Debuguer une application sous Linux en C/C++

Gilles Maire

2017

Titre du cours

Linux

Rubriques

- Compilateurs C et C++
- Options du compilateur
- Les librairies
- Les binutils

Compilateurs C et C++

Les compilateurs C et C++

- http://gcc.gnu.org/
- Le nom du compilateur C est gcc, le nom du compilateur C++ est g++
- Le compilateur fournit les fichiers entêtes pour ses fonctions
- gcc regroupe les compilateurs suivants : C, C++, Objective-C, Fortran, Java, Ada, and Go
- quand on compile gcc, on spécifie au moment de la configuration la liste des langages supportés par l'option –enable-languages=c,c++ etc
- Gcc est assez compliqué à compiler
- on peut par argument intégrer as et ld dans la compilation

Le compilateur GCC (suite)

- Le compilateur sait produire du code natif pour les processeurs suivants :
- ARM, AVR,Blackfin, CRIS, FRV, M32, MIPS, MN10300,
- PowerPC, SH, v850, i386, x86_64, IA64, Xtensa
- Pour chacun de ces processeurs des sous variantes sont possibles. Par exemple
- un processeur ARM peut contenir les extensions thumb ou un jeu d'instructions plus ancien comme armv4t armv5t
- pour la famille des processeurs x86 on a les architectures pentium, i88, i38 etc

Usage

- Dans la famille des compilateurs Gnu :
  - le compilateur **C** est gcc
  - le compilateur **C++** est g++
  - à ne pas confondre avec
    - **gpp** qui est le préprocesseur générique
    - **cpp** le préprocesseur du C
- En C++
  - les fichiers contenant les codes définitions sont les fichiers d'extension cpp
  - les fichiers contenant les déclarations sont les fichiers d'extension h
  - le programme main est contenu généralement dans main.cpp
- En C
  - Les header sont dans les fichiers.h
  - Les sources dans les fichiers .c

Les étapes de compilation

- Quatre étapes
- Preprocessing
  - gcc -E programme.c > programme.i
- Compilation vers l'assembleur
  - gcc -S programme.i produit le fichier programme.s
- Assembleur vers le code machine
  - gcc -c programme.s produit le fichier programme.o
- Édition des liens
  - gcc -o programme programme.o produit l'exécutable programme
- Mais en général on fait les 4 en une étape raccourcie :
  - gcc -o programme programme.c

Options du compilateur

Options

- `-o prog` : produit en sortie le programme `prog`. Si cette option est omise, le fichier `a.out` sera produit
- `-c prog.c` : produit le fichier `prog.o` qui devra être lié avec d'autres modules et librairies
- `-g` : ajoute au binaire les informations utiles au débugage
- `-I<chemin>` : ajoute chemin aux répertoires dans lesquels le compilateur ira chercher les fichiers header
- `-L<chemin>` : ajoute chemin aux répertoires dans lesquels le compilateur ira chercher les librairies
- `-l<nomlibrairie>` : indique qu'il faut lier la librairie au binaire
- `-ansi` : indique qu'on souhaite rendre le compilateur compatible avec la norme ANSI
- `-std=<standard>` : indique la version de révision de langage à utiliser C99, c++98,c++11, c++14
- `-D valeur` : est équivalent à #define valeur
- `@fichier` : les options sont contenues dans le fichier

Optimisations

- `-O` ou `-O1` : réduit la taille du code et le temps d'exécution sans faire d'optimisations nécessitant de longs temps de compilation
- `-O2` : améliore la réduction de la taille du code et le temps d'exécution mais nécessite plus de temps de compilation
- `-O3` : améliore encore davantage en mettant en oeuvre entre autre:
  - les fonctions inline
  - les boucles avec conditions invariables en dehors de la boucle
  - optimisation des arbres avec boucle de vecteurs
- `-O0` : défaut, produit la compilation la plus rapide
- `-Os` : produit une compilation optimisée pour la taille du code
- `-Ofast` : produit du code le plus rapide possible
- `-Og` : produit du code optimisé pour le débugage

Exemples

- Production d'un fichier fichier.o (l'option -c s'arrête avant l'édition de liens)

```
gcc -c  fichier.c
```

- Deux exemples de production du fichier exécutable programme avec édition de liens

```
gcc -o programme  fichier1.o fichier2.o -lm
gcc -o prg fichier1.o fichier2.o -L/usr/local/lib -lpet -lreadline
```

- **Attention :** quand l'éditeur de lien est appelé par gcc ou g++ l'ordre des arguments passés est important : il faut que les ordres du linker soient à la fin.

Mécanisme de Warning

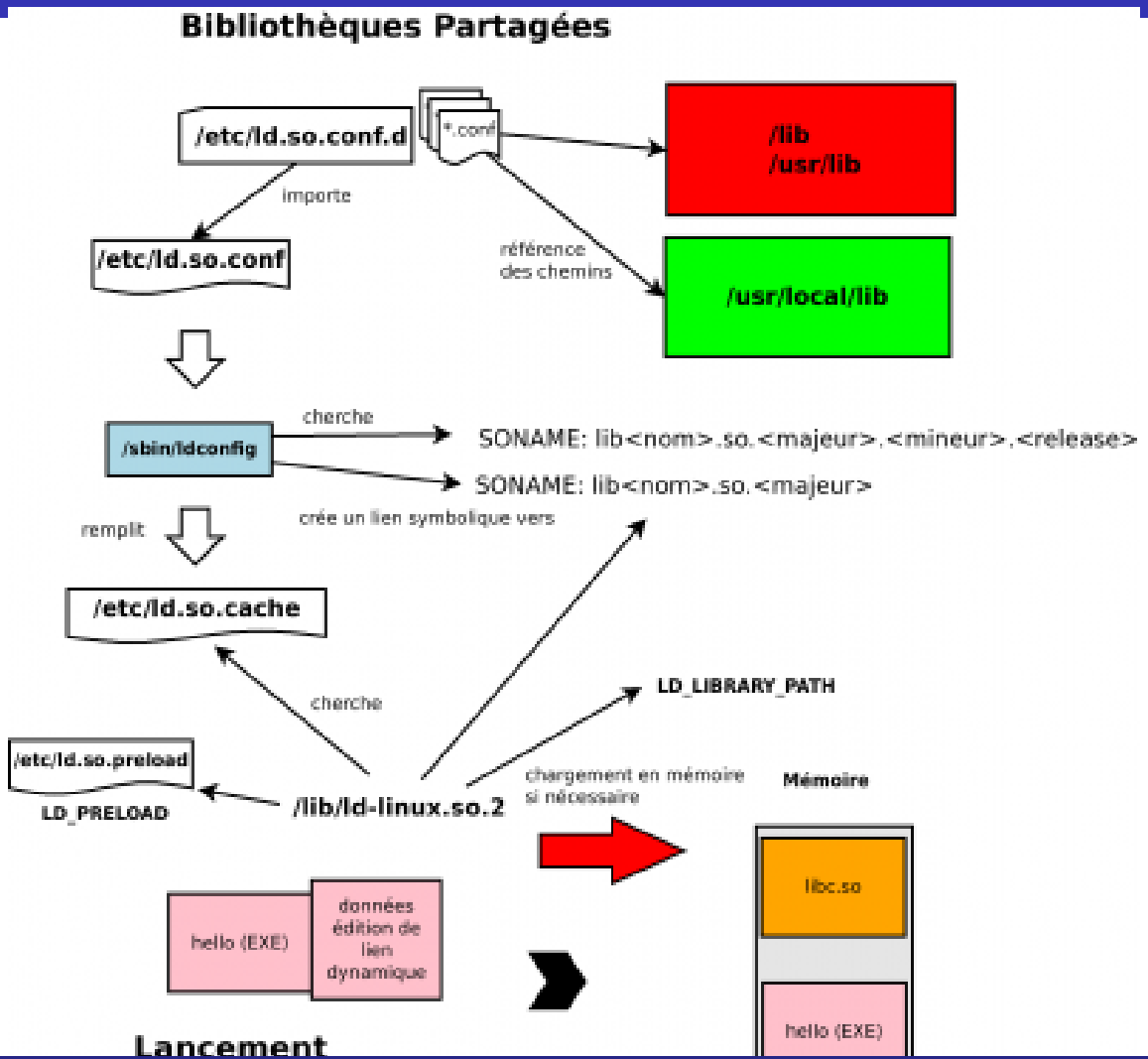
- Nous allons regarder quelques options des compilateurs gcc et g++ qui concernent la production de Warning
- **-Wpedantic (-pendandic) :** les programmes doivent suivre la norme ISO, pas d'extension gnu
- **-w :** inhibe les messages de Warning
- **-Werror :** transforme tout message de Warning en erreurs
- **-Werror= :** transforme le message de Warning donné en erreur (-Werror=unused-but-set-variable voir la liste dans man gcc)
- **-Wall :** positionne la plupart des drapeaux de Warning
- **-Wextra :** positionne quelques warnings qui ne sont pas positionnés par *-Wall*
- **-Wfatal-errors :** arrêt de la compilation à la première erreur

Quelques Warnings utiles :

- **-Wuninitialized :** une variable auto est utilisée sans être initialisée
- **-Wmissing-declarations :** provoque un warning si une fonction n'a pas été déclarée
- **-Wcomment :** provoque un warning si une marque de début de commentaire apparaît dans une commentaire
- **-Wformat :** teste les printf et les scanf pour vérifier que les types données en paramètres sont corrects
- **-Wparentheses :** vérifie que les parenthèses ne sont pas omises notamment dans les expressions booléennes composées
- **-Wswitch-default :** vérifie que les instructions switch ont bien une étiquette défaut
- **Wunused-but-set-variable :** variable inutilisée mais initialisée

Les librairies

Librairies


Bibliothèques Partagées
/etc/ld.so.conf.d
*.conf
/lib
/usr/lib
importe
/etc/ld.so.conf
référence
des chemins
/usr/local/lib
/sbin/ldconfig
cherche
SONAME: lib<nom>.so.<majeur>.<mineur>.<release>
SONAME: lib<nom>.so.<majeur>
remplit
crée un lien symbolique vers
/etc/ld.so.cache
cherche
LD_LIBRARY_PATH
/etc/ld.so.preload
LD_PRELOAD
/lib/ld-linux.so.2
chargement en mémoire
si nécessaire
Mémoire
libc.so
hello (EXE)
données
édition de
lien
dynamique
hello (EXE)
Lancement

Édition de lien

- La compilation d'un programme produit un fichier .o

```
gcc -c exemple.o exemple.c
```

- Imaginons que ce programme ait besoin d'une librairie malibrairie est rangée dans /usr/lib/libmalibrairie.a
- l'éditeur de lien ld rassemble la librairie et exemple.o pour en faire un exécutable exemple

```
ld -L/usr/lib -lmalibrairie exemple.o -o exemple
```

- l'option -static force l'édition de lien à ne prendre que des librairies statiques.
- On peut aussi appeler l'édition de lien directement par gcc ou g++

Utilisation des librairies

- Pour voir les librairies utilisées par un programme :

```
ldd programme
```

- Pour lancer un programme en indiquant un chemin temporaire pour une librairie :

```
LD_LIBRARY_PATH=/chemin/vers/la/librairie  programme
```

