Debugger et utilisation de IDB au CTIG*

Olivier Filangi[12]
olivier.filangi@rennes.inra.fr

[1] **CATI SICPA**
Systèmes d'Informations et Calcul pour le Phénotypage Animal

[2] **UMR PEGASE**
Physiologie, Environnement et Génétique pour l'Animal et les Systèmes d'Élevage

révision 27/11/2012

*Centre de Traitement de l'Information Génétique

PLAN

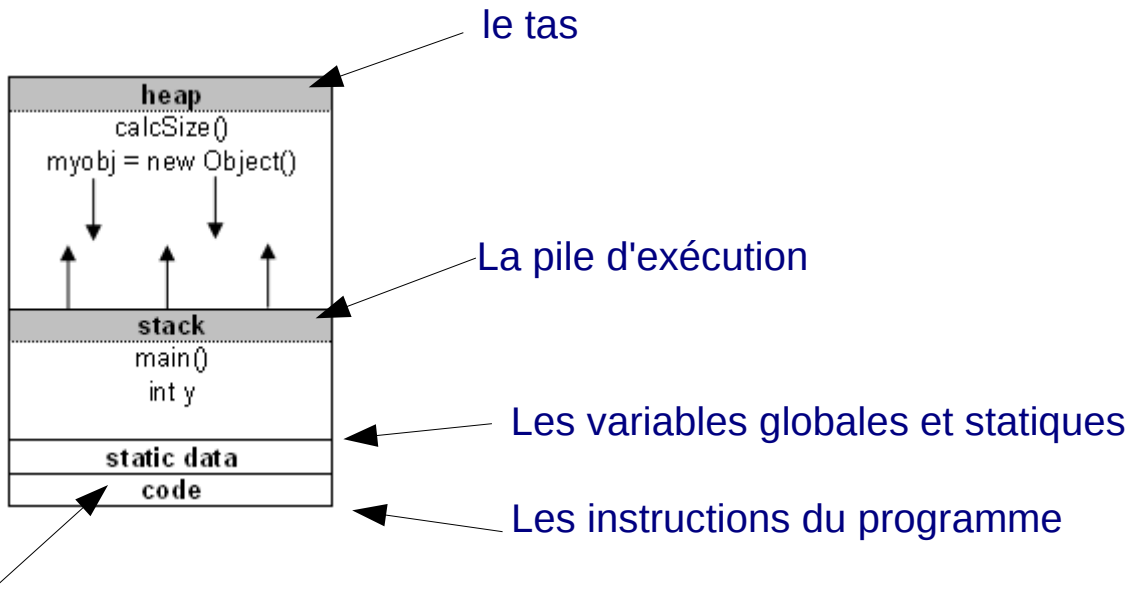
- Segmentation d'un programme
- Pagination
- Allocation des variables dans un processus
- Les erreurs de segmentation
- Utilisation de IDB

Segmentation d'un programme en mémoire

Tous les programmes ont un espace d'adressage propre divisé en segment.

Un segment mémoire est un espace d'adressage indépendant et défini par deux valeurs (*base* et *offset*)

- Chaque composante du processus (heap,stack,...) a son propre segment
- Il y a des droits d'accès aux segments (R,W,R/W)
- La taille d'un segment peut être variable (heap,stack)
- Un segment peut être partagé par plusieurs processus



Le code assembleur du programme produit par le compilateur

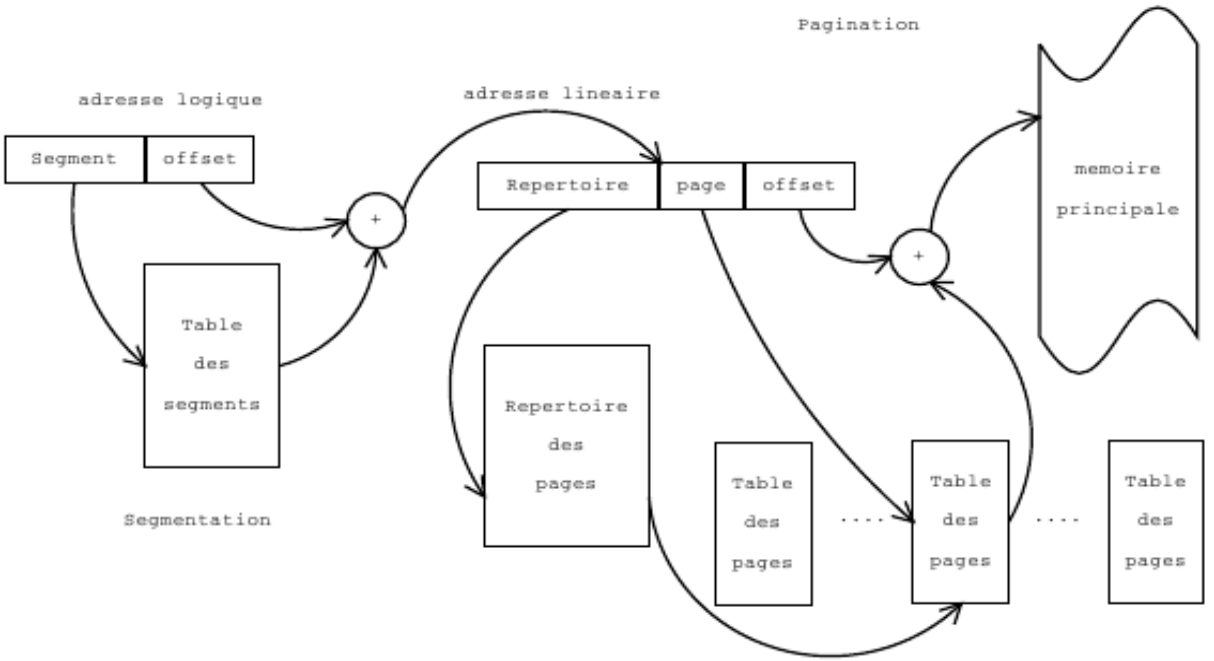
```
movb $0x61,%al   ⇒   10110000 01100001
```

