Série 12 : Programmation *C* - Debugging and profiling.

Buts

Le but de cette série d'exercices est de vous apprendre à utiliser un debugger, mais aussi à «optimiser» votre code.

Les exercices 3 et suivants ont pour but de vous faire pratiquer la programmation C. Si un point vu dans les séances précédentes ne vous semble pas clair, je vous conseille plutôt de le reprendre et faire les exercices qui lui sont relatifs plutôt que les exercices 3 et suivants de cette série. Les exercices 1 et 2 abordent par contre les nouveaux thèmes de cette semaine.

Rappel

Avez-vous pris connaissance des **conseils relatifs à ces séries d'exercices** ?

Exercice 1 : déverminage (niveau 1)

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 par ailleurs d'**autres GUI pour gdb, voir p.ex. https://sourceware.org/gdb/wiki/GDB%20Front%20Ends** .

Le déverminer utilisé ici est `gdb`, mais vous pouvez aussi bien utiliser un *autre* déverminer, comme par exemple [**`lldb` (http://lldb.llvm.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.llvm.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 déverminer) ; 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/allexperimental/mac-gdb-install.html**

A - Déverminage avec `Geany`

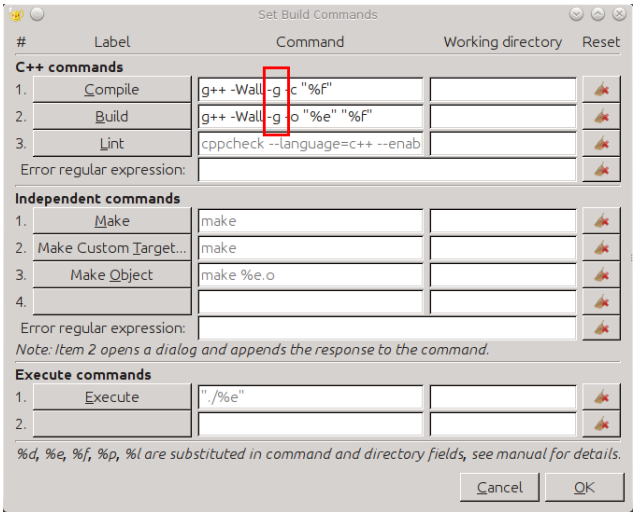
- **Attention,** si vous travaillez sur vos propres machines (hors VM fournies), le « debugger » de Geany **ne** fonctionne **pas** :
  - sous Unix avec des versions de Geany antérieures à 1.26, ni sur la 1.36
  - sous Windows < 10
  - sous Windows 10, avec des versions de Geany inférieures à 1.28.

- 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).

A.1 Compiler pour le déverminage

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

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



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

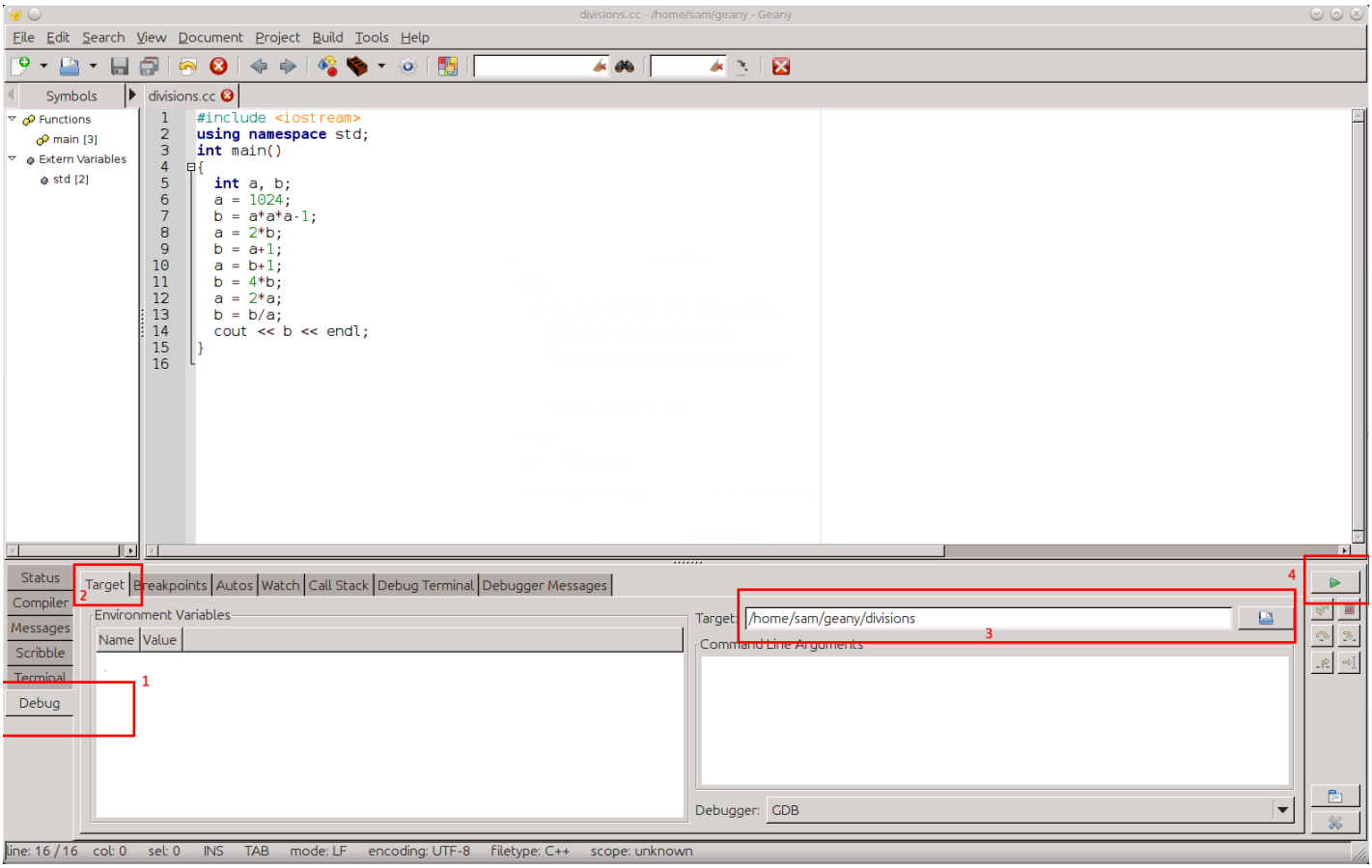
Si vous lancez l'exécution (dans Geany ou dans un terminal), le programme s'arrête avant la fin, et vous obtenez un message d'erreur : « `Floating exception (core dumped)` ».