- Pour installer définitivement une librairie dans un emplacement déclarer le chemin dans un fichier conf présent dans /etc/ld.so.conf.d et ne pas oublier de lancer la commande

```
ldconfig
```

Utilisation de librairie C en C++

- Si la fonction toto est déclarée en C plusieurs solutions :

```
extern "C" void toto( arguments ) ;
```

- ou

```
extern "C" {
   void  fonction1();
   void fonction2();
}
```

- ou

```
extern "C" {
#include fichier.h
}
```

Installation des paquetages dev

- Quand vous installez une distribution elle n'est pas forcément orientée développement et vous devrez installer un ensemble de compilateurs et d'outils de développements (build-essential)
- Les paquets de développement dev de chaque librairie utilisée permettant de trouver les headers des librairies.
- Ces fichiers headers portent en général les noms des libairies augmenté de l'extension -dev
- Sous Redhat ils comportent l'extension -devel qui reste parfois quand les paquets sont d'origine rpm et qu'ils ont été convertis via l'utilitaire alien (alien convertit des fichiers rpm en deb et vice versa)

Les binutils

Les binutils

- sont les utilitaires permettant de traiter les objets une fois la compilation effectuée (http://www.gnu.org/software/binutils)
- **addr2line :** utile pour le débugage permet de connaître la ligne et le programme correspondant à une adresse dans un exécutable
- **ar :** permet de retrouver, de créer, de modifier les fichiers d'une archive (utilisé pour la gestion des librairies)
- **as :** l'assembleur qui sait gérer beaucoup de processeurs ainsi que leurs variantes (faire man as est instructif)
- **c++filt :** ce programme, utilisé pour les programmes en Java et C++, permet de reconstituer les noms des méthodes surchargées (même nom arguments différents) car en assembleur les noms de ces méthodes ont été différenciés

Binutils (suite)

- **elfedit :** met à jour les entêtes des fichiers ELF
- **gprof :** lit le profil d'exécution permettant de connaître le temps passé dans chaque procédure (option -pg au compilateur)
- **ld :** c'est le linker qui rassemble les différents objets issus du C ou C++ ainsi que les objets issus des librairies
- **nm :** liste les symboles d'un fichier objet
- **objjcopy, objdump :** recopie ou dump les fichiers obj en permettant leur relocation
- **ranlib :** génère l'index d'une librairie et la stocke dans celle-ci
- **readelf :** affiche des informations au sujet des fichiers ELF
- **size :** affiche les sections taille d'un objet ou de chaque objet d'une librairie
- **strings :** affiche les chaînes de plus de quatre caractères présentes dans un fichier
- **strip :** supprime les symboles d'un programme, ce qui le rend plus compact mais impossible à débuguer

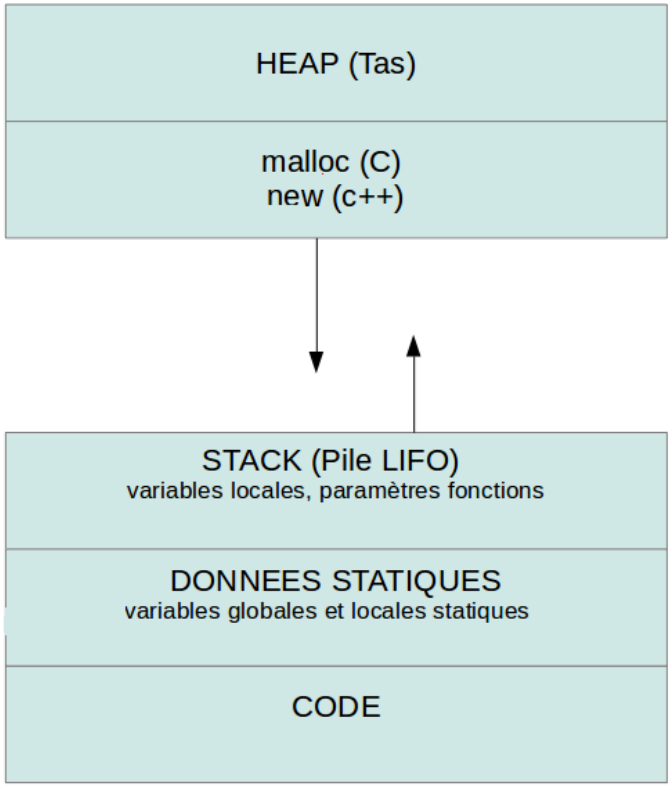
Problèmes

Rubriques

- Mémoire
- Problèmes de cadrage des informations
- Perte de mémoire

Mémoire

Organisation de la mémoire



- Les variables statiques non initialisées sont initialisées à 0
- Elles sont déclarées précédées du mot static

Exercice Visualisation mémoire

- Écrire un programme en C qui affiche
  - les adresses mémoire de variables contiguës statiques, initialisées ou pas,
  - les adresses de variables locales
  - les adresses d'un tableau (on affiche l'adresse par le paramètres %p dans la fonction printf)
- à l'aide d'un pointeur sur une variable, modifier une variable contiguë en incrémentant le pointeur.

Dépassements de tableaux

- en C et C++ la déclaration d'un tableau d'une longueur donnée n'interdit pas de dépasser les bornes de ce tableau
- en C avant C99, on était tenu de déclarer un tableau statique avec une valeur entière `int tab[10]`, depuis C99 on peut donner une variable à la dimension du tableau mais on peut dépasser les bornes des tableaux.
- On peut également dépasser les bornes d'un tableau par l'emploi d'un pointeur puisque on a identité entre les deux écritures.

```
int tab[10] ;
int *tab; tab = malloc(10);
```

Problèmes de cadrage des informations

Initialisation

- Seules les variables statiques en C ou C++ sont initialisées à 0
- Dans les autres cas, la valeur peut être initialisée à 0 si la mémoire qu'elle occupe est à zéro mais rien ne l'assure.
- Les variables non initialisées sont malheureusement la bête noire des compilateurs Gnu.
- Les options de Warning n'ont pas d'effet mis à part la variable d d'optimisation -O2
- Compiler avec g++ et gcc le programme mixte suivant :

```
int main()
{
int i;
int j=i;
}
```

Différence de taille en mémoire

- Le typage des données en C et C++ ne fixe pas la taille des données :
  - le type `int` peut valoir 16, 32 ou 64 bits suivant les architectures
- On ne connaît que la taille minimale et on peut dire que :
  - **char** >= 8bits
  - **short** >= 16 bits
  - **int** >= 16 bits
  - **long** >=32 bits
  - **long long** >= 64 bits
  - **float** >= 6 chiffres décimaux
  - **double** >= 10 chiffres décimaux
  - **long double** >= 10 chiffres décimaux
  - **long double** >= 10 chiffres décimaux
- Ceci impose de définir des types de longueur fixe que ce soit par des librairies connexes comme Qt ou Boost, ou bien dans ses propres définitions.

Exemple

- Allocation et libération d'un pointeur

```
char * p= malloc(10);
for ( int i=0; i<10, *; i++) p[i]=1;
free(p);
```

- Boucle avec perte de mémoire

```
while ( 1!=0)
{
   char *p = malloc(32000);
}
```

Perte de mémoire

Allocation

- Chaque *malloc* (C), chaque *new*(C++) renvoie un pointeur

```
machin *p = new machin();
```

- Ce pointeur doit être conservé pour effectuer une libération de mémoire
- Lorsqu'une fonction demande un pointeur sur une classe en argument, il se peut qu'elle gère la libération de la zone mémoire pointée, dans ce cas et seulement dans ce cas on peut utiliser la syntaxe suivante à

l'appel

```
// void MaClasse::MaFonction( AutreClasse *ptr);
MaFonction(new  AutreClasse) ;
```

- Cette notation que l'on rencontre ne libère pas la mémoire allouée dans la majorité des cas.

Garbage collector

- Il existe trois garbage collecteur capables de désallouer la mémoire automatiquement en C/C++
  - la librairie GC installée dans le build-essential sous Debian et contenue dans la librairie libc1c2
  - les pointeurs intelligents de la STL, n'autorisant qu'une seule allocation ou comptabilisant les allocations.
  - les classes maîtresses comme Qt/QObjet maintenant la liste de tous les enfants de chaque objet et permettant la destruction de ceux-ci en cas de destruction de l'objet.

Profilage sans source

Rubriques

- strace et ltrace
- La mémoire
- lsof, netstat

strace et ltrace

Présentation de strace et ltrace

- *strace* permet de visualiser les appels systèmes appelés par un programme.

```
strace ls
```

```
execve("/bin/ls", ["ls"], [/* 61 vars */]) = 0
brk(0)                                  = 0x125b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS,-1, 0) = 0x7f9634326000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

- *ltrace* permet d'analyser les appels à des fonctions de librairies dynamiques

```
ltrace ls /
```

Intérêt de ltrace et strace

- Les deux sont très utiles pour débuguer un programme dont on ne possède pas les codes sources
- *strace* permet de localiser un plantage par rapport au dernier un appel système
- *ltrace* permet de localiser le plantage par rapport aux fonctions des librairies appelées
- *strace* permet d'analyser les temps d'exécution entre chaque appel système ou de voir les temps pris par les appels systèmes
- très utile pour voir les ressources utilisées par un programme
- permet de suivre les temps de chaque appel de librairie afin de déterminer les causes de lenteur d'un programme.
- un fichier de décodage des principaux appels de ltrace est situé dans /etc/ltrace.conf.

Options communes de ltrace et strace

- **-o fichier :** affiche le résultat dans le *fichier*
- **-p pid :** exécute **strace** sur le processus *pid*

```
sudo strace -p 1725 -o firefox_trace.txt
```

- **-c :** affiche un sommaire des appels systèmes avec les temps d'exécution et un compteur de passage
- **-r :** affiche un timestamp relatif sur chaque appel
- **-t :** affiche l'heure d'appel au format hh:mm:ss
- **-tt :** affiche l'heure d'appel suivi des microsecondes
- **-T :** donne le temps de chaque appel

strace avec option -e

- l'option -e permet de spécifier les appels systèmes que l'on veut afficher.
- Pour sélectionner l'appel système *open* :

```
strace -e open ls
```

- Pour sélectionner plusieurs appels système, par exemple *open* et *close* on ajoute le mot *trace*= suivis des appels système

