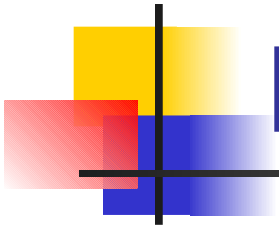
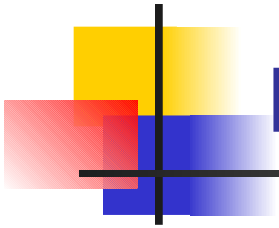
OUTILS DE DEVELOPPEMENT

- homogénéiser l'écriture
- construction des binaires
- mise au point
- gestion des versions
- gestion mémoire
- performance
- documentation
- déploiement



MAKE

- Les sources d'une application réelle :
  - plusieurs fichiers sources, pas forcément tous avec le même langage
  - plusieurs fichiers include,
  - …
- la génération, au sens large, consiste à créer plusieurs librairies et exécutables



make

- **Make** est une commande permettant de générer une application d'après la description qui est faite dans un fichier appelé *makefile*.
  - Cette génération est optimisée, **seules les opérations nécessaires** sont effectuées.
  - En cas d'échec **d'une** opération, make s'arrête.

- Le principe de base est celui de ***cible/dépendance* – action** qui forme une arborescence.
  - En réalité, peut s'appliquer à d'autres sujets que la génération d'application

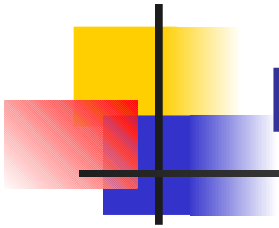
make

- Format du fichier de description

```
cible: liste de dépendances
<tab> action
```

- Algorithme de make

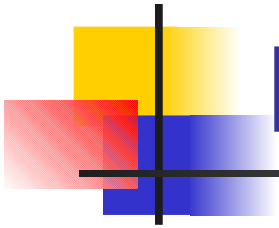
```
si (
(la cible n'existe pas) ou
(il n'y a pas de dépendances) ou
(la cible est plus ancienne qu'au moins une
  dépendance)
)
  alors
  exécuter action (1 ou n commandes)
```



make

- Autre manière d'exprimer cet algorithme :

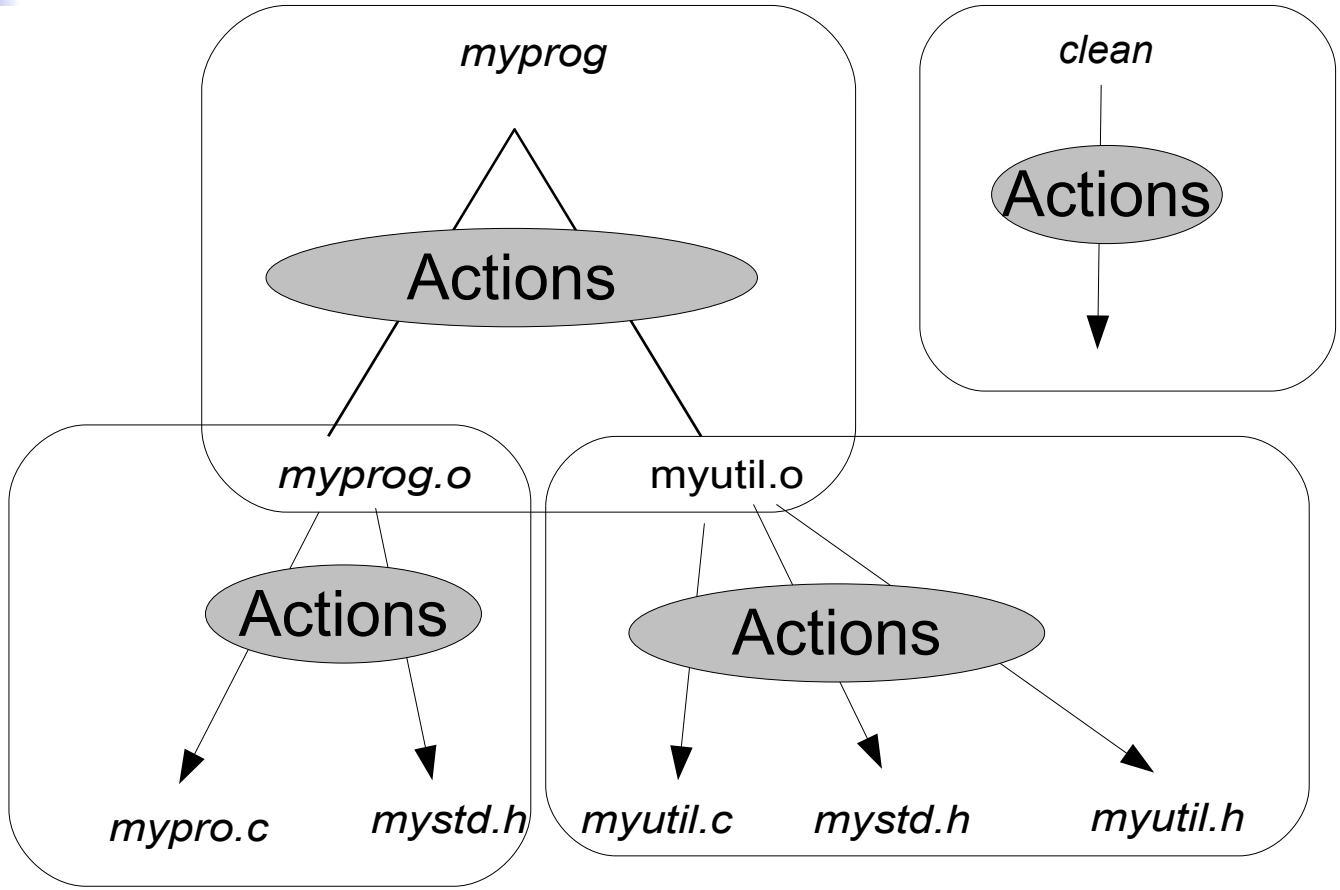
  "Si au moins une de *dépendances* est plus récente que la *cible*, les *actions associées* sont exécutées (en général, génération de la cible)."