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

A.2. Lancer le dévermineur

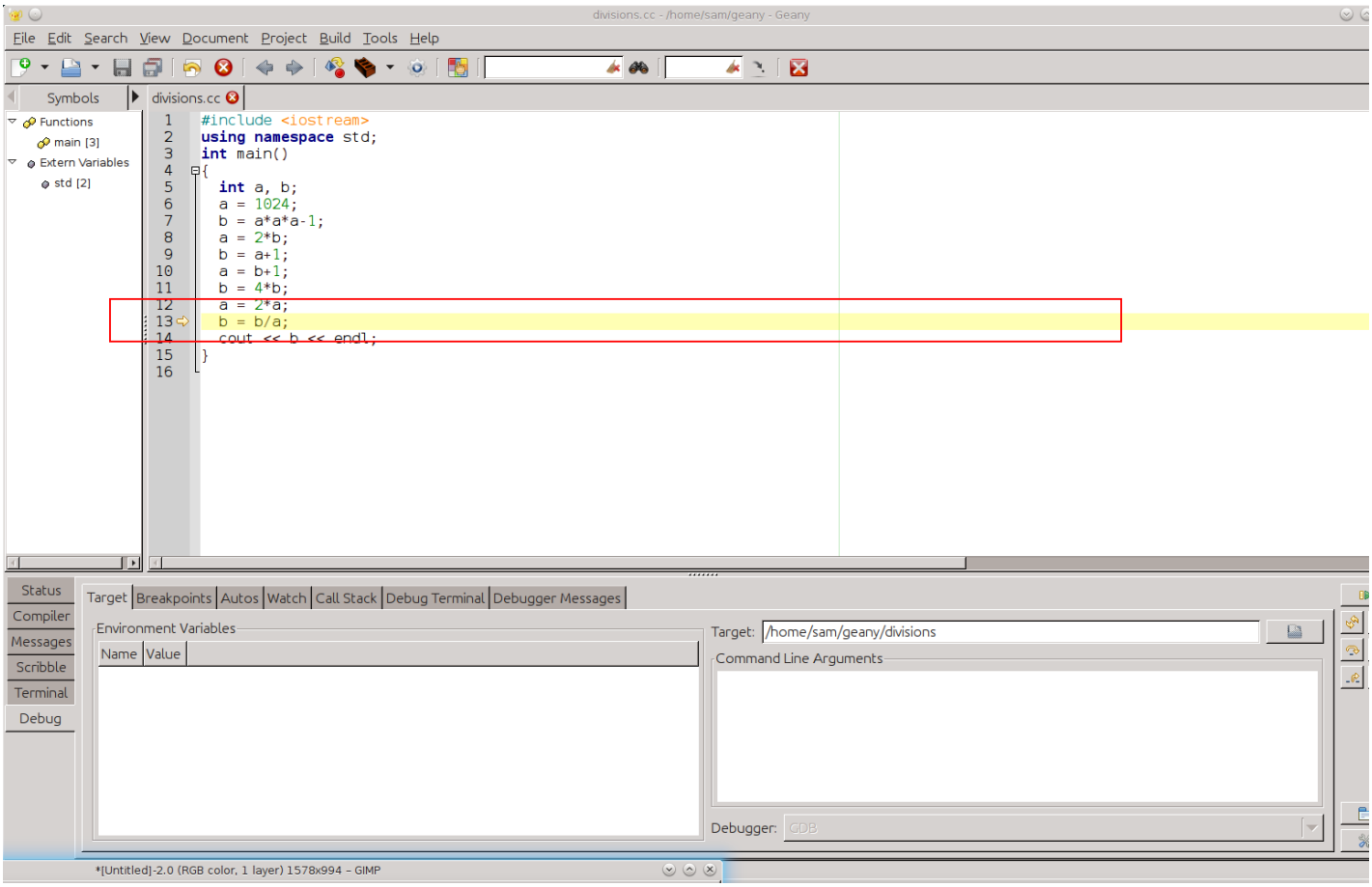
Si le module « Debbuger » n'est pas encore activé dans votre Geany (pas d'onglet «Debug» en bas), allez dans Tools, Plugin Manager et cochez «Debugger».

Pour lancer l'exécution du programme au moyen du 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.c`, mais sans l'extension `.c`) ;
4. puis cliquez sur la petite flèche verte en haut à droite de la fenêtre de «debugging».

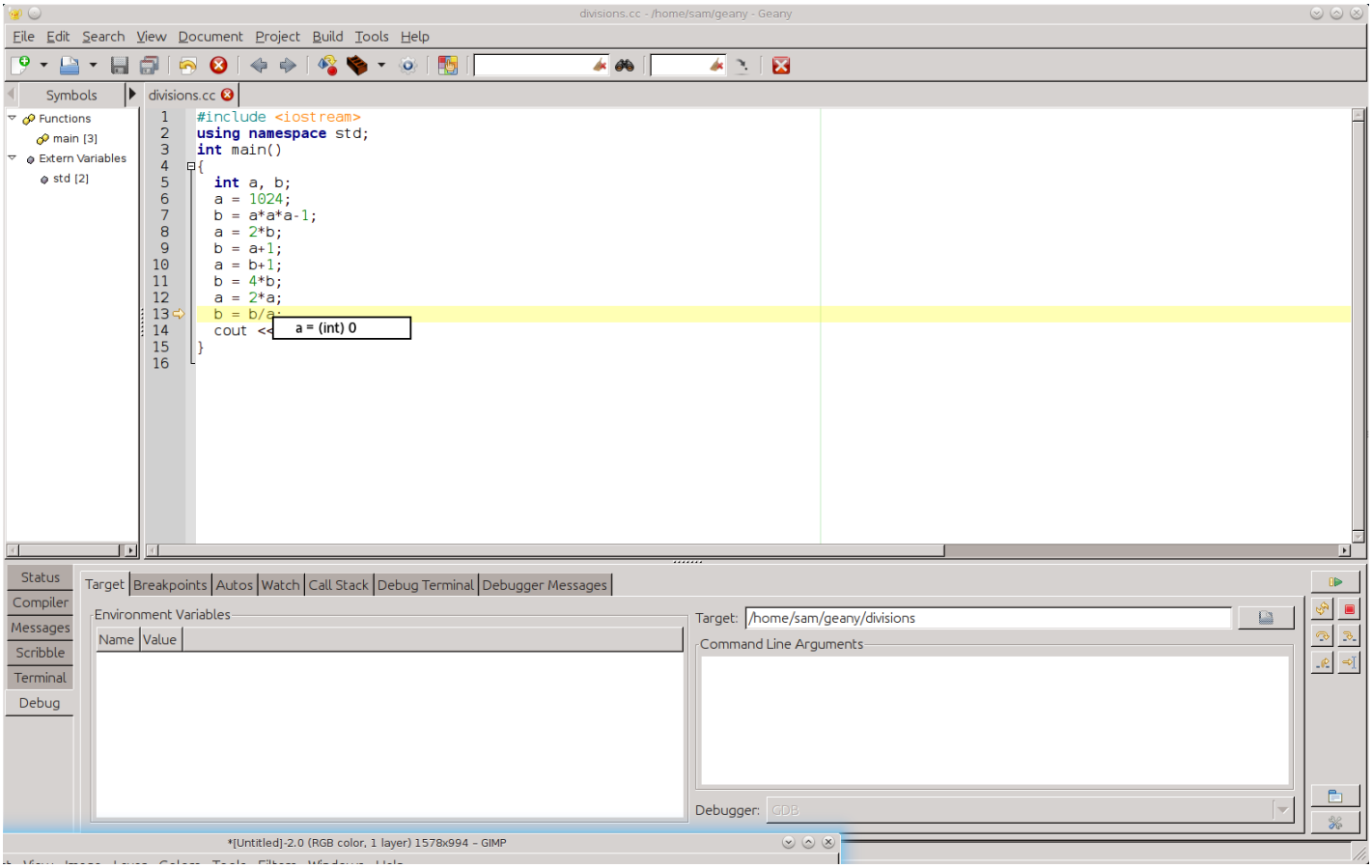
Vous devriez voir s'afficher une fenêtre d'alerte indiquant que le programme s'est terminé avec une erreur. Lorsque vous fermez cette fenêtre vous pouvez voir que la ligne de code ayant provoqué l'erreur est désignée par une flèche dans `Geany` :



A.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.

