Réalisation d'un logiciel informatique en C

PHELMA

Sommaire

Grenoble INP phelma

Projet d'équipe multifichier en programmation C [Kernighan et Ritchie, 2004]

Débutant = un logiciel est réalisé par un(e) seul(e) programmeur(se) à partir d'un seul fichier source.

Projet d'équipe multifichier en programmation C [Kernighan et Ritchie, 2004]

Débutant = un logiciel est réalisé par un(e) seul(e) programmeur(se) à partir d'un seul fichier source.

Réalité = un logiciel est réalisé par une équipe avec plusieurs fichiers sources

Projet d'équipe multifichier en programmation C [Kernighan et Ritchie, 2004]

Débutant = un logiciel est réalisé par un(e) seul(e) programmeur(se) à partir d'un seul fichier source.

Réalité = un logiciel est réalisé par une équipe avec plusieurs fichiers sources

Pourquoi plusieurs fichiers ?

- modularité
- partage du travail
- débogage/validation aisée
- grande ré-utilisabilité

Projet d'équipe multifichier en programmation C [Kernighan et Ritchie, 2004]

Débutant = un logiciel est réalisé par un(e) seul(e) programmeur(se) à partir d'un seul fichier source.

Réalité = un logiciel est réalisé par une équipe avec plusieurs fichiers sources

Pourquoi plusieurs fichiers ?

- modularité
- partage du travail
- débogage/validation aisée
- grande ré-utilisabilité

Quels types de fichier ?

- fichier.c
- fichier.h

Les fichiers **.c**

Par convention, les fichiers **.c** contiennent les définitions de fonctions et sont de la forme :

```
<Commandes du préprocesseur>
<Fonctions>
```

Structures des fonctions :

```
<type de retour> nom_de_fonction(<Déclaration des arguments>)
{ <Déclaration locales>
   <instructions>
}
```

Les fichiers **.c**

Par convention, les fichiers **.c** contiennent les définitions de fonctions et sont de la forme :

```
<Commandes du préprocesseur>
<Fonctions>
```

Structures des fonctions :

```
<type de retour> nom_de_fonction(<Déclaration des arguments>)
{ <Déclaration locales>
   <instructions>
}
```

Comment savoir ce que l'on met dans un fichier **.c** ?

Les fichiers **.c** : Exemple

Les fichiers **.c** contiennent soit : un main, un test ou un ensemble de fonctions réalisant une tâche du programme.

Exemple : Soit un programme de traitement d'images. Il sera nécessaire d'avoir des fonctions pour :

```
/* test_image.c teste
   le programme */
main() {
  Image im = readBMP("
   coucou.bmp");
  im = inverse(im);
  writeBMP(im,"
   coucou_inverse.bmp")
   ;
}
```

```
/* io_image.c gere les
   fichiers*/
Image readBMP(char*
   nom_fichier) {...
}
Image readJPG(char*
   nom_fichier) {...
}
void writeBMP(Image im,
    char* nom_fichier)
   {...
}
```

```
/* transforme_image.c
   traitements
de base de l'image*/
#include<math.h>
Image inverse(Image im)
    {...
}
Image erode(Image im)
   {...
}
Image masque(Image im,
   int seuil) {...
}
```

Les fichiers **.c** : Exemple

Les fichiers **.c** contiennent soit : un main, un test ou un ensemble de fonctions réalisant une tâche du programme.

Exemple : Soit un programme de traitement d'images. Il sera nécessaire d'avoir des fonctions pour :

```
/* test_image.c teste
   le programme */
main() {
  Image im = readBMP("
   coucou.bmp");
  im = inverse(im);
  writeBMP(im,"
   coucou_inverse.bmp")
   ;
}
```

```
/* io_image.c gere les
   fichiers*/
Image readBMP(char*
   nom_fichier) {...
}
Image readJPG(char*
   nom_fichier) {...
}
void writeBMP(Image im,
    char* nom_fichier)
   {...
}
```

```
/* transforme_image.c
   traitements
de base de l'image*/
#include<math.h>
Image inverse(Image im)
    {...
}
Image erode(Image im)
   {...
}
Image masque(Image im,
   int seuil) {...
}
```

Où mettre la définition de Image ?

Comment `test_image.c` **peut-il compiler ?**

Les **h**eaders (en-têtes)

Par convention, les headers contiennent :

- les définitions de types (structure, union...)
- les constantes et énumérations
- les prototypes de fonctions

Les headers sont généralement de la forme suivante :

```
<Commandes du préprocesseur>
<Définitions de types>
<Déclarations de Fonctions>
```

Les **h**eaders : Exemple

Image est définie dans un en-tête et les prototypes des fonctions dans un autre.

```
/* image.h */
#define TAILLE_IMAGE_MAX 1E9
typedef double ** Image;
typedef struct complx{float re,im;} Complexe;

/*fonctions_image.h*/
#include"image.h"
Image readBMP(char* nom_fichier);
Image readJPG(char* nom_fichier);
void  writeBMP(Image im, char* nom_fichier);
Image inverse(Image im);
Image erode(Image im);
Image masque(Image im, int seuil);
```

Exemple complet

```c
/* image.h */
#define TAILLE_IMAGE_MAX 1E9
typedef double ** Image;
typedef struct complx{float re,im
   ;} Complexe;
```

```c
/* fonctions_image.h */
#include"image.h"
Image readBMP(char* nom_fichier);
...
Image erode(Image im);
```

```c
/* test_image.c teste le
   programme */
#include"image.h"
#include"fonctions_image.h"
main() {
  Image im = readBMP("coucou.bmp"
   );
  im = inverse(im);
  writeBMP(im,"coucou_inverse.bmp
   ");
}
```

```c
/* io_image.c gere les fichiers*/
#include"image.h"
Image readBMP(char* nom_fichier)
   {...
}
Image readJPG(char* nom_fichier)
   {...
}
void writeBMP(Image im, char*
   nom_fichier) {...
}
```

Retour sur la compilation

Compilation ?

Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

Compilation ?

Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

1. la **pré-compilation** : exécute notamment les directives du préprocesseur (remplacement de textes, inclusion de fichiers, suppression des commentaires) et produit une sortie **texte**.

Compilation ?

Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

1. la **pré-compilation** : exécute notamment les directives du préprocesseur (remplacement de textes, inclusion de fichiers, suppression des commentaires) et produit une sortie **texte**.
2. la **compilation** : effectue l'analyse syntaxique, la vérification des erreurs et produit une sortie **texte** dans un langage pivot (ici assembleur)

Grenoble INP phelma

Compilation ?

Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

1. la **pré-compilation** : exécute notamment les directives du préprocesseur (remplacement de textes, inclusion de fichiers, suppression des commentaires) et produit une sortie **texte**.
2. la **compilation** : effectue l'analyse syntaxique, la vérification des erreurs et produit une sortie **texte** dans un langage pivot (ici assembleur)
3. l'**assemblage** : traduit le code assembleur en code **binaire** pour l'architecture cible (Intel, ARM, MIPS, etc.)

Grenoble INP phelma

Compilation ?

Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

1. la **pré-compilation** : exécute notamment les directives du préprocesseur (remplacement de textes, inclusion de fichiers, suppression des commentaires) et produit une sortie **texte**.
2. la **compilation** : effectue l'analyse syntaxique, la vérification des erreurs et produit une sortie **texte** dans un langage pivot (ici assembleur)
3. l'**assemblage** : traduit le code assembleur en code **binaire** pour l'architecture cible (Intel, ARM, MIPS, etc.)
4. l'**édition des liens** : rassemble les différents fichiers binaires en un exécutable **binaire** unique (ou une bibliothèque)

Grenoble INP phelma

Compilation ?

Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

1. la **pré-compilation** : exécute notamment les directives du préprocesseur (remplacement de textes, inclusion de fichiers, suppression des commentaires) et produit une sortie **texte**.
2. la **compilation** : effectue l'analyse syntaxique, la vérification des erreurs et produit une sortie **texte** dans un langage pivot (ici assembleur)
3. l'**assemblage** : traduit le code assembleur en code **binaire** pour l'architecture cible (Intel, ARM, MIPS, etc.)
4. l'**édition des liens** : rassemble les différents fichiers binaires en un exécutable **binaire** unique (ou une bibliothèque)
   - statique : recopie chaque binaire dans l'exécutable

Grenoble INP phelma

Compilation ?

