Premiers programmes en C

Compilation, entrées-sorties, calculs et débogage

Consignes

— Tout le code est écrit dans `main` : aucune autre fonction ne doit être définie.
— Les macros du préprocesseur sont autorisées.
— Compiler systématiquement avec `gcc -Wall` et lire chaque avertissement.
— Pour les exercices de lecture de code, prévoir le résultat *avant* de compiler, puis le vérifier.
— Sauf indication contraire, les saisies sont supposées valides.
— Les rappels renvoient au cours et à la documentation cppreference.

1 Programme vide et compilation

Rappel

— Le point d'entrée est `main` ; `return 0;` signale au système que tout s'est bien passé, une autre valeur signale une erreur.
— Étapes : prétraitement (`.i`), compilation vers l'assembleur (`.s`), assemblage (`.o`), puis édition des liens.
— `gcc -Wall -c f.c` produit `f.o` ; `gcc f.o -o f` produit l'exécutable ; `-save-temps` conserve les fichiers intermédiaires.
— Documentation : fonction main.

Exercice 1. Compiler un programme vide

Créer `vide.c` :

```c
int main(void)
{
    return 0;
}
```

a) Compiler avec `gcc -Wall vide.c -o vide`, puis exécuter `./vide`. Que se passe-t-il à l'écran ?
b) Afficher le code de retour avec `echo $?`.
c) Remplacer `return 0;` par `return 7;`, recompiler et afficher à nouveau le code de retour. Que constate-t-on ?
d) Supprimer la ligne `return`. Le programme compile-t-il ? Quel est alors le code de retour ?

Exercice 2. Les étapes de la compilation

Avec le fichier `vide.c`, exécuter :

```
gcc -Wall -save-temps -c vide.c
ls
```

a) Quels fichiers ont été créés ? À quelle étape correspond chacun ?

b) Le fichier `vide.o` peut-il être exécuté directement ? Pourquoi ?
c) Produire l'exécutable avec `gcc vide.o -o vide`. Quelle étape vient d'être réalisée ?
d) Ouvrir `vide.s` : retrouver l'étiquette correspondant à `main`.

2 Inclusion et affichage avec `printf`

Rappel

— `#include <stdio.h>` insère les déclarations des fonctions d'entrées-sorties : `printf`, `scanf`, etc.
— `printf("Valeur : %d\n", n);` affiche une chaîne formatée. Formats usuels : `%d` (`int`), `%f` (`double`), `%c`, `%s`, `%x`, `%%`.
— Largeur et précision : `%5d`, `%05d`, `%.2f`, `%10.2f`.
— Documentation : printf, #include, séquences d'échappement.

Exercice 3. Premier message

a) Écrire un programme qui affiche `Bonjour !` suivi d'un retour à la ligne.
b) Supprimer la ligne `#include <stdio.h>` et recompiler avec `-Wall`. Relever le message (avertissement ou erreur selon la version de `gcc`) et l'expliquer.
c) Retirer le `\n` final. Que constate-t-on dans le terminal après l'exécution ?

Exercice 4. Formats d'affichage

Écrire un programme qui déclare `int n = 42;` et `double pi = 3.14159265;`, puis affiche, chacun sur une ligne :

```
[42]
[   42]
[00042]
[2a]
[3.141593]
[3.14]
[      3.14]
```

Chaque ligne s'obtient avec un seul `printf` et un format différent.

Exercice 5. Caractères spéciaux

Écrire un programme qui affiche exactement :

```
Il a dit : "Bonjour".
Chemin : C:\TP\exercice.c
Progression : 50%
Colonne1    Colonne2
```

La dernière ligne contient une tabulation.

3 Variables et calcul de $ax + b$

Rappel

— Une variable se déclare avec son type : `int x;`, puis s'affecte : `x = 20;`, ou les deux : `int y = 10;`.
— Types usuels : `char`, `short`, `int`, `long`, `float`, `double`. Leur taille dépend de l'architecture : l'opérateur `sizeof` la donne.
— Division entière si les deux opérandes sont entiers, réelle sinon ; `%` (modulo) n'existe que pour les entiers.
— Affectations composées : `+=`, `-=`, `*=`, `/=`, `%=`.
— Documentation : types arithmétiques, opérateurs arithmétiques, sizeof.

Exercice 6. Déclarer des variables et mesurer leur taille

a) Déclarer `int a = 3, x = 4, b = 2;` et afficher `a = 3, x = 4, b = 2`.
b) Afficher avec `sizeof` la taille en octets de `char`, `int`, `long`, `float` et `double` (format `%zu`). Comparer avec les valeurs du cours pour votre machine.

