

Utiliser des Outils

Pr. Olivier Gruber

olivier.gruber@univ-grenoble-alpes.fr

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

- Développer modulaire avec les outils de GNU
- Utiliser un IDE (Integrated Development Environment)

- Développer modulaire avec les outils de GNU
- Utiliser un IDE (Integrated Development Environment)
 - VsCode
 - Eclipse
 - Autres...



Ils sont pour beaucoup équivalents sur beaucoup d'aspects.

VsCode est d'approche un peu plus facile qu'Eclipse, surtout pour des projets simples.

Mais Eclipse offre un meilleur support pour Java et un meilleur support pour gdb. Et en fait un support workspace/multi-projets qui me semble mieux.

L'idéal ? Apprenez les deux !

- Plate-forme pour intégrer des outils
 - Pour le développement logiciel ou autre
 - Supporte une multitude de langages
 - Dont le C et Java, via des « plugins »
 - Au minimum, Eclipse permet l'édition, la compilation, et le debug
- Pourquoi apprendre Eclipse
 - C'est un outil de productivité
 - De **votre** productivité
- Comment apprendre Eclipse
 - Comprendre un usage basique
 - Puis approfondir à chaque occasion



Ce n'est pas le plus captivant comme apprentissage
Mais sur le long terme, c'est un apprentissage qui
va alléger votre charge de travail en augmentant
votre productivité...

Acquérir la maîtrise de vos outils
est votre responsabilité...

- Plate-forme à plugins
 - Les plugins apportent les fonctionnalités/outils spécifiques, comme les outils pour le C ou Java
- Eclipse propose deux canevas d'intégration
 - L'espace de travail (workspace)
 - Organisé en projets
 - Les projets contribuent des ressources
 - Souvent des sources, mais pas toujours
 - Le banc de travail (workbench)
 - Pour l'intégration dans l'interface graphique
 - Organisé en perspectives et en vues/éditeurs



On va installer la version pour les développeurs Java et on installera les plugins pour le C.

Un apprentissage...
Deux usages !

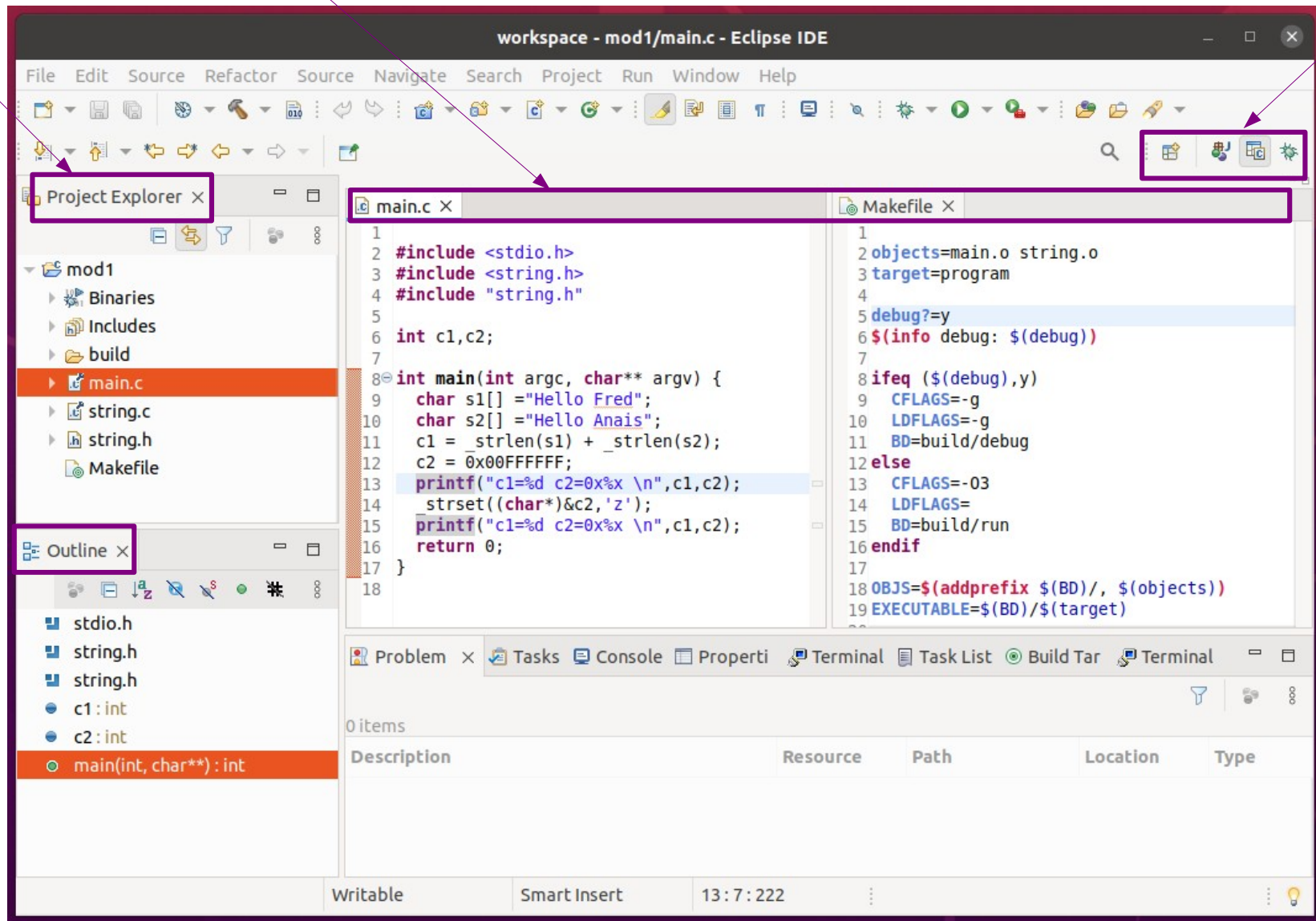
Eclipse – Workbench – Perspective pour le C

6

Package Explorer

Editors

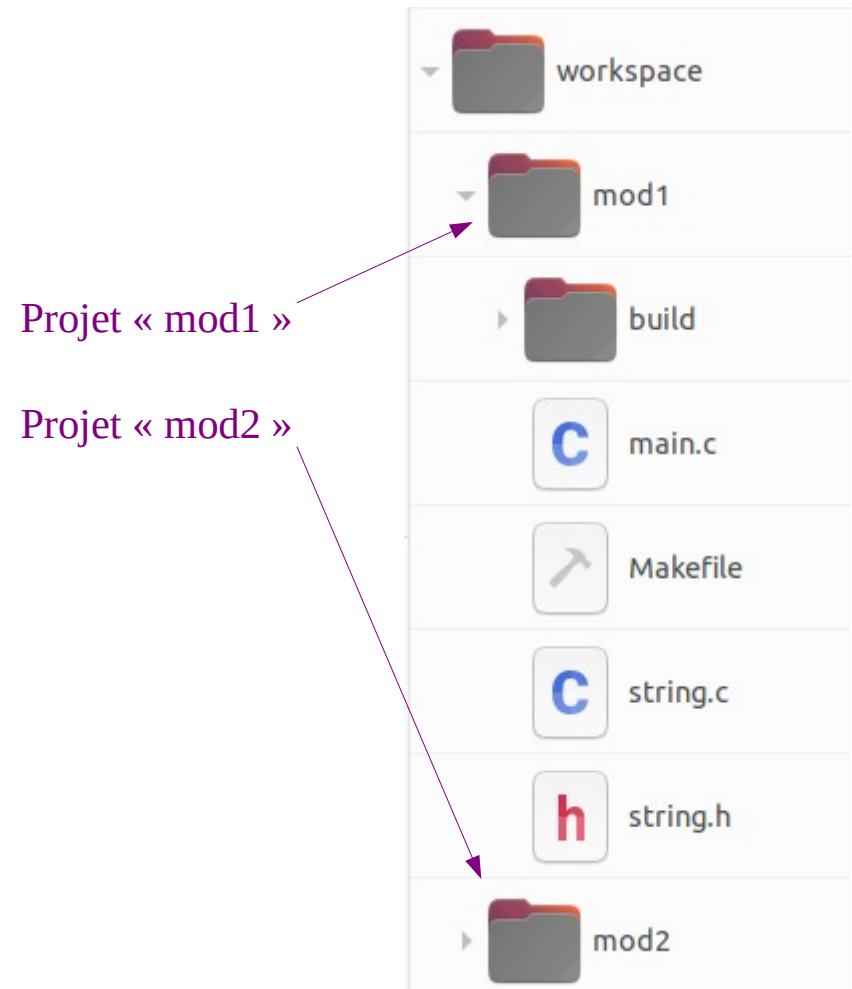
Perspective toolbar



Eclipse – Workspace dans le système de fichiers

7

- Le concept d'espace de travail (workspace)
 - C'est un dossier sur le système de fichiers
 - Il contient un ou plusieurs projets
- Chaque projet est aussi un dossier
 - Qui contient des ressources
 - Telles que des sources
 - Ou des fichiers de données
- Chaque projet a une nature
 - On parle aussi de « builder »
 - Projet « mod1 »
 - Un exemple d'un projet avec « Makefile »



Eclipse – Workspace Flexibility

8

- Project location
 - Par défaut dans le dossier « workspace »
 - Mais pas forcément
- Linked folders / files
 - Eclipse permet les liens symboliques
- Config par langage
 - Version du compilateur
 - Version des librairies
 - Autres...

