Plan

I. Choisir un langage (Vincent)

II. Adopter une méthode de développement

III. Coder

IV. Partager, travailler en équipe

V. Profiling

VI. Débugger

V. Profiling

I. Choisir un langage (Vincent)

II. Adopter une méthode de développement

III. Coder

IV. Partager, travailler en équipe

V. Profiling

V. Débugger

Partie basée sur le support de Violaine Louvet

Le profiling : Pourquoi?

Améliorer la vitesse d'exécution

Réduire l'empreinte mémoire du programme

Diminuer la consommation d'énergie (systèmes embarqués)

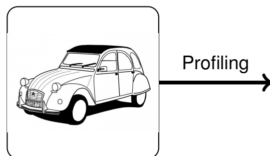
Consommer moins de ressources (réseau, I/O)

Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus

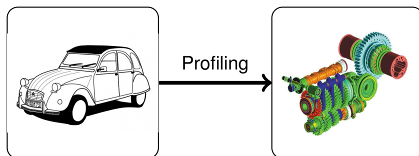
Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution



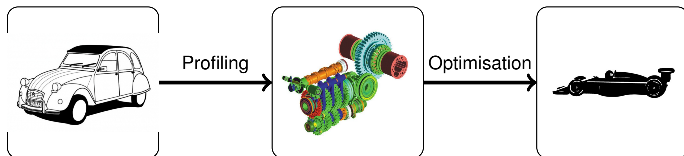
Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution
- Améliorer les parties les plus critiques



Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution
- Améliorer les parties les plus critiques
- Vérifier et valider le code au cours du processus d'optimisation



Objectifs de ce cours

Performances

- Déterminer les parties du code les plus coûteuses en temps.
- Déterminer les fonctions sur lesquelles faire porter l'effort d'optimisation.
- Savoir utiliser les outils de profiling.

Optimisations

- Réduire le temps de calcul.
- Réduire l'empreinte mémoire.
- Connaître les techniques d'optimisation permettant d'atteindre ces objectifs.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Définitions

Outils de profiling

Outils indispensables pour optimiser de manière pertinente un code : ils permettent d'identifier les parties du code (hot spots, points chaud) sur lesquelles il va falloir travailler :

- Temps passé dans chaque partie du programme.
- Nombre d'appels des fonctions.
- Interaction du programme avec l'environnement : accès mémoire, accès concurrents aux données ...

Définitions

Différentes techniques

- Echantillonnage : l'exécution est échantillonnée régulièrement pour savoir quelles fonctions sont appelées. Pas d'intrusion mais résultat dépendant notamment de la fréquence d'échantillonnage. Plus la durée d'exécution est longue, plus les résultats sont précis.
- Instrumentation : le compilateur ajoute à chaque appel de fonction une fonction d'instrumentation qui va mesurer le temps d'appel, le nombre d'instructions exécutées ...
- Emulation : exécute le programme sur un processeur virtuel. Tout peut ainsi être mesuré de manière exacte mais cette méthode est très lente.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Mesure simple du temps

Evaluer de façon non intrusive le temps d'exécution du programme : utilisation de la commande *time*.

```
$ time ./prog

real 0m2.671s
user 0m2.275s
sys 0m0.327s
```

- real : temps réel écoulé lors de l'exécution (temps passé, y compris lorsque le processus est en attente). Issu de l'appel système *gettimeofday*.
- user : temps d'utilisation CPU en mode utilisateur (les autres processus et les temps d'attente du processus ne sont pas comptabilisés). Issu des appels système *times* et *wait*.
- sys : temps d'utilisation CPU en mode noyau (appels système par opposition aux appels du code et des bilbiothèques utilisées). Issu des appels système *times* et *wait*.

✓ user+sys donne le temps CPU utilisé par le process, sur tous les CPU.

✓ Cas de plusieurs threads : user+sys $\geq$ real

Mesure simple du temps

- Evaluer plus finement de façon intrusive : utilisation de la routine système *gettimeofday* qui permet de connaître le temps passé dans une partie du programme.

```
#include <sys/time.h>

struct timeval start,end;
long int useconds;

gettimeofday (&start, (struct timezone*)0);
// Code a mesurer
gettimeofday (&end, (struct timezone*)0);
useconds = (end.tv_sec - start.tv_sec)*1000000
    + end.tv_usec - start.tv_usec;
```

```
$ time ./prog
Temps total : 2825498

real 0m2.671s
user 0m2.275s
sys 0m0.327s
```

Profiling

Techniques précédentes limitées à des évaluations ponctuelles sur le temps d'exécution
Nécessité d'automatiser et de pouvoir avoir d'autres données d'analyse

- Intrumentation statique : effectuée au plus tard à la compilation.
- Instrumentation dynamique : effectuée lors de l'exécution.

Données recueillies par le profiler

- Temps d'exécution de chaque fonction (profil plat).
- Graphe d'appels du programme.
- Temps inclusive : fonction appelante + fonction appelée.
- Temps exclusive : fonction appelante uniquement.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

gprof

gprof utilise à la fois des techniques d'échantillonnage et d'instrumentation

Instructions de profiling ajoutées à la compilation

Option de compilation : *-p -g*

- Chaque fonction est modifiée à la compilation pour mettre à jour les structures de données stockant la fonction appelante et le nombre d'appels.
- Le temps d'exécution est évalué par échantillonnage.
- Pour des fonctions non instrumentées (appel de bibliothèques par exemple), seule l'information du temps passé dans la fonction est disponible.