⇒ (10110000 = movb %al
01100001 = $0x61)

Pagination

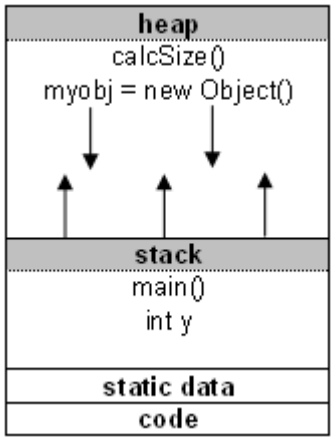
Pour accéder à un octet dans un segment, on utilise une adresse logique (base+offset).

Un segment s'étend sur plusieurs pages. Le système propose un mécanisme de traduction d'adresse relative en adresse physique



Le mécanisme de pagination permet de conserver en mémoire centrale une partie des segments utilisés par les processus et de stocker le reste sur le disque

Allocations de mémoire



Allocation des variables dynamiques
- allocate, deallocate
- Variable allocatable ou pointer
- *Allocation et libération de la mémoire à la charge de l'utilisateur*

Allocation des variables automatiques
- Variables locales des fonctions et subroutines
- *Allocation et libération de la mémoire à la charge du programme*

Allocation statique
- Variables globales et statiques (save) du programme
- *Allocation à la charge du compilateur*

Les instructions du programme

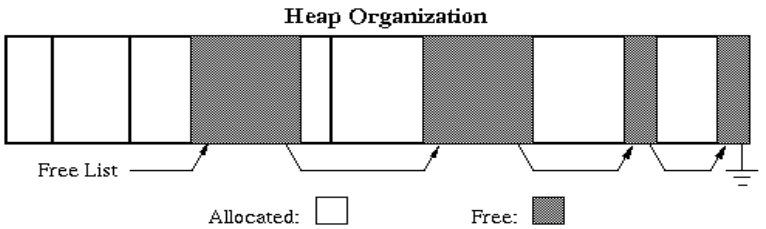
Allocations de mémoire (2)

Allocation statique

- Quantité de mémoire consommée constant et connue avant l'exécution
- Peux coûteuse (tps d'exec)
- Inflexible et insuffisante pour les programmes dont les besoins peuvent varier de façon imprévisible

Exemple : blk.incl 

Allocation dynamique

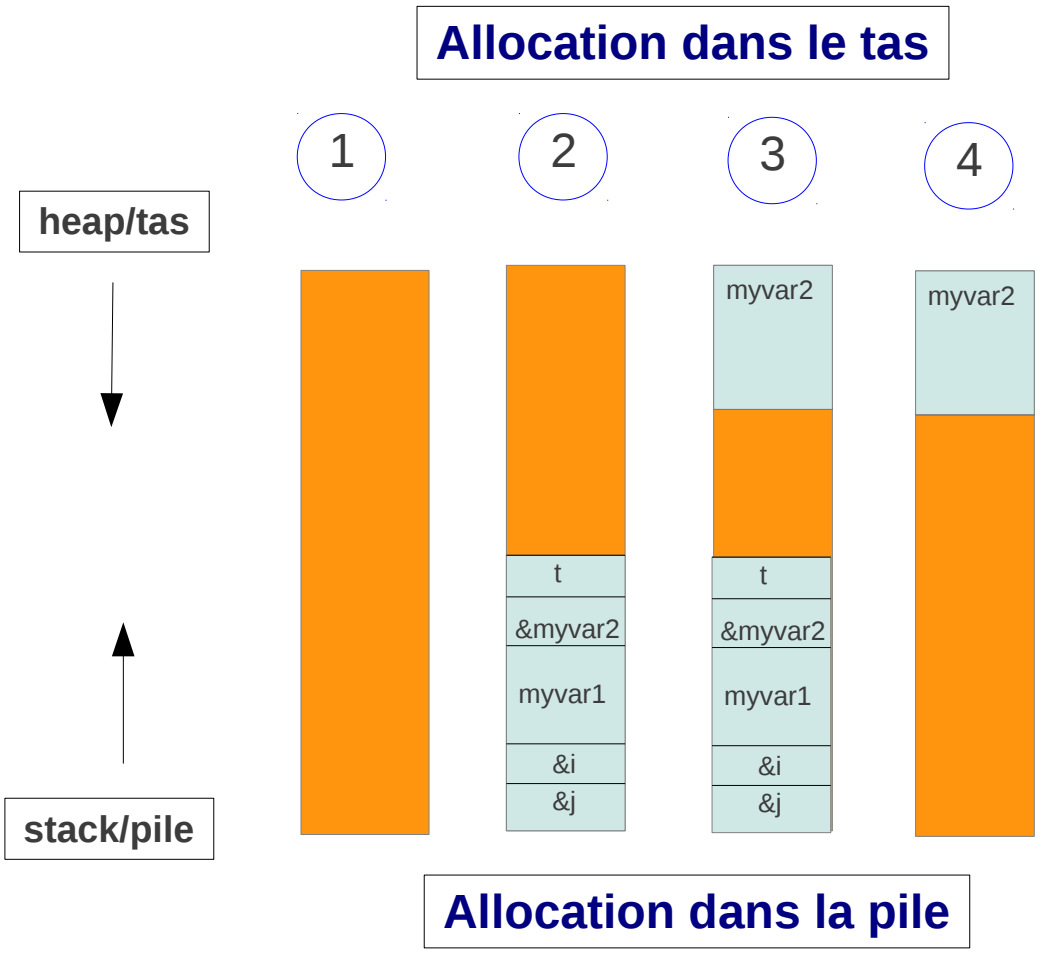


- Utilisation d'un allocateur : allocate/deallocate
- Coûteux (appel, réorganisation de la mémoire)
- Flexible => permet un contrôle arbitraire de la mémoire
- Peut être problématique (ex : mémory leaks)