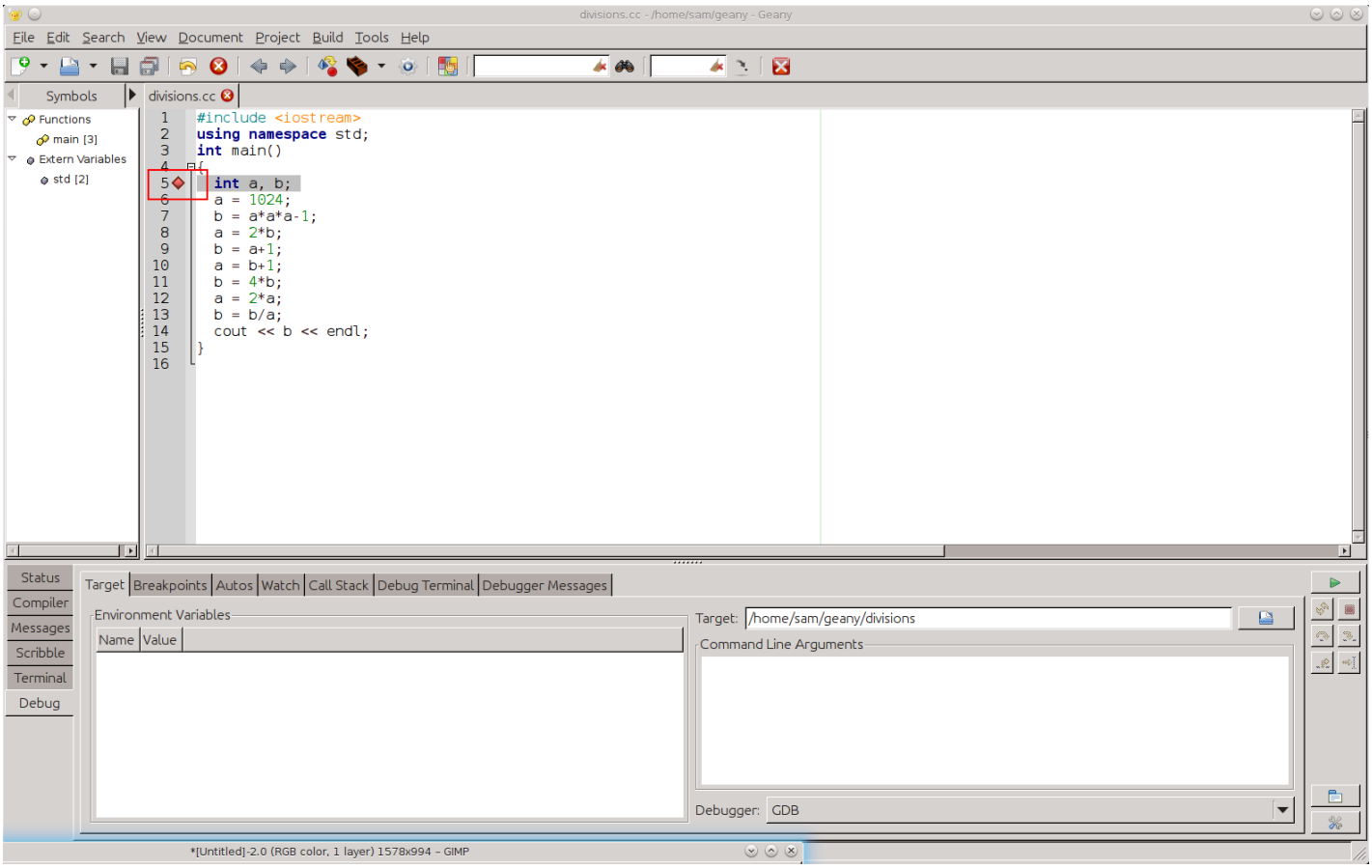
Vous devez pouvoir ainsi observer les valeurs `a=0` et `b=-4`. Ce sont les valeurs des variables au moment où l'erreur a été détectée. La cause de l'erreur devient évidente : la division par `a=0`.

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

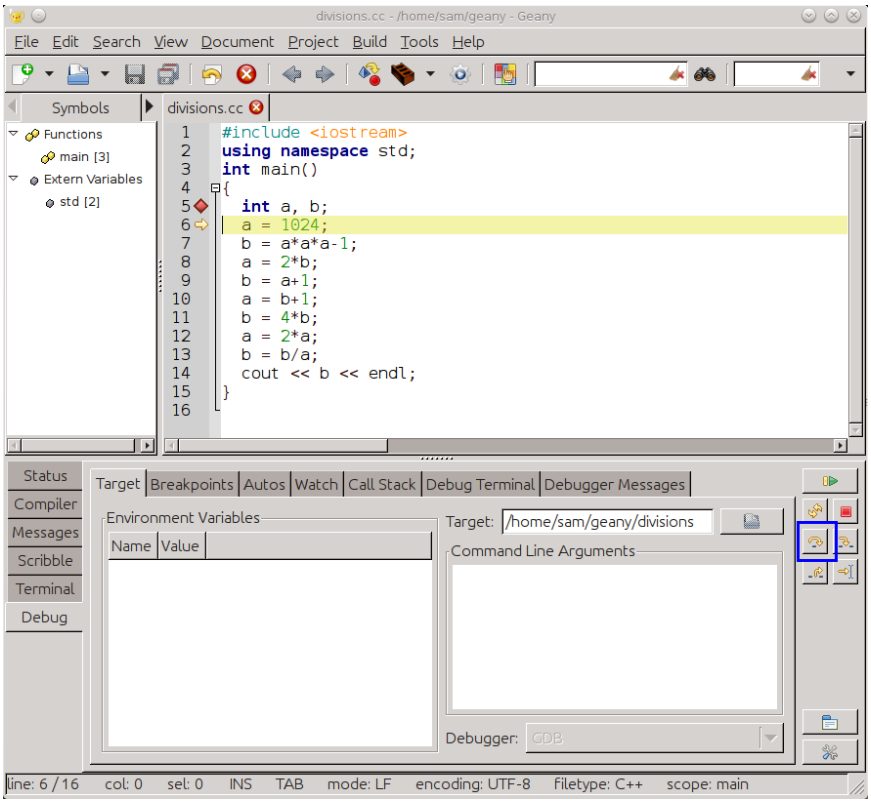
A.4. Exécuter le programme pas-à-pas

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 :

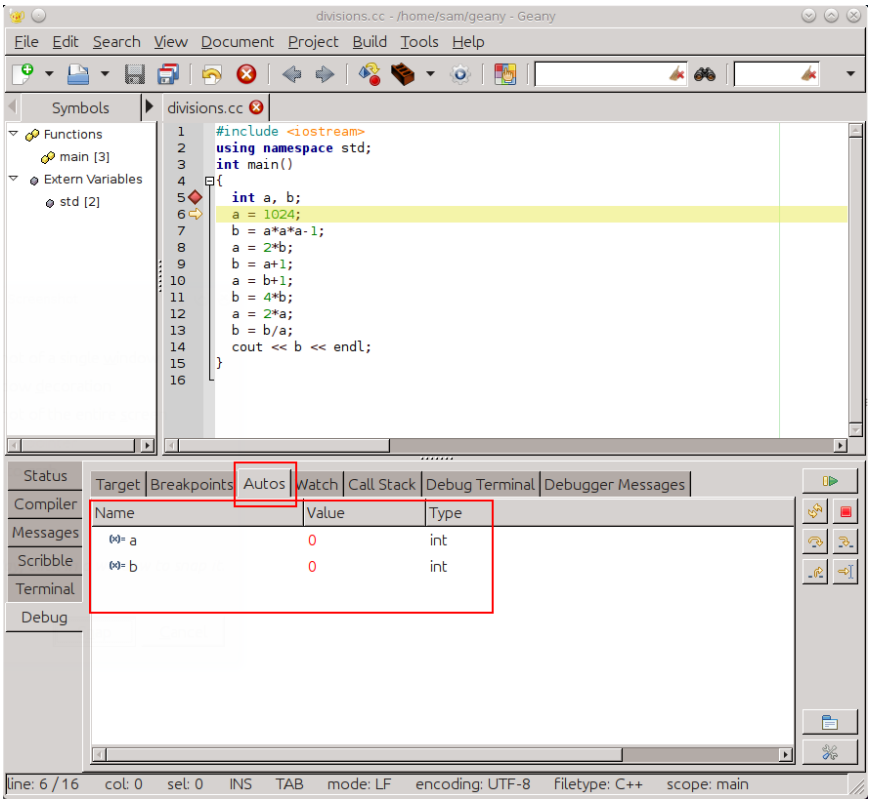


- pour exécuter une ligne à la fois, cliquez sur **`step over`**, (bouton encadré en bleu ci-dessus).