```
strace -e trace=open,read ls
```

strace : résultat statistique

% time	seconds	usecs/call	calls	errors	syscall
-- --	-				
25.31	0.000308	12	26		mmap
13.80	0.000168	11	16		mprotect
12.74	0.000155	16	10		open
11.26	0.000137	15	9	9	access
5.51	0.000067	7	9		read
5.01	0.000061	6	11		fstat
4.68	0.000057	19	3		munmap
4.35	0.000053	4	13		close
3.62	0.000044	15	3		brk
3.20	0.000039	20	2		write
3.12	0.000038	38	1		execve
2.14	0.000026	13	2	2	statfs
1.89	0.000023	12	2		getdents

Fichier de configuration

- Trouver le fichier de configuration lu par votre navigateur

La mémoire

Rappels sur la mémoire

- **Mémoire physique :** la quantité de RAM sur les barrettes mémoire
- **Mémoire swap :** espace de mémoire sur disque, utilisé quand la RAM est totalement utilisée. Les pages **inactives** sont transférées de la RAM vers le swap
- **Buffers :** mémoire tampon est une zone de mémoire où sont stockées les données en attente d'écriture sur le disque
- **La mémoire cache :** zone mémoire servant à stocker des données calculée afin d'éviter les mêmes calculs ultérieurs. On y trouve usuellement les partitions montées en mémoire, les données applicatives, la mémoire vive de machines virtuelles
- **La mémoire partagée :** c'est la mémoire partagée entre des applications,.
- **Mémoire active :** mémoire utilisée en RAM (clean mem)
- **Mémoire non active :** mémoire placée en swap (dirty mem)
- **Résident Set Size (RSS) :** Quantité de mémoire allouée pour le processus uniquement en RAM
- **Proportional Set Size (PSS) :** Quantité de mémoire partagée avec d'autres processus. Si le processus meurt elle n'est pas libérée. Si 3 processus partagent 30Ko, chaque processus aura un PSS de 10Ko.
- **Unique/Unshared Set Size (USS) :** Ensemble de pages unique allouées à un processus. Si le processus meurt, elle est libérée.
- **Virtual Set Size (VSS) :** c'est la quantité d'adresse totale accessible pour un

La commande free

```
free -ht
             total       used       free     shared    buffers  cached
Mem:          7,7G       6,4G      1,3G       306M       657M      3,9G
-/+ buffers/cache:       1,9G       5,8G
Mémoire d'échange :  3,9G          0B       3,9G
Total :         11G        6,4G       5,1G
```

- Dans cet exemple on 7,7G de RAM sont utilisés, on pourrait croire que l'on dispose de 1,3 G de mémoire RAM libre, en fait on dispose de la RAM libre + les 3,9 Go de cache - la mémoire cache des machines virtuelles - partitions en mémoire = 5.1 Go (ligne total free)

vmstat

- Affiche les informations sur les processus, la mémoire, la pagination, les interruptions et l'activité des processeurs
  - Le premier rapport produit présente les moyennes depuis le dernier démarrage.
  - Les rapports ultérieurs présentent un compte rendu tous les intervalles
- La syntaxe générale de la commande est

```
vmstat intervalle total
```

- **intervalle :** intervalle de temps entre deux mises à jour
- **total :** nombre de mise à jour demandée. En cas d'omission de ce paramètre, le total est infini.

```
vmstat 1 10
```

vmstat : Les champs non explicites

- **procs** (processus)
  - **r :** Nombre de processus exécutables (en cours ou en attente d'exécution).
  - **b :** Nombre de processus en sommeil non interruptible.
- **memory**
  - **swpd :** Quantité de mémoire virtuelle utilisée.
  - **buff :** Quantité de mémoire tampon.
- **system** (système)
  - **in :** Nombre d'interruptions par seconde, incluant l'horloge.
  - **cs :** Nombre de bascules du contexte par seconde.
- **processeur :** en %
  - **us :** Temps consommé par les processus hors noyau
  - **sy :** Temps consommé par le noyau (temps système)
  - **id :** Temps d'inactivité
  - **wa :** Temps d'attente des entrées et sorties
  - **st :** Temps volé par une machine virtuelle
- Différentes options
  - **-a :** Afficher la mémoire active et inactive
  - **-d :** Afficher les statistiques d'accès disque
  - **-m :** Afficher le slabinfo (en root, file system, info dma etc)

lsof, netstat

smem et pmap

- **smem :** affiche la mémoire utilisée sous différentes vues
  - sans option affiche les informations par process
  - - **m :** affiche la mémoire utilisée par chaque librairie
  - **-u :** affiche la mémoire utilisé par utilisateur
  - **-w :** affiche un sommaire de la mémoire utilisée
  - **-p :** affiche en pourcentage
- **pidof :** affiche le pid d'un programme (pidof mozilla)
- **pmap :** affiche la mémoire utilisée pour un process
  - **-x :** format étendu
  - **-X :** afficher plus de précision
  - **-XX :** afficher tout ce que le noyau fournit

lsof

- *lsof* donne la liste des fichiers ouverts sur le système pour tous les process. Beaucoup d'options sont accessibles et explicitée dans la page man.
- **-p pid :** affiche la liste pour *pid*
- **-u utilisateur :** affiche la liste pour *utilisateur*
- **-d numéro :** affiche la liste pour le descripteur *numéro* (champ FD)
- **-i :** affiche les connexions
- **-F format :** permet de formater la sortie
- le symbole ^ inverse la sélection
- **-a :** permet d'effectuer un AND logique
- **-c chaîne** : affiche les fichiers ouverts pour les processus dont le nom commence par chaîne
- **fichier** : donne les processus qui ouvrent le fichier donné
- **+D répertoire :** donne la liste des processus qui ouvrent des fichiers contenus dans le répertoire
- **-t :** affiche uniquement le numéro du processus

lsof : exemples

```
lsof
lsof -u ^gilles
lsof -p 20012
lsof -u gilles -a -u root
lsof /var/log/syslog
lsof +D /var/log/
kill -9 `lsof -t -u lakshmanan`
lsof -d cwd
lsof -iTCP
lsof -i:22
lsof -i@192.168.0.32
lsof -i -sTCP:LISTEN
lsof -i -sTCP:ESTABLISHED
```

Utiliser gdb

Rubriques

- Premiers pas
- Affichage des variables

Premiers pas

Présentation de gdb

- Le programme central permettant de débuguer s'appelle *gdb*, c'est un débugueur en mode commande sans interface graphique.
- Il existe plusieurs débugueurs graphiques qui s'appuient sur gdb sans fournir toutes ses fonctionnalités.
- Il est donc intéressant de connaître les possibilités de gdb pour savoir se passer de débugueur graphique dans certains cas.
- Il est des fonctions que le débugueur *gdb* possède et qui sont rares dans les débugueurs graphiques :
  - sortie des traces sur fichier
  - interface avec le shell
  - historique avec rappel des commandes et complétion
- gdb peut être chargé sur un équipement embarqué en phase de test, là où les débugueur graphiques sont très gourmands en ressource
- gdb supporte les langages suivants : C, C++, D, Go, ObjectiveC, Fortran, Pascal, Mosula2, Ada

Principe gdb

- Pour lancer gdb avec un programme compilé avec l'option -g
- Pour lancer gdb :

```
gdb programme
```

- **start** : exécute le programme et s'arrête à la première ligne
- **list** : liste les lignes du programme précédées par leur numéro
- **break 10**: pose un break à la ligne 10
- **continue** : va au break suivant ou finit le programme si pas de break
- **next** : va à la ligne suivante sans entrer dans la fonction si la ligne suivante est un appel de fonction
- **step** : va à la ligne suivante et entre dans la fonction si la ligne suivante est un appel de fonction
- **print i** : affiche la variable i
- **quit** : quitter gdb
- **info break** : liste les break en affichant leur numéro
- **delete 3** : supprimer le 3 eme break
- **clear 3** : supprime le break de la ligne 3

Signals

- gdb détecte l'arrivée des signaux UNIX, on peut préparer le le comportement de gdb pour chaque signal
- **info signals [numero]:** affiche le comportement de gdb pour chacun des signaux
- **handle signal MOT CLE:** indique le comportement à avoir sur chacun des signaux. MOT CLE peut être :
  - **nostop :** gdb ne stoppe pas le programme quand le signal est reçu
  - **stop :** gdb stoppe le programme quand le signal est reçu
  - **print :** le signal provoque l'affichage d'un message
  - **printstop :** le signal ne provoque pas l'affichage d'un message
  - **pass :** gdb laisse le programme voir le message-
  - **passstop :** gdb ne laisse pas le programme voir le message

Revenir en arrière

- Tous les OS n'acceptent pas de revenir en arrière, car cela nécessite une gestion sophistiquée des registres. Linux accepte en mode Intel 32 bits ou 64 bits, VMWARE également
- D'autre part cette option n'est disponible qu'avec la version gdb 7.0 minimum
- Les distributions n'incluent pas systématiquement l'option arrière jugée non stable. Il faut donc recompiler gdb dans certains cas.
- Les commandes step, stepi, next, nexti, finish et continue admettent le préfix *reverse-* pour rembobiner la séquence en sens inverse avec les arguments de compteur identique à la version non reverse

```
reverse-step 2
```

La pile d'appel

- Quand le debugueur est arrêté sur une instruction, il peut être intéressant de déterminer comment on est arrivé à l'instruction.
- Chaque fois qu'une procédure est appelée, toutes les informations d'appel sont stockées dans un bloc d'information appelé le *stack frame* ou cadre de pile. Chaque appel provoque la création d'un nouveau frame.
- la commande `frame` remonte dans la liste des frames emmagasinées et disponibles, la commande `select-frame` produit le même effet sans afficher d'informations à l'écran

```
frame  2
select-frame 2
```

- **frame 0** : permet de revenir au niveau actuel
- **backtrace :** ou bt sans argument donne la liste des frames appelées pour arriver au point courant
- **bt n :** montre les n premières instances de pile
- **bt -n :** montre les n dernières instances de pile
- **up [n] :** remonte de n occurrences dans la pile
- **down [n] :** descend dans la pile
- **where :** affiche le niveau actuel dans la pile

Affichage des variables

Modification du contexte

- **print variable=valeur :** change la variable C ou C++ en valeur et affiche cette valeur. Valeur peut être calculée :

```
print i=j+2
```

- **set var variable=valeur :** la même chose sans affichage de la valeur.

```
set var i=2
set {int}0x83040 = 4
```

- la variable $pc est l'adresse d'exécution ou compteur de registre on peut l'altérer comme une variable, tout comme $sp qui est le pointeur de pile.
- On préfère cependant utilise la commande `jump` vers une adresse qui

provoque un continue à partir de l'adresse donnée en sautant les instructions entre le $pc et la ligne de code demandée.