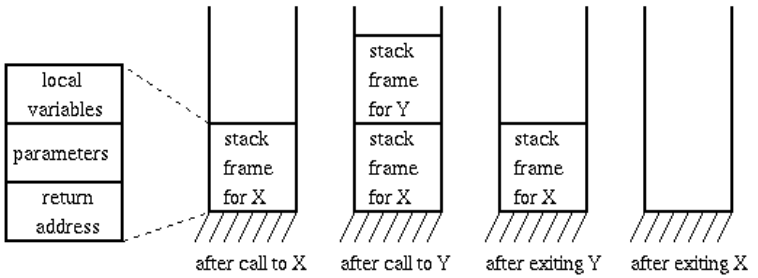
Allocation dans la pile d'exécution

- Quantité de mémoire consommée dépendant des paramètres de la fonction en cours d'appel
- Pas trop coûteux
- Pas de gestion de mémoire

L'allocation (variables automatiques et dynamiques)




```
program monprog
   integer ,save :: i,j

   i=1
   j=myproc(i)

contains

function myproc(ii) result(t)
   integer, intent(inout)              :: ii
   integer, dimension(10)              :: myvar1
   integer, dimension(:),allocatable   :: myvar2
   integer                             :: t
   …
   allocate(myvar2(10))
   …

end function myproc

end program monprog
```



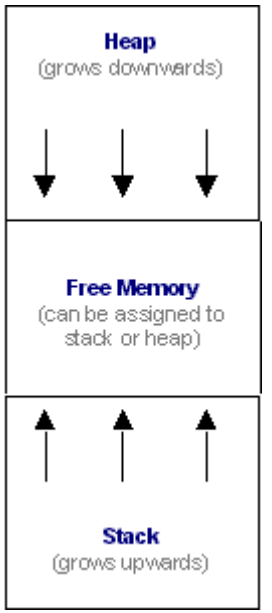
Stack overflow

(débordement de la pile d'exécution)

- Récursivité infinie
- Allocation de variables trop grande dans la pile

Stack size : `ulimit -s` =>10 Mo

The other supported memory parameters, data segment, total virtual memory, and total real memory, are unrestricted.



If the large arrays are stack-based, the potential exists for exceeding the inherited stack size limit. When this happens, the process is terminated abnormally with a "**Segmentation violation**" signal.

Segmentation fault/Erreur de segmentation

- Pointeur NULL
- Accès à la partie réservée au noyau
- Accès à une adresse inexistante (en dehors de l'adressage d'un processus)
- Écrire dans un segment Read-only
- ...

Heap vs Stack

```
program segfault1
  implicit none

  integer :: n,nloop=100,iloop
  character(len=10) :: d,t,z
  integer,dimension(8) :: v1,v2
  integer , dimension(:),allocatable :: bench1,bench2

  n = 40960000

  allocate (bench1(nloop),bench2(nloop))

  do iloop=1,nloop

  call date_and_time (d,t,z,v1)

  call foo(n)

  call date_and_time (d,t,z,v2)

  bench1(iloop) = (v2(6)*60*1000+v2(7)*1000+v2(8))-(v1(6)*60*1000+v1(7)*1000+v1(8))

  call foo2(n)

  call date_and_time (d,t,z,v1)

  bench2(iloop) = (v1(6)*60*1000+v1(7)*1000+v1(8))-(v2(6)*60*1000+v2(7)*1000+v2(8))

  print *,iloop
  end do

  print *," foo m temps (ms):",real(sum(bench1))/real(nloop)
  print *," foo2 m temps (ms):",real(sum(bench2))/real(nloop)

  deallocate (bench1,bench2)

contains
  subroutine foo(n)
    integer, intent(in) :: n
    integer, dimension(n) :: a, b
    a = 1
    b = 2
    a = a + b
  end subroutine foo

   subroutine foo2(n)
    integer, intent(in) :: n
    integer, dimension(:), allocatable :: a, b
    allocate(a(n), b(n))
    a = 1
    b = 2
    a = a + b
    deallocate(a, b)
  end subroutine foo2

end program segfault1
```

command	Exec - dga12	Exec - dga12(qrsh)
`ifort test.f90`	**Erreur de segmentation (core dumped)**	Foo : 200 ms Foo2 : 250 ms
`ifort -heap-array 1024 test.f90`	ok	Foo : 400 ms Foo2 : 250 ms
`ulimit -s`	10240	unlimited

automatic array (stack allocation)

dynamic array (heap allocation)

IDB (Intel Debugger)

