Hélène Collavizza, Stéphane Lavirotte, Erick Gallesio, Jean-Yves Tigli
Polytech'Nice – Sophia
SI3
2019-2020

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

1 Pourquoi un debugger ?

Tout développement logiciel est maintenant accompagné de son outil de debugging. Fini les programmes truffés d'affichages de variables (ex. `printf`) et d'entrées de caractères (ex. `getc`) à la recherche du n^{ième} bug de votre programme. Nous allons dans ce TD étudier les principales fonctionnalités d'un debugger :

- contrôle de l'exécution du programme : points d'arrêt, exécution pas à pas détaillée ou non (i.e. rentre ou non dans les routines)
- visualisation de l'état de la mémoire et des variables, observation d'expressions,
- fonctionnalités avancées : points d'arrêt conditionnels ou sur fonction.

Dans notre cas particulier, nous allons débugger des programmes écrits en C. Le débugguer standard est `gdb`, mais celui-ci est assez austère à utiliser : il s'utilise en ligne de commande et n'affiche pas le retour aux sources.

Nous vous demandons dans ce premier TD d'installer et d'utiliser *Visual Studio Code* (VSC) qui est un éditeur récent, avec les fonctionnalités attendues pour l'aide à l'édition de programmes C, mais surtout avec un debugger graphique, facile d'utilisation et offrant les fonctionnalités classiques d'un debugger citées ci-dessus[1]. Pour vous aidez à installer Visual Studio Code, vous pouvez vous aider de l'Annexe `1` de ce document.

Nous vous encourageons à utiliser VSC pour les TD suivants, mais libre à vous de faire autrement si vous êtes à l'aise avec un autre environnement qui fournit les mêmes fonctionnalités en termes de debugging.

2 Récupération d'un Espace de Travail (Workspace)

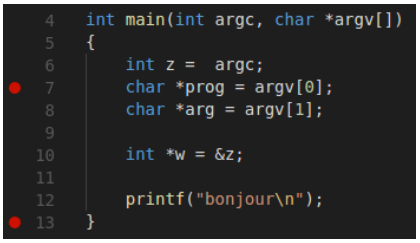
Pour éviter de perdre du temps avec les outils, nous vous fournissons un espace de travail avec les fichiers du TD et la configuration complète de l'outil Visual Studio Code. Pour l'utiliser, il vous suffit de télécharger le fichier suivant et de l'extraire dans un de vos répertoires :

`https://lms.univ-cotedazur.fr/mod/resource/view.php?id=75971`

Puis faire « **Ajouter un dossier à l'espace de travail** » (ou « *Add Folder to Workspace* ») et sélectionnez le répertoire que vous venez d'obtenir après la décompression du fichier zip fourni. Ce répertoire `td01` contient un sous-répertoire `.vscode` qui contient les fichiers de configuration pour l'outil Visual Studio Code. Si vous souhaitez en savoir plus sur la création de ces fichiers, vous pouvez vous reporter à l'Annexe `2`. Il contient aussi un ensemble de fichiers C avec lesquels nous allons travailler ainsi qu'un Makefile.

3 Illustration des fonctionnalités de base du debugger

Avant de lancez le debugger, aller dans le fichier `debugSimple.c` pour mettre un point d'arrêt sur les lignes 7 et 13 en cliquant à gauche de la ligne.

