TME1 -- Méthodologie de Déboguage

Contents

Environnement de Travail du Master SESI

Les salles de TME du Master SESI sont sous le système d'exploitation GNU / Linux (UNIX). L'essentiel de votre travail se fera en utilisant un éditeur de texte comme **`gedit`**, **`gvim`** ou **`emacs`** et un interpréteur de commande (ou *shell*) s'exécutant dans un terminal.

Si vous n'êtes pas familiers avec cet environnement, vous êtes vivement incité à lire le Guide de Survie sous UNIX

Ce TME a pour objectifs de rappeler les principales étapes du flot de compilation d'un programme et de présenter les méthodes de déboguage adaptées à chacune d'elle. Un flot de compilation transforme un ou plusieurs fichiers écrits dans un *language de programmation* comme le C ou le C++ en une suite d'instructions pour le processeur de l'ordinateur codé en language binaire (ou plus simplement de *fichier binaire*).

Étapes du Flot de Compilation

Dans la figure 1, il faut bien distinguer les **fichiers** (points 1, 3 et 5), des **programmes** effectuant les transformations (points 2, 4 et 6).

- **2 -- Compilation.** Transforme le code source (écrit et compréhensible par un humain) en un fichier binaire *incomplet*, ce n'est qu'une partie du programme complet.

  Cette opération est effectuée par un *compilateur*, usuellement `gcc` pour le language C et `g++` pour le C++.

- **4 -- Édition de liens.** Combine les différents fichiers binaires *partiels* pour produire un fichier binaire *complet*, c'est à dire un programme exécutable. On donnera aussi à cette étape la liste des librairies dont dépend le binaire.

- **6 -- Éxecution du programme.**

  Résolution des librairies dynamiques, quand un programme utilise un composant comme QT, le binaire fera référence aux librairies dynamiques fournies par QT. Il existe un mécanisme spécifique aux binaires leur permettant de retrouver ces librairies au moment de l'exécution, il s'appuie sur la variable d'environnement `LD_LIBRARY_PATH`.

  Cet aspect n'est pas couvert par ce TME, cependant, les outils de déboguages associés sont présentés dans Librairies Dynamiques.

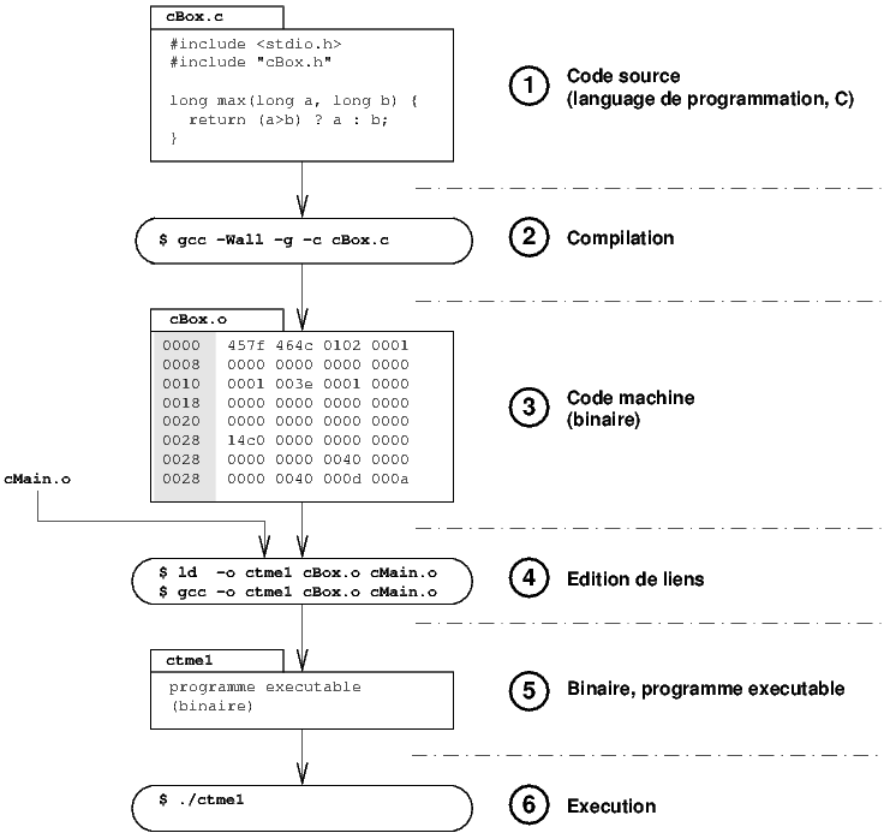

Figure 1: Figure 1 -- Flot de compilation

Chacune de ces différentes étapes est susceptible de générer des erreurs. Pour déboguer efficacement, il est nécessaire de bien identifier dans quelle étape l'erreur s'est produite.

