Information, Calcul, Communication (partie programmation) : Gestion des erreurs

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Objectifs du cours d'aujourd'hui

Apprendre à gérer les erreurs dans vos programmes :

- à anticiper les erreurs des utilisateurs de votre programme, voire d'autres programmeurs utilisant vos fonctions.
  ☞ utilisation des « exceptions »

-  **[optionnel, mais pratique pour vous !]**
  à trouver et corriger vos propres erreurs
  ☞ utilisation d'un dévermineur

Jean-Cédric Chappelier
& Jamila Sam

Gestion des erreurs

Les **exceptions** permettent d'anticiper les erreurs qui pourront potentiellement se produire lors de l'utilisation d'une portion de code.

Exemple : on veut écrire une fonction qui calcule l'inverse d'un nombre réel quand c'est possible :

f
entrée : x sortie : $1/x$
Si $x = 0$ *erreur* ***Sinon*** *retourner* $1/x$

```
double f(double x) {
  if (x == 0.0) {
    // ????
  } else {
    return 1.0 / x;
  }
}
```

☞ Que faire **concrètement** en cas d'erreur ?

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Gestion des erreurs (2)

① retourner une valeur choisie à l'avance :

```
double f(double x) {
  if (x != 0.0) return 1.0 / x;
  else return numeric_limits<double>::max();
}
```

Mais cela

1. **n'indique pas** à l'utilisateur potentiel qu'il a fait une erreur
2. retourne de toutes façons un **résultat inexact** ...
3. suppose une **convention arbitraire** (la valeur à retourner en cas d'erreur)

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Gestion des erreurs (3)

② afficher un message d'erreur
mais que retourner effectivement en cas d'erreur ?...
on retombe en partie sur le cas précédent

```
double f(double x) {
  if (x != 0.0) return 1.0 / x;
  else {
    cerr << "Erreur d'utilisation de la fonction f :"
         << "division par 0"
         << endl;
    return numeric_limits<double>::max();
  }
}
```

De plus, cela est **très mauvais** car produit de gros effets de bord :
modifie `cerr` alors que ce n'est pas du tout dans le rôle de `f`

Pensez par exemple au cas où l'on veut utiliser `f` dans un programme avec une interface graphique...
On ne veut alors plus utiliser `cerr` (mais plutôt ouvrir une fenêtre d'alerte par exemple).

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Gestion des erreurs (4)

③ retourner un code d'erreur :

```
int f(double x, double& resultat) {
  if (x != 0.0) {
      resultat = 1.0 / x;
      return PAS_D_ERREUR;
  }
  else return ERREUR_DIV_ZERO;
}
```

constantes définies plus haut dans le programme

Cette solution est déjà **beaucoup mieux** car elle laisse à la fonction qui appelle `f` le soin de décider quoi faire en cas d'erreur.

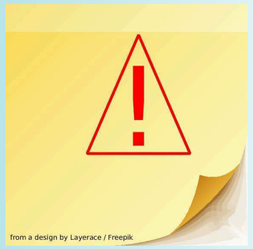
Cela présente néanmoins l'inconvénient d'être assez lourd à gérer :

- écriture peu intuitive :
  `if (f(x,y) == PAS_D_ERREUR) ...`
  au lieu de
  `y=f(x);`
- encore pire en cas d'appel d'appel.... ...d'appel de fonction.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Exceptions

Il existe une solution permettant de généraliser et d'assouplir cette dernière solution : déclencher une **exception**



☞ mécanisme permettant de prévoir une erreur à un endroit
et de **la gérer à un autre endroit**

Principes :

- lorsque qu'une erreur a été détectée à un endroit, on la signale en « lançant » un objet contenant toutes les informations que l'on souhaite transmettre sur l'erreur
  (« lancer » = créer un objet disponible pour le reste du programme)
- à l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut « attraper » l'objet « lancé » (« attraper » = utiliser)
- si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.