- 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.c* 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)

A.5. Programme avec plusieurs sources

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

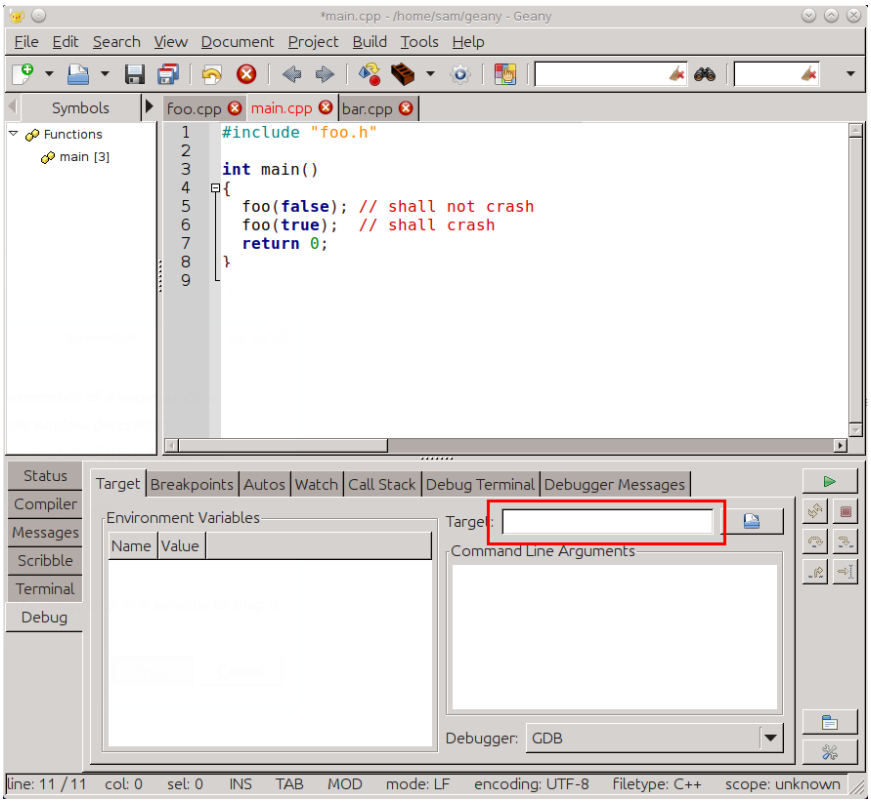
Pour cette sous-section et la suivante, **téléchargez l'exemple fourni** et désarchivez-le dans le dossier de votre choix (depuis le terminal vous pouvez exécuter `unzip gdbTest`).

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 certainement amené(e)s à pratiquer dans votre future vie de programmeuse/programmeur.)

Dans `Geany`, ouvrez le fichier `main.c` (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 débermineur.

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



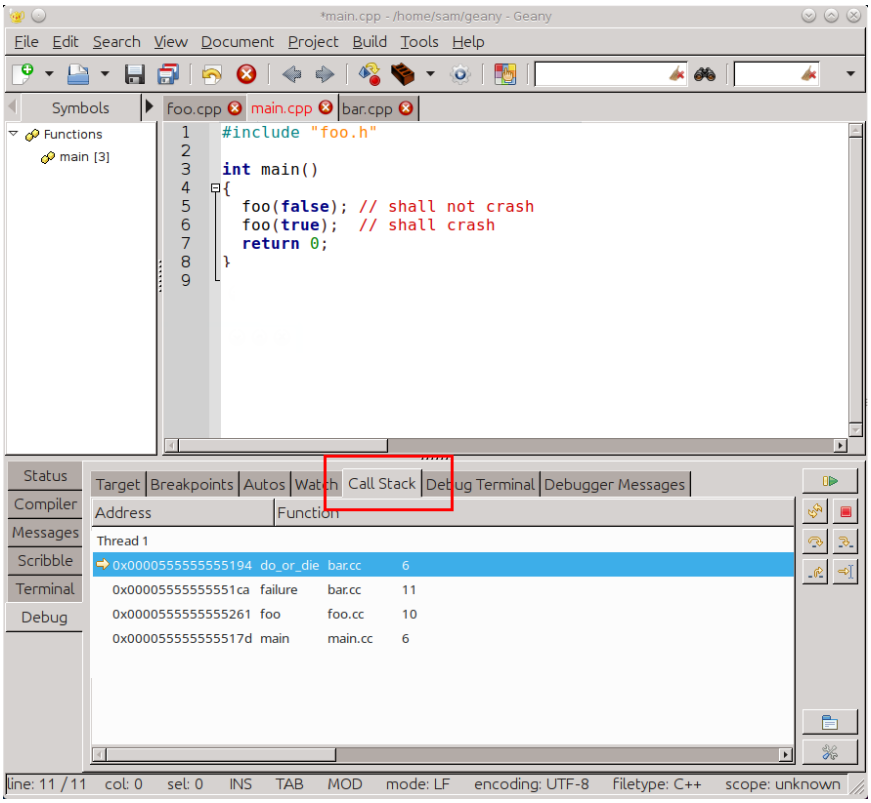
Lancez alors le programme au moyen de la flèche verte, une petite fenêtre s'affiche indiquant qu'une erreur s'est produite. Lorsque l'on ferme cette fenêtre, une flèche indique que l'instruction de la ligne 6 du programme `bar.c` est fautive.

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.

A.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.

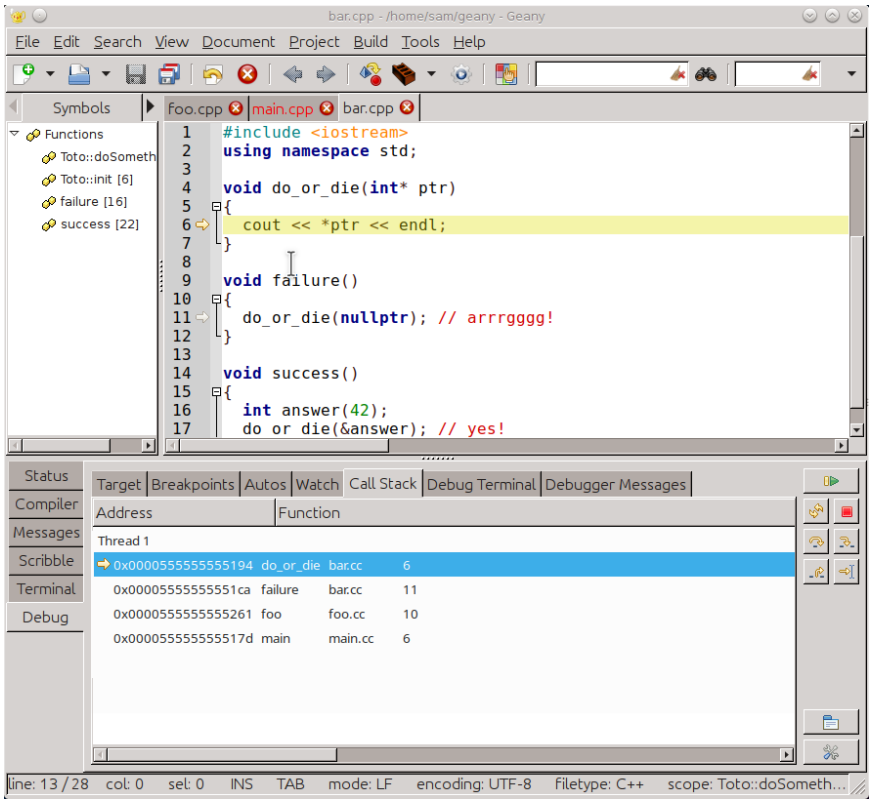
Pour visualiser la pile des appels au moment du crash que nous venons de provoquer, utilisez le bouton `Call Stack`:



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.c: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.