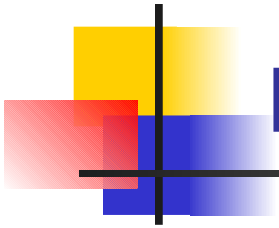


Exemple

```
# Exemple de makefile
myprog : myprog.o myutil.o
   ld myprog.o myutil.o -o myprog

myprog.o : myprog.c mystd.h
   cc -c myprog.c

myutil.o : myutil.c myutil.h mystd.h
   cc -c myutil.c

#effacer les fichiers .o
clean:
    rm -f myprog.o myutil.o
```

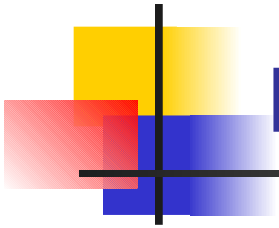
Arbre de dépendances





make

- Les feuilles de l'arbre apparaissent uniquement comme *dépendance*;
- la racine n'apparaît que comme *cible*
- les sommets intermédiaires apparaissent a la fois comme *cible* et comme *dépendance*.



make: lancement

- Sans l'option *-f*, *make* cherche un fichier *makefile* puis *Makefile* (dans le répertoire courant)
  - avec l'option *-f file*, *make* utilise le fichier *file*.

$ make

$ make -f mymakefile

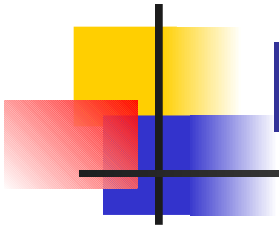
les cibles

- Sans nom de *cible* comme argument, *make* reconstruit la première *cible* trouvée
- Avec un nom de *cible* en argument, *make* considère cette *cible* comme le sommet de *l'arbre des dépendances* et la reconstruit :

```
$ make cible
$ make cible1 cible2 ... ciblen
```

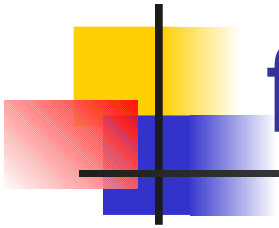
- une cible au sommet d'un arbre et qui n'est pas la première, il faut absolument la nommer pour exécuter les actions associées:

```
$ make clean
```



les options

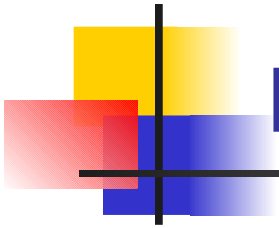
- option *-n*,
  - *make* affiche les commandes mais ne les exécute pas. Cette forme est utilisée pour le test.
- option *-i*
  - permet d'ignorer les code d'erreurs des commandes.
  - make continue même avec des code retour ≠ 0
- option *macro=value*
  - définir des macros (au sens make)