Un tel mécanisme s'appelle une exception.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Exceptions (2)

Avantages de la gestion des exceptions par rapport aux codes d'erreurs retournés par des fonctions :

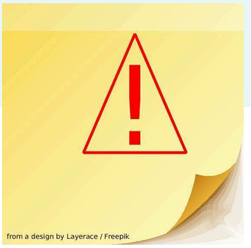
- écriture plus facile, plus intuitive et plus lisible
- la propagation de l'exceptions aux niveaux supérieurs d'appel (fonction appelant une fonction appelant ...) est fait **automatiquement**

  plus besoin de gérer obligatoirement l'erreur au niveau de la fonction appelante
- une erreur peut donc se produire à n'importe quel niveau d'appel, elle sera toujours reportée par le mécanisme de gestion des exceptions

Attention ! Si une erreur peut être gérée localement, le faire et **ne pas** utiliser le mécanisme des exceptions.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Syntaxe de la gestion des exceptions



On cherche à remplir 3 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif aux erreurs) un moyen de gérer leurs erreurs

On a donc 3 mots du langage C++ dédiés à la gestion des exceptions :

`throw` indique l'erreur (c.-à-d. « lance » l'exception)
`try` indique un bloc réceptif aux erreurs
`catch` gère les erreurs associées (c.-à-d. les « attrape »)

Notez bien que :

- l'indication des erreurs (`throw`) et leur gestion (`try`+`catch`) sont à des endroits bien séparés dans le code ;
- chaque bloc `try` possède son/ses `catch` associé(s) ;
- un `catch` est toujours lié à un (et un seul) `try`.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

throw

`throw` est l'instruction qui **signale l'erreur** au reste du programme.

Syntaxe : `throw` *`expression`*`;`

l'expression peut être de tout type : c'est le résultat de son évaluation qui est « lancé » au reste du programme pour être « attrapé »

Exemples :

```
throw 21;  // "lance" un entier

throw "quelle erreur !"s;  // "lance" une string

struct Erreur {
    int code;
    string message;
};
...
Erreur faute({ PAS_D_ERREUR, "pas d'erreur"s });
...
faute = { DIV_ZERO, "Division par 0"s };
throw faute;
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

`throw` (2)

`throw`, en « lançant » une exception, interrompt le cours normal d'exécution et :

- saute au bloc `catch` du bloc `try` directement supérieur (dans la pile des appels), si il existe ;
- quitte le programme (« abort ») si l'exécution courante n'était pas dans au moins un bloc `try`.

Exemple :

```
try {
  ...
  f(); // appel contenant un throw int
  ...
}
catch (int i) {
  ...
}
...
```

En cas d'erreur, saute ici

En cas d'erreur, ce code n'est pas exécuté

Ce code est toujours exécuté

try

`try` (*lit.* « essaye ») introduit un **bloc réceptif aux exceptions** lancées par des instructions, ou des fonctions appelées à l'intérieur de ce bloc (ou même des fonctions appelées par des fonctions appelées par des fonctions... ... ... à l'intérieur de ce bloc)

Exemple :

```
try {
 ...
 y = f(x); // f pouvant lancer une exception
 ...
}
```


Jean-Cédric Chappelier
& Jamila Sam

`catch`

`catch` est le mot-clé introduisant un **bloc dédié à la gestion** d'une ou plusieurs **exceptions**.

Tout bloc `try` doit toujours être suivi d'au moins un bloc `catch` gérant les exceptions pouvant être lancées dans ce bloc `try`.

Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (« `Aborted` »).

Syntaxe :

```
catch (type nom) {
  ...
}
```

intercepte toutes les exceptions de type `type` lancées depuis le bloc `try` directement précédent

(Note : un bloc `catch` ne peut venir qu'après un bloc `try` ou les autres blocs `catch` d'un bloc `try`)

Exemple d'utilisation de `catch`

```
try {
 ...
  z = f(...); // f() peut lancer des exceptions de type string ou int
 ...
}

// attrape les exceptions lancées de type string
catch(string const& erreur) {
  cerr << "Erreur : " << erreur << endl;
}

