TP Débogage

13 décembre 2011

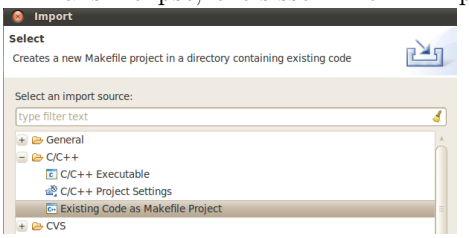
⇒ Les noms de fichiers indiqués en bas des exemples sont les chemins relatifs au répertoire `/home/lem2i` ⇐

1 Récupération du support du TP

Dans un terminal tapez `wget http://hpc-web.u-strasbg.fr/Debug/debug.sh` puis exécutez le script obtenu

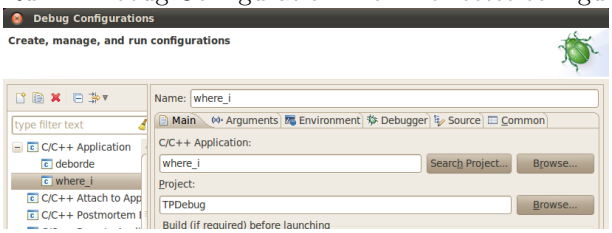
2 Eclipse - Préparation du TP

Dans Eclipse, choisissez File → Import → C/C++ → Existing code as Makefile project :



Choisissez le répertoire `TPDebug`, Toolchain `Linux GCC` Le `Makefile` fournit est analysé par Eclipse, et le premier exécutable (`deborde`) est prêt à être débogué.

Pour permettre le débogage d'autres exécutables, il faut ajouter des configurations de debug : Run → Debug Configuration. Nommez cette configuration et choisissez l'exécutable à déboguer :



L'application sera alors accessible dans la perspective debug.

3 Premiers pas

Un premier programme à problême

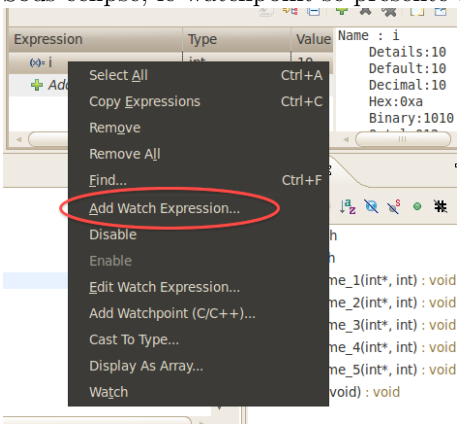
Le programme `TPDebug/why_i` produit un comportement incorrect. Utilisez `gdb`, `ddd` ou `eclipse` pour déboguer ce programme, qui ne rentre pas dans la boucle indiquée. Placez un breakpoint avant la boucle et examinez la valeur de la variable.

Il est également possible de calculer directement la valeur de la condition booléeenne dans le débogueur. Par exemple, dans Eclipse, on peut ajouter une nouvelle expression :

Expression	Type	Value
(x)= i	int	10
(x)= i!=max	int	0
Add new expression		

Mais qui modifie cette variable ?
Travaillez maintenant avec le programme `TPDebug/where_i`. Comme précédemment, utilisez `gdb`, `ddd` ou `eclipse`. Le programme produit un résultat incorrect parce que la variable `result` n'est pas conforme à ce qui est attendu. Seulement, cette variable est susceptible d'avoir été modifiée dans 5 fonctions. Pour savoir quelle fonction modifie la variable, placez un *watchpoint* sur cette variable. Vous pourrez ainsi déterminer la fonction dans laquelle elle a été modifiée.

Sous eclipse, le watchpoint se présente ainsi :



Analyse core dump, 2 variantes Travaillez avec le programme `TPDebug/1/deborde.c` qui produit un segfault (étonnant vu son nom). Nous l'analysons post-mortem avec gdb/ddd et durant son exécution avec valgrind.

