TP 2 : Pointeurs et tableaux

1 Pointeurs

Comme vous avez pu le voir, en programmation les variables stockent des données *e.g.* `int a = 10;` signifie que la variable *a* stocke la valeur 10 sous la forme d'un entier (*int*). Ces variables sont stockées dans un emplacement mémoire spécifique qui est leur adresse. C'est à cette emplacement en mémoire que se trouve la valeur. Afin d'accéder (*i.e.* lire) la valeur stockée, il est possible d'utiliser deux types d'adressage :

- **adressage symbolique** : Le nom de la variable nous permet d'accéder directement à sa valeur.
- **adressage par adresse** : on utilise l'adresse de la variable pour accéder à sa valeur. L'adressage par adresse correspond à la notion de pointeur.

QUESTION 1 ► Écrivez un programme déclarant une variable double *d* contenant la valeur 13.12. Afficher sa valeur ainsi que son adresse avec printf (`printf(''%p'',variable)`).

En C, les pointeurs (et donc l'adressage par adresse) sont à la base de beaucoup de programmes (ainsi qu'une source de beaucoup d'erreurs). Une variable de type pointeur est simplement une variable classique qui va contenir une adresse qui est l'adresse pointé qui va stocker effectivement une valeur. Pour déclarer qu'une variable est un pointeur, il suffit d'ajouter $*$ après le type *e.g.* `int *pa` déclare que la variable *pa* est un pointeur vers une variable de type *int*.
Maintenant que nous avons une variable qui peut stocker l'adresse en mémoire d'une variable, il faut pouvoir récupérer l'adresse d'une variable préalablement déclarée. Pour cela, l'opérateur 'adresse de' est défini en C : &. Pour récupérer l'adresse d'une variable (*e.g. a*), il suffit d'ajouter le & devant le nom de la variable (*e.g.* &*a*). Le & permet de récupérer une adresse mais il nous faut un opérateur symétrique qui permet de récupérer la valeur pointée par un pointeur. Pour cela, l'opérateur 'contenu de' est défini en C par $*$. Pour récupérer la valeur pointée par la variable *pa*, il suffit d'ajouter l'opérateur $*$ *i.e.* $*pa$.

QUESTION 2 ► Reprenez le code de la question précédente et déclarez une variable pointeur *ptr_d*. Affectez l'adresse mémoire de la variable *d* à *ptr_d*. Affichez avec `printf()` la valeur de *d* et de *ptr_d* ainsi que l'adresse de *d* et *ptr_d*.

QUESTION 3 ► Reprenez le code de la question précédente et changez la valeur de *d* ; pas directement mais indirectement en passant par son pointeur (*i.e. ptr_d*).

QUESTION 4 ► Que se passe-t-il si on n'affecte pas d'adresse à un pointeur et qu'on essaye d'afficher son contenu ? Écrivez un programme et essayer. Que pouvons nous observer ?

Afin de déclarer qu'un pointeur ne pointe pas (encore) sur une valeur, la convention est de lui affecter intialement la valeur *NULL* (en général, la valeur 0).
Nous venons de voir que pour déclarer un pointeur, nous utilisons son type suivi de l'opérateur $*$. Pour autant, une adresse mémoire ne change pas suivant le type utilisé. Suivant le processeur et le système d'exploitation, les adresses sont toutes codées sur 32bit ou 64bit. Il est donc possible d'utiliser le type *void*$*$ pour déclarer un pointeur. Ce type est dit *pointeur générique*. L'arithmétique sur les pointeurs génériques est délicate. Par défaut l'arithmétique sera la même que sur le type `char`, c'est-à-dire que si on incrément un pointeur générique de 1, la valeur qu'il contient sera augmentée de 1. Cependant il est possible de copier la valeur de n'importe quel pointeur dans un pointeur générique. Dans ce cas c'est l'arithmétique du type de la valeur copiée qui sera utilisée. Par exemple dans le code suivant :

```
void *ptrgen;
int *t;
typequelconque *ptc;

ptrgen = ptrgen + 1; /* la valeur de ptrgen est incrementee de 1 */
ptrgen = (int *) t;  /* on copie la valeur de t dans ptrgen */
ptrgen = (int *) ptrgen + 1; /* la valeur de ptrgen va etre incrementee de 4 */
ptrgen = (typequelconque *) ptc /* la valeur de ptc est copiee dans ptrgen */
ptrgen =  ptrgen + 1; /* la valeur de ptrgen est incrementee de 1 */
```