Preparing a program for debugging

Compile and link the program using the **-g** option

```
ifort -g test.f90
```

Limitation of core dump :

```
ulimit -c
```

Setting a new value :

```
ulimit -c unlimited
```

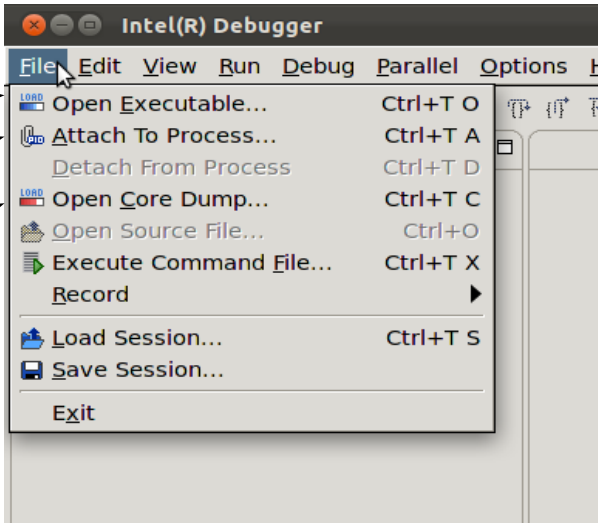
Starting debug session

```
>idb &
```

Exécutable dans un répertoire

Processus qui tourne

Processus qui s'est terminé anormalement avec
Un core dump pour le debuggage



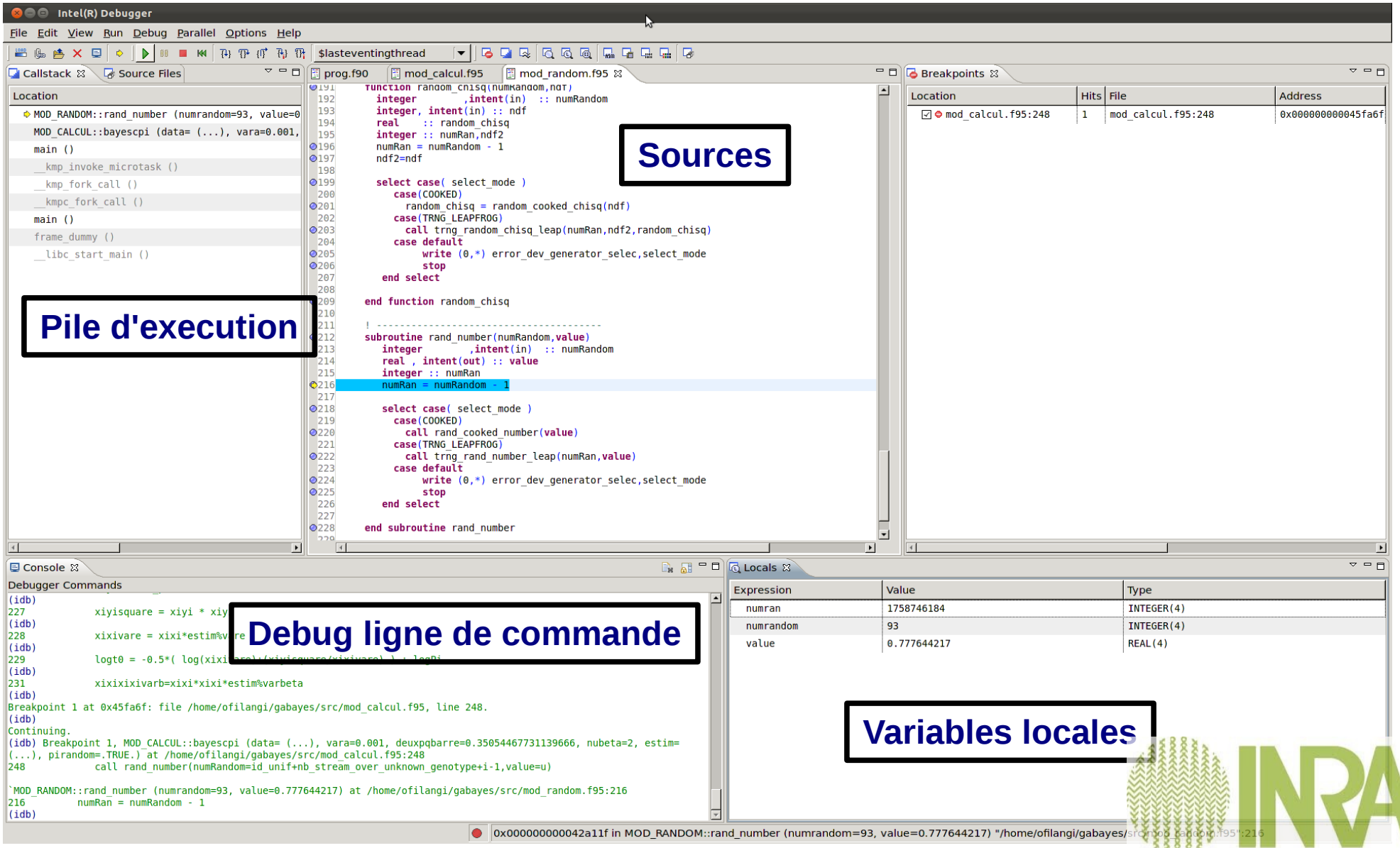
- **idb sur dga12**
- **Le programme doit être lancé via un qsub ou avec un qrsh/qsh**

- Pour être conforme à la politique de gestion des processus par SGE (*le handler ne gère pas les processus exécutés en frontal*)
- Reproduire le bug dans l'environnement d'exécution

TP 1

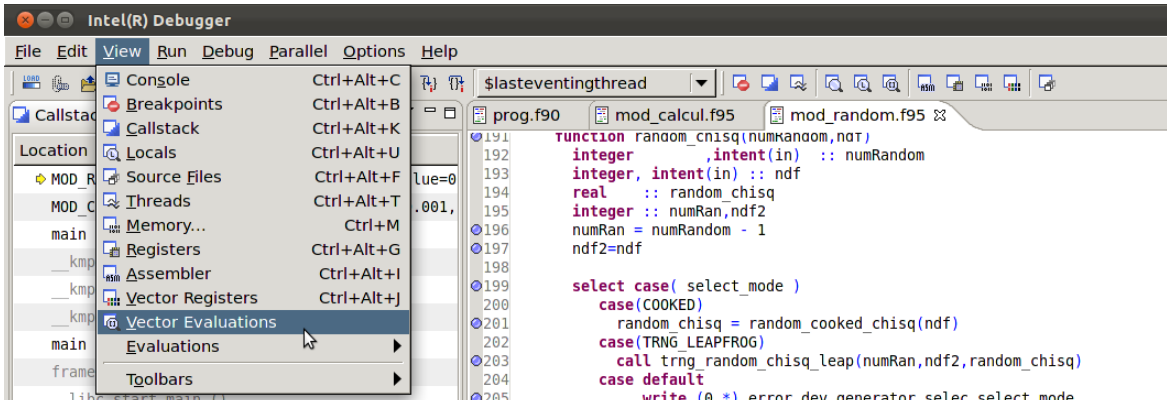
- Téléchargez **env-dga12.sh.gz** et **gabayes-debug-tp.tar.gz** du projet « Formation Debugger-IDB » (Forge DGA)
- Executez l'application avec qsub dans le répertoire sample :
  - `qsub -b y ../build/gabayes p_analyse`
- Exécutez idb et « attachez » votre processus gabayes