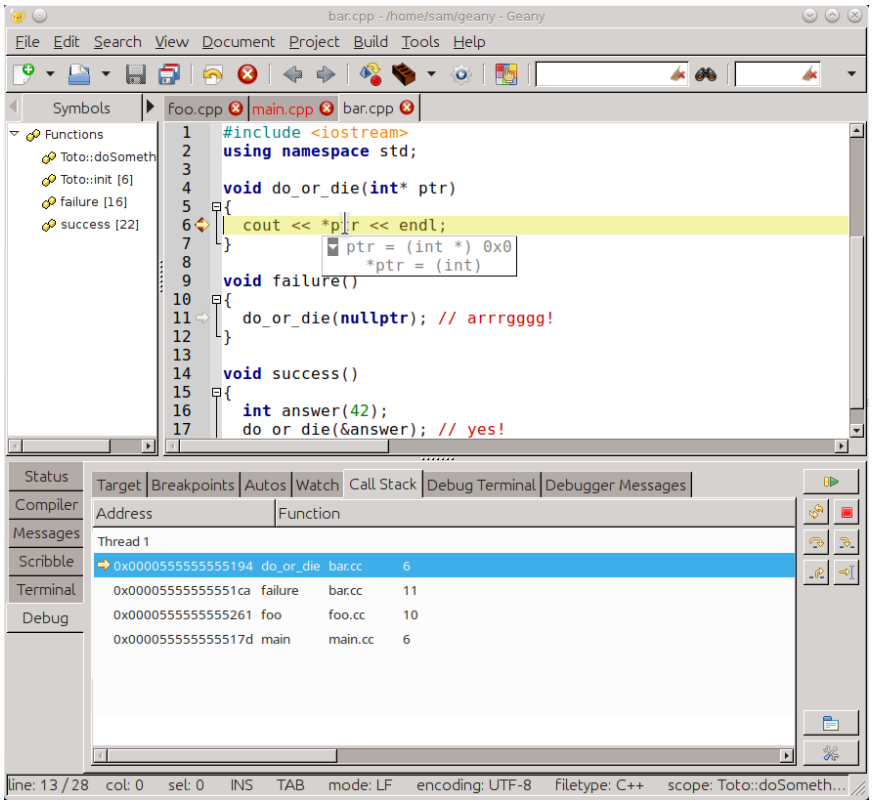
D'après la pile d'appel, la toute dernière instruction provoquant le crash a lieu lors de l'appel de l'opérateur `<<` :



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...

B - 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).

B.1. Compiler pour le déverminage

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

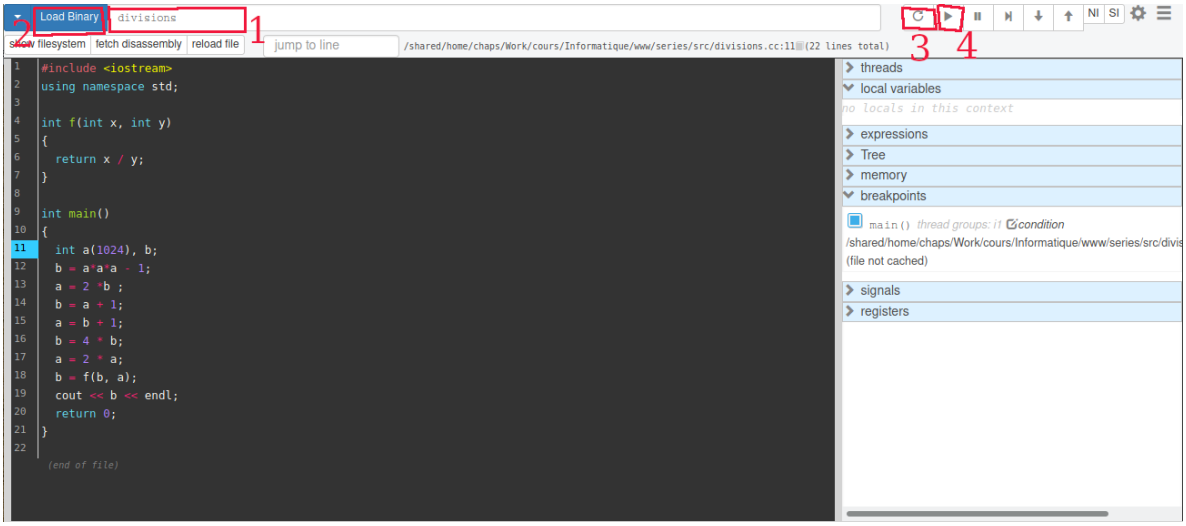
Important : pour pouvoir utiliser un débogueur sur un programme, il faut le compiler avec l'option *-g*.

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

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

B.2. Lancer le débogueur

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



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.

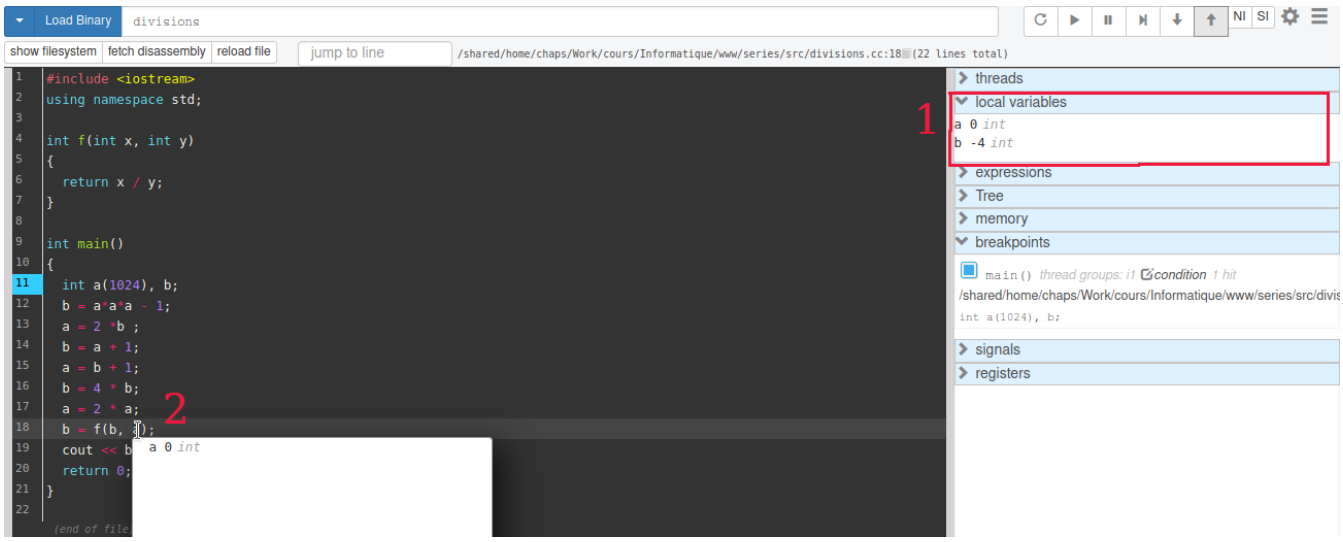
Notez que le débogueur vous indique l'endroit où se produit l'erreur. C'est déjà intéressant pour savoir où un programme plante...
En plus `gdbgui` vous indique à droite («*local variables*») les valeurs des variables locales de l'endroit où il s'est arrêté !

B.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):



B.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 « flêche 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 flêche 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)

B.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 intp ».

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 flêche droite ici (« step over »), l'exécution de `f()` se fera avec pour but du déverminneur 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 flêche vers le bas (« step into »), le déverminneur 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.

B.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 déverminage 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 déverminneur.