L'opérateur de cast de type (ce qu'on met entre parenthèse à droite de l'affectation) permet de signifier au compilateur qu'il faut faire l'affectation en interprétant la valeur comme le type spécifié. Ainsi `ptrgen = (int*) t` signifie "copie la valeur de `t` dans `ptrgen` en considérant que c'est un pointeur d'entiers". La même chose se passe quand on écrit `ptrgen = (type*) ptrgen + 1` : le compilateur va comprendre qu'on veut augmenter de 1, en considérant que `ptrgen` est de type `type*`. Si aucun type n'est spécifié le compilateur ne peut pas savoir vers quel type particulier un pointeur générique pointe. En effet on peut songer au code suivant :

```
scanf("%d",&choix);
if (choix)
     ptrgen = t;
   else
     ptrgen = ptc;
ptrgen = ptrgen +1; /* le type de ptrgen depend de l'execution */
```

QUESTION 5 ► Pour se persuader de cela, il est possible d'utiliser l'opérateur *sizeof* qui permet de récupérer la taille (en octet) d'une variable ou d'un type. Écrivez un programme avec une variable *float* et un pointeur sur un float. Afficher la taille avec *sizeof* de ces 2 variables.

QUESTION 6 ► Reprener le code de la question précédente et rajouter un pointeur générique et afficher sa taille.
Mais alors pourquoi dans la plupart du temps, nous n'utilisons pas $void*$? Tout simplement que c'est une source d'erreurs (et de failles de sécurité). Pour cela, il faut comprendre qu'au-delà de pouvoir contenir l'adresse d'une variable, l'intérêt des pointeurs vient également de la capacité à pouvoir les manipuler pour se déplacer en mémoire d'une case mémoire à l'autre *i.e.* arithmétique de pointeurs.
Afin d'illustrer cela plus simplement, nous allons d'abord revenir sur les tableaux et leur lien avec les pointeurs.

2 Tableaux

QUESTION 7 ► Déclarer un tableau *tab* de 10 entiers. Afficher l'adresse de la variable *tab* ainsi que l'adresse de chacune des cases du tableau (*i.e.* *tab*[*i*]). Que pouvons nous observer ?
En pratique, lorsque nous déclarons un tableau, ce tableau est un pointeur vers un espace mémoire de la taille du tableau (10 dans l'exemple précédent) multiplié par la taille d'un élément du tableau (ici la taille d'un *int*).

QUESTION 8 ► Dessiner l'espace mémoire que vous observez dans la question précédente.
Jusqu'à présent, lorsque nous voulions accéder à une case du tableau, nous utilisions l'opérateur []. Pour accéder séquentiellement à toutes les cases d'un tableau, nous écrivions une boucle *for* avec un indice allant de 0 à la taille du tableau - 1 *e.g.* de 0 à 9 dans le cas d'un tableau de taille 10 et oui en informatique, on commence à compter à partir de 0 et pas de 1. Mais il est tout à fait possible d'utiliser de l'arithmétique sur les pointeurs pour parcourir un tableau, accéder à une case et plus généralement manipuler un tableau. Soit un pointeur *ptr_tab* qui stocke l'adresse de la première case d'un tableau, il est possible d'incrémenter de 1 (en utilisant +1 ou l'opérateur ++) le pointeur pour passer d'une case à l'autre. Bien entendu, il est possible d'accéder à la case précédente en décrémentant de 1 (-1 ou $--$). Plus généralement, il est possible à partir d'une case d'accéder à la *n*-ième case suivante en ajoutant *n* au pointeur (la réciproque pour accéder à la *n*-ième case précédente est possible en utilisant l'opérateur de soustraction).

QUESTION 9 ► Déclarer un tableau d'entiers de taille 10 contenant la suite 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Afficher chacune des valeurs du tableau en le parcourant avec un pointeur *ptr_tab*. Puis afficher uniquement les cases impaires en partant de la fini, toujours en utilisant le pointeur.

3 Arithmétique sur les pointeurs

Maintenant que nous avons vu les similaritudes entre les pointeurs et les tableaux, nous pouvons revenir sur les problématiques de *void*$*$ et de l'arithmétique des pointeurs.