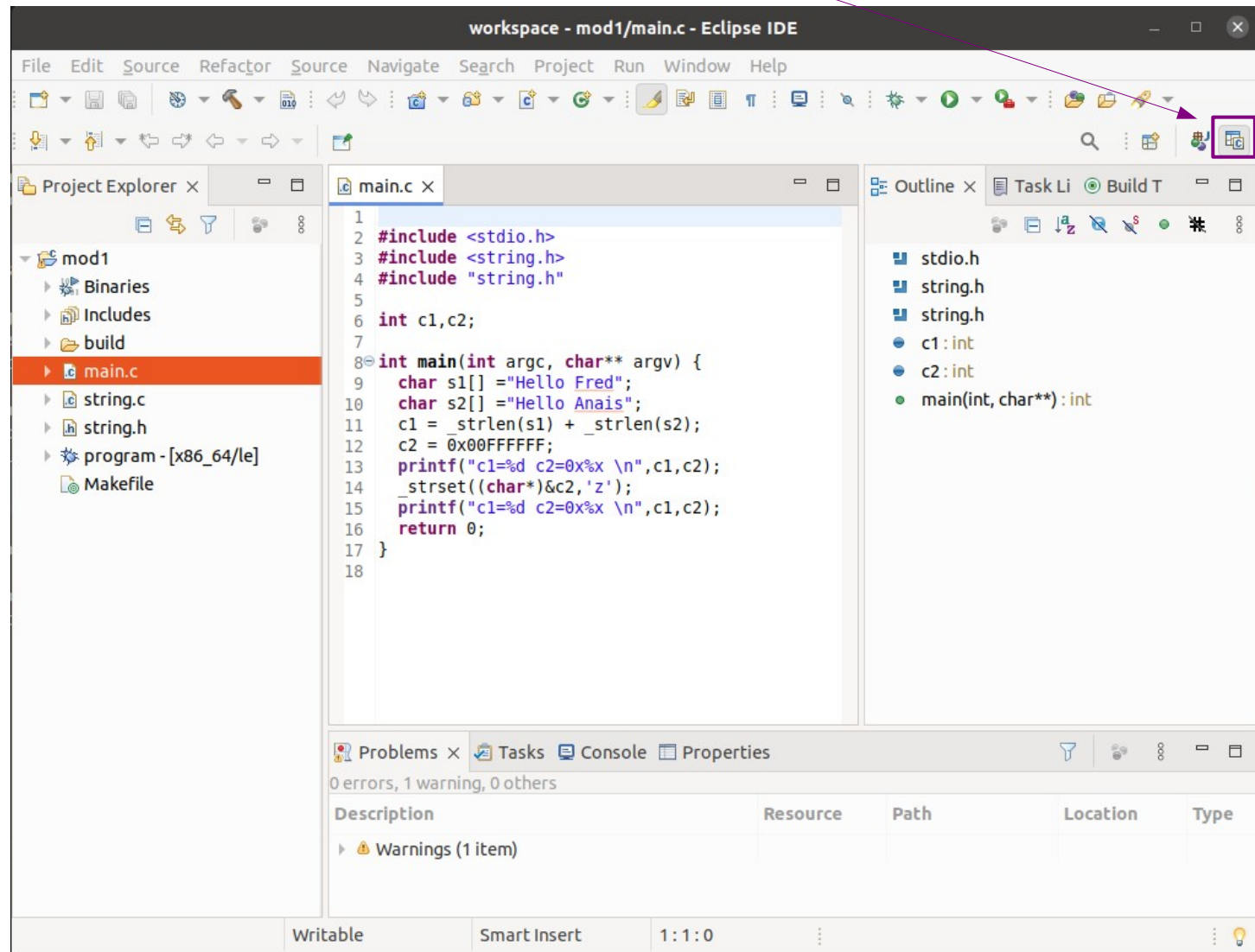
C'est **important** de maîtriser **vos configs**, la flexibilité offerte par Eclipse est une des raisons pour choisir Eclipse.

The screenshot shows the 'New Java Project' dialog box in Eclipse. The title bar says 'New Java Project'. The main section is 'Create a Java Project' with the instruction 'Enter a project name.' Below this is a text field for 'Project name:'. A checkbox 'Use default location' is checked, and the location is '/home/ogruber/Tries/ooop' with a 'Browse...' button. The 'JRE' section has three radio buttons: 'Use an execution environment JRE:' (selected, with 'JavaSE-11' in the dropdown), 'Use a project specific JRE:' (with 'jre' in the dropdown), and 'Use default JRE 'jre' and workspace compiler preferences' (with a 'Configure JREs...' link). The 'Project layout' section has two radio buttons: 'Use project folder as root for sources and class files' and 'Create separate folders for sources and class files' (selected, with a 'Configure default...' link). The 'Working sets' section has a checkbox 'Add project to working sets' (unchecked), a dropdown menu, and 'New...' and 'Select...' buttons. The 'Module' section has a checkbox 'Create module-info.java file' (unchecked) and a text field. At the bottom, there is an information icon and a message: 'The default compiler compliance level for the current workspace is 21. The new project will use a project specific compiler compliance level of 11. Configure...'. The bottom right has buttons for '< Back', 'Next >', 'Cancel', and 'Finish'.

La Perspective pour le C

9

Vous avez la perspective pour le « C »,
avec l'organisation par défaut de ses vues.



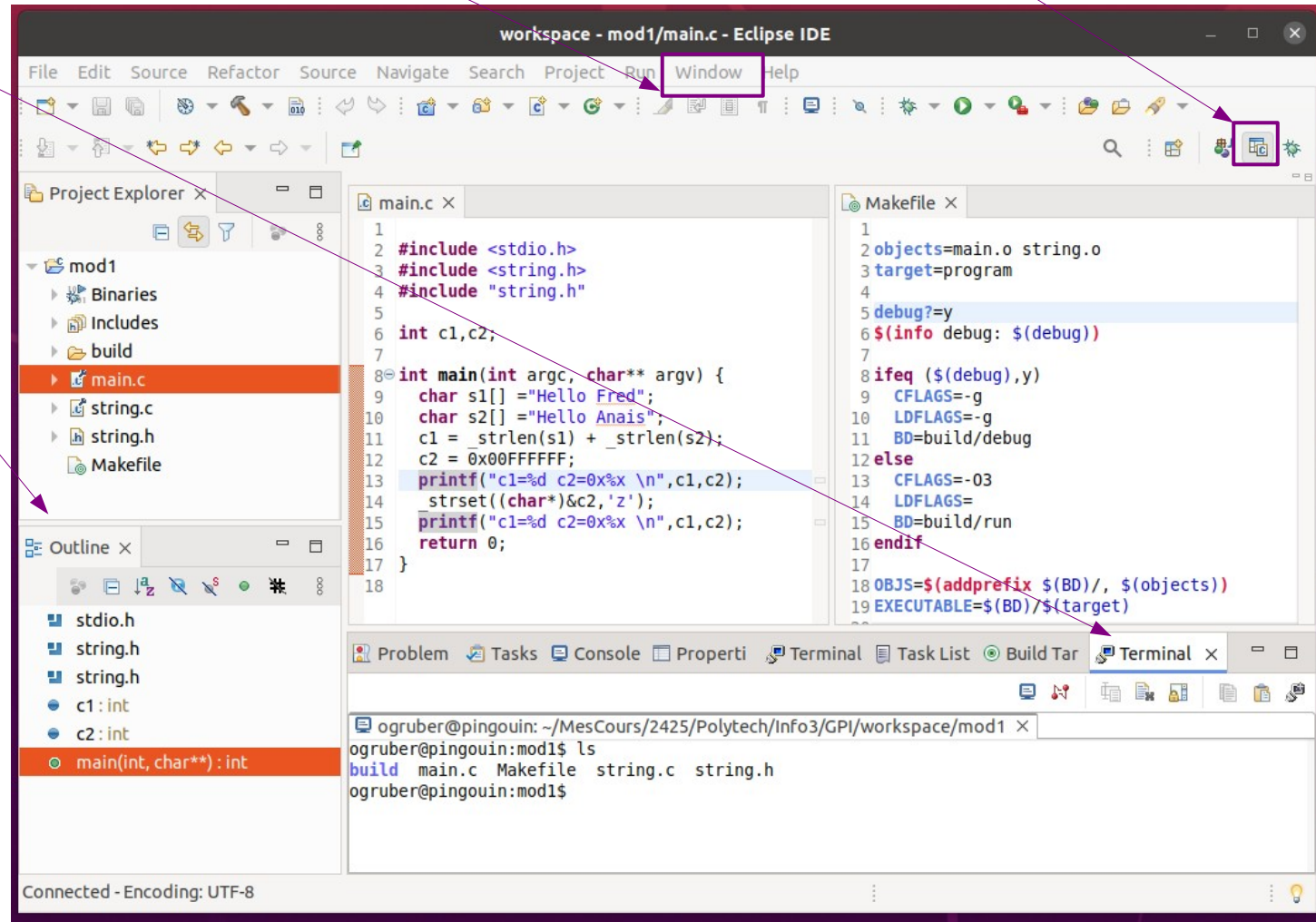
Configurez vos perspectives...

10

Vous pouvez réorganiser les vues et en ouvrir d'autres.

