Choisissez votre style : **colorisé**, **impression**

Série 12 (Niveau 0):
Utilisation d'un dévermineur

Le but de cet exercice est de vous montrer comment utiliser un dévermineur (« debbuger »). Les dévermineurs sont des outils vous permettant de traquer les problèmes d'exécution dans un programme. **L'utilisation d'un dévermineur est vivement recommandée dans le cadre du projet**.

Déverminage sous Geany

Si vous développez en utilisant un éditeur autre que Geany, vous pouvez utiliser le programme `ddd` (**voir plus bas**), il n'est tout de fois pas d'un confort optimal.

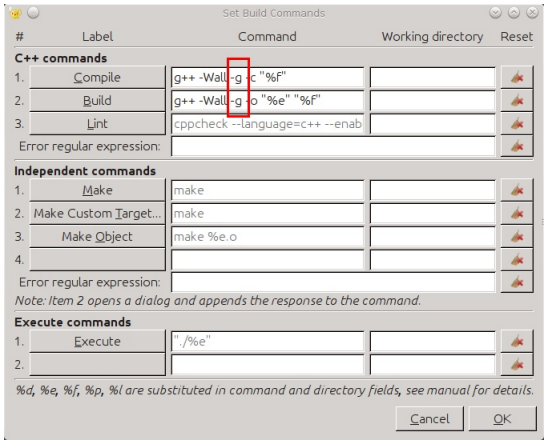
Déverminage avec `Geany`

1. Compiler pour le déverminage

Dans `Geany`, ouvrez un fichier `divisions.cc`, et introduisez-y le programme suivant :

```
#include <iostream>
using namespace std;
int main()
{
  int a, b;
  a = 1024;
  b = a*a*a-1;
  a = 2*b;
  b = a+1;
  a = b+1;
  b = 4*b;
  a = 2*a;
  b = b/a;
  cout << b << endl;
}
```

Important : pour pouvoir utiliser le dévermineur associé à `Geany`, il faut régler les options de compilation pour le compiler avec l'option *-g* : `Build > Set Build Commands` :

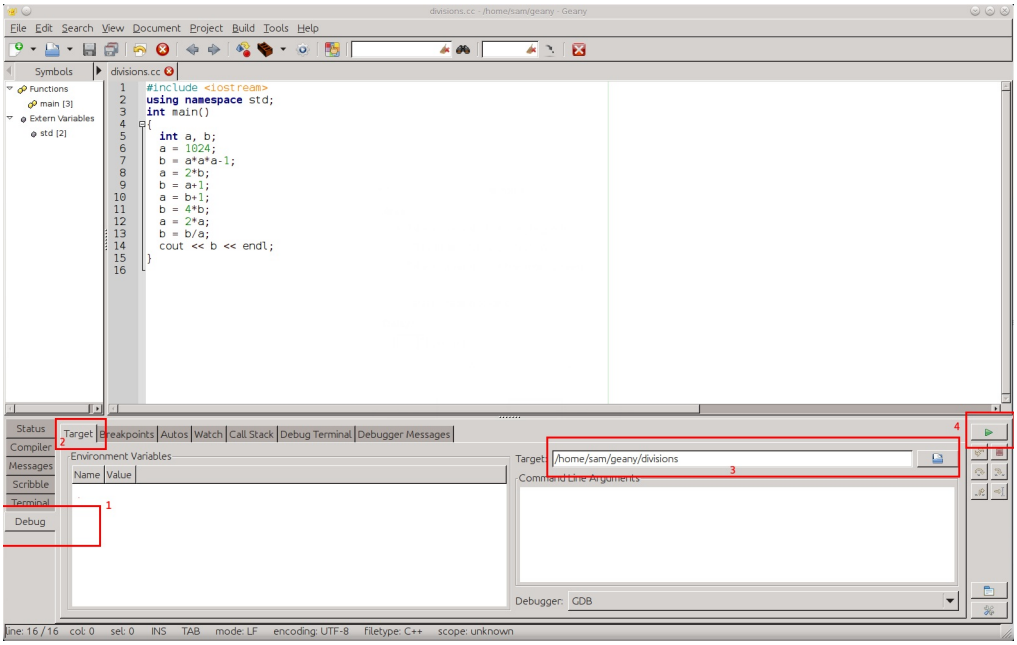


Lancez la compilation de `divisions.cc` dans `Geany` (bouton `F9`). Tout devrait se passer comme d'habitude. Si vous lancez l'exécution (dans Geany ou dans un terminal), le programme s'arrête avant la fin, et vous obtenez un message d'erreur : `Floating exception (core dumped)`

Le dévermineur (« Debugger ») va vous permettre de localiser l'erreur dans le programme et d'en déterminer la cause.