Exercice 7. Calcul entier de $ax + b$

a) Avec les variables de l'exercice précédent, déclarer `y`, lui affecter $ax + b$, puis afficher `Pour x = 4, y = 14`. Le résultat ne doit pas être écrit en dur dans `printf`.
b) Ajouter, sans nouvelle variable, les instructions `x += 5;` puis `x *= 2;` et afficher `x`. Prévoir la valeur avant d'exécuter.
c) Recalculer `y` avec la nouvelle valeur de `x`. Pourquoi faut-il réécrire l'affectation de `y` ?

Exercice 8. Calcul avec des réels

a) Avec `double a = 2.5, x = 1.2, b = -3.0;`, calculer $y = ax + b$ et afficher les quatre valeurs avec deux décimales.
b) Déclarer `int p = 7, q = 2;` et afficher `p / q`, `p % q`, `(double)p / q` et `p / 2.0`. Expliquer chaque résultat.
c) Pourquoi `%` est-il refusé sur des `double` ? Vérifier la réaction du compilateur.

4 Saisie avec `scanf` et équation du second degré

Rappel

— `scanf` lit sur l'entrée standard et reçoit l'*adresse* de la variable : `scanf("%d", &n);`.
— Formats de lecture : `%d` (`int`), `%f` (`float`), `%lf` (`double`), `%c`. Attention : en lecture, `double` se lit avec `%lf`, alors que `printf` accepte `%f` pour un `double`.
— `scanf` renvoie le nombre de valeurs correctement lues.
— Le discriminant de $ax^2 + bx + c = 0$ est $\Delta = b^2 - 4ac$: deux solutions réelles si $\Delta > 0$, une double si $\Delta = 0$, aucune réelle si $\Delta < 0$.

— `sqrt` est déclarée dans `<math.h>` ; l'édition des liens demande `-lm` : `gcc prog.c -o prog -lm`.
— Documentation : scanf, sqrt, if.

Exercice 9. Saisir a, x et b

a) Écrire un programme qui demande trois réels `a`, `x`, `b` et affiche `Expression : (a) * (x) + (b) = y` avec deux décimales.
b) Remplacer `%lf` par `%f` dans le `scanf`. Que dit `-Wall` ? Que se passe-t-il si l'on exécute quand même ?
c) Récupérer la valeur de retour de `scanf` dans un entier et l'afficher quand on saisit `3 4 5`, puis `3 abc`.

Exercice 10. Discriminant

Écrire un programme qui lit trois `double` `a`, `b`, `c` (avec $a \neq 0$), calcule et affiche Δ, puis indique par un `if`/`else if`/`else` s'il y a deux solutions réelles, une solution double ou aucune solution réelle.

Exercice 11. Solutions réelles

Compléter le programme pour afficher les solutions :

$$x_{1,2} = \frac{-b \mp \sqrt{\Delta}}{2a} \quad (\Delta > 0), \qquad x_0 = \frac{-b}{2a} \quad (\Delta = 0).$$

a) Compiler sans `-lm`. Relever le message d'erreur, puis recompiler correctement.
b) Tester avec le tableau ci-dessous et vérifier les résultats à la main.

a	b	c	Résultat attendu
1	-3	2	$x_1 = 1$, $x_2 = 2$
1	2	1	$x_0 = -1$
1	0	1	aucune solution réelle

Exercice 12. Cas où a est nul

Traiter, en utilisant des tests imbriqués, les cas :

— $a = 0$ et $b \neq 0$: résoudre $bx + c = 0$;
— $a = b = 0$ et $c \neq 0$: afficher `Aucune solution` ;
— $a = b = c = 0$: afficher `Une infinite de solutions`.

Vérifier qu'aucune division par zéro ni racine d'un nombre négatif n'est possible. Pourquoi la comparaison de `double` avec `==` est-elle risquée en général ?

5 Compilation conditionnelle : `DEBUG` et `LOG`

Rappel

— Le préprocesseur travaille sur le texte, avant la compilation.
— `#define NOM valeur` définit une macro ; `#undef NOM` la supprime. Une macro n'a ni type ni mémoire.
— `#ifdef NOM` ... `#endif` conserve le code si `NOM` est défini, quelle que soit sa valeur ; `#if expr` teste la valeur d'une expression constante (un identifiant non défini vaut 0).
— On peut définir une macro à la compilation : `gcc -DDEBUG prog.c -o prog`.
— Opérateur `#` : transforme l'argument en chaîne. Macros prédéfinies : `__FILE__` et `__LINE__`.
— Documentation : inclusion conditionnelle, #define et macros.