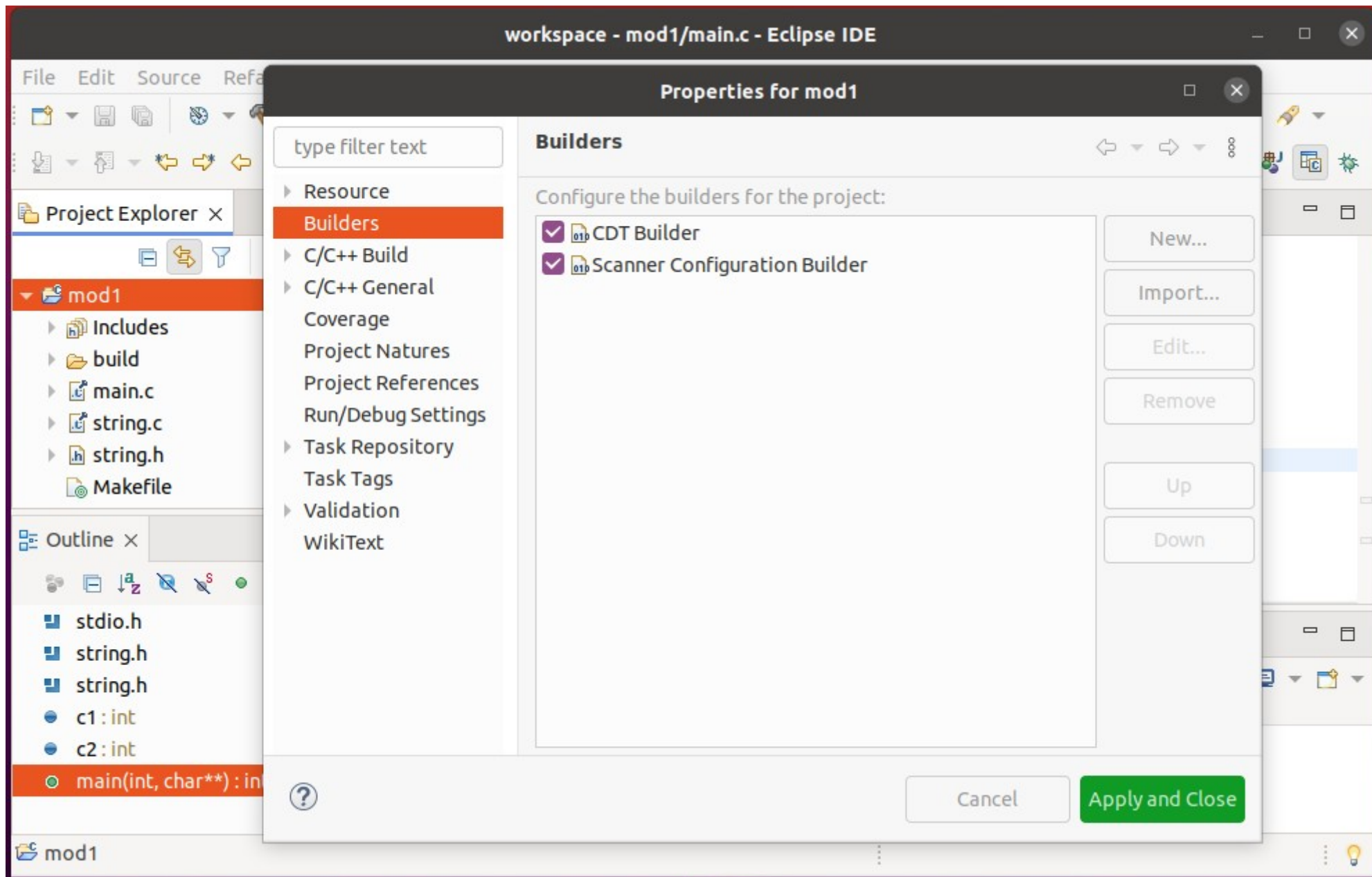
Window → Show View

Vous pouvez faire un reset du layout si nécessaire (click droit).



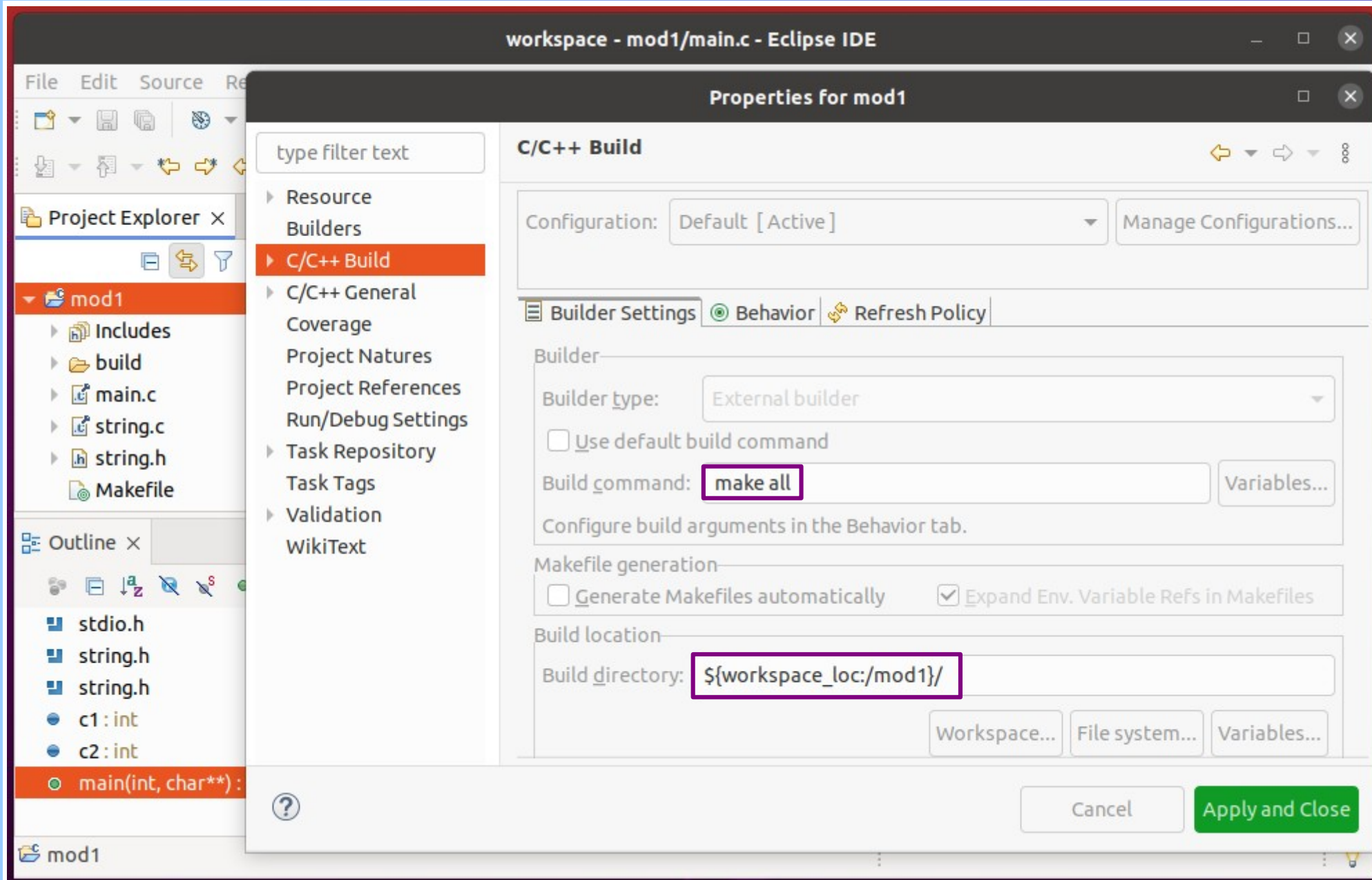
Project Properties – Builders

11



Project Properties – Build Command

12

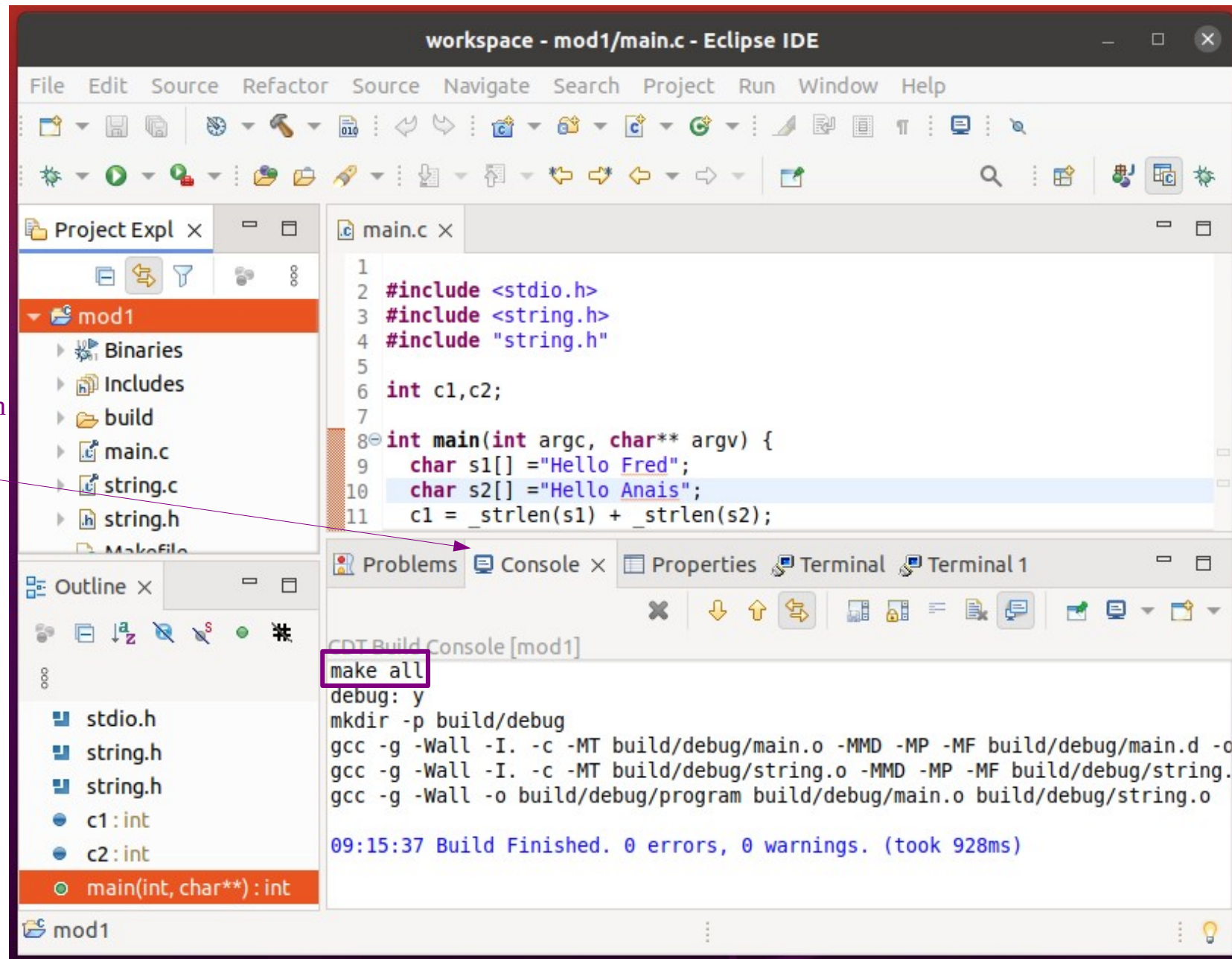


« build » votre projet

13

Click-droit sur le
project et choisir
« build » ou « clean »