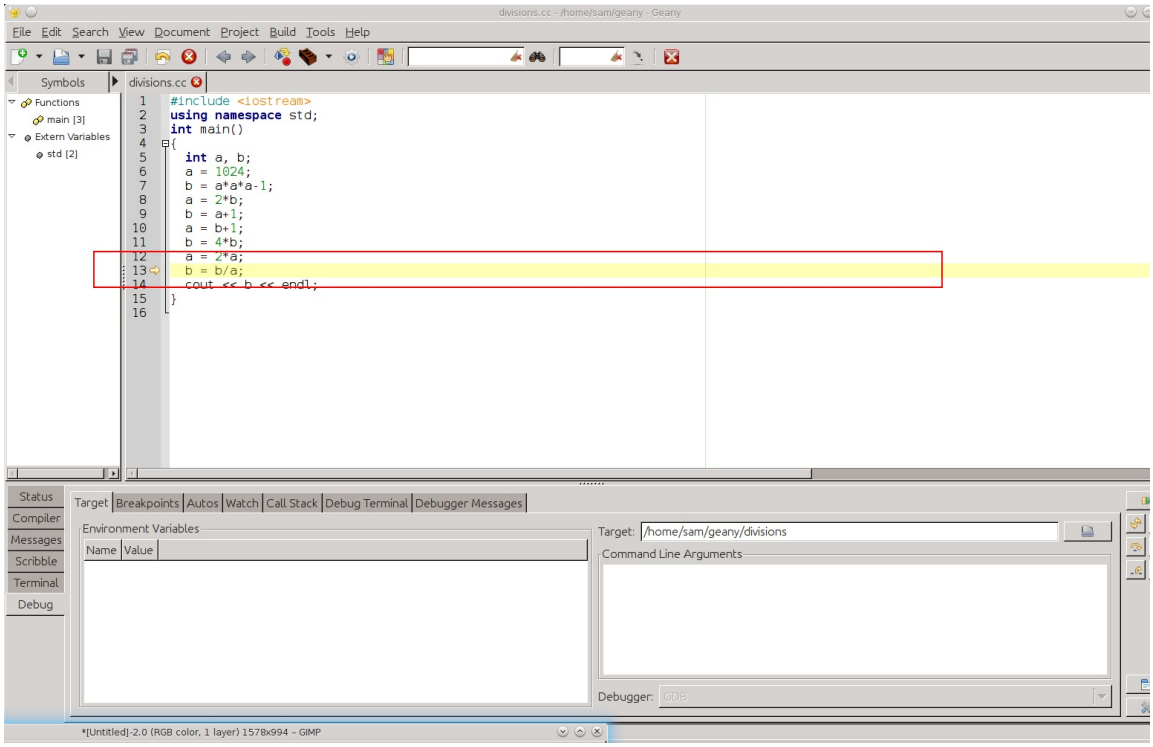
2. Lancer le dévermineur

Pour lancer l'exécution du programme au moyen du dévermineur :



1. cliquez sur le bouton `Debug` (en bas à gauche) dans `Geany`
2. cliquez sur le bouton `Target` ;
3. sélectionnez l'exécutable de votre programme (dans le répertoire où est stocké le programme `divisions.cc`, mais sans l'extension `.cc`)
4. puis cliquez sur la petite flèche verte en haut à droite de la fenêtre de «debugging».

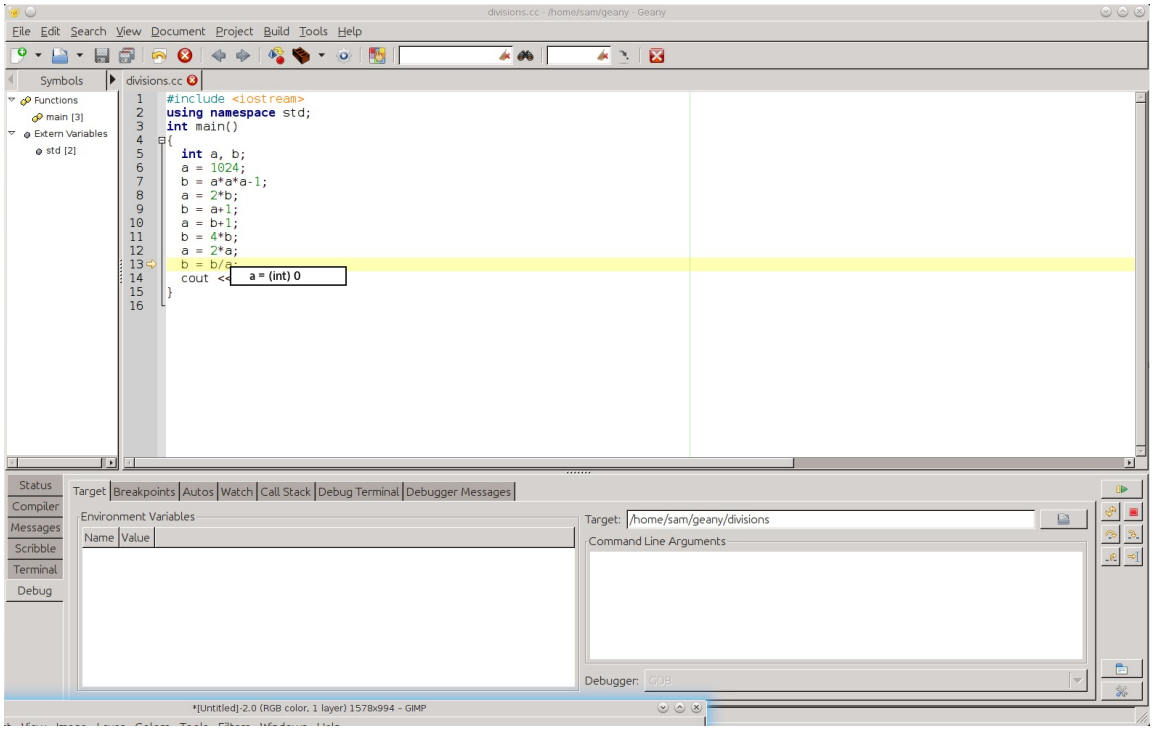
Vous devriez voir s'afficher une fenêtre d'alerte indiquant que le programme s'est terminé avec une erreur. Lorsque vous fermez cette fenêtre vous pouvez voir que la ligne de code ayant provoqué l'erreur est désignée par une flèche dans `Geany` :



2.Afficher la valeur des variables

Un premier pas vers l'identification des causes de l'erreur consiste à examiner la valeur des variables impliquées dans la ligne fautive.

Faites le pour les variables `a` et `b`, simplement en plaçant votre curseur dessus



L'information sur la valeur de la variable disparaît dès que vous déplacez le pointeur.