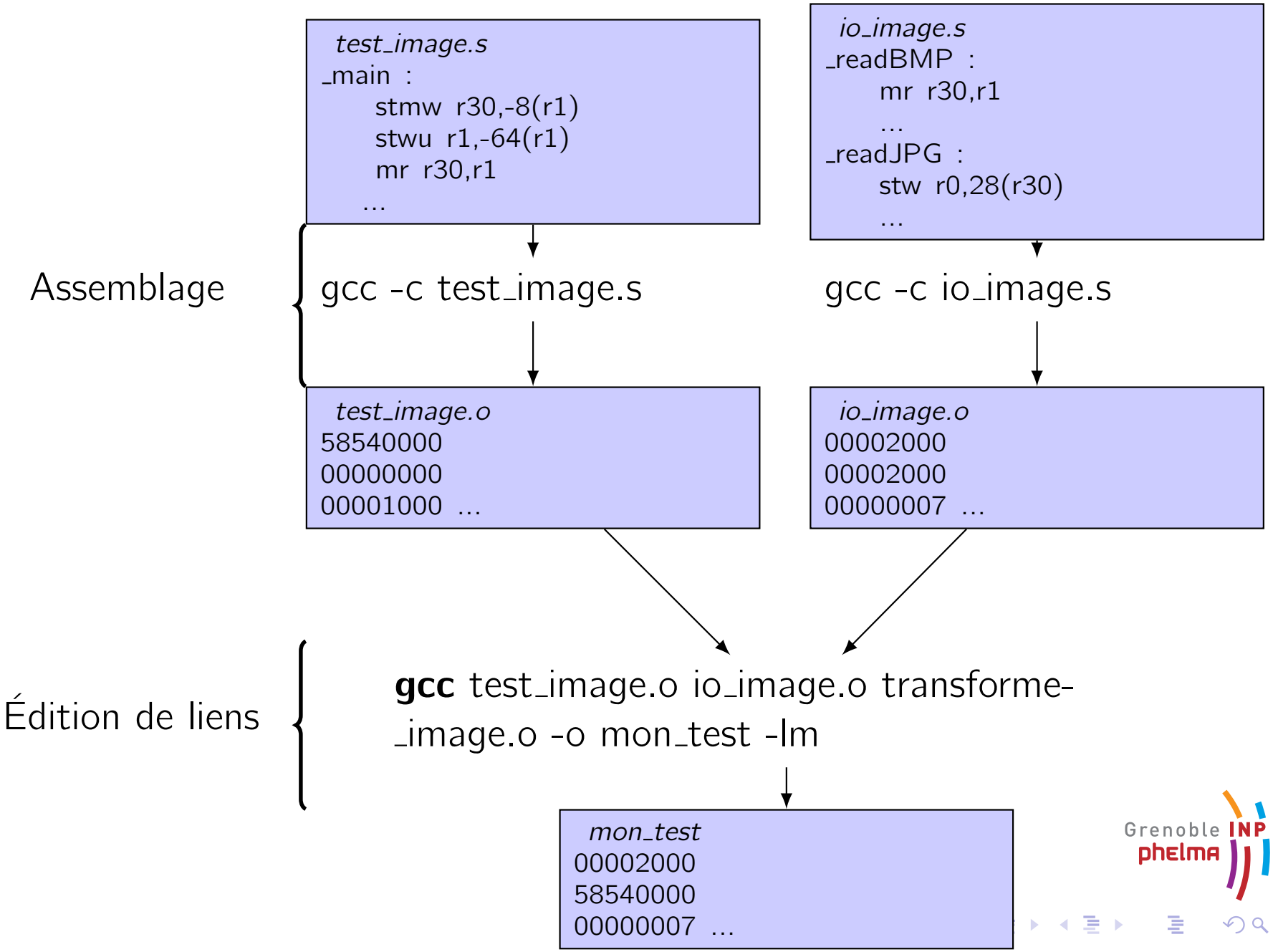
Compilation : terme impropre, ce n'est qu'une étape de la chaîne.

Les étapes de création d'un exécutable avec gcc :

1. la **pré-compilation** : exécute notamment les directives du préprocesseur (remplacement de textes, inclusion de fichiers, suppression des commentaires) et produit une sortie **texte**.
2. la **compilation** : effectue l'analyse syntaxique, la vérification des erreurs et produit une sortie **texte** dans un langage pivot (ici assembleur)
3. l'**assemblage** : traduit le code assembleur en code **binaire** pour l'architecture cible (Intel, ARM, MIPS, etc.)
4. l'**édition des liens** : rassemble les différents fichiers binaires en un exécutable **binaire** unique (ou une bibliothèque)
   - statique : recopie chaque binaire dans l'exécutable
   - dynamique : les binaires partagés ne sont recopiés qu'au chargement en mémoire

La chaîne de compilation

La chaîne de compilation



Le préprocesseur

Le Préprocesseur[The GCC team, 2011] effectue des traitements lexicaux AVANT la compilation.
Il modifie donc le code source !

Parmi les directives du préprocesseur :

`#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`, `#endif`

Les deux directives les plus connues sont le `#include` et le `#define`.

Inclusion de fichiers sources :

```
#include "NOM DE FICHIER"
#include <NOM DE FICHIER>
```

Définition de Macro :

```
#define N 10
#define MESSAGE_ACCUEIL "bienvenu chez nous"
```

Variables prédéfinies du préprocesseur :

`__LINE__`, `__STDC__`, `__DATE__`, `__FILE__`, `__TIME__`

Raccordement de ligne : “\” en fin de ligne

Nécessité de précompilation conditionnelle

Si on tente de compiler test_image.c alors

```
In file included from fonctions_image.h:1:0,
                 from test_image.c:2:
image.h:3:16: erreur: redefinition of 'struct complx'
```

Nécessité de précompilation conditionnelle

Si on tente de compiler test_image.c alors

```
In file included from fonctions_image.h:1:0,
                 from test_image.c:2:
image.h:3:16: erreur: redefinition of 'struct complx'
```

Que se passe-t-il ? En exécutant

```
$ gcc -E test_image.c
```

```
#define TAILLE_IMAGE_MAX 1E9
typedef double ** Image;
typedef struct complx {float re,im;} Complexe;
...
typedef struct complx {float re,im;} Complexe;
...
```

La définition de structure est incluse deux fois !

Inclusion conditionnelle avec des commandes du préprocesseur

les `includes` emboîtés d'un même en-tête conduisent à des redéfinitions de types ou de fonctions

La solution c'est l'inclusion conditionnelle qui permet de pré-compiler certaines parties du code source selon certains critères en utilisant les directives :

```
#if #ifdef #ifndef
#else #endif #elif
```

Exemple d'inclusion conditionnelle

```
/* image.h */
#ifndef __IMAGE_H_
#define __IMAGE_H_
#define TAILLE_IMAGE_MAX 1E9
typedef double ** Image;
typedef struct complx{float re,im;} Complexe;
#endif

/*fonctions_image.h*/
#ifndef __FONCTIONS_IMAGE_H_
#define __FONCTIONS_IMAGE_H_
#include"image.h"
Image readBMP(char* nom_fichier);
Image readJPG(char* nom_fichier);
void  writeBMP(Image im, char* nom_fichier);

Image inverse(Image im);
Image erode(Image im);
Image masque(Image im, int seuil);
#endif
```

Automatisation de la compilation : Makefile

Automatiser la compilation : make [The GNU team, 2010]

make est un utilitaire présent dans toutes les distributions de Linux.
`$ man make`

Détermine automatiquement quelles sont les parties d'un gros programme qu'il faut recompiler grâce à un fichier Makefile qui décrit

- les dépendances entre les fichiers
- les commandes nécessaires à la mise à jour de chacun d'entre eux

Utilisation (le fichier `Makefile` doit être dans le répertoire courant) :
`$ make nom_de_la_cible`

make a une mise à jour intelligente des cibles.
make n'est pas destiné uniquement au langage C ! ! !

Le Makefile

Dit à make comment créer un fichier cible à partir de fichiers sources.
Fonctionnement à base de règles.

```
cible : dépendance_1 dépendance_2 ... dépendance_n
[TABULATION] commande shell
```

Exemple :
pour faire progexe à partir de prog.c

```
progexe : prog.o
        gcc -o progexe prog.o
prog.o: prog.c
        gcc -c prog.c
```

Makefile plus générique

Déclaration de constantes

```
CIBLE = progexe
DEPENDANCE = prog.o
OPTCOMPIL = -g
$(CIBLE): $(DEPENDANCE)
	gcc -o $(CIBLE) $(DEPENDANCE)
prog.o: prog.c
	gcc $(OPTCOMPIL) -c prog.c
```

Makefile plus générique

Déclaration de constantes

```
CIBLE = progexe
DEPENDANCE = prog.o
OPTCOMPIL = -g
$(CIBLE): $(DEPENDANCE)
	gcc -o $(CIBLE) $(DEPENDANCE)
prog.o: prog.c
	gcc $(OPTCOMPIL) -c prog.c
```

Exécute :

```
$ make progexe
gcc -g -c prog.c
gcc -o progexe prog.o
```

Makefile encore plus générique

```
$@  Le nom de la cible
$<  Le nom de la première dépendance
$^  La liste des dépendances
$*  Le nom de la cible sans suffixe

progexe: prog.o fonctions.o
        gcc -o $@ $^
%.o: %.c
        gcc -c $<
```

Makefile encore plus générique

```
$@  Le nom de la cible
$<  Le nom de la première dépendance
$^  La liste des dépendances
$*  Le nom de la cible sans suffixe

progexe: prog.o fonctions.o
        gcc -o $@ $^
%.o: %.c
        gcc -c $<
```

Exécute :

```
$ make
gcc -c prog.c
gcc -c fonction.c
gcc -o progexe prog.o fonction.o
```

Exercices Makefile

1. Écrire un Makefile avec toutes les règles pour compiler `test_image.c, io_image.c, fonctions_image.c` et créer l'exécutable `test_image1`. Quelles sont les commandes pour créer l'exécutable, lancer l'exécutable et compiler uniquement `fonctions_image.c` ?
2. Écrire un Makefile plus générique pour réunir les options de compilation dans une variable et n'écrire qu'une règle pour la compilation des fichiers c. Quelle est la commande pour compiler uniquement `fonctions_image.c` ?
3. Écrire une règle pour effacer tous les fichiers .o du répertoire courant.
4. Écrire une règle pour lister tous les fichiers sources.
5. Écrire une règle pour créer l'archive de tous les fichiers .c et .h du répertoire courant.

Corriger les erreurs : Déboguer

Messages d'erreurs à la compilation

warning : initialization makes integer from pointer without a cast
essaye d'affecter un pointeur à un entier :

```
char c = "\n";  /* incorrecte */
```

warning : control reaches end of non-void function
il manque un return dans une fonction non void

```
int display (const char * str) { printf("%s\n", str);}
```

Messages d'erreurs à l'édition des liens

undefined reference to 'foo'
collect2 : ld returned 1 exit status
utilise une fonction ou une variable qui n'existe pas dans le programme.

/usr/lib/crt1.o(.text+0x18) : undefined reference to 'main'
il manque une fonction main pour pouvoir réaliser un exécutable

Messages d'erreurs à l'exécution

Segmentation fault (ou Bus error)

- libération d'un pointeur non alloué ou déjà libéré.
- accès en dehors des indices d'un tableau
- utilisation incorrecte de `malloc, free` etc.
- utilisation de `scanf` avec des arguments invalides

floating point exception
division par zéro ou autre opération illégale.