En pratique

- Compilation : *g++ -p -g prog.cpp -o prog*
- Exécution normale du programme : *./prog*
- Génération d'un fichier de données *gmon.out* dans le répertoire courant.
- Exécution de *gprof* : *gprof prog gmon.out*

Description des sorties de gprof : profil plat

% time : pourcentage du temps d'exécution total passé à exécuter cette fonction.

cumulative seconds : temps total cumulé que le processeur a passé à exécuter cette fonction, ajouté au temps passé à exécuter les fonctions précédentes dans le tableau.

self seconds : nombre de secondes passées à exécuter cette seule fonction.

calls : nombre d'appels de la fonction.

self ms/calls : nombre de millisecondes passées dans la fonction par appel.

total ms/calls : nombre de millisecondes passées dans cette fontion et ses enfants.

name : nom de la fonction

Description des sorties de gprof : graphe d'appels

- Différentes parties séparées par des tirets.
- Chaque partie débute par la ligne primaire :
  - Elle comprend en début de ligne un nombre entre crochets.
  - Elle se termine par le nom de la fonction concernée.
  - Les lignes précédentes décrivent les fonctions appelantes.
  - Les lignes suivantes décrivent les fonctions appelées : fonctions enfants.
  - Les entrées sont classée par temps passé dans la fonction et ses enfants.

index : Index référençant la fonction.

% time : pourcentage du temps passé dans la fonction, incluant les enfants.

self : temps total passé dans la fonction.

Children : temps total passé dans les appels aux enfants.

called : nombre d'appels de la fonction. Si appels récursif, la sortie est de la forme n+m, n désigne le nombre d'appels non récursifs et m le nombre d'appels récursifs.

name : nom de la fonction avec son index.

Profiling plus détaillé : Oprofile

Profiling ligne à ligne. Ne nécessite qu'une compilation avec option de debuggage.

En pratique

- Compilation : *g++ -g prog.cpp -o prog*
- Exécution du programme sous profiling : *operf ./prog*
- Premier niveau de profiling : *opreport –callgraph*
- Deuxième niveau de profiling : *opannotate –source*
- 1ère colonne : nbre d'échantillons de la ligne, 2ème colonne : % relatif par rapport à l'échantillonnage total.

```
  299   0.4644 :       for (int j = 0 ; j < m ; j++) {
  280   0.4349 : l = j + i * m;
 1596   2.4788 : arr2[i][j]=arr1[l]*log(i+1);
               :       }
               :
               :     for (int j = 0; j < m ; j++)
 1025   1.5920 :       for (int i = 0 ; i < n ; i++) {
  275   0.4271 : l = j + i * m;
20452  31.7647 : arr1[l]=arr2[i][j]*sin((2*i)/(j+1));
               :       }
```

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Valgrind

Suite d'outils de profilage et de débogage mémoire :

- Memcheck : un détecteur de fuites mémoires
- Cachegrind : un simulateur de caches
- Callgrind : un profileur

Fonctionnement

Le code est exécuté dans une machine virtuelle émulant un processeur équipé de nombreux outils d'analyse donnant des informations poussées sur le comportement d'un programme, qui ne pourraient être obtenues à l'aide d'un processeur matériel.

Quelques précisions

- Valgrind dégrade énormément les performances
- Ne surtout pas lancer une analyse Valgrind sur un programme complet long à exécuter

Débogage mémoire

En pratique

- Compilation du code :

```
$ g++ -g prog.cpp -o prog
```

- On peut vérifier que l'exécution du code se déroule correctement :

```
$ ./prog
Fin du programme
```

- Exécution sous valgrind :

```
$ valgrind --tool=memcheck ./prog
```

Débogage mémoire : évaluer les fuites mémoires

```
...
==4264==
==4264== HEAP SUMMARY:
==4264==     in use at exit: 0 bytes in 0 blocks
==4264==   total heap usage: 25 allocs, 25 frees, 24,020,009 bytes allocated
==4264==
==4264== All heap blocks were freed -- no leaks are possible
```

Fuites mémoire

- Si le programme est complexe, on peut lancer *vagrind* avec l'option *–leak-check=full* pour avoir plus d'informations sur la provenance des erreurs.
- A noter : le compilateur se charge de nettoyer correctement la mémoire, mais si ce n'est pas le cas, notre programme provoque des fuites mémoire

Utilisation du profiler

```
$ valgrind --tool=callgrind --dump-instr=yes ./prog
```

- *–dump-instr=yes* : permet d'enregistrer les instructions exécutées (facilite la comparaison avec le source).

- Génération d'un fichier *callgrind.out.numéro_pid*.
- Utilisation de l'outil KCacheGrind pour l'analyser.

Analyse de l'utilisation des caches

```
$ valgrind --tool=cachegrind ./prog
```

- Il est possible de spécifier la configuration des caches I1/D1/L2.

```
==29727==
==29727== I   refs:      3,358,368
==29727== I1  misses:        1,288
==29727== L2i misses:        1,271
==29727== I1  miss rate:      0.03%
==29727== L2i miss rate:      0.03%
==29727==
==29727== D   refs:      1,475,337  (1,149,931 rd   + 325,406 wr)
==29727== D1  misses:       16,516  (   12,225 rd   +   4,291 wr)
==29727== L2d misses:        9,531  (    5,755 rd   +   3,776 wr)
==29727== D1  miss rate:       1.1% (      1.0%     +     1.3%   )
==29727== L2d miss rate:       0.6% (      0.5%     +     1.1%   )
==29727==
==29727== L2 refs:          17,804  (   13,513 rd   +   4,291 wr)
==29727== L2 misses:        10,802  (    7,026 rd   +   3,776 wr)
==29727== L2 miss rate:        0.2% (      0.1%     +     1.1%   )
```

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

gcov

- Outil de couverture de code.
- Permet de savoir quelles lignes de code sont effectivements exécutées et combien de fois elles le sont.
- Complémentaire de gprof.
- Permet notamment de vérifier que le jeu de test utilisé pour valider un programme est suffisant.