IDB : le tableau de bord

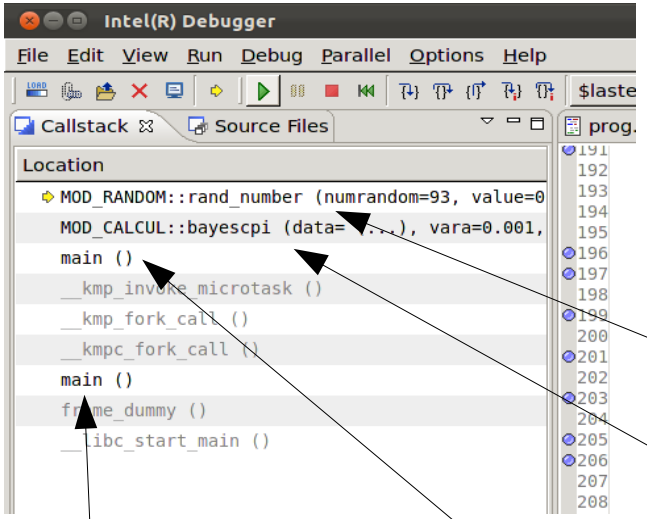


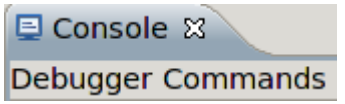
Les vues eclipse dans la perspective IDB

- L'onglet view donne l'ensemble des vues disponibles de la perspective IDB
- Accessibles via les raccourcis (icône,clavier)
- Les vues sont modifiables à volonté (taille,fermeture,ouverture, choix du cadre via un drag and drop, « detached view »)



Examiner la pile d'exécution






```
>backtrace
```

Subroutine `rand_number` du module `mod_random`

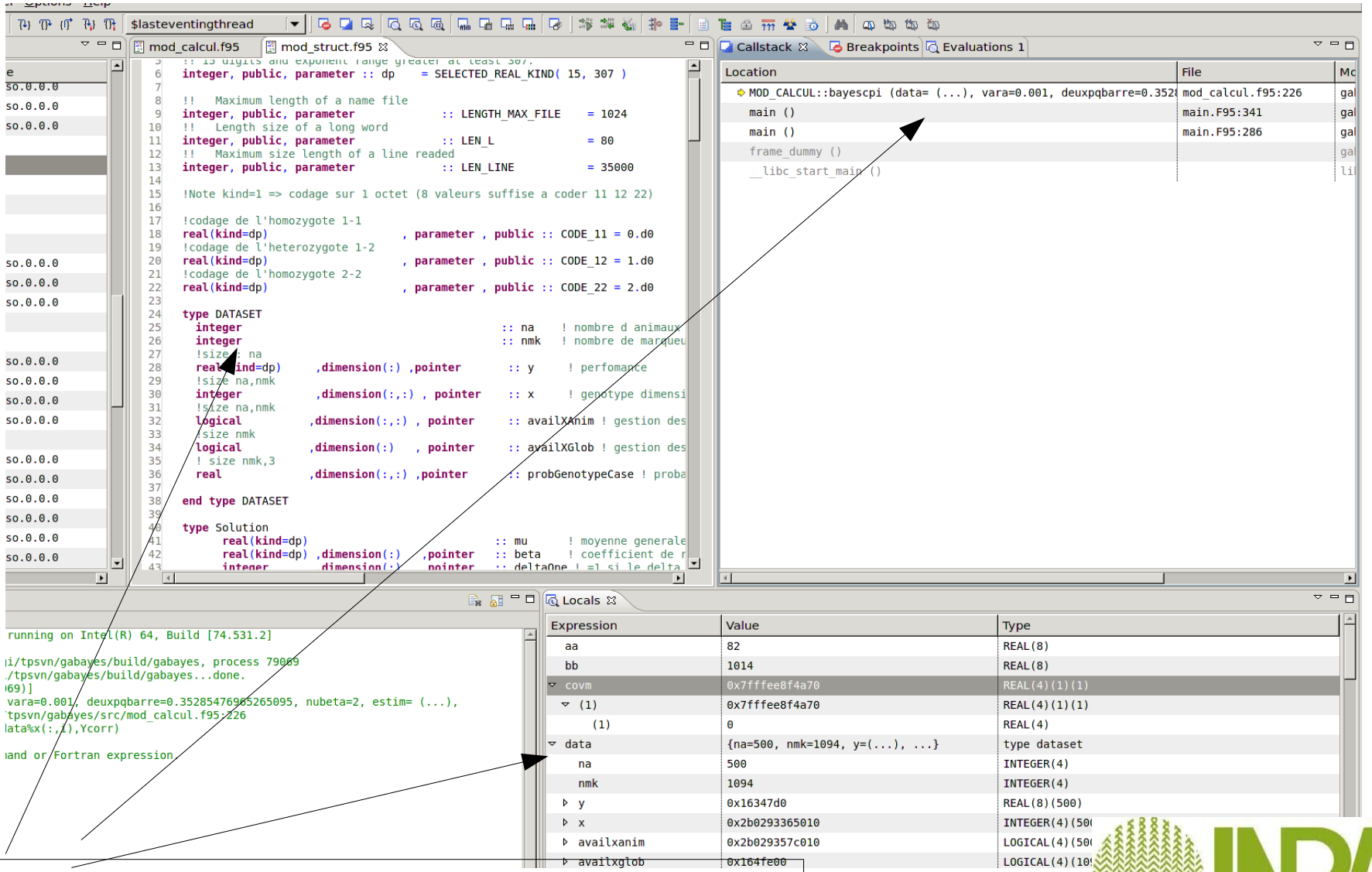
Racine du programme
Main() généré par ifort