1. Compilez le programme avec l'option `-g`. La première exécution de ce programme produit une erreur de segmentation et un fichier core
2. Lancez le programme sous le débogueur ddd : `ddd ./a.out core`. Le débogueur vous indique l'endroit où s'est produit le plantage.
3. Affichez la valeur de `i`. Que remarquez-vous ?
4. Lancez-le programme sous valgrind (`valgrind ./a.out`) . Un grand nombre d'erreurs sont affichées. Regardez la première erreur mémoire.

4 Visualisation de données

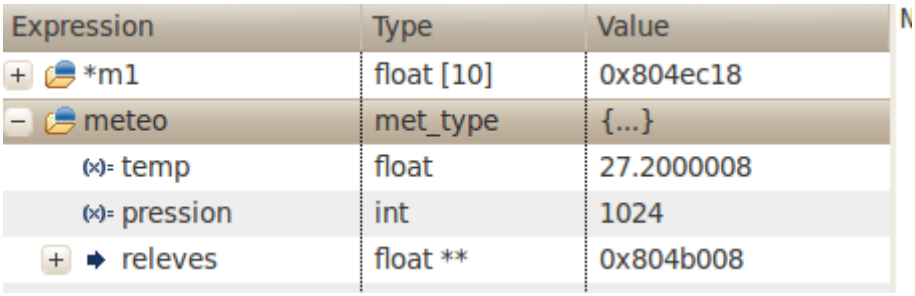
L'exercice suivant est disponible en C. Il se trouve dans le répertoire `TPDebug/2/`
Le programme :

1. crée un type de donnée structuré dont un des champs est une matrice. Très simplement, le but du TP est de visualiser les différents champs.
2. effectue un tri à bulles buggé (appelé fort élégament `tri_a_bug`). Cela nous permettra de diagnostiquer le problème en suivant l'évolution des données.

4.1 Version C

Compilez le fichier `tabl.c` : `gcc -g tabl.c -lm -o tabl` (ou via le Makefile fourni)

1. Placez un breakpoint à la ligne 101 En double-cliquant sur la variable meteo, les champes de la structure 2D apparaissent dans le workbench de ddd ou dans eclipse. La suite du TP se déroulera avec ddd :

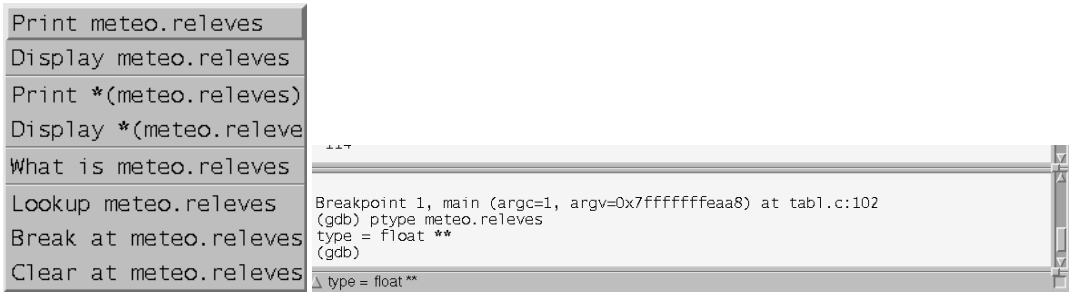


Sous Eclipse

Sous ddd

Il est possible de déréfencer (suivre) les pointeurs en double-cliquant sur un membre du type pointeur. ddd affiche alors le 1er élément pointé, sans bien savoir qu'il s'agit d'un tableau.

2. Le type inféré par le débogueur sous-jacent (`gdb` dans notre cas) est visible par le choix `whatis` *variable*.

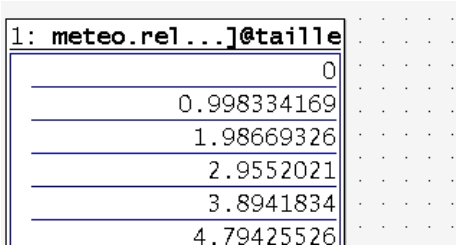


Dans la fenêtre basse, on observe que le type de `meteo.releves` est `float ** ddd`.

