

Débogage de code*



Mardi 13 décembre 2011

Romaric DAVID

david@unistra.fr

Université de Strasbourg - Direction Informatique

Pôle HPC

***ou l'art de chercher la petite bête**

Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Introduction

Pôle HPC = le méso-centre de l'Université de Strasbourg

- ▶ Nous assistons les chercheurs à la mise en place de leurs applications sur nos ressources de calcul
- ▶ Nous assurons l'assistance au développement
- ▶ Cela nous conduit à inspecter de nombreux programmes

Ce cours a pour but de vous familiariser avec les techniques et les outils de débogage

Nous parlerons principalement d'outils libres

Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Débogage : définition

Le débogage consiste en l'analyse d'un programme présentant un comportement incorrect.

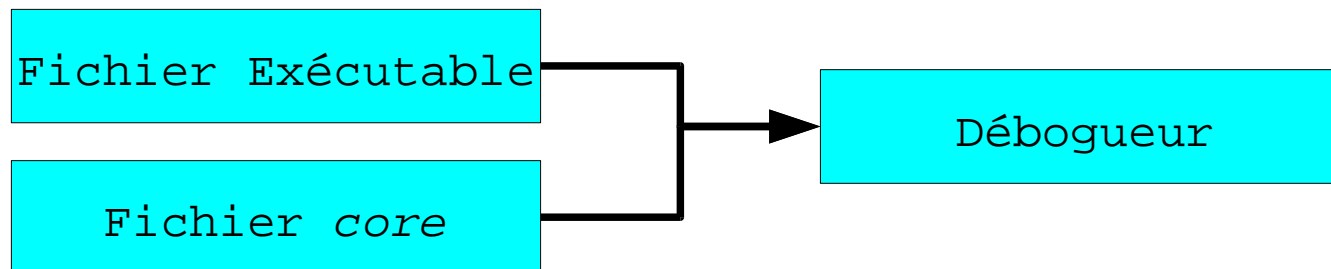
Un comportement incorrect peut être :

- ▶ Un plantage du programme
- ▶ Un programme « qui boucle », qui n'avance pas, etc...
- ▶ Un programme qui calcule faux

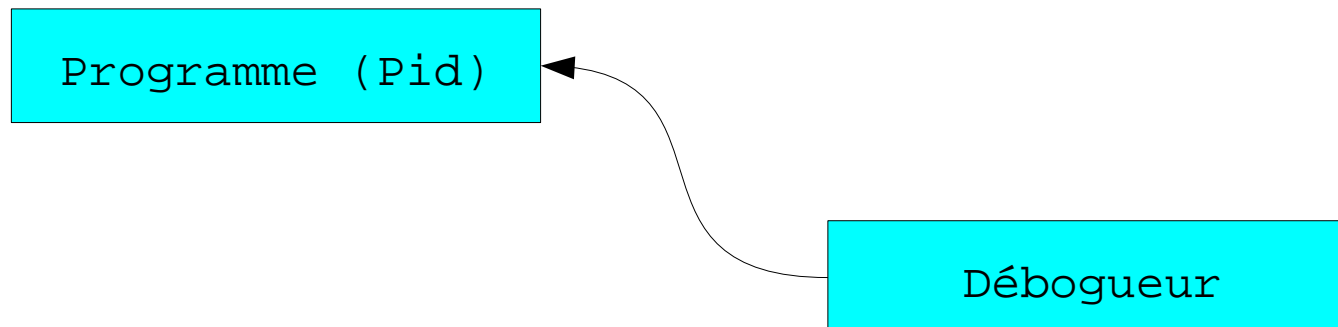
Débogage : définition

L'analyse du programme peut se réaliser :

► Post-mortem (plantage)



► In vivo (pré-plantage ou autres cas)



Débogage : apports

Les outils de débogage nous aident en dans la recherche de l'origine d'un problème :

- ▶ Identifier rapidement la ligne du code source du programme où se produit le comportement suspect, sans recherche par sortie écran (`printf`, `cout`)
- ▶ Contrôler le déroulement du programme en mode *pas à pas*, pratique pour arrêter le programme avant qu'il ne se plante
- ▶ Se concentrer sur les variables, en les explorant interactivement

Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Traceur d'appels système

- ▶ De notre expérience, un grand nombre de programmes se plantent dès le lancement.
- ▶ Très souvent lié à des soucis d'ouvertures de fichiers (applicatifs = fichiers d'input)
- ▶ Peut également venir de soucis d'allocation mémoire
- ▶ Permet de pallier la non prise en compte du retour d'erreur dans les programmes de calcul

Traceur d'appels système

- ▶ Outil utilisé : strace
- ▶ strace fonctionne en espace utilisateur
- ▶ `strace programme arguments`
- ▶ Produit une sortie conséquente : le programme vu par le prisme des appels système
 - Au niveau bas, l'ouverture d'un fichier se déroule via l'appel système `open`. Le résultat de `open` fera partie des éléments à regarder
 - Intéressant de comparer 2 cas : un où le code fonctionne, l'autre où il sort en erreur

Traceur d'appels système : exemple

- ▶ Principe : un appel système raté renvoie une valeur inférieure à 0
- ▶ `strace programme arguments >& fichier_sortie`
- ▶ Il faut creuser dans `fichier_sortie` pour trouver le problème

```
open("/usr/lib/locale/fr_FR.utf8/LC_CTYPE", O_RDONLY) = 3
```

```
open("/home/john/data.dat", O_RDONLY|O_LARGEFILE) = -1 ENOENT  
(No such file or directory)
```

```
open("/usr/share/locale/fr_FR.utf8/LC_MESSAGES/libc.mo",  
O_RDONLY) = -1 ENOENT (No such file or directory)
```