Program `main`

```
1 program main
2     use mod_struct
3     use mod_lecture
4     use mod_calcul
5     use mod_random
6
```

Subroutine `bayescpi` du module `mod_calcul`

Examiner l'état des variables locales



La variable **data** de type **dataset** dans la subroutine **bayescpi** du module **mod_calcul**

Variables

Les variables « parameter » ne sont pas accessibles (optimisation du compilateur sinon il faut utiliser l'option `-debug-parameter all`)

Console
Debugger Commands

```
>print ma_variable
```

Ou utiliser l'adresse de la variable et afficher l'état de la mémoire

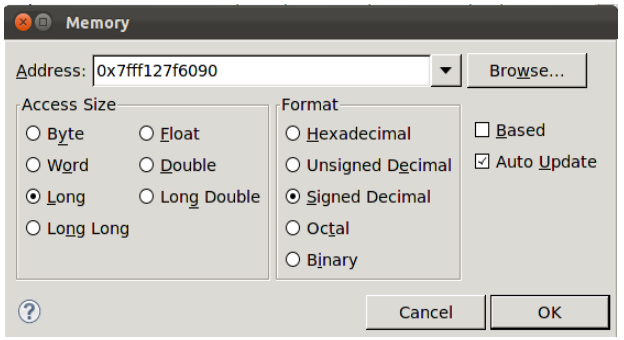
Console
Debugger Commands

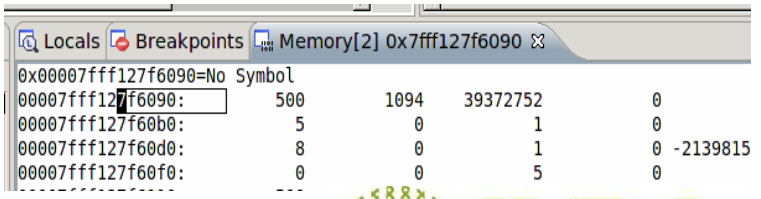
```
>print &ma_variable
```

View memory

Dump de la mémoire

```
(idb) print &data
$5 = 0x7fff127f6090
```





Variables

Accéder aux variables d'un type structuré :

```
> print data.nmk
```

Accéder à l'adresse d'une variable d'un type structuré :

```
> print &data.na
```

Evaluer une expression

```
> print data.nmk+data.na
```

Afficher le contenu d'un module (variables et fonctions)

```
> print mod_random
```

Afficher une variable privée ou publique d'un module

```
> print mod_random%select_mode
```

Regarder l'état de vos variables locales et afficher le contenu de la variable privée nb_stream_over_unknown_genotype du module mod_calcul

Changer le contexte d'exécution

Modifier le contenu d'une variable

```
> set mod_random%select_mode=1
```

Créer une variable pour le debuggage

```
> set $myCountLoop=1
```

Appeler une fonction/subroutine

```
> call mod_random%init_random(1)
> print mod_random%init_random(1)
```

Testez la fonction random_normal du module mod_cooked_random

Contrôler l'exécution du programme (débuggage pas à pas)

stop l'exécution du programme

prochaine ligne dans le code source

prochaine ligne dans le code source, saute un appel de fonction

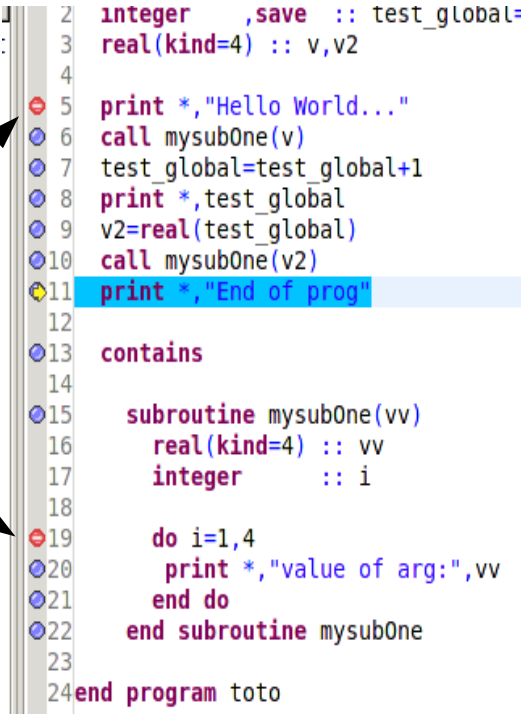
sort de la fonction courante, prochaine instruction dans le source

prochaine instruction machine, saute un appel de fonction

Prochaine instruction machine

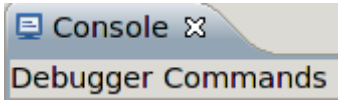
Breakpoint

le programme s'arrête à une ligne spécifique du source



Breakpoint
(**double click**)

