Programmation C et Système

Modules et Compilation séparée

Régis Barbanchon

L2 Informatique

Fil rouge : un prog utilisant flottants, points 2D et 3D

Ce programme crée 2 flottants `a`, `b` et affiche s'il sont égaux.
Il fait de même pour deux points 2D `p`, `q`, puis deux points 3D `s`, `t`.
Les égalités sont testées avec des tolérances que l'on peut changer.

```
// File: prog.c
...                                 // some code hidden here and shown later
int main (void) {
  Double_setTolerance (0.1);
  double a= 3.0, b= 3.1;
  printf ("a is near b ? %s\n", Double_isNear (a, b) ? "yes" : "no");

  Coord2D p= Coord2D_make (5.0, 6.0), q= Coord2D_make (5.1, 5.8);
  Coord2D_setTolerance (0.1);
  printf ("p is near q ? %s\n", Coord2D_isNear (p, q) ? "yes" : "no");

  Coord3D s= Coord3D_make (5.0, 6.0, 7.0), t= Coord3D_make (5.1, 5.9, 7.1);
  Coord3D_setTolerance (0.2);
  printf ("s is near t ? %s\n", Coord3D_isNear (s, t) ? "yes" : "no");
  return 0;
}
```

Le code caché utilise `fabs()` du header `<math.h>`
donc on utilise `-lm` pour lier le code avec la librairie `libm` :

```
$ clang   -std=c99 -W -Wall -pedantic   -lm    prog.c -o prog
$ ./prog
a is near b ? no
p is near q ? no
s is near t ? yes
```

La partie cachée gérant les flottants

Dans la partie gérant les flottants, on a...

- une macro pour la valeur de la tolérance par défaut, et une variable globale pour la tolérance implicite :

```
#define DEFAULT_TOLERANCE 0.0001
double Double_tolerance= DEFAULT_TOLERANCE;
```

- un setter sur cette variable :

```
void Double_setTolerance (double tolerance) {
  Double_tolerance= fabs (tolerance);
}
```

- deux prédicats d'égalité, l'un utilisant une tolérance explicite, l'autre utilisant la tolérance implicite :

```
bool Double_isNearBy (double self, double other, double tolerance) {
  return fabs (self - other) <= tolerance;
}

bool Double_isNear (double self, double other) {
  return Double_isNearBy (self, other, Double_tolerance);
}
```

La partie cachée gérant les points en 2D

Dans la partie gérant les points 2D, on a...

- une définition de type structuré pour les points 2D :

```
typedef struct Coord2D { double x, y; } Coord2D;
```

- une fonction de fabrication de point 2D :

```
Coord2D Coord2D_make (double x, double y) {
  return (Coord2D) { .x= x, .y= y };
}
```

- un code sur les points 2D analogue à celui des flottants :

```
double Coord2D_tolerance= DEFAULT_TOLERANCE;

void Coord2D_setTolerance (double tolerance) {
  Coord2D_tolerance= fabs (tolerance);
}

bool Coord2D_isNearBy (Coord2D self, Coord2D other, double tolerance) {
  return Double_isNearBy (self.x, other.x, tolerance)
    &&   Double_isNearBy (self.y, other.y, tolerance);
}

bool Coord2D_isNear (Coord2D self, Coord2D other) {
  return Coord2D_isNearBy (self, other, Coord2D_tolerance);
}
```

La partie cachée gérant les points en 3D

Dans la partie gérant les points 3D, on a...

