

Déverminage (debugging) et exceptions

Buts

Cette série vise deux buts :

1. vous faire pratiquer la gestion des exceptions ;
2. vous entraîner à utiliser le déverminage.

Vous pourrez aussi faire quelques révisions.

Exercice 1 : arithmétique rationnelle (révisions, niveau 1)

Exercice n°40 (page 95 et 276) de l'ouvrage *C++ par la pratique*.

Le but de cet exercice, préambule au suivant qui s'intéressera aux exceptions, est de coder les bases de l'arithmétique des nombres rationnels.

Un nombre rationnel est défini par deux entiers p et q , tels que :

1. $q > 0$
2. p et q sont premiers entre eux.

p représente le numérateur et q le dénominateur

Par exemple : $1/2$: $p=1$, $q=2$; $-3/5$: $p=-3$, $q=5$; 2 : $p=2$, $q=1$.

Quelle type de données utiliser pour représenter les nombres rationnels ?

Écrivez un programme `rationnels.cc`, contenant cette structure de donnée et une fonction `affiche()` permettant d'afficher un tel rationnel.

Testez votre programme avec les 3 rationnels donnés en exemple ci-dessus. Le programme devra afficher :

```
1/2
-3/5
2
```

On veut maintenant implémenter les quatre opérations élémentaires :

l'addition

```
Rationnel addition(Rationnel r1, Rationnel r2);
```

définie par

$$p_1/q_1 + p_2/q_2 = \text{reduction}(p_1 \cdot q_2 + p_2 \cdot q_1 / (q_1 \cdot q_2))$$

où `reduction(p/q)` trouve les nombres p_0 et q_0 tels que $p_0/q_0 = p/q$ et p_0 et q_0 vérifient la définition donnée plus haut (en particulier sont premier entre eux)

Pour la fonction réduction, on utilisera le calcul de PGDC proposé dans l'exercice supplémentaire de la semaine 5.

Exemples :

```
1/2 + -3/5 = -1/10
2 + -3/5 = 7/5
2 + 2 = 4
```

Remarque : toute sophistication de la formule ci-dessus (en particulier si $q_1 = q_2$) peut évidemment être implémentée pour la fonction `addition`.

la soustraction

```
Rationnel soustraction(Rationnel r1, Rationnel r2);
```

définie par

$$p1/q1 - p2/q2 = \text{reduction}(p1*q2 - p2*q1 / (q1*q2))$$

Exemples :

$$1/2 - -3/5 = 11/10$$

$$2 - -3/5 = 13/5$$

$$-3/5 - -3/5 = 0$$

la multiplication

```
Rationnel multiplication(Rationnel r1, Rationnel r2);
```

définie par

$$p1/q1 * p2/q2 = \text{reduction}(p1*p2 / (q1*q2))$$

Exemples :

$$1/2 * -3/5 = -3/10$$

$$2 * -3/5 = -6/5$$

$$-3/5 * -3/5 = 9/25$$

la division

```
Rationnel division(Rationnel r1, Rationnel r2);
```

définie par

$$p1/q1 / (p2/q2) = \text{reduction}(p1*q2 / (q1*p2))$$

si $p2 > 0$, par

$$p1/q1 / (p2/q2) = \text{reduction}((-p1)*q2 / (q1*(-p2)))$$

si $p2 < 0$ et non définie si $p2 = 0$.

Exemples :

$$1/2 / (-3/5) = -5/6$$

$$2 / (-3/5) = -10/3$$

$$-3/5 / (-3/5) = 1$$

Exercice 2 : exceptions (niveau 1)

Exercice n°68 (page 181 et 375) de l'ouvrage *C++ par la pratique*.

Reprendre l'exercice précédent en levant une exception de type `string` (contenant un message d'erreur) dans la fonction `division` si le diviseur est nul ($p2 = 0$).

Testez dans la fonction `main()`.

Exercice 3 : déverminage (niveau 1)

Le but de cet exercice est de vous montrer comment utiliser un dévermineur (« debugger »). 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 du second semestre.**

Cet exercice est proposé en trois variantes, libre à vous de choisir :

- **debugger dans Geany;**
- **gdbgui;**
- **gdb directement à la ligne de commande** (= dans le terminal).

Il existe encore d'autres GUI, voir p.ex. <https://sourceware.org/gdb/wiki/GDB%20Front%20Ends> pour gdb.

Le dévermineur utilisé ici est `gdb`, mais vous pouvez aussi bien utiliser un autre dévermineur, comme par exemple [[lldb](http://lldb.linux.org/) (<http://lldb.linux.org/>) ; les

principes de base restent les mêmes que ceux présentés ici. La correspondance entre les commandes de `gdb` et celles de `lldb` se trouve à l'adresse suivante : <http://lldb.lvm.org/lldb-gdb.html>.

NOTE pour MAC OS : depuis OS X 10.9, Apple est passé à LLVM ; il n'y a donc plus `gdb` de base. Si vous êtes sur Mac, vous avez alors deux options :

1. soit utiliser `lldb` mentionné ci-dessus ;
2. . soit installer `gdb` (via `brew`) **et** le signer ; OS X a un mécanisme de contrôle d'accès aux autres processus qui nécessite un binaire signé (ce qui est nécessaire pour un dévermineur) ; pour signer le binaire `gdb` après son installation, il faut suivre les instructions qu'on peut trouver sur Internet ; par exemple :
 - <http://andresabino.com/2015/04/14/codesign-gdb-on-mac-os-x-yosemite-10-10-2/>
 - <https://www.ics.uci.edu/~pattis/common/handouts/macmingweclipse/allexperimer-gdb-install.html>

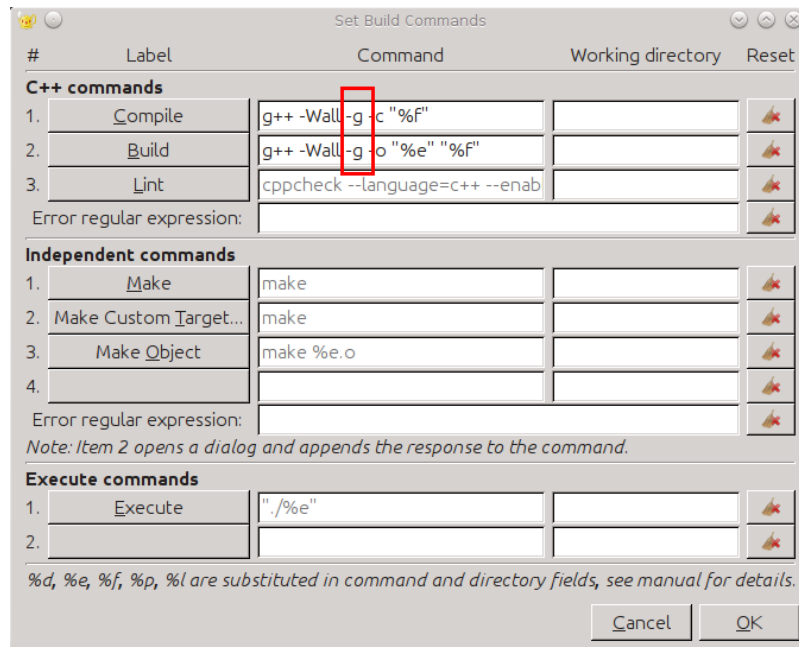
Déverminage avec Geany

- **Attention**, si vous travaillez sur vos propres machines (hors VM fournies), le « debugger » de Geany **ne** fonctionne **pas** :
 - sous Unix avec des versions de Geany antérieures à 1.26, ni sur la 1.36
 - sous Windows < 10
 - sous Windows 10, avec des versions de Geany inférieures à 1.28.
- Pour pouvoir utiliser ce « debugger », il faut avoir coché l'option `Debugger` sous `Tools > Plugin manager`. Si cette option n'apparaît pas, c'est que les « plugins Geany » ne sont pas installés sur votre ordinateur (voir : **les indications pour linux** ou **l'installateur pour Windows**).
- Sous Mac, selon la version du système d'exploitation utilisée, des problèmes de fonctionnement du « debugger » ont été reportés. Les utilisateurs Mac utilisent souvent plutôt le **debugger intégré à Xcode** (qui est d'un mode d'emploi très similaire).

1. Compiler pour le déverminage

Dans `Geany`, ouvrez **le fichier `divisions.cc` fourni** (suivre le lien).

Important : pour pouvoir utiliser un dévermineur sur un programme, il faut le compiler avec l'option `-g`. Dans `Geany`, vous pouvez le faire ici : `Build > Set Build Commands` :



Lancez la compilation (« Construction ») de `divisions.cc` dans Geany (bouton F9). Tout devrait se passer comme d'habitude.