```
break 32
jump 32
```

print variable

- La commande la plus basique pour afficher une variable est la commande *print* ou son abréviation *p*

```
p i
$1 = 1
```

- l'affichage concerne la frame en cours, ce qui fait qu'en cas de remontée dans la frame, on visualise la valeur de la variable dans la frame.
- la variable doit être accessible au moment où on demande son affichage
- les variables statiques peuvent être affichées sous la forme `fichier::variable` ou `fonction::variable`

```
print 'main.c::i'
```

- le symbole :: peut être utilisé pour donner la référence d'une fonction, c'est à dire différencier une variable d'une fonction à une variable de même nom. En cas de conflit de notation, la notation C++ prime.
- **print /f expression :** permet l'affichage d'une expression suivant le format f
- **f** peut être : x : impression hexadécimal, d : entier décimal, u : décimal signé, o: octal, a : format adresse, c: caractères, f : flottant, s : string ...

Autres affichages

- La commande `display /fmt` affiche une variable après chaque instruction, ce qui est pratique pour suivre une ou plusieurs variables
- La commande *explore* permet d'afficher la valeur et le type d'une variable

```
explore valeur
```

- Affichage de zone mémoire : se fait par la commande x :

```
x [/nfu] adresse
```

- **n :** taille en unité à afficher
- **u :** b pour 1 octet, h pour 2 octets, w pour 4 octets, g 8 octets
- **f :** format x, d, u, o, t, a, c, f, s ou i pour voir l'instruction assembleur

```
x/3uh 0x54320
```

Printf dynamique

- *dprintf* combine un point d'arrêt à l'affichage de données le résultat d'un printf appels dans le programme, sans avoir à recompiler.
- La syntaxe est `dprintf adresse,format, expression1,...`

```
dprintf 45,"i=%d,j=%d\n",i,j
```

- La commande `save breackpoints fichier` qui sauvegarde l'ensemble des points d'arrêt sauvegarde également les printf dynamiques.
- La fonction `set dprintf-style style` permet de stipuler la fonction d'impression suivant la valeur de style :
  - `gdb` : la fonction printf de gdb
  - `call` : une fonction du programme déterminée par `set dprintf-function`
  - `agent` : en cas compilation à distance. Dans ce cas, il faut que gdbserver accepte le dprintf distant(>2014)
- `set dprintf-function fonction` : donne la fonction d'impression si style = call
- `set dprintf-channel canal` : permet de déporter le debug sur un log par exemple, il suffit que le IO/Stream standard soit assigné à la variable `canal`

Break avec liste de commandes

- les watchdogs, les breakpoints et les catchpoints peuvent être assignés à des suites de commandes
- les commandes sont encadrées par les mots clés `commands` et `end`
- à l'intérieur on utilise généralement le mot clé `silent` qui indique au break de rester en mode silencieux on inclue un ordre continue, step ou next, si on souhaite que le point d'arrêt se comporte comme un point de modification de code et non plus comme un point d'arrêt

```
break 567
commands
silent
set x = y + 5
continue
end
```

- Pour détruire une liste de commandes on déclare une séquence vide entre `commands` et `end`

Debug graphiques

Rubriques

- ddd
- Eclipse cdt

ddd

Clic droit dans le code source

- Sur les variables
  - Pour afficher les variables bouton droit dans le code sur la variable afin d'obtenir un menu permettant d'obtenir les fonctions *print*, *display*, *whatis* ou de poser un *break*
  - Une fois un break posé, on peut par un clic droit dessus, mettre une condition sur le break, mettre un compteur, l'activer ou désactiver
  - possibilité plus rare il implémente le `commands` de gdb permettant d'enregistrer plusieurs commandes sur un point d'arrêt.
- Sur la première colonne le clic droit provoque l'apparition d'un menu qui permet :
  - de mettre un *breakpoint*
  - de mettre un *breakpoint temporaire*
  - d'exécuter *until* jusqu'à l'adresse

La fonction Display

- La fonction *display* (click droit) affiche les variable dans la zone supérieure du débugueur, les tables sont affichées sous forme de tableaux
- Les Listes chaînées permettent par un double click d'afficher l'élément suivant ce que l'on peut tester par l'exemple suivant :

```
#include <malloc.h>

typedef struct elem elem;
    struct elem {
    int i ;
    struct elem *next;
} ;
int main()
{
    elem  *p; elem *first;  elem *last;
    p = malloc (sizeof(elem));p->i=1;p->next=0;
    last=p; first=p; p=first;
    p = malloc (sizeof(elem));p->i=2;p->next=0; last->next=p;last=p;
    p = malloc (sizeof(elem));p->i=3;p->next=0; last->next=p;last=p;
    return (0);
```

La fonction plot

- Cette fonction nécessite que *gnuplot* soit installé (apt-get install gnuplot)
- Passer dans *Edit/Preferences/Helpers/Plot Windows* sur *external*
- L'appui dans la barre d'outil sur la fonction plot permet de dessiner le tableau graphiquement

```
int main(int argc, char **pArgs)
{
  int pTab[10];
  int i;

  for(i=0;i<10;i++)
    {
      pTab[i] = i*i;
    }
}
```

Eclipse cdt

Présentation

- L'installation d'eclipse pour C/C++ se fait sous Debian par le paquetage *eclipse-cdt* et donc par la commande