QUESTION 10 ► Déclarer un tableau de double de taille 10 contenant la suite 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.10. Afficher chacune des valeurs du tableau en le parcourant avec un pointeur de type *double*$*$. Puis faire de même avec un pointeur *void*$*$, que se passe-t-il ? Faire un dessin du tableau et des valeurs successives des 2 pointeurs. Comment expliquer ce qui a été observé ?
De manière plus générale, il faut faire attention avec l'arithmétique sur les pointeurs car ni à la compilation ni à l'exécution, il n'y a vérification que les adresses mémoires accédées ont été effectivement allouées au programme. Il est donc très simple de provoquer un crash du programme mais également de provoquer des fuites de données. C'est au programmeur de faire attention à cela et de bien vérifier les bornes.

QUESTION 11 ► Déclarer un tableau d'entiers de taille 10 contenant la suite 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Parcourez le tableau avec un pointeur, affichez la valeur de chaque case, mais en allant jusqu'à la case 11. Que se passe-t-il ?

QUESTION 12 ► Déclarer deux tableaux d'entier de taille 10 contenant respectivement les suites 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 et 101, 102, 103, 104, 105, 106, 107, 108, 109, 110. Nous considérons que le premier tableau contient des données publiques et le second des données qui doivent rester privées. Accéder à la 15-ième case du tableau via une variable pointant sur la première case du premier tableau. Que pouvons-nous observer ? Que pouvons-nous en déduire en terme de sécurité ?
Avant de continuer sur les pointeurs et leurs différentes utilisations en C, nous revenons sur les tableaux. Pour l'instant, nous nous sommes limités aux tableaux à 1 dimension *i.e.* des vecteurs. En C, il est possible également de déclarer des tableaux à 2 dimensions *i.e.* des matrices. Bien entendu, il est possible de déclarer des tableaux de n'importe quelles dimensions même si au delà de 3, il devient plus difficile pour l'esprit humain d'appréhender ces tableaux. Il est possible de "simuler" un tableau à N-dimensions en créant un tableau à une dimension (*e.g.* pour une matrice *NxM*, nous pouvons déclarer un vecteur de taille $N * M$) mais il faut ensuite faire des opérations arithmétiques pour accéder à une case à partir des coordonnées en 2D. Il est souvent plus simple de tout simplement déclarer directement un tableau à 2-dimensions. Par exemple, pour une matrice d'entiers de taille *NxM*, il faut déclarer le tableau comme suit `int tab[N][M];`.

QUESTION 13 ► Déclarer une matrice de `double` de taille 10*x*2 et afficher les adresses de chacune des lignes ainsi que des cases.

QUESTION 14 ► Déclarer un vecteur de `double` de taille 20 et afficher les adresses de chacune des cases. Comparer avec le résultat de la question précédente.

4 Les pointeurs et les fonctions

Lees pointeurs sont utilisés pour passer certains paramètres lors d'un appel de fonction.

QUESTION 15 ► Écrivez une fonction `swap(int a, int b)` qui prend 2 entiers *a* et *b* et qui échange leur contenu. Écrivez un programme qui déclare 2 variables entières contenant respectivement 10 et 20. Appellez la fonction *swap* avec ces variables. Affichez la valeur des 2 variables avant et après l'appel à la fonction *swap*(). Que pouvons-nous observer ?
En effet, lorsqu'on fait appel à une fonction qui prend en paramètres des variables, le contenu des variables est copié dans des variables internes à la fonction. À la fin de la fonction, ces variables internes sont détruites. En pratique, à aucun moment les variables ne sont modifiées.

QUESTION 16 ► Afin de vérifier cela, modifiez le programme de la question précédente pour que les variables du programme, *i.e.* celles de la fonction `main()`, soient des variables globales. Affichez les valeurs et adresses des 4 variables au début et à la fin de la fonction `swap()`. Que pouvons-nous observer ?
Comme dit précédemment, une nouvelle variable est créée lors de l'appel d'une fonction avec paramètres *i.e.* un nouvel espace mémoire est utilisé et la valeur est recopiée dedans. Afin d'effectuer un échange de valeurs entre les 2 variables qui aient une portée supérieure à celle de la fonction, il faudrait donc que ce soit les espaces mémoires passés en paramètre qui soient utilisés en interne de la fonction *i.e.* que des duplicats des valeurs ne soient pas fait mais des duplicats des adresses mémoires qui seront par la suite pour modifier la valeur de l'adresse pointée.