Émulateur

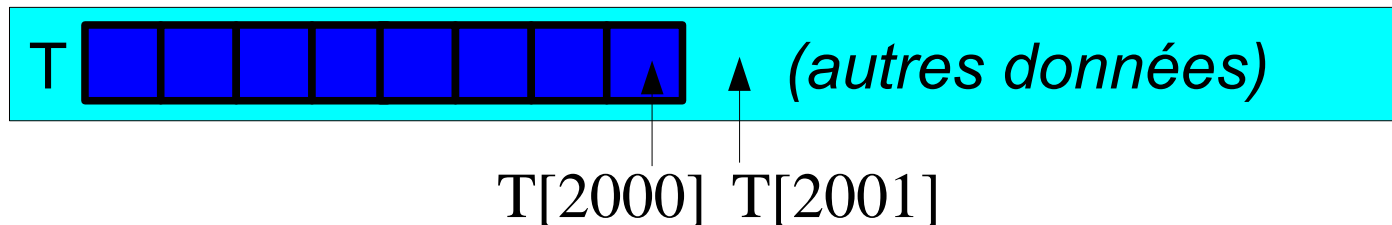
Une des grandes causes de dysfonctionnement est un accès mémoire erroné qui peut être causé par :

- ▶ Un accès à une zone non associée au programme (hors des *segments* affectés par le système) ;



Exemple : Accès par un pointeur mal initialisé. Sera signalé par le système

- ▶ Un accès à une case mémoire d'indice hors bornes d'un tableau.

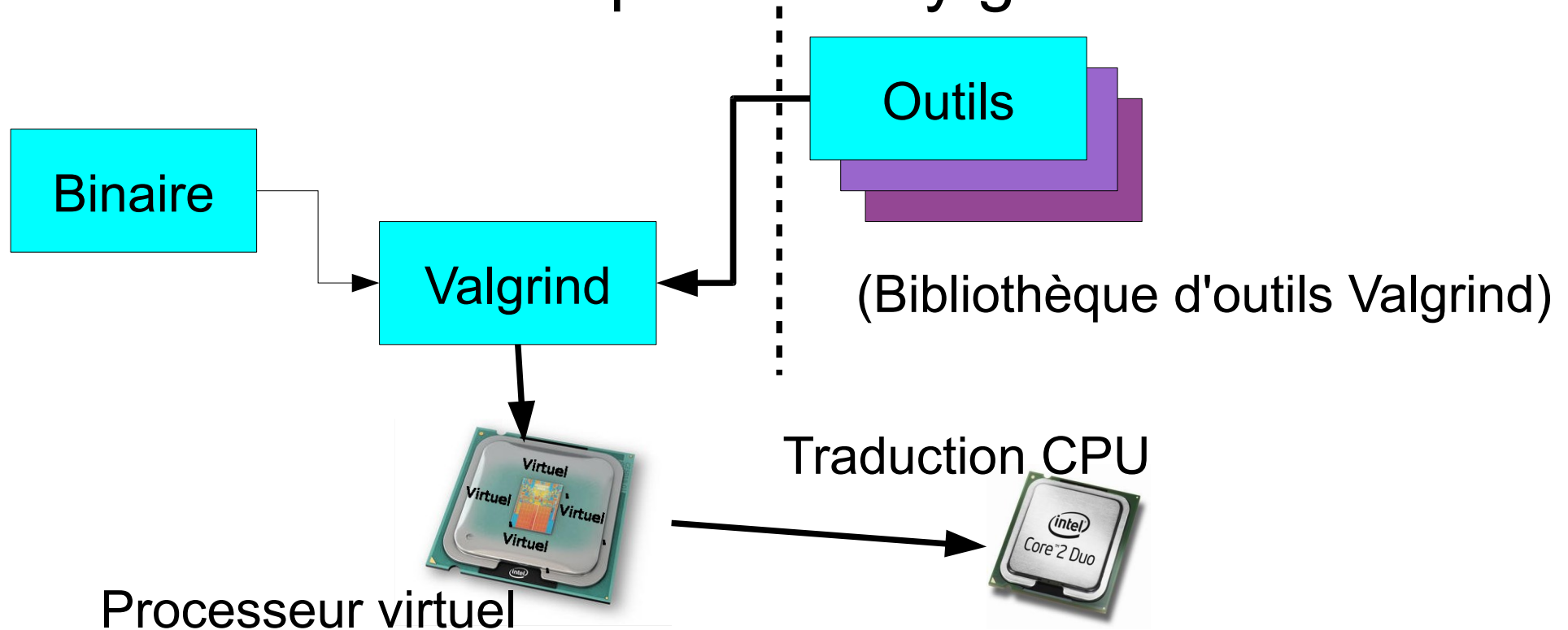


Émulateur

- ▶ Un débogueur pourra intercepter le signal `segfault`. Cela pourra alors nous indiquer la ligne de code source correspondante
- ▶ Les recherches de la cause sont de notre ressort
- ▶ Quels outils pour nous assister ?
 - Insertion automatique de code de vérification des bornes d'un tableau. Dans `gcc`, cette fonctionnalité est limitée à Java et Fortran. En C, coder manuellement des `assert`
 - Bibliothèques au dessus de `malloc` (*mais autres langages et routines d'allocation ?*)
 - Comment détecter les pointeurs mal initialisés ?