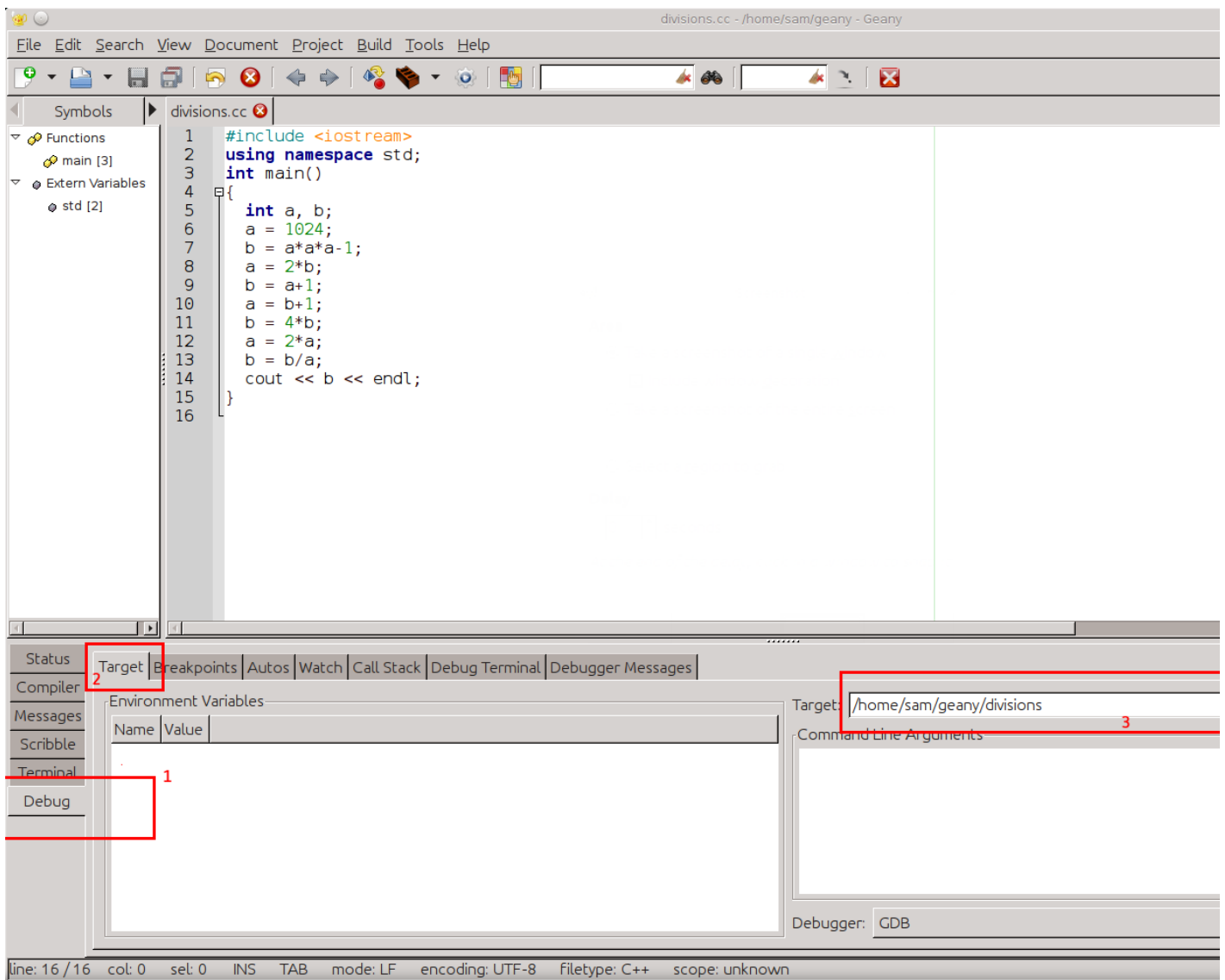
Si vous lancez l'exécution (dans Geany ou dans un terminal), le programme s'arrête avant la fin, et vous obtenez un message d'erreur : « `Floating exception (core dumped)` ».

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

2. Lancer le débogueur

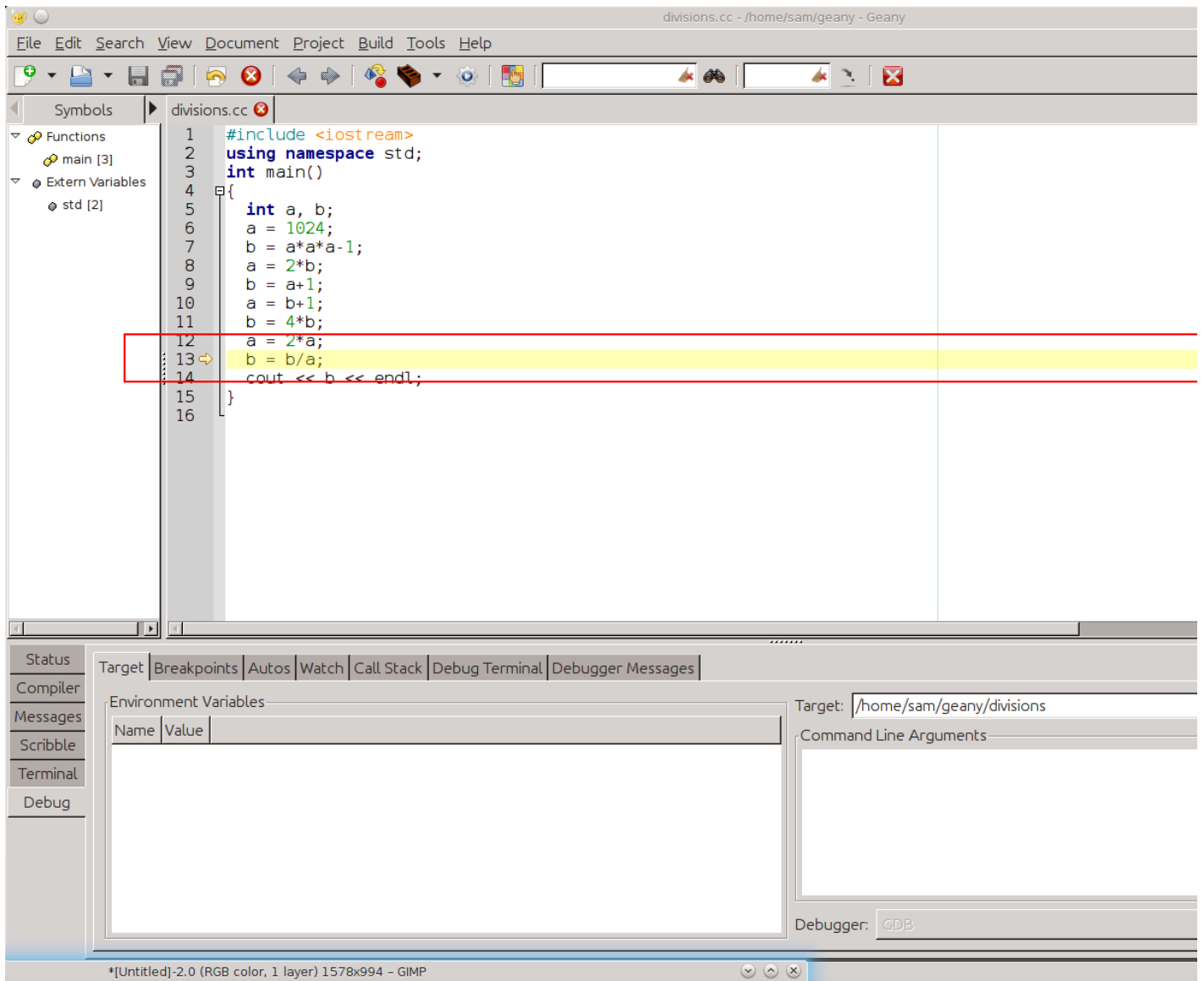
Si le module « Débogueur » n'est pas encore activé dans votre Geany (pas d'onglet « Debug » en bas), allez dans Tools, Plugin Manager et cochez « Debugger ».

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



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».

