

Question 1

Correct

Note de 1,00 sur 1,00

🚩 Marquer la question

Quels mots-clés utilise-t-on pour déclarer qu'une variable est une constante ?

- ☐ a. `static`
- ☒ b. `const` ✓
- ☐ c. `virtual`
- ☐ d. `blocked`

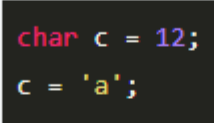

Quelles règles doit suivre un identificateur ?

- ☒ a. Il ne doit pas correspondre à un mot-clé du langage. ✓
- ☐ b. Il ne doit contenir plus que 12 caractères.
- ☐ c. Deux variables déclarées dans un même bloc peuvent avoir le même identificateur.
- ☒ d. Il ne doit pas commencer par un chiffre. ✓

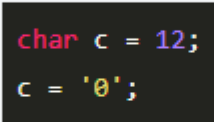

De base, quelles initialisations sont correctes pour un booléen ?

- ☐ a. `_Bool b = true;`
- ☐ b. `_Bool b = 2;`
- ☒ c. `_Bool b = 0;` ✓
- ☐ d. `_Bool b = TRUE;`

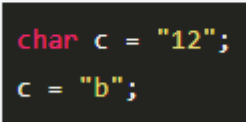

Quels codes sont corrects ?

☒ a.  

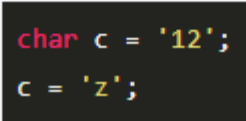

```
char c = 12;  
c = 'a';
```

☒ b.  

```
char c = 12;  
c = '\0';
```



☐ c.  

```
char c = "12";  
c = "b";
```


☐ d.  

```
char c = '12';  
c = 'z';
```

Quels types existent par défaut dans le langage C ?

- ☐ a. rational
- ☒ b. double 
- ☒ c. short 
- ☐ d. tall


Ordonnez les types par taille :

signed char 




short 

int 

long 

long long 

Quels types existent par défaut dans le langage C ?

- ☒ a. signed _Bool 
- ☒ b. short int 
- ☐ c. unsigned double
- ☒ d. signed short 

Quels types existent par défaut dans le langage C ?

- ☐ a. integer
- ☒ b. char ✓
- ☒ c. float ✓
- ☐ d. string

Ordonnez les types par taille :

float ✓ double ✓ long double ✓

Quels sont les éléments obligatoires d'une variable ?

- ☒ a. L'identificateur ✓
- ☐ b. Le contexte
- ☒ c. Le type ✓
- ☐ d. La portée

Remplacez les caractères spéciaux :

\n ✓ : saut de ligne

\t ✓ : tabulation

\\ ✓ : le symbole \

\r ✓ : retour charriot

\m \b \rt \a ^

[Format de saisie]

Donnez l'indicateur de conversion de chacun de ces trois types.

long double

Lf



double

f



float

f



Quels codes sont corrects?

☒ a. 

```
char c;  
scanf("%c", &c);
```

☐ b.

```
char c;  
scanf("%c", c);
```

☐ c.

```
char c;  
scanf("%d", &c);
```

☐ d.

```
char c;  
scanf("%d", c);
```

Quels codes sont corrects?

☐ a.

```
char c = 'a';  
printf("%f\n", c);
```

☒ b. 

```
_Bool b = 0;  
printf("%d\n", b);
```

☒ c. 

```
char c = 'a';  
printf("%c\n", c);
```

☐ d.

```
_Bool b = 0;  
printf("%b\n", b);
```

Les opérations de calcul entre entiers ...

- ☒ a. produisent des entiers. ✓
- ☒ b. s'appellent les opérations arithmétiques. ✓
- ☐ c. sont les suivantes : +, -, *, ÷.
- ☒ d. sont les suivantes : +, -, *, /, %. ✓

Pour lire un `float` et un `double`, les indicateurs de conversion spécifiés dans `scanf` sont identiques.

- ☒ Vrai ✗
- ☐ Faux

Comment spécifier une précision de 2 chiffres après la virgule pour l'affichage d'un réel ?

- ☐ a. `%2f`
- ☐ b. `%f.2`
- ☐ c. `%f2`
- ☒ d. `%.2f` ✓

Comment appelle-t-on un fichier dont le nom se termine par ".h" ?

- ☐ a. Un fichier standard
- ☐ b. Un fichier binaire
- ☐ c. Un fichier source
- ☒ d. Un fichier d'en-tête ✓

Quelle est la valeur de l'expression $27. / 10$?

- ☐ a. 2
- ☐ b. Aucune des réponses précédentes (il y a une erreur).
- ☐ c. 3
- ☒ d. 2,7 ✓

Il est possible que l'instruction suivante n'affiche rien.

```
printf("Bonjour")
```

- ☒ Vrai ✓
- ☐ Faux

Remplacez les opérateurs du **moins** prioritaire au **plus** prioritaire :

! ✗ >= ✗ == ✓ && ✗ || ✗

Quels codes sont corrects ?

☐ a.

```
int i = 0;
for(i < 10; i++) {
    printf("%d\n", i);
}
```

☒ b.

```
int i;
for(i = 0; i < 10; i++)
    printf("%d\n", i);
```



☒ c.

```
int i;
for(i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```



☒ d.

```
for(int i = 0; i < 10; i++)
    printf("%d\n", i);
```



Que peut-on dire sur la boucle `do-while` ?

☒ a. Il y a au moins un tour de boucle.

☒ b. Il est toujours possible de réécrire une boucle `do-while` en utilisant une boucle `while`.

☐ c. Elle est identique à la boucle `while`.

☐ d. Il est possible de ne pas rentrer dans la boucle.

Dans le code suivant:

```
int a = 2, b = 4;
_Bool c = (a != 2) || (b == 4);
```

- ☒ a. Les tests `a!=2` et `b==4` sont tous les deux évalués. ✓
- ☐ b. Seul le test `a!=2` est évalué.
- ☒ c. La valeur de `c` est 1. ✓
- ☐ d. La valeur de `c` est 0.

Avec le code suivant, il y a 2 lignes affichées.