QUESTION 17 ► Reprenez le code de la question 17 et modifiez le pour que les paramètres soient maintenant des pointeurs.Affichez les valeurs et adresses des 4 variables au début et à la fin de la fonction *swap*(). Que pouvons-nous observer ?

5 Chaînes de caractères

Une chaîne de caractères, en C, est simplement un tableau rempli de caractères ASCII, plus précisément un tableau de `char`, terminé par le caractère de code 0, c'est à dire `'\0'`.

QUESTION 18 ► Dans un `main()`, affectez à une variable `my_string` la chaîne de caractère `"<-_->"`.

QUESTION 19 ► Dans votre main parcourez cette chaîne de caractère et affichez-les sur la sortie standard séparés par des '.'

QUESTION 20 ► Modifiez votre programme pour qu'il utilise la fonction `size_t strlen(const char *s);` qui renvoie la taille de la chaîne passée en paramètre.

QUESTION 21 ► La fonction `int strcmp(const char *s1, const char *s2);`, utilisable en incluant `<string.h>`, est une fonction (de la librairie C) qui compare les deux chaînes passées en paramètre et renvoie 0 si elles sont égales, un entier infèrieur à 0 si `s1` est plus petite que `s2`, un entier supérieur à 0 sinon. Implémentez `int TP_strcmp(const char *s1, const char *s2);` qui fait la même chose (et interdiction d'inclure `<string.h>` ! je vous vois venir). Retournez une valeur dès que le code ASCII du *i*-ème caractère de `s1` est infèrieur/supérieur au *i*-ème de `s2`.
Que ce soit par des intéractions sur l'entrée / sortie standard mais aussi lors de la lecture de fichiers textes (ou comme nous le ferons plus tard, lors de passage d'arguments à programme), il est souvent nécessaire de transformer une chaîne de caractères en une valeur d'un autre type. Par exemple, via une entrée standard, l'utilisateur saisit le chiffre 1.254, pour l'utiliser comme un flottant, il faut convertir cette chaîne de caractères (contenant les caractères '1', '.', '2', '5', et '4' en un flottant). C'est également le cas dans le sens inverse quand un programme affiche avec `printf()` un type qui n'est pas un caractère. Pour cela, la bibliothéque C propose un ensemble de fonctions qui permettent la transformation dans les deux sens.

QUESTION 22 ► Écrivez une fonction `atoi()` qui transforme une chaine de caractères ASCII en un entier (interdiction d'utiliser le `atoi()` du langage C mais pensez à regarder les valeurs des codes ASCII des chiffres).

QUESTION 23 ► Écrivez une fonction `itoa()` qui transforme un entier en une chaine de caractères ASCII et qu'il l'affiche (interdiction d'utiliser le `itoa()` du langage C mais pensez à regarder les valeurs des codes ASCII des chiffres).

Les pointeurs sont également utilisés pour pouvoir passer des arguments (*i.e.* des paramètres) à un programme. C'est une fonctionnalité extrêmement utile pour éviter de recompiler un programme pour tester différentes valeurs. Par exemple, un programme génére une suite de nombres entiers entre deux chiffres *e.g.* il génére tous les nombres entiers entre 0 et 4 (0 1 2 3 4). Si il est nécessaire de recompiler le programme à chaque fois que ces chiffres changent, la facilité d'utilisation du programme est faible et nécessite des connaissances (*i.e.* savoir compiler un programme) qui est loin d'être nécessaire pour une utilisation simple d'un programme. Le langage C propose donc de pouvoir passer des paramètres à la fonction `main()` et donc au programme. Pour cela, il y a 2 paramètres : `int argv` et `char **argv`. La fonction `main()` se déclare donc `int main(int argc, char** argv)`. Le premier paramètre `argc` permet de récupérer combien de paramètres ont été passés au programme. Le second paramètre `argv` permet de récupérer ces paramètres. En revenant sur ce que nous avons appris dans ce TP, nous pouvons en déduire que `argv` est un tableau de pointeurs de chaines de caractères. En effet, `char** argv` peut également s'écrire `char* argv[]`. Nous avons vu précédemment que `char *chaine` est un pointeur sur une chaine de caractères. Par conséquence, `char** argv` est ni plus ni moins qu'un tableau de chaines de caractères contenant la liste des paramètres passée.