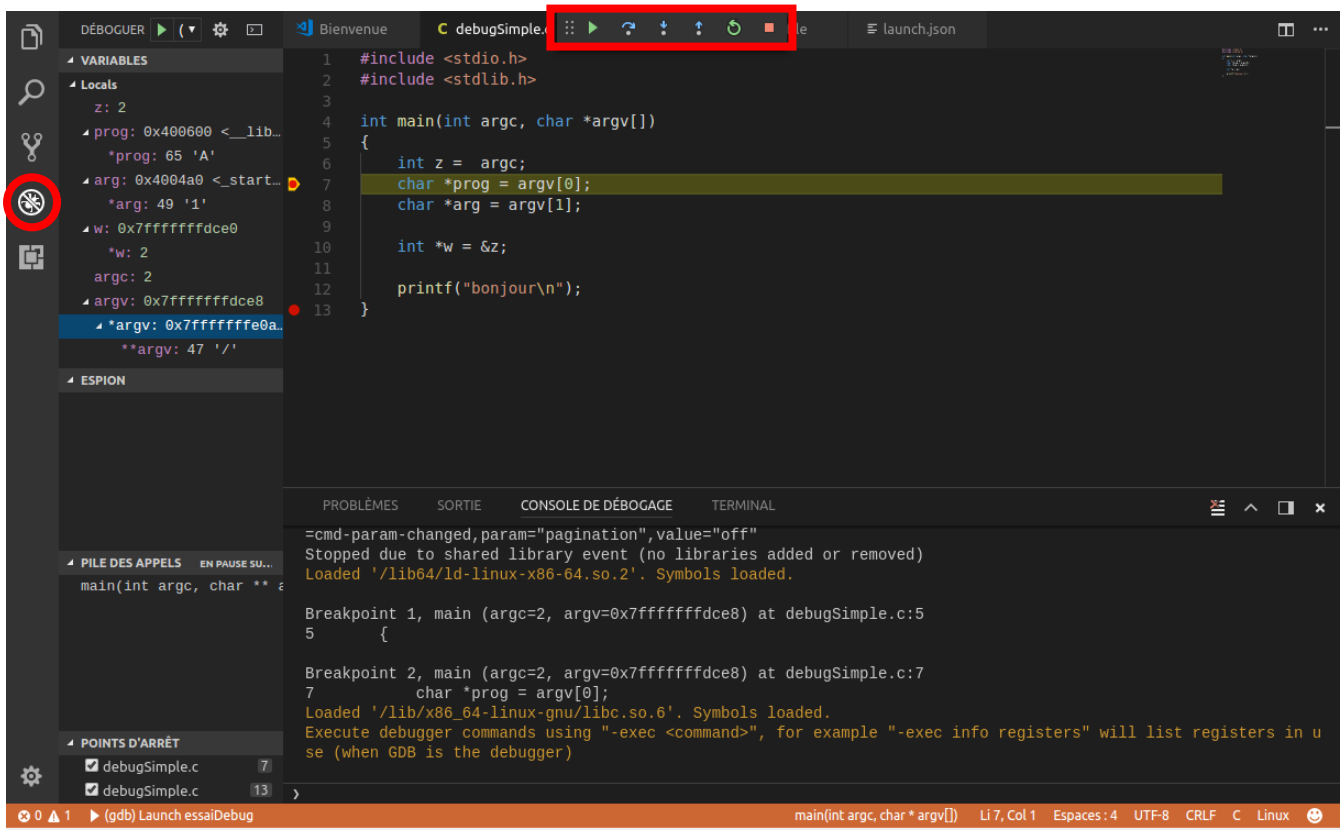


1 VSC est similaire à `sublime-text` mais celui-ci ne dispose pas d'un debugger assez sophistiqué pour illustrer notre propos.

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

Pour passer dans l'environnement de debugging, il vous suffit de cliquer sur le petit cafard dans le menu de gauche.

Pour lancer le debug, cliquez sur le triangle vert en haut à gauche (ou en bas à gauche). On obtient la fenêtre ci-dessous où l'on retrouve les fonctionnalités de base d'un debugger.



1 Gestion de l'exécution : groupe de boutons du haut (encadré rouge) avec de gauche à droite :
   - Continuer : exécution jusqu'au prochain point d'arrêt,
   - Pas à pas principal : exécution instruction par instruction sans entrer dans les sous-routines,
   - Pas à pas détaillé : exécution instruction par instruction en entrant dans les sous-routines,
   - Pas à pas sortant : sortir de l'exécution de la sous-routine courante,
   - Redémarrer, Arrêter
2 Contenu des variables : VARIABLES en haut à gauche, variables locales et globales s'il y en a
3 Observation d'expressions : ESPION
4 Pile des appels : la suite des appels de fonction en cours
5 Liste des points d'arrêt

La console de débogage en bas correspond à ce qui est exécuté par `gdb`. La ligne de commande permet de donner des instructions directement à gdb pour ceux qui sauraient le maitriser directement.

Une fenêtre « terminal » s'est ouverte pour afficher `stdout` et lire `stdin`.

La fenêtre TERMNAL vous permet d'accéder à un Shell.

Exercice n°1: Regardez le contenu de `argc` et `argv`. Est-ce cohérent avec le contenu de `launch.json` ? Quel est le lien entre le caractère / et 47 ?