Automatisation de la Compilation des Binaires

La figure 1 donne le détail des commandes qui sont exécutées pour créer un programme complet. **Cependant, lorsque le programme se compose de dizaines de fichiers sources, il n'est plus praticable de lancer ces commandes une par une à la main.**

Pour automatiser l'appel des commandes `gcc` / `g++` et `ld`, nous utiliserons `cmake` et `make`.

1. `cmake` prend en entrée le fichier `CMakeLists.txt` à partir duquel il génère un fichier `Makefile`. On appelle `cmake` une seule fois au début du projet.
2. `make` prend en entrée le fichier `Makefile` et lance les appels à `gcc`, `g++` et `ld`. On doit appeler `make` *à chaque* fois que les fichiers contenant le code source sont modifiés.

Ce qui donnera la séquence de commandes suivantes:

```
etudiant@pc:dir> mkdir build                    # Une seule fois.
etudiant@pc:build> cd build                     # Une seule fois.
etudiant@pc:build> cmake ../src                 # Une seule fois.
etudiant@pc:build> make install
etudiant@pc:build> # Modification du code...
etudiant@pc:build> make install
etudiant@pc:build> # Modification du code...
etudiant@pc:build> make install
etudiant@pc:build> # Modification du code...
etudiant@pc:build> make install
```

Note

Reportez vous à l'Introduction pour une description plus complète de `cmake`, `make` et de la procédure de compilation.

Débogage des Différentes Étapes du Flot

Erreurs liées à la compilation (2)

La syntaxe C/C++ dans un ou plusieurs de vos fichiers source n'est pas bonne. Pour utiliser une analogie, vous avez fait des fautes d'orthographe.

```
ego@home:build> make
Scanning dependencies of target ctme1
[ 25%] Building C object CMakeFiles/ctme1.dir/cBox.c.o
src/cBox.c: In function 'swap':
src/cBox.c:9:19: error:
                                'aa' undeclared (first use in this function)
 { long tmp = *a; *aa = *b; *b = tmp; }
                   ^
src/cBox.c:9:19: note:
each undeclared identifier is reported only once for each function it appears in
make[2]: *** [CMakeFiles/ctme1.dir/cBox.c.o] Error 1
make[1]: *** [CMakeFiles/ctme1.dir/all] Error 2
make: *** [all] Error 2
ego@home:build>
```

A ce stade, l'outil de correction est l'éditeur de texte. Le compilateur vous indique toujours le fichier et le numéro de ligne où s'est produite l'erreur. **Lire attentivement le message** puis se reporter à l'emplacement indiqué. Dans l'exemple ci-dessus, l'erreur se trouve à la ligne **9** dans le fichier **cBox.cpp** (la variable **aa** n'a pas été déclarée).

Note

Certaines erreurs, comme par exemple, l'oubli d'une accolade fermante `}`, génèrent des erreurs en cascade. C'est à dire que du code syntaxiquement correct mais situé *après* cette première erreur devient faux et entraîne la génération de messages.
Il faut donc toujours corriger les erreurs en partant de la première et pas de la dernière affichée sur le terminal.

Erreurs liées à l'édition de liens (4)

Les seules erreurs pouvant se produire à ce niveau sont l'absence d'un symbole. Un *symbole* est une variable ou une fonction. Exemple d'erreur:

```
ego@home:build> make
Scanning dependencies of target ctme1
[ 25%] Building C object CMakeFiles/ctme1.dir/cBox.c.o
Linking C executable ctme1
CMakeFiles/ctme1.dir/cBox.c.o: In function `boxIntersection':
cBox.c:(.text+0x454): undefined reference to `boxIntersect'
CMakeFiles/ctme1.dir/cMain.c.o: In function `testFunction':
cMain.c:(.text+0x12d): undefined reference to `boxIntersect'
collect2: error: ld returned 1 exit status
make[2]: *** [ctme1] Error 1
make[1]: *** [CMakeFiles/ctme1.dir/all] Error 2
make: *** [all] Error 2
ego@home:build>
```

Les lignes intéressantes sont celles indiquant *undefined reference to boxIntersect*. Puisque l'étape de compilation a passé, la fonction `boxIntersect()` a été déclarée. Ce que nous dit l'éditeur de lien `ld`, c'est que le **corps** ou la **définition** de la fonction n'est présent nulle part dans le code source.