Data Display Debugger (ddd)

Pourquoi ? Comment ?

Rappel sur le débogage

Déboguer c'est chercher et corriger les erreurs d'un programme.

En C on a surtout des erreurs :

- de syntaxe : à la compilation (*error : expected ';' before 'for'*)
- de fonctionnement : à l'exécution (*Segmentation fault*)

Avant d'utiliser un débogueur, il faut d'abord **comprendre le message d'erreur et réfléchir...**

Pourquoi ?

Mise au point d'un programme ayant une syntaxe correcte mais dont l'exécution s'avère fantaisiste.

Compilation spéciale :

```
gcc -g monprogramme.c -o toto.exe
```

Exécution sous contrôle :

```
ddd   toto.exe
```

Exemple

```c
#include <stdio.h>

void affiche (int n, float u[], float v[]){
  int i;
  for (i=0;i<n;i++)
     printf("\n\t u[%d] = %f \t v[%d]=%f\n",i,u[i],i,v[i]);
  printf("\n\n");
}
```

```c
void decale (int n, float u[], float v[], float c){
  int i;
  for (i=0;i<=n;i++)
    v[i] = u[i] +c;
}
```

```c
main(){
  int i,dim=3;
  float x[3] = {1,2,3}, y[3] = {5,5,5};
  decale(dim,x,y,-1);
  affiche(dim,x,y);
}
```

Exécution

```
$ gcc PETDebugger.c -o exple.exe -g
$
$ ./exple.exe

u[0] =-1.000000 v[0]=0.000000
u[1] = 2.000000 v[0]=1.000000
u[2] = 3.000000 v[0]=2.000000
```

Exécution

```
$ gcc PETDebugger.c -o exple.exe -g
$
$ ./exple.exe

u[0] =-1.000000 v[0]=0.000000
u[1] = 2.000000 v[0]=1.000000
u[2] = 3.000000 v[0]=2.000000
```

Comment `u[0]` peut-il valoir -1 ?

Exécution

```
$ gcc PETDebugger.c -o exple.exe -g
$
$ ./exple.exe

u[0] =-1.000000 v[0]=0.000000
u[1] = 2.000000 v[0]=1.000000
u[2] = 3.000000 v[0]=2.000000
```

Comment `u[0]` peut-il valoir -1 ?

```
$ddd exple.exe
```

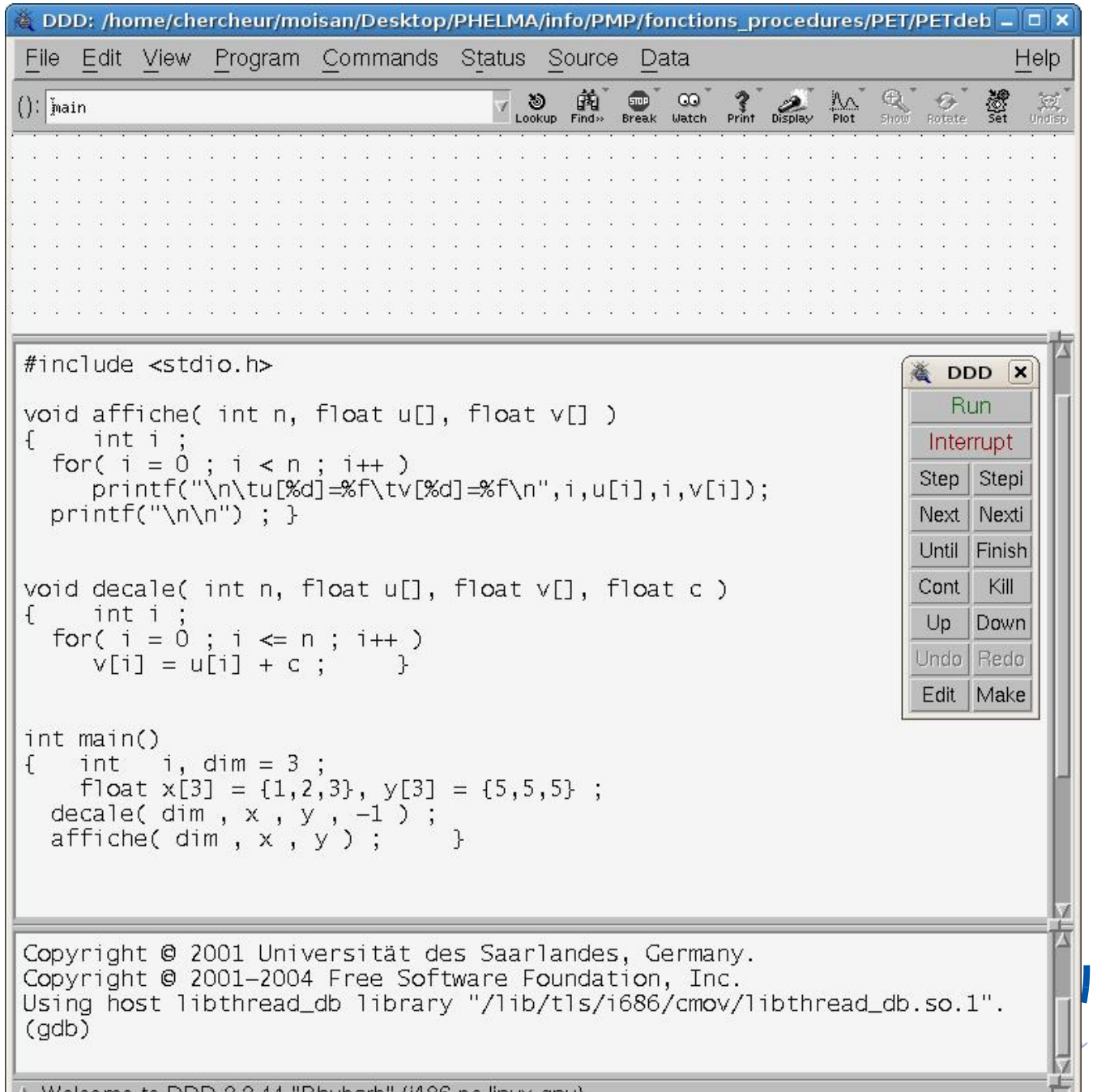
Comment ?