QUESTION 24 ► Écrivez un programme qui affiche le nombre de paramètres passés au programme ainsi que la liste de tous ces paramètres. Que pouvons nous observer de particulier pour le premier argument ?

QUESTION 25 ► Écrivez un programme qui prend en argument un chiffre entier et calcul tous les nombres premiers entre 1 et ce chiffre. Penser à écrire le code permettant de vérifier qu'il y a bien un argument et un seul sinon afficher une aide du type le nom de l'exécutable et le fait qu'il faut un paramêtre (*e.g.* ./*exec*12.

6 Gestion des erreurs

Même un programme écrit parfaitement sans aucune erreur doit pouvoir gérer des erreurs. En effet, il ne faut pas perdre de vue que la première cause de plantage est due à l'interface clavier/chaise. Il faut donc détecter les erreurs et les gérer. C'est ce que nous allons voir dans la fin de ce TP.
La première méthode pour remonter des erreurs est via les codes de retour que ce soit dans un programme ou dans une fonction. Pour commencer, nous allons nous intéresser aux fonctions internes de C qui retourne une valeur alors qu'elles n'ont pas besoin d'en retourner une.
Nous verrons en détail lors du TP4 les entrées/sorties en C mais pour illustrer la gestion des erreurs, nous allons utiliser une fonction qui permet de récupérer des valeurs sur l'entrée standard : `scanf`. La fonction `scanf` prend un nombre d'arguments variables mais nous pouvons les séparer en 2 groupes :

- le premier paramètre est une chaine de caractères (similaire à celle de `printf`). Par exemple, si nous voulons lire un entier, nous allons spécifier la chaine de caractère " %d ".
- les autres paramètres sont les variables dans lesquelles vont être stockées les valeurs lues. Par exemple, une variable entière dans lequelle la valeur va être stockée.

Par conséquence, pour lire un entier, il nous faut définir une variable entière `int a` puis faire l'appel `scanf(''%d'',&a)` (oui il faut passer l'adresse, rappellons nous le milieu du TP). Mais si nous regardons la spécification de `scanf`, il y a une valeur de retour (un entier) : `int scanf(const char *format, ...)`. Mais à quoi peut servir cette valeur de retour alors que fonctionnellement elle ne sert à rien. Une fois de plus, il suffit de lire la documentation de `scanf` : si la lecture sur l'entrée standard a fonctionnée, alors la valeur de retour est le nombre de valeurs lues sinon 0 est retournée.

QUESTION 26 ► Écrivez un programme qui permet de lire sur l'entrée standard un entier et qui affiche la valeur de retour de `scanf`. Essayer de saisir sur l'entrée standard les valeurs : 4, 42, 13.12 et 'abc'. Qu'observons nous ?
Dés que c'est nécessaire, il faut penser à implémenter des comportements similaires quand nous écrivons une nouvelle fonction.
Si nous revenons à la Section sur les pointeurs et les fonctions de ce TP, nous nous souvenons que la fonction `main()` a également une valeur de retour (un entier) : `int main(int argc, char** argv)`. Le but de cette valeur de retour est de coder une gestion d'erreurs. La plupart des programmes de base sous Linux dispose de tel retour d'erreurs. Par exemple, la commande *ls* implémente un code de retour pour faire de potentielles erreurs. En regardant la documentation de *ls*, nous pouvons voir que *ls* a 3 valeurs de retour :

- 0 si tout s'est bien passé
- 1 si il y a eu des problèmes mineurs *e.g.* ne pas pouvoir accéder à un sous-repertoire
- 2 si il y a eu des problèmes sérieurs *e.g.* ne pas pouvoir accéder à un fichier ou répertoire passé sur la ligne de commande en paramètre.

Ces codes de retour d'erreur sont extrêmement utiles lorsque plusieurs programmes sont "liés" entre eux via un script. Sous Linux, il est courant d'utiliser un script BASH pour écrire un enchainement de plusieurs commandes. Tout comme pour un programme C, il est utile de vérifier que les commandes ont bien fonctionnées et c'est via les codes de retour d'erreur que cela peut être vérifier au niveau du script BASH. Pour information, il faut utiliser la variable $? pour récupérer ce code d'erreur dans un script BASH. Il est donc particulièrement recommander de programmer de tel code d'erreurs pour faciliter l'utilisation de votre programme surtout si ce dernier va être utilisé dans un script BASH.
Mais revenons au C, la fonction `main()` permet donc un retour de code d'erreur qui sera le code d'erreur retourné par le programme. Tout comme pour n'importe quelle fonction, nous utiliserons le mot-clef *return* pour provoquer le retour d'une valeur au sein du main.