- une définition de type structuré pour les points 3D :

```
typedef struct Coord3D { double x, y, z; } Coord3D;
```

- une fonction de fabrication de point 3D :

```
Coord3D Coord3D_make (double x, double y, double z) {
  return (Coord3D) { .x= x, .y= y, .z = z };
}
```

- un code sur les points 3D analogue à celui des flottants :

```
double Coord3D_tolerance= DEFAULT_TOLERANCE;

void Coord3D_setTolerance (double tolerance) {
  Coord3D_tolerance= fabs (tolerance);
}

bool Coord3D_isNearBy (Coord3D self, Coord3D other, double tolerance) {
  return Double_isNearBy (self.x, other.x, tolerance)
    &&   Double_isNearBy (self.y, other.y, tolerance)
    &&   Double_isNearBy (self.z, other.z, tolerance);
}

bool Coord3D_isNear (Coord3D self, Coord3D other) {
  return Coord3D_isNearBy (self, other, Coord3D_tolerance);
}
```

Vue générale et résumée de la totalité code

```c
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
```

```c
#define DEFAULT_TOLERANCE 0.0001
double Double_tolerance= DEFAULT_TOLERANCE;

void Double_setTolerance (double tolerance) {...}
bool Double_isNearBy (double self, double other, double tolerance) {...}
bool Double_isNear (double self, double other) {...}
```

```c
typedef struct Coord2D { double x, y; } Coord2D;
double Coord2D_tolerance= DEFAULT_TOLERANCE;

Coord2D Coord2D_make (double x, double y) {...}
void Coord2D_setTolerance (double tolerance) {...}
bool Coord2D_isNearBy (Coord2D self, Coord2D other, double tolerance) {...}
bool Coord2D_isNear (Coord2D self, Coord2D other) {...}
```

```c
typedef struct Coord3D { double x, y, z; } Coord3D;
double Coord3D_tolerance= DEFAULT_TOLERANCE;

Coord3D Coord3D_make (double x, double y, double z) {...}
void Coord3D_setTolerance (double tolerance) {...}
bool Coord3D_isNearBy (Coord3D self, Coord3D other, double tolerance) {...}
bool Coord3D_isNear (Coord3D self, Coord3D other) {...}
```

```c
int main (void) {...}
```

Organisation en modules

On souhaite découper le code en 4 modules :

- un module `Double` pour la gestion des flottants.
- un module `Coord2D` pour la gestion des points 2D.
- un module `Coord3D` pour la gestion des points 3D.
- un module `Main` pour le programme principal.

Certains modules dépendent d'autres modules :

- `Coord2D` dépend de `Double` car utilise `Double_isNearBy()`.
- `Coord3D` dépend de `Double` car utilise `Double_isNearBy()`.
- `Main` dépend de `Double`, `Coord2D`, et `Coord3D`.

Pour chaque module `ModuleName`, on va avoir :

- un fichier d'entête `ModuleName.h` (sauf pour `Main`),
- un fichier d'implémentation `ModuleName.c`.

Contenu général des fichiers d'entête et d'implémentation

Le fichier d'entête (ou *header*) d'un module contient :

- la définition de ses macros (si elle sont publiques),
- la définition de ses types (s'ils sont publics),
- la déclaration de ses variables globales (si elle sont publiques),
- la déclaration de ses fonctions publiques (leurs prototypes).

Le fichier d'implémentation d'un module contient :

- la définition de ses variables globales,
- la définition de ses fonctions.

Définir une variable, une fonction : c'est lui donner une existence (allouer la mémoire de la variable, associer du code à la fonction).

Déclarer une variable, une fonction : c'est dire qu'un objet de ce nom et de ce type existe (est défini) ailleurs, mais sans le définir ici.

Le module `Double` : son fichier d'entête `Double.h`

```
// File: Double.h
#ifndef DOUBLE_H // guard against multiple inclusion
#define DOUBLE_H // marks the file as being now included
```

```
#include <stdbool.h>

#define DEFAULT_TOLERANCE 0.0001
extern double Double_tolerance;

void Double_setTolerance (double tolerance);
bool Double_isNearBy (double self, double other, double tolerance);
bool Double_isNear (double self, double other);
```

```
#endif // end of guard against multiple inclusion
```

On trouve dans le header `Double.h` :

- une garde contre l'inclusion multiple (encadrés 1 et 3),
- l'inclusion de `<stdbool.h>` car le header utilise `bool`,
- la déf. de la macro `DEFAULT_TOLERANCE`,
- la décl. de la var. globale `Double_tolerance` avec `extern`,
- la décl. des fonctions publiques `Double_xxx()`.