Fenêtre d'affichage

Code source

Panneau de commande

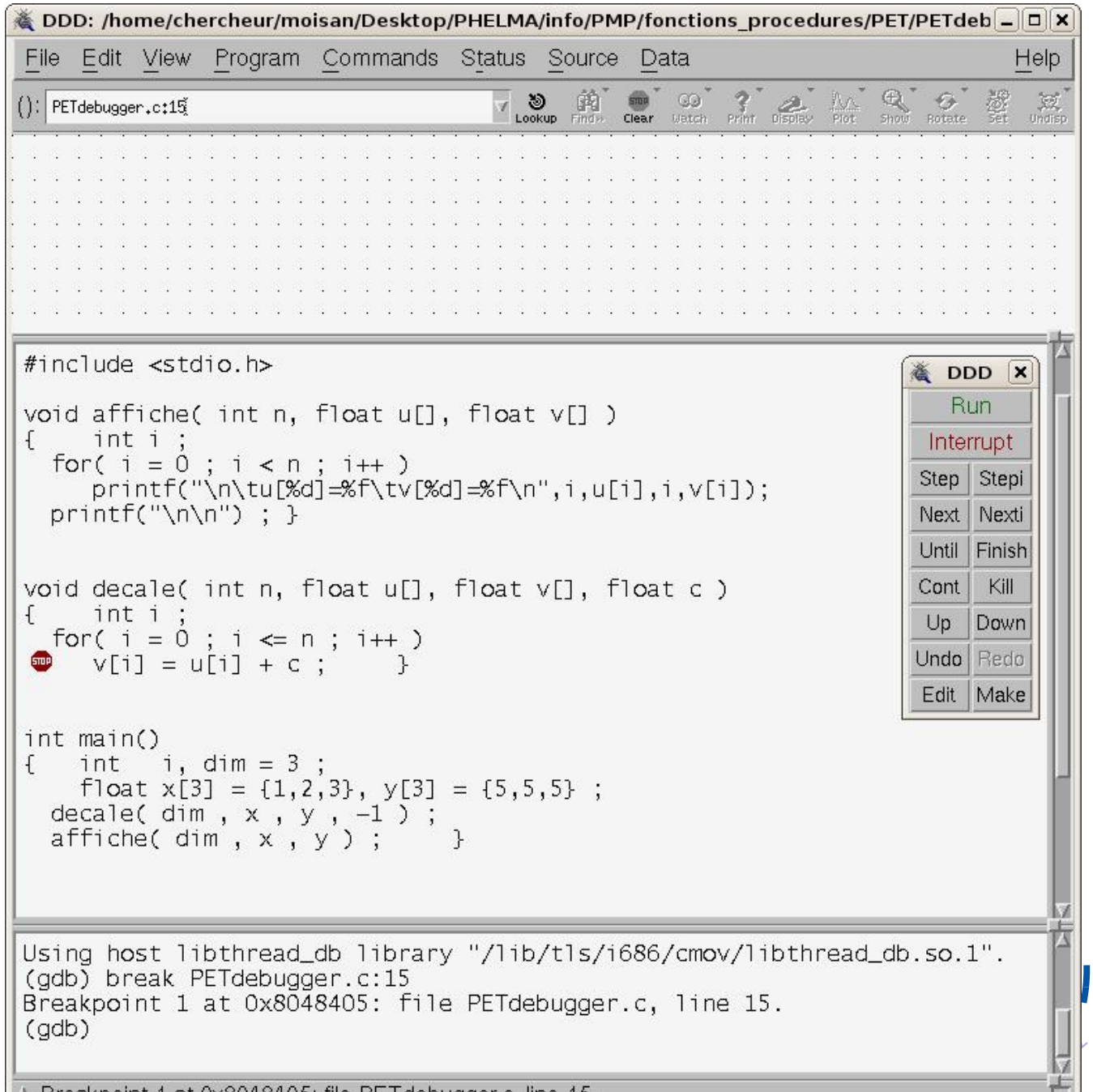
Fenêtre de dialogue



Mise en place d'un point d'arrêt

Pointer l'instruction choisie

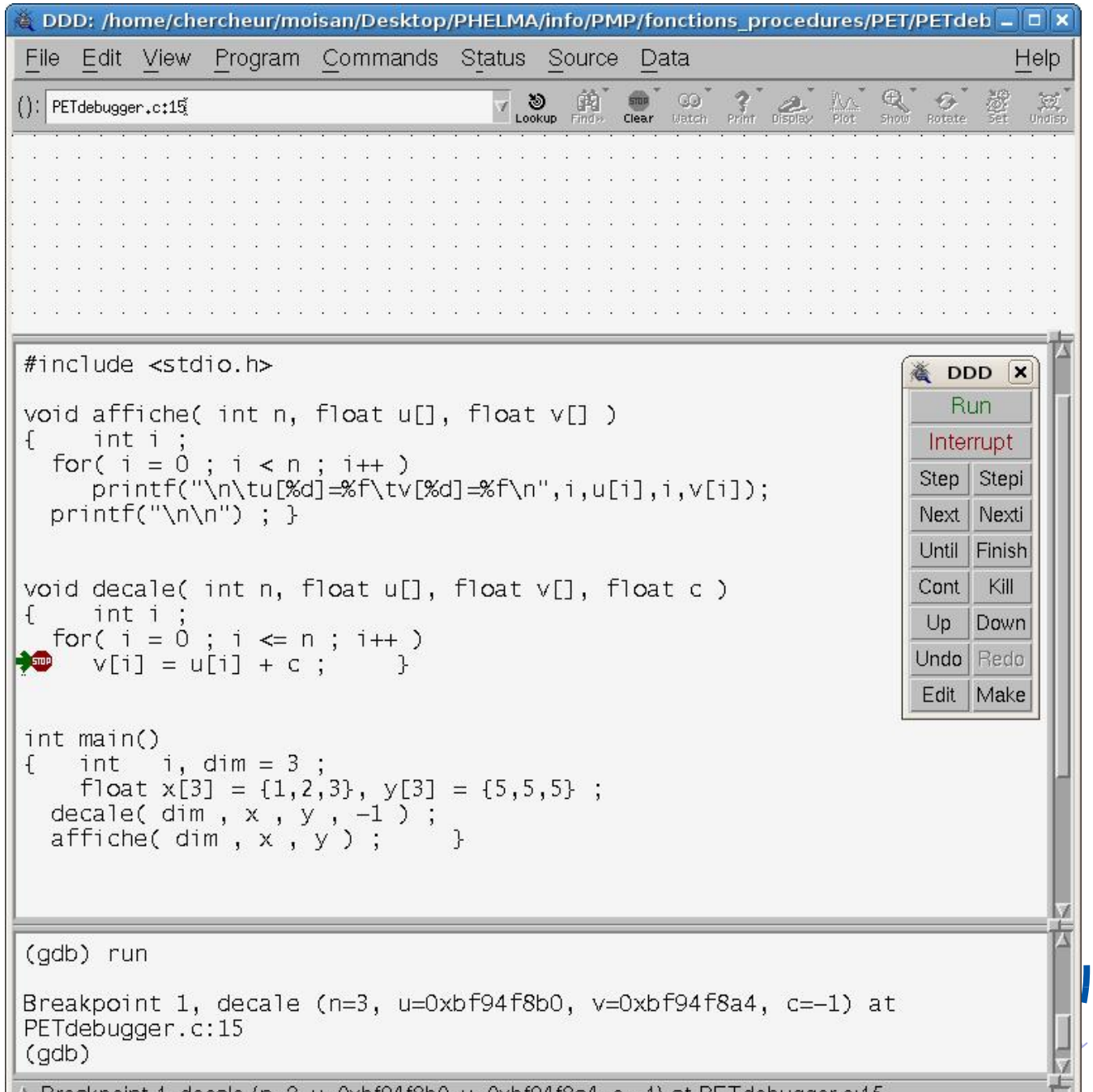
Clic droit : set breakpoint



Exécution jusqu'au point d'arrêt

Cliquer sur le bouton Run

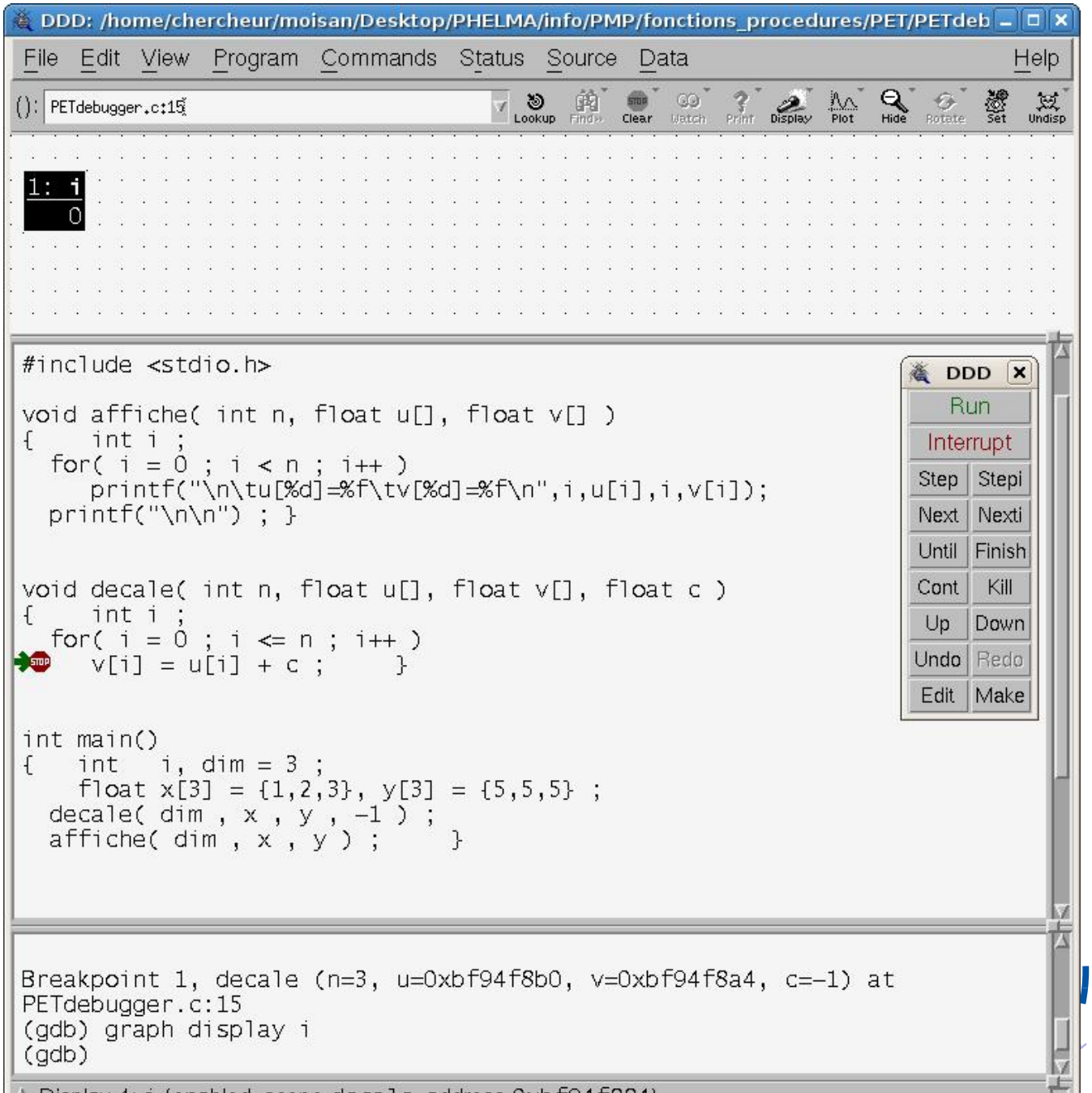
On part du début pour s'arrêter devant le stop