Émulateur

- ▶ Valgrind est une machine virtuelle capable d'exécuter du code binaire (sans recompilation)
- ▶ De nombreux outils peuvent s'y greffer



Émulateur

- ▶ Outil de base de valgrind : memcheck
 - Vérifie que les lectures/écritures mémoires se font à une adresse correspondant à une variable
 - Émet des avertissements sur des comportements gênants comme des valeurs ou des pointeurs non initialisées
 - Lors d'un accès illégal, Valgrind indique l'emplacement de l'adresse accédée par rapport au bloc de mémoire alloué (nombre d'octets en trop)
 - Détecte les doubles libérations de mémoires (dont le comportement est indéfini) et les fuites mémoires

Émulateur

- ▶ Pour cela, memcheck, dispose de sa propre version de malloc (utile car *in fine* toujours appelé au niveau système)
- ▶ Valgrind étant un émulateur, le programme sera ralenti de 5 à 100 fois (chiffres donnés par Valgrind)
- ▶ Attention aux tableaux statiques : ne détecte pas le débordement d'indice dès le 1er octet
- ▶ Utilisable avec MPI
 - Il est facile de se tromper avec les pointeurs dans MPI, surtout avec les opérations collectives

Émulateur : Exemple d'utilisation

```
)== Memcheck, a memory error detector
)== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
)== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright

)== Command: ./a.out
)==
)== Invalid write of size 4
)==   at 0x804844C: main (wa.c:10)
)==   Address 0x4198034 is 0 bytes after a block of size 12 alloc'd
)==   at 0x4024F20: malloc (vg_replace_malloc.c:236)
)==   by 0x8048428: main (wa.c:6)
)==
```

Remarquez le nom des routines : `vg_replace_malloc`

Émulateur : remarques pratiques

- ▶ Vous remarquerez qu'à l'installation de Valérien (sur vos machines virtuelles), un paquet `libc6-dbg` est installé
- ▶ Ce paquet contient uniquement les *symboles de débogage* de la lib6, permettant à Valgrind de savoir dans quelle fonction de la lib6 une erreur mémoire peut avoir lieu
- ▶ Les symboles de débogage sont présentés dans la suite

Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Débogueur

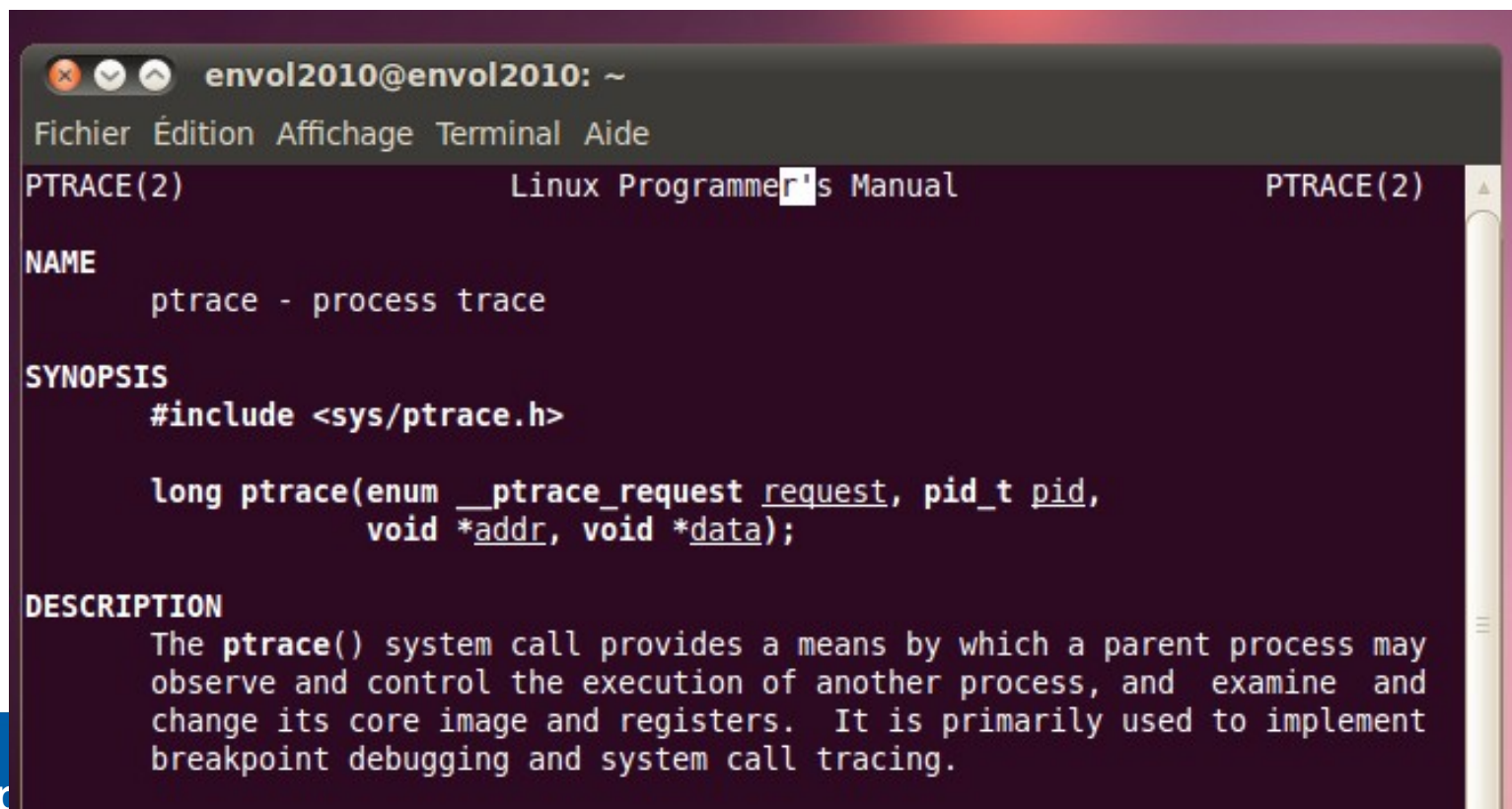
Les outils présentés précédemment sont spécialisés à certains types de problèmes. Ils permettent une recherche rapide dans les cas particuliers.

- ▶ Passons à l'outil généraliste, le couteau suisse
- ▶ Dans la suite :
 - Principe de fonctionnement des débogueurs (communs à tous les produits)
 - Panorama des outils
 - Étude d'un débogueur en particulier

Débogueur

Écrire un débogueur, c'est facile !