```
apt-get install eclipse-cdt
```

- L'installation demande 250 Mo sur le disque
- Eclipse intégrant le mode debug en même temps que le mode éditeur, il n'est pas toujours facile de trouver les fonctions propres au debug si on ne sait pas qu'on doit entrer dans la perspective Debug pour y accéder.
- Il faut s'assurer au préalable que les programmes sont créés en mode Debug et Release
- Une fois le programme compilé on passe en Vue Debug en actionnant l'icone de debug pour entrer dans l'univers de Debug où sont accessibles les fonctions *step*, *next*, *skip*
- Eclipse permet l'ajout de point d'arrêts en première colonne du code source avant d'entrer en mode Debug
- Les points d'arrêt supportent l'affichage dynamique, les commandes, le mode enable/disable, les compteurs
- L'ajout des Watch se fait dans la zone Éditeur par le bouton droit en mode Debug

Ajout de points d'arrêt

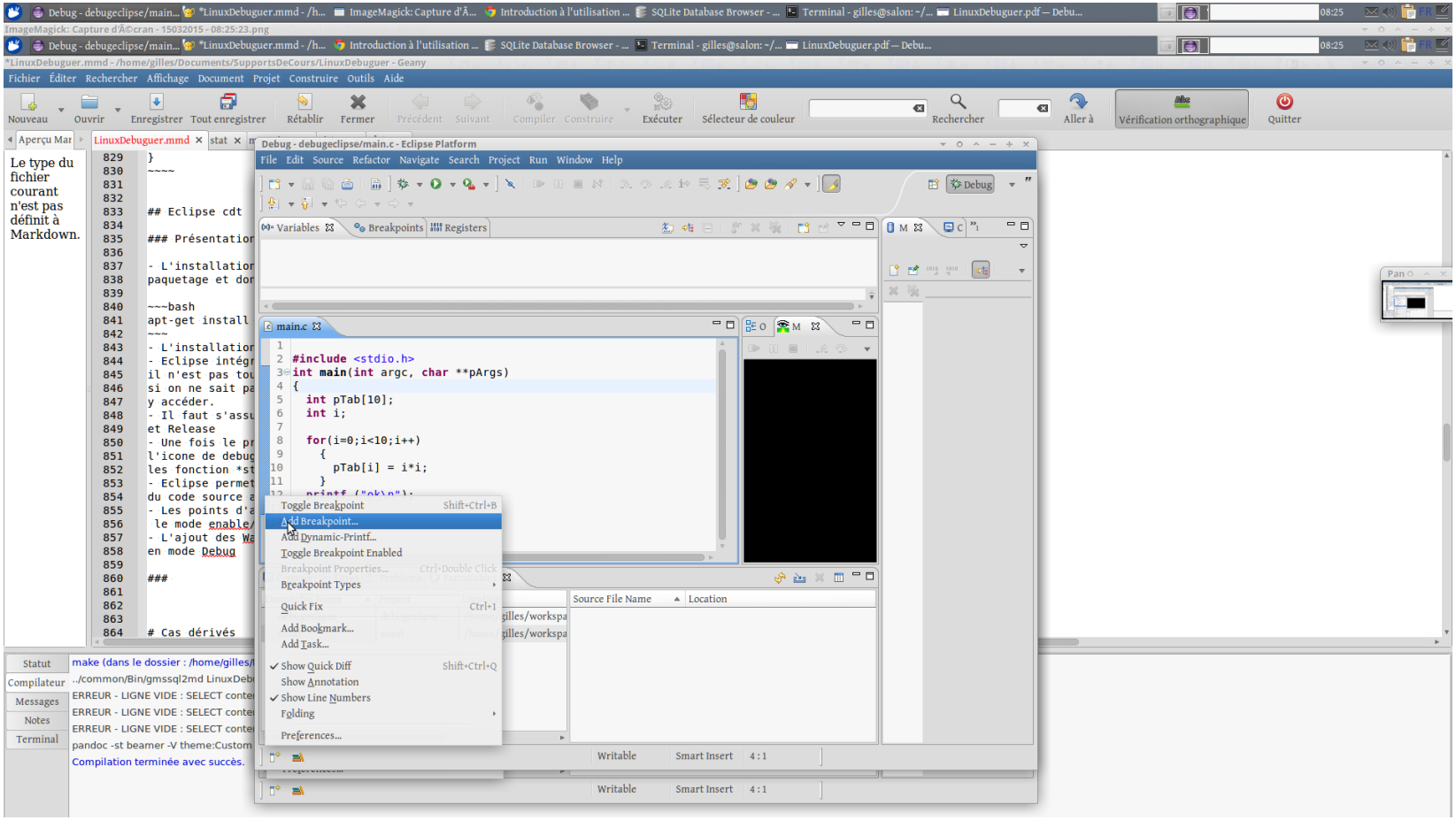


Figure 1: Ajout d'un point d'arrêt

Propriété du projet en mode Debug

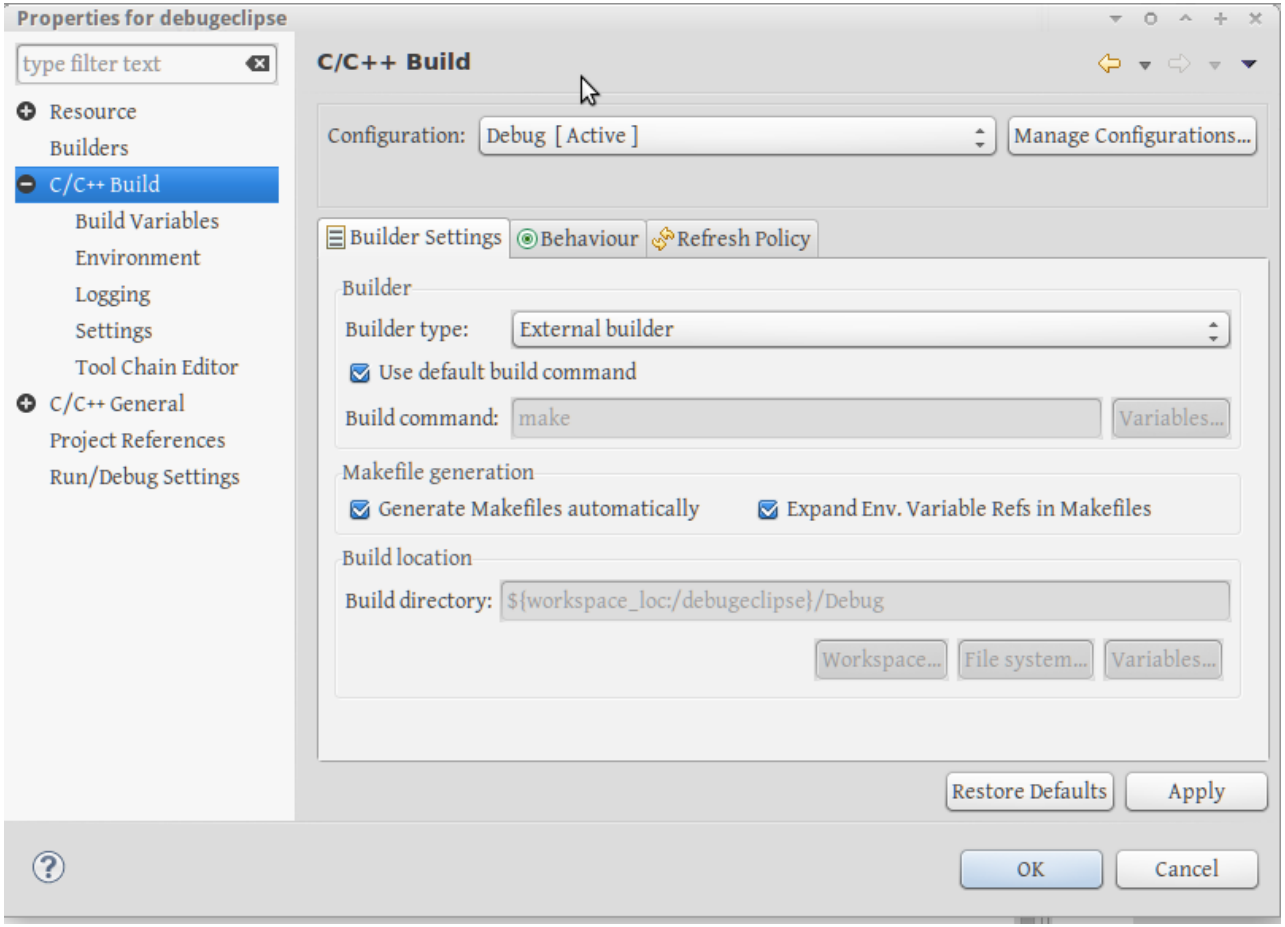


Figure 2: Projet en mode Debug

Accès à la perspective Debug

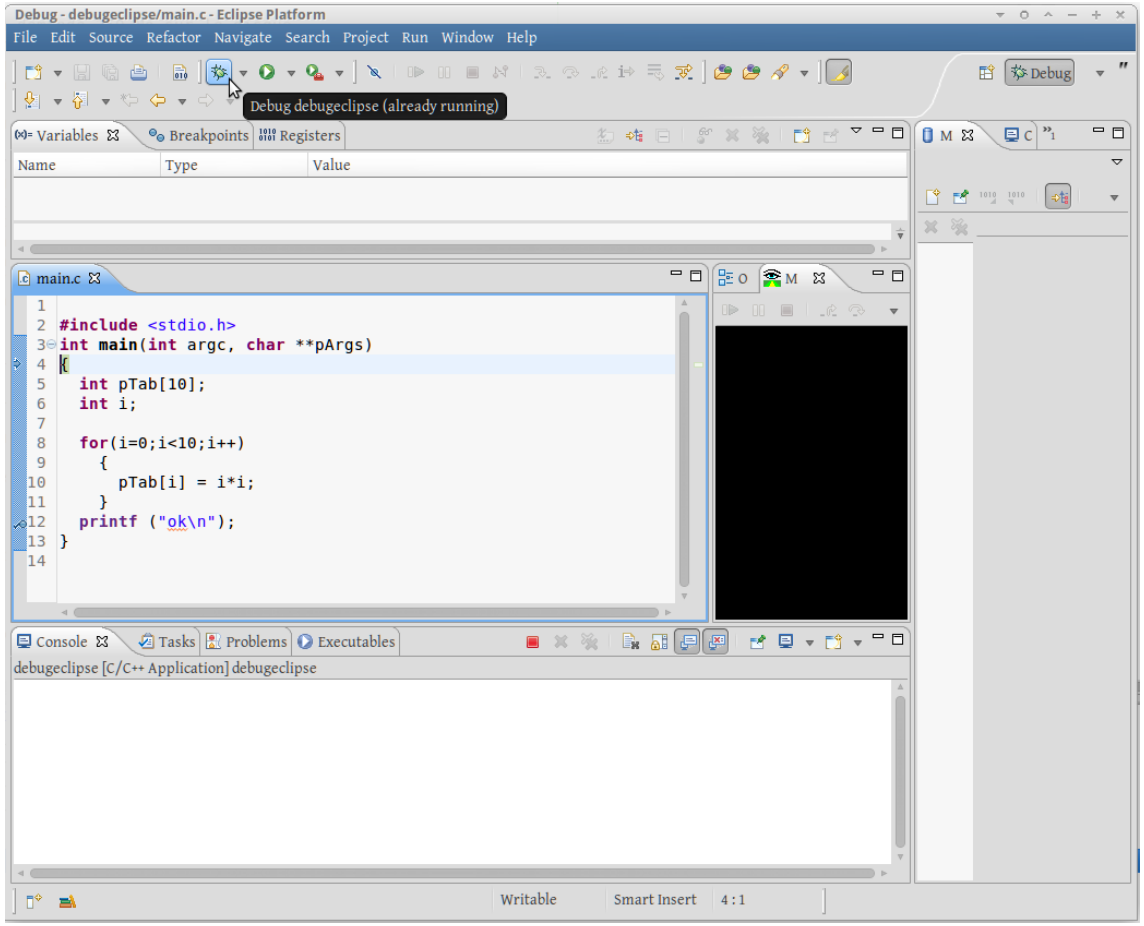


Figure 3: Accès perspective Debug

Perspective Debug

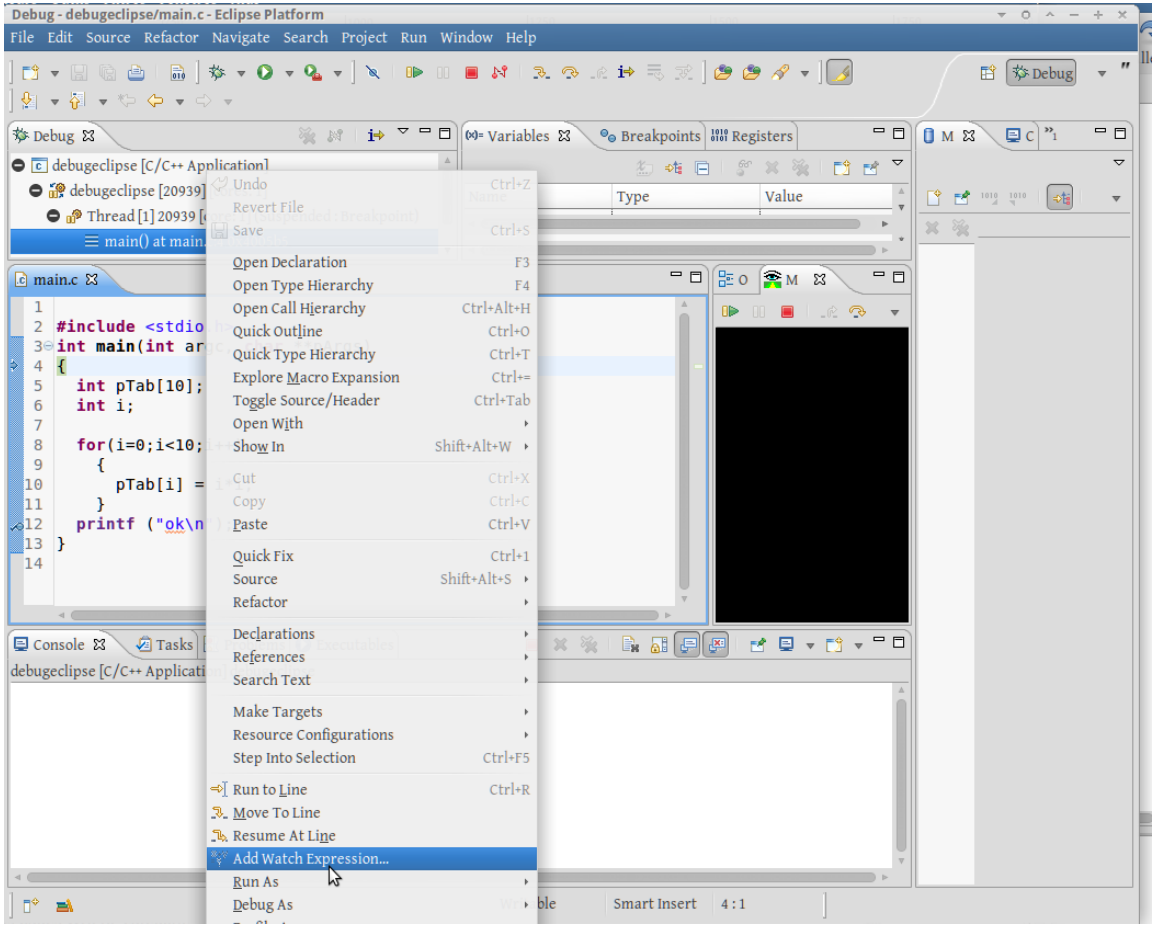


Figure 4: Accès aux views

Cas avancés

Rubriques

- Debuguer un programme en exécution
- Debuguer plusieurs programmes simultanément
- Debuguer via un core dump
- Debuguer à distance

Debuguer un programme en exécution

Contraintes

- Si on souhaite voir les sources du programme en mode debug, il faut que le programme en cours d'exécution ait été compilé en mode -g
- Attention : certaines configurations Linux interdisent de tracer les processus non fils aux utilisateurs non root
- Si on utilise gdb on peut passer en mode root pour lever cette interdiction. Par contre lancer Eclipse ou ddd en mode root n'est pas toujours possible.
- Pour autoriser les utilisateurs non root à tracer des programmes on peut également procéder comme suit :
  - Pour un usage permanent éditer le fichier `/etc/sysctl.d/10-ptrace.conf` et positionner la ligne `kernel.yama.ptrace_scope = 1` à valeur 0
  - Pour un usage ponctuel exécuter `echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope`
- on récupère le PID du programme tournant par

```
ps awx | grep programme
```

Sous gdb ou ddd

- on peut attraper le programme tournant de deux façons :