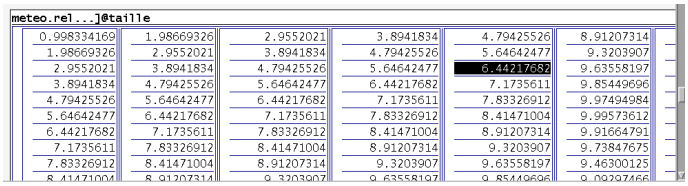
3. Pour afficher le contenu du tableau, il faut explicitement indiquer à gdb la taile de la zone mémoire à considérer (*artificial arrays*[1]). La syntaxe à utiliser est *premier_element@nombre d'elements*. Le contenu de la première ligne du tableau s'affiche dans le workbench en

[1] Contrainte : toutes les variables utilisées dans l'expression doivent être connues au moment de l'interrogation

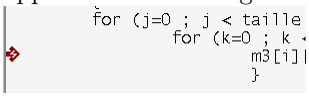
tapant : `graph display meteo.releves[0][0]@taille` :



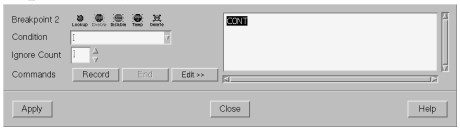
4. Affichage par tranches. Dans chaque dimension, on peut afficher un sous-ensemble des indices : `graph display meteo.releves[0..2][1..5]@taille`



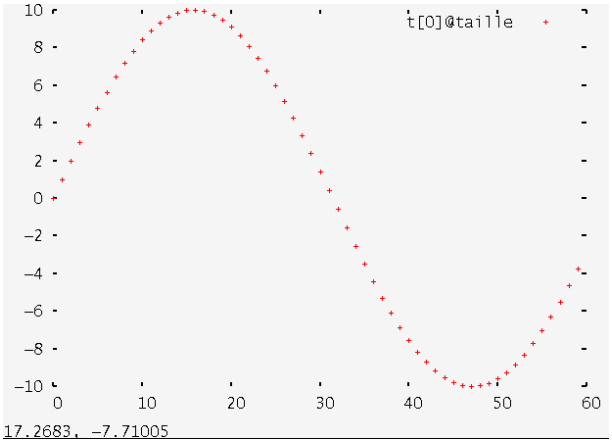
5. (optionnel) Observation du comportement de l'interface graphique Il s'agit d'observer une des indications données par le debugger en cours d'exécution du programme.
   (a) Relancez le programme sous le débogueur ;
   (b) Placez un breakpoint **ligne 96** ;
   (c) Affichez la variable `taille` et changez sa valeur à 1000 ;
   (d) Une fois le programme `CONT`inué, vous remaquerez d'une part le curseur clignotant en bas à droite de la fenêtre graphique. Au bout d'un moment, un symbole ensommeillé apparaît devant la ligne de code source active :



6. **Affichage graphique des données**
   (a) Relancez le programme sous le débogueur ;
   (b) Placez un breakpoint **temporaire** ligne 63 ;
   (c) Affichez les données du tableau passez en argument à la fonction : `graph plot t[0]@taille`
   (d) Tracez le contenu du tableau
   (e) Le contenu du graphique sera ré-actualisé à chaque nouveau franchissement breakpoint. Placez un breakpoint ligne 74. Pour que ce breakpoint soit franchi automatiquement, il faut l'enrichir d'une instruction `cont`, (clic droit sur le breakpoint) :



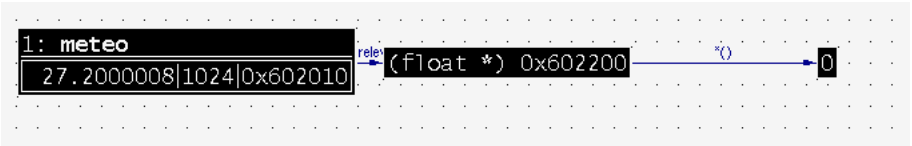
   (f) Le tracé animé doit vous permettre de comprendre le bug présent dans le programme.