Exercice 13. Affichage de diagnostic avec `DEBUG`

Reprendre le programme de l'exercice 10 et ajouter, après la saisie :

```
#ifdef DEBUG
    printf("[DEBUG] a=%f x=%f b=%f\n", a, x, b);
#endif
```

a) Compiler sans `DEBUG` : le message apparaît-il ?
b) Ajouter `#define DEBUG` au début du fichier et recompiler.
c) Retirer cette ligne et compiler avec `gcc -DDEBUG`. Comparer avec la question précédente.
d) Avec `gcc -Wall -save-temps -c`, comparer les fichiers `.i` avec et sans `-DDEBUG`.

Exercice 14. Une macro `LOG`

Placer avant `main` :

```
#ifdef DEBUG
#define LOG(msg) printf("[DEBUG] %s:%d : %s\n", __FILE__, __LINE__, msg)
#else
#define LOG(msg) ((void)0)
#endif
```

a) Dans le programme du second degré, ajouter trois appels : `LOG("saisie terminee");`, `LOG("discriminant calcule");` et `LOG("fin");` aux endroits appropriés.
b) Compiler avec puis sans `-DDEBUG`. Que contiennent les fichiers `.i` à la place de chaque appel ?
c) Écrire une seconde macro `LOG_D(x)` qui affiche le nom et la valeur d'un `double`, par exemple `delta = 1.000000`, en utilisant l'opérateur `#`.
d) Pourquoi `LOG` n'est-elle pas une fonction ?

6 Compréhension et lecture de code

Exercice 15. Qu'affiche le programme suivant ?

```
#include <stdio.h>

int main(void)
{
    int a = 7, b = 2;
```

```c
    double x = 7;

    printf("%d\n", a / b);
    printf("%d\n", a % b);
    printf("%.1f\n", x / b);
    printf("%.1f\n", (double)(a / b));
    printf("%d %d\n", a > b, a == b);

    return 0;
}
```

Donner l'affichage exact, puis justifier chaque ligne.

Exercice 16. Ordre des affectations

```c
#include <stdio.h>

int main(void)
{
    int a = 2, b = 3, x = 4;
    int y = a * x + b;
    int i = 3, j;

    x = 10;
    printf("x = %d, y = %d\n", x, y);
    y = a * x + b;
    printf("y = %d\n", y);

    j = i++ * 2;
    printf("%d %d\n", j, i);
    j = ++i * 2;
    printf("%d %d\n", j, i);

    i += 2;
    i *= 3;
    printf("%d\n", i);

    return 0;
}
```

a) Donner l'affichage exact.
b) Expliquer pourquoi modifier `x` ne modifie pas `y`.
c) Expliquer la différence entre `i++` et `++i` dans les deux affectations de `j`.

Exercice 17. Une macro piégeuse

```c
#include <stdio.h>

#define CARRE(x) x * x
#define DOUBLE(x) x + x

int main(void)
```

```c
{
    int a = 2, b = 3;

    printf("%d\n", CARRE(a + b));
    printf("%d\n", 3 * DOUBLE(2));

    return 0;
}
```

a) Réécrire chaque expression telle que le préprocesseur la produit.
b) Donner l'affichage exact.
c) Corriger les deux macros pour qu'elles se comportent comme prévu.

Exercice 18. `#ifdef` contre `#if`

```c
#include <stdio.h>

#define DEBUG 0

int main(void)
{
    int a = 2, x = 5, b = 1;

#ifdef DEBUG
    printf("Diagnostic actif\n");
#endif

#if DEBUG
    printf("Diagnostic detaille\n");
#endif

    printf("y = %d\n", a * x + b);

    return 0;
}
```

a) Donner l'affichage exact.
b) Donner l'affichage si la définition devient `#define DEBUG 1`.
c) Donner l'affichage si la ligne `#define` est supprimée.
d) Donner l'affichage si le programme est compilé avec `-DDEBUG` et sans la ligne `#define`.

Exercice 19. Effet de bord dans un `LOG`

```c
#include <stdio.h>

#ifdef DEBUG
#define LOG(msg) printf("[DEBUG] %s\n", msg)
#else
#define LOG(msg) ((void)0)
#endif

```

```c
int main(void)
{
    int a = 2, b = 1, x = 3;

    LOG((x = x + 1, "x modifie"));

    printf("x = %d\n", x);
    printf("y = %d\n", a * x + b);

    return 0;
}
```

L'expression `(e1, e2)` évalue `e1`, puis `e2`, et vaut `e2`.

a) Donner l'affichage exact avec `-DDEBUG`.

b) Donner l'affichage exact sans `-DDEBUG`.

c) Expliquer pourquoi le résultat dépend du mode de compilation.

d) Corriger le programme pour que `y` soit identique dans les deux modes, tout en gardant un message de débogage conditionnel.

e) Énoncer une règle sur le contenu des arguments d'une macro de journalisation.