Pour le contrôle de l'exécution des programmes (pas à pas), les débogueurs utilisent l'appel système *ptrace*



```
envol2010@envol2010: ~
Fichier Édition Affichage Terminal Aide
PTRACE(2)          Linux Programmer's Manual          PTRACE(2)

NAME
    ptrace - process trace

SYNOPSIS
    #include <sys/ptrace.h>

    long ptrace(enum __ptrace_request request, pid_t pid,
                void *addr, void *data);

DESCRIPTION
    The ptrace() system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.
```

Débogueur

Le programme à analyser est en général le processus fils du débogueur. Il recevra l'une des commandes suivantes (dépendantes du système) :

- ▶ Continuer jusqu'à la prochaine instruction du compteur de programme (singlestep)
- ▶ Continuer (jusqu'à la prochaine action du débogueur)
- ▶ Lire / Écrire une valeur (octet, mot) en mémoire dans les espaces d'instructions ou de données du processus

Débogueur

Un programme vu par la machine, cela ressemble à :

```
popl    %ebx
movl    12(%ebp), %eax
movl    (%eax), %eax
movl    %eax, 4(%esp)
leal    LC0-"L000000001$pb"(%ebx), %eax
```

Apport du débogueur ⇒ Faire le lien avec le code source

Débogueur

Des données en mémoire, cela ressemble à (*ici la suite de 0 et de 1 de votre choix*) :

(Espace d'expression libre)

Apport du débogueur ⇒ Faire le lien avec les variables et structures de données définies par le programmeur

Débogueur

- ▶ Des indications supplémentaires sont indispensables au débogueur pour afficher données et instructions.
- ▶ Ces *symboles de débogage* sont ajoutés à la *compilation* (option `-g`)
- ▶ Il existe différents formats de symboles : stabs, COFF, XCOFF, DWARF 2 (<http://dwarfstd.org/>)
- ▶ Ils contiennent par exemple :
 - Des descripteurs d'emplacement, indiquant le type et la taille des variables
 - Le numéro de ligne dans le code source, le nom du fichier source (indispensable pour déboguer), ...

Couplage Débogueur / Compilateur

- ▶ Le débogueur lit dans le programme les informations placées par le compilateur
- ▶ Certains compilateurs et débogueurs ne sont pas interopérables. Pour cela, les compilateurs commerciaux sont livrés avec leurs débogueurs
- ▶ Le support de certains langages dépend du couplage. Par exemple, l'accès aux structures de données peut être incomplet

Débogueur – Fonctionnement de base

Le premier usage que nous faisons d'un débogueur est de savoir où en est le programme !

- ▶ Nous utilisons souvent les débogueurs pour vérifier qu'un programme avance (en particulier communications)
- ▶ Les débogueurs affichent la liste des appels de fonctions empilés à l'instant courant, ainsi que leurs paramètres (stack frame). On peut naviguer dans cette liste.
- ▶ Si lors de plusieurs interrogations successives, un programme est toujours dans la même fonction, on peut soupçonner un blocage

Débogueur – Fonctionnement de base

▶ Exemple d'utilisation :

```
gdb -pid 5091
#0  0x00000000000008193a0 in
m_bas_mp_calcbas1dotopbas0sq_ ()
...
#6  0x00000000000004a2dac in MAIN__ ()
#7  0x00000000000004a2b1c in main ()
```

▶ Fonction qui peut bloquer : select, listen, ...

Débogueur – Fonctionnement de base

Mise en place de breakpoints : *halte-là !*

- ▶ Lorsque le programme atteint l'emplacement indiqué, il s'arrête et rend la main au débogueur
- ▶ Le breakpoint peut être placé devant n'importe quelle instruction, source ou assembleur
- ▶ L'arrêt a lieu **juste avant** :
 - la suite d'instructions assembleur correspondant à la ligne de code source
 - L'instruction assembleur si on travaille à ce niveau
- ▶ Syntaxe sous gdb : `break ligne / nom_fonction`

Débogueur – Fonctionnement de base

Utilité des breakpoints

- ▶ Réaliser un instantané des données du programme
Ex : juste avant une ligne dont on a déterminé qu'elle est la cause du plantage

Limitations

- ▶ L'arrêt est systématique :
 - Les débogueurs permettent d'imposer un arrêt tous les n franchissements (utiles pour les boucles)
 - Il faut savoir où chercher : avant un appel de fonction, une boucle...

Débogueur – Fonctionnement de base

Placer des breakpoints conditionnels

- ▶ La condition peut être :
 - Liée aux données : une valeur de variable
 - Liée au contrôle : le nombre de franchissements du breakpoint
- ▶ Permet de limiter le nombre d'alertes remontées par le débogueur
- ▶ Syntaxe sous gdb : `break 7 if i==10`

Débogueur – Fonctionnement de base

Placer des watchpoints

- ▶ À chaque modification d'une variable (adresse mémoire), le débogueur reprend le contrôle et indique **la ligne source** correspondante
- ▶ Utile pour déterminer la portion de code problématique avant d'autres investigations
- ▶ Syntaxe sous gdb : `watch foo`

Débogueur – Fonctionnement de base

Avancer dans le code (step, next)

```
        procedure do_something (a,n)
            real,dimension(n) :: a
Step  → a(n-5)=2*a(n-4)
        end procedure do_something

        program complex_computation
Next → n=3000
Next → n2=3002
        call read_array(a,n)
Next → call do_something(a,n)
        end program complex_computation
```

Débogueur – Fonctionnement de base

Visualiser des données

- ▶ À partir des symboles de débogage :
 - le débogueur associe adresse mémoire et nom/type de variable (et dont la portée s'étend sur la zone de programme examiné)
 - les affiche à la demande
 - Il faut parfois l'aider (cas des tableaux)
- ▶ Cette exploration *interactive* est utile si on ne sait pas exactement quelle variable est en tort (évite de tout afficher avec un print dans le code)