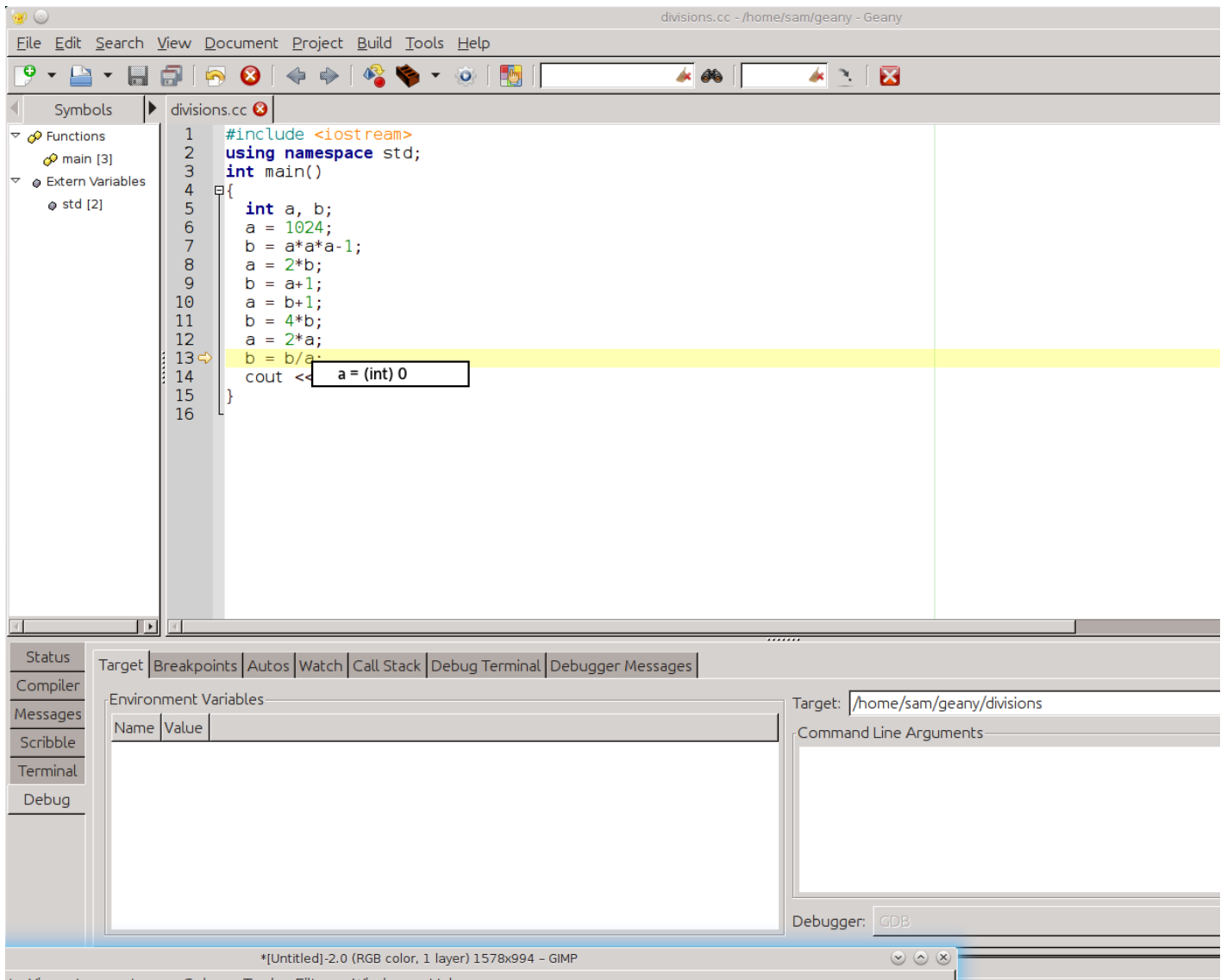
Vous devriez voir s'afficher une fenêtre d'alerte indiquant que le programme s'est terminé avec une erreur. Lorsque vous fermez cette fenêtre vous pouvez voir que la ligne de code ayant provoqué l'erreur est désignée par une flèche dans **Geany** :



3. 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.

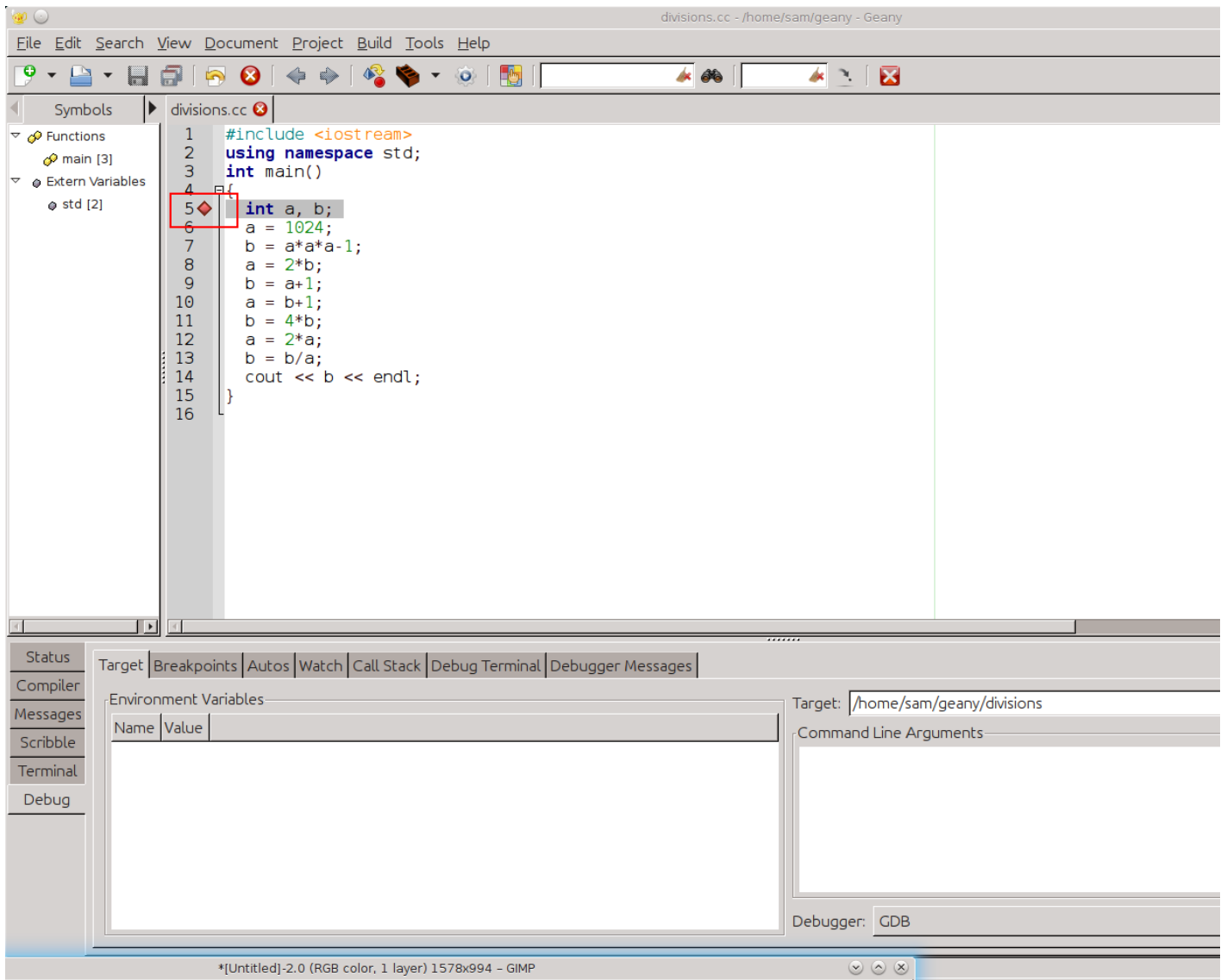
Vous devez pouvoir ainsi observer les valeurs $a=0$ et $b=-4$. Ce sont les valeurs des variables au moment où l'erreur a été détectée. La cause de l'erreur devient évidente : la division par $a=0$.

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

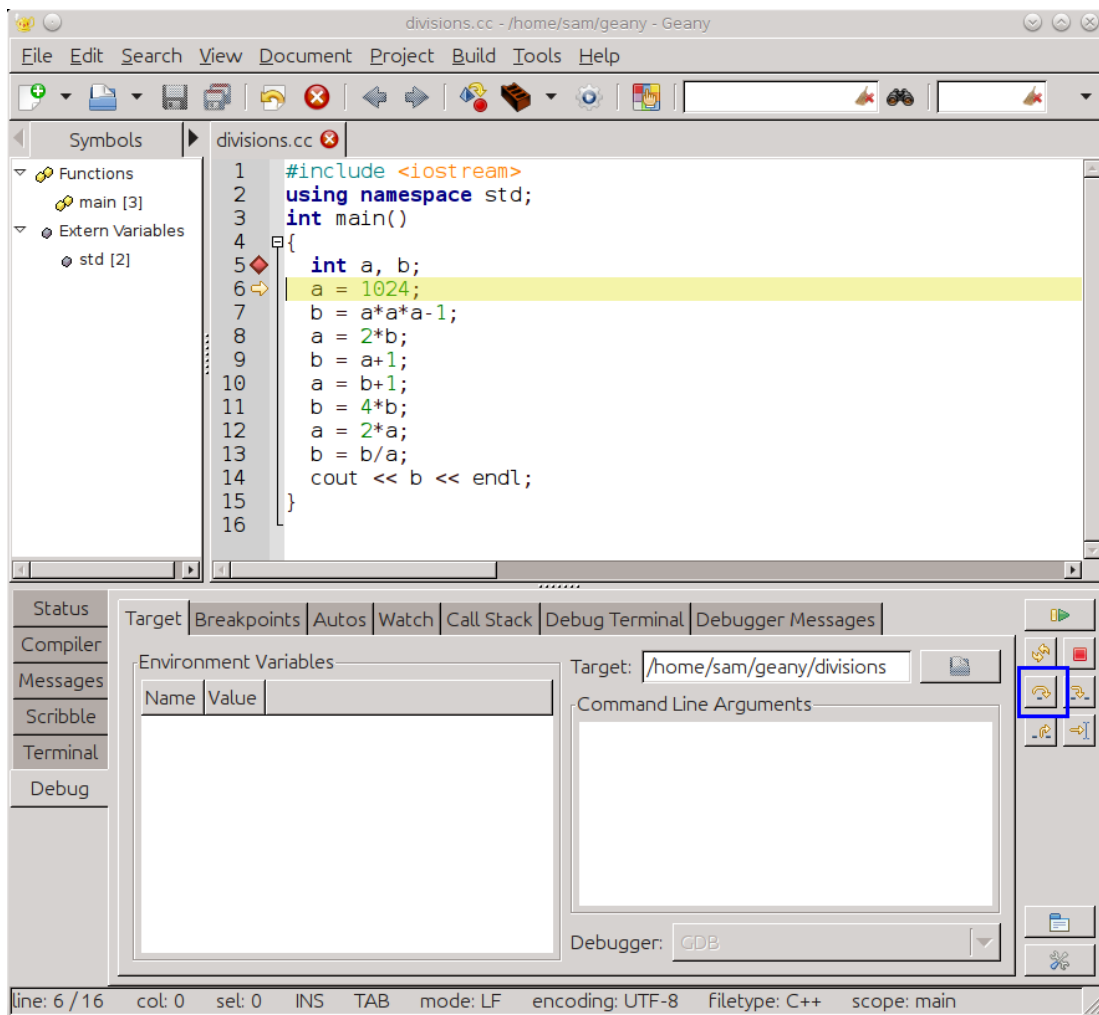
4. Exécuter le programme pas-à-pas

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 de droite où apparaissent les numéros de ligne avec le bouton gauche de la souris. Un point d'arrêt apparaît sur la ligne sélectionnée, symbolisé par petit losange rouge :