4.2 Version C++

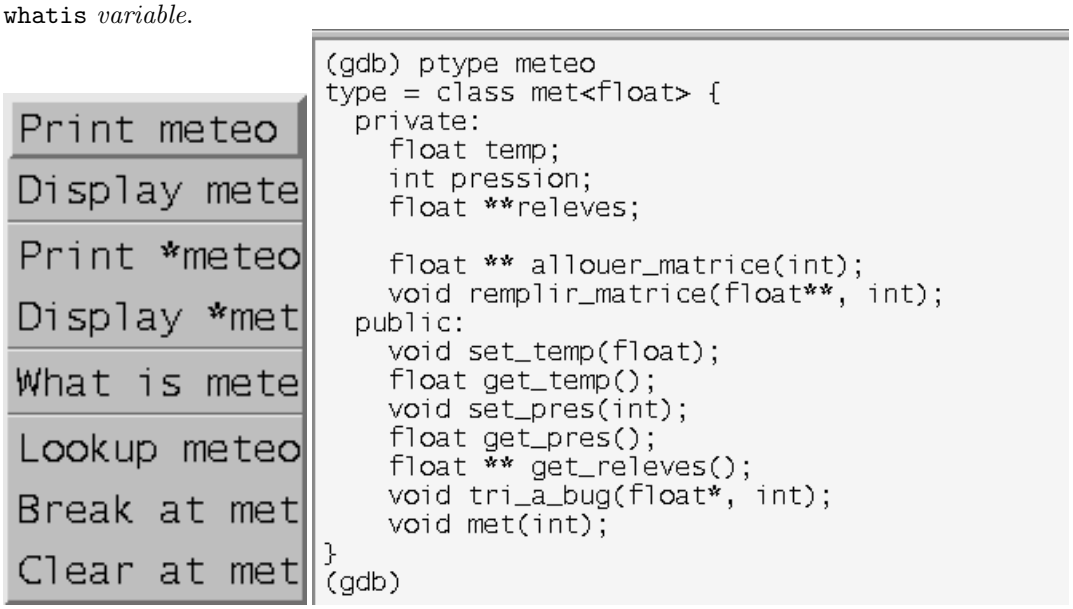
Compilez le fichier `tabl.cxx` : `g++ -g tabl.cxx -lm -o tabl`

1. Placez un breakpoint à la ligne 152 En double-cliquant sur la variable meteo, les champes de la structure 2D apparaissent dans le workbench de ddd :



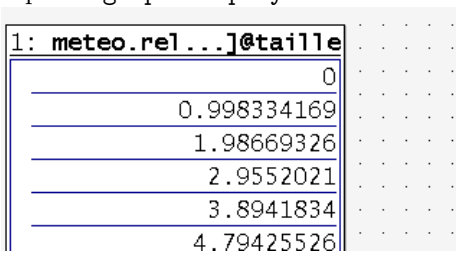
   Il est possible de déréfencer (suivre) les pointeurs en double-cliquant sur un membre du type pointeur. ddd affiche alors le 1er élément pointé, sans bien savoir qu'il s'agit d'un tableau.

2. Le type inféré par le débogueur sous-jacent (`gdb` dans notre cas) est visible par le choix `whatis` *variable*.

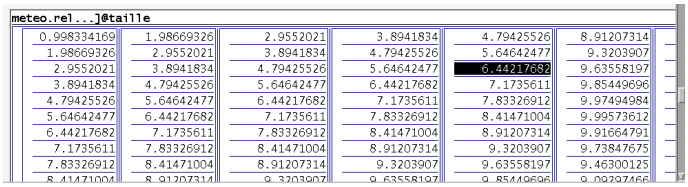


Dans la fenêtre basse, on observe que le type de `meteo.releves` est `float **`.