```
gdb -p 12122
```

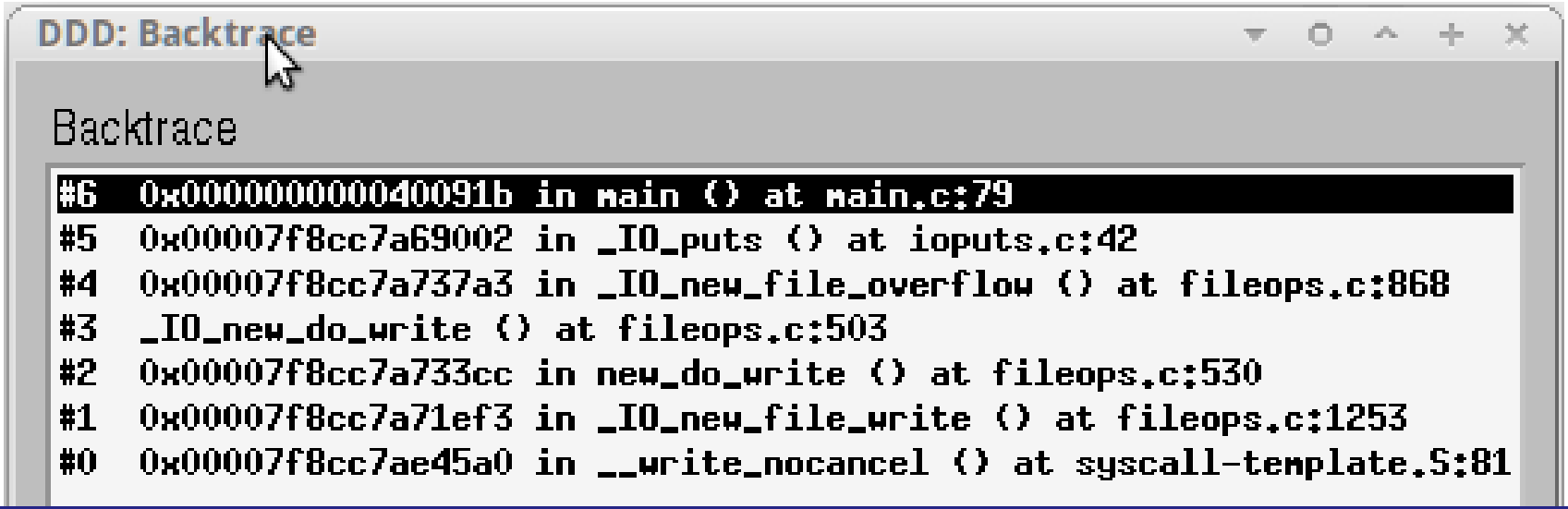
ou

```
gdb
attach 12122
```

- une fois le processus attaché, l'exécution est stoppée et on peut prendre la main dans le débugueur.
- une commande `where` permet de connaître les frames et on exécute une commande up autant de fois que nécessaire pour se retrouver dans une fonction dont on possède les sources.
- une fois qu'on lancera la commande detach (ou qu'on quittera le Debuguer), l'exécution du processus reprendra normalement.

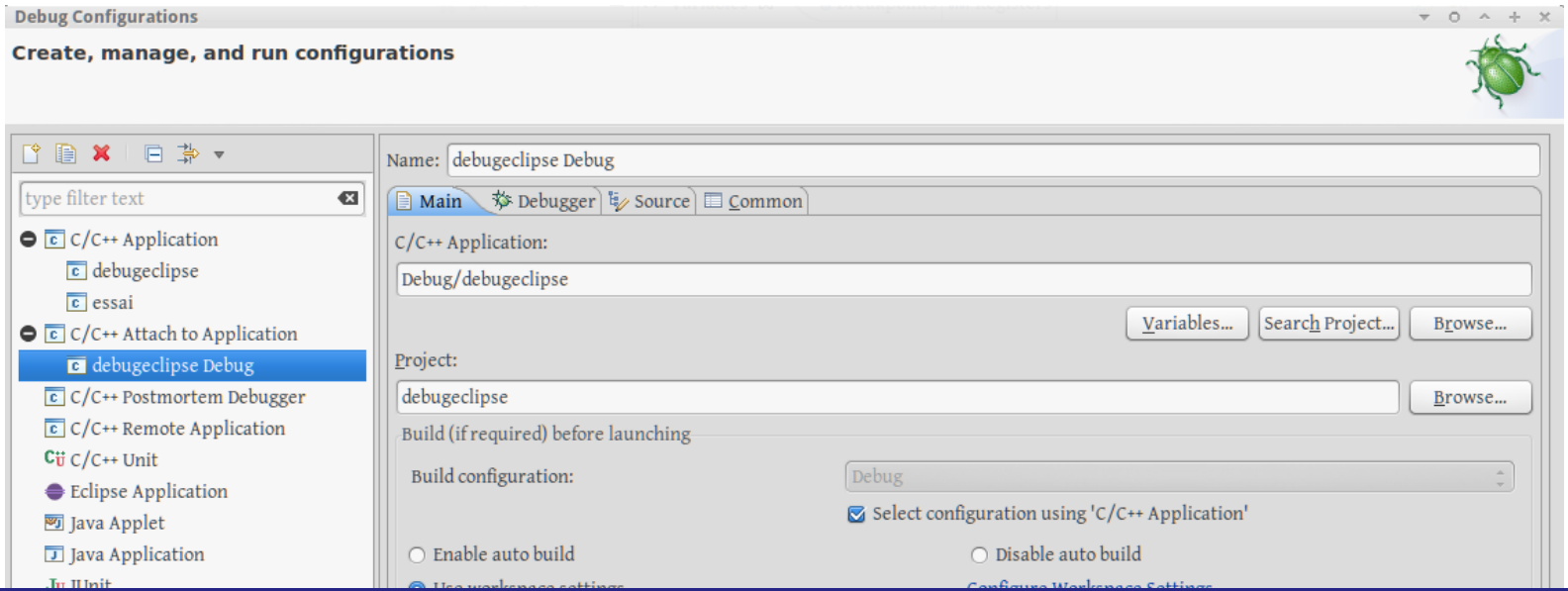
Cas de ddd

- ddd prend l'option -p numéro de process en argument ce qui permet de parvenir au même résultat avec ddd.
- On peut aussi lancer ddd sans option de process, ouvrir le binaire afin de charger le code source.
- Puis *Menu File/Attach to Process*, sélectionner le process dans la liste des processus présentés.
- Ensuite la fenêtre de frame est présentée par le menu *Status/Backtrace*



Cas d'Eclipse

- Sous Eclipse, il faut que le projet concernant le programme existe et qu'il soit ouvert.
- Ensuite aller dans le menu *Run/Debug Configuration*
- Cliquer sur *C/C++ Attach to Application*
- bouton droit dans la partie droite et *New* puis après avoir rempli le nom de l'exécutable cliquer sur *Debug* la liste des process tournant est présentée et il faut trouver le processus correspondant.



Debuguer plusieurs programmes simultanément

Sélection de la frame comprenant les sources

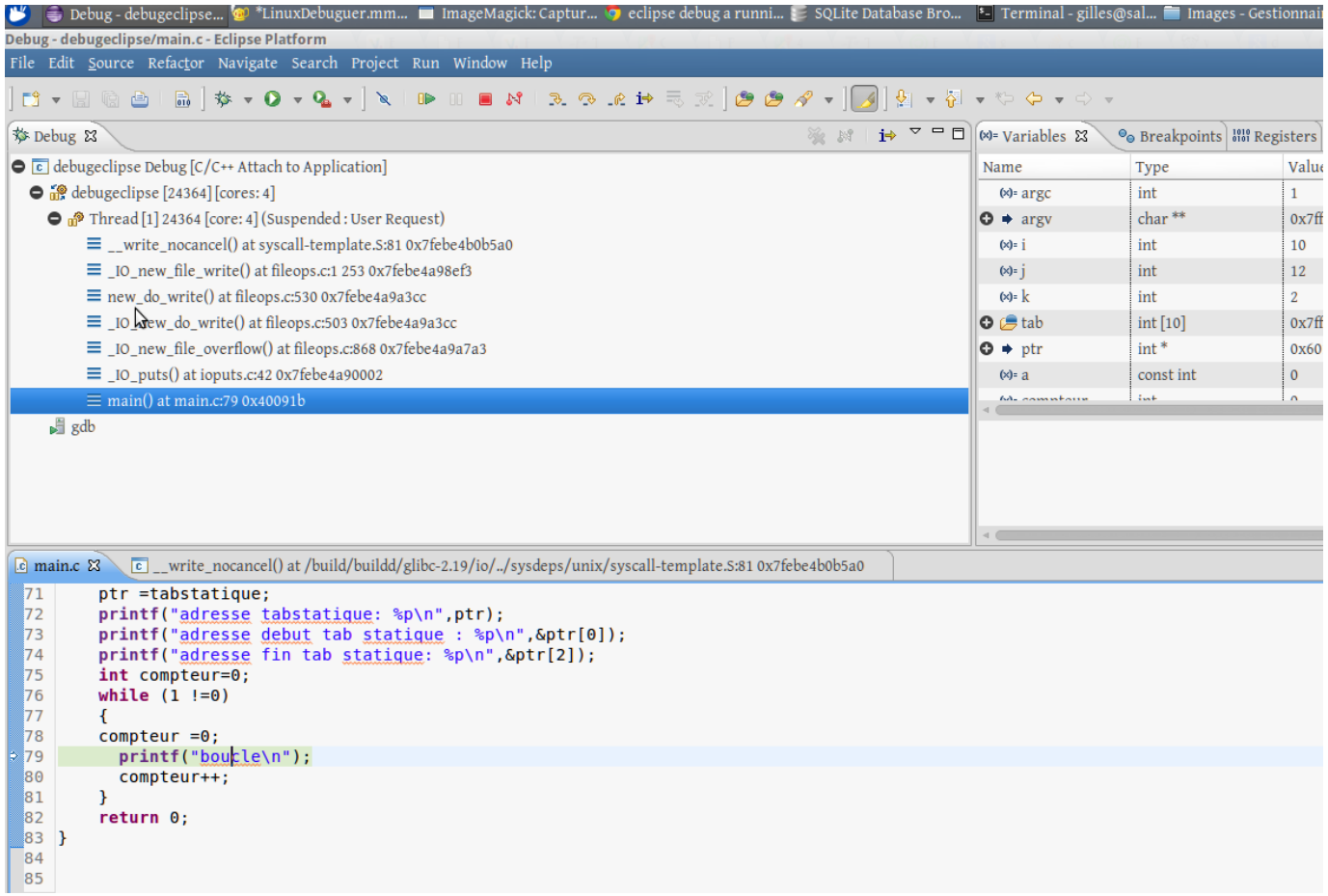


Figure 7: Accès aux views

Exposé

- *gdb* permet de débuguer plusieurs programmes simultanément y compris s'ils tournent
- *gdb* représente l'état de chaque programme qui s'exécute par un objet appelé *inférieur*, cet *inférieur* peut être créé avant que le processus tourne et peut être conservé après son exécution.
- pour afficher la liste des inférieurs des programmes en cours de débugage :

```
info inferiors
```

- Cette commande affiche un * devant le processus courant, le numéro d'ordre du programme, une information par défaut nulle et le

nom du programme. - Pour ajouter une exécution

```
add-inferior -exec executable
```

- Pour rendre le focus sur un autre inférieur :

```
inferior numero
```

- on peut utiliser *clone-inferior*, *remove-inferior*, *detach-inferior* et *kill-inferiors num*

Debuguer via un core dump

Cas d'Eclipse et de ddd

- ddd ne supporte pas cette option
- Eclipse la supporte en permettant d'ouvrir plusieurs projets en parallèle

Debug après plantage : core dump

- Sous Linux on peut facilement mettre en place un mécanisme permettant de débuguer un programme en analysant après coup un fichier trace appelé *core*.
- C'est via le débugueur que ce fichier va être rejoué.
- Il faut cependant s'assurer que ce fichier est généré en cas de plantage, ce qui n'est pas forcément le cas sur toutes les plate-formes.
- Il faut également connaître précisément la version du logiciel qui a planté et être en mesure de disposer des mêmes sources. Ceci est rendu possible par l'utilisation de gestionnaires de versions dont le plus répandu est aujourd'hui *git*.
- Il faut donc savoir précisément actionner la génération d'un fichier core (core dump)
- Il est conseillé de renseigner la version git à l'intérieur de l'exécutable afin de savoir la retrouver via l'exploration d'une variable par exemple.
- En outre, il faut garder à l'esprit que beaucoup de distributions génèrent ce fichier core mais laissent le programme Apport l'intercepter, en demandant l'autorisation à l'utilisateur d'envoyer le core au développeur. Il nous faut donc savoir désactiver ce mécanisme

Tuning de génération du core

- Vérification qu'un fichier core n'est pas intercepté