QUESTION 27 ► Modifiez le programme de la question 26 pour qu'il retourne 0 si tout s'est bien passé et 1 sinon.
Le retour des fonctions est très pratique pour signaler une erreur mais comment faire quand nous avons effectivement besoin de retourner une valeur et que nous voulons aussi implémenter une gestion d'erreurs. Par exemple, la fonction `strlen()` que nous avons vu précédemment retourne la taille de la chaine de caractère, nous ne pouvons pas en même temps récupérer une valeur de code d'erreur. En effet, les fonctions retournent une valeur et pas un ensemble. Alors comment faire dans ce cas ?

La première approche qui est couramment utilisée est de retourner une valeur particulière pour coder les erreurs. Par exemple, les valeurs de retour sont forcément des entiers positifs, nous pouvons donc utiliser des valeurs négatives pour coder des erreurs. Mais ce n'est pas toujours possible de trouver des valeurs particulières et aussi les fonctions ne retournent pas forcément des entiers. Il faut donc utiliser une autre approche dans ce cas. Pour cela, le langage C fournit une variable globale nommée *errno* (il faut inclure `<errno.h>`) qui permet à différentes fonctions d'indiquer une erreur en modifiant la valeur de celle-ci. Attention, *errno* est une variable globale, il est donc nécessaire de toujours la remettre à 0 avant d'appeller une fonction qui est susceptible de la modifier. En effet, elle pourrait contenir la valeur qu'une autre fonction lui aurait assignée auparavant. Les fonctions mathématiques utilisent régulièrement cette fonctionnalité.

QUESTION 28 ► Écrivez un programme qui calcul la puissance de deux nombres via la fonction *pow* (*pow* dépend de la bibliothéque mathématique de C, il faut que vous inclusiez *math.h* et que vous liez votre binaire à la bibliothèque, pour cela ajouter $-lm$ à la fin de votre ligne de compilation) affichez la valeur de *errno*. Essayez de calculer 2^4 et $-1^{\frac{1}{2}}$.
Pour information, C prévoit par défault 3 valeurs possible de *errno* :

- *EDOM* : pour le cas où le résultat d'une fonction mathématique est impossible
- *ERANGE* : dépassement de capacité (voir les prochains TPs)
- *EILSEQ* : erreurs de conversions (même chose)

Afficher des valeurs entières donnent une indication mais dans le cas de fonction compliquée, cela peut devenir rapidement obscure. La fonction `strerror()` permet de transformer un code erreur en chaine de caractère qui doit permettre d'afficher un message plus compréhensible par l'humain. Cette fonction est utile pour les fonctions internes en C.
Nous verrons en détails la gestion des erreurs pour les allocations de mémoire ainsi que les entrées/sorties dans les 2 TPs suivant.
Pour finir, nous allons étudier le fonctionnement de la (macro)fonction *assert*. Elle sert à placer des tests à certains points d'un programme. Dans le cas où un de ces tests est faux, un message d'erreur est affiché (ce dernier comprend la condition dont l'évaluation est fausse, le nom du fichier et la ligne courante) après quoi la fonction *abort*() est appelée afin de produire une image mémoire (*i.e.* un core dump). *assert* est utile pour détecter rapidement des erreurs au sein d'un programme pendant le développement. La convention veut que les assertions soient placées en début de fonction afin de vérifier un certain nombre de condition. Par exemple, pour une fonction *doublediv*(*doublea*, *doubleb*) qui effectue une division entre 2 doubles, il serait possible d'utiliser *assert* pour vérifier que *b* n'est jamais égale à 0.
Comme dit précédemment, les *assert* sont utilisées lors des phases de développement et test des programmes. Elles induisent un sur-coût de calcul (il faut bien vérifier les assertions). Il est donc conseillé de les désactiver lorsque le programme est en production. Pour cela, pas besoin de commenter les assertions, il suffit d'ajouter lors de la compilation l'option $-DNDEBUG$.

QUESTION 29 ► Écrivez et testez la fonction *div* décrite ci-dessus. Essayez de faire la division de 10.0 par 11.0, de 13.0 par -12.0 et de 42.0 par 0. Faites cela avec et sans $-DNDEBUG$.