En pratique

- Compilation : *g++ -fprofile-arcs -ftest-coverage prog.cpp -o prog*
- Exécution normale du programme : *./prog*
- Génération de fichiers de données **.gcda* et **.gcno* dans le répertoire courant pour chaque fichier compilé avec les options précédentes.
- Exécution de *gcov* : *gcov prog.cpp*. Génération de fichiers de données **.gcov* : indexation des lignes par le nombre d'appels par ligne et leur numéro.

Description des sorties de gcov

- Réécriture du programme source en préfixant chaque ligne par le nombre de fois où elle a été exécutée. Les résultats sont clairement dépendants des données : des exécutions avec des données différentes donneront des résultats différents.
- Pour un branchement, pourcentage représentant le nombre de fois où la branche a été prise, divisé par le nombre de fois où le test a été exécuté. Si la branche n'a jamais été exécutée, message « never executed ».
- Pour un appel de fonction : nombre de sortie en fin de fonction sur le nombre d'appels (en général 100% sauf en cas d'exit dans la fonction).
- Les traces d'exécutions (contenues dans les fichiers .gcda) s'accumulent : ceci permet de mener des campagnes de statistiques sur un grand nombre de données, afin d'obtenir des informations plus fiables.

VI. Débugger

I. Choisir un langage (Vincent)

II. Adopter une méthode de développement

III. Coder

IV. Partager, travailler en équipe

V. Profiling

VI. Débugger

VI. Débugger

Partie basée sur le support de Romaric David

Ça y est ça compile! Mais…

C'est quoi ce seg fault?

Le résultat n'est pas celui que j'attendais… il y a un bug!
Où ça?

Segmentation fault / Erreur de Segmentation?!?

Plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué.

Exemple :

- Une application écrit en dehors de la mémoire qui lui est allouée.
- Déréférencement d'un pointeur non initialisé : il pointait sur une zone mémoire quelconque qui n'est probablement pas allouée à l'application.

Le Débugger : Analyse de code à l'exécution

Exécution pas à pas

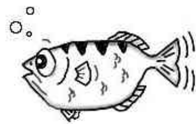
Contenu des variables disponible : permet de détecter la divergence avec le comportement escompté

Au moment d'un segmentation fault : consultation de la pile des derniers appels pour détecter la fonction appelante qui a provoqué l'erreur

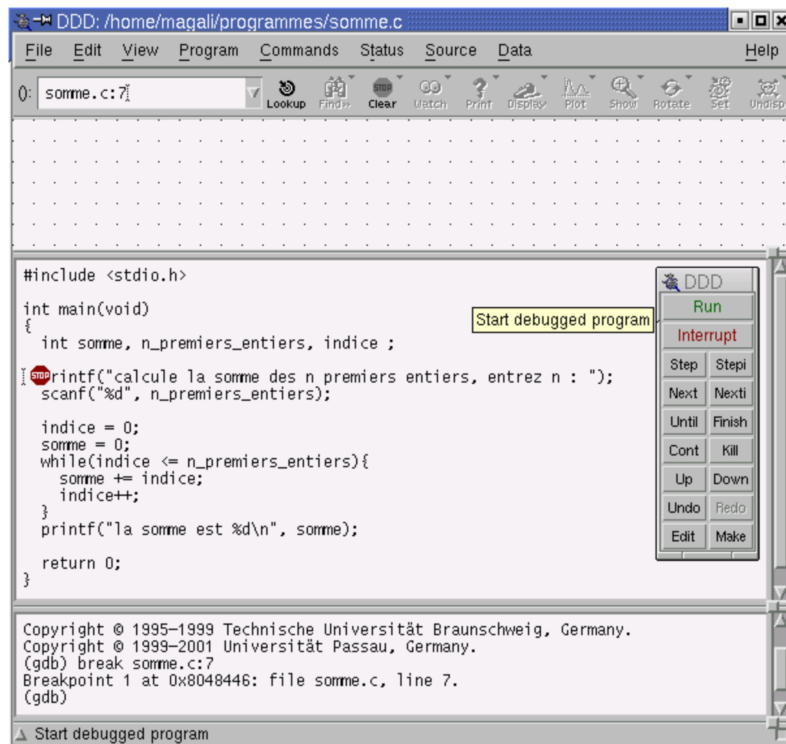
Analyse de code à l’exécution – Exemple 1 : gdb

GDB: The GNU Project Debugger

[bugs] [committee] [contributing] [current cvs] [documentation]