Affichage des variables locales

Pointer la variable choisie

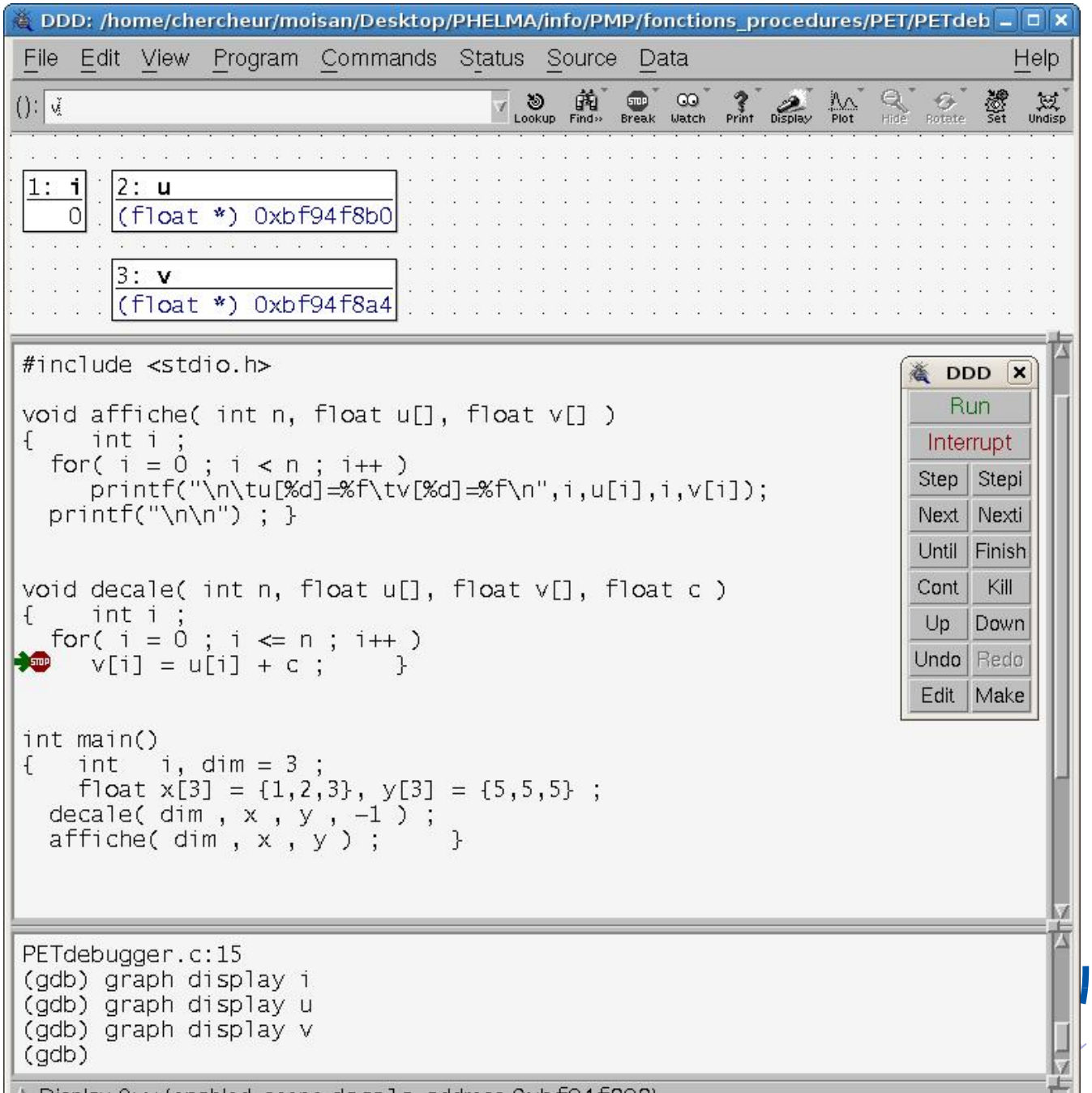
Clic droit : display



Affichage des tableaux

Pointer un tableau

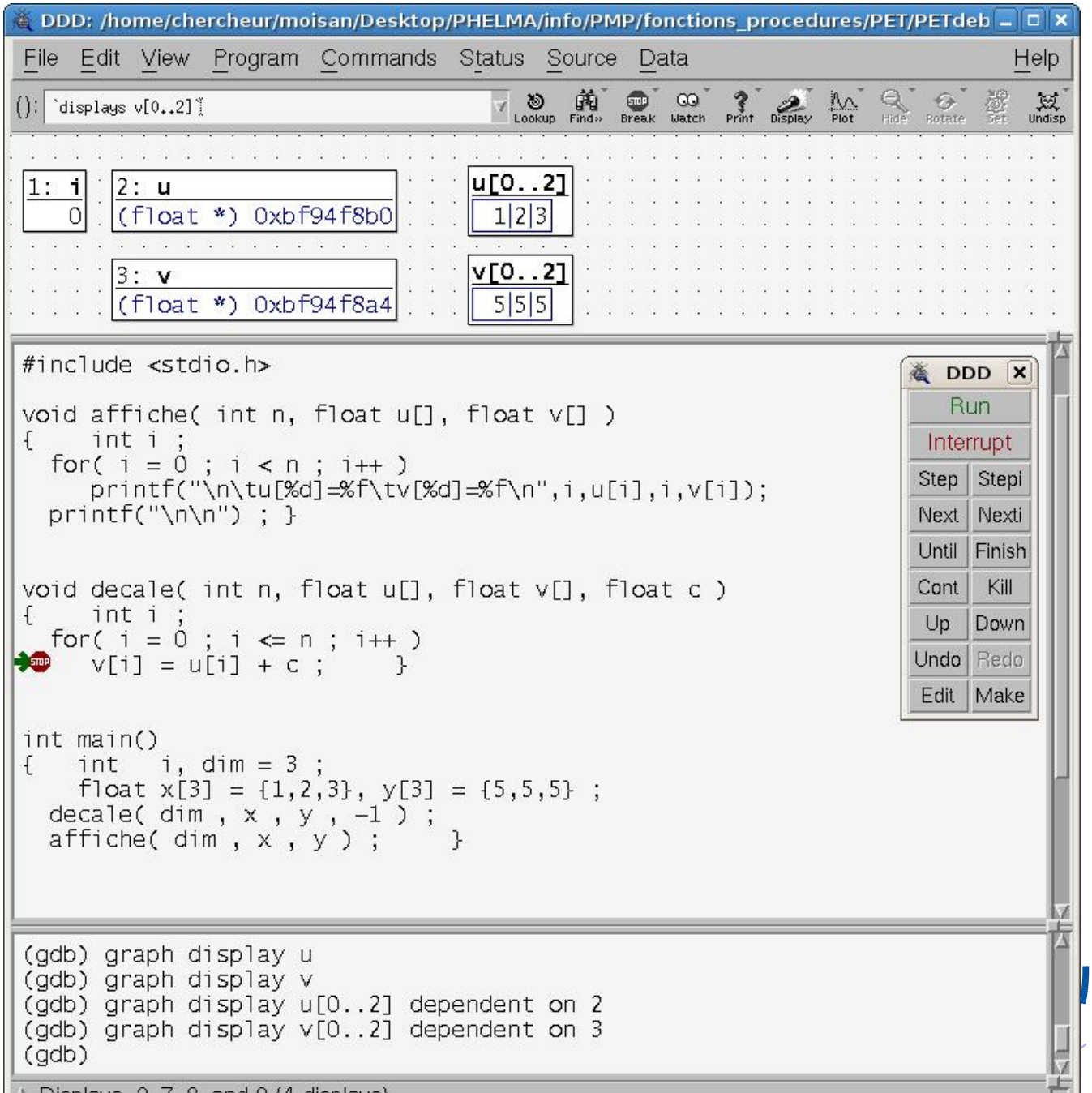
Clic droit : display donne une adresse



Affichage détaillé des tableaux

Pointer le tableau dans la fenêtre affichage

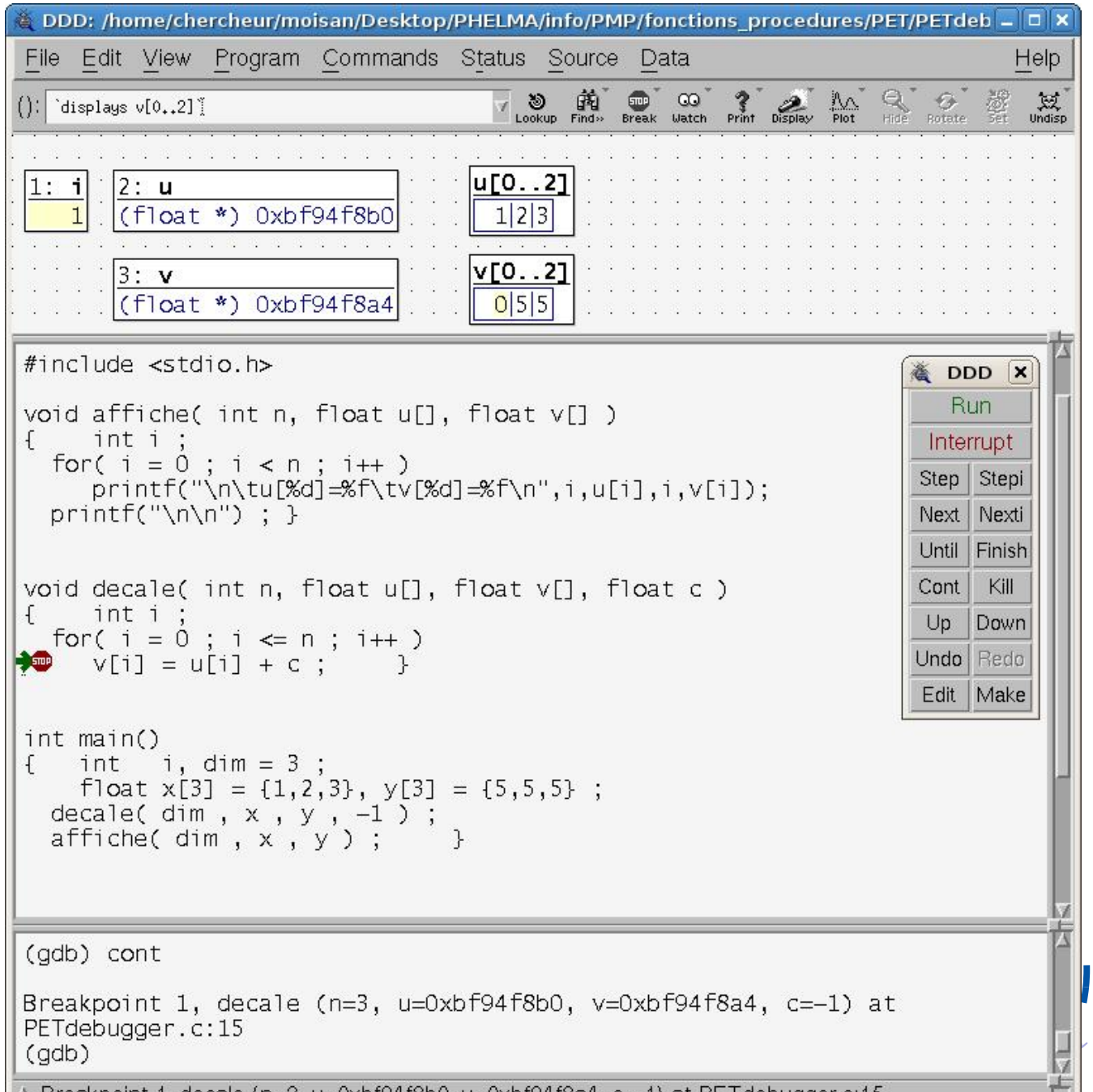
Clic droit : new display other compléter u[0 .. 2]