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

Exercice n°2: Avancez pas à pas jusqu'à la ligne `11`. Observez les valeurs de `prog` et `arg`. Notez qu'une info bulle affiche leur contenu quand vous passez sur la variable dans l'éditeur.

Exercice n°3: On remarque que `*w = 2` ainsi que z, ce qui est normal puisque `*w=z`. On aimerait voir le contenu de `&w`. On peut pour cela ajouter un espion. Cliquer + dans la zone ESPION et tapez `&w`. Dépliez la valeur de `&w`. Comprenez-vous les relations entre : `&w`, `*&w` et `**&w` ?

Exercice n°4: Avancez jusqu'au dernier point d'arrêt et regardez la sortie dans le terminal.

4 Fonctionnalités avancées

Nous présentons ici quelques fonctionnalités avancées présentes dans la plupart des debugger.

- Observation d'une zone mémoire : placer un espion de la forme `x@n` permet d'observer le contenu de la mémoire de `n` cases à partir de `x` (la taille des cases est liée au type des données.
- Point d'arrêt conditionnel : clic droit dans la marge permet de choisir d'ajouter un point d'arrêt conditionnel en fonction d'une expression.
- Menu « Déboguer -> Nouveau point d'arrêt -> Point d'arrêt sur fonction » permet d'ajouter un point d'arrêt sur une fonction. Le debugger s'arrêtera à chaque appel de la fonction.
- Menu « Déboguer -> Nouveau point d'arrêt -> Point d'arrêt sur colonne » permet d'ajouter un point d'arrêt sur une ligne et une colonne particulière, là où se trouve le curseur. Utile par exemple pour tracer une expression complexe.
- Quand le debugger est lancé, dans la fenêtre de débogage en bas, il y a une invite de commandes qui permet de lancer les commandes de gdb.

Exercice n°5: Enlevez tous les points d'arrêt. Placez un point d'arrêt conditionnel sur la ligne `10` à condition que `z == 2`. Lancez le debug, vous devez constater que vous vous arrêtez bien.

Exercice n°6: Placez un espion sur `*argv@3`.

Exercice n°7: En ligne de commande du debugger, taper « `-exec info registers` », cela affiche le contenu des registres de la machine (Cf votre cours d'exécution des programmes ou https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture).

5 Compréhension de l'organisation de la mémoire

Nous allons maintenant travailler avec le source `memoire.c`. Compilez-le et observez les warnings du compilateur.

1. Placez un point d'arrêt sur chacune des lignes du programme.
2. Placez les espions suivants :
   - `&tab`
   - `&ii`
   - `*ii@4`
   - `&si`
   - `*si@8`
   - `&ci`
   - `*ci@16`
3. Tracez pas à pas jusqu'à avoir exécuté l'instruction `char *ci=tab;`. Que déduisez-vous de l'organisation mémoire ? Comment sont placés les octets à l'intérieur d'un `int` ?
4. Y a-t-il une différence entre `*ci@16` et `*si@8` ?

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

5. Tracez pas à pas jusqu'à la fin du programme et observez bien `ii`, `si` et `ci`. Pour quelle raison `ii++` et `si++` n'ont pas le même effet sur la valeur de `ii` et `si` ? De même, pourquoi `ii++` et `ci++` n'ont pas le même effet sur la valeur de `ii` et `ci` ?

6 Mise en pratique du debugging sur des cas concrets

Vous allez maintenant mettre en œuvre le debugger pour partir à la chasse au bug !

6.1 Remplir un tableau, c'est pourtant simple…

Soit le programme `stackCanary.c`. Si vous tentez de l'exécuter après compilation, vous obtiendrez le résultat suivant :

```
Okay Houston, we've had a problem here!
```

Si vous regardez le code, cela pose un problème car la variable `sig` est définie comme constante et n'est jamais modifiée dans le code…

Exercice n°8: Tentez de découvrir quel est le bug de ce programme. Aide : il peut être judicieux de stopper dans la boucle et de faire une exécution pas à pas jusqu'à la fin de la boucle (vous pourrez aussi utiliser un point d'arrêt conditionnel pour éviter de faire toutes les valeurs de `i`). Vous observez quand la valeur de la variable `sig` est modifiée et tenterez de découvrir pourquoi. Mais ne corrigez pas ce problème tout de suite.