Le module Double : son fichier d'impl. Double.c

```c
// File: Double.c
#include "Double.h"
#include <math.h>

double Double_tolerance= DEFAULT_TOLERANCE;

void Double_setTolerance (double tolerance) {
  Double_tolerance= fabs (tolerance);
}

bool Double_isNearBy (double self, double other, double tolerance) {
  return fabs (self - other) <= tolerance;
}

bool Double_isNear (double self, double other) {
  return Double_isNearBy (self, other, Double_tolerance);
}
```

On trouve dans le fichier d'implémentation Double.c :

- l'inclusion de "Double.h" car un .c inclut toujours son .h,
- l'inclusion de <math.h> car le code utilise fabs(),
- la déf. de la var. globale Double_tolerance et son init.,
- la déf. des fonctions Double_xxx().

Le module Coord2D : son fichier d'entête Coord2D.h

```
// File: Coord2D.h
#ifndef COORD2D_H // guard against multiple inclusion
#define COORD2D_H // marks the file as being now included
```

```
#include <stdbool.h>

typedef struct Coord2D { double x, y; } Coord2D;
extern double Coord2D_tolerance;

Coord2D Coord2D_make (double x, double y);
void Coord2D_setTolerance (double tolerance);
bool Coord2D_isNearBy (Coord2D self, Coord2D other, double tolerance);
bool Coord2D_isNear   (Coord2D self, Coord2D other);
```

```
#endif // end of guard against multiple inclusion
```

On trouve dans le header `Coord2D.h` :

- une garde contre l'inclusion multiple (encadrés 1 et 3),
- l'inclusion de `<stdbool.h>` car le header utilise `bool`,
- la déf. du type structuré `Coord2D`,
- la décl. de la var. globale `Coord2D_tolerance` avec `extern`,
- la décl. des fonctions publiques `Coord2D_xxx()`.

Le module Coord2D : son fichier d'impl. Coord2D.c

```c
// File: Coord2D.c
#include "Coord2D.h"
#include "Double.h"
#include <math.h>

double Coord2D_tolerance= DEFAULT_TOLERANCE;

void Coord2D_setTolerance (double toler) { Coord2D_tolerance= fabs (toler); }

Coord2D Coord2D_make (double x, double y) { return (Coord2D){ .x= x, .y= y }; }

bool Coord2D_isNearBy (Coord2D self, Coord2D other, double tolerance) {
  return Double_isNearBy (self.x, other.x, tolerance)
    &&   Double_isNearBy (self.y, other.y, tolerance);
}

bool Coord2D_isNear (Coord2D self, Coord2D other) {
  return Coord2D_isNearBy (self, other, Coord2D_tolerance);
}
```

On trouve dans le fichier d'implémentation Coord2D.c :

- l'inclusion de "Coord2D.h" car un .c inclut toujours son .h,
- l'inclusion de "Double.h" pour Double_isNearBy(), etc,
- l'inclusion de <math.h> pour fabs(),
- la déf. de la var. globale Coord2D_tolerance et son init.,
- la déf. des fonctions Coord2D_xxx().

Le module Coord3D : son fichier d'entête Coord3D.h

```
// File: Coord3D.h
#ifndef COORD3D_H // guard against multiple inclusion
#define COORD3D_H // marks the file as being now included
```

```
#include <stdbool.h>

typedef struct Coord3D { double x, y, z; } Coord3D;
extern double Coord3D_tolerance;

Coord2D Coord3D_make (double x, double y);
void Coord3D_setTolerance (double tolerance);
bool Coord3D_isNearBy (Coord3D self, Coord3D other, double tolerance);
bool Coord3D_isNear   (Coord3D self, Coord3D other);
```

```
#endif // end of guard against multiple inclusion
```

Le header Coord3D.h est analogue à Coord2D.h.