fichier makefile

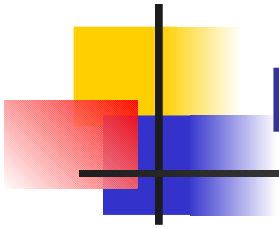
- peut contenir les types de lignes suivantes :
  - lignes blanches: ignorées
  - commentaires : `#…`
  - ligne de dépendance: `cible: dep1 dep2 …`
  - commande: commence par un TAB
  - règles implicites: `.x.y:`
  - définition de macros `macro=val`
  - inclusion de fichier `include autremakefile`
  - (instruction conditionnelle, …)

- + une ligne vide à la fin



make : pourquoi ?

- les buts:
  - optimiser
  - simplifier : *make* fait tout
  - sécuriser
    - usage facile pour le développeur utilisateur
    - cohérence de l'ensemble obtenu
  - Support de contextes lourds
    - des dizaines de milliers de fichiers, des centaines de modules, des dizaines de développeurs …
- Ca vaut le coup de compliquer un peu la syntaxe !



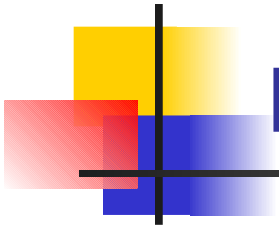
make : macros

- Objectifs
  - simplifier les makefiles, les rendre plus portables, plus évolutifs !
- Définir une macro
  - dans le makefile:

    ```
    macro1=valeur1
    ```
  - lors de l'appel de make (prioritaire) :

    ```
    $ make macro1=valeur1 macro2=valeur2
    ```
- Pour faire référénce à une macro dans le makefile

  ```
  $(macro1)
  ```



make: exemples de macros

- définir les répertoires où se trouvent les fichiers include

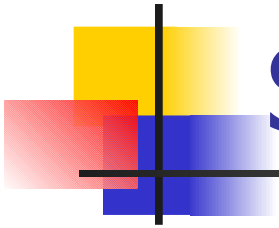
```
INCLUDE=-I../../malib/include -I../include
```

- définir les commandes à utiliser:

```
CCPP=g++
CC=gcc
LINK=g++
```

- définir les flags de compilation

```
DEBUGFLAGS=-g -DDEBUG
CFLAGS=$(CFLAGS) $(DEBUGFLAGS)
```



Subsitution de macros

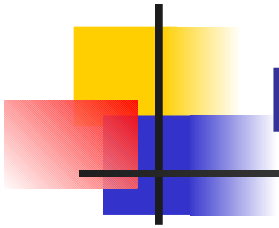
- Objectifs : simplifier les makefiles, les rendre plus portables, plus évolutifs !
- Syntaxe

```
$(macro1:substr=replace)
```

  - remplace *substr* par *replace* dans la macro *macro1.*

- Exemple

```
SRC=main.c utils.c test.c
OBJ=$(SRC:.c=.o)
exec : $(OBJS)
  …
```



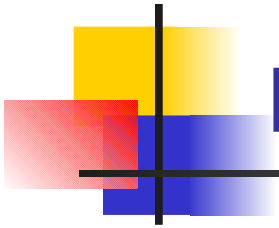
macros prédéfinies

- liste des dépendances plus récentes que la cible :
  - `$?`
    - `Exemple : très utile avec la mise à jour des librairies static`

```
lib.a: $(OBJS)
        ar r lib.a $?
```

- nom de la cible courante
  - `$@`
    - `Très utilisée pour limiter la redondance`

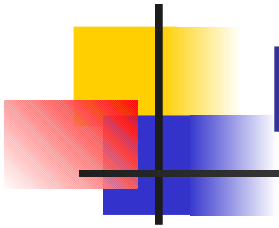
```
lib.a: $(OBJS)
        ar rvu $@ $?
```



macro prédéfinies

- les modificateurs D et F permettent d'obtenir le répertoire uniquement ou le nom de fichier uniquement des macros *$@ (et $* et $<)*

```
OBJS = main.o utils.o test.o
CFLAGS = -ansi -DDEBUG -g
exec: $(OBJS)
  echo "build de $(@F) dans $(@D)"
  $(CC) $(OBJS) -o $@
```