`sig` est déclaré juste après le tableau `t`, donc contigu en mémoire. La boucle va jusqu'à `i = 10` (donc une valeur de trop), donc `t[i] = 0` va écrire dans la variable `sig` déclarée. Donc `sig` repasse à zéro d'où le message affiché.

Avant de corriger le bug que vous avez identifié, vous modifierez le `Makefile` pour supprimer l'option `-fno-stack-protector`. Après avoir recompilé le programme `stackCanary`, lors de son exécution, vous obtiendrez le message suivant :

```
*** stack smashing detected ***: <unknown> terminated
Abandon (core dumped)
```

En fait, le compilateur `gcc` introduit automatiquement une signature juste après la variable `t` (s'il n'y a pas l'option `-fno-stack-protector`). Cette signature (« Stack Canary ») permet de détecter à l'exécution un dépassement de buffer (source de nombreux problèmes de sécurité).

- https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries

Vous pouvez maintenant corriger le bug et vérifié qu'il n'y a plus de problème.

Pour de plus amples explications sur les raisons du message « `stack smashing detected` », vous pourrez lire les informations suivantes :

- https://stackoverflow.com/questions/1345670/stack-smashing-detected

Certains processeurs permettent de faire ces vérifications au niveau hardware :

- https://en.wikipedia.org/wiki/Tagged_architecture
- https://www.microsoft.com/en-us/research/uploads/prod/2019/07/Pointer-Tagging-for-Memory-Safety.pdf

6.2 Programme de tri

Mais le compilateur ne peut pas toujours vous aider à trouver vos erreurs de code… En voici un exemple.

Hélène Collavizza, Stéphane Lavirotte, Erick Gallesio, Jean-Yves Tigli
Polytech'Nice – Sophia
SI3
2019-2020

TD n° 1
Debugging de Programmes :
Mise en œuvre avec Visual Studio Code

L'exemple de programme `tri.c` présente le bug suivant : normalement, le programme tri doit trier et imprimer ses arguments numériquement, comme dans l'exemple suivant:

```
$ ./tri 8 7 5 4 1 3
     1 3 4 5 7 8
```

Cependant, avec certains arguments, cela ne marche pas :

```
$ ./tri 8000 7000 5000 1000 4000
      0 1000 4000 5000 7000
```

Quand on vous dit que les tests sont importants ! Et qu'il ne faut pas seulement tester avec un seul exemple… Dans le deuxième cas, bien que la sortie soit triée et contienne le bon nombre d'arguments, certaines valeurs sont remplacées par d'autres comme `8000` est remplacé par `0`. C'est étrange tout de même. Utilisons le débuggueur pour voir ce qui se passe.

Maintenant que vous avez pris en main le débogueur de programme C, vous pouvez vous mettre en chasse du problème qui fait que le programme de tri ci-dessus ne donne pas toujours le bon résultat. Une fois que vous aurez identifié le problème, expliquez pourquoi le problème n'apparaît pas forcément à tous les tests.

Exercice n°9: Quel algorithme de tri implémente la fonction `mysort` ?

L'algorithme de tri implémenté est le `shell_sort` : https://en.wikipedia.org/wiki/Shellsort

Exercice n°10: Placer des points d'arrêts et des observateurs d'expression. Aide : vous pourrez vous inspirer démarche utilisée pour l'exercice précédent. Il peut être judicieux d'observer les valeurs des cases mémoire après le tableau !

Exercice n°11: Corrigez maintenant le problème et recompilez avec ctrl-shift-b.

Le bug contenu dans le programme vient du fait que l'argument passé à l'algorithme de tri pour la taille du tableau à trier est trop grand d'un élément. En effet, le tableau contient argc-1 éléments et l'appel à mysort donne une taille de argc éléments. L'algorithme tri donc n+1 éléments au lieu des n éléments passés en paramètres. Or dans la n+1 case considérée, celle-ci peut contenir n'importe quelle valeur. Le tableau d'entiers alloué par l'instruction malloc récupère une zone mémoire qui peut contenir n'importe quelle valeur. Dès lors, si la n+1 ème valeur est supérieur au n arguments passés, on ne peut constater le bug.

Exercice n°12: Dans les programmes fournis dans l'archive, vous avec le programme `list.c`. Tentez de le déboguer le plus rapidement possible.