OU




```
>break mod_cooked_random%random_normal
>break mod_calcul.f95:156
```

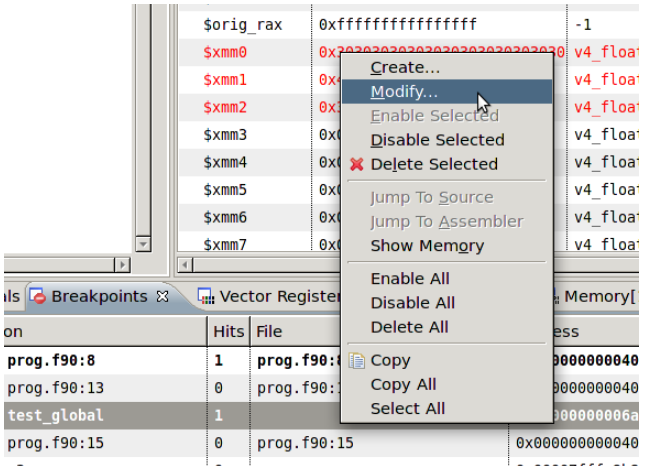
Fonction/subroutine

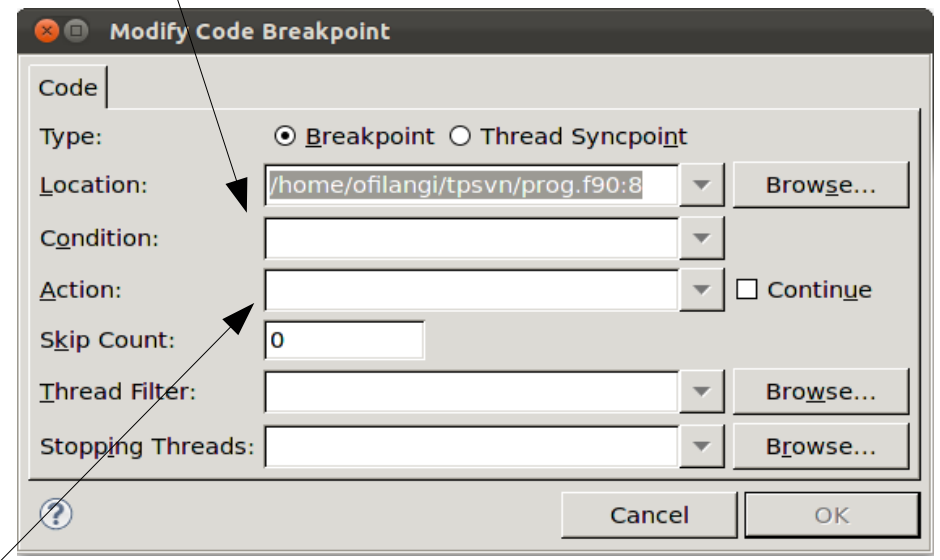
Ligne de code

Configuration du Breakpoint

View breakpoint => Modify

Expression en fortran (**.or.**,**.and.**,...)





```
call
return
step
continue
goto
next
expression
```

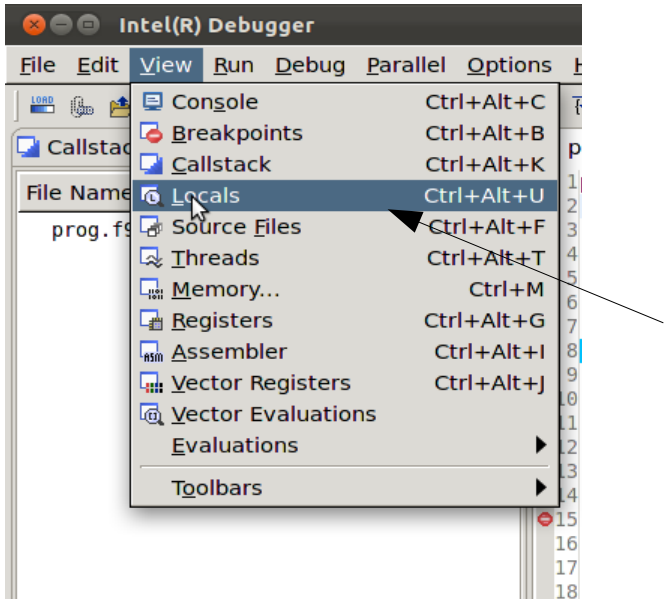
On peut mettre une seule action

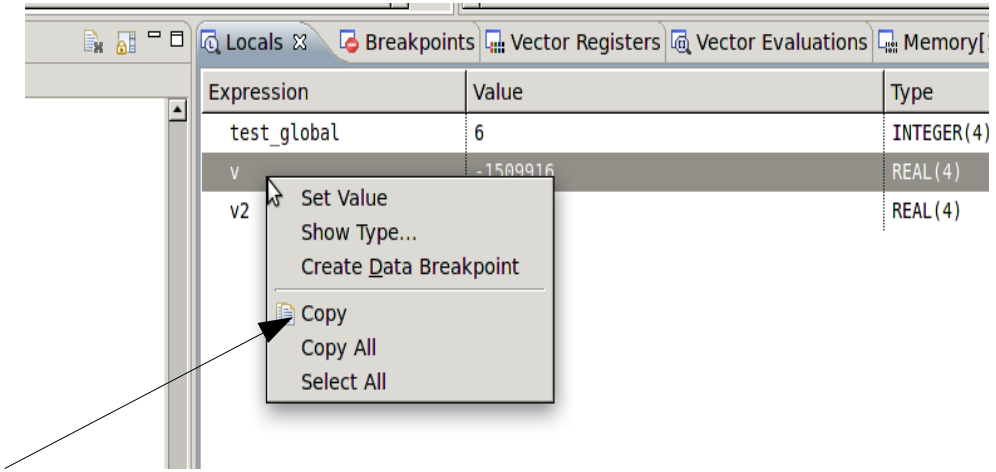
TP

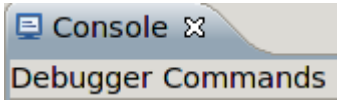
- Mettre un breakpoint dans la fonction random_chisq (module mod_random)
- Modifier la variable select_mode pour appeler la routine random_cooked_chisq du module mod_cooked_random
- Conditionnez l'arrêt du breakpoint dans la fonction random_chisq avec :
  - `(ndf>600).and.(numRandom==2)`

