TD1B

VARIABLES, OPÉRATEURS ET ERREURS DE COMPILATION

Compétences visées

- Déboguer les erreurs de compilation simples (syntaxe, types)
- Manipuler les types de base (int, float, char) et les opérateurs

Table des matières

EXERCICE 1: Corriger un programme

Corrige les erreurs du code ci-dessous:

```c
#include <stdio.h>
int main() {
  int caractere printf(« Donner une valeur / n »);
  scanf(«% c », caractere);
  verification(caractere);
  return 0;
}
```

EXERCICE 2: Table de multiplication

Écrire un programme qui demande un entier positif (entre 1 et 1000) et qui affiche sa table de multiplication.

```
Indiquer un nombre entier entre 1 et 1000: 7
 1 x 7 = 7
 2 x 7 = 14
 3 x 7 = 21
 4 x 7 = 28
 5 x 7 = 35
 6 x 7 = 42
 7 x 7 = 49
 8 x 7 = 56
 9 x 7 = 63
```

EXERCICE 3: Un rectangle

(1) Écrire un programme qui demande au clavier la longueur et la largeur d'un rectangle et qui affiche le périmètre et l'aire.

```
Indiquez la largeur en m: 2.5
Indiquez la longueur en m: 7
Le périmètre du rectangle est 19.00 m.
L'aire du rectangle est 17.50 m.
```

EXERCICE 4: Boucles `while`

(2) Écrivez un programme qui affiche les entiers de 1 à 10 en utilisant une boucle `while`.

(3) Modifiez le programme pour qu'il affiche uniquement les entiers pairs entre 1 et 20.

(4) Écrivez une fonction `estPair` qui prend un entier en paramètre et retourne 1 s'il est pair, 0 sinon. Utilisez cette fonction dans un programme qui affiche les entiers pairs entre 1 et 20.

EXERCICE 5: Fonctions et opérateurs logiques

(5) Écrivez une fonction `estDivisible` qui prend deux entiers en paramètre et retourne 1 si le premier est divisible par le second, 0 sinon.

(6) Utilisez cette fonction pour afficher tous les diviseurs d'un entier saisi par l'utilisateur.

(7) Écrivez un programme qui lit un entier et affiche s'il est premier.

EXERCICE 6: ACLASSER

EXERCICE 7: Utiliser le débogueur sur un programme simple

(8) Écrire un programme calculant l'aire d'un cercle. Pour cela, le programme demande à l'utilisateur de saisir une valeur pour le diamètre du cercle.

Dans le fichier `main.c`, coder une fonction `rayon` qui renvoie le rayon du cercle à partir du diamètre.

Le programme calcule ensuite l'aire du cercle (en utilisant la fonction rayon) puis affiche la valeur.

```
Valeur du diamètre (m): 2
La surface vaut 3.141590 m2
```

Astuce

Nous allons utiliser l'outil debug de `VSCode`.

https://code.visualstudio.com/docs/cpp/cpp-debug

Cet outil permet d'avoir des détails sur le déroulement du programme en particulier les valeurs des variables en mémoire et la hiérarche des appels de fonctions(*call stack*).

Sous `VSCode`, pour lancer le débogage, il suffit de:

- Créer un ou plusieurs points d'arrêt. Pour cela, il suffit de cliquer avec le bouton gauche de la souris dans la zone à côté du numéro de ligne. On effectue la même opération pour retirer un point d'arrêt.
- Lancer le programme avec le débogage(Bouton Play avec un insecte(*bug*) en haut de l'écran).

Une fenêtre s'affiche à gauche pour afficher l'état du programme au point d'arrêt. Et un menu s'affiche en haut de l'écran pour controller l'avancée de l'exécution du programme.