7 D'autres outils disponibles pour le debug

7.1 Introduction à ltrace

`ltrace` est un autre outil de débogue de Linux. Il permet de tracer les appels que fait un programme aux bibliothèques partagées qu'il utilise.

Exercice n°13: Exécutez votre programme de tri corrigé avec `ltrace`. Quelles est (sont) la (les) bibliothèque(s) partagées que votre programme `tri` utilise. Quelles sont les fonctions de cette (ces) bibliothèques(s) qui sont utilisées ?

La commande `ltrace` permet de visualiser les appels faits par votre programme aux bibliothèques partagées qu'il utilise. Si on utilise la commande ltrace avec notre programme de tri et les arguments précédents, on obtient la trace suivante.

```
__libc_start_main(0x40074c, 6, 0x7ffcb97bb538, 0x400840 <unfinished ...>
malloc(20)                                       = 0x766010
```

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

```
atoi(0x7ffcb97bd25e, 0x766020, 8, 0x7f3cd8c01b20) = 8000
atoi(0x7ffcb97bd263, 8000, 16, 0x7ffcb97bd261)    = 7000
atoi(0x7ffcb97bd268, 7000, 24, 0x7ffcb97bd266)    = 5000
atoi(0x7ffcb97bd26d, 5000, 32, 0x7ffcb97bd26b)    = 1000
atoi(0x7ffcb97bd272, 1000, 40, 0x7ffcb97bd270)    = 4000
printf("%d ", 0)                                  = 2
printf("%d ", 1000)                               = 5
printf("%d ", 4000)                               = 5
printf("%d ", 5000)                               = 5
printf("%d ", 7000)                               = 5
putchar(10, 1, 0x7f3cd8c03780, 0x7ffffffb0 1000 4000 5000 7000
)       = 10
free(0x766010)                                    = <void>
+++ exited (status 0) +++
```

Celle-ci indique que notre programme utilise la bibbliothèque partagée libc et qu'il fait appel aux fonctions : `malloc`, `atoi`, `printf`, `putchar` et `free`.

7.2 Introduction à `strace`

`strace` est un outil de débogage sous Linux pour surveiller les appels système utilisés par un programme, et tous les signaux qu'il reçoit. Il a été rendu possible grâce à une fonctionnalité du noyau Linux appelée `ptrace`. Comme `strace` ne détaille que les appels système, il ne peut pas être utilisé comme un débogueur de code, tel que `gdb`. Il reste cependant plus simple à utiliser qu'un débogueur de code.

L'utilisation la plus courante est de lancer un programme en utilisant `strace`, qui affiche une liste des appels système faits par le programme. C'est utile lorsque le programme plante continuellement, ou ne se comporte pas comme souhaité. Par exemple, utiliser `strace` peut révéler que le programme tente d'accéder à un fichier qui n'existe pas ou qui ne peut pas être lu.

Exercice n°14: Utilisez le programme `strace` pour trouver où se trouve(nt) la (les) bibliothèques(s) qui sont chargées.

Le logiciel strace permet de connaître les appels systèmes (appels réalisés au noyau) qui sont réalisés par el programme. On peut ainsi avoir accès à plusieurs autres informations qui peuvent être utilent pour comprendre le comportement du programme.

```
execve("./tri", ["./tri", "8000", "7000", "5000", "1000", "4000"], [/* 66 vars */]) = 0
brk(NULL)                               = 0x871000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fabc74b6000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=90027, ...}) = 0
mmap(NULL, 90027, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fabc74a0000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1864888, ...}) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fabc6eca000
mprotect(0x7fabc7089000, 2097152, PROT_NONE) = 0
mmap(0x7fabc7289000,   24576,   PROT_READ|PROT_WRITE,   MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,   3,
0x1bf000) = 0x7fabc7289000
mmap(0x7fabc728f000,  14752,  PROT_READ|PROT_WRITE,  MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,  -1,  0)  =
0x7fabc728f000
close(3)                                = 0
```