Le module Coord3D : son fichier d'impl. Coord3D.c

```c
// File: Coord3D.c
#include "Coord3D.h"
#include "Double.h"
#include <math.h>

double Coord3D_tolerance= DEFAULT_TOLERANCE;

void Coord3D_setTolerance (double tolerance) {
  Coord2D_tolerance= fabs (tolerance);
}

Coord2D Coord2D_make (double x, double y, double z) {
  return (Coord3D){ .x= x, .y= y, .z= z };
}

bool Coord3D_isNearBy (Coord3D self, Coord3D other, double tolerance) {
  return Double_isNearBy (self.x, other.x, tolerance)
    &&   Double_isNearBy (self.y, other.y, tolerance)
    &&   Double_isNearBy (self.z, other.z, tolerance);
}

bool Coord3D_isNear (Coord3D self, Coord3D other) {
  return Coord3D_isNearBy (self, other, Coord3D_tolerance);
}
```

Le fichier Coord3D.c est analogue à Coord2D.c.

Le module Main : son unique fichier Main.c

```c
// File: Main.c
#include <stdio.h>
#include "Double.h"
#include "Coord2D.h"
#include "Coord3D.h"

int main (void) {
  Double_setDefaultTolerance (0.1);
  double a= 3.0, b= 3.1;
  printf ("a is near b ? %s\n", Double_isNear (a, b) ? "yes" : "no");

  Coord2D p= Coord2D_make (5.0, 6.0);
  Coord2D q= Coord2D_make (5.1, 5.8);
  Coord2D_setDefaultTolerance (0.1);
  printf ("p is near q ? %s\n", Coord2D_isNear (p, q) ? "yes" : "no");

  Coord3D s= Coord3D_make (5.0, 6.0, 7.0);
  Coord3D t= Coord3D_make (5.1, 5.9, 7.1);
  Coord3D_setDefaultTolerance (0.2);
  printf ("s is near t ? %s\n", Coord3D_isNear (s, t) ? "yes" : "no");
  return 0;
}
```

Le fichier Main.c inclut :

- <stdio.h> pour printf(),
- "Double.h" pour Double_xxx(),
- "Coord2D.h" pour Coord2D et Coord2D_xxx(),
- "Coord3D.h" pour Coord3D et Coord3D_xxx().

Nécessité des gardes contre les inclusions multiples

Le contenu d'un header `ModuleName.h` est toujours à l'intérieur d'une garde contre l'inclusion multiple, de la forme :

```
#ifndef MODULE_NAME_H
#define MODULE_NAME_H
...
#endif
```

Le risque d'inclusion multiple arrive lorsqu'un fichier est atteint indirectement par plusieurs chaînes de directives `#include`.
Par exemple `<stdbool.h>` est atteint trois fois par `Main.c` :

- via la chaîne `Main.c` → `"Double.h"` → `<stdbool.h>`,
- via la chaîne `Main.c` → `"Coord2D.h"` → `<stdbool.h>`,
- via la chaîne `Main.c` → `"Coord3D.h"` → `<stdbool.h>`.

Or, on ne doit jamais atteindre deux fois une même définition.

Si l'on devait écrire le header `<stdbool.h>`, on l'écrirait tel que montré ci-contre, afin que les macros `bool`, `true`, `false` ne soient jamais redéfinies plus d'une fois.

```
// stdbool.h
#ifndef STDBOOL_H
#define STDBOOL_H
#define bool _Bool
#define true  1
#define false 0
#endif
```

Compilation séparée

Pourvu que le répertoire ne contienne pas d'autres sources...

```
$ ls *.c *.h
Coord2D.c  Coord2D.h  Coord3D.c  Coord3D.h  Double.c  Double.h  Main.c
```

on peut produire l'exécutable `prog` en une ligne de commande :

```
$ clang -W -Wall -std=c99  -lm    *.c -o prog
```

La compilation séparée consiste à compiler d'abord chaque module `ModuleName` séparément dans un fichier binaire `ModuleName.o`, via l'option `-c` (*compile source, but do not link into executable*) :

```
$ clang -W -Wall -std=c99   -c *.c
$ ls *.o
Coord2D.o  Coord3D.o  Double.o  Main.o
```

... puis à lier tous ces binaires en un exécutable (édition de liens) :

```
$ clang   -lm   *.o -o prog
```