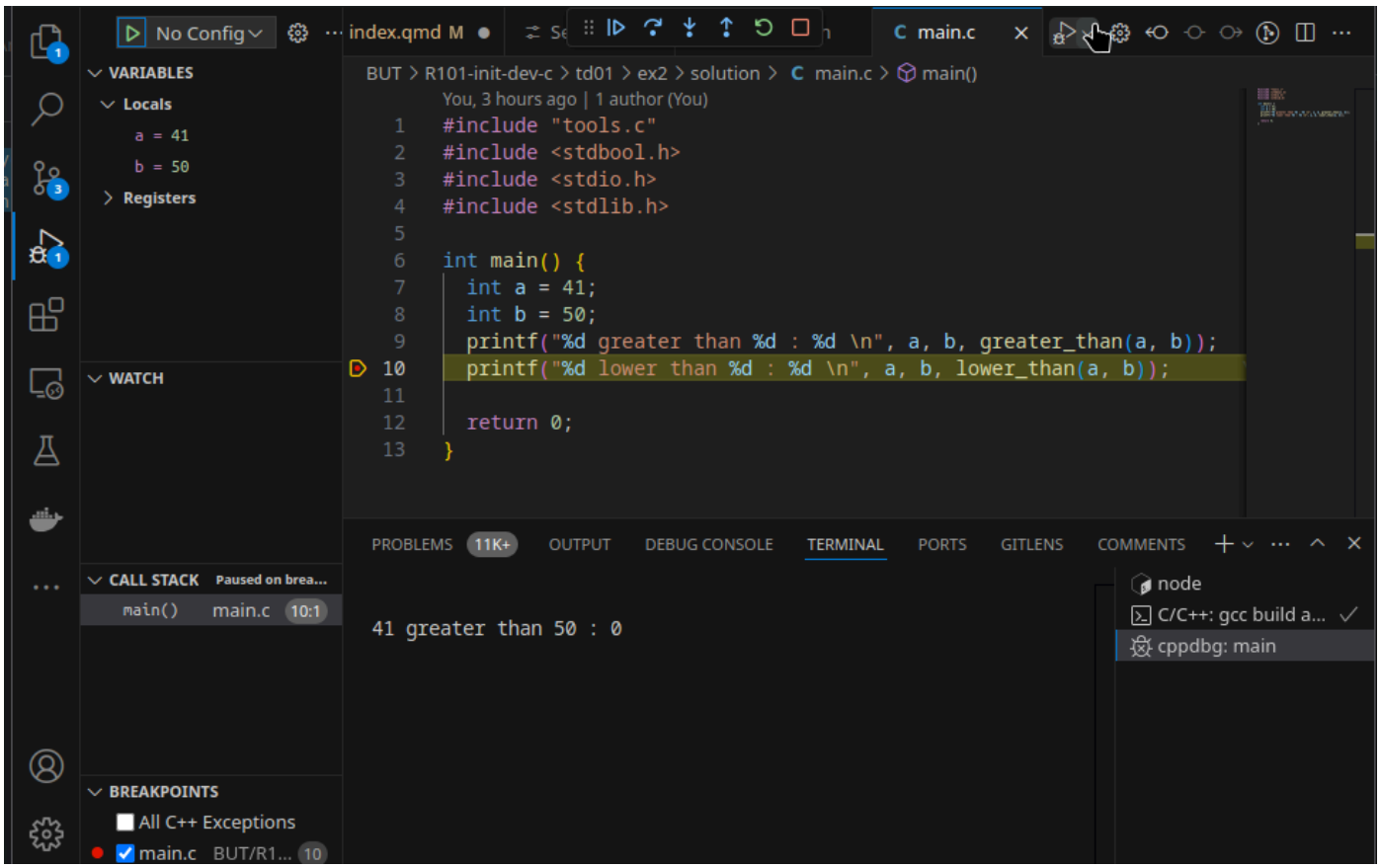

Figure 1. – Débogage dans VSCode

(9) Créer les trois points d'arrêt suivants:

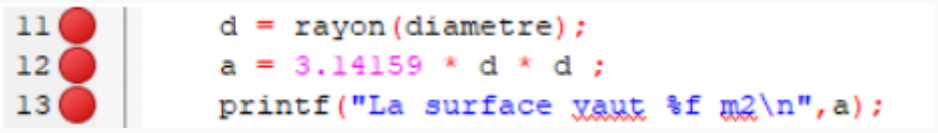

Figure 2. – Points d'arrêt du programme

Ensuite, on clique sur Debug/continue. Le programme s'exécute jusqu'au point d'arrêt. La fenêtre nous demande la valeur du diamètre puis le programme s'arrête et affiche la valeur dans watches. La pile d'appel (*call stack*) permet de voir que le programme est dans la fonction rayon.