// attrape les exceptions lancées de type int
catch(int erreur) {
 cerr << "Avertissement : je n'aurais pas du avoir"
      << " la valeur "
      << erreur
      << endl;
}
```


Jean-Cédric Chappelier
& Jamila Sam

`catch` (flot d'exécution 1/3)

Un bloc `catch` n'est exécuté **que** si une exception de type correspondant a été lancée depuis le bloc `try` correspondant.

Sinon le bloc `catch` est simplement ignoré.

Si un bloc `catch` est exécuté, le déroulement continue ensuite normalement **après** ce bloc `catch` (ou après le dernier des blocs `catch` du même bloc `try`, lorsqu'il y en a plusieurs).

En aucun cas l'exécution ne reprend après le `throw` !

Jean-Cédric Chappelier
& Jamila Sam

`catch` (flot d'exécution 2/3)

Exemple :
en cas d'erreur (lancement d'une exception) :

En cas d'erreur, ce code n'est pas exécuté

```
try {
    ...
    z = f(x); // appel contenant un throw int
    ...
}
catch (int i) {
    ...
    ...
    ...
}
...
```

En cas d'erreur, saute ici

puis on continue ensuite ici.

`catch` (flot d'exécution 3/3)

Exemple :
si il n'y a pas d'erreur (**pas** de lancement d'exception) :

```
try {
    ...
    z = f(x); // appel contenant un throw int
    ...
}
catch (int i) {
    ...
    ...
    ...
}
...
```

S'il n'y a *pas* d'erreur, ce code **est** exécuté...

... puis on saute ici

Exemple (1/3)

```
#include <iostream>
#include "mesures.h"
// etc.
using namespace std;

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  plot_temp_inverse(temperatures);
  return 0;
}

void plot_temp_inverse(Mesures const& temp) {
  for (double t : temp) {
    plot(inverse(t));
  }
}

double inverse(double x) {
  return 1.0/x;
}
```


Jean-Cédric Chappelier
& Jamila Sam

Exemple (2/3)

```
...
using namespace std;

constexpr int DIVZERO(33);

void plot_temp_inverse(Mesures const&);
double inverse(double);

...

double inverse(double x) {
  if (x == 0.0) throw DIVZERO;
  return 1.0/x;
}
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

Exemple (3/3)

```
...
int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  try {
    plot_temp_inverse(temperatures);
  }
  catch (int i) {
    if (i == DIVZERO) {
      cerr << "Courbe des températures erronée" <<endl;

      /* on fait quelque chose, par exemple refaire
       * les mesures, mais à ce stade le programme
       * n'est pas encore stoppé.
       */
    }
  }
  return 0;
}
...
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

`catch` (Remarques)

Notes :

- « `catch(...)` » permet d'intercepter n'importe quel type d'exceptions
  mais, dans le cas où il y a plusieurs `catch` associés à un même `try`,
  « `catch(...)` » doit être le dernier.

-  comme pour les fonctions, on préfèrera passer les exceptions de type complexe par *références constantes* :

```
catch (Erreur const& e)
```


Jean-Cédric Chappelier
& Jamila Sam

« Relancement »

Une exception peut être **partiellement traitée** par un bloc `catch` et attendre un traitement plus complet ultérieur (c'est-à-dire à un niveau supérieur).

Il suffit pour cela de **« relancer » l'exception** au niveau du bloc n'effectuant que le traitement partiel.
(Il faudra bien sûr pour cela que l'appel à ce bloc `catch` soit lui-même dans un autre bloc `try` à un niveau supérieur).

Pour « relancer » une exception, il suffit simplement d'écrire `throw` (tout seul, c.-à-d. sans argument).

Exemple :

```
catch (int erreur) {
  // traitement partiel :
  cerr << "Hmm... pour l'instant je ne sais pas trop "
       << "quoi faire" << endl
       << "avec l'erreur " << erreur << endl;
  // relance l'exception reçue :
  throw;
}
```

Exemple, avec traitement partiel

```
int main() {
  ...
  try {
    plot_temp_inverse(temperatures);
  }
  catch (int i) {
    if (i == DIVZERO) {
      cerr << "Courbe des températures erronée" <<endl;

      // effectue ici un traitement de plus haut niveau
      ...
    }
...
void plot_temp_inverse(Mesures const& t) {
  for (size_t i(0); i < t.size(); ++i) {
    try {
      plot(inverse(t[i]));
    }
    catch (int j) {
      /* Traiter partiellement le problème et relancer l'exception.
       * Cette partie du programme peut par exemple signaler
       * l'indice de la valeur erronée.
       */
      cerr << "Problème avec la température d'indice " << i << endl;
      throw;
    }
  }
```

Exemple complet avec reprise (1/3)