```
cat /proc/sys/kernel/core_pattern
```

- par exemple par `|/usr/share/apport/apport %p %s %c`
- que le fichier /etc/security/limits.conf contienne * soft core unlimited et * hard core unlimited
- Si on veut que la modification soit temporaire, on exécute

```
- en root : echo /tmp/core.prog_%e.sig_%s.proc_%p>/proc/sys/kernel/core_pattern
- en utilisateur : ulimit -c unlimited
```

- Si on veut qu'elle soit permanente :
  - ulimit se règle sous Debian en mettant dans le fichier `/etc/security/limits.conf` : `* soft core unlimited`
  - ajouter `kernel.core_pattern = /tmp/core.prog_%e.sig_%s.proc_%p` dans un fichier `/etc/sysctl.d/50-coredump.conf`
  - supprimer le programme apport (apt-get remove apport)
- On peut vérifier que le répertoire /tmp contient bien un fichier core après lancement de : `sleep 100 & kill -SIGSEGV %%` (%% est une variable bash représentant le PID de la dernière tâche suspendue)

Technique de débugage du core

- On s'assure que le programme qui a produit le core était bien compilé en mode débug
- Si on utilise gdb on lance :

```
gdb programme fichiercore
```

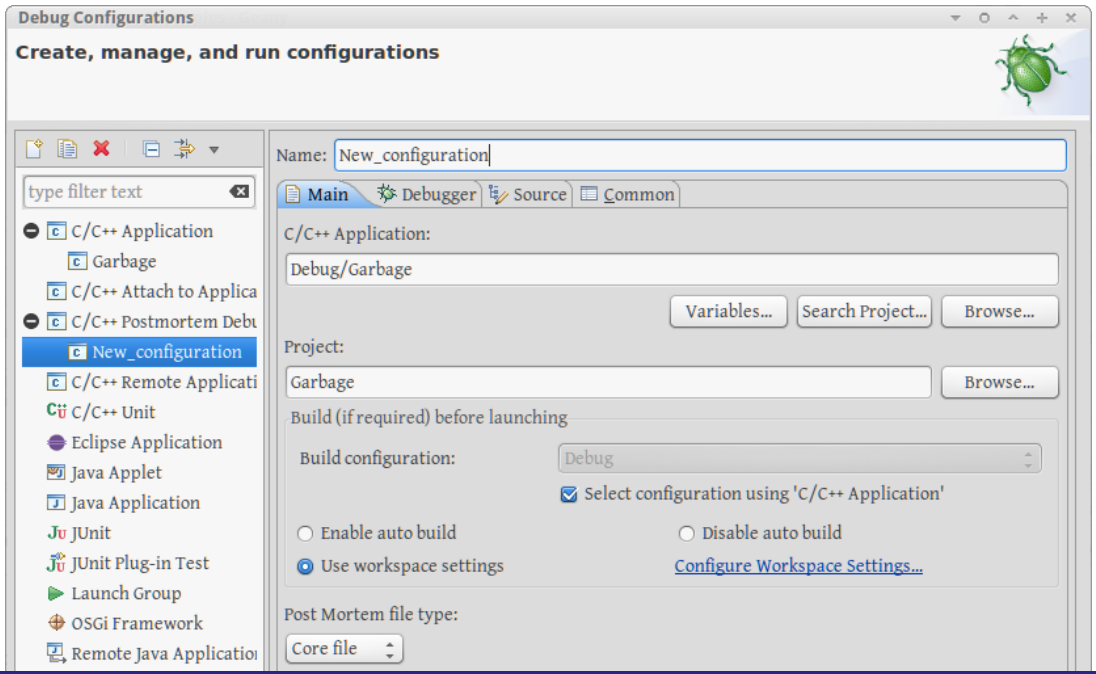
- le programme doit être généré avec les options de debug
- le fichier core doit être de la forme `/tmp/core.programm_bash.sig_11.proc_29255`
- Le débugueur montre l'instruction qui a provoqué le plantage
- on remonte via autant de commandes *up* que nécessaire jusqu'à trouver son code
- on inspecte sous le débugueur les variables afin de comprendre pourquoi l'instruction a provoqué le plantage

Debuguer à distance

Debugage du core via Eclipse

- Une fois le programme compilé avec la fenêtre de dialogue d'options

debug ouverte, on actionne le menu *Run/Debug Configurations*.
- Dans la fenêtre de dialogue aller à la rubrique *C/C++ Postmortem Debugger*, remplir le champ C/C++ Application - Remplir le champ *Core file* - Cliquer sur le bouton *Debug*



Debug croisé

- On appelle debug à distance la possibilité de débuguer un logiciel tournant sur un équipement distant depuis un autre équipement.
- Ce cas est très utile en embarqué où les machines cibles sont pourvues de peu d'espace de stockage et munis de processeurs de petite capacité demandant des compilations croisées. C'est pourquoi cette technique s'appelle également debug croisé.
- *gdb* permet le débug croisé via un port série ou via TCP/IP
- *gdb* permet de débuguer un code distant sur un processeur différent du processeur de la machine possédant le débugueur. Il gère même les cas de byte ordering différent.
- Sur la carte distante, on rencontre deux cas :
  - un serveur gdb est installé car on est sur une machine unixlike, on communique de débugueur client à débugueur serveur
  - un serveur n'est pas installé dans ce cas on doit implémenter sur le logiciel de la carte le protocole stub de communication de gdb.
- Nous nous bornerons aux cas d'utilisation avec le protocole gdbserver en mode TCP/IP

Installation sur l'équipement distant

- nous supposons que ce serveur a l'adresse 192.168.0.237
- sur le serveur distant installer gdbserver
  - sur architecture debian : `apt-get install gdbserver`
  - sur embarqué busybox disponible dans les config
- La machine distante et la machine host doivent disposer d'un binaire construit à partir des mêmes sources avec les librairies idoines
- sur la machine distante : le binaire compilé sans option -g, stripé ou non stripé, les sources ne sont pas nécessaires
- Sur le serveur distant on lance le serveur gdbserver avec la syntaxe suivante :

```
gdbserver 192.168.0.32:2345 programme arguments
```

- 192.168.0.32 est l'adresse IP de la machine Host
- 2345 est le port TCP que vous choisissez
- programme est le programme que vous débuguez juste compilé

Mise en route sur l'équipement host

- Nous supposons que cet équipement a l'adresse 192.168.0.32
- Nous compilons le programme avec l'option -g
- Sur le poste de travail du débugueur on lance

```
gdb nom du programme
```

- On peut ainsi débuguer à distance après avoir informé le débugueur par la première commande :

```
target remote 192.168.0.237:2345
```

- le programme étant lancé on fait généralement un **break main** ou

un **continue**

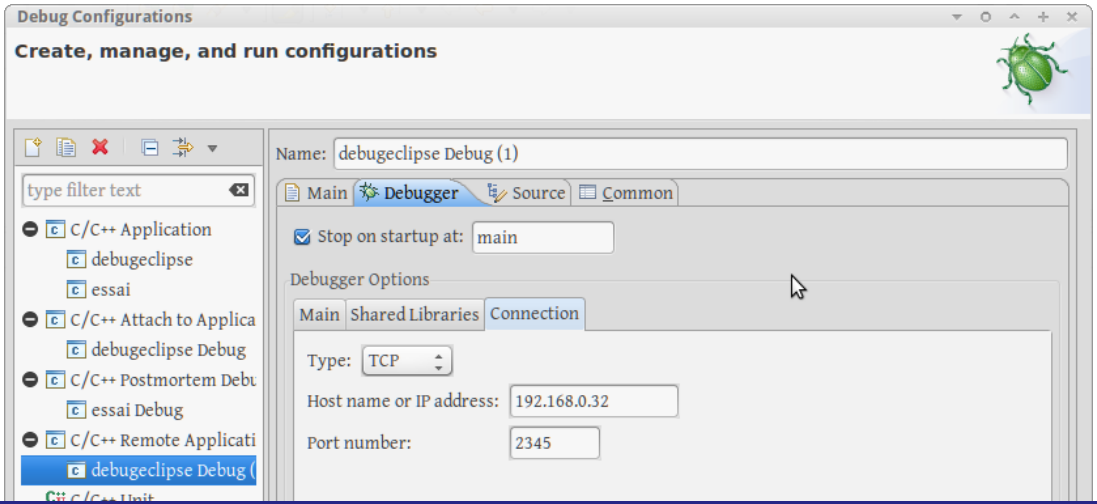
Remarques sur le debug croisé

- Depuis fin 2013 de nouveaux protocoles ont fait leur apparition d'abord sur le client gdb 7.5 puis sur le serveur gdb.
- Ceci rend compliqué le protocole entre un équipement d'après 2014 et un équipement d'avant 2014
- Plus d'information sur https://sourceware.org/gdb/wiki/LinkerInterface
- En embarqué si vous utilisez busybox, les dernières versions permettent de choisir des versions compatibles en fonction de votre host.
- Utilisation avec ddd :
  - soit mettre `put target remote 192.168.0.237:3456` dans un fichier et utiliser l'option `--command fichier` pour l'utiliser
  - soit mettre `put target remote 92.168.0.237:3456` dans le fichier .gdbinit dans votre répertoire d'accueil
  - soit lancer

```
ddd --eval-command="target remote 192.168.0.237:1234" main
```

Mise en oeuvre sur Eclipse

- Sous Eclipse, il faut que le projet concernant le programme existe et qu'il soit ouvert.
- Ensuite aller dans le menu *Run/Debug Configuration*
- Cliquer sur *C/C++ Remote Application*
- bouton droit dans la partie droite et *New* puis après avoir rempli le nom de l'exécutable, cliquer sur l'onglet *Debugger*, puis sur l'onglet *Connection* où vous pouvez remplir :
  - type TCP/Série
  - les informations d'IP et de port ou de device série et de vitesse



Valgrind

Rubriques

- Présentation
- Exemples
- Profile de code
- Présentation
- Utilisation

Présentation

Fonctionnalités

- Valgrind est plus qu'un outil c'est presqu'un Framework pour fabriquer ses propres outils de débugage
- Valgrind inclut les 9 outils suivants :
  - **memcheck** détecteur d'erreur mémoire principalement pour les programmes C/C++
  - deux détecteurs de thread **helgrind** et **DRD**
  - **cachegrind** un profiler de branche et de cache afin de diagnostiquer où les programmes perdent du temps
  - **callgrind** un générateur de graphe permettant d'illustrer le travail de cachegrind
  - **helgrind** un détecteur d'erreur sur l'utilisation des threads
  - **DRD** un autre détecteur de problèmes de thread
  - **massif** un profileur de pile
  - **SGcheck** outil de dépassement de pile et de tableau
  - **DHAT** un autre profileur de pile
  - **BBV** un générateur de point de blocage

Diverses informations

- Plate-formes supportées
  - X86/Linux, AMD64/Linux
  - ARM/Linux, ARM64/Linux, S390X/Linux,
  - PPC32/Linux, PPC64/Linux, PPC64BE/Linux
  - MIPS32/Linux, MIPS64/Linux,
  - Android ARM (2.3.x ++), X86(4.0 +), MIPS32/,
  - Darwin X86 et AMD64/Darwin (MacOSX)
- Ressources
  - http://valgrind.org/ avec une documentation en langue anglaise
  - Version 3.10
  - Dernière version novembre 2014
  - Aide sur #valgrind-dev sur le réseau irc.freenode.net

Installer Valgrind

- Sur souche Debian :