Vous voyez le
résultat de l'exécution
du makefile dans la
vue « console »



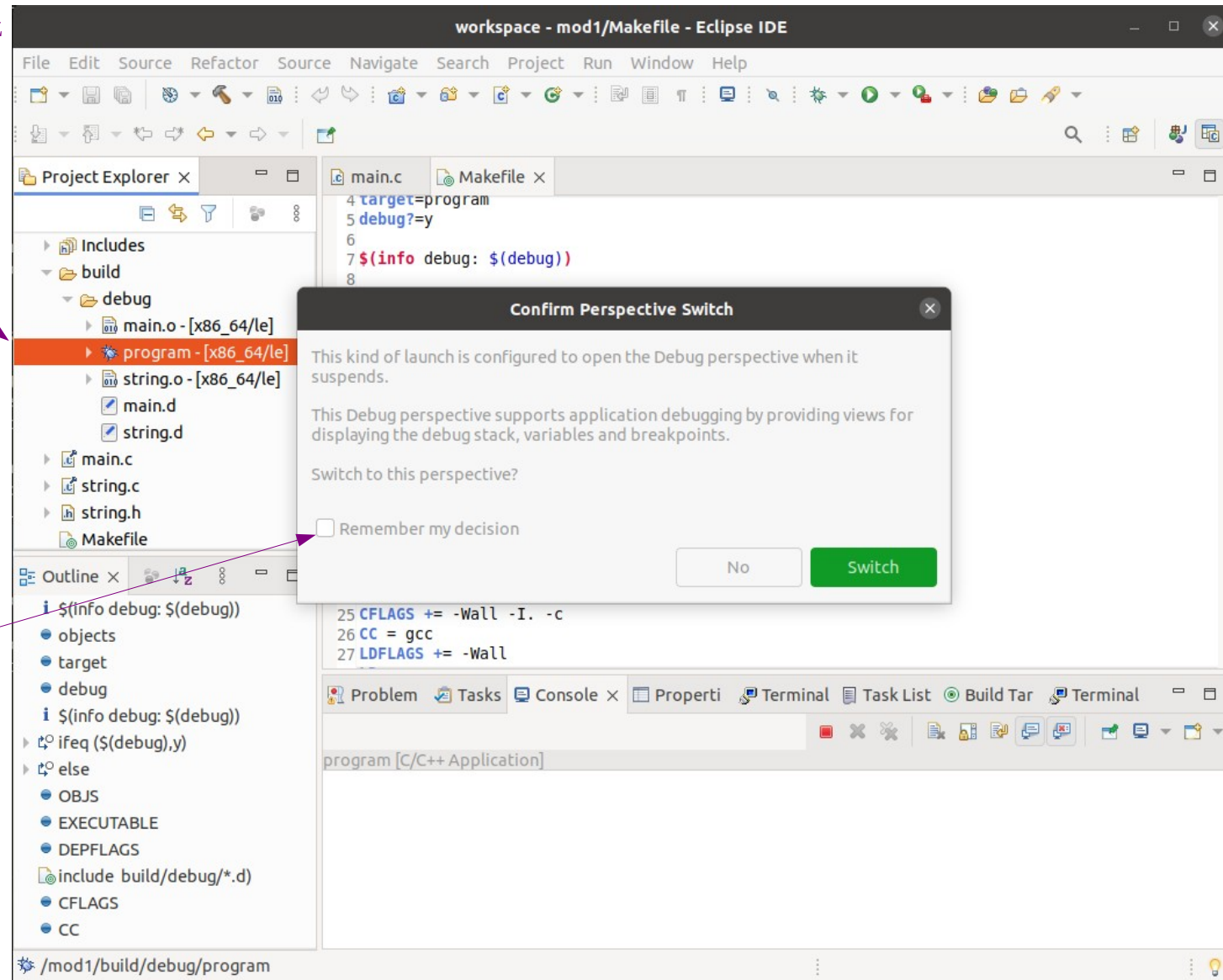
Lancer une session de debug

14

Si le build a réussi, vous pouvez sélectionner l'exécutable et demander une session de debug (clique-droit → debug-as)

Eclipse vous indique qu'il va changer de perspective et vous demande si vous voulez faire le switch automatiquement les prochaines fois.

Conseil : dire oui.



Debug Perspective – Aperçu Globale

15

The screenshot displays the Eclipse IDE in the Debug Perspective. The central editor shows the source code of `main.c`, with a breakpoint set at line 12. The left sidebar contains the **Debug** console, **Outline**, and **Registers** views. The right sidebar contains the **Variables**, **Breakpoint**, and **Expression** views. The bottom status bar shows the **Console**, **Registers**, **Debug Shell**, **Executables**, **Debugger Console**, and **Memory** views.

Source Code (main.c):

```
1 #include <stdio.h>
2 #include <string.h>
3 #include "string.h"
4
5 int c1,c2;
6
7 int main(int argc, char** argv) {
8     char s1[] = "Hello Fred";
9     char s2[] = "Hello Anaïs";
10    c1 = _strlen(s1) + _strlen(s2);
11    c2 = 0x00FFFFFF;
12    printf("c1=%d c2=0x%x \n",c1,c2);
13    strset((char*)&c2,'z');
14    printf("c1=%d c2=0x%x \n",c1,c2);
15    return 0;
16 }
```

Variables View:

Name	Type	Value
argc	int	1
argv	char **	0x7fffffff5a8
s1	char [11]	0x7fffffff481
s2	char [12]	0x7fffffff48c

Debugger Console:

```
program [C/C++ Application] gdb (9.2)
(gdb)
Temporary breakpoint 1, main (argc=21845, argv=0x7fffffff496) at main.c:8
8     int main(int argc, char** argv) {
(gdb)
```


Debug Perspective – Contrôler l'exécution

16

workspace - mod1/main.c - Eclipse IDE

File Edit Source Refactor Source Navigate Search Project Run Window Help

Apprenez les raccourcis clavier ! 😊

main.c

```
1
2 #include <stdio.h>
3 #include <string.h>
4 #include "string.h"
5
6 int c1,c2;
7
8 int main(int argc, char** argv) {
9     char s1[] = "Hello Fred";
10    char s2[] = "Hello Anais";
11    c1 = _strlen(s1) + _strlen(s2);
12    c2 = 0x00FFFFFF;
13    printf("c1=%d c2=0x%x \n",c1,c2);
14    strset((char*)&c2,'z');
15    printf("c1=%d c2=0x%x \n",c1,c2);
16    return 0;
17 }
```

Debug

program [C/C++ Application]

program [319892] [cores:1]

Thread #1 [program] 3

main() at main.c:12

gdb (9.2)

Outline

- stdio.h
- string.h
- string.h
- c1:int
- c2:int
- main(int, char**) : int

Variables

Name	Type	Value
argc	int	1
argv	char **	0x7fffffff5a8
s1	char [11]	0x7fffffff481
s2	char [12]	0x7fffffff48c

Name : s2
Details: "Hello Anais"
Default: 0x7fffffff48c
Decimal: 140737488344204
Hex: 0x7fffffff48c

Console Registers Debug Shell Executables Debugger Console Memory

program [C/C++ Application] gdb (9.2)

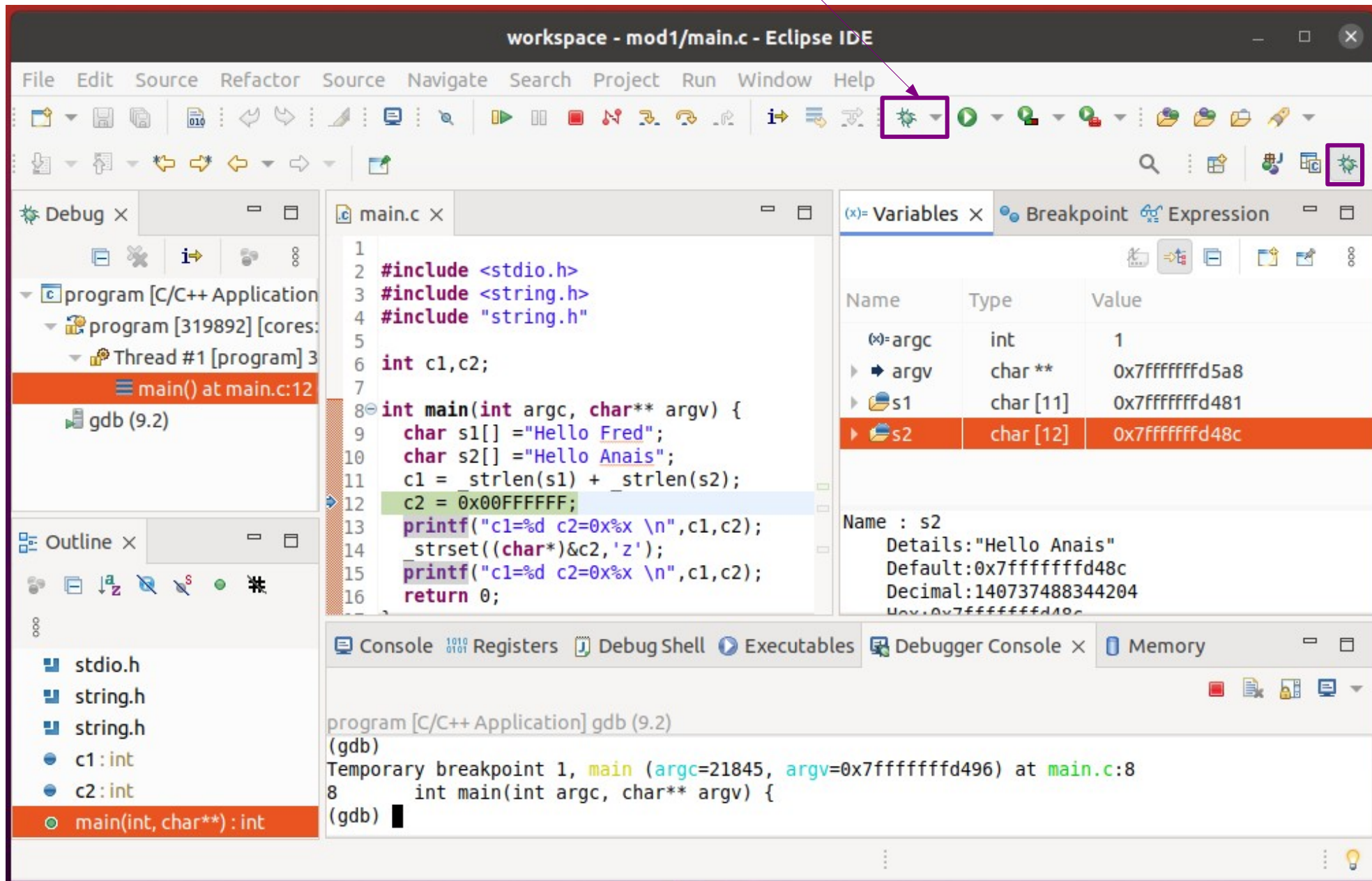
(gdb)