```
...
using namespace std;

const int DIVZERO(33);

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main()
{
  Mesures temperatures;
  constexpr unsigned int MAX_ESSAIS(2);
  unsigned int nb_essais(0);
  bool restart(false);

  do {
    ++nb_essais; restart=false;
    acquerir_temp(temperatures);
    try {
      plot_temp_inverse(temperatures);
    }
```

Exemple complet avec reprise (2/3)

```
    catch (int i) {
      if (i == DIVZERO) {
        if (nb_essais < MAX_ESSAIS) {
          cout << "Il faut re-saisir les valeurs" << endl;
          restart = true;
        } else {
          cout << "Il y a déjà eu au moins " << MAX_ESSAIS
               << " essais." << endl;
          cout << " -> abandon" << endl;
        }
      } else {
        cout << "Ne sais pas quoi faire -> abandon" << endl;
      }
    }
  } while (restart);

  return 0;
}

void plot_temp_inverse(Mesures const& t)
{
  for (size_t i(0); i < t.size(); ++i) {
```

Exemple complet avec reprise (3/3)

```
    // Exemple de traitement local partiel du problème
    // (ce n'est pas obligatoire).
    try {
      plot(inverse(t[i]));
    }
    catch (int j) {
      cerr << "Erreur : ";
      if (j == DIVZERO) {
        cerr << "la valeur d'indice " << i << " est nulle.";
      } else {
        cerr << "autre problème avec la valeur d'indice " << i;
      }
      cerr << endl;
      throw;
    }
  }
}

double inverse(double x)
{
  if (x == 0.0) throw DIVZERO;
  return 1.0/x;
}
```

Exception lancée par `new`

`new` (allocation dynamique de pointeur), retourne une exception de type `bad_alloc` (défini dans la bibliothèque « `new` ») si l'allocation dynamique ne se passe pas correctement.

Il est donc conseillé d'écrire par exemple :