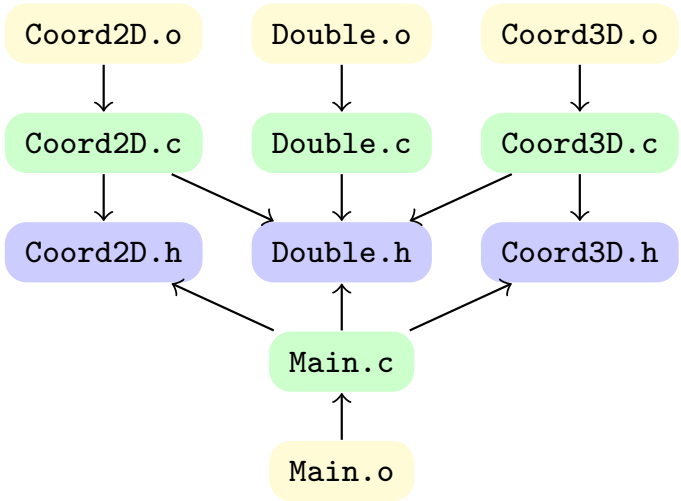
C'est seulement lors de l'édition de liens qu'on lie avec `libm`, via l'option `-lm` (car une libraire `libtruc.so` se lie via `-ltruc`).

Graphe de dépendances et clôture transitive

Chaque fichier binaire `.o` dépend de son fichier source `.c`, et chaque source ou header dépend des headers `.h` qu'il inclut.

La relation de dépendance entre fichiers induit un graphe des dépendances, qui doit toujours être acyclique.

Pour `prog`, on a le graphe de dépendance dessiné ci-contre.



`clang -MM *.c` affiche la clôture transitive des dépendances de chaque binaire `.o` que produirait la compilation d'un source `.c` :

```
$ clang -MM *.c
Coord2D.o: Coord2D.c Coord2D.h Double.h
Coord3D.o: Coord3D.c Coord3D.h Double.h
Double.o: Double.c Double.h
Main.o: Main.c Double.h Coord2D.h Coord3D.h
```

Table des symboles des modules (1/3)

Remarque : `clang -MM *.c` ne traverse que les `#include "*"`.
là où `clang -M *.c` traverserait aussi les `#include <*>`.

```
$ clang -MM *.c
Coord2D.o: Coord2D.c Coord2D.h Double.h
Coord3D.o: Coord3D.c Coord3D.h Double.h
Double.o: Double.c Double.h
Main.o: Main.c Double.h Coord2D.h Coord3D.h
```

On voit qu'un `.o` ne dépend d'aucun autre `.c` que le sien.
Un module ne connaît donc pas le code des modules qu'il utilise.
Par exemple, le module `Main` ne connaît que le code de `main()`
et ne connaît pas le code des autres fonctions qu'elle utilise.
On le voit via `nm` qui affiche la table des symboles d'un binaire :

```
$ nm Main.o
                 U Coord2D_isNear
                 U Coord2D_make
                 U Coord2D_setTolerance
                 U Coord3D_isNear
                 U Coord3D_make
                 U Coord3D_setTolerance
                 U Double_isNear
                 U Double_setTolerance
0000000000000000 T main
                 U printf
```

Dans `Main.o`, seul `main` a une adresse, et est marqué `T` (*with Text*).

Les autres symboles sont marqués `U` (*Undefined*).

Table des symboles des modules (2/3)

```
$ nm Double.o
0000000000000070 T Double_isNear
0000000000000030 T Double_isNearBy
0000000000000000 T Double_setTolerance
0000000000000000 D Double_tolerance
                 U fabs
```

```
$ nm Coord2D.o
00000000000000e0 T Coord2D_isNear
0000000000000070 T Coord2D_isNearBy
0000000000000000 T Coord2D_make
0000000000000040 T Coord2D_setTolerance
0000000000000000 D Coord2D_tolerance
                 U Double_isNearBy
                 U fabs
```

```
$ nm Coord3D.o
0000000000000120 T Coord3D_isNear
0000000000000070 T Coord3D_isNearBy
0000000000000000 T Coord3D_make
0000000000000040 T Coord3D_setTolerance
0000000000000000 D Coord3D_tolerance
                 U Double_isNearBy
                 U fabs
```

Les trois autres tables montrent que chaque module n'a que le texte de ses propres fonctions.