Continue

Cliquer sur le bouton Cont

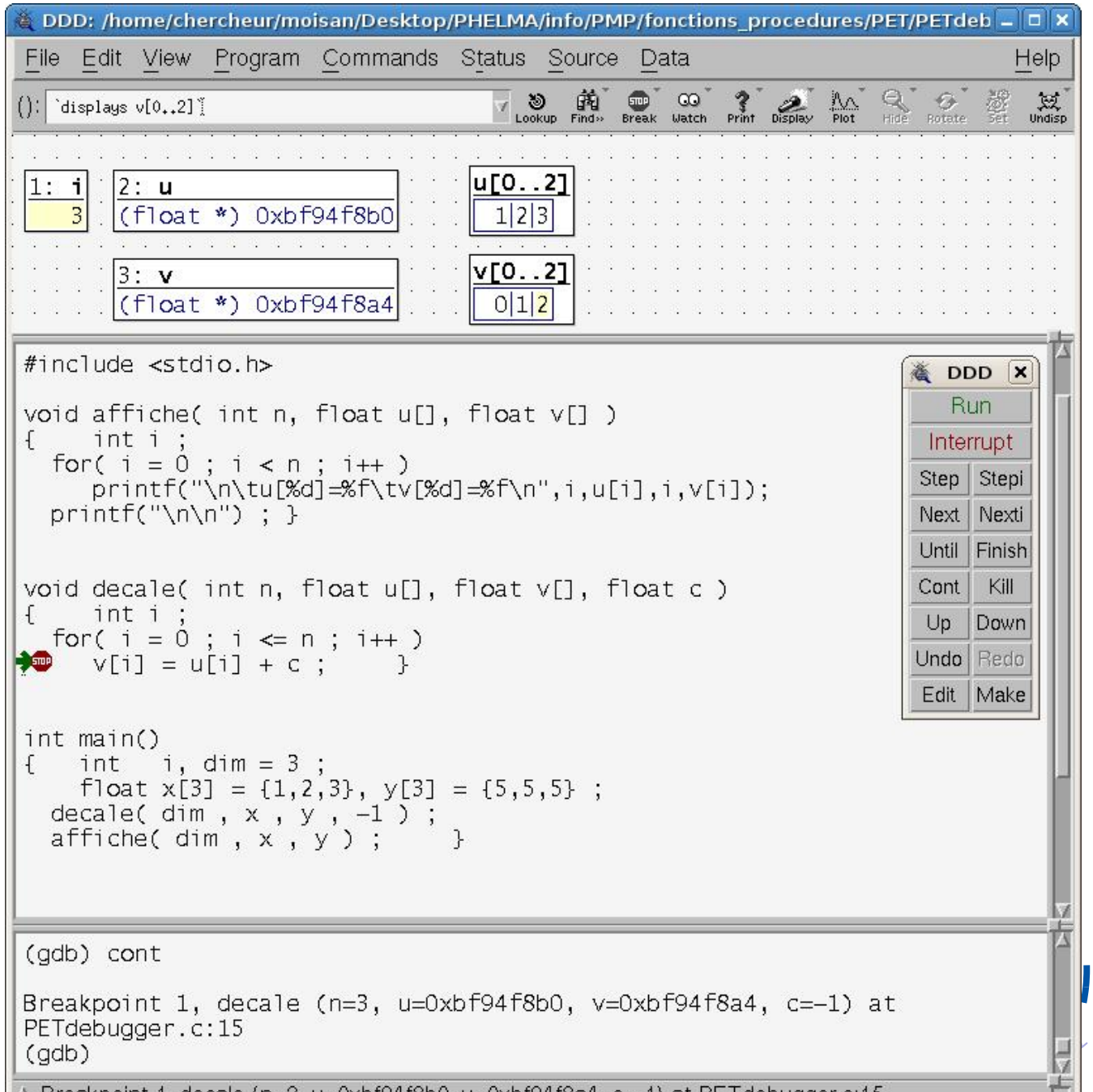
Exécution jusqu'au prochain stop
v[0] = 0
i = 1



Continue (2 fois)