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fabc749f000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fabc749e000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fabc749d000
arch_prctl(ARCH_SET_FS, 0x7fabc749e700) = 0
mprotect(0x7fabc7289000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7fabc74b8000, 4096, PROT_READ) = 0
munmap(0x7fabc74a0000, 90027)           = 0
brk(NULL)                               = 0x871000
brk(0x892000)                           = 0x892000
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
write(1, "0 1000 4000 5000 7000 \n", 230 1000 4000 5000 7000
) = 23
exit_group(0)                           = ?
+++ exited with 0 +++
```

Par exemple, dans notre cas, on voit dans ces traces que le programme commence par charger les fichiers de configuration pour savoir où trouver les bibliothèques dynamiques (access "`/etc/ls.so`..." puis charge la bibliothèques dynamique `/lib/x86_64-linux-gnu/libc.so.6`, avant d'allouer de la mémoire (`brk`, que nous verrons plus tard dans le cours) puis d'écrire le résultat (`write`)).

8  Défi : Déboguer le programme suivant en trois coups ....

Exercice n°15: Enfin, voici un dernier programme `rechercheBinaire.c` pour vous entrainer au debug d'un programme C dans l'environnement Visual Code. Pour tester cette recherche binaire, vous veillerez à passer des données triées bien entendu.

Hélène Collavizza, Stéphane Lavirotte, Erick Gallesio, Jean-Yves Tigli
Polytech'Nice – Sophia
SI3
2019-2020

TD n° 1
Debugging de Programmes :
Mise en œuvre avec Visual Studio Code

Annexe 1 : Installation de Visual Studio Code

1 Installation de VSC sous Linux

1.1 Installation

Allez à l'adresse :

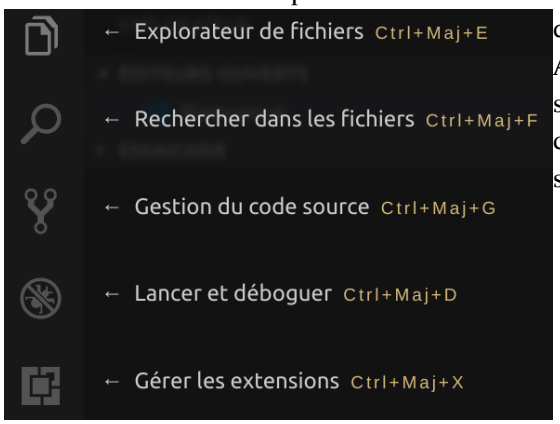
`https://code.visualstudio.com/`

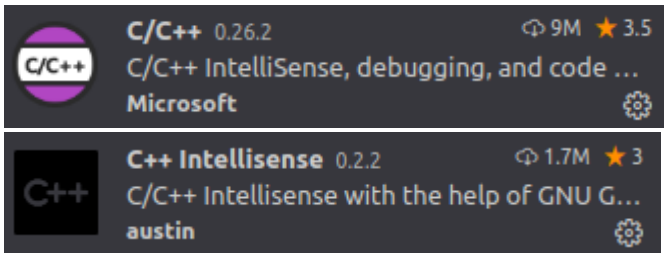
Selon votre configuration du browser, cela lance directement l'installation. Sinon :

- sauvez le fichier .deb
- faire `sudo dpkg -i <fichier>.deb` pour l'installer.

1.2 Lancement de VS et configuration pour C

L'exécutable est tout simplement « `code` ». Lancez `code` dans un `xterm` et préservez-le dans le lanceur. Les boutons de la fenêtre initiale ont le rôle décrit dans la copie d'écran à gauche. Avant toute chose, il faut installer l'extension pour C/C++ (Microsoft) en cliquant sur le bouton avec l'icône carré en bas et en cherchant l'extension C/C++. Cliquez sur installer. Ajoutez aussi l'extension C++ Intellisense (Austin).





TD n° 1
Debugging de Programmes :
Mise en œuvre avec Visual Studio Code

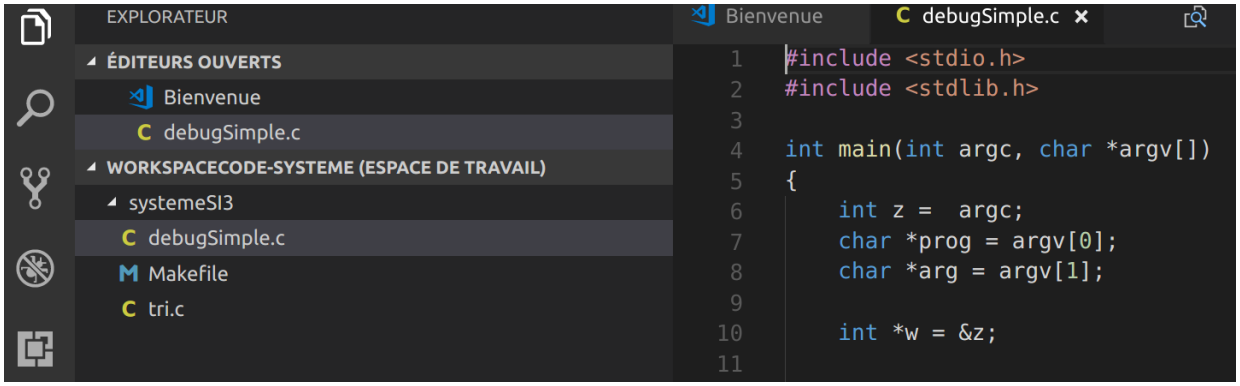
Annexe 2 : Création d'un espace de travail

2 Configuration pour compiler et exécuter

Dans un premier temps, il s'agit de créer un espace de travail contenant un `Makefile` et des exemples de programme C et de paramétrer votre environnement afin de compiler et exécuter vos programmes via le `Makefile`.

2.1 Dossier de travail

- Faire « Ajouter un dossier à l'espace de travail ». Choisissez un nom et un emplacement pertinent.
- Récupérez les fichiers `Makefile` et `debugSimple.c` que vous trouverez à l'adresse suivante :
  `http://trolen.polytech.unice.fr/cours/progsys/td01/`
- Copiez-les dans le dossier de travail que vous venez de créer,
- Faites « Ouvrir un fichier ». Vous voyez maintenant les 3 fichiers de votre dossier de travail avec une icône qui représente leur type.



2.2 Compilation

Le plus simple et facile à utiliser est de créer une tâche de génération par défaut :

- Menu « Tâches -> Configurer la tâche de génération par défaut » (en bas).
- Cliquer sur « Créer le fichier `task.json` à partir d'un modèle »
- Choisissez MSBuild. Cela ouvre le fichier `json.task`.

Il vous faut à présent modifier le fichier `json.task` de la façon suivante :

- Changez `"command": "msbuild"` en `"command": "make"`
- Changez l'argument de `"problemMatcher"` par `$gcc`
- Enlever les éléments entre `[]` dans `args`.
- Faites à nouveau Menu « Tâches -> Configurer la tâche de génération par défaut » et sélectionner la tâche que vous venez de créer.

Vous pouvez maintenant tapez le raccourci Ctrl-Shift-b pour « `build` », cela lance le `make`.
Vous pouvez compléter votre fichier `json.task` pour créer par exemple une tâche pour faire clean ou pour lancer la cible de test, téléchargeable sur `http://trolen.polytech.unice.fr/cours/progsys/td01/`

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
```