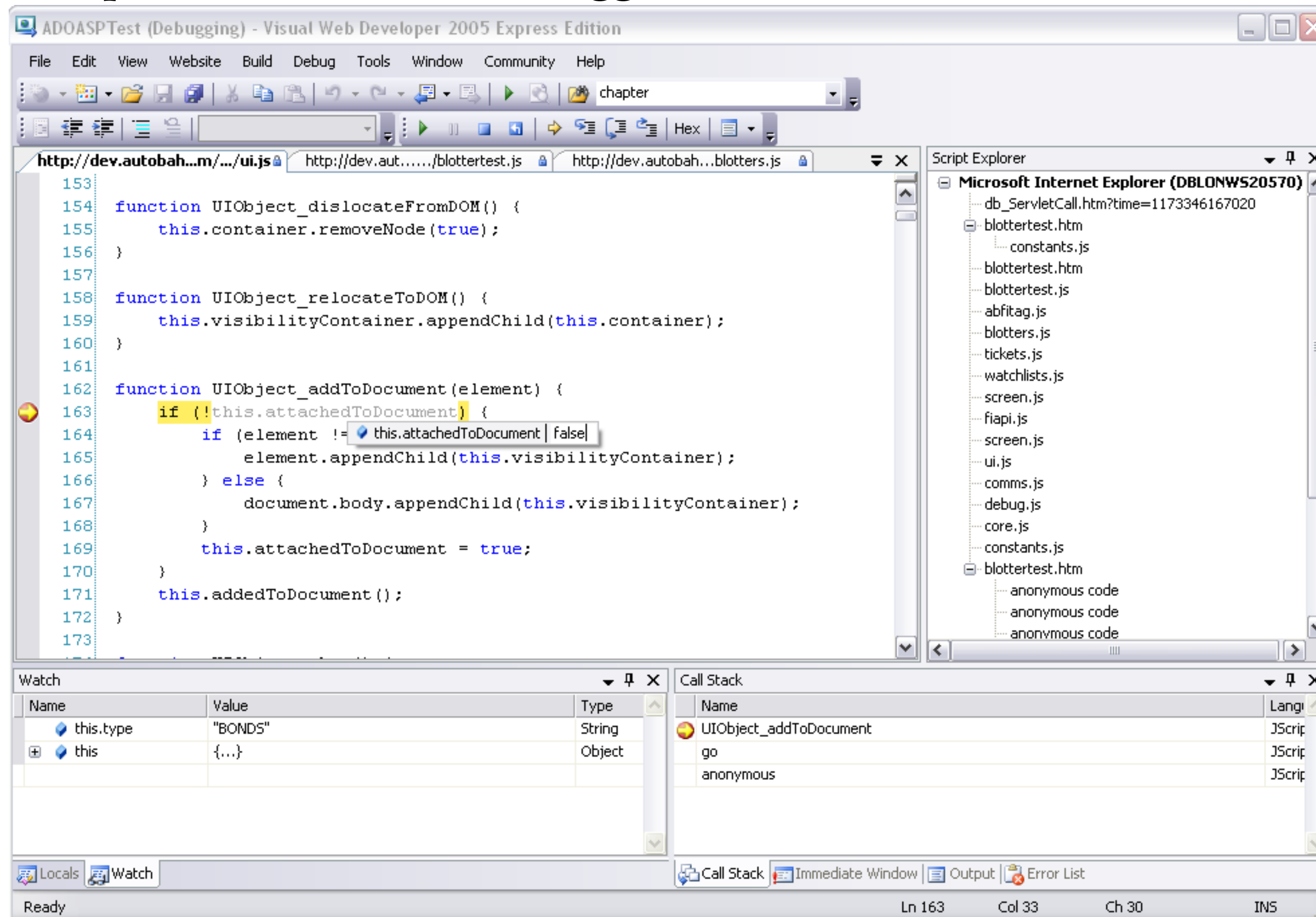


```
(gdb) break main
Breakpoint 1 at 0x8048446: file somme.c, line 7.
(gdb) break 11
Breakpoint 2 at 0x8048470: file somme.c, line 11.
(gdb) run
Starting program: /home/magali/programmes/somme

Breakpoint 1, main () at somme.c:7
7           printf("calcule la somme des n premiers entiers, entrez n : ");
(gdb) next
8           scanf("%d", n_premiers_entiers);
(gdb) next
calcule la somme des n premiers entiers, entrez n : 4
10          indice = 0;
(gdb) next

Breakpoint 2, main () at somme.c:11
11          somme = 0;
(gdb) next
12          while(indice <= n_premiers_entiers){

(gdb) watch indice==3
Hardware watchpoint 2: indice == 3
```



VI. Débugger

Exemple 2 : Visual Studio Debugger



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.

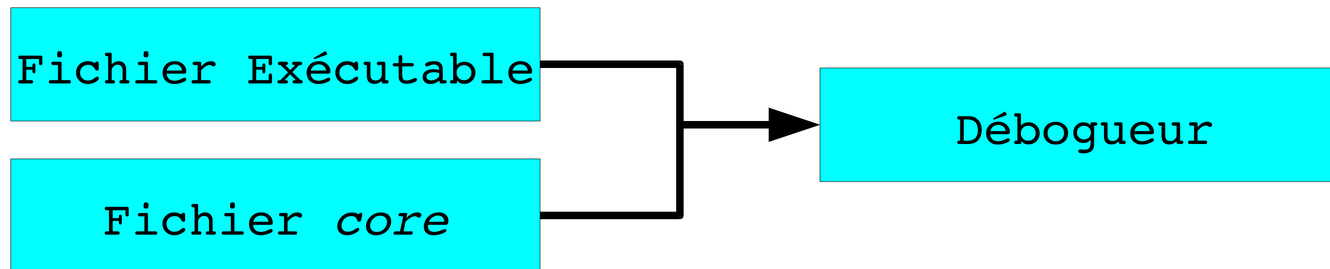
Un comportement incorrect peut être :