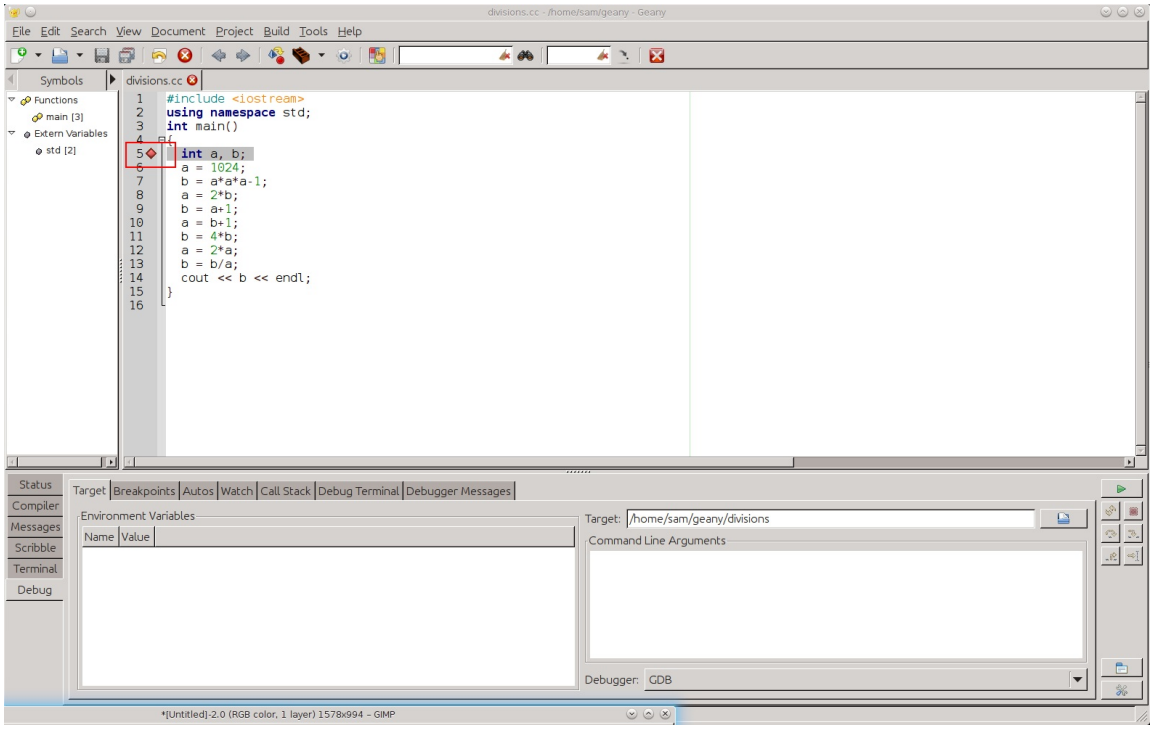
Vous devez pouvoir ainsi observer les valeurs `a=0` et `b=-4`. Ce sont les valeurs des variables au moment où l'erreur a été détectée. La cause de l'erreur devient évidente : la division par `a=0`.

Dans la suite, vous allez exécuter le programme pas-à-pas, pour comprendre à quel moment les résultats des calculs deviennent aberrants.

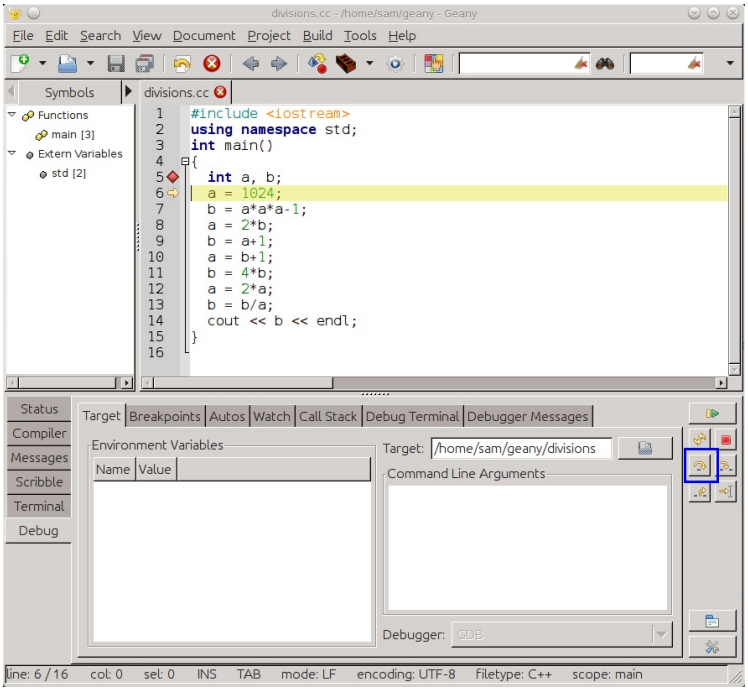
3. Exécuter le programme pas-à-pas

Arrêtez le programme en cliquant sur le petit carré rouge en dessous de la flèche verte que vous avez utilisée pour lancer le programme dans le debugger.

Pour exécuter le programme pas-à-pas, il faut commencer par mettre un point d'arrêt (**breakpoint**) à l'endroit où l'on veut commencer l'observation. Dans cet exemple, on va observer le déroulement du programme depuis le début, c'est-à-dire depuis la première ligne après "`main() {`". Cliquez sur cette ligne dans la marge où apparaissent les numéro de ligne avec le bouton droit de la souris. Un point d'arrêt apparaît sur la ligne sélectionnée, symbolisé par petit losange rouge :



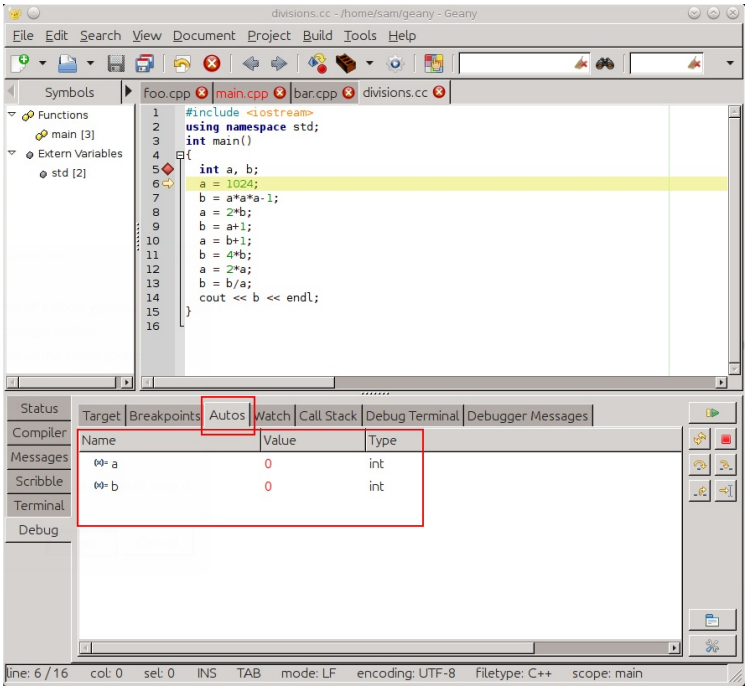
Lancez alors le programme avec la flèche verte. Il s'arrête à la première instruction suivant le point d'arrêt. La flèche dans la zone de programme indique la prochaine ligne qui doit être exécutée :



- pour exécuter une ligne à la fois, cliquez sur **`step over`**, (bouton encadré en bleu ci-dessus).
- si l'instruction est un appel de fonction, il est possible d'exécuter pas-à-pas le corps de la fonction en utilisant le bouton **`step into`** (ce n'est pas utile dans cet exemple);
- Pour continuer le programme jusqu'à la fin, sans s'arrêter à chaque ligne, cliquez sur la flèche verte.