Débogueur – Liste d'outils

▶ gdb : <http://www.gnu.org/software/gdb/>

- C, C++, Fortran, python
- Dernière version Septembre 2010

▶ dbx :

<http://www.oracle.com/technetwork/server-storage/solarisstudio/dbx>

- C, C++, Fortran, Pascal
- Dernière version Septembre 2010

▶ Produits associés à des compilateurs commerciaux

- Intel, PGI, Pathscale

▶ Produits spécialisés : Totalview, DDT

Débogueur – Interfaces graphiques

- ▶ La manipulation des débogueurs en ligne de commande est parfois fastidieuse ⇒ il existe naturellement des interfaces graphiques
 - Visant à intégrer le débogueur à l'environnement de développement (cf. Eclipse)
 - Permettant de rehausser le confort d'utilisation du débogueur sous-jacent
- ▶ Parmi les interfaces graphiques, certaines sont généralistes, d'autres dédiées à un débogueur

Débogueur – Interfaces graphiques

- ▶ Liées à Gdb
 - Eclipse CDT (C/C++ Development Toolkit)
 - kdbg (sous KDE)
 - Intégration de gdb à Emacs
- ▶ Liées à des outils commerciaux
 - Totalview, DDT
- ▶ Multi-débogueur
 - ddd (gdb, dbx)

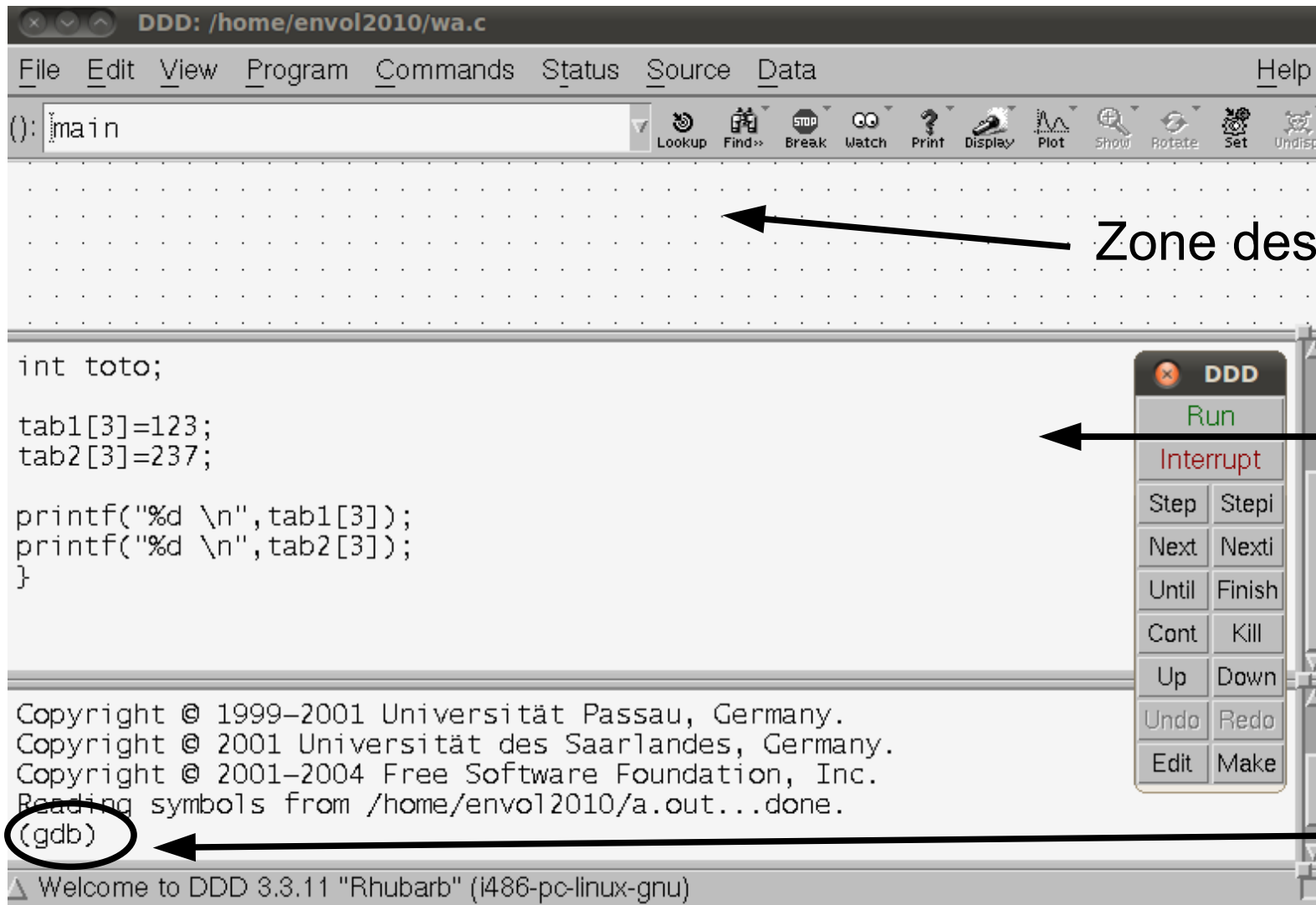
Débogueur – Interfaces graphiques

- ▶ Les interfaces graphiques facilitent en particulier l'affichage des données complexes (tableaux, structures, instances de classes)
- ▶ Elles permettent d'explorer aisément les données
- ▶ Certaines disposent de fonction de visualisation (Totalview, DDT, ddd)
- ▶ Les interfaces graphiques peuvent faciliter le débogage à distance

Débogueur – data display debugger

- ▶ Écrit en C++ / Motif. Version 3.3.11 sur la vm, la dernière est la 3.3.12
- ▶ Principaux atouts :
 - Interactivité
 - Beaucoup moins rébarbatif que gdb !
 - Possibilité de visualisation des données
 - Manipulation des breakpoints
- ▶ Limitations
 - Nécessite de connaître la syntaxe du débogueur sous-jacent pour les opérations avancées
 - Réactivité du support sur la mailing-list ?