Ce type d'erreur peut aussi se produire si un fichier objet (étape **3**) a été oublié dans la liste des fichiers composant le binaire complet. Dans ce cas c'est une erreur au niveau du `Makefile` (`make`) ou `CMakeLists.txt` (`cmake`).

Erreurs à l'exécution du programme (6)

Deux cas peuvent se présenter:

a. Le programme s'arrête brutalement suite à une faute mémoire ou une exception. Dans 99% des cas nous sommes face à un problème lié à un pointeur et il faut déboguer la mémoire du programme.

b. Le programme s'exécute jusqu'au bout, mais le résultat escompté n'est pas bon, c'est à dire qu'il ne passe pas le jeu de test. La méthode est de générer une trace d'exécution du programme (avec des `printf()`) et de vérifier les calculs effectués par rapport au jeu de test.

Déboguer la mémoire d'un programme (6.a)

Tout d'abord, il faut autoriser le système d'exploitation à générer une image mémoire (*core dump*) du programme au moment où celui-ci est interrompu:

```
ego@home:work> ulimit -c unlimited
```

Puis relancer le programme:

```
ego@home:work> ../install/bin/ctme1
boxCreateEmpty()
boxCreate() "b1" [15 10 30 20]
bash: segmentation fault (core dumped)  ../install/bin/ctme1
ego@home:work> ls -l
-rw------- 1 ego user 397312 Sep 19 18:12 core.13784
```

On obtient un fichier `core.PID` (où `PID` est le numéro de processus du programme). Nous pouvons alors utiliser `gdb` qui permet de faire une analyse post-mortem du programme:

```
ego@home:work> gdb ../install/bin/ctme1 core.18393
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-94.el7
Copyright (C) 2013 Free Software Foundation, Inc.
[New LWP 18393]
Core was generated by `../install/bin/ctme1'.
Program terminated with signal 11, Segmentation fault.
#0  0x00007ff346075fa3 in __strncpy_sse2_unaligned () from /lib64/libc.so.6
Missing separate debuginfos, use: debuginfo-install glibc-2.17-196.el7.x86_64
(gdb) up
#1  0x0000000000400e48 in boxCreate (name=0x4019cf "b1", x1=15, y1=10, x2=30, y2=
    at /dsk/l1/jpc/cours/M1-MOBJ/TME/1/corrige/src/cBox.c:50
50      strncpy( box->name_, name, 1023 );
(gdb)
```

Pour comprendre ce que vous permet `gdb`, il faut bien saisir ce qu'est un programme en cours d'exécution. A tout moment, c'est un ensemble d'appels de fonctions imbriqués qui forment une pile (ou *stack*). La fonction `main()` se trouvant à la base de cette pile. Le fichier `core.PID` est l'image de cette pile au moment exact où le programme fait une erreur provoquant son arrêt par le système d'exploitation. `gdb` vous permet d'inspecter la pile de fonctions, donc de savoir exactement où (dans le programme source), l'erreur s'est produite ainsi que de connaître la valeur de chaque variable dans les fonctions.

Par exemple la pile ici est:

```
(gdb) backtrace
#0  0x00007ff346075fa3 in __strncpy_sse2_unaligned () from /lib64/libc.so.6
#1  0x0000000000400e48 in boxCreate (name=0x4019cf "b1", x1=15, y1=10, x2=30, y2=
    at cBox.c:50
#2  0x00000000004015a0 in main (argc=1, argv=0x7ffcbcad2f68 "iQ\255\274\374\177")
    at cMain.c:32
(gdb)
```

Note

Correspondance entre un programme et un core. Un fichier `core` correspond exactement au programme qui l'a généré.
A chaque fois que vous recompilez, vous devez recréer le `core`. `gdb` vous prévient si le `core` ne correspond pas au programme.