```
#include <new>

try {
   ...
   ptr = new ...;
   ...
}
catch (std::bad_alloc const& e) {
    cerr << "Erreur : plus assez de mémoire !" << endl;
    exit 1;
}
```


Jean-Cédric Chappelier
& Jamila Sam

Exceptions

`throw` *`expression`*`;` lance l'exception définie par l'expression

`try { ... }` introduit un bloc sensible aux exceptions

`catch (`*`type nom`*`) { ... }` bloc de gestion de l'exception

Tout bloc `try` doit toujours être suivit d'un bloc `catch` gérant les exceptions pouvant être lancées dans ce bloc `try`.

Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (« `Aborted` »).

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Plan

- Gestion des exceptions
-  Déverminage 

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Erreurs en programmation

Il existe plusieurs types d'erreurs :

① erreurs de syntaxe : le programme est mal écrit et le compilateur ne comprend pas ce qui est écrit.
Erreurs relativement faciles à trouver : le compilateur signale le problème, indiquant souvent l'endroit de l'erreur.

Jean-Cédric Chappelier
& Jamila Sam

Erreurs en programmation

Il existe plusieurs types d'erreurs :

① erreurs de syntaxe : le programme est mal écrit et le compilateur ne comprend pas ce qui est écrit.
Erreurs relativement faciles à trouver : le compilateur signale le problème, indiquant souvent l'endroit de l'erreur.

② erreurs d'implémentation : la syntaxe du programme est correcte (il compile), mais ce que fait le programme est erroné
(par exemple une division par zéro se produit, ou une variable n'a pas été initialisée correctement).
Ces erreurs ne se détectent qu'à l'exécution du programme, soit par un arrêt prématuré (p.ex. cas de la division par zéro), soit par des résultats erronés (p.ex. cas de la mauvaise initialisation).