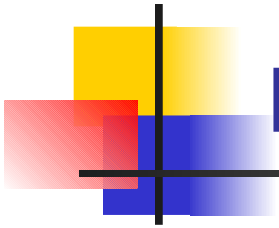
Règles d'inférences implicites

- Objectifs : simplifier les makefiles, les rendre plus portables, plus évolutifs.
  - une règle unique décrit comment générer certains fichiers
  - Par exemple, pour générer un module objet à partir d'un fichier source en C.

```
.SUFFIXES:
.SUFFIXES: .o .c
.c.o :
    $(CC) $(CFLAGS) -c $<
```

Règles d'inférences implicites

- Macros spécifiques
  - nom de la dépendance qui entraîne la reconstruction de la cible (car plus récente)
    - $<
  - nom de la cible sans le suffixe
    - $*
- Utilisation des modifications F et D
  - $(<F) $(*D)



make : divers

- pour avoir le $, il faut utiliser *$$*

```
echo $$HOME
```

- ignorer le code d'erreur d'une commande ponctuellement : -

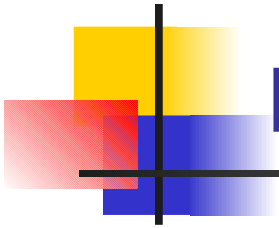
```
-rm -f $(TMPFILE)
```

- ne pas afficher la ligne de commande : @

```
@$(CC) ....
```

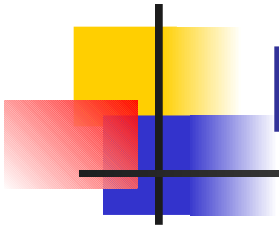
- continuer sur la ligne suivante : caractère \

```
SRC= 1.c 2.c .... 12.c \
  13.c ...
```



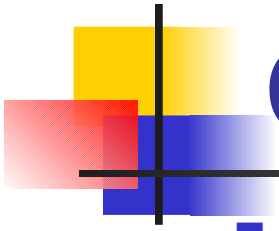
make : pour les experts

- Parties conditionnelles
- include de makefile
- métacaractères du shell (*,?,[abc]) accepté dans les noms de fichiers
- métacaractère spécifique %



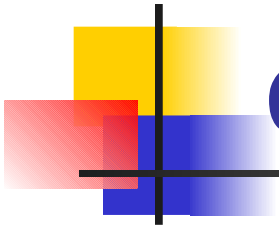
Dépendances automatiques

- Pour les fichiers include, il est très fastidieux et quasi-impossible à grande échelle de lister tous les .h inclus par les modules C !
- 2 approches :
  - générer automatiquement la liste dépendances correspondantes
    - makedepend
  - supprimer les .o correspondants pour forcer la regénération.
    - Petit programme de 300 ou 400 lignes !



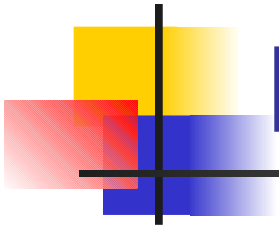
Génération des makefiles

- Les makefiles sont indispensables dans un environnement réel.
  - leur gestion peut être lourde et source d'erreurs.
- autoconf, automake, configure
  - Adaptation automatique au système d'exploitation, aux différents compilateurs, aux bibliothèques (recherche de la présence des fonctions) et utilitaires présents sur le système (yacc/bison, cc/gcc, lex/flex, awk/gawk,...)
  - Très employé dans les distributions GNU
- imake
  - utilisation d'un fichier modèle (template) Imakefile.
- cmake



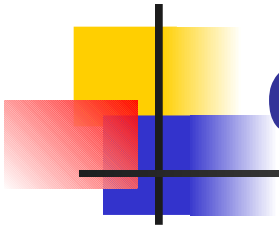
développement et/ou déploiement

- make est adapté dans le domaine du développement
  - mais aussi pour installer des logiciels,
  - appliquer des séries de traitements à des fichiers
  - si un arbre de dépendances a un sens, make aussi !