WatchPoint

le programme s'arrête quand on écrit à une adresse



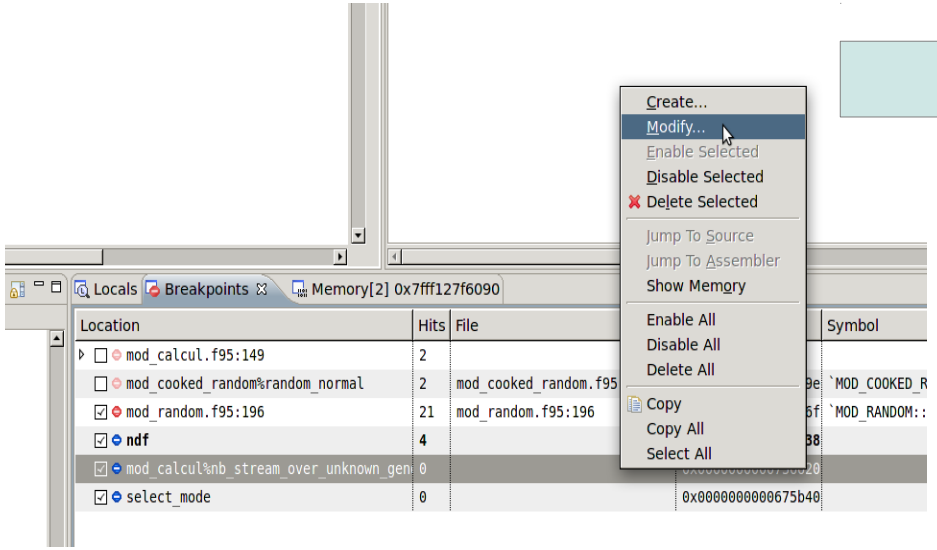


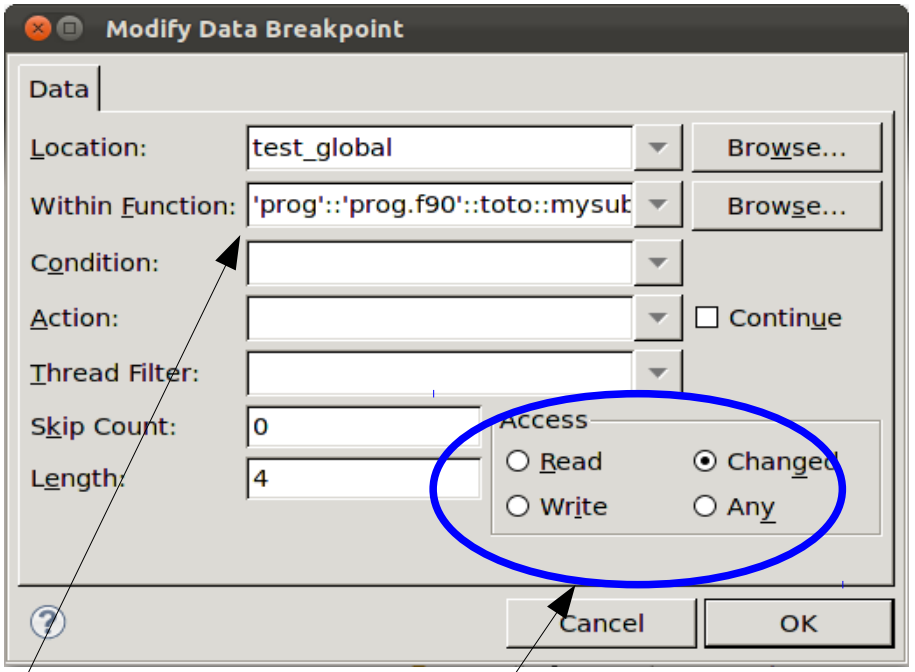


Variables globales

```
>watch mod_calcul%nb_stream_over_unknown_genotype
```

WatchPoint





Pas d'arrêt dans idb....

Contexte d'exécution

Type d'accès à la variable

Possibilité d'utiliser gdb en ligne de commande

Mode GDB...

IDB est configuré pour avoir la même syntaxe que GDB

Il peut être utilise de lancez un debugger en mode console pour obtenir une information rapidement (localiser l'erreur)

Principales commandes :

- run - r
- continue - c
- break FCT (fonction ou subroutine) - b
- break NUM (ligne de code) – b
- delete/enable NUM (le numero du point d'arrêt)
- step (pas élémentaire) - s
- next (saute l'instruction) - n
- finish (sort de la fonction et arrêt)
- list (lignes de codes par paquet de 10) - l
- where (affiche la position courante dans le code)

Cas d'utilisation après une modification du source :

```
>make
>gdb a.out
Starting program: /home/ofilangi/...../a.out
>r
Program received signal SIGSEGV, Segmentation fault.
0x0000000000402c9a in sub (x=..., y=...,
n=@0x7fff40a00000) at seg_one.f:11
11                x(i) = i**2
>where
#0  0x0000000000402c9a in sub (x=..., y=...,
n=@0x7fff40a00000) at seg_one.f:11
#1  0x0000000000402b87 in main$seg_one_$BLK () at
seg_one.f:4
#2  0x0000000000402b2c in main ()
```

TP

Téléchargez les programmes seg_one.f et seg_two.tar.gz du projet Formation Debugger-IDB.

Détectez les problèmes avec IDB