Temporary breakpoint 1, main (argc=21845, argv=0x7fffffff496) at main.c:8

```
8      int main(int argc, char** argv) {
(gdb) █
```


Debug Perspective – Debug Configurations

17



The screenshot shows the Eclipse IDE in the Debug Perspective. The main editor displays a C program with a breakpoint at line 12. The Variables view on the right shows the state of variables: argc=1, argv points to a memory address, s1 is 'Hello Fred', and s2 is 'Hello Anaïs'. The Console view at the bottom shows the program's execution output.

main.c

```
1
2 #include <stdio.h>
3 #include <string.h>
4 #include "string.h"
5
6 int c1,c2;
7
8 int main(int argc, char** argv) {
9     char s1[] ="Hello Fred";
10    char s2[] ="Hello Anaïs";
11    c1 = _strlen(s1) + _strlen(s2);
12    c2 = 0x00FFFFFF;
13    printf("c1=%d c2=0x%x \n",c1,c2);
14    strset((char*)&c2,'z');
15    printf("c1=%d c2=0x%x \n",c1,c2);
16    return 0;
17 }
```

Variables

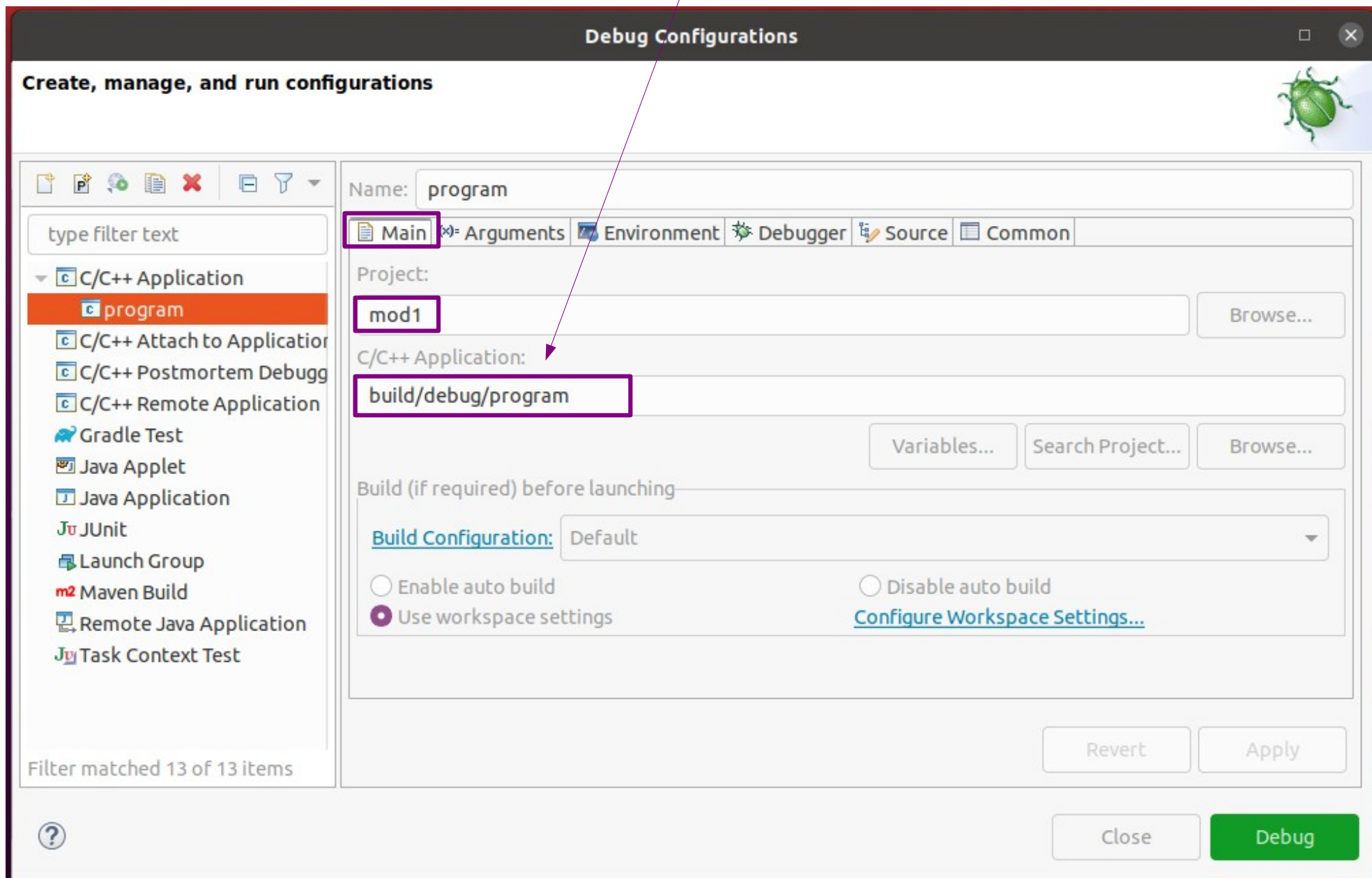
Name	Type	Value
argc	int	1
argv	char **	0x7fffffff5a8
s1	char [11]	0x7fffffff481
s2	char [12]	0x7fffffff48c

Console

```
program [C/C++ Application] gdb (9.2)
(gdb)
Temporary breakpoint 1, main (argc=21845, argv=0x7fffffff496) at main.c:8
8      int main(int argc, char** argv) {
(gdb)
```

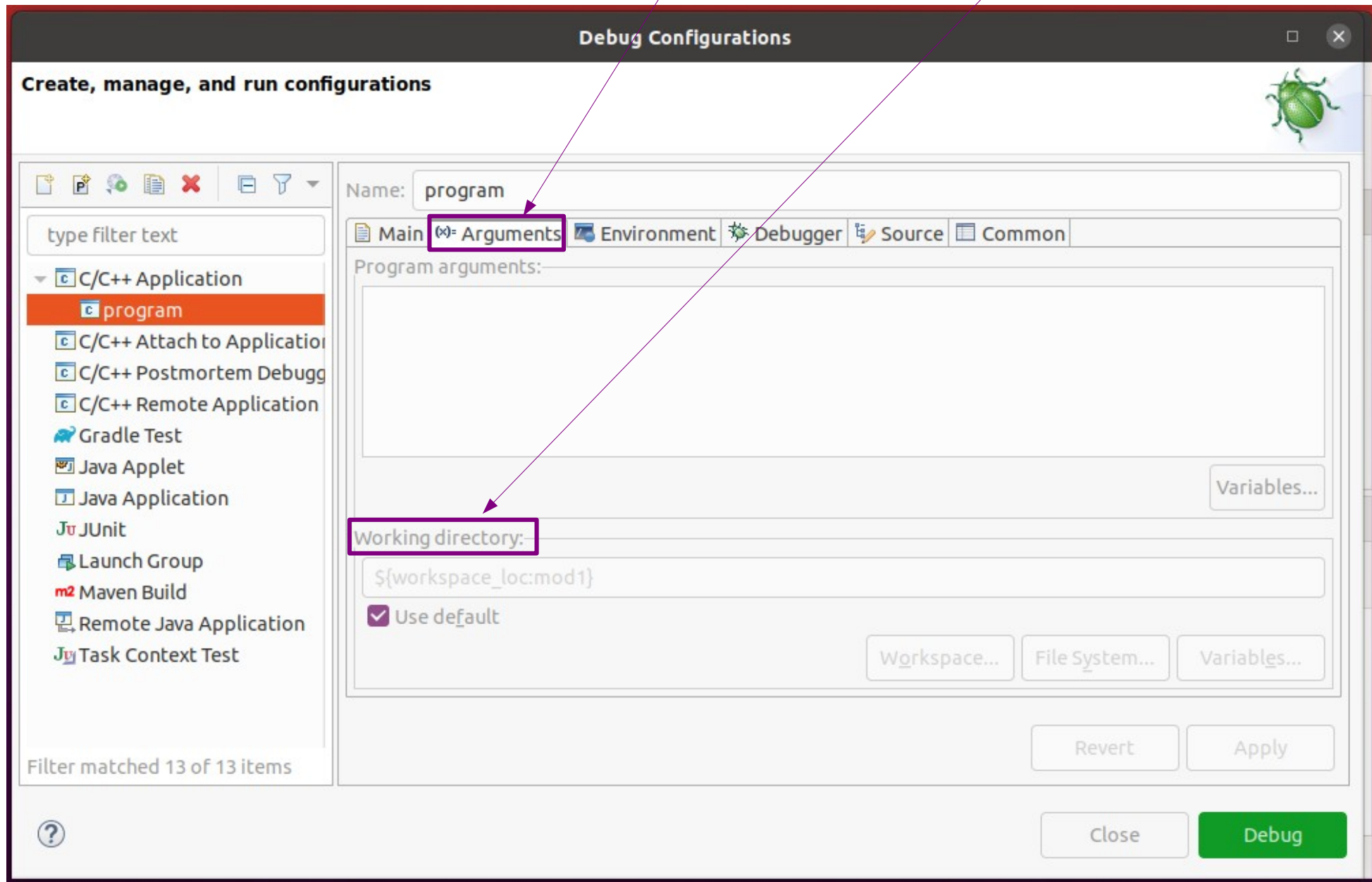
Debug Configurations – L'exécutable

18



Debug Configurations – Les arguments et le dossier courant

19



- Développer modulaire en C avec les outils de GNU 
- Utiliser un IDE (Integrated Development Environment)