7 Questions à choix multiple

Une seule proposition est correcte par question.

Question 1. Que signifie `return 0;` à la fin de `main` ?

☐ A. Le programme affiche zéro.
☐ B. Le programme se termine en signalant un succès.
☐ C. Le programme recommence.
☐ D. Les variables sont remises à zéro.

Question 2. Quelle commande produit uniquement le fichier objet `prog.o` ?

☐ A. `gcc prog.c -o prog`
☐ B. `gcc -Wall -c prog.c`
☐ C. `gcc -lm prog.c`
☐ D. `gcc prog.o -o prog`

Question 3. Quelle option indique l'utilisation de la bibliothèque mathématique lors de l'édition des liens ?

☐ A. `-Wall`
☐ B. `-c`
☐ C. `-lm`
☐ D. `-DMATH`

Question 4. Quelle directive permet d'utiliser `printf` ?

☐ A. `#include <math.h>`
☐ B. `#include <stdio.h>`
☐ C. `#define stdio`
☐ D. `include stdio`

Question 5. Quelle séquence représente un retour à la ligne dans une chaîne ?

☐ A. `/n`
☐ B. `%n`
☐ C. `\n`
☐ D. `\\n`

Question 6. Que produit `printf("%.2f", 3.14159);` ?

☐ A. `3.14`
☐ B. `3.1`
☐ C. `3.15`
☐ D. `3.14159`

Question 7. Que vaut `7 / 2` en C avec des opérandes entiers ?

☐ A. `3.5`
☐ B. `4`
☐ C. `3`

☐ D. `1`

Question 8. Quelle instruction lit un `double` nommé `x` ?

☐ A. `scanf("%lf", &x);`
☐ B. `scanf("%lf", x);`
☐ C. `scanf("%d", &x);`
☐ D. `scanf(&x, "%lf");`

Question 9. Pourquoi passe-t-on `&x` à `scanf` ?

☐ A. Pour convertir `x` en réel.
☐ B. Pour afficher `x`.
☐ C. Pour fournir l'adresse de la variable à modifier.
☐ D. Pour multiplier `x` par la valeur saisie.

Question 10. Quelle expression calcule correctement $b^2 - 4ac$?

☐ A. `b ^ 2 - 4 * a * c`
☐ B. `b * b - 4 * a * c`
☐ C. `(b * b - 4) * a * c`
☐ D. `b * (b - 4 * a * c)`

Question 11. Si $a \neq 0$ et $\Delta < 0$, que peut-on conclure ?

☐ A. Une solution réelle double.
☐ B. Deux solutions réelles distinctes.
☐ C. Aucune solution réelle.
☐ D. Une infinité de solutions.

Question 12. Que teste `#ifdef DEBUG` ?

☐ A. Que `DEBUG` vaut `1`.
☐ B. Que `DEBUG` est une macro définie.
☐ C. Que `DEBUG` est une variable non nulle.
☐ D. Que le programme contient une erreur.

Question 13. Avec `DEBUG` non défini, que vaut la condition de `#if DEBUG` ?

☐ A. Elle est fausse : l'identifiant vaut `0`.
☐ B. Elle est vraie.
☐ C. Elle provoque une erreur dans tous les cas.
☐ D. Elle dépend de la valeur de la variable `DEBUG`.

Question 14. Que vaut `3 * DOUBLE(2)` avec `#define DOUBLE(x) x + x` ?

☐ A. `12`
☐ B. `10`
☐ C. `8`
☐ D. `6`

Question 15. Que vaut `x` à la fin, compilé *sans* `-DDEBUG` ?

```c
#ifdef DEBUG
#define LOG(m) printf("%s\n", m)
#else
#define LOG(m) ((void)0)
#endif
int x = 4;
LOG((x = 9, "modif"));
```

☐ A. `4`
☐ B. `9`
☐ C. Indéterminée.
☐ D. Le programme ne compile pas.

Documentation de référence

— printf ; scanf
— Inclusion conditionnelle ; Macros ; #include
— Types arithmétiques ; Opérateurs arithmétiques ; sqrt