On voit que les symboles de variables apparaissent dans les tables, marquées d'un `D` (*initialized Data*).

Table des symboles des modules (3/3)

```
$ clang *.o   -lm  -o prog
$ nm prog
00000000004006c0 T Coord2D_isNear
0000000000400650 T Coord2D_isNearBy
00000000004005e0 T Coord2D_make
0000000000400620 T Coord2D_setTolerance
0000000000602038 D Coord2D_tolerance
0000000000400830 T Coord3D_isNear
0000000000400780 T Coord3D_isNearBy
0000000000400710 T Coord3D_make
0000000000400750 T Coord3D_setTolerance
0000000000602040 D Coord3D_tolerance
0000000000400900 T Double_isNear
00000000004008c0 T Double_isNearBy
0000000000400890 T Double_setTolerance
0000000000602048 D Double_tolerance
                 U fabs@@GLIBC_2.2.5
0000000000400940 T main
                 U printf@@GLIBC_2.2.5
```

Après édition de liens des modules et de `libm` pour générer l'exécutable `prog`, la table de `prog` montre que toutes les fonctions utilisateur ont leur texte (`T`). Seules les 2 fonctions de librairie `fabs` et `printf` sont indéfinies (`U`).

Leurs codes restent en librairie pour économiser de la place. Avec `-static`, on peut forcer leur code à intégrer l'exécutable `prog` pour le rendre autonome (mais alors sa taille augmente).

```
$ clang *.o   -static -lm  -o prog-statically-linked
$ ls -s -h prog*
16K prog   896K prog-statically-linked
```

Petit aparté sur `fabs` et la table des symboles

En fait, on a un peu triché pour les besoins du cours :

`fabs` n'apparaît pas vraiment dans les tables de symboles,
du moins tel que le code est actuellement écrit,
car `clang` optimise le code pour éliminer les appels à `fabs`.

Pour forcer `fabs` à apparaître dans les tables de symboles,
on a écrit son adresse dans un pointeur sur fonction.

Par exemple, dans `Double.c`, on a écrit l'adresse de `fabs`
dans le pointeur `f`, et on a invoqué la fonction via `f` :

```
void Double_setTolerance (double tolerance) {
  double (* f)(double)= fabs;      // pointer f now points to function fabs
  Double_tolerance= f (tolerance); // calls function fabs() via pointer f()
}
```

Ainsi, `fabs` apparaît dans la table des symboles de `Double.o`.

Variables privées à un module (1/3)

Aucun des modules n'utilise directement la variable globale de tolérance définie par les autres modules. On peut donc :

- supprimer leur déclaration des headers `.h`, et
- les définir privées dans les sources `.c` avec le mot-clé `static`.

Elles sont ainsi inaccessibles depuis les autres modules.
De plus, leurs noms deviennent locaux aux modules, c'est-à-dire, des vars privées de modules différents peuvent avoir le même nom.

Si `{Double,Coord2D,Coord3}_tolerance` deviennent `static`, elles peuvent être renommées `currentTolerance` toutes les trois, sans conflit de nom lors de la fusion des modules à l'édition de lien.

Variables privées à un module (2/3)

Par exemple, dans le module `Double`, on effectue les modifs :

```
// File: Double.h
#ifndef DOUBLE_H
#define DOUBLE_H
#include <stdbool.h>
#define DEFAULT_TOLERANCE 0.0001
extern double currentTolerance; // line removed
void Double_setTolerance (double tolerance);
bool Double_isNearBy (double self, double other, double tolerance);
bool Double_isNear (double self, double other);
#endif
```

```
// File: Double.c
#include "Double.h"
#include <math.h>

static double currentTolerance= DEFAULT_TOLERANCE;

void Double_setTolerance (double tolerance) {
  currentTolerance= fabs (tolerance);
}
bool Double_isNearBy (double self, double other, double tolerance) {
  return fabs (self - other) <= tolerance;
}
bool Double_isNear (double self, double other) {
  return Double_isNearBy (self, other, currentTolerance);
}
```

Et idem dans les modules `Coord2D` et `Coord3D`.