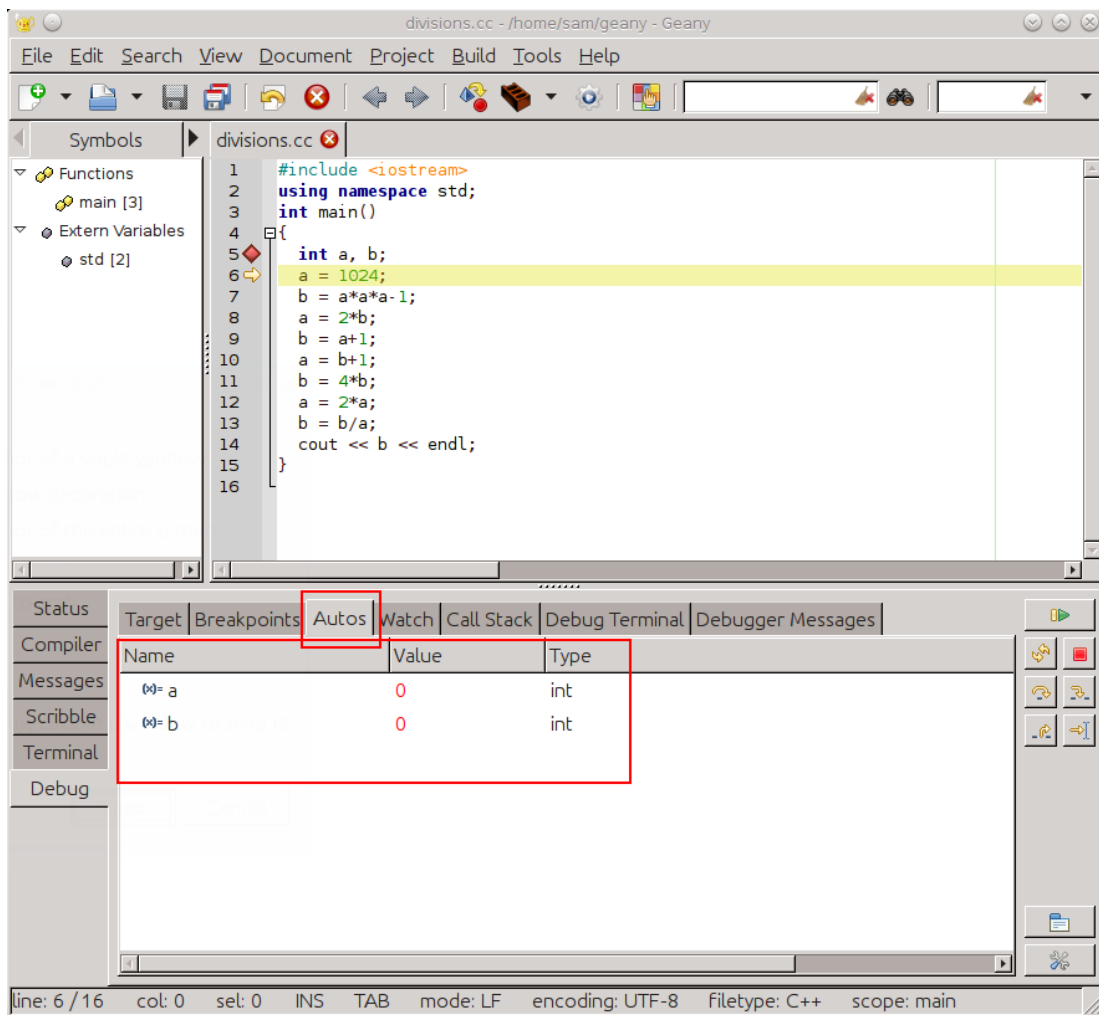
Lancez alors le programme avec la flèche verte. Il s'arrête à la première instruction suivant le point d'arrêt. La flèche dans la zone de programme indique la prochaine ligne qui doit être exécutée :



- 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 `[-numeric_limits<int>::max() - 1, numeric_limits<int>::max()]`.

Pour l'ordinateur, si `b=numeric_limits<int>::max()`, alors `b+1 = -numeric_limits<int>::max() - 1` !!!

Et si `a=-numeric_limits<int>::max() - 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 usuelle !

Le tout est de le savoir ! (cf cours ICC)

5. 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 gdbTest`).

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.)

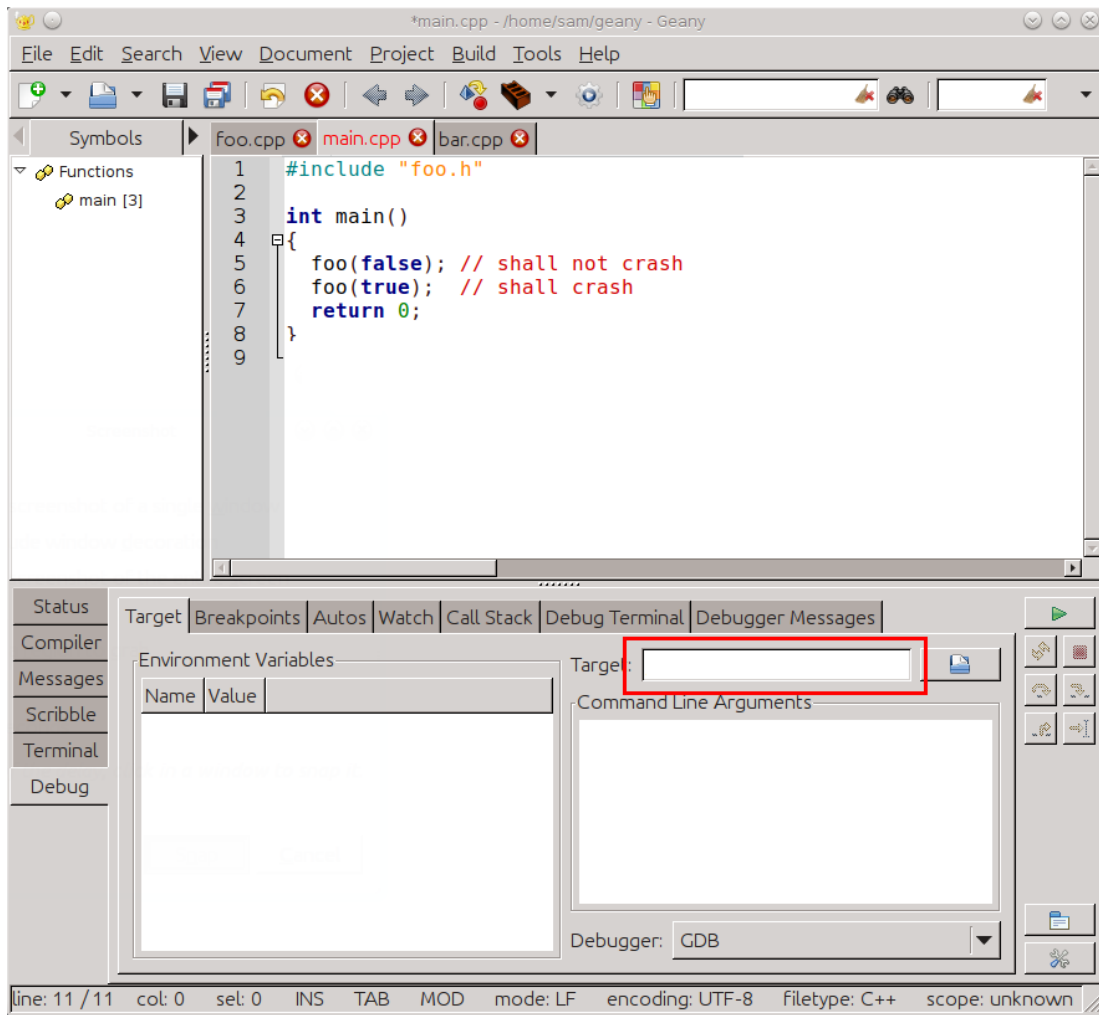
Dans *Geany*, ouvrez le fichier *main.cc* (fourni) et compilez le en utilisant *Make* (dans le menu « Construire »; ou

simplement par « Maj+F9 »). Ne vous préoccupez pas de cet aspect, nous reviendrons à la compilation séparée en temps voulu au début du second semestre.

Lancez ensuite l'exécution.

Vous remarquerez que le programme ne fonctionne pas (« plante »). Nous allons voir pourquoi à l'aide du débogueur.