Bouton Cont
Cont

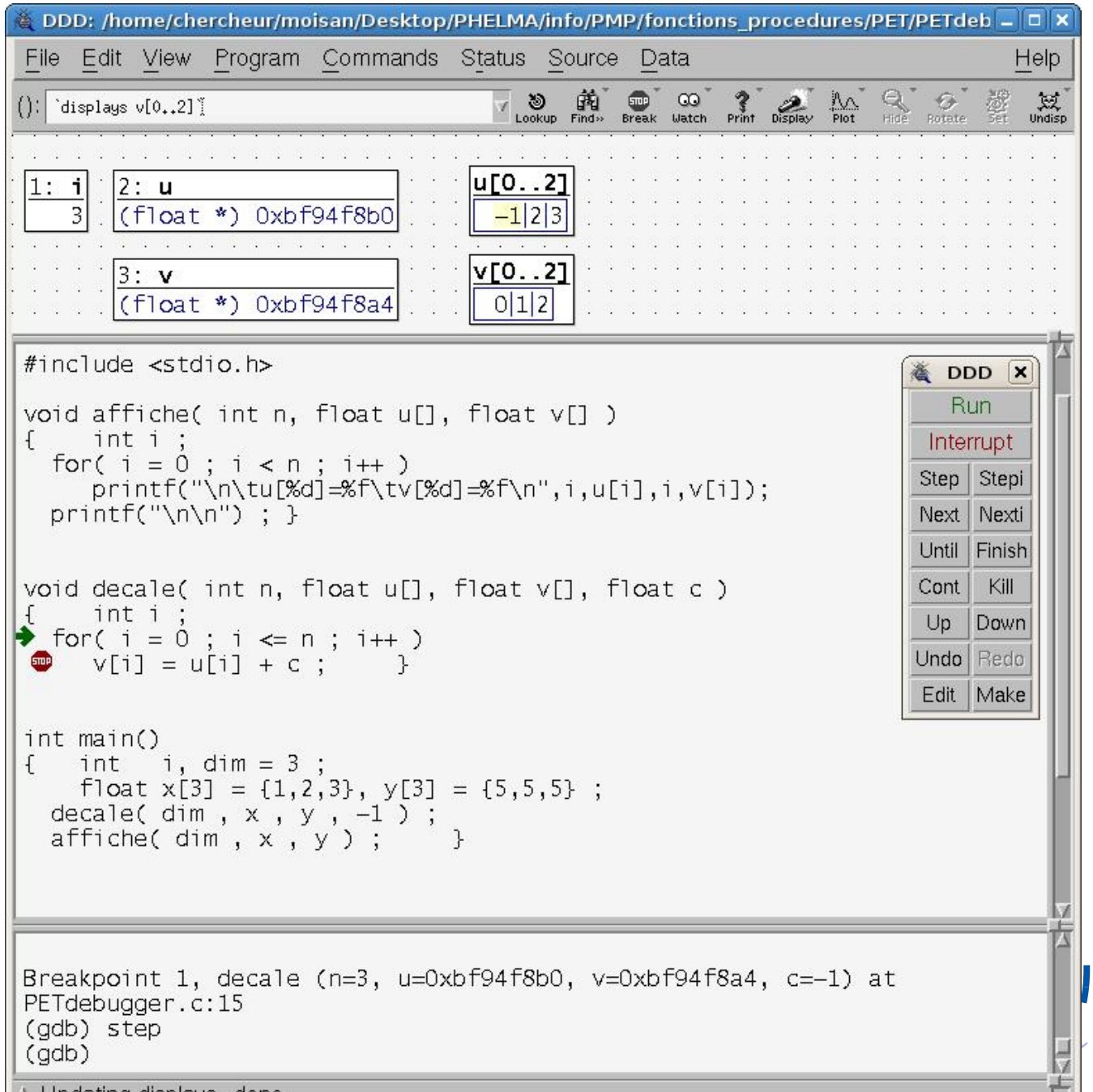
$v[2] = 2$
$i = 3$



Exécution pas à pas

Cliquer sur le bouton Step

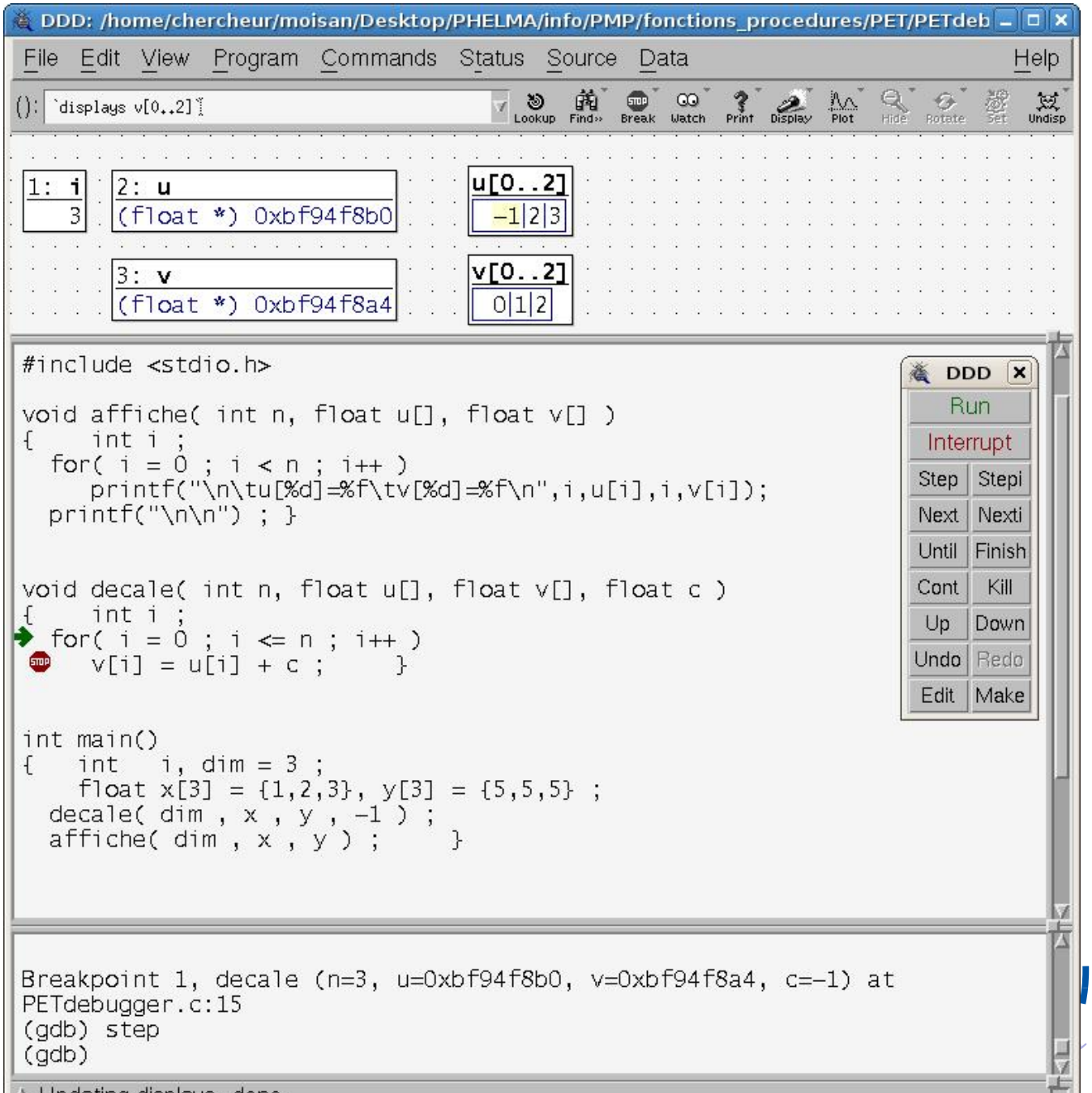
exécution de l'instruction suivante



Explication

i = 3
u[3] = ?
v[3] = ?-1
mais
v + 3
= u + 0

(8a4+3x4=8b0)

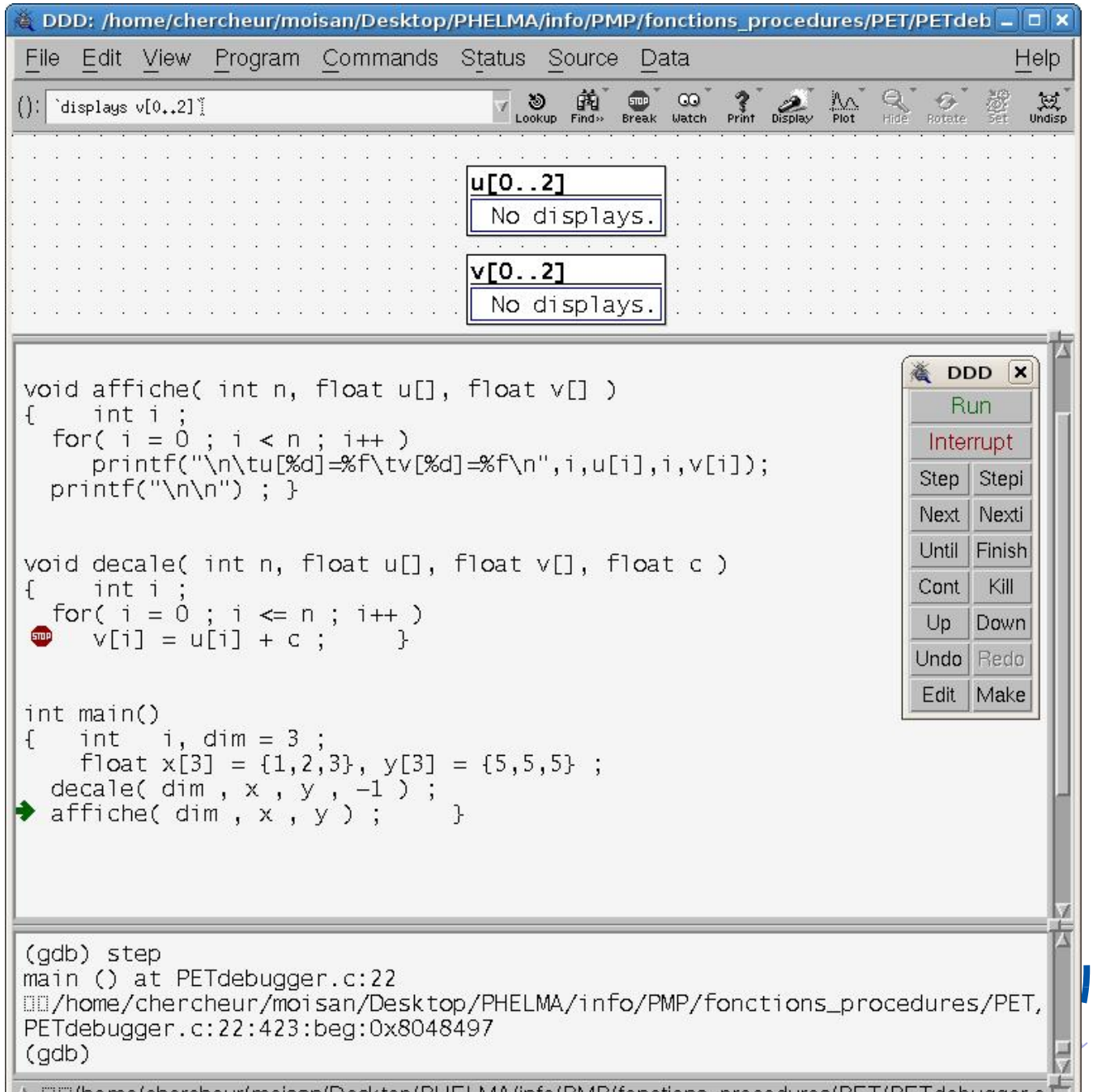


Retour au main

Step

Step

u et v sont inaccessibles dans cette fonction



Résumé des commandes du débogueur

step	exécute une ligne du code source.
next	exécute jusqu'à la prochaine ligne sans entrer dans les fonctions.
until	continue l'exécution jusqu'à la ligne pointée par la souris.
cont	continue l'exécution jusqu'au prochain arrêt.
finish	Pour aller jusqu'à la fin de la fonction.
kill	arrête le programme.

Débogueur en ligne de commande : Valgrind

Valgrind

Le déboguer permet d'exécuter le programme pas à pas mais n'indique pas directement quels sont les problèmes ni quelle en est la cause.
Valgrind est une suite d'outils de débug mémoire sous licence GPL, qui permet de détecter automatiquement des problèmes de gestion mémoire.

Fonctionnement :
Le programme à débugger doit être lancé dans l'environnement de Valgrind. Toutes les accès mémoires (p.ex. : malloc, free) sont interceptés pour être analysés.

```
$ vagrind ./monprog monoption monparam1
```

Plus d'info sur
`http://valgrind.org/docs/manual/QuickStart.html`.

Valgrind : accès mémoire invalide

```
/* acces_mem.c*/
...
main(){
  int tab[10], *p=NULL;
  p = calloc(10,sizeof(*p));
  p[10]=3; // ecriture invalide
  printf("val =%d\n", p[10]); // lecture invalide
...
}
```

On lance vagrind avec l'executable

```
$gcc acces_mem.c -g -o acces_mem_exe
$valgrind --leak-check=full ./acces_mem_exe
==3004== Invalid write of size 4
==3004==    at 0x400587: main (acces_mem.c:7)
==3004==  Address 0x51f1068 is 0 bytes after a block of size 40 alloc'd
==3004==    at 0x4C29E46: calloc (in /.../vgpreload_memcheck-amd64-linux.so)
==3004==    by 0x40057A: main (acces_mem.c:6)
==3004==
==3004== Invalid read of size 4
==3004==    at 0x400595: main (acces_mem.c:8)
==3004==  Address 0x51f1068 is 0 bytes after a block of size 40 alloc'd
```

Valgrind : détection de fuite mémoire

```c
/* fuite.c*/
main(){
  int tab[10], *p=NULL;
  p = calloc(10,sizeof(*p));
  p=tab; // fuite memoire ! on perd l'adresse de la memoire allouee
  ...
}
```

On lance vagrind avec l'option `--leak-check=full`

```
$gcc fuite.c -g -o fuite_exe
$valgrind --leak-check=full ./fuite_exe
==2867== HEAP SUMMARY:
==2867==     in use at exit: 40 bytes in 1 blocks
==2867==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==2867==
==2867== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2867==    at 0x4C29E46: calloc (in /.../vgpreload_memcheck-amd64-linux.so)
==2867==    by 0x40053A: main (fuite.c:4)
```

Méthode de travail

Méthode de développement

Pour les petits projets, un cycle en V est parfaitement adapté ; nous vous recommandons donc de procéder ainsi :

1. **Définir** le cahier des charges (les entrées, les sorties et le comportement attendu du logiciel)
2. **Concevoir (sur papier)** les types et les étapes de traitement dont vous aurez besoin (structures, algorithmes, fichiers, tests)
3. **Réaliser** le programme de la manière suivante :
   - 3.1 Écrire les données (headers).
   - 3.2 Écrire les prototypes (headers).
   - 3.3 Partager l'écriture des fichiers **c** en fonctionnalités isolées.
   - 3.4 tester chaque fichier **c** séparément et corriger/continuer.
4. **Intégrer** toutes les parties pour réaliser le programme final.
5. **Valider** le programme par DES tests montrant que le logiciel répond au cahier des charges.