- Approche simple
 - Un seul source...
 - Une simple compilation...

source file
« fact.c »

```
12 int fact(int v0) {  
13     int v1 = 0;  
14     int v2 = 1;  
15     do {  
16         v1 = v1 + 1;  
17         v2 = v2 * v1;  
18     } while (v1 < v0);  
19     return v2;  
20 }
```

```
36 int main(int argc, char** argv) {  
37     int n = fact(6);  
38     return 0;  
39 }
```

```
$ gcc -g -Wall -o fact fact.c
```

```
$ ls -al
```

```
-rw-rw-r--  ... fact.c
```

```
-rwxrwxr-x  ... fact
```

```
$
```

↑ compilation (avec « debug symbols »)

↓ session de debug

```
$ gdb fact
```

```
(gdb) breakpoint fact.c:37
```

```
Breakpoint 1 at fact.c:37
```

```
(gdb) run
```

```
Breakpoint 1, fact(n=3) at fact.c:37
```

```
int n = fact(6);
```

```
(gdb)
```

Source et Compilation – Exécution / Processus

22

Source file : main.c

```
#include <stdio.h>

int main(int argc, char** argv) {
    char* s = "Hello World!";
    printf("%s\n", s);
    return 0;
}
```

Permet d'avoir la définition
de la fonction « printf »

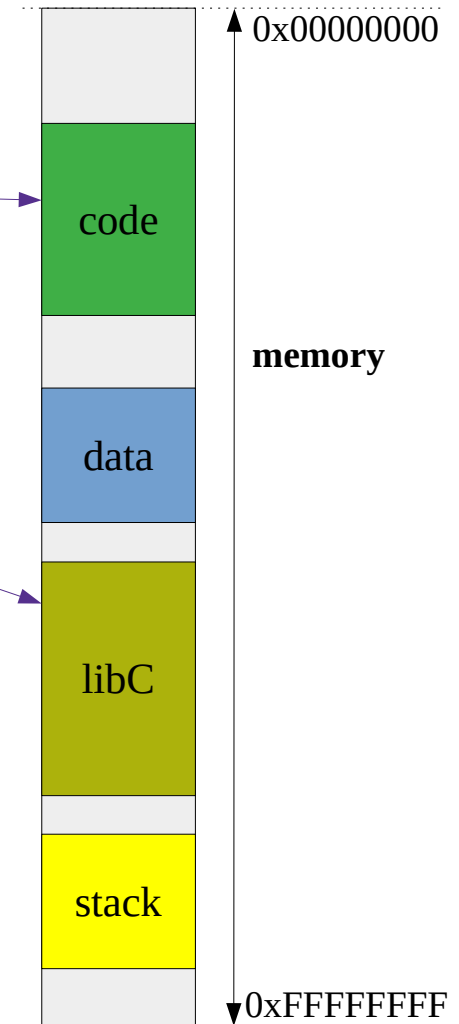
la fonction « main »
fait partie de votre code

la fonction « printf »
fait partie de la libC

Shell session : compiler et exécuter

```
$ gcc -Wall -o main main.c
$ ./main
Hello World!
$ file main
ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked,
for GNU/Linux 3.2.0, not stripped
```

processus



main.c

```
#include <stdio.h>
#include <string.h>
#include "string.h"

int c1,c2;

int main(int argc, char** argv) {
    char s1[] ="Hello Fred";
    char s2[] ="Hello Anaïs";
    c1 = _strlen(s1) + _strlen(s2);
    c2 = 0x00FFFFFF;
    printf("c1=%d c2=0x%x \n",c1,c2);
    _strset((char*)&c2,'z');
    printf("c1=%d c2=0x%x \n",c1,c2);
    return 0;
}
```

Inclue le fichier

string.h

```
int _strlen(char *s);
int _strset(char *s, char c);
```

string.c

```
#include "string.h"

int _strlen(char *s) {
    int count=0;
    while (*s != '\0') {
        s = s + 1;
        count = count+1;
    }
    return count;
}

int _strset(char *s, char c) {
    int count=0;
    while (*s != '\0') {
        *s = c;
        s = s + 1;
        count = count+1;
    }
    return count;
}
```

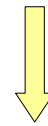
Développer modulaire, c'est développer des morceaux, en répartissant le code entre plusieurs fichiers sources (source files) et des fichiers entêtes (header files).

Notez les appels de fonctions traversant les frontières de fichier sources. Pour pouvoir écrire de tels appels, il faut inclure les fichiers entêtes correspondant.

string.h

```
int _strlen(char *s);  
int _strset(char *s, char c);
```

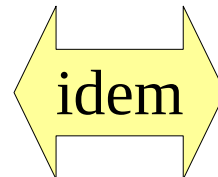
Quel est le sens ? C'est celle d'une inclusion du texte du fichier référencé.



main.c

```
#include <stdio.h>  
#include <string.h>  
#include "string.h"  
  
int c1,c2;  
  
int main(int argc, char** argv) {  
    char s1[] ="Hello Fred";  
    char s2[] ="Hello Anais";  
    c1 = _strlen(s1) + _strlen(s2);  
    c2 = 0x00FFFFFF;  
    printf("c1=%d c2=0x%x \n",c1,c2);  
    _strset((char*)&c2,'z');  
    printf("c1=%d c2=0x%x \n",c1,c2);  
    return 0;  
}
```

Il faut éviter
les conflits de noms
avec la librairie standard.



```
#include <stdio.h>  
#include <string.h>  
  
int _strlen(char *s);  
int _strset(char *s, char c);  
  
int c1,c2;  
  
int main(int argc, char** argv) {  
    char s1[] ="Hello Fred";  
    char s2[] ="Hello Anais";  
    c1 = _strlen(s1) + _strlen(s2);  
    c2 = 0x00FFFFFF;  
    printf("c1=%d c2=0x%x \n",c1,c2);  
    _strset((char*)&c2,'z');  
    printf("c1=%d c2=0x%x \n",c1,c2);  
    return 0;  
}
```


main.c

```
#include <stdio.h>
#include <string.h>
#include "string.h"

int c1;

int main(int argc, char** argv) {
    char s1[] = "Hello Fred";
    char s2[] = "Hello Anaïs";
    c1 = _strlen(s1) + _strlen(s2);
    // c2 = 0x00FFFFFF;
    printf("c1=%d c2=0x%x \n", c1, c2);
    _strset((char*)&c2, 'z');
    printf("c1=%d c2=0x%x \n", c1, c2);
    return 0;
}
```

Inclue le fichier

string.h

```
int _strlen(char *s);
int _strset(char *s, char c);
export int c2;
```

string.c

```
#include "string.h"
int c2 = 0x00FFFFFF;
int _strlen(char *s) {
    ...
}
int _strset(char *s, char c) {
    ...
}
```

Dans « string.h », on trouve la déclaration de la variable « c2 », c'est le mot clé « export » qui indique que c'est une déclaration. La définition est dans le source « string.c ».

main.c

```
#include <stdio.h>
#include <string.h>
#include "string.h"

int c1;
export int c3;
export int c4; // c'est une erreur

int main(int argc, char** argv) {
    ...
    return 0;
}
```

Inclue le fichier

string.h

```
int _strlen(char *s);
int _strset(char *s, char c);
export int c2;
```

string.c

```
#include "string.h"
int c2 = 0x00FFFFFF;
int c3 = 0;
static int c4 = 0;

static int _strzero(char *s) {
    ...
}

int _strlen(char *s) {
    ...
}

int _strset(char *s, char c) {
    ...
}
```

L'usage de « export » n'est pas réservé aux headers, rappelez vous, il s'agit que d'une inclusion textuelle.

Du coup, avec le mot clé « static », il est possible de rendre Privé, à une unité de compilation, une variable. Ici, c'est le cas de la variable « c4 ».