Spécifiez le nouvel exécutable `main` (qui est dans le répertoire `gdbTest/`) comme nouvelle cible du débogueur via `Debug > Target`:



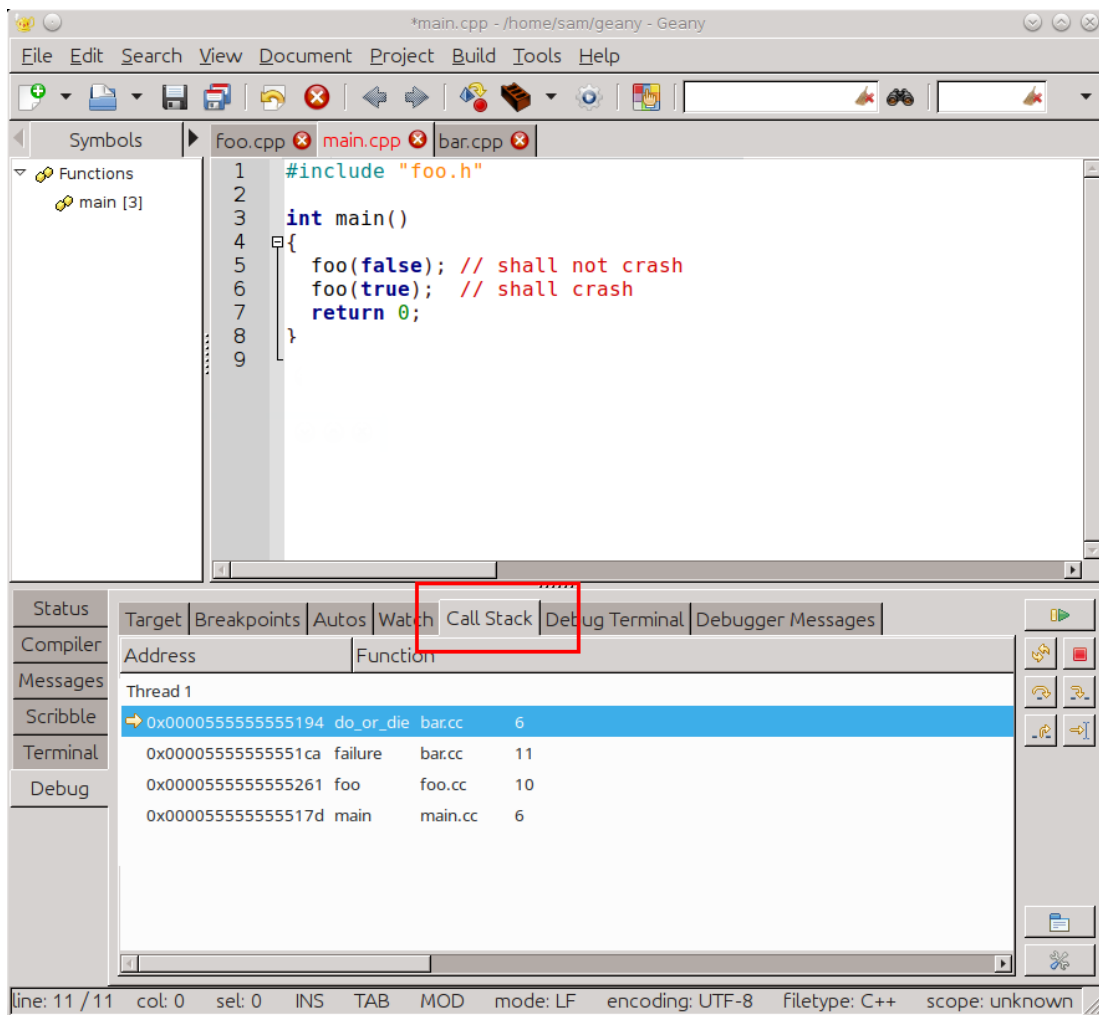
Lancez alors le programme au moyen de la flèche verte, une petite fenêtre s'affiche indiquant qu'une erreur s'est produite. Lorsque l'on ferme cette fenêtre, une flèche indique que l'instruction de la ligne 6 du programme `bar.cc` est fautive.

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

6. Pile d'appels

La **pile d'appels** (*call stack* ou *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.

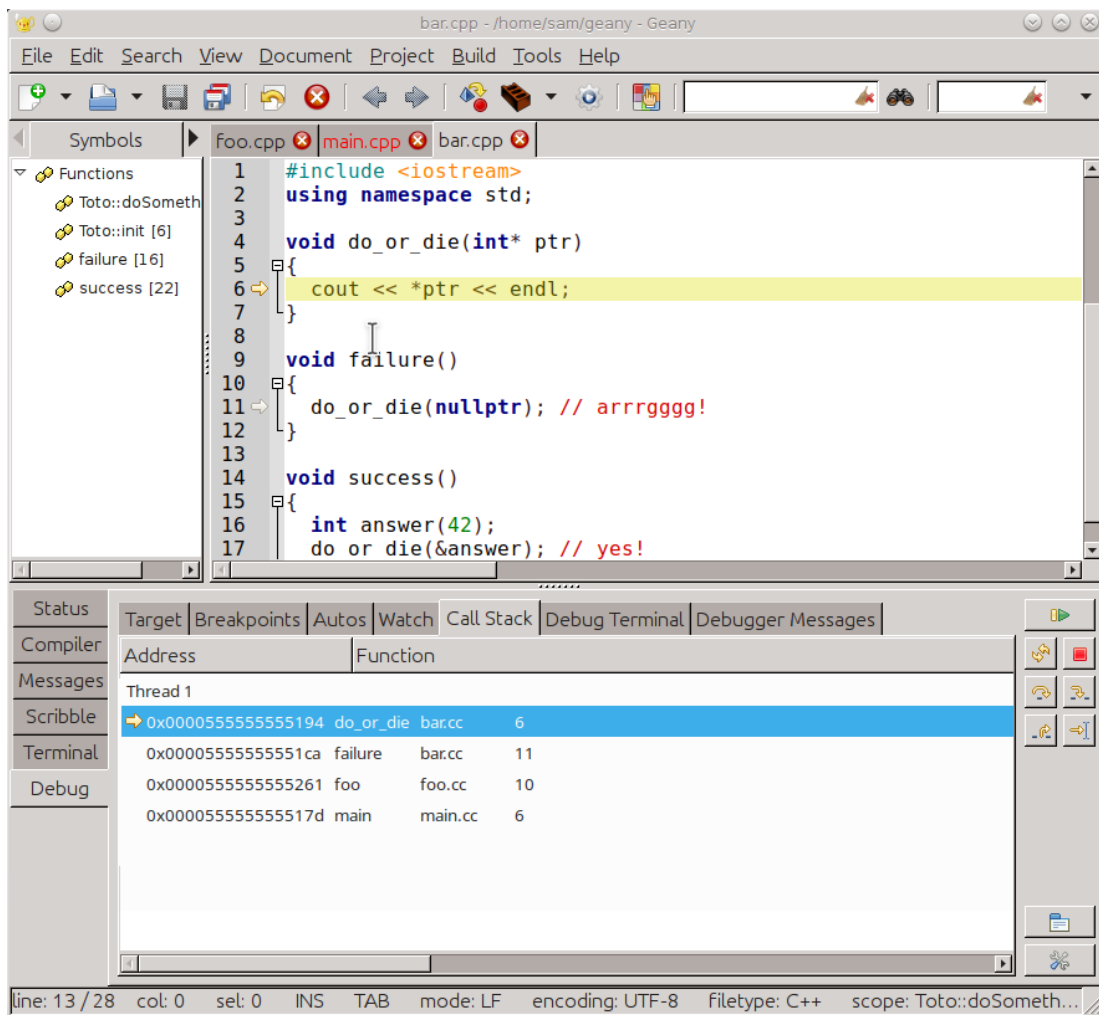
Pour visualiser la pile des appels au moment du crash que nous venons de provoquer, utilisez le bouton `Call Stack`:



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.cc:6`) (déroulez sur la droite la fenêtre contenant la pile des appels pour voir ces informations).