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

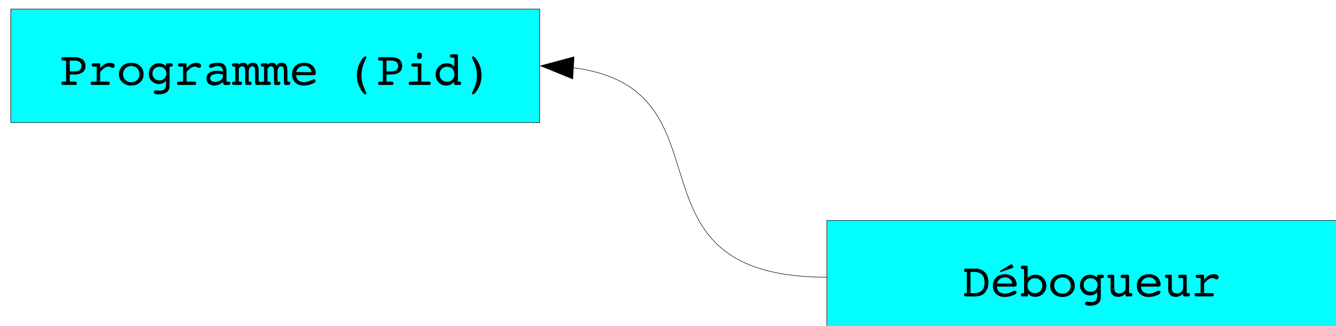
Débogage : définition

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)
```

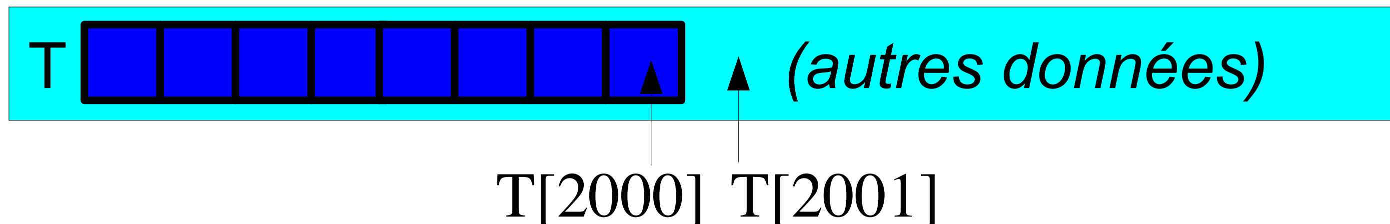
Émulateur

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.

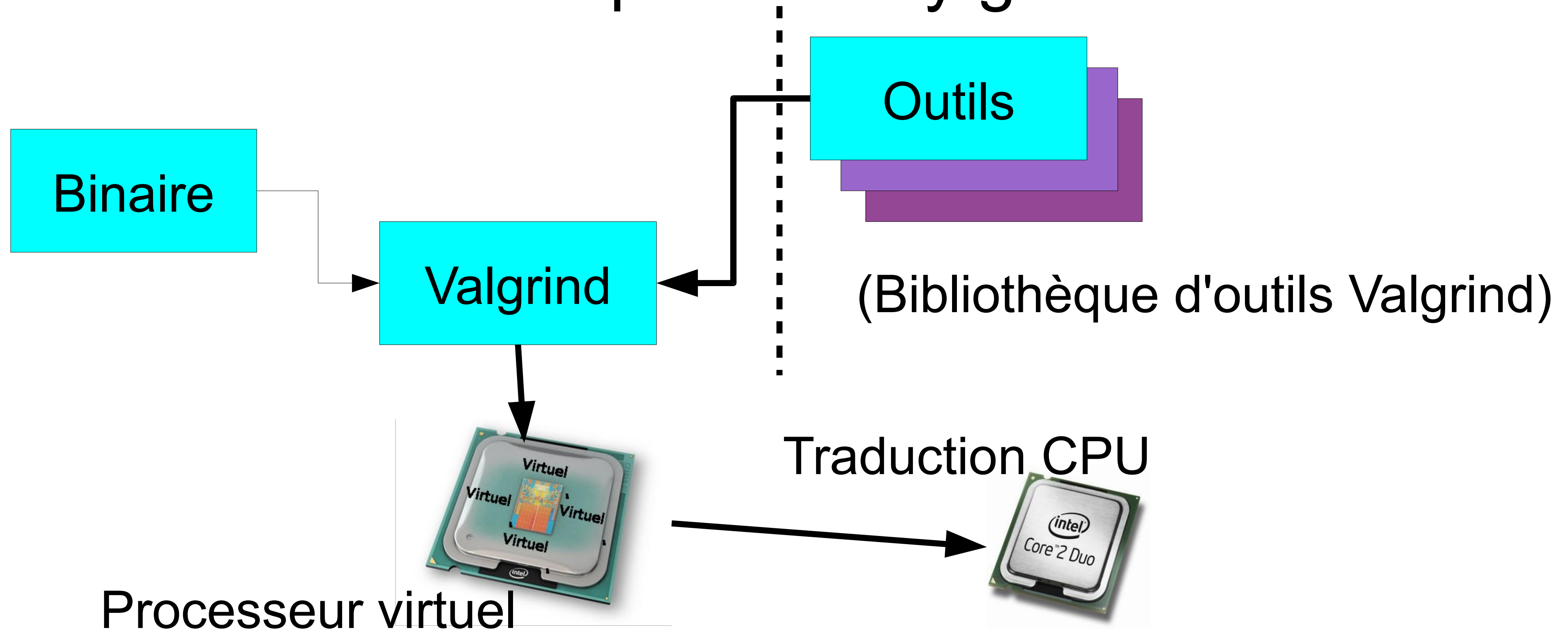


Émulateur

- 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

Plan

Introduction

Débogage, quelques définitions

Deux outils spécifiques

Étude d'un débogueur

Problèmes rencontrés

Conclusion

Débogueur

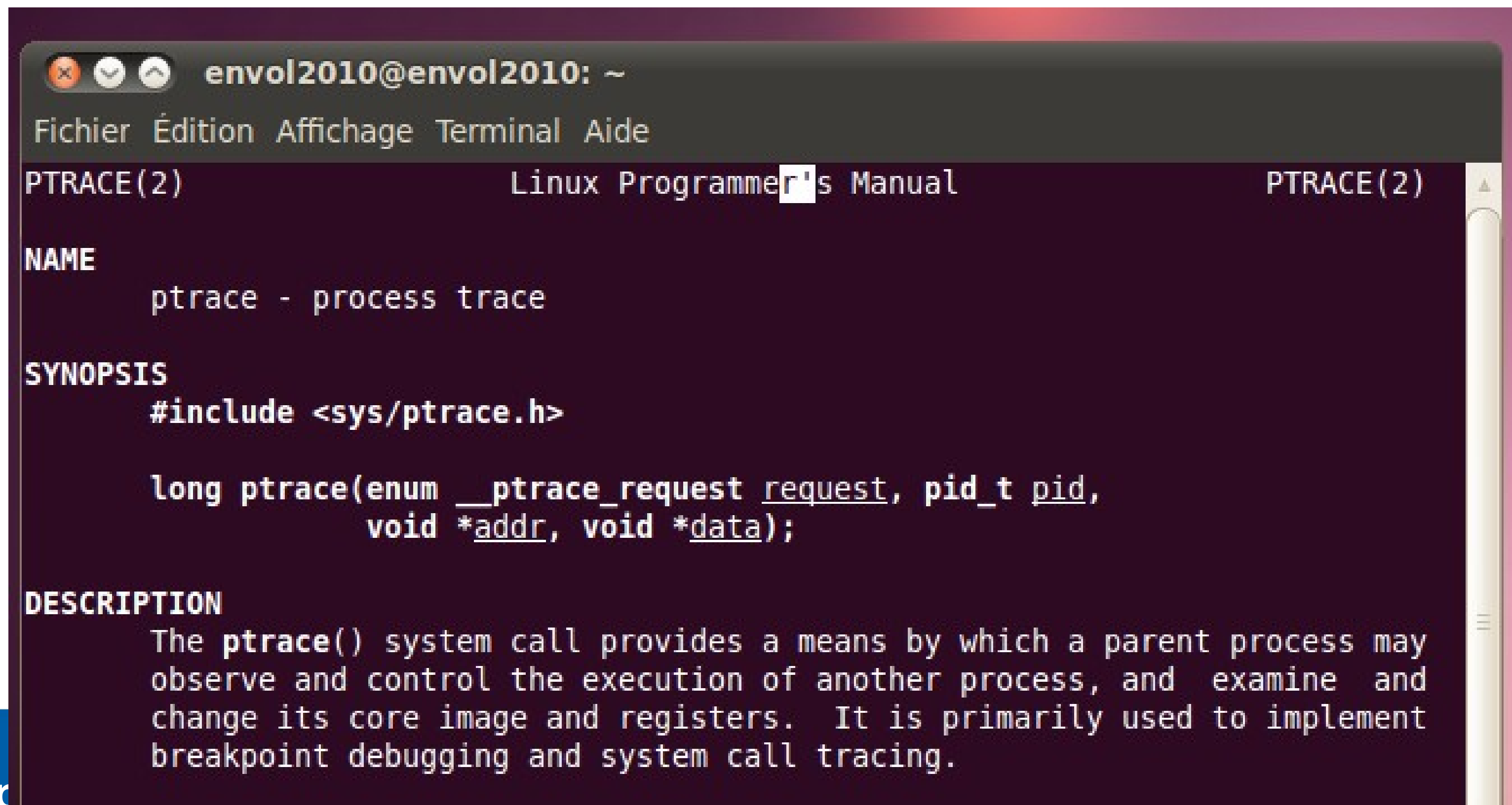