Exécutez le programme pas-à-pas en cliquant sur *Step Over*, et observez l'évolution des valeurs des variables.

Vous noterez que lorsque vous exécutez pas à pas vous pouvez aussi examiner le contenu des variables en sélectionnant l'onglet `Autos` :



À quel moment ces valeurs deviennent-elles aberrantes ?

NB : Le but de cet exercice est de vous faire exécuter un programme pas-à-pas en suivant l'évolution des variables, et non de comprendre pourquoi le programme *divisions.cc* se comporte bizarrement.

Voici cependant, à titre documentaire, l'explication succincte de son comportement :

Le programme a un comportement anormal à partir de la ligne

```
a = b+1
```

En effet, à ce moment-là, la valeur de `b` est la plus grande valeur possible pour une variable de type `int`. En effet le type `int` n'est pas un vrai type entier au sens mathématique du terme. Les variables de ce type sont en fait bornées dans l'intervalle `[-MAX_INT - 1, MAX_INT]`.

Pour l'ordinateur, si `b=MAX_INT`, alors `b+1 = -MAX_INT - 1` !!!

Et si `a=-MAX_INT - 1`, alors `2*a = 0` !!!

Bref, dès que l'on dépasse les capacités de représentation, les résultats donnent n'importe quoi du point de vue de l'arithmétique !

Le tout est de le savoir !

4. Programme avec plusieurs sources

Fermez le fichier `divisions.cc` dans `Geany`.

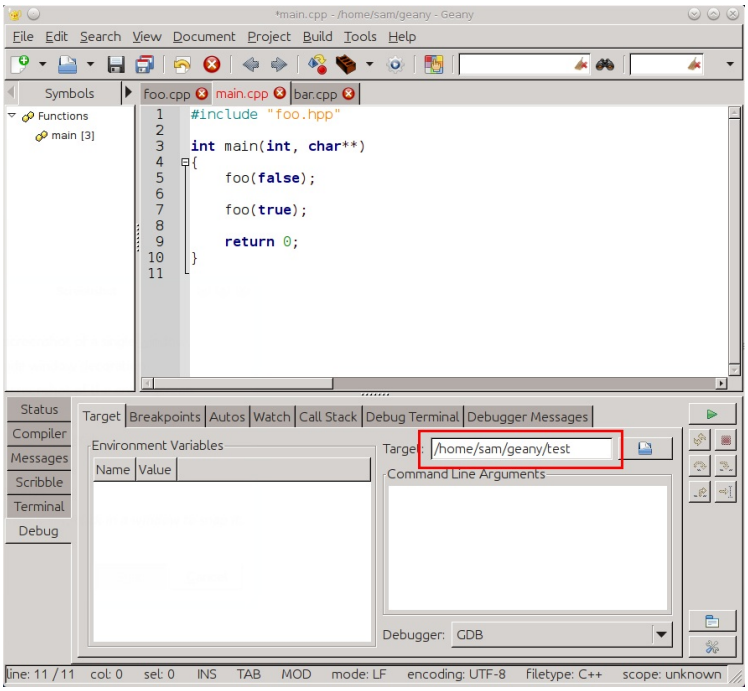
Pour cette sous-section et la suivante, **téléchargez l'exemple fourni** et désarchivez-le dans le dossier de votre choix (depuis le terminal vous pouvez exécuter `unzip dddTest`).

Il s'agit d'un programme constitué de plusieurs fichiers (le but étant de vous montrer comment l'outil de déverminage vous permet de naviguer entre plusieurs fichiers source. Ce que vous serez amenés à pratiquer intensivement au semestre de printemps!)

Rendez-vous dans un terminal dans ce dossier et exécutez `make` afin de compiler le programme (ne vous préoccupez pas de cet aspect, nous reviendrons à la compilation séparée en temps voulu). Vous pouvez à présent lancer le programme avec `./test`. Vous remarquerez que le programme ne fonctionne pas ("plante"). Nous allons voir pourquoi et en profiter pour explorer certaines fonctionnalités du dévermineur de `Geany`.

Commencez par ouvrir tous les fichiers impliqués dans `Geany`: `main.cpp, foo.cpp, bar.cpp`. Sélectionnez la fenrêtre contenant le programme principal (`main.cpp`).

La commande `make` a en fait produit dans le répertoire où se trouvent ces fichiers un exécutable nommé `test` qu'il faudra spécifier comme nouvelle cible du dévermineur via `Debug > Target`:



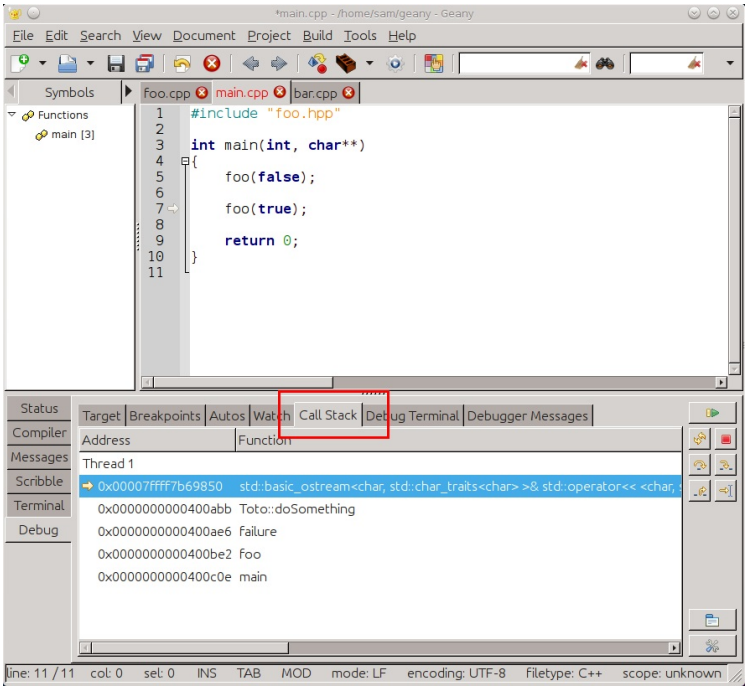
Lancez alors le programme au moyen de la flèche verte, une petite fenêtre s'affiche indiquant qu'une erreur s'est produite. Lorsque l'on ferme cette fenêtre, une flèche indique que l'instruction de la ligne 7 du programme principal est fautive. Cette instruction en tant que telle ne comporte cependant rien d'anormal et implique l'appel à la fonction `foo` placée dans un autre fichier.