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

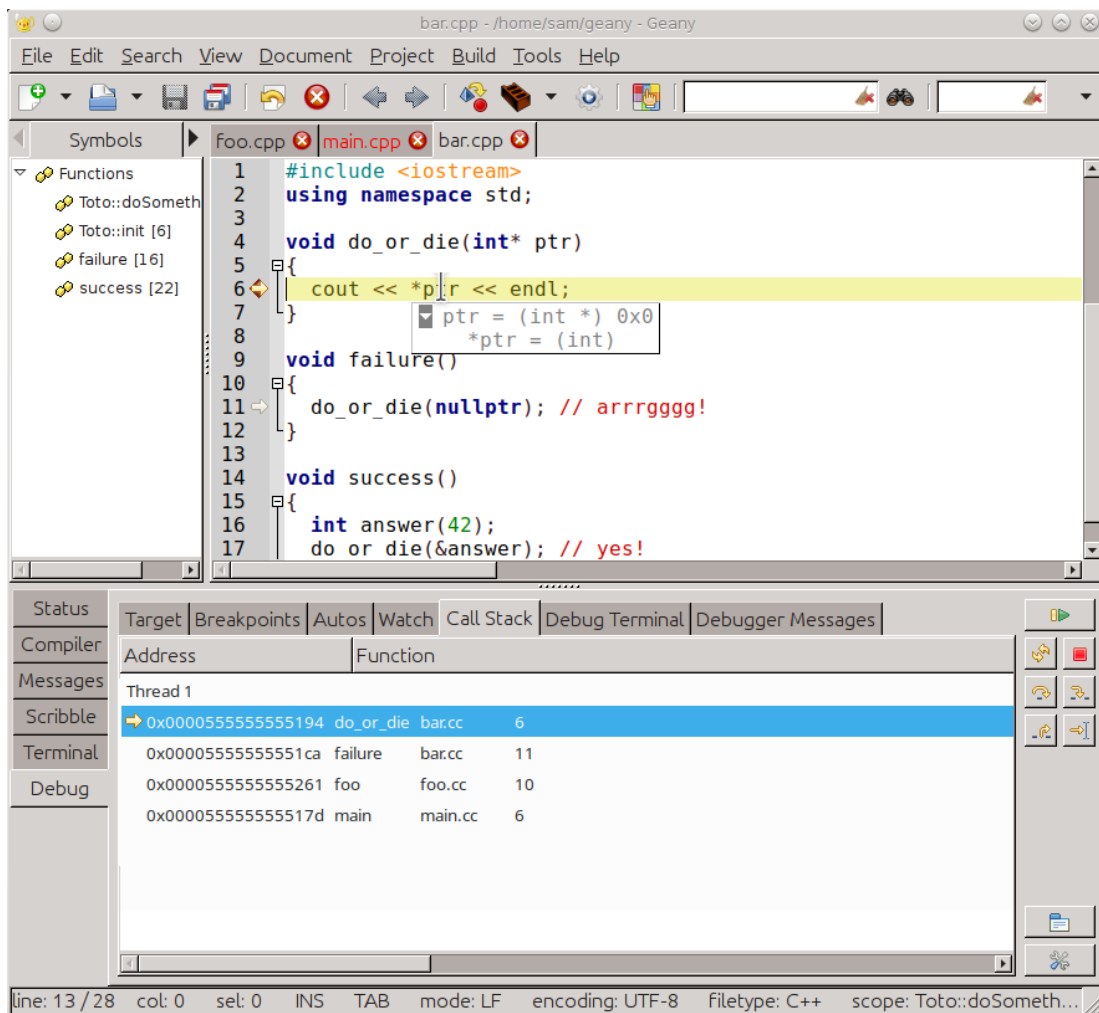
D'après la pile d'appel, la toute dernière instruction provoquant le crash a lieu lors de l'appel de l'opérateur `<<` :



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 (appel avec un pointeur nul), mais supposons que ce soit moins évident. La chose à faire ici serait de :

- placer un point d'arrêt juste avant la source soupçonnée d'erreur (ici au début de la fonction `failure()`) ;
- puis relancer l'exécution au moyen de la flèche verte (si nécessaire, stopper l'exécution courante avec le carré rouge juste en dessous d'elle).

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 malheureusement assez fréquent...

Déverminage avec `gdbgui`

`gdbgui` (site officiel : <https://gdbgui.com/> ; site GitHub (code source) : <https://github.com/cs01/gdbgui/>) est une interface graphique pour `gdb` utilisant un simple navigateur (comme Firefox, par exemple).

Il se télécharge et s'installe simplement depuis [son site](#) (pour peu que vous ayez Python).

1. Compiler pour le déverminage

Dans votre outil de développement usuel, ouvrez [le fichier `divisions.cc` fourni](#) (suivre le lien), puis compilez le.

Important : pour pouvoir utiliser un dévermineur sur un programme, il faut le compiler avec l'option `-g`.

Si vous lancez l'exécution, 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 `gdbgui`, ayez un navigateur ouvert (par exemple Firefox) et allez dans le répertoire où se trouve l'exécutable à déverminer (ou dans n'importe quel répertoire situé au dessus) et lancez la commande `gdbgui` ; puis saisissez le nom de l'exécutable à déverminer (`divisions` pour nous ici) dans la barre du haut (1) et cliquez sur « Load Binary » (2) :

```

1 #include <iostream>
2 using namespace std;
3
4 int f(int x, int y)
5 {
6     return x / y;
7 }
8
9 int main()
10 {
11     int a(1024), b;
12     b = a*a*a - 1;
13     a = 2 * b ;
14     b = a + 1;
15     a = b + 1;
16     b = 4 * b;
17     a = 2 * a;
18     b = f(b, a);
19     cout << b << endl;
20     return 0;
21 }
22
(end of file)

```

Pour lancer l'exécution du programme, cliquez sur la petite flèche en rond, en haut à droite (3). Comme `gdbgui` met par défaut un point d'arrêt dès l'entrée de `main()`, le programme s'arrête de suite. Pour continuer son exécution, cliquez sur la flèche triangle-horizontale juste à droite (4). Vous devriez alors voir s'afficher dans les deux premières fenêtre du bas, un message comme quoi le programme s'est terminé avec une erreur.

Notez que le débogueur vous indique l'endroit où se produit l'erreur. C'est déjà intéressant pour savoir où un programme plante...

En plus `gdbgui` vous indique à droite (« *local variables*») les valeurs des variables locales de l'endroit où il s'est arrêté !

3. 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.

Dans `gdbgui`, c'est très simple :

1. dans la fenêtre de droite, vous pouvez consulter la valeur des variables locales à l'endroit actif (c.-à-d. la profondeur de code correspondant à l'endroit où l'on est dans la pile d'appel : **voir plus bas**);
2. vous pouvez aussi regarder la valeur d'une variable précise en pointant la souris dessus (si tant est que cette variable soit locale à l'endroit actif; ce second moyen est surtout pratique lorsqu'il y a beaucoup de variables):

```

1 #include <iostream>
2 using namespace std;
3
4 int f(int x, int y)
5 {
6     return x / y;
7 }
8
9 int main()
10 {
11     int a(1024), b;
12     b = a*a*a - 1;
13     a = 2 * b ;
14     b = a + 1;
15     a = b + 1;
16     b = 4 * b;
17     a = 2 * a;
18     b = f(b, a);
19     cout << b << endl;
20     return 0;
21 }
22
(end of file)

```

Local variables:

- a 0 int
- b -4 int

4. Exécuter le programme pas-à-pas

Pour demander d'arrêter le programme à un endroit précis, il faut mettre ce que l'on appelle des « point d'arrêt »

« breakpoint ».

Dans `gdbgui`, il y a de toutes façons au départ un breakpoint au début du programme, c'est-à-dire à la première ligne du `main()`. Il est visualisé par le numéro de ligne en fond bleu (11 dans notre exemple).

Relancez le programme depuis le début en cliquant sur la flèche en rond en haut à droite (comme au début). Notez que le débogueur s'arrête **AVANT** l'instruction correspondant au point d'arrêt.

- Pour exécuter une ligne à la fois, appuyez simplement sur la touche « flèche droite » de votre clavier ;
- pour continuer le programme jusqu'à la fin, sans s'arrêter à chaque ligne (mais bon, là il va encore planter), cliquez sur la flèche en haut à droite (comme vous avez fait à l'étape 4 au tout début) ou tapez simplement sur la touche 'c' (de votre clavier).

Exécutez le programme pas-à-pas et observez l'évolution des valeurs des variables (fenêtre de droite).

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 ce programme se comporte bizarrement.

Voici cependant, à titre documentaire, l'explication succincte de son comportement (mais essayez de comprendre par vous-même avant de lire la suite) :

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 `[-numeric_limits<int>::max() - 1, numeric_limits<int>::max()]`.

Pour l'ordinateur, si `b=numeric_limits<int>::max()`, alors `b+1 = -numeric_limits<int>::max() - 1 !!!`

Et si `a=-numeric_limits<int>::max() - 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 usuelle !

Le tout est de le savoir ! (cf cours ICC, leçon I.4)

5. Différence entre « step over » et « step into »

Lorsqu'un point d'arrêt est positionné sur une instruction contenant un appel de fonction, il y a *deux* façon de continuer l'exécution

- soit en restant au même niveau de code, c.-à-d. sans regarder les détails de l'exécution de la fonction ; on appelle cela « step over » ;
- soit en descendant dans la fonction pour y regarder les détails de son exécution de la fonction ; on appelle cela « step into ».

Illustrons cela sur notre programme.

- Commencez par supprimer le breakpoint à la ligne 11 simplement en cliquant dessus (dans le carré bleu); puis ajoutez en un nouveau ligne 18, à nouveau simplement en cliquant sur le numéro de ligne;
- puis relancez l'exécution.

Le programme s'arrête donc juste avant la ligne 18.

- Si vous tapez sur la flèche droite ici (« step over »), l'exécution de `f()` se fera avec pour but du débogueur de passer à la ligne 19; mais bien sûr ici le programme plantera à nouveau. Vous pouvez essayer de refaire cela avec une autre version du programme dans laquelle vous avez modifié la ligne 17 pour ne pas avoir 0 comme valeur de `a`).
- Si par contre vous tapez sur la flèche vers le bas (« step into »), le débogueur va rentrer dans l'exécution de `f()` (sans la commencer) et vous serez donc juste avant que l'erreur ne se produise. Un autre « step over » (ou « step into; ici, ça ne change rien puisqu'il n'y a pas d'appel de fonction ligne 6) de plus provoquera l'erreur.