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

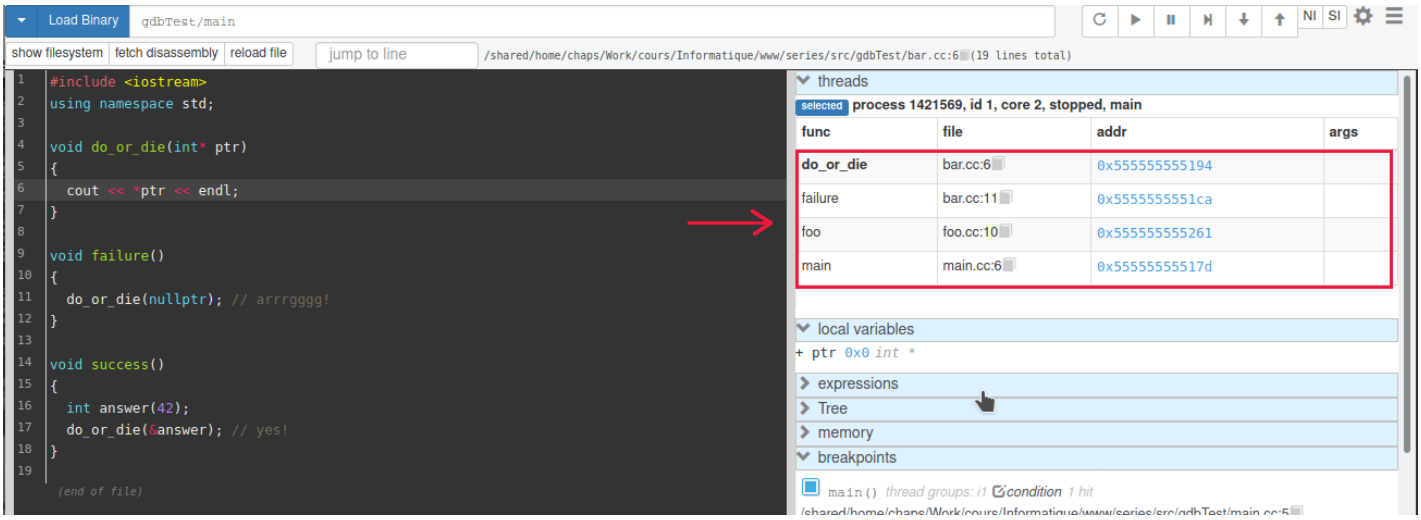
Puis lancez l'exécution du programme.
Vous devriez voir s'afficher un message comme quoi le programme s'est terminé avec une erreur à l'instruction de la ligne 6 du programme `bar.c`. (Le nom du programme courant est affiché juste au dessus de la fenêtre de code, à droit du petit cadre blanc « jump to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

B.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 :



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.c: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.c`).

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...

C - Déverminage dans le terminal avec gdb

C.1. Compiler pour le déverminage

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

Important : pour pouvoir utiliser un déverminueur 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 déverminueur (« debugger ») va vous permettre de localiser l'erreur dans le programme et d'en déterminer la cause.

C.2. Lancer le déverminueur

Pour lancer l'exécution du programme au moyen du déverminueur `gdb` dans le terminal, allez dans le répertoire où se trouve l'exécutable à déverminer et lancer 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 :

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

C.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'abréger 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.

C.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
```

puisqu'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)

C.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 déverminer 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 déverminer 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`).

C.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 déverminage 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 déverminer.

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

```
gdb main
```

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

```
layout src
```

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

```
run
```

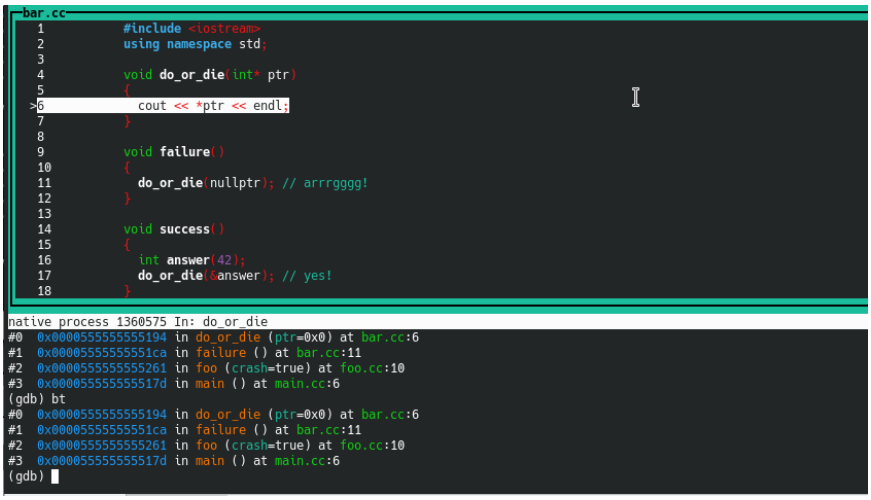
Vous devriez voir s'afficher un message comme quoi le programme s'est terminé avec une erreur à l'instruction de la ligne 6 du programme `bar.c`.

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.

C.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`:



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.c: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 2 : Profiling (niveau 1)

Cet exercice s'intéresse au «profiling».

Commençons par choisir un cadre d'étude simple. Programmons la suite de Fibonacci de deux manières différentes: itérativement et récursivement, et comparons leur temps de calcul.

2.1 Suite de Fibonacci

Les nombres de Fibonacci sont définis par la suite :

`F(0) = 0`
`F(1) = 1`
`F(n) = F(n-1)+ F(n-2)` avec `n>1`

Dans le fichier `fibonacci.c`, prototypez puis définissez la fonction

```
unsigned int fibonacci(unsigned int n)
```

qui calcule la valeur de `F(n)` de manière récursive.

Pour comparaison, voici la manière itérative de calculer les `n` premiers termes de la suite :

```
unsigned int fibonacciIteratif(unsigned int const n)
{
  unsigned int Fn   = 0; /* stocke F(i)  , initialisé par F(0)      */
  unsigned int Fn_1 = 0; /* stocke F(i-1), initialisé par F(0)      */
  unsigned int Fn_2 = 1; /* stocke F(i-2), initialisé par F(1)-F(0) */

  if (n != 0) {
    unsigned int i;
    for (i = 1; i <= n; ++i) {
      Fn   = Fn_1 + Fn_2;    /* pour n>=1 on calcule F(n)=F(n-1)+F(n-2) */
      Fn_2 = Fn_1;           /* et on decale... */
      Fn_1 = Fn;
    }
  }
  return Fn;
}
```

Note : la méthode récursive est coûteuse en temps de calcul (elle est « exponentielle »), ne lancez pas le calcul pour des nombres trop élevés (disons supérieurs à 40). Complétez votre programme avec la fonction `demander_nombre` (cf **série 4, exercice 1**) de sorte à obtenir le déroulement suivant :

```
Entrez un nombre entier compris entre 0 et 40 : 0
Méthode itérative :
    F(0) = 0
Méthode récursive :
    F(0) = 0
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 1
Méthode itérative :
    F(1) = 1
Méthode récursive :
    F(1) = 1
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 2
Méthode itérative :
    F(2) = 1
Méthode récursive :
    F(2) = 1
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 3
Méthode itérative :
    F(3) = 2
Méthode récursive :
    F(3) = 2
Voulez-vous recommencer [o/n] ? o
Entrez un nombre entier compris entre 0 et 40 : 7
Méthode itérative :
    F(7) = 13
Méthode récursive :
    F(7) = 13
Voulez-vous recommencer [o/n] ? n
```

2.2 Profiling

On veut ici examiner le temps mis par chaque fonction (profiling).

Pour cela compiler votre programme `fibonacci.c` avec l'option `-pg` :

```
gcc -pg fibonacci.c -o Fibonacci
```

Lancez votre programme :

```
fibonacci
```

Faites-le calculer une grosse valeur (par exemple 40).

Puis regardez les résultats :