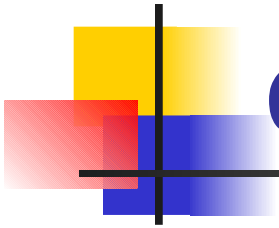
DEBUGGER

- *Besoin :* trouver les erreurs dans un programme
- *Solution*
  - un programme capable de **simuler l'exécution** d'un programme,
  - pouvoir faire le **lien entre l'exécution du programme et les fichiers sources** ayant servi à le générer.
  - pouvoir **arrêter la simulation** à un instant donné, **la reprendre**,
  - pouvoir **afficher la valeur** d'une expression ou d'une variable,
  - pouvoir consulter la **pile des appels** de fonctions



debugger

- Et aussi:
  - **attacher** un programme ayant commencer à s'exécuter
  - pouvoir définir à l'avance les **points d'arrêt** (breakpoint)
  - point d'arrêt conditionnel
  - **gestion des signaux** (SEGV)
  - **modification** de variables



debugger : table des symboles

- le compilateur doit permettre de stocker des informations relatives aux fichiers sources :
  - *la table des symboles*
    - les noms et localisations dans les sources des variables et fonctions, ainsi que les informations de typage.
    - liens entre instructions machines et fichiers sources
- Ces informations ne sont pas utiles à l'exécution mais uniquement au debogage.

- Avec *gcc*, on ajoute ces informations par l'option ***-g***

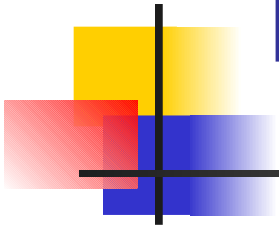
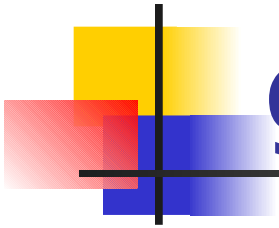


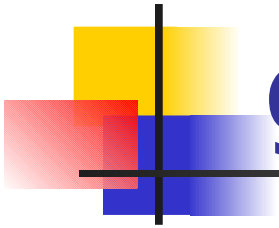
Table des symboles

- Regarder nm pour comprendre ...
  - gcc -c 1.c
  - nm -l 1.o
  - gcc -c -g 1.c
  - nm -l 1.o
  - gcc -c -O3 1.c
  - nm -l 1.o



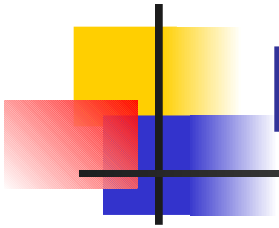
gdb

- Nous parlerons ici du debugger GNU, compatible avec *gcc* : *gdb*
- Interfaces graphiques
  - s'appuient sur *gdb :*
    - *ddd, xxgdb*, *mxgdb*, *kgdb* (kdevelop), *ddd*, ...
- Langages de programmation
  - *gdb* supporte plusieurs langages de programmations : C, C++, Modula-2, ...



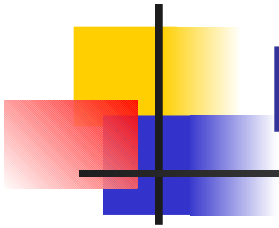
gdb : fonctions principales

- Exécution instruction par instruction
- points d'arrêts (breakpoints), arrêt sur condition,
- arrêt sur signal (par exemple SIGSEGV)
- affichage des valeurs des variables
- évaluation d'expressions
- modification de la valeur de variable
- affichage de la pile d'appel des fonctions
- affichage du code source ou assembleur



Démarrer une session

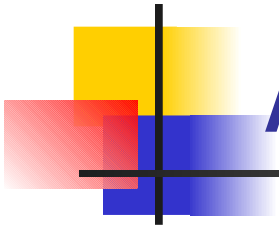
- *$ gdb a.out*
- *$ gdb*
  - (gdb) *file a.out*
- si le programme s'exécute dèjà:
  - *$ gdb*
  - *(gdb) attach 4425*



Exécution

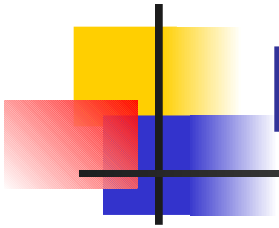
- *(gdb) run arg1 arg2 ...*
  - Les arguments sont donnés comme sur la ligne de commande.