Note

Pour pouvoir utiliser `gdb`, il est impératif de passer l'argument `-DCMAKE_BUILD_TYPE=Debug` à `cmake`. Il active le mode de *debug* du compilateur (flag `-g`). Cf. Introduction.

Quelques commandes de `gdb`

`> gdb [program] [core]`	Lance `gdb` sur le binaire `program` en utilisant l'image mémoire `core` Une fois lancé, `gdb` passe en mode interactif et affiche le *prompt* `(gdb)` en l'attente d'une commande
commande	**action**
`up`	Remonte d'un niveau dans la pile des appels de fonctions
`down`	Descend d'un niveau dans la pile des appels de fonctions
`list`	Affiche le code source correspondant au point ou l'on se trouve dans le programme
`backtrace`	Affiche toute la pile de fonctions
`print [variable]`	Affiche le contenu d'une variable de la fonction dans laquelle on se trouve

Note

Une fois le débogage terminé, ne pas oublier d'effacer tous les fichiers *core* générés, ils deviennent rapidement encombrants...

Déboguer l'exécution d'un programme (6.b)

Il s'agit de comparer la trace d'exécution obtenue avec celle issue d'un jeu de test. L'objectif du jeu de test est de vérifier de la façon la plus complète possible les différentes fonctionalités du programme. Le jeu de test et la trace associée sont fournis ci-après.

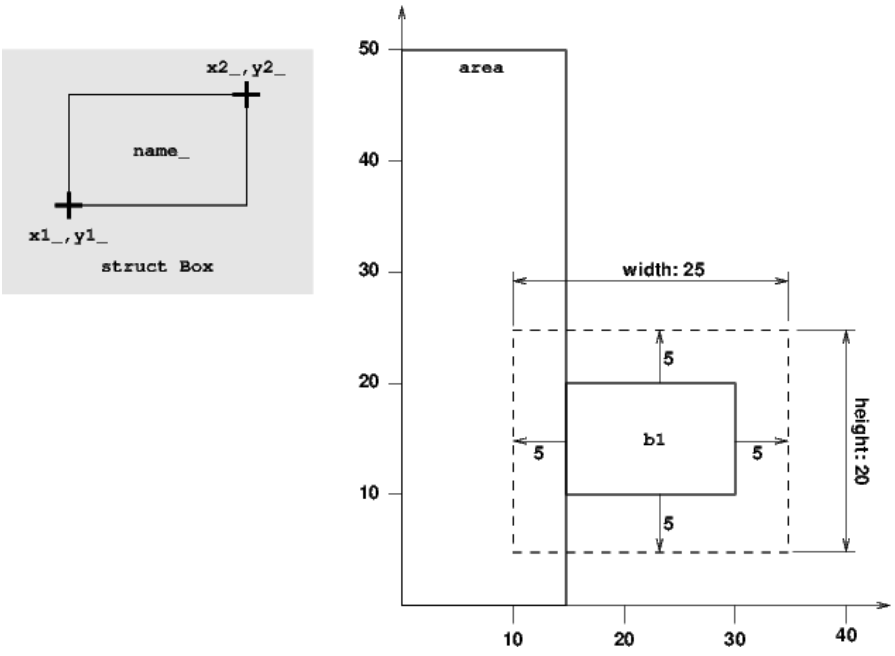

Figure 2: Figure 2 -- Jeu de Test

Trace de référence du jeu de test:

```
ego@home:work> ../install/bin/ctme1
boxCreateEmpty()
boxCreate() "b1" [15 10 30 20]
Allocated boxes: 2

Test 1
<"b1" [15 10 30 20]>
  + testFunction() called.
boxCreate() "area" [0 0 15 50]
  | Intersection between:
  |   <"area" [0 0 15 50]>
  |   <"b1" [15 10 30 20]>
  | Gives:
  |   No intersection.
boxDelete() "area"

Test 2
<"b1" [10 5 35 25]>
  + testFunction() called.
boxCreate() "area" [0 0 15 50]
  | Intersection between:
  |   <"area" [0 0 15 50]>
  |   <"b1" [10 5 35 25]>
  | Gives:
boxCreateEmpty()
  |   <"area.b1" [10 5 15 25]>
boxDelete() "area.b1"
boxDelete() "area"

Test 3
+ Box "b1" empty:  0
| Box "b1" width:  25
| Box "b1" height: 20
+ Box "b1" empty:  1 (inflated -11)
| Box "b1" width:  0
| Box "b1" height: 0

boxDelete() "b1"
boxDelete() "Unknown"
Allocated boxes: 0
```

Sources du Programme à Déboguer

L'objectif de ce TME est de corriger les erreurs de tout type dont le programme suivant est truffé...