Pour situer plus finement la source de l'erreur, il est nécessaire d'examiner l'enchaînement des appels de fonctions ayant abouti à l'erreur. Il faut dans ce cas utiliser la pile des appels comme expliqué ci-dessous.

5. Backtrace

La **backtrace** d'un programme est la liste des fonctions qu'il a exécutées jusqu'à un moment donné, par exemple un crash ou un breakpoint.

Pour visualiser la backtrace au moment du crash que nous venons de provoquer, utilisez le bouton `Call Stack`:

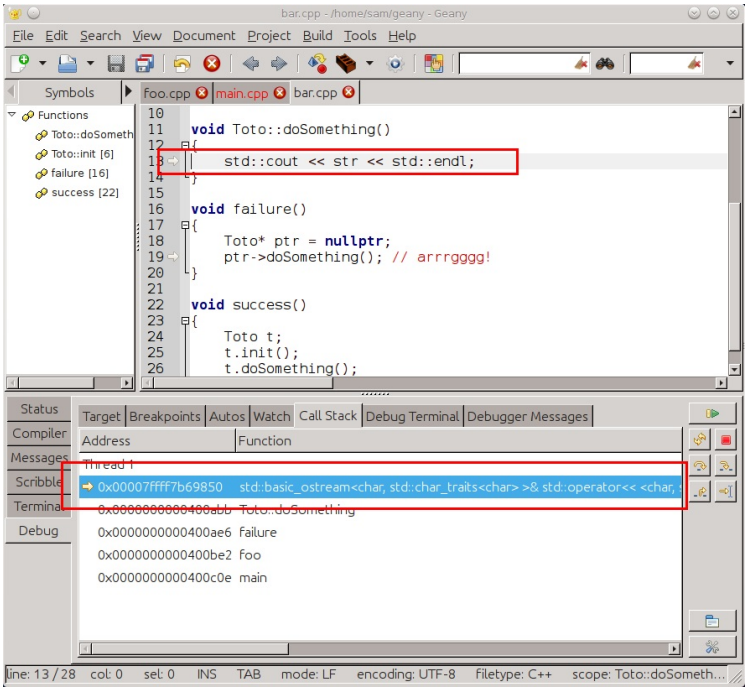


Les fonctions exécutées par le programme sont listées de la plus récente à la plus ancienne avec, pour chaque fonction, le nom du fichier source où elle est implémentée et la

dernière ligne exécutée dans la fonction (par exemple `main.cpp:7`) (déroulez sur la droite la fenêtre contenant la pile des appels pour voir ces informations).

Vous pouvez cliquer sur chacune des fonctions dans la backtrace et verrez à chaque fois la dernière instruction exécutée marquée par une petite flèche dans la marge.

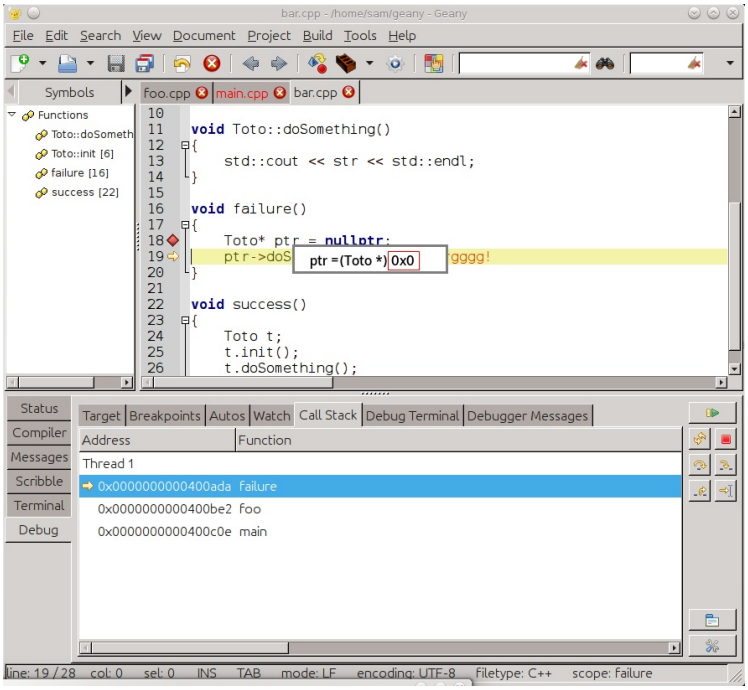
D'après la backtrace, la toute dernière instruction provoquant le crash a lieu lors de l'appel de l'opérateur `<<` fourni par la librairie standard :



Il est très peu probable que ce soit l'opérateur `<<` qui soit erroné! Il faut garder en tête que le crash peut être dû à une erreur en amont dans le code. Remontez alors d'un cran dans la pile des appels (il peut être parfois nécessaire de remonter plus haut). Vous vous retrouverez au niveau de la fonction `failure`. L'erreur saute en principe aux yeux (accès via un pointeur nul), mais supposons que ce soit moins évident. La chose à faire ici serait de :

- stopper le programme au moyen du petit carré rouge dans le débermineur;
- placer un point d'arrêt au début de la fonction `failure`;
- puis relancer l'exécution au moyen de la flèche verte.

L'examen du contenu des variables vous montrera alors le pointeur nul:



Dans la «vraie vie», il faudrait alors comprendre pourquoi ce pointeur a une telle valeur et apporter la correction nécessaire. Ce type d'erreur est assez fréquent!

Déverminage avec `ddd`

1. Compiler pour le déverminage

Important : pour pouvoir déverminer les programmes avec un dévermineur (`ddd`), il faut le compiler avec l'**option de compilation *-g***.