- L'environnement
  - hérité de GDB et donc du shell ayant lancé gdb.
  - modifié par :
    - *(gdb) set environment variable=value*
  - répertoire courant de gdb, donc du shell ayant lancé gdb.
  - modifié par :
    - *(gdb) cd /var/tmp*



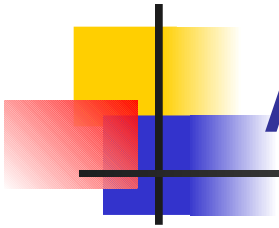
Afficher les fichiers sources

- La commande *list* (ou *l*) avec optionnellement une référence à une ligne, qui peut être:
  - absolue :
    - nom du fichier:numéro de ligne
    - (gdb) l test1.c:52
  - absolue pour le fichier courant:
    - (gdb) l 52
  - relative à la position courante
    - (gdb) l +6
    - (gdb) l -10
  - une référence à un nom de fonction
    - (gdb) l test1.c:unefonc
    - (gdb) l unefonc



Point d'arrêts

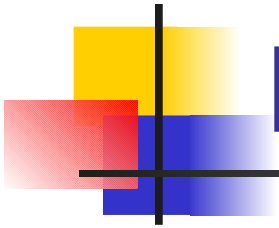
- Possibilité de mettre des breakpoints en référence à une ligne ou à une fonction
  - (gdb) break main
  - (gdb) b main
  - (gdb) b 556
- Commandes principales: *break*, *clear*
- possibilité d'indiquer une condition booléenne
- *(gdb) help breakpoint*



Après un breakpoint

- *continue* (c) pour continuer l'exécution
- *step* (s) pour s'arrêter à la ligne suivante, en rentrant éventuellement dans les appels de fonctions
- *next* (n) pour s'arrêter à la ligne suivante, sans rentrer dans les appels de fonctions
- *finish* pour aller jusqu'à la fin de la fonction courante.

- *(gdb) help running*



La pile des appels

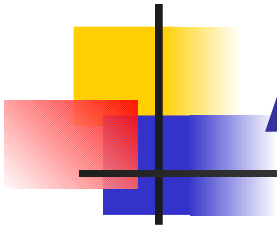
- *bt (ou where)* : affiche la pile des appels

- *up* : *remonter* dans la pile des appels
- *down* : *descendre* dans la pile des appels
  - pour inspecter des variables
  - pas de modification d'exécution !

- *(gdb) bt full* pour voir aussi les variables locales
  - Intéressant pour bien comprendre les appels récursifs

Pile + variable locales

```
(gdb) bt
#0  f (k=2, z=3.14159203) at td2_ex3.c:10
#1  0x08048426 in f (k=3, z=3.14159203) at td2_ex3.c:7
#2  0x080484dd in main (argc=1, argv=0xbff27fb4) at td2_ex3.c:25
(gdb) bt full
#0  f (k=2, z=3.14159203) at td2_ex3.c:10
        tmp = 3.14159203
#1  0x08048426 in f (k=3, z=3.14159203) at td2_ex3.c:7
        tmp = 0
#2  0x080484dd in main (argc=1, argv=0xbff27fb4) at td2_ex3.c:25
        a = 3.14159203
        b = 3
        r = 0
```

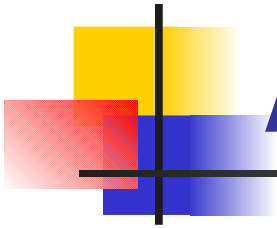


Affichage des valeurs

- *print p* ou *print(p)*
  - affiche la valeur d'une expression
- *printf*
  - utilisation de description de format ressemblant au printf du C
- *display var* ou *expression*
  - display automatique à chaque point d'arrêt

Informations sur le programme

- La commande *info* permet d'obtenir de nombreuses informations sur le programme en cours de debug.
  - *info source* : nom du fichier courant
  - *info sources* : liste des fichiers sources
    - Y compris ceux qu'on ne connaît pas ! (libs)
  - *info types* : description des types connus
  - *info functions* : description des fonctions
  - *info variables* : description des variables
    - Y compris stdin, stdout, ...
  - ...



Aide …

- Interne
  - (gdb) *help*
  - *(gdb) help <sous-rubrique>*
  - (gdb) *help <commande>*
- Intégration emacs
  - ALT-x compile
  - ALT-x gdb
    - affichage des sources
    - positionner les breakpoints
- Aide GNU *: plus de 200 pages !!!*
  - http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

A propos des symboles ...

- `nm`
  - examiner les symboles
- `strip`
  - suppression des symboles

```
$ nm -l 1.o
$ strip 1.o
$ nm -l 1.o
```