Jean-Cédric Chappelier
& Jamila Sam

Erreurs en programmation

Il existe plusieurs types d'erreurs :

③ erreurs d'algorithme : l'algorithme implémenté ne fait pas ce que l'on croit (ce qu'il devrait).
Assez proche du cas précédent, mais ici, c'est plus la méthode globale qui est erronée, plutôt qu'une étourderie ou un manque de précision dans une des étapes du codage de l'algorithme.
Il existe pour ce type d'erreurs des tests formels permettant de trouver les erreurs.
Mais ce genre de techniques est trop complexe pour être abordé dans ce cours.

Jean-Cédric Chappelier
& Jamila Sam

Erreurs en programmation

Il existe plusieurs types d'erreurs :

③ erreurs d'algorithme : l'algorithme implémenté ne fait pas ce que l'on croit (ce qu'il devrait).
Assez proche du cas précédent, mais ici, c'est plus la méthode globale qui est erronée, plutôt qu'une étourderie ou un manque de précision dans une des étapes du codage de l'algorithme.
Il existe pour ce type d'erreurs des tests formels permettant de trouver les erreurs.
Mais ce genre de techniques est trop complexe pour être abordé dans ce cours.

④ erreurs de conception : ici c'est carrément l'approche du problème qui est erronée, souvent en raison d'hypothèses trop fortes ou non explicitées.
Elles relèvent du domaine de l'ingénierie informatique (le « génie logiciel »), et ne seront pas traitées dans ce cours.

Jean-Cédric Chappelier
& Jamila Sam

Erreurs en programmation

Il existe plusieurs types d'erreurs :

① erreurs de syntaxe
② erreurs d'implémentation
③ erreurs d'algorithme
④ erreurs de conception