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

```
    "tasks": [
        {
            "label": "make",
            "type": "shell",
            "command": "make",
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        },
        {
            "label": "make clean",
            "type": "shell",
            "command": "make",
            "args": ["clean"],
            "problemMatcher": [
                "$gcc"
            ],
            "group":  "build",
        },
        {
            "label": "make test",
            "type": "shell",
            "command": "make",
            "args": ["test"],
            "problemMatcher": [
                "$gcc"
            ],
            "group":  "test",
        }
    ]
}
```

Le label est le nom associé à la tâche et permet de choisir la tâche dans le menu « Tâches -> Exécuter une tâche ». Il peut y avoir plusieurs tâches du même groupe (`build` ou `test`). Celle qui est marquée « `isDefault` » est la tâche de build par défaut, elle s'active par Ctrl-Shift-b. Le tag « `args` » permet de passer des arguments à la ligne de commande Shell.

TD n° 1
Debugging de Programmes : Mise en œuvre avec Visual Studio Code

3 Configuration du debugger

Nous allons illustrer les fonctions de base du debugger sur l'exemple simple du programme `debugSimple.c`. Pour lancer le debugger, cliquer sur l'icône du bug à gauche et choisissez C/C++ gdb. Cela créé un fichier `launch.json` par défaut de type *launch cppdb* qui lance le debugger.

- Dans la balise « program » mettez remplacez `a.out` par `debugSimple`
- Mettez par exemple `456` dans `args`.