Avec un éditeur, ouvrez un fichier `divisions.cc`, et introduisez-y le programme suivant :

```
#include <iostream>
using namespace std;
int main()
```

```
{
  int a, b;
  a = 1024;
  b = a*a*a-1;
  a = 2*b;
  b = a+1;
  a = b+1;
  b = 4*b;
  a = 2*a;
  b = b/a;
  cout << b << endl;
}
```

Cliquez sur le bouton *Compile*, et changez la commande de compilation en :

```
c++ -g divisions.cc -o divisions
```

(si vous utilisez un `scons` pour compiler, ajoutez l'option `-g` à `CCFLAGS` (déjà mis dans les fichiers fournis pour le projet))

La compilation s'exécute de la même façon que précédemment. Lancez alors le programme dans un terminal, par la commande *divisions* (dans le répertoire où il se trouve). Le programme s'arrête avant la fin, et vous obtenez un message d'erreur : `Floating exception (core dumped)`

Le dévermineur va vous permettre de localiser l'erreur dans le programme et d'en déterminer la cause.

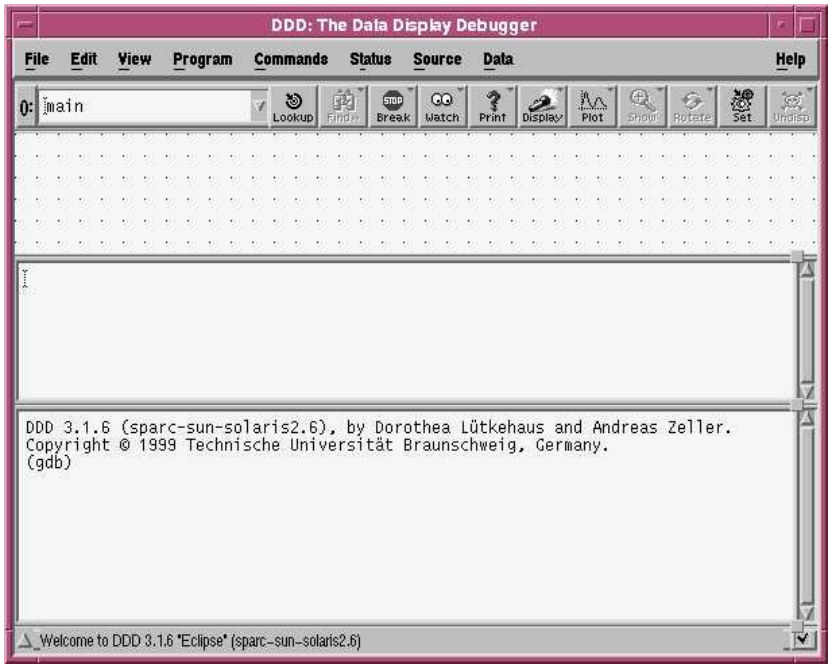
2. Lancer *ddd*

Dans le répertoire où vous avez le fichier source `divisions.cc` et l'exécutable `divisions` (*a priori* dans le répertoire `serie14`), tapez la commande : `ddd divisions &`

(Remarque : le signe '`&`' après une commande signifie que vous lancez la commande « en tâche de fond », c'est-à-dire libérée du terminal dans lequel vous l'avez lancée. Elle ne bloque donc plus votre interpréteur de commande (essayez sans le '`&`' et vous verrez que votre terminal est bloqué jusqu'à ce que `ddd` soit terminé))

Le dévermineur apparaît. Fermez la boîtes de dialogue 'Tip of the day'.

Sélectionnez dans le menu : *View* -> *Data Window*, puis *View* -> *Command tool*.



La fenêtre du dévermineur est alors composée de trois parties : en haut, une zone permettant l'affichage des variables du programme, au milieu une zone où s'affichera le code source du programme, en bas une zone de dialogue avec le dévermineur (`gdb`).

Vous devez voir de plus une mini-fenêtre, que vous pouvez déplacer indépendamment de la grande, et qui contient une série de boutons de commande.

3. Chercher l'erreur

Vous pouvez lancer le programme en cliquant sur **Run** dans la mini-fenêtre de commandes. Vous obtenez un message du genre :

```
 Program received signal SIGFPE, Arithmetic exception.
0xef7770cc in main() at division.cc:14
...
```

Une flèche est apparue devant la ligne concernée dans la zone du milieu du dévermineur. C'est donc là que c'est produit l'erreur.

Si ce n'est pas le cas (configuration de `ddd`), sélectionnez *Status* -> *Backtrace...* dans les menus.
Une boîte de dialogue s'affiche, contenant la liste des appels de fonction en cours. Repérez-y la ligne contenant le nom du fichier que vous déverminez (*divisions.cc*). Elle doit ressembler à :

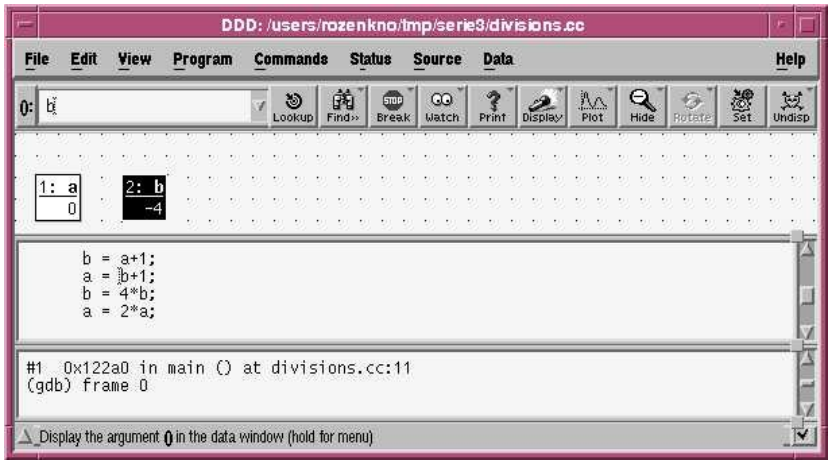
```
#1 0x176d8 in main() at divisions.cc:14
```

Elle indique que le programme s'est interrompu dans la fonction *main* du programme *divisions.cc*, à la ligne 14. Cliquez dessus : la flèche apparaît ! Fermez la boîte de dialogue.