Partage du travail

Normalement, chaque étape du cycle est effectuée par un ou plusieurs spécialistes.
Dans votre cas nous vous donnons les conseils suivants :

- Faites la définition et la conception ensemble avant d'attaquer le code. Chacun propose des approches dans les parties dans lesquelles il(elle) se sent le plus à l'aise.
- Avant d'attaquer l'écriture du code, commencez par organiser votre projet (quel répertoire, combien de fichiers, leur noms etc.)
- Écrivez les headers en premier et en même temps mais gardez en tête que vous serez sûrement amenés à les modifier.
- Partagez vous les fonctions et écrivez les fichiers de tests en premier :
  - La personne qui écrit le code d'une fonction ne doit pas être celle qui écrit le test
  - Écrivez un test pour casser le code de votre binôme !
- Faites une intégration itérative. C'est-à-dire, réalisez la fonction main progressivement et vérifiez son comportement à chaque étape.

Indentation d'un code

Indenter le code, c'est introduire à bon escient des espaces ou des tabulations de façon à **aérer** et mettre en valeur la structure **logique** du code (bloc de boucle, corps de fonction). C'est indispensable pour permettre la **relecture** et **trouver les erreurs** plus vite.

NON ! ! ! Très dur à déboguer.

```
int j=N;i=0;char ligne[256];
while(i<N)
fgets(ligne,256,stdin);i++;
if (strlen(ligne) ==256) if (ligne[strlen(ligne)-2]!='\n')
printf("ligne trop longue"); else ;
else printf("ligne OK");
printf("reste %d lignes a lire\n",j);
```

Indentation d'un code : exemple

OUI ! ! ! On repère tout de suite l'organisation du code, c'est lisible.

```
int j=N;
i=0;
char ligne[256];
while(i<N)
    fgets(ligne,256,stdin);
i++;
if (strlen(ligne) ==256)
    if (ligne[strlen(ligne)-2]!='\n')
        printf("ligne trop longue");
    else ;
else
    printf("ligne OK");
printf("reste %d lignes a lire\n",j);
```

Formatage du code : règles [Torvalds, 1995]

Comme pour toutes les autres conventions de codage, il existe plusieurs principes pour l'indentation. Insistons sur les choses essentielles :

- Une seule unique instruction par ligne (un seul « ; »). Passer à la ligne pour l'instruction suivante.
- Le contenu d'un nouveau bloc doit avoir un niveau d'indentation en plus (décalé vers la droite)
- L'accolade fermant le bloc est alignée immédiatement en dessous du début de l'instruction qui a déclenché l'ouverture du bloc

Exemple

```
1 do {
2         body of do-loop
3 } while (condition);
```

```
1 if (x == y) {
2         ...
3 } else if (x > y) {
4         ...
5 } else {
6         ...
7 }
```

Indentation automatique

Il existe plusieurs moyens d'indenter votre code plus ou moins automatiquement :

- Dans un terminal `indent -nbc -npsl -npcs -di1 mon_fichier.c`
- Dans un terminal **astyle mon_fichier.c**
- Dans Gedit : Édition > Préférences > Éditeur > Activer l'indentation automatique
- Dans Kate : Dans le fichier .kateconfig, il faut ajouter la ligne suivante
  `indent-mode "cstyle"`
- Dans vim : taper `1G=G`

Conventions de nommage

Comment choisir les noms des identifieurs : types, constantes, fonctions, paramètres des fonctions, variables ?

C programmers do not use cute names like `ThisVariableIsATemporaryCounter` *but* `tmp` [Torvalds, 1995].

Néanmoins il faut choisir des noms signifiants, qui indiquent ce qu'est la chose nommée. Il vaut mieux des noms longs et clairs que courts mais abscons.

autres conseils

- Les noms de constantes sont toujours en MAJUSCULE séparés par des "_" `#define TAILLE_SIGNAL 10` et non `#define taille_signal 10`.
- Variables et fonctions : Minuscule pour le premier mot, majuscule ensuite `energieSignal`.

Conventions de nommage : exemple

Que fait la fonction ci dessous ?

```
1 void f1 (double* t, int x){
2 int i; double z=0;
3 for(i=0;i<x;i++)z+=t[i];
4 for(i=0;i<x;i++)t-=z/x;
5 }
```

On comprend rapidement le code :

```
void centre_signal (double* signal, int taille){
int i; double moyenne=0;
for(i=0;i<x;i++)moyenne+=signal[i];
moyenne/=taille;
for(i=0;i<x;i++)signal-=moyenne;
}
```

Commenter son code

Commenter le code est essentiel à sa relecture et compréhension. Mais c'est tout un art dont voici quelques règles :

- Commenter le code pendant l'écriture du code (après, c'est trop tard).
- Les commentaires se trouvent généralement avant chaque fonction précisant le contrat de la fonction et non dans le corps de la fonction.
- Commenter ce que fait le code, pas comment il le fait (le code doit être suffisamment clair pour que le « comment » se lise tout seul).
- On commente le corps de la fonction uniquement pour expliquer un code complexe.

Commenter son code

Commentaires inutiles :

```
1  void centre_signal (double* signal, int taille){
2  /* on declare un indice et une moyenne*/
3  int i; double moyenne=0;
4  /* pour chaque echantillon on somme la valeur de l'echantillon*/
5  for(i=0;i<x;i++)moyenne+=signal[i];
6  moyenne/=taille; /* la moyenne c'est la somme sur le total*/
7  /* pour chaque echantillon on soustrait la moyenne precedement calculee*/
8  for(i=0;i<x;i++)signal-=moyenne;
9  /* et voila c'est fini*/
10 }
```

Commentaires utiles (surtout si on appelle sa fonction f1...) :

```
/* fonction qui centre un signal passe en parametre
 * (soustraction de la moyenne) */
void centre_signal (double* signal, int taille){
int i; double moyenne=0;
for(i=0;i<x;i++)
        moyenne+=signal[i];
moyenne/=taille;
for(i=0;i<x;i++)
        signal-=moyenne;
}
```

Trucs et astuces

Main avec arguments en ligne de commande

Comment font les exécutables `ls`, `cd`, `rm`, `etc.` pour prendre des arguments en ligne de commande ?

Main avec arguments en ligne de commande

Comment font les exécutables `ls, cd, rm, etc.` pour prendre des arguments en ligne de commande ?

Le prototype complet de la fonction main() d'un programme C est : `int main(int argc, char *argv[])` où :
int argc est le nombre de paramètres effectivement passé au programme. Ce nombre est toujours ≥ 1 car le premier paramètre est toujours le nom de l'exécutable.
char * argv [] est un tableau de chaîne de caractères contenant les paramètres effectivement passés au programme.
Il convient de convertir les nombres de la chaîne de caractères vers la représentation binaire. P.ex., de "1645" vers la valeur 1645 dans un `int`.

argv[0] contient toujours le nom du programme (ou plus précisément le chemin utilisé pour accéder au fichier exécutable)

Main : Exemple

Le prototype complet de la fonction main() d'un programme C est :

```
1 int main ( int argc , char * argv[] ) {
2    int i;
3    printf("Nombre d'arguments passes au programme : %d\n",argc);
4    for(i = 0 ; i< argc ; i ++) {
5       printf(" argv[%d] : %s\n", i, argv[i]);
6    }
7    return 0 ;
8 }
```

```
$ ./prg.exe test.txt 2
```

alors le programme affichera dans le Terminal :

```
Nombre d'arguments passes au programme : 3
   argv[0] : ./prg.exe
   argv[1] : test.txt
   argv[2] : 2
```

Les Macros

```
1 #define N 256
2 #define PRINT(x) printf("x vaut %d\n",x)
3 main() { int t[N];
4   for (i=0; i<=N-1; i++) t[i]=N/2;
5   PRINT(i);
6 }
```

#define permet des codes complexes

```
1 #define SWAP(x,y,type) { type inter=x; x=y; y=inter; }
2 main() { int a,b; double c,d;
3   SWAP(a,b,int); SWAP(c,d,double);
4 }
```

devant un identificateur empêche sa substitution. Le paramètre est remplacé par la chaîne comportant son nom

```
1 #define PR(x) printf(#x" = %d\n",x)
2
3 PR(var) -> printf("var"" = %d\n",var) => printf("var = %d\n",var)
```

Les Macros, danger !

Attention `#define abs(x) (x>0 ? x : -x)` EST FAUX ! ! !
(essayez abs(a+b) avec a+b $\leq$ 0)

Et peut être dangereux
(essayer abs(a++))

Les Macros, danger !

Attention `#define abs(x) (x>0 ? x : -x)` EST FAUX ! ! !
(essayez abs(a+b) avec a+b $\leq$ 0)

Et peut être dangereux
(essayer abs(a++))

Évitez les macros avec des résultats de fonctions ou d'opération, ne l'utiliser qu'avec des variables

Références I

Kernighan, B. et Ritchie, D. (2004).
Le langage C : Norme ANSI.
Sciences sup. Dunod.

The GCC team (2011).
Gcc online documentation.
http://gcc.gnu.org/onlinedocs/cpp/.

The GNU team (2010).
Gnu make manual.
http://www.gnu.org/software/make/manual/.

Torvalds, L. (1995).
Linux kernel coding style.
http://www.kernel.org/doc/Documentation/CodingStyle.