L'usage de static n'est pas limité aux variables, il s'applique aussi aux fonctions.

main.c

```
#include <stdio.h>
#include <string.h>
#include "string.h"

int c1,c2;

int main(int argc, char** argv) {
    char s1[] ="Hello Fred";
    char s2[] ="Hello Anaïs";
    c1 = _strlen(s1) + _strlen(s2);
    c2 = 0x00FFFFFF;
    printf("c1=%d c2=0x%x \n",c1,c2);
    _strset((char*)&c2,'z');
    printf("c1=%d c2=0x%x \n",c1,c2);
    return 0;
}
```

string.h

```
int _strlen(char *s);
int _strset(char *s, char c);
```

string.c

```
#include "string.h"

int _strlen(char *s) {
    int count=0;
    while (*s != '\0') {
        s = s + 1;
        count = count+1;
    }
    return count;
}

int _strset(char *s, char c) {
    int count=0;
    while (*s != '\0') {
        *s = c;
        s = s + 1;
        count = count+1;
    }
    return count;
}
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o string.o string.c
```

Compilation par « morceaux »,
ce qui produit un « object file »
par morceau. Notez l'option « -c »

main.c

```
#include <stdio.h>
#include <string.h>
#include "string.h"

int c1,c2;

int main(int argc, char** argv) {
    char s1[] ="Hello Fred";
    char s2[] ="Hello Anaïs";
    c1 = _strlen(s1) + _strlen(s2);
    c2 = 0x00FFFFFF;
    printf("c1=%d c2=0x%x \n",c1,c2);
    _strset((char*)&c2,'z');
    printf("c1=%d c2=0x%x \n",c1,c2);
    return 0;
}
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o string.o string.c
$ gcc -o program main.o string.o
```

Edition de liens
entre les « object files »,
produit par la compilation
des différents sources.
Cette édition de liens
produit un exécutable,
dans un fichier,
ici c'est « program ».

string.h

```
int _strlen(char *s);
int _strset(char *s, char c);
```

string.c

```
#include "string.h"

int _strlen(char *s) {
    int count=0;
    while (*s != '\0') {
        s = s + 1;
        count = count+1;
    }
    return count;
}

int _strset(char *s, char c) {
    int count=0;
    while (*s != '\0') {
        *s = c;
        s = s + 1;
        count = count+1;
    }
    return count;
}
```

main.c

```
#include <stdio.h>
#include <string.h>
#include "string.h"

int c1,c2;

int main(int argc, char** argv) {
    char s1[] ="Hello Fred";
    char s2[] ="Hello Anaïs";
    c1 = _strlen(s1) + _strlen(s2);
    c2 = 0x00FFFFFF;
    printf("c1=%d c2=0x%x \n",c1,c2);
    _strset((char*)&c2,'z');
    printf("c1=%d c2=0x%x \n",c1,c2);
    return 0;
}
```

```
$ gcc -c -o main.o main.c
$ gcc -c -o string.o string.c
$ gcc -o program main.o string.o
$ ./program
c1=21 c2=0xffffffff
c1=21 c2=0x7a7a7a
$
```

Après compilation et
édition de liens,
on peut exécuter...

string.h

```
int _strlen(char *s);
int _strset(char *s, char c);
```

string.c

```
#include "string.h"

int _strlen(char *s) {
    int count=0;
    while (*s != '\0') {
        s = s + 1;
        count = count+1;
    }
    return count;
}

int _strset(char *s, char c) {
    int count=0;
    while (*s != '\0') {
        *s = c;
        s = s + 1;
        count = count+1;
    }
    return count;
}
```

On peut utiliser un « makefile »...

Avec plusieurs cibles, avec les deux plus classiques que sont « all » et « clean »

La première est celle par défaut.

```
$ make all
gcc -c -o main.o main.c
gcc -c -o string.o string.c
gcc -o program main.o string.o
$ ./program
c1=21 c2=0xffffffff
c1=21 c2=0x7a7a7a
$ make clean
rm main.o string.o program
$
```

Makefile

```
all: program

clean:
rm main.o string.o program

program: main.o string.o
gcc -o program main.o string.o

main.o: main.c string.h
gcc -c -o main.o main.c

string.o: string.c string.h
gcc -c -o string.o string.c
```

Ici, vous écrivez une ligne de commande du shell, après une tabulation.

Makefile – Les Bases...

31

On peut utiliser un « makefile »...

\$ make all

```
gcc -c -o main.o main.c
gcc -c -o string.o string.c
gcc -o program main.o string.o
Done.
```

\$./program

```
c1=21 c2=0xffffffff
c1=21 c2=0x7a7a7a
$ make clean
rm main.o string.o program
$
```

Vous pouvez avoir plusieurs lignes de commande par cible

Avec le prefix « @ », la commande n'est pas affichée par makefile avant son exécution.

Attention : il faut permettre au « rm » d'échouer, c'est le rôle du « - » devant!

Makefile

```
all: program
```

```
clean:
```

```
-rm main.o string.o program
```

```
program: main.o string.o
```

```
gcc -o program main.o string.o
```

```
@echo "Done."
```

```
main.o: main.c string.h
```

```
gcc -c -o main.o main.c
```

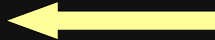
```
string.o: string.c string.h
```

```
gcc -c -o string.o string.c
```

Attention : il faut utiliser ici un « tab » et pas des espaces avant une commande!

Utiliser un « makefile »
c'est bénéficier aussi de compilation incrémentale...

```
$ ./program
c1=21 c2=0xffffffff
c1=21 c2=0x7a7a7a
$ make
make: Nothing to be done for 'all'.
$ touch string.c
$ make all
gcc -g -Wall -c -o string.o string.c
gcc -g -Wall -o program main.o string.o
Done.
$
```



Makefile

```
all: program

clean:
    rm main.o string.o program

program: main.o string.o
    gcc -o program main.o string.o
    @echo "Done."

main.o: main.c string.h
    gcc -c -o main.o main.c

string.o: string.c string.h
    gcc -c -o string.o string.c
```

Touch : simule l'édition du fichier « string.c »
en mettant à jour la date de la dernière écriture dans le fichier.

Notez que le makefile ne provoque que la compilation nécessaire : celle de string.c
Puis, il refait l'édition de lien pour générer le nouvel exécutable.

- Améliorons notre makefile
 - Utilisons des **variables** pour éviter les redites
- Pour différentes raisons
 - Pour le nom du compilateur et du linker
 - Et pour les options données au compilateur
 - **-g** pour le debug
 - **-Wall** pour tous les warning
 - **-c** pour empêcher l'édition de lien
 - Et pour les options données au linker
 - **-g** pour indiquer au linker de garder les informations de debug

Makefile

```
CC=gcc
CFLAGS=-g -Wall -c
LD=gcc
LDFLAGS=-g

all: program

clean:
    -rm main.o string.o program

program: main.o string.o
    $(LD) $(LDFLAGS) -o program main.o string.o

main.o: main.c string.h
    $(CC) $(CFLAGS) -o main.o main.c

string.o: string.c string.h
    $(CC) $(CFLAGS) -o string.o string.c
```

- Améliorons notre makefile
 - Utilisons les variables pour éviter les redites
- Pour différentes raisons
 - Pour la liste des « morceaux »
 - Pour le nom de l'exécutable

C'est bien, mais il y a encore beaucoup de redites...

Makefile

```
OBJECTS=main.o string.o
EXECUTABLE=program

CC=gcc
CFLAGS=-g -Wall -c
LD=gcc
LDFLAGS=-g

all: $(EXECUTABLE)

clean:
    -rm $(EXECUTABLE) $(OBJECTS)

$(EXECUTABLE): $(OBJECTS)
    $(LD) $(LDFLAGS) -o $(EXECUTABLE) $(OBJECTS)

main.o: main.c string.h
    $(CC) $(CFLAGS) -o main.o main.c

string.o: string.c string.h
    $(CC) $(CFLAGS) -o string.o string.c
```

- Améliorons notre makefile
 - Utilisons les variables pour éviter les redites
- Pour différentes raisons
 - Pour la liste des « morceaux »
 - Pour le nom de l'exécutable
 - Pour les *dépendances*

C'est un peu mieux avec les variables contextuelles :

- `$@` : la cible
- `$<` : le premier fichier des dépendances
- `$^` : tous les fichiers des dépendances

Makefile

```
OBJECTS=main.o string.o
EXECUTABLE=program

CC=gcc
CFLAGS=-g -Wall -c
LD=gcc
LDFLAGS=-g

all: $(EXECUTABLE)

clean:
    -rm $(EXECUTABLE) $(OBJECTS)

$(EXECUTABLE): $(OBJECTS)
    $(LD) $(LDFLAGS) -o $@ $^

main.o: main.c string.h
    $(CC) $(CFLAGS) -o $@ $<

string.o: string.c string.h
    $(CC) $(CFLAGS) -o $@ $<
```

- Améliorons notre makefile
 - Utilisons des variables pour éviter les redites
 - *Utilisons des règles*

Encore mieux avec l'usage d'une règle de compilation.

Makefile a beaucoup de règles par défaut et on ne souhaite pas toujours qu'elles soient appliquées.

L'option -r interdit l'usage des règles par défaut.
\$ make -r all

L'option -d affiche ce que fait make, c'est pour le debug du makefile.
\$ make -d all

Makefile

```
OBJECTS=main.o string.o
EXECUTABLE=program

CC=gcc
CFLAGS=-g -Wall -c
LD=gcc
LDFLAGS=-g

all: $(EXECUTABLE)

clean:
    -rm $(EXECUTABLE) $(OBJECTS)

$(EXECUTABLE): $(OBJECTS)
    $(LD) $(LDFLAGS) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<

main.o: main.c string.h

string.o: string.c string.h
```

Better Makefile ?

37

- Peut-on améliorer encore ?