```
gprof fibonacci | more
```

ou si vous préférez lire un fichier :

```
gprof fibonacci > fibonacci.prof
```

puis ouvrez le fichier `fibonacci.prof` pour le lire.

Exemple de résultats : voir le cours.

Pour les détails sur le format des fichiers produits par `gprof` :

```
man gprof
```

Exercice 3 : calcul de PGCD (algorithme d'Euclide, niveau 1)

(PGCD = Plus Grand Commun Diviseur)

Note: Pensez que vous pouvez maintenant utiliser le dévermineur pour vous aider dans le développement de vos programmes.

3.1 Énoncé

Écrivez, de façon modulaire, un programme qui :

1. demande à l'utilisateur d'entrer deux entiers `a` et `b` strictement positifs (et effectue les vérifications nécessaires),
2. trouve les entiers `u`, `v` et `p` satisfaisant l'identité de Bezout : `u a + v b = p`. Dans ce cas, on a `p = pgcd(a,b)`

3.2 Méthode

La méthode utilisée est **l'algorithme d'Euclide**.

On procédera par itération, comme suit (en notant `x / y` le quotient et `x % y` le reste de la division entière de `x` par `y`) :

0 : initialisation			$u_0 = 1$	$v_0 = 0$
	$x_1 = a$	$y_1 = b$	$u_1 = 0$	$v_1 = 1$
	...	...	...	...
i+1 : itération	$x_{i+1} = y_i$	$y_{i+1} = x_i \% y_i$	$u_{i+1} = u_{i-1} - u_i(x_i / y_i)$	$v_{i+1} = v_{i-1} - v_i(x_i / y_i)$
	...	...	...	...
Valeurs finales	x_{k-1}	y_{k-1} **!= 0**	u_{k-1}	v_{k-1}
condition d'arrêt: quand $y_k = 0$	**$p = x_k$**	$y_k = 0$	inutile	inutile

C'est-à-dire que l'on va calculer de proche en proche les valeurs de `x`, `y`, `u` et `v`. En calculant à chaque fois les nouvelles valeur en fonction des anciennes (et en faisant bien attention de mémoriser ce qui est nécessaire à un calcul correct, voir les indications ci-dessous).

Par exemple, $y_{i+1} = x_i \% y_i$ veut dire : "la nouvelle valeur de `y` vaut l'ancienne valeur de `x` modulo l'ancienne valeur de `y`".

Programmez ces calculs dans une boucle, qui s'exécute tant que la condition d'arrêt n'est pas vérifiée.

Pensez à initialiser correctement vos variables avant d'entrer dans la boucle.

3.3 Indications

Vu les dépendances entre les calculs, vous aurez besoin de définir (par exemple) les variables : `x`, `y`, `u`, `v` **et** `q=x/y`, `r=x%y`, `prev_u`, `prev_v`, `new_u` et `new_v`.

Vous mettrez ces variables à jour à chaque itération, à l'aide des formules de la ligne *i+1* et des relations temporelles évidentes entre elles (par exemple `prev_u = u`).

Testez que `y` est non nul avant d'effectuer les divisions !

3.4 Exemple d'execution

```
Entrez un nombre entier supérieur ou égal à 1 : 654321
Entrez un nombre entier supérieur ou égal à 1 : 210
Calcul du PGCD de 654321 et 210

  x         y         u         v
210       171         1     -3115
171        39        -1      3116
 39        15         5    -15579
 15         9       -11     34274
  9         6        16    -49853
  6         3       -27     84127
  3         0        70   -218107

PGCD(654321,210)=3
```

3.5 Notes

1. Ceux qui se sentent assez à l'aise peuvent le faire en utilisant la compilation séparée (conseillé pour la suite) et en profiter pour réutiliser leur "librairie" `demander_nombre.o` (voir l'**exercice 1 de la série 4**).
2. Remarquez que pour le seul calcul du PGCD, le calcul de `x` et `y` par l'algorithme ci-dessus suffit. Pas besoin de `u` et `v`. Ils sont ici pour trouver l'équation de Bezout (et vous faire programmer des suites imbriquées).

Exercice 4 : arithmétique rationnelle (niveau 1)

Nous nous intéressons ici aux nombres rationnels et à leur arithmétique.

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.
```

Quel type de données utiliser pour représenter les nombres rationnels ?

Écrivez un programme `rationnels.c`, 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 4 opérations élémentaires :

l'addition

```
Rationnel* addition(Rationnel const * r1, Rationnel const * r2, Rationnel* rez);
```

définie par

`p1/q1 + p2/q2 = reduction((p1*q2+p2*q1)/(q1*q2))`

où `reduction(p/q)` trouve les nombres `p0` et `q0` tels que `p0/q0 = p/q` et `p0` et `q0` vérifient la définition donnée plus haut (en particulier sont premiers entre eux).

On utilisera pour la fonction de réduction le calcul de PGCD implémenté dans l'**exercice précédent**.

Exemples :

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

Remarques :

1. toute sophistication de la formule ci-dessus (en particulier si `q1 = q2`) peut évidemment être implémentée pour la fonction `addition`.
2. On fera particulièrement attention à la gestion de la mémoire. La fonction `addition` créant un nouveau rationnel si le dernier argument est NULL. Sinon, le résultat est stocké dans `rez` et `rez` est également retourné comme valeur. Ce qui permet d'écrire `addition(addition(&z1,&z2,&z3),&z4,&z5)` (pour "`z5 = (z1+z2)+z4`", `z3` contenant le résultat intermédiaire `z1+z2`).

la soustraction

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

définie par

```
p1/q1 - p2/q2 = 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, Rationnel* rez);
```

définie par

```
p1/q1 * p2/q2 = 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, Rationnel* rez);
```

définie par:

- `p1/q1 / (p2/q2) = reduction(p1*q2/(q1*p2))` **si** `p2 > 0`,
- `p1/q1 / (p2/q2) = reduction((-p1)*q2/(q1*(-p2)))` **si** `p2 < 0`,
- non définie **si** `p2 = 0`.

Exemples :

```
1/2 / (-3/5) = -5/6
2 / (-3/5) = -10/3
-3/5 / (-3/5) = 1
```

Testez votre programme avec du code comme (faire ce qui faut pour que cela fonctionne !!)

```
affiche(addition(multiplication(&z1,&z2,&z3), multiplication(&z2,&z4,&z5), &z6));
```

par exemple pour calculer `1/2 * (-3/5) + (-3/5) * 2` qui vaut `-3/2`.

Exercice 5 : Compression RLE (niveau 2)

5.1 Introduction

La compression RLE (*Run Length Encoding*) est une ancienne technique de compression, utilisée en particulier à l'époque dans le codage des images. Le principe très simple de cette technique est le suivant : toute séquence de caractères *c* 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» (i.e. pas de caractère *c*, et *L* = 0).