6. Programme avec plusieurs 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 gdbTest`).

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

Dans le terminal, allez dans le répertoire `gdbTest/` et compilez le programme en utilisant la commande `make`. Ne vous préoccupez pas de cet aspect, nous reviendrons à la compilation séparée en temps voulu au début du second semestre.

Lancez ensuite l'exécution en tapant

```
./main
```

Vous remarquerez que le programme ne fonctionne pas (« Segmentation Fault »). Nous allons voir pourquoi à l'aide du débogueur.

Spécifiez le nouvel exécutable `main` (qui est dans le répertoire `gdbTest/`, donc : « `gdbTest/main` ») comme nouvelle cible du débogueur.

Puis lancez l'exécution du programme.

Vous devriez voir s'afficher un message comme quoi le programme s'est terminé avec une erreur à l'instruction de la ligne 6 du programme `bar.cc`. (Le nom du programme courant est affiché juste au dessus de la fenêtre de code, à droite du petit cadre blanc « jump to line ».)

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

7. Pile d'appels

La **pile d'appels** (*call stack* ou *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.

Cette pile des appels est visualisée dans l'onglet « threads » à droite :

func	file	ac
do_or_die	bar.cc:6	0>
failure	bar.cc:11	0>
foo	foo.cc:10	0>
main	main.cc:6	0>

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.cc:6`).

Pour vous déplacer dans la pile d'appels, il suffit de cliquer dans ce tableau sur le niveau d'appel désiré (le niveau où l'on est (= dont le code source est présenté à gauche) est en gras).

D'après la pile d'appel, la toute dernière instruction provoquant le crash a lieu lors de l'appel de l'opérateur `<<` (ligne 6 de `bar.cc`).

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 (appel avec un pointeur nul), mais supposons que ce soit moins évident. La chose à faire ici serait de :

- placer un point d'arrêt juste avant la source soupçonnée d'erreur (ici au début de la fonction `failure()`);
- puis relancer l'exécution.

Dans `do_or_die()` (y retourner), l'examen du contenu de la variable `ptr` 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 malheureusement assez fréquent...

Déverminage dans le terminal avec `gdb`

1. Compiler pour le déverminage

Dans votre outil de développement usuel, ouvrez **le fichier `divisions.cc` fourni** (suivre le lien), puis compilez le.

Important : pour pouvoir utiliser un dévermineur sur un programme, il faut le compiler avec l'option `-g`.

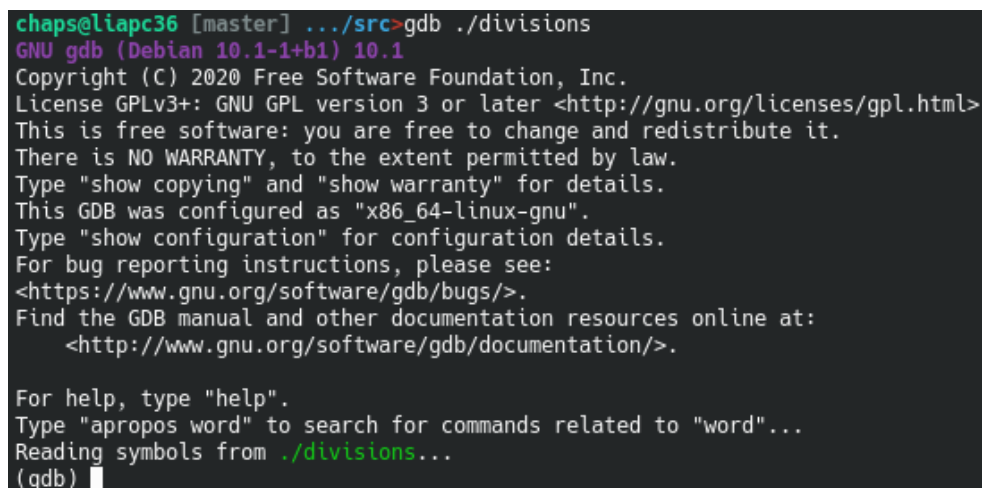
Si vous lancez l'exécution, le programme s'arrête avant la fin, et vous obtenez un message d'erreur : « `Floating exception (core dumped)` ».

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 `gdb` dans le terminal, allez dans le répertoire où se trouve l'exécutable à déverminer et lancez la commande `gdb` avec comme paramètre le nom de l'exécutable à déverminer :

```
gdb ./division
```



```
chaps@liapc36 [master] ../src>gdb ./divisions
GNU gdb (Debian 10.1-1+b1) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./divisions...
(gdb) █
```

Pour voir le code source, tapez (dans `gdb`) :

```
layout src
```

Pour lancer l'exécution du programme, tapez (dans `gdb`) :

```
run
```

Vous devriez voir s'afficher un message comme quoi le programme s'est terminé avec une erreur :

```
divisions.cc
5      {
6      >6      return x / y;
7      }
8
9      int main()
10     {
11         int a(1024), b;
12         b = a*a*a - 1;
13         a = 2 * b ;
14         b = a + 1;
15         a = b + 1;
16         b = 4 * b;
17         a = 2 * a;
18         b = f(b, a);
19         cout << b << endl;
20         return 0;
21     }
```

```
native process 1318816 In: f
(gdb) run
Starting program: /shared/home/chaps/Work/cours/Informatique/www/series/src/divisions
Program received signal SIGFPE, Arithmetic exception.
0x0000055555555173 in f (x=-4, y=0) at divisions.cc:6
(gdb) █
```

Notez que le débogueur vous indique l'endroit où se produit l'erreur. C'est déjà intéressant pour savoir où un programme plante...

3. 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 `x` et `y` à l'endroit de l'arrêt en tapant (dans gdb) :

```
print x
print y
```

A noter que la commande `print` peut s'abrégier tout simplement `p` :

```
p {x,y}
```

Vous devez pouvoir ainsi observer les valeurs `y=0` et `x=-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 `y=0`.

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

4. Exécuter le programme pas-à-pas

Pour demander d'arrêter le programme à un endroit précis, il faut mettre ce que l'on appelle des « point d'arrêt » « breakpoint » en utilisant la commande (gdb) `break`. On peut soit `break` à une ligne donnée, soit à une fonction donnée.

Commençons, par exemple, à vouloir observer le déroulement du programme depuis le début, c'est-à-dire depuis la première ligne du "`main()`".

Pour cela, on peut *soit* taper (attention, **NE faites PAS** les deux !)

```
break main
```

soit taper (attention, **NE faites PAS** les deux !)

```
break 11
```

puisque ici, pour nous dans ce programme, la ligne 11 est la première ligne du `main()`

Ceci (l'un ou l'autre) fait, le point d'arrêt apparaît à droite de la ligne sélectionnée, symbolisé par « b+ ».

Relancez alors le programme avec la commande `run` (et répondez 'y'). Le dévermineur s'arrête **AVANT** l'instruction correspondant au point d'arrêt.

- Pour exécuter une ligne à la fois, entrez la commande `next` (ou simplement 'n');
- pour continuer le programme jusqu'à la fin, sans s'arrêter à chaque ligne (mais bon, là il va encore planter), entrez la commande `cont`.

Exécutez le programme pas-à-pas en

1. demandant d'afficher systématiquement les valeurs de `a` et `b` :

```
disp {a,b}
```

2. entrant une fois 'n' (pour `next`),
3. puis en tapant sur « Entrée » (touche « Return ») pour répéter plusieurs fois (« Entrée » tout seul répète simplement la dernière commande qui, ici, est `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 ce programme se comporte bizarrement.

Voici cependant, à titre documentaire, l'explication succincte de son comportement (mais essayez de comprendre par vous-même avant de lire la suite) :

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 `[-numeric_limits<int>::max() - 1, numeric_limits<int>::max()]`.

Pour l'ordinateur, si `b=numeric_limits<int>::max()`, alors `b+1 = -numeric_limits<int>::max() - 1 !!!`

Et si `a=-numeric_limits<int>::max() - 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 usuelle !

Le tout est de le savoir ! (cf cours ICC, leçon I.4)

5. Différence entre `next` et `step`

Lorsqu'un point d'arrêt est positionné sur une instruction contenant un appel de fonction, il y a *deux* façon de continuer l'exécution

- soit en restant au même niveau de code, c.-à-d. sans regarder les détails de l'exécution de la fonction ; on appelle cela « step over » et cela se fait avec la commande `next` ;
- soit en descendant dans la fonction pour y regarder les détails de son exécution de la fonction ; on appelle cela « step intp » et cela se fait avec la commande `step`.

Illustrons cela sur notre programme.

- Commencez par supprimer notre breakpoint précédent : `clear 11`
- puis ajoutez en un nouveau ligne 18 : `break 18`
- Relancez l'exécution : `run` (puis répondez 'y' si nécessaire)

Le programme s'arrête donc juste avant la ligne 18.

- Si vous tapez `next` ici, l'exécution de `f()` se fera avec pour but du dévermineur de passer à la ligne 19; mais bien sûr le programme plantera à nouveau. Vous pouvez essayer de refaire cela avec une autre version du programme dans laquelle vous avez modifié la ligne 17 pour ne pas avoir 0 comme valeur de `a`).
- Si vous tapez `step` ici, le dévermineur va rentrer dans l'exécution de `f()` (sans la commencer) et vous serez donc juste avant que l'erreur ne se produise. Un autre `step` (ou `next`; ici, ça ne change rien puisqu'il n'y a pas d'appel de fonction ligne 6) de plus provoquera l'erreur.

