked)
done
$ more txt.c                      <--- en effet
static char rcsid [] = "$Header: /home/jb/revisions/RCS/txt.c\
,v 1.4 1997/11/23 12:16:09 jb Exp jb $";
#include <stdio.h>
...
$
```

L'utilitaire **ident** recherche toutes les instances d'un motif **$cle:texte$** et les affiche.

```
$ ident txt.c
txt.c:
     $Header: /home/jb/revisions/RCS/txt.c,v 1.4 1997/11/23\
 12:16:09 jb Exp jb $
$ gcc -o Bonjour txt.c
$ Bonjour
Bonjour, tout le monde !
$ ident Bonjour                 <--- meme dans executable
Bonjour:
     $Header: /home/jb/revisions/RCS/txt.c,v 1.4 1997/11/23\
 12:16:09 jb Exp jb $
$
```

Les informations sont

- le fichier des versions
- le numéro de la version
- la date et l'heure
- l'auteur
- l'état : `Exp` (expérimental), `Stab` (stable), `Rel` (distribué)
- le verrouilleur.

Fichiers multiples

Un programme (important) est composé de plusieurs (dizaines) de fichiers compilés indépendamment.

Un seul contient une fonction `main()`.

Le lien entre les modules est fait par l'éditeur de lien.

```
gcc -g -o hal nx.c trad.c hal.c
```

Fichier `hal.c` contenant `main()` :

```
#include "trad.h"
  typedef char mot[12];

  main () {
    mot x = "IBM";
    mot y;
    recule(y,x);
    printf("Avant %s, il y avait %s\n", x, y);
  }
```

Le résultat est:

```
Avant IBM, il y avait HAL
```

Fichier `trad.h` inclu par `hal.c` :

```
/* trad.h */
   #include "nx.h"

   void avance(char [], char []);
   void recule(char [], char []);
```

Fichier `nx.h` inclu par `trad.h` :

```
/* nx.h */
   #ifndef NX_H
   #define NX_H

   char suiv(char);
   char prec(char);

   #endif /* NX_H */
```

La construction

```
#ifndef NX_H
   #define NX_H
   ...
   #endif /* NX_H */
```

évite l'inclusion multiple. La règle de formation des noms est traditionnelle.

Diagramme des inclusions :

```
hal.c  <-- trad.h  <-- nx.h

   trad.c <-- trad.h (<-- nx.h)
          <-- nx.h

   nx.c   <-- nx.h
```

Fichier `trad.c` contenant les définitions de `avance()` et `recule()` :

```
/* trad.c */
  #include "nx.h"
  #include "trad.h"

  void avance(char s[], char t[]) {
    int i;
    for (i = 0; t[i] ; i++)
      s[i] = suiv(t[i]);
    s[i]=0;
  }

  void recule(char s[], char t[]) {
    int i;
    for (i = 0; t[i] ; i++)
      s[i] = prec(t[i]);
    s[i]=0;
  }
```

Fichier `nx.c` contenant les définitions de `suiv()` et `prec()` :

```
/* nx.c */
  #include "nx.h"

  char suiv(char c) {
    return c+1;
  }

  char prec(char c) {
    return c-1;
  }
```