V 1.1

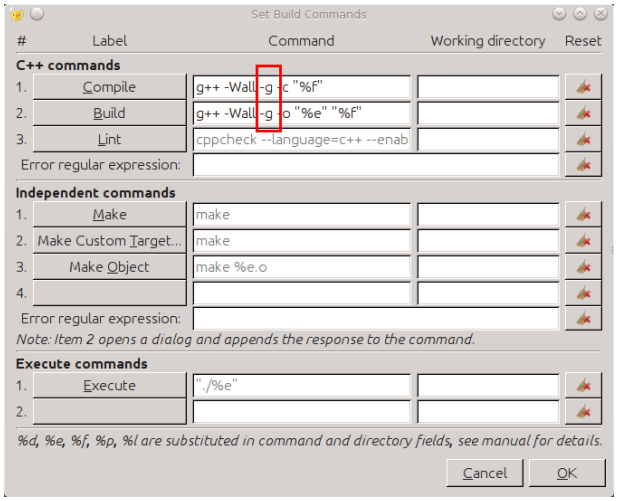
VM: guide de déverminage dans le terminal avec gdb

Version adaptée à partir du document écrit par J-C Chappelier

1. Compiler pour le déverminage

Dans geany, ouvrez le fichier **division.cc** fourni dans le fichier archive Code_tests_debugger sur moodle, puis compilez le pour produire un executable s'appelant **division** .

Important : pour pouvoir utiliser un déverminneur sur un programme, il faut le compiler avec l'option **-g**. Voici comment ajouter cette option avec geany: Build > Set Build Commands :



Si vous lancez l'exécution, le programme s'arrête avant la fin, et vous obtenez un message d'erreur : « Floating exception (core dumped) ». Le déverminneur (« debugger ») va vous permettre de localiser l'erreur dans le programme et d'en déterminer la cause.

2. Lancer le déverminneur

Pour lancer l'exécution du programme au moyen du déverminneur **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[1] :