Les outils présentés précédemment sont spécialisés à certains types de problèmes. Ils permettent une recherche rapide dans les cas particuliers.

- 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*



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-"L00000001$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  0x00000000008193a0 in
m_bas_mp_calcbas1dotopbas0sq_ ()
...
#6  0x00000000004a2dac in MAIN__ ()
#7  0x00000000004a2b1c 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 donnés : 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
       ------>   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/d
  - 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++ Developpement Toolkit)
  - kdbg (sous KDE)
  - Intégration de gdb à Emacs
- Liées à des outils commerciaux
  - Totalview, DDT
- Multi-debogueur
  - ddd (gdb, dbx)

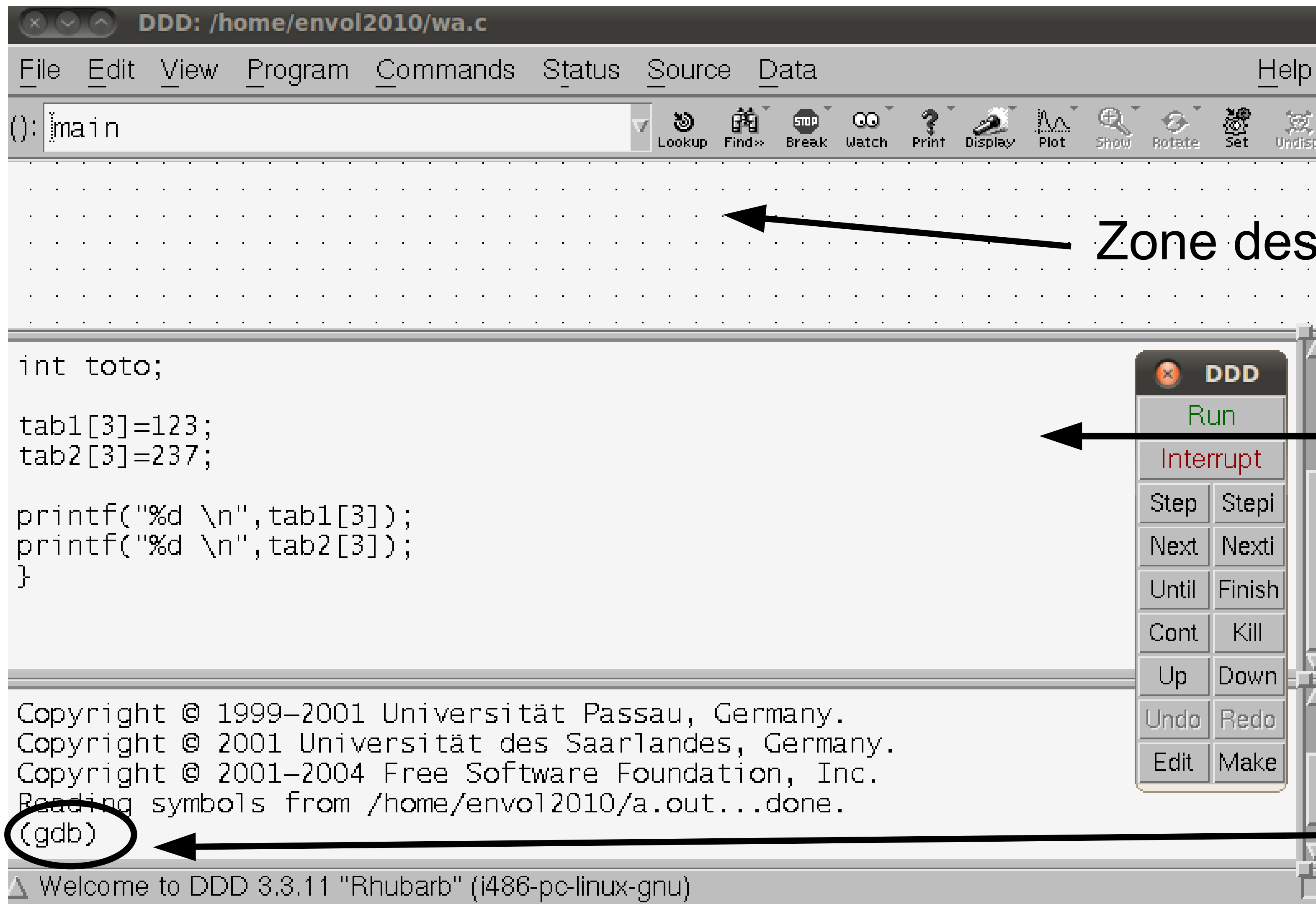
Débogueur – Interfaces graphiques

- 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

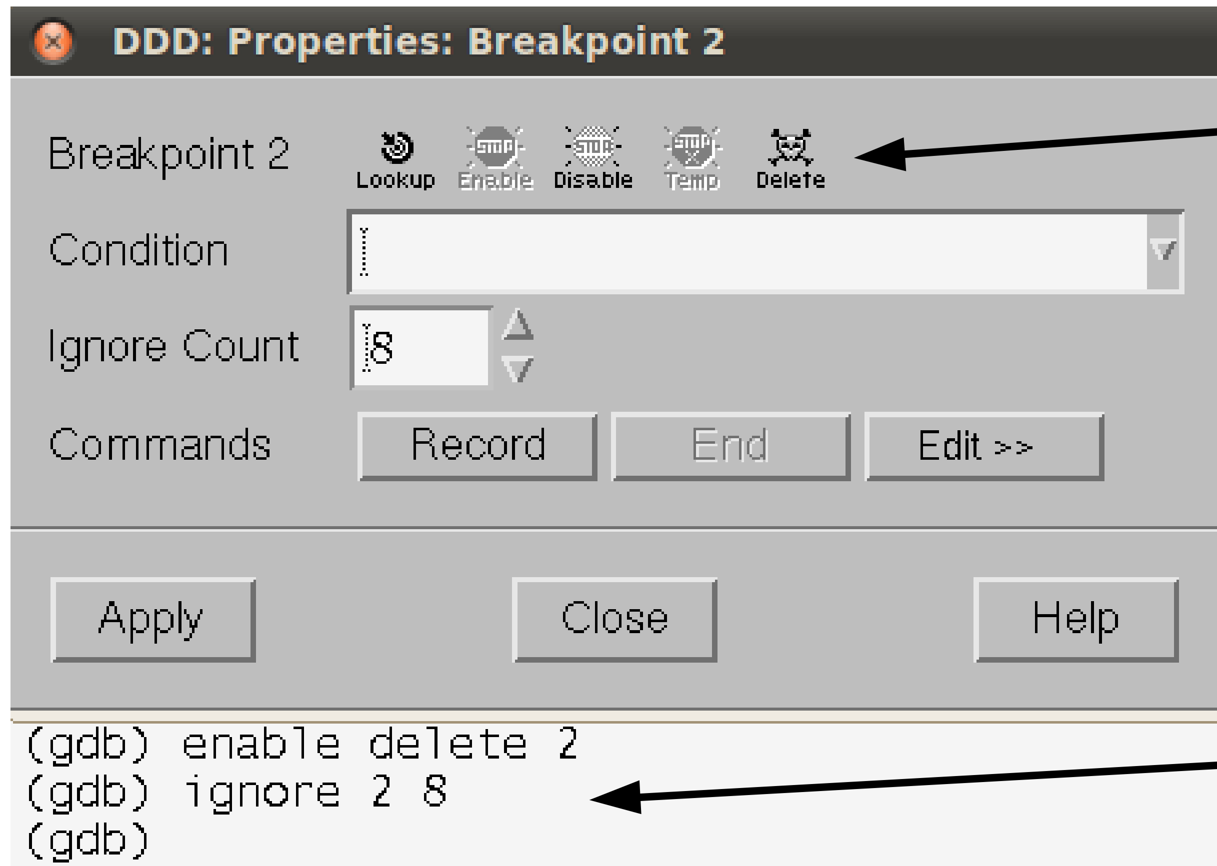
code source programme

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

UNIVERSITÉ DE STRASBOURG

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

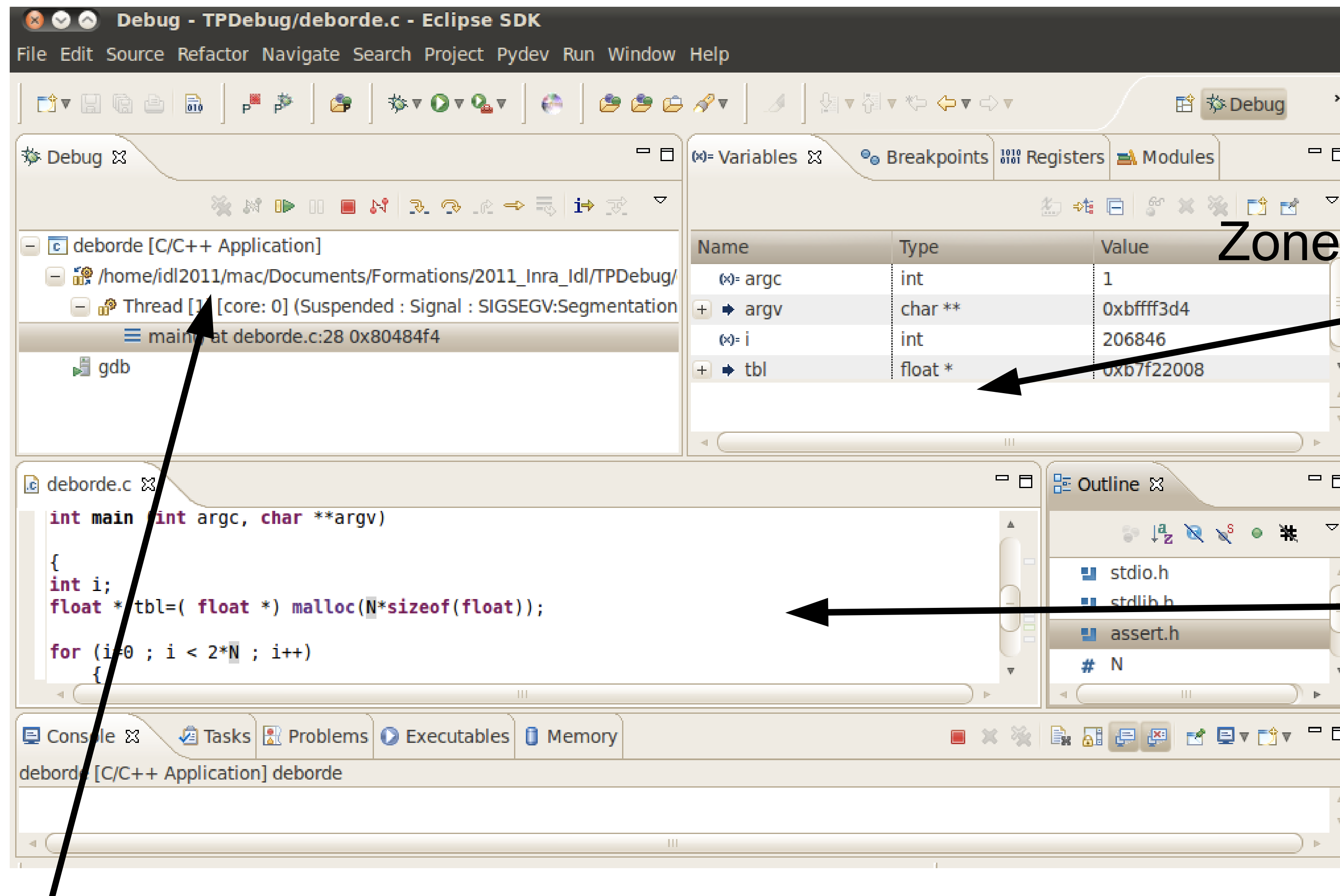
```
meteo.temp=27.2;
meteo.pression=1024;