Débogueur – ddd

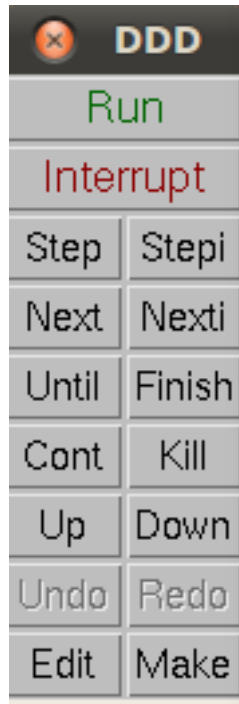


Zone des variables

code source
programme

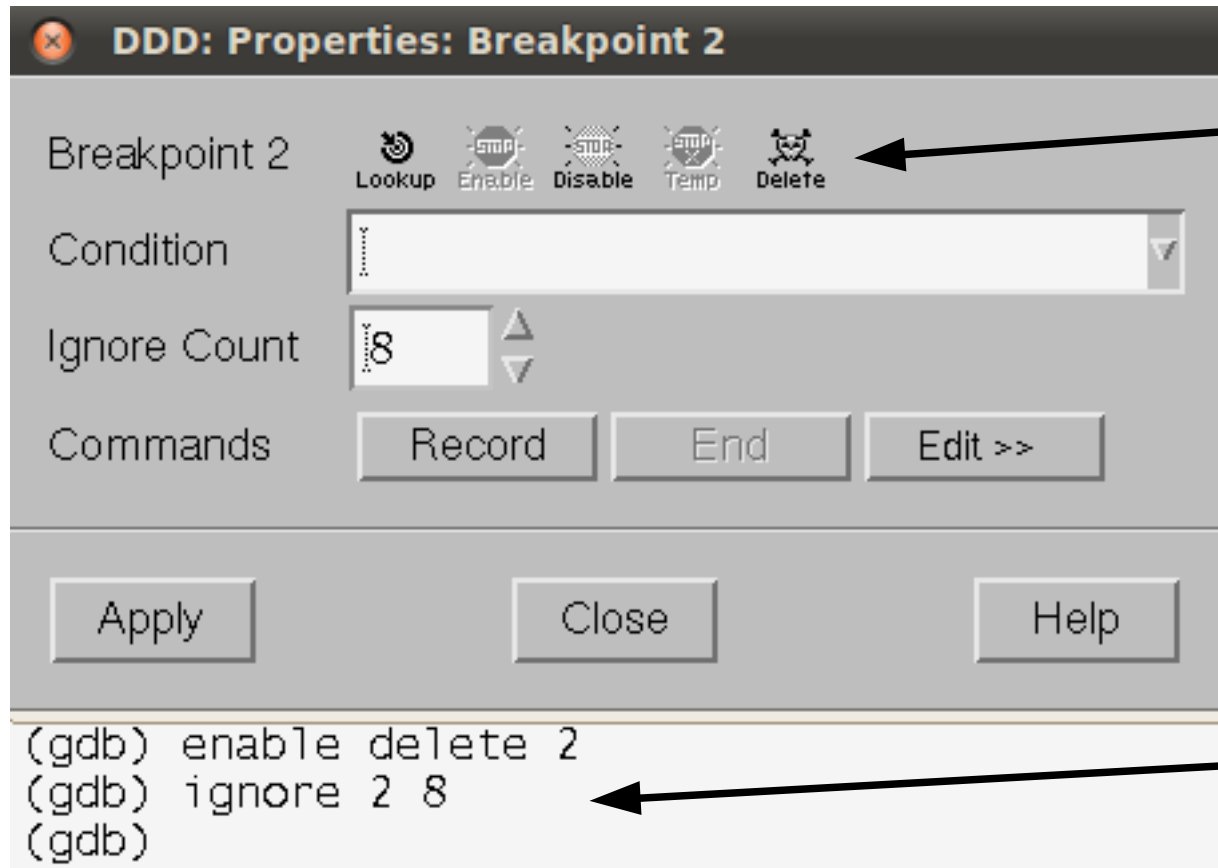
Interaction gdb

Débogueur – ddd - base



- ▶ Interrupt : arrêter ici
- ▶ Step/Stepi : Exécuter une ligne source/assembleur
- ▶ Next/Nexti : Idem mais laisser se termine les appels de fonctions
- ▶ Until : Exécuter jusqu'à après la ligne courante
- ▶ Finish : terminer la fonction courante
- ▶ Up / Down : parcourir la pile d'appels

Débogueur – ddd - facilités



Caractéristiques des Don breakpoints

Traductions en commandes gdb

Débogueur – ddd – historique des valeurs



Ddd conserve un historique des valeurs aux points d'arrêt précédents

- ▶ Undo : se rend au point d'arrêt précédent et affiche les valeurs correspondantes
- ▶ Redo : revient au point d'arrêt que l'on vient de quitter