Variables privées à un module (3/3)

Dans les tables, les variables privées sont marquées `d` (et non `D`) :

```
$ clang -std=c99 -W -Wall -pedantic *.c -c
$ nm Double.o
0000000000000000 d currentTolerance
0000000000000080 T Double_isNear
0000000000000040 T Double_isNearBy
0000000000000000 T Double_setTolerance
                 U fabs
```

Dans `prog`, il a bien 3 variables distinctes `currentTolerance` :

```
$ clang *.o   -lm   -o prog
$ nm prog
0000000000400740 T Coord2D_isNear
00000000004006d0 T Coord2D_isNearBy
0000000000400650 T Coord2D_make
0000000000400690 T Coord2D_setTolerance
00000000004008c0 T Coord3D_isNear
0000000000400810 T Coord3D_isNearBy
0000000000400790 T Coord3D_make
00000000004007d0 T Coord3D_setTolerance
0000000000602040 d currentTolerance
0000000000602048 d currentTolerance
0000000000602050 d currentTolerance
00000000004009a0 T Double_isNear
0000000000400960 T Double_isNearBy
0000000000400920 T Double_setTolerance
                 U fabs@@GLIBC_2.2.5
00000000004009e0 T main
                 U printf@@GLIBC_2.2.5
```

Fonctions privées à un module (1/2)

On peut aussi avoir des fonctions privées avec le mot-clé `static`.

On peut ainsi avoir un setter privé `setTolerance()`
qui ne vérifie pas le signe de la tolérance passée en argument.
Par exemple, dans le module `Double`, on effectue les modifs :

```
#include "Double.h"
#include <math.h>

static double currentTolerance= DEFAULT_TOLERANCE;

static void setTolerance (double tolerance) { // private unsafe setter
  currentTolerance= tolerance;
}

void Double_setTolerance (double tolerance) { // public safe setter
  setTolerance (fabs (tolerance));
}

bool Double_isNearBy (double self, double other, double tolerance) {
  return fabs (self - other) < tolerance;
}

bool Double_isNear (double self, double other) {
  return Double_isNearBy (self, other, currentTolerance);
}
```

Et de même dans les modules `Coord2D` et `Coord3D`.

Fonctions privées à un module (2/2)

Dans les tables, les fonctions privées sont marquées `t` (et non `T`) :

```
$ clang -std=c99 -W -Wall -pedantic *.c -c
$ nm Double.o:
0000000000000000 d currentTolerance
00000000000000a0 T Double_isNear
0000000000000060 T Double_isNearBy
0000000000000000 T Double_setTolerance
                 U fabs
0000000000000040 t setTolerance
```

Dans `prog`, il a bien 3 fonctions distinctes `setTolerance` :

```
$ clang *.o   -lm   -o prog
$ nm prog
```

```
0000000000400760 T Coord2D_isNear
00000000004006f0 T Coord2D_isNearBy
0000000000400650 T Coord2D_make
0000000000400690 T Coord2D_setTolerance
0000000000400900 T Coord3D_isNear
0000000000400850 T Coord3D_isNearBy
00000000004007b0 T Coord3D_make
00000000004007f0 T Coord3D_setTolerance
0000000000602040 d currentTolerance
0000000000602048 d currentTolerance
0000000000602050 d currentTolerance
```

```
0000000000400a00 T Double_isNear
00000000004009c0 T Double_isNearBy
0000000000400960 T Double_setTolerance
                 U fabs@@GLIBC_2.2.5
0000000000400a40 T main
                 U printf@@GLIBC_2.2.5
00000000004006d0 t setTolerance
0000000000400830 t setTolerance
00000000004009a0 t setTolerance
```