/* Utiliser les 5 prochai
*/
meteo.releves=allouer_mat

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

Commande gdb correspondante

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

Débogueur - CDT



Zone **des** programmes

Débogueur – CDT - base

Debug

Reprendre, pause

Stepping

Terminer, détacher

UNIVERSITÉ DE STRASBOURG

Débogueur – CDT – breakpoint

Properties for

type filter text

Actions
Common
Filter

Actions

Actions for this breakpoint:

Name	Type	Summary
Bip	Sound Action	
Let's go again	Resume Action	Resume after 10 secon

Remove Up Down

Available actions:

Name	Type	Summary
Bip	Sound Action	
Let's go again	Resume Action	Resume after 10 secon

Music debugging

Console Tasks Problems Executables Memory Progress

Execute breakpoint actions

Resume after 2 seconds

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
gdb	lui-même	tableaux	oui	client/serveur	Tous
idb	idbc		non	gui	Linux, Mac
ddd	gdb, dbx, pydb, xdb, wdb	courbes 1D, (un peu) 2D	oui	gui, représentation données	Linux, Mac
cdt	gnu	non	oui	gui, intégration eclipse	Tous
ddt	lui-même	oui	non	gui, parallèle, GPU	Linux
totalview	tv9cli	tableaux, tranches	non	gui, parallèle, GPU	Linux, Mac
lldb	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)

Fin du cours…

À vous la main!

Des questions avant de plonger dans le code?