5.2 Énoncé

Dans le fichier `rle.c`:

1. prototypez et définissez la fonction :

   ```
   char* compresse(char const * 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 pourrait être de 4.

   Comme nous codons la longueur par un caractère lisible (`0` à `9`), la longueur **maximale** d'une séquence compressée est alors de 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 :

   ```
   char* decompresse(char const * rledata, char flag);
   ```

   de sorte qu'elle retourne la chaîne `rledata` sous sa forme décompressée.
3. Utilisez la valeur `NULL` en cas d'erreur dans la fonction de décompression.

5.3 Indications

Pour transformer une valeur entière `i` comprise entre 0 et 9 en sa représentation sous forme de caractère, utilisez le truc suivant : `'0' + i`.

Et réciproquement, si `c` contient un caractère représentant un chiffre (entre 0 et 9 donc !), la valeur `(int) (c-'0')` représente cette valeur sur un entier.

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

```
Entrez les données à comprimer : #aaaaa
Forme compressée:   #0a#5
[ratio = 83.3333%]
décompression de vérification ok.
Entrez des données à décompresser : #0a#3
décompressées : #aaa

Entrez les données : aa#4baaaabb#ddddddddddd###aaaaaaaaaaaaaaa
Forme compressée : aa#04ba#4bb#0d#9dd#0#0#0a#9a#6
[ratio = 73.17%]
décompression de vérification ok.
Entrez les données à décompresser : aa#04ba##4bb#0d
Erreur de décompression !
```

Exercice 6 : Machine de Turing (programmation, niveau 3)

Le but est d'écrire un programme `turing.c` permettant de simuler une machine de Turing .

Une machine de Turing est une machine à états (i.e. elle se trouve à chaque instant dans un certain état. On peut par exemple désigner les états par des numéros) qui posséde un «ruban» infini sur lequel on peut lire et écrire des données, et une tête de lecture/écriture (c'est-à-dire en fait une position) sur ce ruban.

Le fonctionnement de la machine de Turing est décrit par une table dite «de transition» qui pour chaque couple (état de la machine, caractère lu) donne le nouvel état de la machine, le caractère à écrire sur le ruban et le déplacement (avance ou recule) de la tête de lecture (la machine est obligée de réécrire un caractère et de déplacer sa tête).

Cette table est donc indexée par 2 éléments : un numéro d'état et un caractère lu.

Par exemple (en pseudo-code) si la machine est dans l'état "12" et a lu le caractère 'a', on pourrait avoir `transition[12]['a'] = (25, 'x', AVANCE)` ce qui signifie que la machine écrit `x` sur le ruban (à la place de `a`) avance la tête de lecture (d'une «case») et se met dans l'état 25.

Par convention, un état qui n'est pas dans la table de transition (c'est-à-dire négatif ou plus grand que le nombre de lignes de la table) signifie que la machine s'arrête.

Passons donc maintenant à la simulation d'une machine de Turing.

1. Pour simuler une bande de longueur infinie, vous pouvez utiliser un tableau dynamique ou une liste chaînée ou encore 2 demi-bandes, etc (définissez par exemple un type `Bande`) :
   - Lorsque l'on cherchera à déplacer la tête au-delà du dernier élément, il suffira d'ajouter en fin de tableau/chaîne un élément supplémentaire, initialisé avec le symbole 'epsilon' (à définir).
   - Lorsque l'on cherchera à déplacer la tête avant le premier élément, il suffira d'insérer en début de tableau/chaîne le symbole 'epsilon' (ou en fin de demi-bande arrière pour la solution à 2 demi-bandes).
2. La tête de lecture sera représentée simplement par un pointeur, indiquant sa position courante dans la bande. Je vous conseille aussi de définir un type ici, par exemple `Tete`.
3. Créez ensuite les fonctions implémentant les primitives de lecture, écriture et positionnement de la tête de lecture :
   `char lecture(Bande* b, Tete t);` *lit le caractère 'courant'.*
   `void ecriture(char c, Bande* b, Tete t);` *remplace le caractère courant par la valeur transmise en argument.*
   `void avance(Tete* t);` *déplace la tête de lecture vers la droite* Peut être une macro.
   `void recule(Tete* t);` *déplace la tête de lecture vers la gauche* Peut être une macro

   Notez que, dans l'implémentation proposée, `lecture` et `ecriture` peuvent changer la valeur de `tete` : on simulera ainsi le fait que la bande est infinie, en insérant le bon nombre d'epsilon en début de bande lorsque la position de la tête est négative (naturellement, lorsque les espilons auront étés insérés, on devra repositionner la tête à zéro). [on aurait également pu choisir d'effectuer ces opérations - simuler une bande infinie - lors des déplacements de la tête...]

4. Définissez une fonction `initialise(Bande * b, char const * valeur)` permettant d'initialiser la bande au moyen d'une chaîne de caractères
5. Pour représenter la table de transitions, utilisez un tableau statique d'au plus, disons, 512 lignes de transitions :

   - chaque 'case' définissant une transition est une structure de 3 champs ([prochain état, caractère, déplacement]);
   - chaque ligne de transition est un tableau de 3 'cases', une par valeur possible du caractère courant ([0, 1, epsilon]);

   ```
   struct Transition
   {
     Etat etat;
     char caractere;
     int deplacement;
   }
   typedef Transition Ligne[3]; // [ 0 , 1 , epsilon ]
   ```

   Avec une telle représentation, une table de transitions sera simplement un tableau de (au plus 512) `Ligne`s.

6. Écrivez une structure **`TMachine`**, simulant une machine de Turing.

   Vous aurez besoin des champs suivants :

   - un entier (ou un type prédéfini par vous) modélisant l' **état actuel** de la machine;
   - une bande ;
   - une tête de lecture
   - une table de transitions
7. Définissez les fonctions suivantes :
   - une fonction permettant de créer une machine de Turing
     `void construit(TMachine* m);`
   - une fonction permettant de ré-initialiser la machine (pour recommencer avec une nouvelle entrée)
     `void reinitialise(TMachine* m, char const * entree);`
   - une fonction permettant de démarrer la simulation de la machine.
     Choisissez une convention sur la valeur de l'état pour stopper la machine (par exemple, une valeur négative).

Application : Test de Parité

Testez votre machine avec l'exemple suivant testant la parité de l'entrée.

Avec la modélisation proposée, utilisez la syntaxe suivante pour définir une table de transition (dans la fonction `construit`) :

```
TableTransition table = {
  { {    1,     '0', +1}, {    1,     '1', +1}, {    2, EPSILON, -1} },
  { {    3, EPSILON, -1}, {    4, EPSILON, -1}, {UNDEF, EPSILON, -1} },
  { {    3, EPSILON, -1}, {    3, EPSILON, -1}, {    5,     '1', +1} },
  { {    4, EPSILON, -1}, {    4, EPSILON, -1}, {    5,     '0', +1} },
  { {UNDEF, EPSILON, -1}, {UNDEF, EPSILON, -1}, {UNDEF, EPSILON, -1} }
};
```

La constante `UNDEF` représente un état non défini également utilisé pour "l'ordre de fin d'exécution".

Dans cet exemple, on suppose que le déplacement vers l'avant est représenté par la valeur +1, et le déplacement vers l'arrière par -1.

Exemple (détaillé) d'execution :

```
Lancement de la machine de Turing
   --------------------------------
   Etat actuel : 1
   Bande : 10e
           ^ (Tete : 0)
   --------------------------------
 lu : 1, nouvel état : 1, écrit : 1, avance
   --------------------------------
   Etat actuel : 1
   Bande : 10e
            ^ (Tete : 1)
   --------------------------------
 lu : 0, nouvel état : 1, écrit : 0, avance
   --------------------------------
   Etat actuel : 1
   Bande : 10e
             ^ (Tete : 2)
   --------------------------------
 lu : e, nouvel état : 2, écrit : e, recule
   --------------------------------
   Etat actuel : 2
   Bande : 10e
            ^ (Tete : 1)
   --------------------------------
 lu : 0, nouvel état : 3, écrit : e, recule
   --------------------------------
   Etat actuel : 3
   Bande : 1ee
           ^ (Tete : 0)
   --------------------------------
 lu : 1, nouvel état : 3, écrit : e, recule
   --------------------------------
   Etat actuel : 3
   Bande : eee
           ^ (Tete : -1)
   --------------------------------
 lu : e, nouvel état : 5, écrit : 1, avance
   --------------------------------
   Etat actuel : 5
   Bande : 1eee
            ^ (Tete : 0)
   --------------------------------
 lu : e, nouvel état : 0, écrit : e, recule
   --------------------------------
   Etat actuel : 0
   Bande : 1eee
           ^ (Tete : -1)
   --------------------------------
Arrêt de la machine de Turing.
```

Dernière mise à jour : $Date: 2013/01/11 13:26:45 $ (`$Revision: 1.13 $`)