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)
```

[1] voir la section 8 pour un programme qui demande des arguments sur la ligne de commande

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

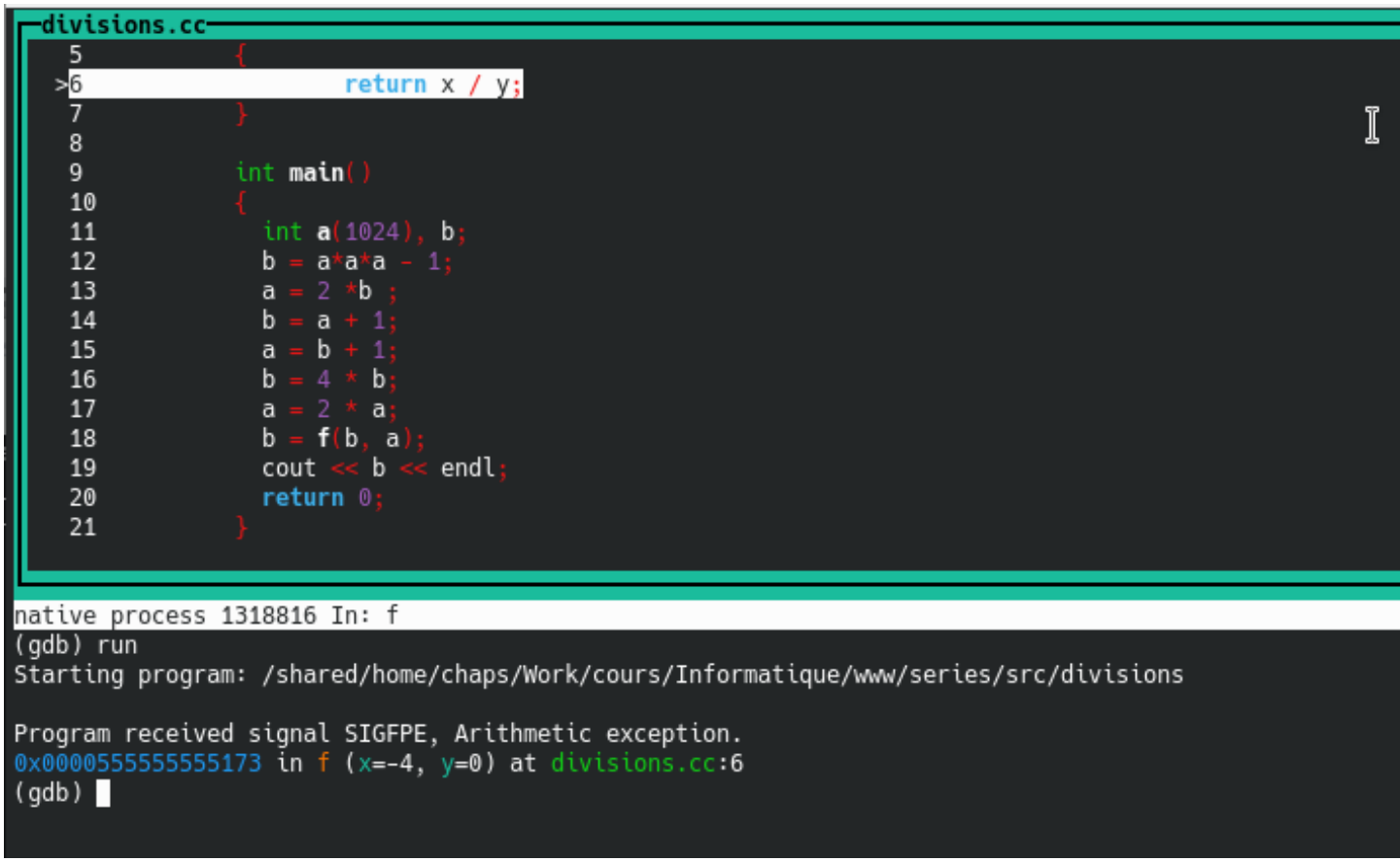
layout src

Cette première commande n'est pas suffisante pour voir le code source. Il faut lancer l'exécution du programme, en tapant (dans gdb) :

run

Vous n'avez pas forcément l'option "couleur" pour l'affichage du code source.

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...

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.

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** car les variables de type **int** sont représentées sur 4 octets en complément à deux et donc elle sont bornées dans l'intervalle **$[-2^{31}, 2^{31}-1]$.**

En complément à deux, on produit *une violation du domaine couvert* quand on ajoute une unité à **$2^{31}-1$**.

On obtient le minimum des entiers négatifs **-2^{31}** qui a un 1 comme bit de signe et des 0 partout ailleurs.

Si, ensuite, on multiplie par 2 cette quantité, en théorie on devrait avoir **-2^{32}** MAIS dans la pratique on ne dispose que de 32 bits et on sait que **-2^{32}**, tout comme **2^{32}**, ne sont pas représentables sur 32 bits.

Le résultat est en fait le motif binaire nul car le motif binaire de **-2^{31}** a été décalé d'un bit vers la gauche.

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-Théorie)

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* façons 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 into* » 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).

6. Programme avec plusieurs sources

Pour cette sous-section et la suivante, récupérez le fichier archive **gdbTest** dans Code tests debugger sur moodle puis désarchivez-le dans le dossier de votre choix. 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)

Dans le terminal, allez dans le répertoire **gdbTest/** et compilez le programme en utilisant la commande **make**. 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

Cette première commande n'est pas suffisante pour voir le code source. Il faut lancer l'exécution du programme, en tapant (dans gdb) :

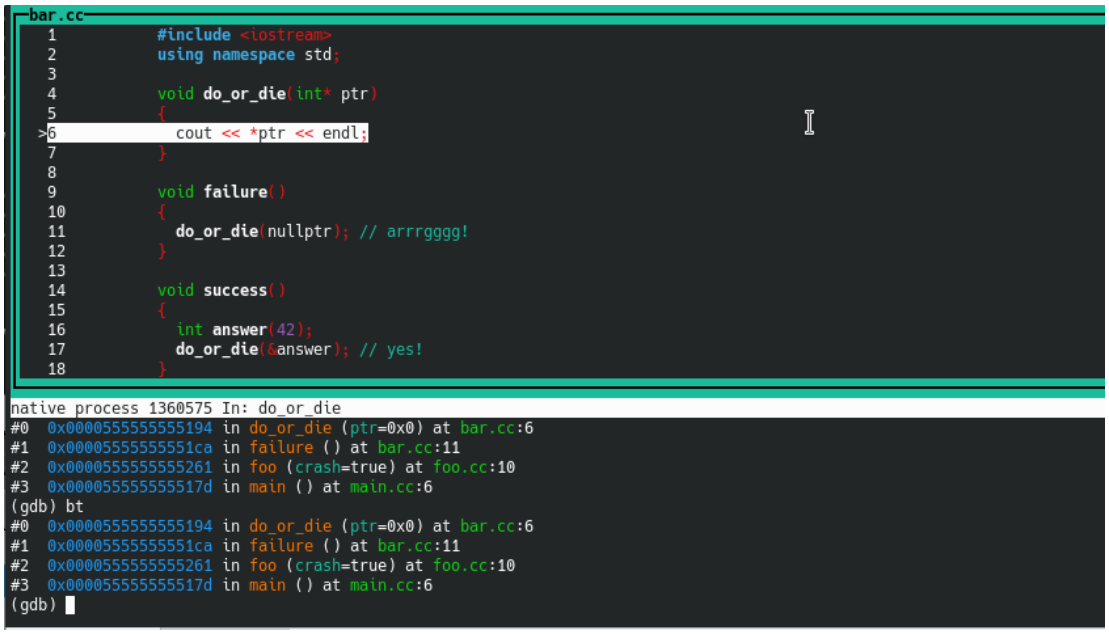
run

Vous n'avez pas forcément l'option "couleur" pour l'affichage du code source. Vous devriez voir s'afficher un message comme quoi le programme s'est terminé avec une erreur à 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**:



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 à la première instruction de la fonction **failure()**) ;
- puis relancer l'exécution.
- Faire alors **step** pour entrer dans la function **do_or_die()**. Vous voyez alors l'affichage de la valeur du parameter ptr ; on constate bien qu'il est nul. On peut demander confirmation avec la commande **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...

8. Comment utiliser gdb pour un programme qui demande des arguments sur la ligne de commande ?

Supposons que l'exécutable s'appelle **projet**. Le lancement de gdb rest le même: **gdb ./projet**
C'est au moment de lancer la commande **run** qu'il faut ajouter les arguments après **run**. Par exemple, si ce programme demande un nom de fichier, on lancera la commande **: run test33.txt**
Où **test33.txt** est un nom de fichier présent dans le répertoire courant.