```
apt-get install valgrind
```

- via les sources

```
./configure; make ; make install
```

Philosophie d'utilisation

- On compile un projet puis on exécute *valgrind* sur le programme pour obtenir un fichier de résultat.
- Les options de compilation usuelles pour valgrind sont :
  - `-g` permet d'inclure les symboles de débug
  - `-O0` est conseillé
  - `-O1` et `-O2` suppriment quelques informations et sont moins recommandées
  - `-fno-inline` on supprime les fonctions *inline*
- Options pour exécution de valgrind en mode détection de fuite de mémoire:
  - une fois le programme compilé on appelle un programme qui se lancerait `programme option1 option2` par la syntaxe

```
valgrind --leak-check=yes programme option1 option2
```

Suppression de messages

- par l'option –default-suppressions=no aucune suppression de message n'est faite
- sinon on peut ajouter des fichiers décrivant les erreurs à ne pas afficher –suppressions=/path/to/file.supp
- Des outils comme Qtcreator ou Eclipse otn leur propre fichier de configuration de suppression de message

Exemples

Recommandations

- Valgrind va générer un grand nombre d'entrées sorties sur la console
- En conséquence, il est conseillé d'éliminer toutes les sorties console du programme avant de lancer valgrind
- De même, il est recommandé de ne pas avoir de warning à la compilation afin de limiter au maximum le nombre de problèmes possibles
- L'exécution du programme est considérablement ralentie par Valgrind, il est donc recommandé d'utiliser un équipement de débogage puissant

Programme C : dépassement de mémoire

```
1  #include <stdlib.h>
2
3  void f(void)
4    {
5      int* x = malloc(10 * sizeof(int));
6      x[10] = 0; // 1er problème : dépassement de la pile
7    }            // 2ème problème : perte de mémoire pas de free
8
9  int main(void)
10   {
11     f();
12     return 0;
13   }
```

Résultat

- On lance un programme compilé avec les options -O0 -g ~~bash valgrind ./programme~~

```
==10677== Invalid write of size 4
==10677==    at 0x400554: f (valgrind.c:6)
==10677==    by 0x400564: main (valgrind.c:11)
==10677==  Address 0x51fc068 is 0 bytes after a block
 of size 40 alloc'd
==10677==    at 0x4C2ABA0: malloc (in /usr/lib/valgrind
/vgpreload_memcheck-amd64-linux.so)
==10677==    by 0x400547: f (valgrind.c:5)
==10677==    by 0x400564: main (valgrind.c:11)
```

- le numéro 10677 est le numéro de processus
- Invalid write of size 4 : indique une écriture dans une zone non allouée

Exemple de programme

```
main
int condition ;
if ( condition ) {
 int i ;
 i++;
}
```

- deux problèmes d'initialisation condition et i

Profile de code

Deux compilations

```
g++  -o result essai.cppp
valgrind ./essai

Conditional jump or move depends on uninitialised value(s)
==18998==    at 0x4004FE: main (in result)
ERROR SUMMARY: 1 errors from 1 contexts

g++ -g -o result essai.cpp
valgrind ./essai

Conditional jump or move depends on uninitialised value(s)
==18998==    at 0x4004FE: main (variablenoinitialized.cpp:6)
ERROR SUMMARY: 1 errors from 1 contexts
```

Définition

- Un profil de code est un outil qui permet de repérer les fonctions et instructions qui prennent le plus de temps
- Généralement un programme prend 70% à 80% de son temps dans 30 à 20% du code
- On distingue :
  - le profil par échantillonnage : on regarde tous les n cycles la fonction en cours d'exécution
  - le profil par instrumentation : on modifie le code par ajout à chaque appel de fonction une fonction d'instrumentation.
  - le profil par émulation : on exécute le processus sur un processeur virtuel qui mesure toutes les instructions
- On utilise `valgrind` comme profileur avec l'option `--tool=callgrind`
- Le résultat est un fichier `callgrind.out.numéro` que l'on analyse avec le programme `KCacheGrind`

Avec Valgrind

```
gcc -g -o programme programme.cpp
    valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes \
        programme
kcachegrind callgrind.out.20561
```

- Pour associer les sources aller dans 'Settings/Configure KCacheGrind/Annotations
- On peut associer les sources pour chaque librairies et bien sûr pour le programme en indiquant le répertoire où se trouve les sources pour chaque ligne voulue
- Tracer les fuites de mémoire
- Usage de la mémoire
- Lecture et écriture de variables non allouées

Valgrind dans QtCreator

- Valgrind peut être utilisé comme application externe ou comme plugin de QtCreator et ce uniquement sous Linux
- Quel que soit la fonction demandée (analyse mémoire ou profiler) le mode opératoire est le même :
  - on exécute le programme en essayant de passer partout
  - une fois qu'on a quitté le programme, Valgrind affiche ses résultats
- Les temps d'exécution se trouvent très ralentis lors de l'éxecution de Valgrind
- On peut configurer Valgrind dans la partie Outils/Options/Analyseurs/Valgrind.
  - La partie gauche de la configuration concerne l'analyse mémoire
  - La partie droite concerne le profilage
- Via le menu Analyse/Analyseur de mémoire avec Valgrind on analyse les pertes de mémoire, ceci est à utiliser avec une compilation en mode débug
- Via le menu Analyse/Profiler de fonction mémoire, on analyse le temps passé dans chaque fonction ceci se fait avec ou sans le mode débug

Electric Fence

Présentation

Introduction

- Electric Fence permet de traquer les erreurs d'accès mémoire
- Il est fourni sous forme d'une librairie qui provoque un arrêt immédiat quand l'un des cas suivants se produit :
  - accès en dehors des zones allouées par la fonction *malloc()*
  - accès à une zone mémoire retournée au système par un appel à la fonction *free()*
  - détection des problèmes d'alignement.
- Il détecte aussi bien les problèmes d'accès en lecture qu'en écriture

Installation

- Sous souche Debian on installe la librairie via la commande :

```
sudo apt-get install electric-fence
```

- Sous Red Hat
- Via les sources à l'adresse https://directory.fsf.org/wiki/Electric_Fence

Utilisation

- On choisit d'utiliser la bibliothèque *libefence* explicitement lors de l'édition de liens en ajoutant l'option *-lefence* lors de l'appel à la commande gcc
- On peut également forcer l'édition de liens au moment de l'exécution grâce à la variable d'environnement LD_PRELOAD.
  - `LD_PRELOAD` permet de préciser les bibliothèques à utiliser avant d'aller les chercher dans les chemins par défaut

```
LD_PRELOAD=libefence.so.0.0 ./monprogramme
```

Exemple sur un programme C fautif

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int  *p ,i;
    p = (int p*)malloc(10*sizeof(int)) ;
    for ( i=0;i<=10;i++)
    p[i]=i;
    printf ("%d\n",i);
}
```

Compilation du programme C

- En mode normal :

```
gcc -o exemple exemple.cpp
./exemple
```

- En mode electric fence

```
LD_PRELOAD=libefence.so.0.0 ./exemple
Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens
<bruce@perens.com>
Erreur de segmentation (core dumped)
```

Utilisation

Débugage sous gdb

- On lance gdb ./exemple

```
(gdb) set environment LD_PRELOAD=libefence.so.0.0
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x8048520 in main (argc=1, argv=0xbffffd44) at eftest1.c:12
12          p[i]=i;
(gdb) print i
```

Exemple de programme C++ fautif

```
#include <iostream>

using namespace std;
int main()
{
    int  *p ,i;
    p = new int(10) ;
    for ( i=0;i<=10;i++)
    p[i]=i;
    std::cout<<i<<std::endl;
}
```

Compilation du programme

- En mode normal :

```
g++ -o exemple exemple.cpp
./exemple
```

- En mode electric fence

```
LD_PRELOAD=libefence.so.0.0 ./exemple
Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens
<bruce@perens.com>
Erreur de segmentation (core dumped)
```

Débugage

- On lance gdb ./exemple

```
(gdb) set environment LD_PRELOAD=libefence.so.0.0
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x8048520 in main (argc=1, argv=0xbffffd44) at eftest1.c:12
12          p[i]=i;
(gdb) print i
```

- utilisation : buffer overruns" et “buffer underruns”
- Analyse des plantages via EF
- Mise en pratique

CLang

Rubriques

- Présentation de LLVM et CLang
- Clang en pratique

Présentation de LLVM et CLang

Présentation de LLVM

- LLVM pour Low Level Virtual Machine est une infrastucture de compilateur permettant l'optimisation du code dans la phase de compilation, d'édition de liens, durant l'exécution en utilisant les temps morts pour effectuer des précalculs.
- LLVM est conçu pour tous les langages : C++, C ou autres et propose une machine virtuelle similaire à JAVA :
  - elle propose un compilateurJust in Time (JIT),
  - du bytecode JAVA
  - des interfaces à Python
- LLVM fournit un compilateur C, C++ et Objective C appelé CLang
- LLVM fournit des types de bases comme des entiers à taille fixe, des pointeurs, des tableaux, des vecteurs, des structures et des fonctions

Présentation de Clang

- Clang est un compilateur C/C++/ObjectiveC++ lié à LVVM, il est utilisé par Apple pour remédier aux faiblesses des compilateurs g++ et gcc
- Clang préserve la structure du code généré à la différence des compilateurs GNU qui optimisent et transforment le code. Cela veut dire que sous les compilateurs GNU l'assembleur généré n'est pas proche de l'assembleur généré.
- Clang fournit des rapports d'erreur plus détaillés, il permet d'indexer le code source et vérifier sa syntaxe en évitant les erreurs traditionnelles du C (variables non initialisées, pointeurs déférencés, etc)
- Clang est beaucoup plus rapide que gcc
- Clang fournit un analyseur statistique de code sous formes de checkers

Principaux Checkers de Clang

- core.CallAndMessage : arguments non initialisés
- core.DivideZero : division par zéro
- core.NullDereference : déréférence d'un pointeur
- core.uninitialized.UndefReturn : teste que les retours de fonction ne sont pas non initialisés
- core.uninitialized.Branch : un if n'utilise pas une valeur non itialisée
- core.uninitialized.ArraySubscript : une valeur non initialisée d'un tableau n'est pas utilisée
- core.UndefinedBinaryOperatorResult : le résultat d'une opération n'est pas fait à partir de variables non initilaisées
- cplusplus.NewDelete : teste qu'on ne fait pas de double free sur une allocation
- cplusplus.NewDeleteLeaks : teste qu'une allocation mémoire soit bien désallouée
- deadcode.DeadStores: teste que toutes les variables sont utilisées

Clang en pratique

Clang dans QtCreator

- Clang est utilisable dans QtCreator pour profiter des tests du compilateur Clang qui sont plus efficients que ceux des compilateurs GNU
- Dans le menu Outils, Options, Analyseurs, on peut vérifier que l'emplacement de Clang est correct
- Le menu Analyse/Clang permet de lancer une compilation en mode Clang bénéficiant des checkers de Clang. Pour chaque erreur, Clang montre la ligne défecteuse.