Les points d'arrêt successifs permettent de vérifier que les valeurs des variables locales sont correctes après chaque ligne exécutée.

> ! Important
>
> Les outils de débogage permettent d'avoir des informations sur le code sans ajouter des instructions supplémentaires de `c`.

Entraîner vous à placer des points d'arrêt et à utiliser les fenêtres de *WATCH* et de *CALL STACK*.

(10) Réorganiser le code pour placer la fonction `rayon` dans un fichier `geom.c` et ajouter le fichier de prototype `geom.h`.

EXERCICE 8: Conversions implicites de types (DÉBRANCHÉ)

Expressions	Traduction en français	Valeur numérique	Valeur logique (vrai/faux)
`!(x >= 2)`	`!x`	`!y`	

Expliquez pourquoi les expressions suivantes ne doivent pas être utilisées

- `6 < x < 10`
- `x = 2`

EXERCICE 9: Conversion explicite de type (*casting*) (DÉBRANCHÉ)

Prédire sur le papier les valeurs des expressions `(float)i/j`, `(float)(i/j)` et `(float)i/(float)j` pour $i = 1$ et $j = 4$. Vérifier votre réponse

EXERCICE 10: Cast

Écrire un programme C qui lit un nombre réel et affiche sa partie fractionnaire en utilisant le casting.

```
Entrer un nombre reel:
3.14159
La partie fractionnaire de 3.141590 est 0.141590.
```

EXERCICE 11: `int` VS `unsigned int`

Observer l'effet de la conversion de type entre des valeurs `int` et `unsigned int`, en particulier dans le cas de valeurs négatives. Utiliser la fonction `sizeof` ainsi que les formats d'affichage `d`, `u`, `ld`, `lu`, `x`, `lx` (précédés du caractère `%`).

Même question avec les types `long` et `unsigned long`.

Exemples de sorties attendues:

1. Avec int et unsigned int.

```
size of int = 4
size of unsigned int = 4
Entrer un entier a positif: 1024
Vous avez entré la valeur a = 1024 = X400
et sa valeur unsigned est uns_a = 1024 = X400
Entrer un entier a négatif: -1024
Vous avez entré la valeur a = -1024 = Xfffffc00
et sa valeur unsigned est uns_a = 4294966272 = Xfffffc00
```

2. Avec `long` et `unsigned long`.

```
size of int = 4
size of unsigned int = 4
Entrer un entier a positif: 1024
Vous avez entré la valeur a = 1024 = X400
et sa valeur unsigned est uns_a = 1024 = X400
Entrer un entier a négatif: -1024
Vous avez entré la valeur a = -1024 = Xfffffffffffffc00
et sa valeur unsigned est uns_a = 18446744073709550592 = Xfffffffffffffc00
```

EXERCICE 12: Opérations et *overflow*

Dans un nouveau fichier `td02/ex5/main.c`, écrire et exécuter le programme ci-dessous:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  // entiers sur seize bits
  unsigned short int a = 65535;
  unsigned short int b = 0;
  printf("a vaut %i\n", a);
  a = a + 1;
  printf("a vaut %i\n", a);
  printf("b vaut %i\n", b);
  b = b - 1;
  printf("b vaut %i\n", b);
  return 0;
}
```

(11) Que fait ce programme? Quels sont les résultats inattendus?

Ce programme met en évidence le concept d'overflow (dépassement).

(12) Pour éviter cette situation, on utilise les instructions de test: if ... { } et if ... { } else { }. L'instruction de test sert à protéger les opérations. L'addition sera effectuée uniquement si a < 65535 (sa valeur maximale). Dans le cas contraire, on retournera le code d'erreur `1` dans la fonction `main`.

```
if (a < 65535) {
  a = a + 1;
} else {
  printf("Dépassement de capacité");
  return 1;
}
```

De la même manière, protéger la soustraction en introduisant un deuxième test.

(13) Sans utiliser les tests pour protéger les opérations, peut-on une autre solution qui permettrait d'« éviter » les deux overflow observés?