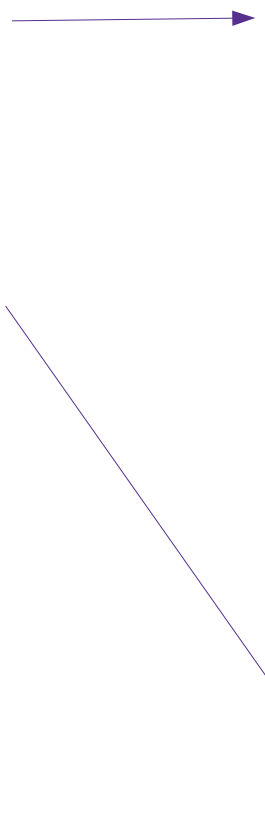
- Premier point

- Comment séparer les sources et les objets dans des dossiers différents, c'est à dire avoir un dossier « build »

- Deuxième point

- Pas de gestion des dépendances à la main qui est généralement source d'erreurs

Makefile



```
OBJECTS=main.o string.o
EXECUTABLE=program

CC=gcc
CFLAGS=-g -Wall -c
LD=gcc
LDFLAGS=-g

all: $(EXECUTABLE)

clean:
    -rm $(EXECUTABLE) $(OBJECTS)

$(EXECUTABLE): $(OBJECTS)
    $(LD) $(LDFLAGS) -o $@ $^

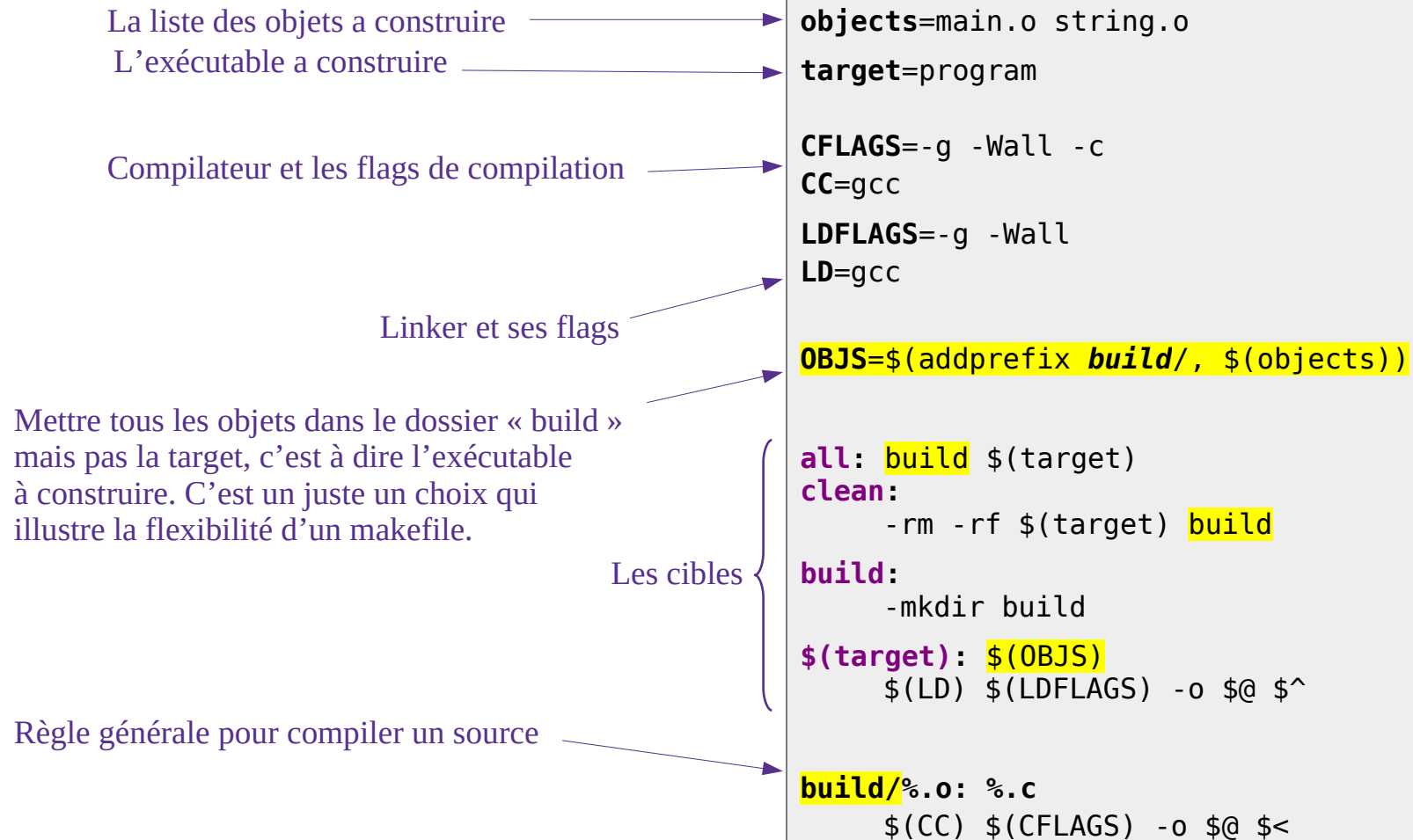
%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<

main.o: main.c string.h

string.o: string.c string.h
```

Makefile – Dossier « build »

38



Makefile – Les sources par wildcard

39

Gestion de la liste des sources
à compiler, automatique,
et non plus une liste manuelle
des objets.

Remplace les « .c » par des « .o »

Mettre tous les objets dans le dossier « build »

```
sources=$(wildcard *.c)
target=program

CFLAGS=-g -Wall -c
CC=gcc
LDFLAGS=-g -Wall
LD=gcc

objs=$(sources:%.c=%.o)
OBJS=$(addprefix build/, $(objs))

all: build $(target)
clean:
    -rm -rf $(target) build
build:
    -mkdir build

$(target): $(OBJS)
    $(LD) $(LDFLAGS) -o $@ $^

build/%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<
```

Makefile – Les sources par wildcard

40

Gestion de la liste des sources
pour du C et de l'assembleur

Remplace les « .c » par des « .o »

Remplace les « .s » par des « .o »

Les règles pour le C et l'assembleur

```
c-sources=$(wildcard *.c)
s-sources+=$(wildcard *.s)

target=program

CFLAGS=-g -Wall -c
CC=gcc
LDFLAGS=-g -Wall
LD=gcc

objs=$(c-sources:%.c=%.o)
objs+=$(s-sources:%.s=%.o)
OBJS=$(addprefix build/, $(objs))

all: build $(target)
clean:
    -rm -rf $(target) build
build:
    -mkdir build
$(target): $(OBJS)
    $(LD) $(LDFLAGS) -o $@ $^

build/%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<
build/%.o: %.s
    $(CC) $(CFLAGS) -o $@ $<
```


Développer Modulaire – Automated Dependencies

41

Demander à gcc de gérer les dépendances, dans le dossier « build » pour que « make clean » les effaces.

Règle générale pour compiler un source, rajout des « dependency flags »

Inclusion des dépendances générées par gcc
Ce sont des « makefiles » à inclure.
gcc les génère, à notre demande, dans le dossier build, d'où l'intérêt d'avoir ce dossier pour y mettre tout ce qui est généré : object files et makefiles.

```
objects=main.o string.o
target=program

CFLAGS=-g -Wall -I. -c
CC=gcc
LDFLAGS=-g -Wall
LD=gcc

OBSJ=$(addprefix build/, $(objects))
DEPFLAGS=-MT $@ -MMD -MP -MF build/$*.d

all: build $(target)
clean:
    -rm -rf $(target) build
build:
    -mkdir build

$(target): $(OBSJ)
    $(LD) $(LDFLAGS) -o $(target) $(OBSJ)

build/%.o: %.c
    $(CC) $(CFLAGS) $(DEPFLAGS) -o $@ $<

-include $(wildcard build/*.d)
```

Développer Modulaire – Managing Different Targets

42

Introduisons une variable pour contrôler si on produit un exécutable pour le debug (option -g) ou un exécutable optimisé (option -O3).

Nous allons en conséquence utiliser des dossiers différents pour contenir les fichiers « objets » et l'exécutable. Mais aussi différents drapeaux de compilation.

En effet, il ne faut pas mélanger !

```
objects=main.o string.o
target=program

debug = y

ifeq ($(debug),y)
    CFLAGS=-g
    LDFLAGS=-g
    BD=build/debug
else
    CFLAGS=-O3
    LDFLAGS=
    BD=build/run
endif

CFLAGS += -Wall -I. -c
CC=gcc
LDFLAGS += -Wall
LD=gcc
OBJS=$(addprefix $(BD)/, $(objects))
TARGET=$(BD)/$(target)
DEPFLAGS=-MT $@ -MMD -MP -MF $(BD)/$*.d
```

Développer Modulaire – Managing Different Targets

43

Avec l'assignation conditionnelle, on peut contrôler le comportement du makefile depuis le shell

```
$ make clean all
$ gdb ./build/debug/program

$ debug=n make all
$ ./build/run/program
```

FOO ?= bar



```
ifeq ($(origin FOO), undefined)
  FOO = bar
endif
```

```
objects=main.o string.o
```

```
target=program
```

```
debug ?= y
```

```
ifeq ($(debug),y)
```

```
  CFLAGS=-g
```

```
  LDFLAGS=-g
```

```
  BD=build/debug
```

```
else
```

```
  CFLAGS=-O3
```

```
  LDFLAGS=
```

```
  BD=build/run
```

```
endif
```

```
CFLAGS += -Wall -I. -c
```

```
CC=gcc
```

```
LDFLAGS += -Wall
```

```
LD=gcc
```

```
OBJS=$(addprefix $(BD)/, $(objects))
```

```
TARGET=$(BD)/$(target)
```

```
DEPFLAGS=-MT $@ -MMD -MP -MF $(BD)/$*.d
```

Développer Modulaire – Managing Different Targets

44

Il est parfois utile d'afficher des informations,
voici deux façons de faire :

```
objects=main.o string.o
target=program

debug ?= y
$(info var-debug: $(debug))

ifeq ($(debug),y)
  CFLAGS=-g
  LDFLAGS=-g
  BD=build/debug
else
  CFLAGS=-O3
  LDFLAGS=
  BD=build/run
endif

CFLAGS += -Wall -I. -c
CC=gcc
LDFLAGS += -Wall
LD=gcc

OBJS=$(addprefix $(BD)/, $(objects))
TARGET=$(BD)/$(target)
DEPFLAGS=-MT $@ -MMD -MP -MF $(BD)/$*.d

all: $(BD) $(TARGET)
    @echo "var-debug: $(debug)"
```

- Comment travailler pour réussir ?
 - Il faut reprendre les transparents
 - Il faut donc poser les questions nécessaires pour bien comprendre
 - Il faut aussi faire, mettre en pratique
 - Comprendre ne sera pas suffisant
 - Il faudra y revenir pour mémoriser sur le long terme
- Rappelez-vous
 - Le but n'est pas simplement d'avoir une note
 - Mais d'apprendre un savoir-faire pérenne
- Vos notes / vos exemples
 - Vous aurez des makefiles simples au début et plus compliqué au fil du temps
 - Vous aurez donc besoins de ces compétences dans 6 mois, dans 1 an, etc.