La configuration de l'environnement de compilation est expliquée dans l'Introduction.

- cBox.h
- cBox.c
- cMain.c
- CMakeLists.txt

Librairies Dynamiques

Une *librarie* est simplement un ensemble de fichiers binaires `*.o` rassemblés au sein d'un unique fichier (`.a` ou `.so`) fourni par une tierce partie. Par exemple Qt fournit `libQtCore.so`, `libQtGui.so`, ...

Il existe deux types de libraries :

1. Les librairies *statiques* (dont l'extension est `.a`). Ces librairies sont **incorporées** dans le binaire au moment de l'édition de liens (4). Cela fait grossir le binaire mais celui-ci est complet et aucune autre opération ne sera nécessaire pour l'utiliser.

2. Les librairies *dynamiques* (dont l'extension est `.so`) sont justes conservées sous forme de référence au niveau du binaire (i.e. un genre de pointeur). Ce qui veut dire que le binaire du programme est bien plus léger, mais qu'au moment de l'exécution, celui-ci devra être capable de retrouver le `.so` de la librairie.

   A l'instar de la variable `PATH`, qui fournit une liste ordonnée de répertoires où trouver une commande (i.e. un programme, un binaire), il existe une variable `LD_LIBRARY_PATH` qui donne la liste ordonnée des répertoires dans lesquels rechercher une librairie dynamique.

Vérifier l'Édition de Liens -- `ldd`

La commande **`ldd`**, appliquée à un binaire vous permet de voir comment les librairies dynamiques sont trouvées (ou non) :

```
ego@home:bin> echo $LD_LIBRARY_PATH
/usr/lib64
ego@home:bin> ldd ./tme810
  linux-vdso.so.1 =>  (0x00007ffcf8de8000)
  libunicorn.so.1 => not found
  libQtSvg.so.4 => /lib64/libQtSvg.so.4 (0x00007f855867c000)
  libQtGui.so.4 => /lib64/libQtGui.so.4 (0x00007f85579a1000)
  libQtCore.so.4 => /lib64/libQtCore.so.4 (0x00007f85574b5000)
  libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f8555b83000)
  libm.so.6 => /lib64/libm.so.6 (0x00007f8555881000)
  libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f855566b000)
  libc.so.6 => /lib64/libc.so.6 (0x00007f85552a7000)
  libXext.so.6 => /lib64/libXext.so.6 (0x00007f855263f000)
  libX11.so.6 => /lib64/libX11.so.6 (0x00007f8552301000)
  libdl.so.2 => /lib64/libdl.so.2 (0x00007f85520fc000)
  librt.so.1 => /lib64/librt.so.1 (0x00007f8551ef4000)
  /lib64/ld-linux-x86-64.so.2 (0x0000564a0dc25000)
```

`ldd` affiche à gauche le nom de la librairie recherchée (`libunicorn.so.1`) et à droite, le chemin complet dans le système de fichier. Dans l'exemple toutes les librairies sont trouvées, sauf `libunicorn.so.1` marquée `not found`.

Nous allons donc ajouter au `LD_LIBRARY_PATH` le répertoire où se trouve la librairie manquante : `/home/ego/lib64`.

```
ego@home:bin> export LD_LIBRARY_PATH /home/ego/lib64:/usr/lib64
ego@home:bin> ldd ./tme810
  linux-vdso.so.1 =>  (0x00007ffcf8de8000)
  libunicorn.so.1 => /home/ego/lib64/libunicorn.so.1 (0x00007f855c7a5000)
  libQtSvg.so.4 => /lib64/libQtSvg.so.4 (0x00007f855867c000)
  libQtGui.so.4 => /lib64/libQtGui.so.4 (0x00007f85579a1000)
  libQtCore.so.4 => /lib64/libQtCore.so.4 (0x00007f85574b5000)
```

```
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f8555b83000)
libm.so.6 => /lib64/libm.so.6 (0x00007f8555881000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f855566b000)
libc.so.6 => /lib64/libc.so.6 (0x00007f85552a7000)
libXext.so.6 => /lib64/libXext.so.6 (0x00007f855263f000)
libX11.so.6 => /lib64/libX11.so.6 (0x00007f8552301000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f85520fc000)
librt.so.1 => /lib64/librt.so.1 (0x00007f8551ef4000)
/lib64/ld-linux-x86-64.so.2 (0x0000564a0dc25000)
```