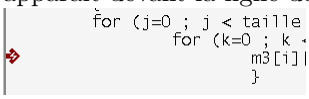
3. Pour afficher le contenu du tableau, il faut explicitement indiquer à gdb la taile de la zone mémoire à considérer (*artificial arrays*[2]). La syntaxe à utiliser est *premier_element*`@`*nombre d'elements*. Le contenu de la première ligne du tableau s'affiche dans le workbench en tapant : `graph display meteo.releves[0][0]@taille` :



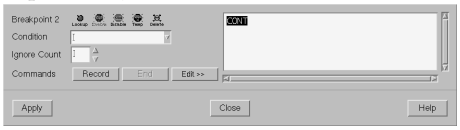
4. Affichage par tranches. Dans chaque dimension, on peut afficher un sous-ensemble des indices : `graph display meteo.releves[0..2][1..5]@taille`



5. (optionnel) Observation du comportement de l'interface graphique Il s'agit d'observer une des indications données par le debugger en cours d'exécution du programme.
   (a) Relancez le programme sous le débogueur ;
   (b) Placez un breakpoint **ligne 150** ;
   (c) Affichez la variable `taille` et changez sa valeur à 1000 ;
   (d) Une fois le programme `CONT`inué, vous remaquerez d'une part le curseur clignotant en bas à droite de la fenêtre graphique. Au bout d'un moment, un symbole ensommeillé apparaît devant la ligne de code source active :

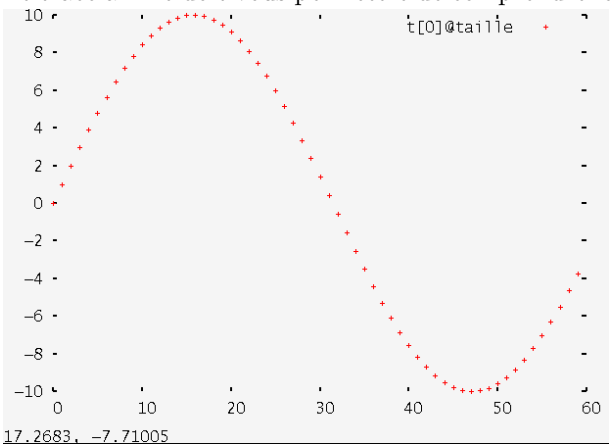


6. **Affichage graphique des données**
   (a) Relancez le programme sous le débogueur ;
   (b) Placez un breakpoint **temporaire** ligne 72 ;
   (c) Affichez les données du tableau passez en argument à la fonction : `graph plot t[0]@taille`
   (d) Tracez le contenu du tableau
   (e) Le contenu du graphique sera ré-actualisé à chaque nouveau franchissement breakpoint. Placez un breakpoint ligne 84. Pour que ce breakpoint soit franchi automatiquement, il faut l'enrichir d'une instruction `cont`, (clic droit sur le breakpoint) :



[2]Contrainte : toutes les variables utilisées dans l'expression doivent être connues au moment de l'interrogation

(f) Le tracé animé doit vous permettre de comprendre le bug présent dans le programme.



4.3 Debugging à distance

Plutôt que d'avoir à se connecter sur une machine pour lancer le débogage, il est possible de se connecter à distance à un debogueur. Cela permet par exemple d'avoir le client (debugger) graphique qui tourne sur une machine et le vrai debogage via un serveur. Nous illustrons via un exemple simple sur gdb

1. Installez gdbserver `sudo apt-get install gdbserver`
2. Lancez `gdbserver why_i localhost:2000`. Le serveur attend les connexions
3. Dans un autre terminal, lancez `gdb`. Il faut maintenant le paramétrer pour qu'il accède au serveur
4. Tapez `target remote localhost:2000` gdb vous indique qu'il n'a pas accès à la table des symboles, stockée dans l'exécutable
5. Tapez `file why_i`. Cela a pour effet de charger la table des symboles dans le débogueur, rendant le source accessible.
6. Placez un breakpoint au début du programme
7. Tapez `cont` et observez le comportement du serveur.