4. Afficher la valeur des variables

Il existe divers moyens d'afficher la valeur des variables :

1. placez le pointeur de la souris sur une variable dans le programme (par exemple sur le *a* de la ligne "*int a, b;*"). Tout en bas de la fenêtre de *ddd*, vous voyez apparaître : `a = 0`. L'information disparaît dès que vous déplacez le pointeur.
2. sélectionnez une variable (par exemple "a") en cliquant dessus avec le bouton droit de la souris, puis sélectionnez *Display a* dans le menu qui apparaît. La variable s'affiche alors dans la première zone du dévermineur, ainsi que sa valeur :



Vous devez pouvoir ainsi observer les valeurs a=0 et b=-4. Ce sont les valeurs des variables au moment où l'erreur a été détectée. La cause de l'erreur devient évidente : la division par a=0.

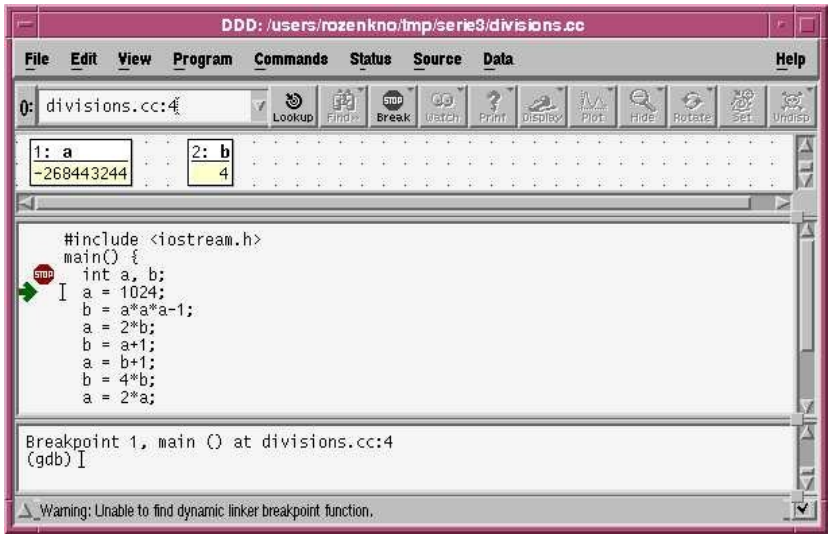
Dans la suite, vous allez exécuter le programme pas-à-pas, pour comprendre à quel moment les résultats des calculs deviennent aberrants. Pour suivre l'évolution des variables pendant l'exécution, commencez par les faire afficher avec la deuxième méthode, si ce n'est pas déjà fait.

5. Exécuter le programme pas-à-pas

Arrêtez le programme en cliquant sur *kill* dans la mini-fenêtre de commandes (lorsqu'une erreur est détectée par le dévermineur, le programme n'est pas arrêté, mais juste en sommeil).

Pour exécuter le programme pas-à-pas, il faut commencer par mettre un point d'arrêt (**breakpoint**) à l'endroit où l'on veut commencer l'observation. Dans cet exemple, on va observer le déroulement du programme depuis le début, c'est-à-dire depuis la première ligne après "`main() {`". Cliquez sur cette ligne avec le bouton droit, et sélectionnez *Set breakpoint* dans le menu qui apparaît. Un point d'arrêt apparaît sur la ligne sélectionnée, symbolisé par un panneau *STOP*.

Lancez le programme par le bouton *Run*. Il s'arrête à la première instruction suivant le point d'arrêt. La flèche dans la zone de programme indique la prochaine ligne qui doit être exécutée :



- Pour exécuter une ligne à la fois, cliquez sur **`next`**, dans la mini-fenêtre de commande.
- Pour continuer le programme jusqu'à la fin, sans s'arrêter à chaque ligne, cliquez sur **`cont`** dans la mini-fenêtre de commande.

Lorsque des variables sont affichées (par la commande *Display()*), leur valeur est mise à jour à chaque pas de l'exécution.

Exécutez le programme pas-à-pas en cliquant sur *next*, et observez l'évolution des valeurs des variables. À quel moment ces valeurs deviennent-elles aberrantes ?

NB : Le but de cet exercice est de vous faire exécuter un programme pas-à-pas en suivant l'évolution des variables, et non de comprendre pourquoi le programme *divisions.cc* se comporte bizarrement.

Voici cependant, à titre documentaire, l'explication succincte de son comportement :

Le programme a un comportement anormal à partir de la ligne

```
a = b+1
```

En effet, à ce moment-là, la valeur de `b` est la plus grande valeur possible pour une variable de type `int`. En effet le type `int` n'est pas un vrai type entier au sens mathématique du terme. Les variables de ce type sont en fait bornées dans l'intervalle `[-MAX_INT - 1, MAX_INT]`.

Pour l'ordinateur, si `b=MAX_INT`, alors `b+1 = -MAX_INT - 1` !!!

Et si `a=-MAX_INT - 1`, alors `2*a = 0` !!!

Bref, dès que l'on dépasse les capacités de représentation, les résultats donnent n'importe quoi du point de vue de l'arithmétique !

Le tout est de le savoir !

6. Parcourir les sources

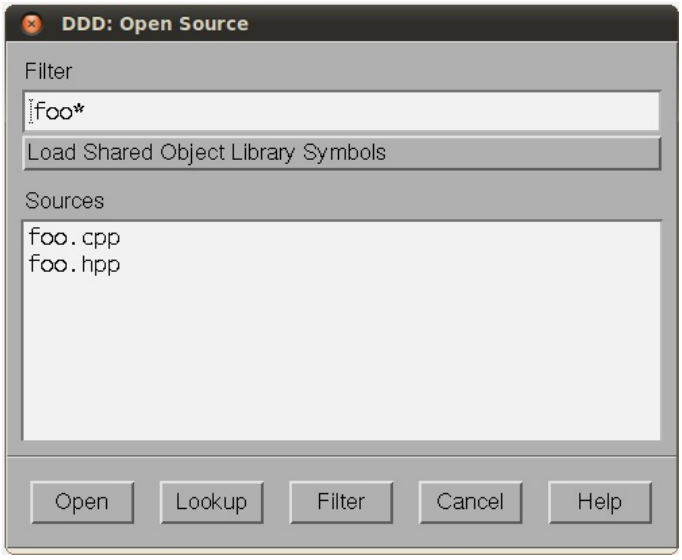
Pour cette sous-section et la suivante, **téléchargez l'exemple fourni** et désarchivez-le dans le dossier de votre choix (depuis le terminal vous pouvez exécuter `unzip dddTest`).