Pour quitter `gdb`, tapez simplement 'q' (pour la commande `quit`).

6. Programme avec plusieurs 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 gdbTest`).

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

Dans le terminal, allez dans le répertoire `gdbTest/` et compilez le programme en utilisant la commande `make`. Ne vous préoccupez pas de cet aspect, nous reviendrons à la compilation séparée en temps voulu au début du second semestre.

Lancez ensuite l'exécution en tapant

```
./main
```

Vous remarquerez que le programme ne fonctionne pas (« Segmentation Fault »). Nous allons voir pourquoi à l'aide du débogueur.

Spécifiez le nouvel exécutable `main` (qui est dans le répertoire `gdbTest/`) comme nouvelle cible du débogueur en lançant la commande `gdb` avec comme paramètre le nom de l'exécutable à déboguer :

```
gdb main
```

Pour voir le code source, tapez (dans `gdb`) :

```
layout src
```

Pour lancer l'exécution du programme, tapez (dans `gdb`) :

```
run
```

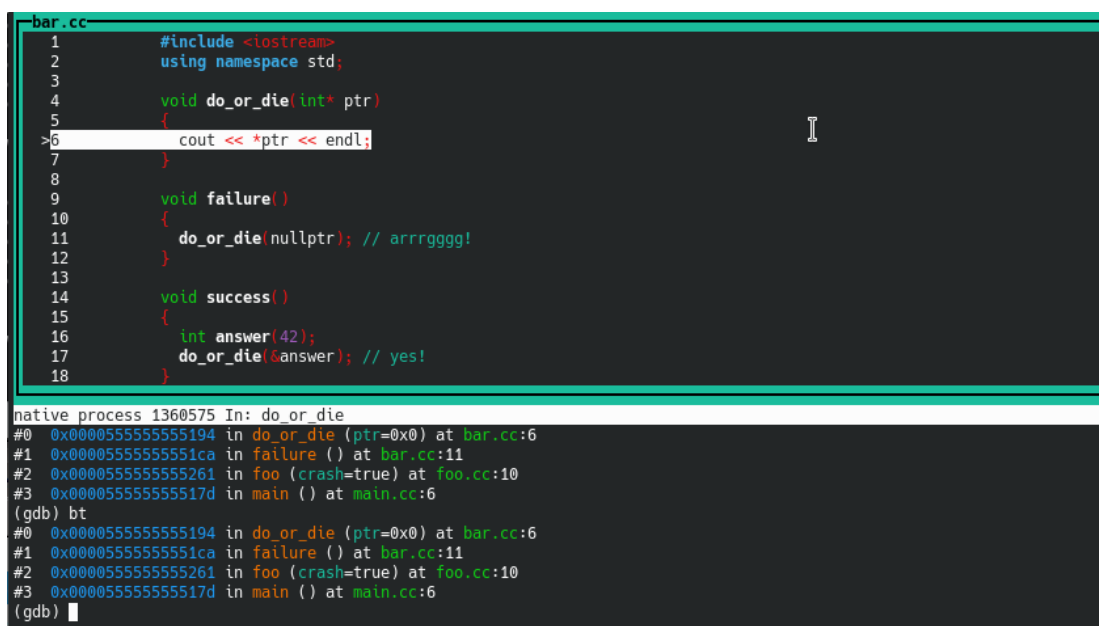
Vous devriez voir s'afficher un message comme quoi le programme s'est terminé avec une erreur à l'instruction de la ligne 6 du programme `bar.cc`.

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

7. Pile d'appels

La **pile d'appels** (*call stack* ou *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 pile des appels au moment du crash que nous venons de provoquer, utilisez la commande `bt` (comme *backtrace*), ou `where`:



```
bar.cc
1  #include <iostream>
2  using namespace std;
3
4  void do_or_die(int* ptr)
5  {
>6  cout << *ptr << endl;
7  }
8
9  void failure()
10 {
11     do_or_die(nullptr); // arrrgggg!
12 }
13
14 void success()
15 {
16     int answer(42);
17     do_or_die(&answer); // yes!
18 }
```

```
native process 1360575 In: do_or_die
#0  0x000055555555194 in do_or_die (ptr=0x0) at bar.cc:6
#1  0x0000555555551ca in failure () at bar.cc:11
#2  0x000055555555261 in foo (crash=true) at foo.cc:10
#3  0x00005555555517d in main () at main.cc:6
(gdb) bt
#0  0x000055555555194 in do_or_die (ptr=0x0) at bar.cc:6
#1  0x0000555555551ca in failure () at bar.cc:11
#2  0x000055555555261 in foo (crash=true) at foo.cc:10
#3  0x00005555555517d in main () at main.cc:6
(gdb)
```

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.cc:6`).

Pour vous déplacer dans la pile d'appels, utilisez les commandes `up` et `down` (ou simplement `do`).

D'après la pile d'appel, la toute dernière instruction provoquant le crash a lieu lors de l'appel de l'opérateur `<<`.

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 (appel avec un pointeur nul), mais supposons que ce soit moins évident. La chose à faire ici serait de :

- placer un point d'arrêt juste avant la source soupçonnée d'erreur (ici au début de la fonction `failure()`) ;
- puis relancer l'exécution.

Dans `do_or_die()` (y retourner avec `down`), l'examen du contenu de la variable `ptr` vous montrera alors le pointeur nul:

```
down
print ptr
```

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 malheureusement assez fréquent...

Exercice 4 : Compression RLE (révisions + exceptions, niveau 2)

Exercice n°69 (page 181 et 376) de l'ouvrage [C++ par la pratique](#).

4.1 Introduction

La compression RLE (*Run Length Encoding*) est une ancienne technique de compression, utilisée en particulier dans le codage des images. Le principe très simple de cette technique est le suivant: toute séquence de caractères identiques, et de longueur L (assez grande) est codée par :

« c $flag$ L » où $flag$ est un caractère spécial, si possible peu fréquent dans les données à compresser

les autres séquences n'étant pas modifiées.

Lorsque le caractère spécial $flag$ est rencontré dans la séquence d'origine, il est codé par la séquence spéciale « $flag$ 0 » (c.-à-d. pas de caractère c , et $L = 0$).

4.2 Énoncé

Dans le fichier `rle.cc`:

1. prototypez et définissez la fonction :

```
string compresse(string_view data, char flag);
```

de sorte qu'elle retourne la chaîne correspondant à la compression par l'algorithme RLE de la séquence `data` donnée en argument, en utilisant `flag` comme caractère spécial.

La longueur 'assez grande' pour les séquences de caractères identiques à compresser est de 3.

Comme nous codons la longueur sous forme de un caractère, la longueur **maximale** d'une séquence compressée est donc 9.

Dans le cas d'une séquence de plus de 9 caractères identiques, les 9 premiers seront codés de manières compressées puis la suite de la séquence sera considérée comme une nouvelle séquence à coder. [c.f. exemple de fonctionnement].

2. Prototypez et définissez la fonction :

```
string decompresse(string_view rldata, char flag);
```

de sorte qu'elle retourne la chaîne `rldata` sous sa forme décompressée.

3. Utilisez des exceptions pour les cas d'erreurs possibles dans la fonctions de décompression.

En cas d'erreur dans la chaîne à décoder, l'exception renvoyée devra contenir :

1. un message d'erreur explicitant l'erreur qui s'est produite [cf l'exemple de fonctionnement]
2. le message déjà décodé (c.-à-d. avant l'erreur)
3. le reste du message à décoder (c.-à-d. après l'endroit où l'erreur s'est produite).

Note : Pour extraire la sous-chaîne de longueur `L` à partir de la position `p` d'une chaîne `s`, faire `s.substr(p,L)`.

Pour extraire la sous-chaîne finale (c.-à-d. jusqu'au bout) à partir de la position `p` d'une chaîne `s`, faire `s.substr(p)`.

4. **Optionnel** : proposez (en commentaire à la fin du fichier) une amélioration de l'algorithme visant à mieux gérer
- i. les cas où les séquences de caractères identiques sont excessivement longues,
 - ii. où le caractère spécial est présent fréquemment (et de manière groupée) dans les données à compresser.

4.3 Exemples de fonctionnement (*flag = '#'*)

```
Entrez les données à compresser : #aaaaa
Forme compressée: #0a#5
[ratio = 83.3333%]
test décompression #0a#5 : ok!
Entrez les données à décompresser : #0a#3
décompressé : #aaa
```

```
Entrez les données : aa#4baaaabb#dddddddddd###aaaaaaaaaaaaaaaa
Forme compressée : aa#04ba#4bb#0d#9dd#0#0#0a#9a#6
[ratio = 73.17%]
test décompression aa#04ba#4bb#0d#9dd#0#0#0a#9a#6 : ok!
Entrez les données à décompresser : aa#04ba##4bb#0d
Erreur de décompression : caractère # incorrect après le flag #
décodé à ce stade : aa#4ba
non décodé : ##4bb#0d
```

Note : pour transformer un caractère représentant un chiffre (p.ex. '3') en sa valeur entière (p.ex. 3), il suffit de lui soustraire le caractère '0' : `c - '0'`.