Nous nous intéressons ici aux erreurs d'implémentation (②)
et d'algorithme (③), mais d'un point de vue pratique :
c'est-à-dire mise en œuvre de **procédures de déverminage**.

On s'intéresse donc à trouver vos erreurs **lors de l'exécution**.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Dévermineur

L'utilisation d'un « dévermineur » (« debugger » en anglais) permet d'ausculter en détail l'exécution d'un programme, et en particulier

- localiser les erreurs
- exécuter un programme pas à pas
- suivre la valeur de certaines variables

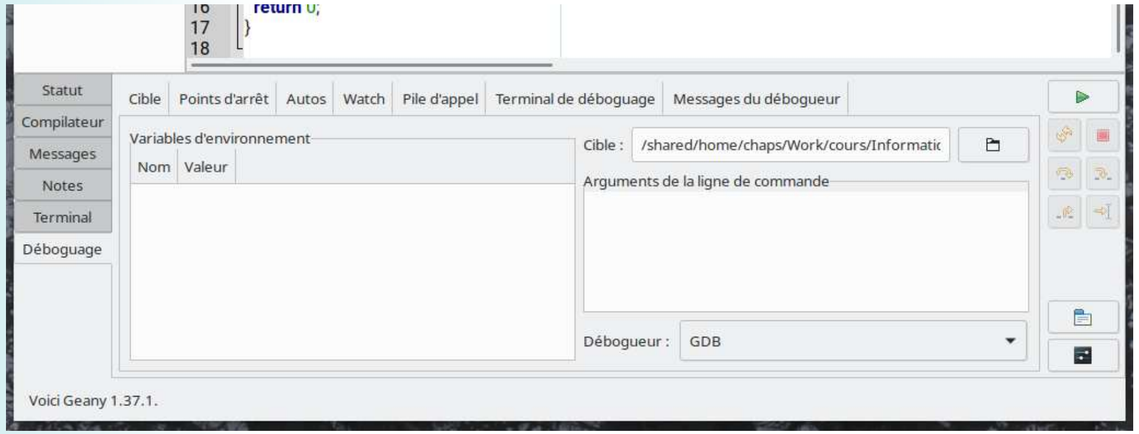
Jean-Cédric Chappelier
& Jamila Sam

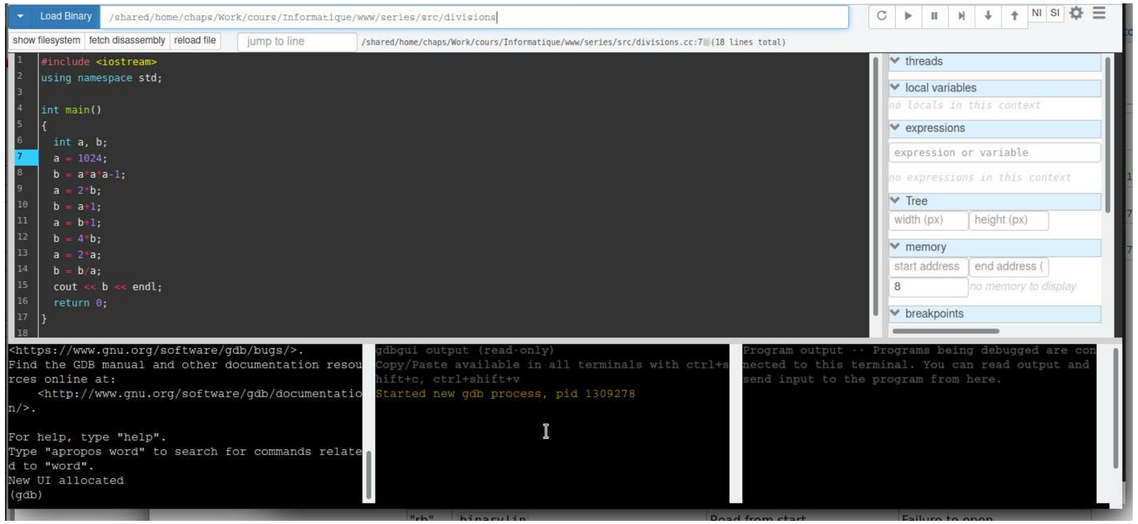
gdb

① Pour pouvoir déboguer un programme, il faut le **compiler avec l'option** `-g`
Cela indique au compilateur de rajouter des informations supplémentaires dans le programme, utiles au dévermineur.

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

② Ensuite il faut lancer le **dévermineur**.
Très souvent, celui-ci est intégré dans une interface graphique.
Par exemple, nous pouvons utiliser le dévermineur `gdb` au travers de son intégration dans Geany, ou utiliser `gdbgui`, ou autre :



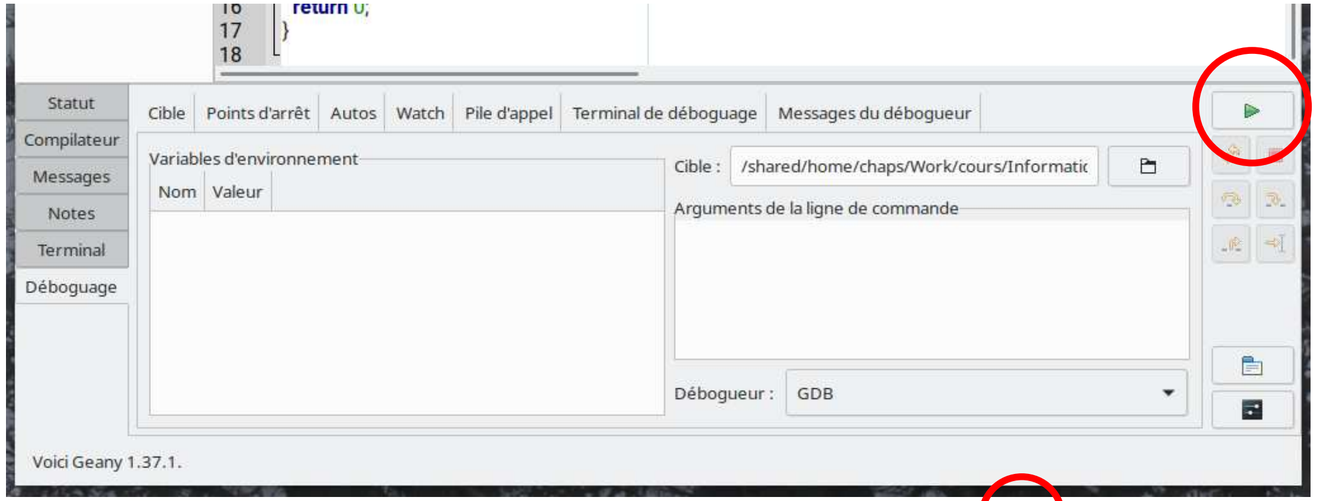


Jean-Cédric Chappelier
& Jamila Sam

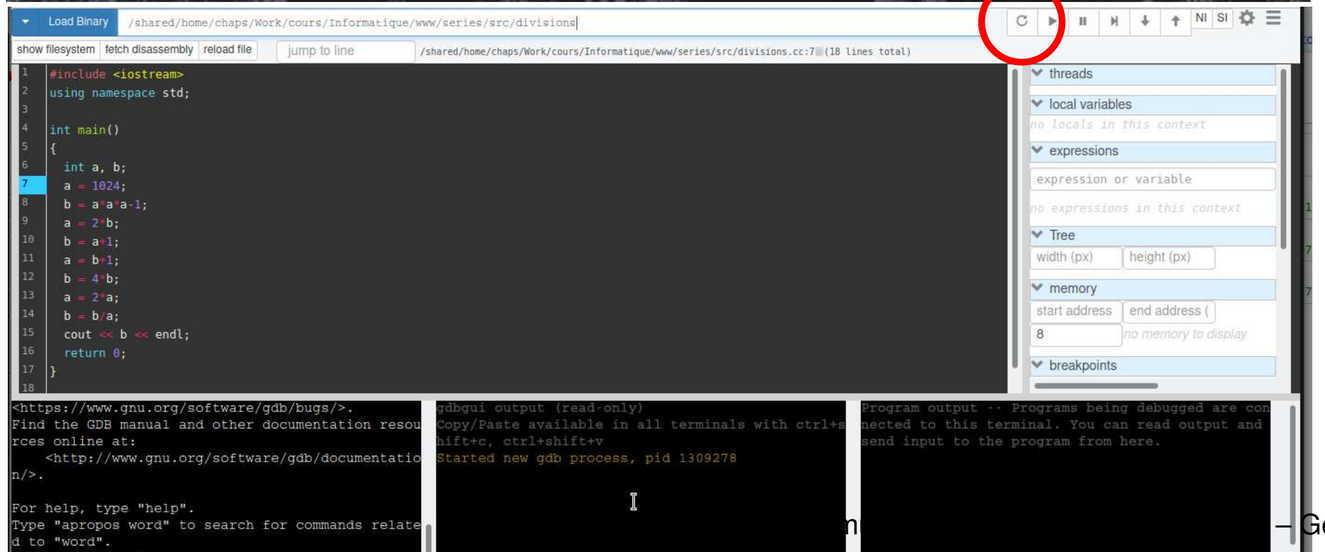
gdb : run

③ Il faut ensuite exécuter le programme à corriger/étudier.
Cela se fait à l'aide d'un bouton, ou de la commande `run` dans `gdb` :
`run` ou `run` *`arguments`*

Geany :

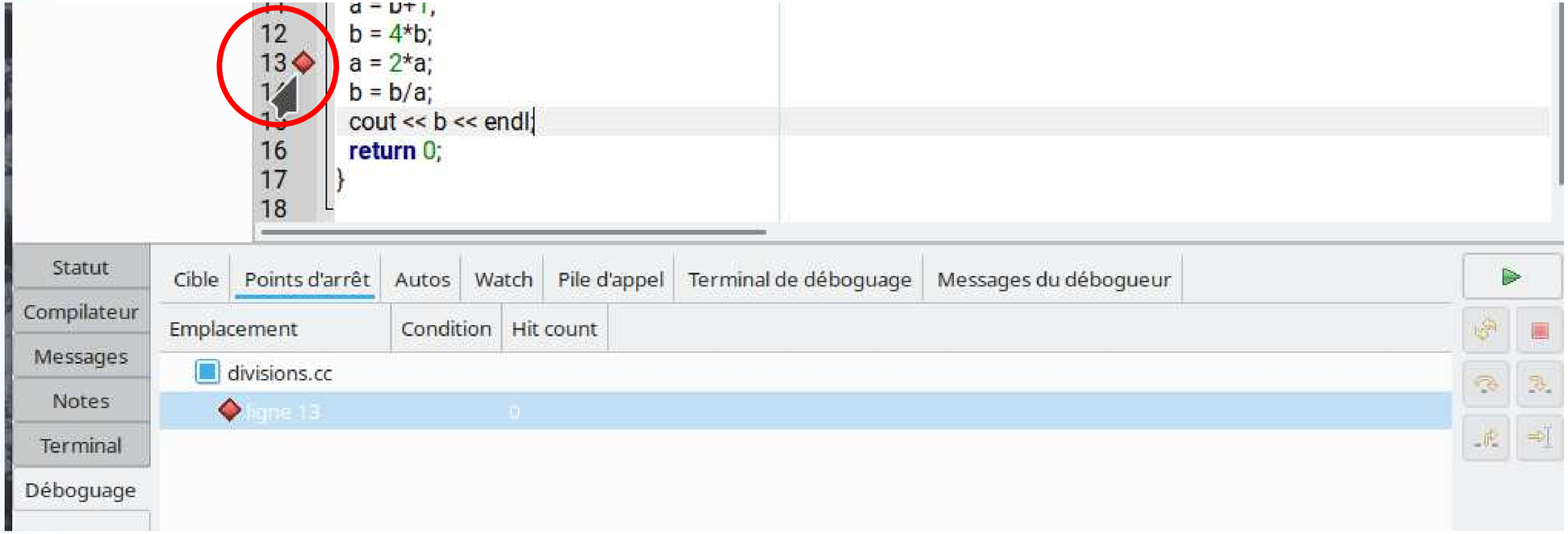


gdbgui :



gdb : breakpoints

④ On peut décider de suspendre l'exécution du programme à des endroits précis en y plaçant des breakpoints (points d'arrêt)



Jean-Cédric Chappelier
& Jamila Sam

gdb : breakpoints