Il s'agit d'un programme constitué de plusieurs fichiers (le but étant de vous montrer comment l'outil de déverminage vous permet de naviguer entre plusieurs fichiers source. Ce que vous serez amenés à pratiquer intensivement au semestre de printemps!)

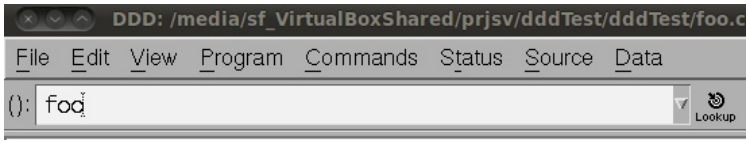
Rendez-vous dans un terminal dans ce dossier et exécutez `make` afin de compiler le programme (ne vous préoccupez pas de cet aspect, nous reviendrons à la compilation séparée au semestre prochain). Vous pouvez à présent lancer le programme avec `./test`. Vous remarquerez que le programme ne fonctionne pas ("plante"). Nous allons voir pourquoi et en profiter pour explorer certaines fonctionnalités de ddd.

Commencez par lancer ddd : `ddd test`.

Initialement, `ddd` s'ouvre sur la fonction `main`. Pour visualiser un autre fichier source, sélectionnez le menu **`File > Open Source...`**. Dans la fenêtre qui s'ouvre alors, il se peut qu'il y ait beaucoup de fichiers listés. Pour n'en afficher que certains, vous pouvez remplir le champ **`Filter`**. Par exemple, pour ne lister que les fichiers dont le nom commence par *Foo*, entrez-y *Foo**, où * sert de joker. Ensuite il suffit de sélectionner le fichier désiré dans la liste et d'appuyer **`Open`**.



Si vous vous intéressez à une fonction en particulier, vous pouvez y accéder directement en introduisant le nom de cette fonction directement dans le champs en dessous des menus puis appuyez sur entrer.



Essayez d'accèder à la fonction `foo` ou à la méthode `Toto::doSomething`.

Le menu **`Source`** de `ddd` offre la possibilité de numéroter les lignes du fichier source.

7. Backtrace

La **backtrace** d'un programme est la liste des fonctions qu'il a exécutées jusqu'à un moment donné, par exemple un crash ou un breakpoint.

A présent, lancez le programme **test** depuis ddd avec **Run**. Le programme devrait alors "planter lamentablement".

Pour visualiser la backtrace au moment du crash, entrez `backtrace` dans la partie inférieure de `ddd`.

```
(gdb) backtrace
#0  0x001b2015 in std::basic_ostream<char, std::char_traits<char> >& std::operator<<
<char, std::char_traits<char>, std::allocator<char> >(std::basic_ostream<char,
std::char_traits<char> >&, std::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&) () from /usr/lib/libstdc++.so.6
#1  0x08048971 in Toto::doSomething (this=0x0) at bar.cpp:22
#2  0x0804899b in failure () at bar.cpp:28
#3  0x08048a6f in foo (crash=true) at foo.cpp:11
#4  0x08048a99 in main () at main.cpp:7
(gdb)
#0  0x001b2015 in std::basic_ostream<char, std::char_traits<char> >& std::operator<< <char, std::char_traits<char>, std::
```

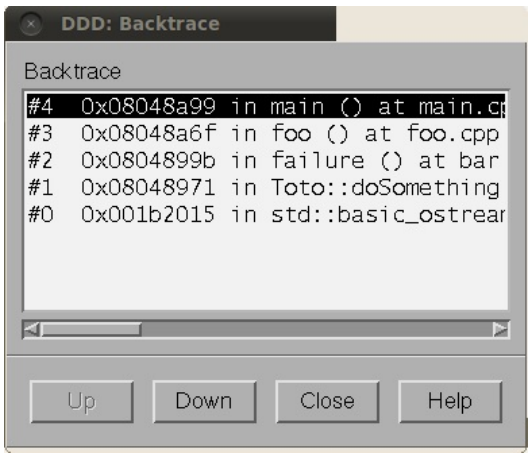
Vous pouvez aussi le faire avec `Status > Backtrace`.

Les fonctions exécutées par le programme sont listées de la plus récente à la plus ancienne avec, pour chaque fonction, le nom du fichier source où elle est implémentée et la

dernière ligne exécutée dans la fonction (par exemple `main.cpp:7`). D'autres informations sont disponibles comme les arguments utilisés pour appeler une certaine fonction (`foo (crash=true)`) ou encore l'adresse de `this` pour les méthodes (`Toto::doSomething (this=0x0)`).

D'après la backtrace, le crash a lieu lors de l'appel de l'opérateur `<<` fourni par la STL. Ce qui est très peu probable. Il faut garder en tête que le crash peut être dû à une erreur en amont dans le code. Pour remonter la trace d'exécution ("backtrace"), utilisez la fonction **Up** de `ddd`. Vous vous retrouvez à présent dans la méthode `Toto::doSomething`. Vous pouvez utiliser **Down** et **Up** pour parcourir la "backtrace".

`ddd` propose aussi une fenêtre pour sauter directement à une certaine fonction de la backtrace sans être obligé d'utiliser 100 fois Up ou Down.



Cette fenêtre est disponible depuis le menu **`Status > Backtrace...`**. Une fois affichée, il vous suffit de sélectionner une fonction pour vous y rendre directement.

Un dernier mot sur l'erreur de segmentation. L'erreur est dans `Toto::doSomething` mais pourtant le code semble valide. Le seul problème est qu'en fait `this` vaut `nullptr` et du coup l'accès à l'attribut `str` est indéterminé (ce qui provoque le crash). Dans la fonction `failure` on remarque que le pointeur `ptr` pointe sur le néant. `ptr->doSomething();` aura pour cette raison un comportement indéterminé (il se peut que le programme ne plante pas). C'est une erreur assez fréquente!

Retour à la série

Dernière mise à jour : $Date: 2014/02/27 16:46:05 $ (`$Revision: 1.3 $`)