Note

Multiples versions de la même librairie dynamique: il arrive que dans un système, plusieurs versions d'une *même* librairie dynamique existent. Dans ce cas, il est *crucial* d'utiliser la version correspondant à votre binaire en positionnant de façon appropriée le `LD_LIBRARY_PATH`, et de vérifier avec **`ldd`**.

Analyse d'une librairie -- `nm` et `c++filt`

`nm` permet de connaître ce que contient une librairie. C'est à dire les variables et les fonctions qu'elle contient (type `T`) ou bien dont elle à besoin (type `U`).

Symbole : le nom d'une *variable* ou d'une *fonction* (ceci incluant les fonctions membres des classes).

Name mangling : littéralement *estropier les noms*, dans le cas d'une librairie C++, les noms des *symboles* n'apparaissent pas en clair, ils sont encodés. Par exemple `_ZN4tme13BoxC2Ev` correspond au contructeur sans argument de la classe `Box` : `tme1::Box::Box()`

```
ego@home:lib> nm libBox.so
0000000000401f8b t _Z41__static_initialization_and_destruction_0ii
00000000004028e3 t _Z41__static_initialization_and_destruction_0ii
0000000000402a51 W _ZlsRSoRKN4tme13BoxE
0000000000604318 B _ZN4tme13Box11allocateds_E
000000000040179e T _ZN4tme13Box13getAllocatedsEv
0000000000401b40 T _ZN4tme13Box7inflateEl
0000000000401b72 T _ZN4tme13Box7inflateEll
0000000000401dd2 T _ZN4tme13Box7inflateEllll
0000000000401f22 T _ZN4tme13BoxaSERKS0_
00000000004019b4 T _ZN4tme13BoxC1ERKS0_
0000000000401890 T _ZN4tme13BoxC1ERKSslll
00000000004017ac T _ZN4tme13BoxC1Ev
00000000004019b4 T _ZN4tme13BoxC2ERKS0_
0000000000401890 T _ZN4tme13BoxC2ERKSslll
00000000004017ac T _ZN4tme13BoxC2Ev
0000000000401aba T _ZN4tme13BoxD1Ev
0000000000401aba T _ZN4tme13BoxD2Ev
0000000000401c52 T _ZNK4tme13Box15getIntersectionERKS0_
0000000000401e4c T _ZNK4tme13Box5printERSo
00000000004029cc W _ZNK4tme13Box7getNameEv
0000000000401fde W _ZNK4tme13Box7isEmptyEv
00000000004029da W _ZNK4tme13Box8getWidthEv
0000000000402a16 W _ZNK4tme13Box9getHeightEv
0000000000401ba8 T _ZNK4tme13Box9intersectERKS0_
ego@home:lib> c++filt _ZN4tme13BoxC2Ev
tme1::Box::Box()
ego@home:lib> c++filt _ZN4tme13BoxC2ERKS0_
tme1::Box::Box(tme1::Box const&)
```

Documentation des Outils

Il n'est présenté dans ce TME qu'une partie très limitée de ce que peuvent faire les outils. Pour aller plus loin, vous pouvez vous référer à leur documentation complète.

Outil	Fonction	Documentation
`gcc`, `g++`	Compilateur C / C++	`man gcc` ou `info gcc`
`gdb`	Déboguage de *core dump*, exécution pas à pas	`info gdb` ou documentation gdb
ddd	Un débogueur alternatif à `gdb` doté d'une interface graphique et de capacités de tracage avancées. Pas forcément nécessaire pour analyser un simple *core dump*	`info ddd`
`nm`	Liste les symboles contenus dans une librarie dynamique	`man nm`
`c++filt`	*demangling* des symboles	`man c++filt`
`cmake`	Générateur de `Makefile`	`man cmake` ou documentation cmake
`make`	Exécution de commandes avec dépendances temporelles	`info cmake` ou `man cmake`