Débogueur – ddd – sauts dans le code

À un breakpoint, ddd permet de déplacer graphiquement le compteur d'instructions

- ▶ Interfaçage graphique à gdb
- ▶ Intérêt : rejouer une portion de code avec des paramètres différents

```
→meteo.temp=27.2;
meteo.pression=1024;

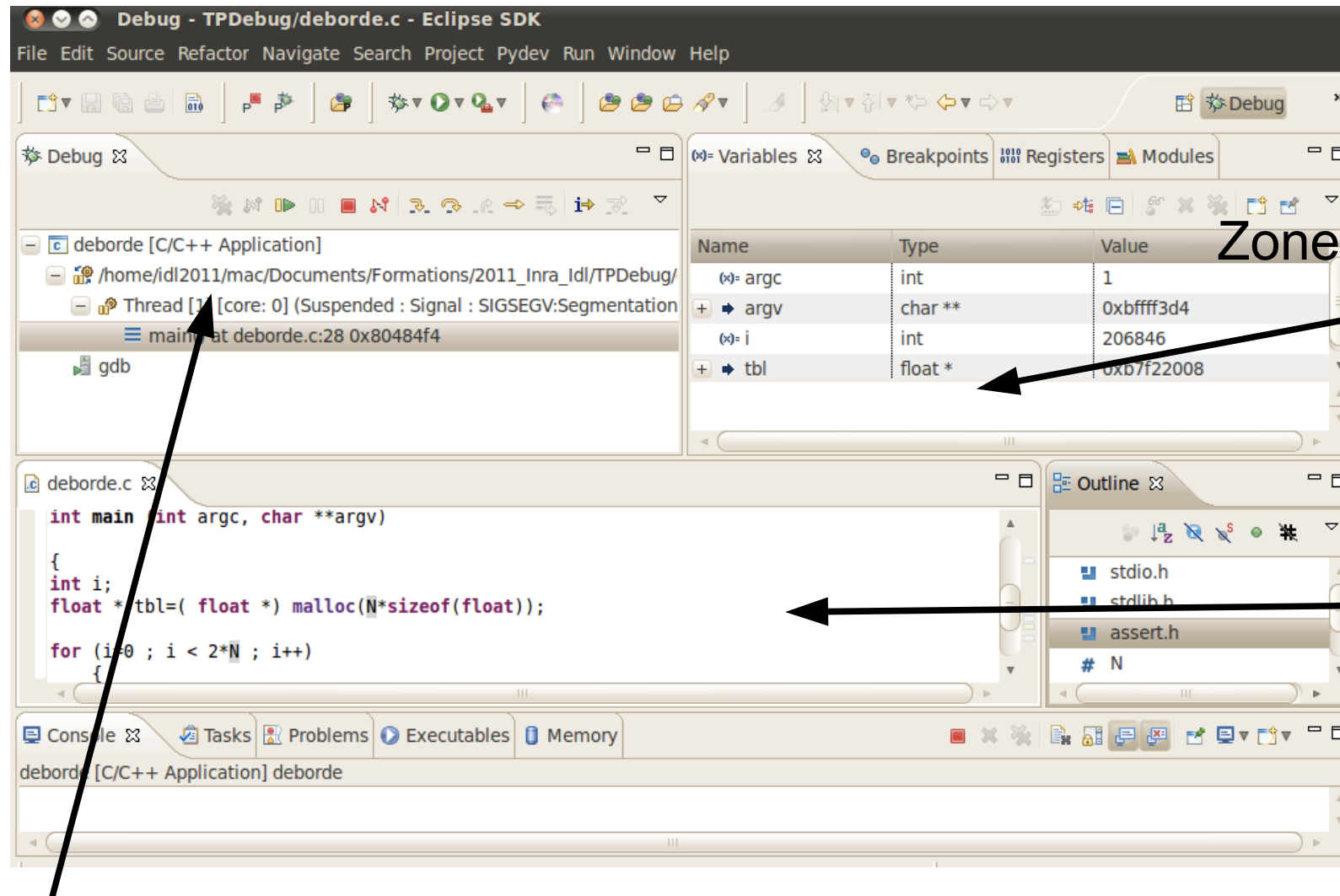
/* Utiliser les 5 prochains
*/
meteo.relevés=allouer_mat
```



```
(gdb) jump tabl.c:94
```