```
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;
```

⑤ Une fois le programme stoppé à un point d'arrêt, on peut continuer à l'exécuter

- soit *pas à pas* au **même niveau** que le point d'arrêt (« *step over* » ; commande gdb `next`) :
  exécute les pas de programme sans « descendre » dans les fonctions appelées ;
- soit *pas élémentaire à pas élémentaire* (« *step into* » ; commande gdb `step`) :
  exécute les pas élémentaires de programme et donc **entre dans les fonctions appelées** ;
- soit *en continu* jusqu'au prochain point d'arrêt (commande gdb `cont`).

Jean-Cédric Chappelier
& Jamila Sam
EPFL

gdb : print & display

⑥ On peut regarder le contenu d'une variable

- soit en mettant la souris dessus

- soit à l'aide de la commande gdb `print`
  qui affiche la valeur de la variable à ce moment là

```
(gdb) print x
$2 = 12
(gdb)
```

- soit à l'aide de la commande `display`.
  La valeur de la variable est alors affichée à chaque pas de programme.

```
(gdb) display x
(gdb) next
1: x = 12
```

Pour plus de détails sur l'utilisation pratique des débogueurs, voir l'exercice correspondant dans la série d'exercices.

Jean-Cédric Chappelier
& Jamila Sam

Déverminage (avec gdb)

Pour utiliser un programme de déverminage, compiler avec l'option `-g`

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

Lancer le dévermineur : `gdb monprogramme` ; puis : `layout src` (dans gdb)

Démarrer mon programme dans `gdb` : `run` ou `run` *`arguments`*

Suspendre l'exécution du programme à des endroits précis : breakpoints :

`break` *`line`*

`break` *`function`*

Exécuter pas à pas : `next` ou `step`

Regarder le contenu d'une variable :

- soit `print` *`nom_variable`*
- soit `display` *`nom_variable`*
  La valeur de la variable est alors affichée à chaque pas de programme.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Ce que j'ai appris aujourd'hui

- à gérer les erreurs pouvant se produire lors de l'exécution (par l'utilisation du mécanisme d'**exception**)
  - ☞ je peux maintenant rendre mon code plus robuste
-  **[optionnel, mais bien pratique !]**
  à utiliser un **programme de déverminage**
  - ☞ je peux maintenant plus facilement corriger mes erreurs

Jean-Cédric Chappelier
& Jamila Sam
EPFL