↑
Commande gdb correspondante

Débogueur - CDT



Zone des variables

code source
programme

Zone des programmes

Débogueur – CDT - base

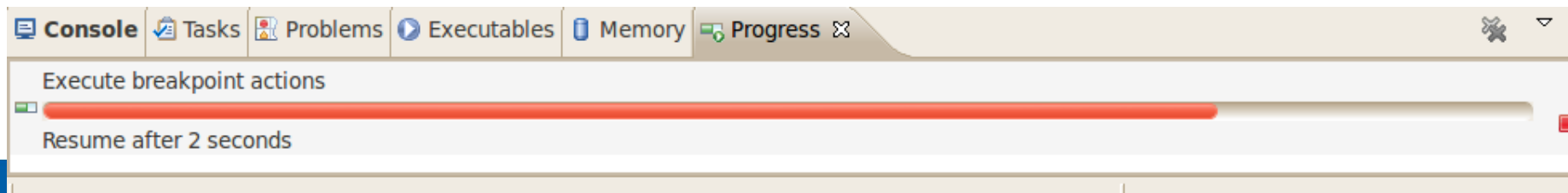
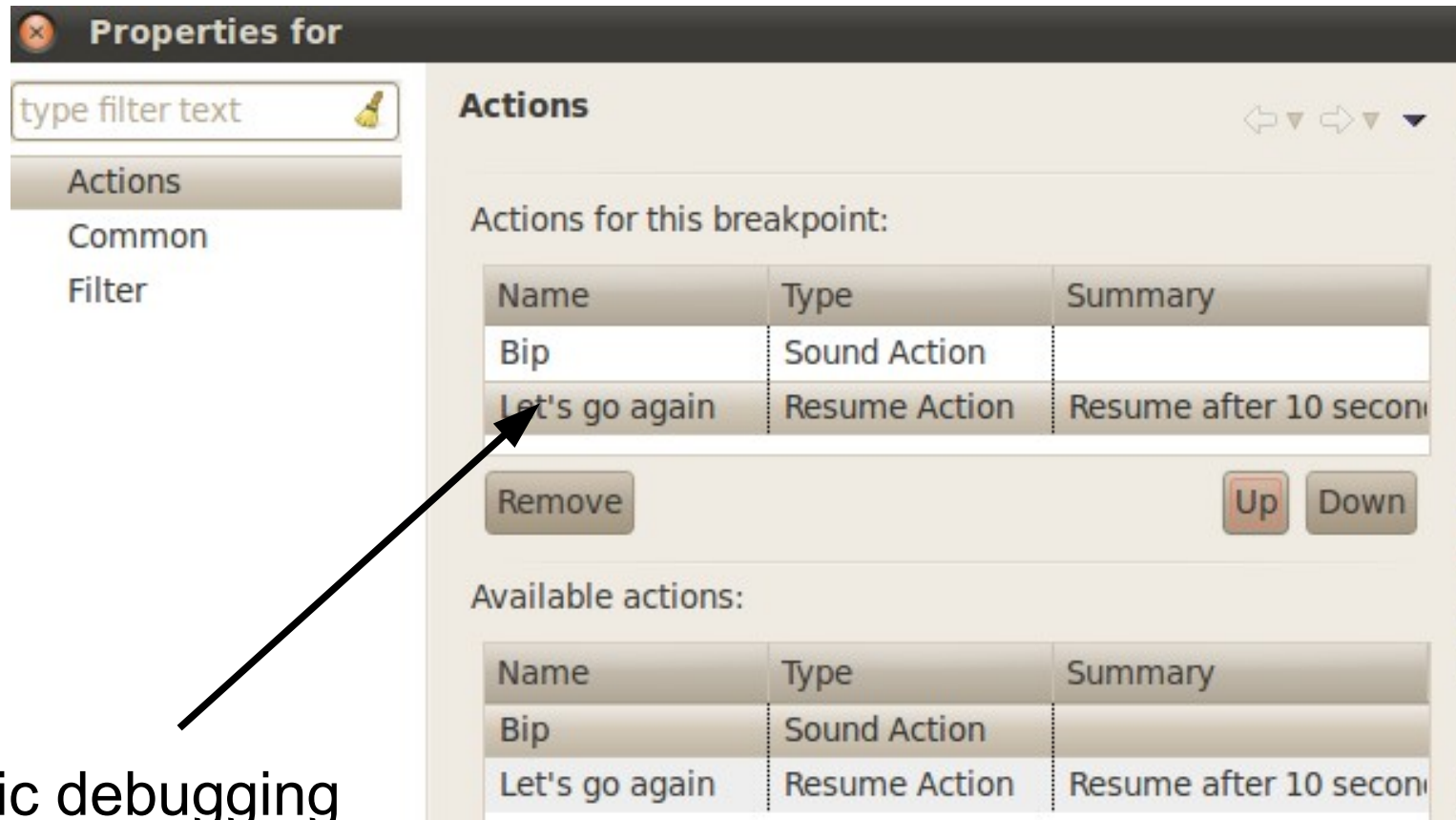


Reprendre, pause

Stepping

Terminer, détacher

Débogueur – CDT – breakpoint



Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Débogueur – Limitations observées

► Support du Fortran dans gdb

- Continue à évoluer dans gdb (obtention des descripteurs de tableaux)
- Retard par rapport aux produits commerciaux (comparaison avec Totalview dans la préparation de ce cours)

```
Breakpoint 1, tab1 () at tab1.f90:22  
(gdb) ptype meteo%releves  
type = real(kind=4) (0:-1,0:-1)  
(gdb)
```

Avec gfortran + gdb



Avec f90 (sun) + dbx



```
(process id 5306)  
t@3073345952 (1@5306) stopped in MAIN at line 24 in file "tab1.f90"  
(dbx) whatis -r meteo%releves  
REAL*4 , releves(1:90,1:90) ! allocatable  
(dbx)
```

```
△ REAL*4 , releves(1:90,1:90) ! allocatable
```

Débogueur – Limitations observées

- ▶ Traçage des données 2D buggé
 - Observation : au delà d'une certaine dimensions, ddd remplace un tracé $z=f(x,y)$ par n tracés $y=f(x)$
 - Reste à déboguer le débogueur...
- ▶ Suivi des bugs. Activité de la liste ddd ?
 - Message envoyé le 8/9/10, pas de réponse
 - Dernière réponse sur la liste le 19/4/10

Débogueur – Fortran

- ▶ Photran + Eclipse. Version 0.6 : Juin 2010
 - Non testé dans le cadre de ce cours
 - Au dessus de gdb
- ▶ Solution la plus satisfaisante : ddd + dbx
 - Bémols sur la visualisation
 - Non testé : Sunstudio, IDE intégrant les compilateurs et outils de sun

Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Conclusion : un petit comparatif

Nom	Débogueur	Plot données	Libre	Plus	OS
<code>gdb</code>	lui-même	tableaux	oui	client/serveur	Tous
<code>idb</code>	<code>idbc</code>		non	gui	Linux, Mac
<code>ddd</code>	<code>gdb</code> , <code>dbx</code> , <code>pydb</code> , <code>xdb</code> , <code>wdb</code>	courbes 1D, (un peu) 2D	oui	gui, représentation données	Linux, Mac
<code>cdt</code>	<code>gnu</code>	non	oui	gui, intégration eclipse	Tous
<code>ddt</code>	lui-même	oui	non	gui, parallèle, GPU	Linux
<code>totalview</code>	<code>tv9cli</code>	tableaux, tranches	non	gui, parallèle, GPU	Linux, Mac
<code>lldb</code>	lui-même	?	oui	code llvm, objective-c	Linux, Mac

Conclusion

- ▶ Vous voilà armés pour le débogage
- ▶ Procédez par complexité croissante d'outils
- ▶ Commencez tôt avec les débogueurs
 - Utiles pour exploration de données également
- ▶ Le débogage d'applications parallèles est tout un sport (cf École CNRS, Choix et Exploitation d'un Calculateur, 2009, <http://calcul.math.cnrs.fr/spip.php?rubrique76>)