```
int a = 1;

switch(a) {
    case 1:
        printf("Bonjour\n");
    case 2:
        printf("Au revoir\n");
        break;
}
```

- ☐ Vrai
- ☒ Faux ✗

Quels codes sont corrects syntaxiquement?

☒ a.

```
int a = 2, b = 4;

if(a < 2 && b < 4)
    printf("Cool !\n");
```



☐ b.

```
int a = 2, b = 4;

if(a < 2) && (b < 4) {
    printf("Cool !\n");
}
```

☒ c.

```
int a = 2, b = 4;

if((a < 2 && b < 4)) {
    printf("Cool !\n");
}
```



☒ d.

```
int a = 2, b = 4;

if(a < 2 && b < 4) {
    printf("Cool !\n");
}
```



Qu'affiche le code suivant?

```
int a = 1;

do {
    printf("%d", a);
    a++;
} while(a < 5);
```

☐ a. 1
2
3
4

☐ b. 1
2
3
4
5

☐ c. 12345

☐ c. 12345

☒ d. 1234 ✓

Comment s'écrit l'opérateur «ou» en C?

☐ a. &&

☐ b. OU

☐ c. OR

☒ d. || ✓

Dans une instruction `switch`, l'instruction `break` est obligatoire.

- ☐ Vrai
- ☒ Faux ✓

Quels codes permettent de réaliser l'affichage suivant?

1234

- ☐ a.

```
int a = 0;
while(a++ <= 4)
    printf("%d", a);
```
- ☒ b.

```
int a = 0;
while(++a <= 4)
    printf("%d", a);
```

 ✓
- ☐ c.

```
int a = 1;
while(a < 4) {
    a++;
    printf("%d", a);
}
```
- ☐ d.

```
int a = 0;
while(a++ < 4)
    printf("%d", a-1);
```

Les valeurs de retour du `main` sont...

- ☒ a. définies dans `stdlib.h` ✓
- ☐ b. `EXIT_SUCCESS`, qui vaut 1
- ☒ c. `EXIT_SUCCESS`, qui vaut 0 ✓
- ☐ d. `EXIT_FAIL`, qui vaut 1

On considère le code suivant:

```
int f(void)
{
    return 5;
}
```

Quelles affirmations sont justes?

- ☐ a. Il s'agit de la déclaration de la fonction `f`.
- ☒ b. Ce code définit la fonction `f`. ✓
- ☒ c. La fonction `f` renvoie un entier sans qu'on lui transmette quoi que ce soit. ✓
- ☒ d. `f` peut être appelée avec `int x = f();` ✓

Parmi les affirmations suivantes sur les paramètres d'une fonction, lesquelles sont justes?

- ☐ a. Une fonction peut avoir autant de paramètres qu'on le souhaite.
- ☒ b. Les paramètres sont les variables qui reçoivent les valeurs des arguments. ✓
- ☒ c. Si une fonction doit être utilisée sans paramètre, on met `void` dans les parenthèses pour sa déclaration et sa définition. ✓
- ☒ d. Dans les prototypes, on peut ne mettre que la liste des types. ✓
- ☒ e. Pendant l'exécution, les paramètres se comportent comme des variables locales. ✓
- ☒ f. Un argument est une expression qui permet d'obtenir la valeur à transmettre au paramètre correspondant. ✓

On considère le code suivant:

```
#include <stdio.h>

void augmenter(int n) {
    n++;
}

int plus1(int a) {
    return a + 1;
}

int main(void) {
    int n = 5;
    int n1, n2, n3;

    n1 = n;
    augmenter(n);    n2 = n;
    n = plus1(n);    n3 = n;

    printf("%d %d %d\n", n1, n2, n3);
    return EXIT_SUCCESS;
}
```

Quel affichage est produit par le programme?

- ☐ a. 5 6 7
- ☒ b. 5 5 6 ✓
- ☐ c. Aucun affichage: ce code contient une/des erreur(s).
- ☐ d. 5 5 5

L'intérêt des fonctions est...

- ☐ a. qu'on les écrit dans des fichiers séparés.
- ☒ b. qu'elles peuvent être utilisées autant de fois que nécessaire. ✓
- ☒ c. qu'elles permettent de structurer le code. ✓
- ☐ d. qu'elles retournent forcément une valeur.

Votre réponse est correcte.

Les réponses correctes sont : qu'elles permettent de structurer le code., qu'elles peuvent être utilisées autant de fois que nécessaire.

En langage C, les fonctions...

- ☐ a. contiennent forcément une instruction `return`.
- ☐ b. peuvent avoir des variables locales dont la portée est de niveau fichier.
- ☒ c. peuvent utiliser des variables globales. ✓
- ☒ d. peuvent avoir des variables locales. ✓

```

#include <stdio.h>
#include <stdlib.h>

int a = 5;

int f(int n) {
    int a = n;
    if (a >= 0)
        printf("+");
    else {
        int a = -n;
    }
    return a + n;
}

int main(void) {
    a = f(a);
    a = f(a);
    printf("%d\n", a);
    return EXIT_SUCCESS;
}

```

Quelles affirmations sont justes?

- ☒ a. L'affichage est: ☑

++20

- ☐ b. La variable globale est masquée dans le `main`.
- ☐ c. Ce code contient une/des erreurs.
- ☐ d. L'affichage est:

+0

Parmi ces codes, quels sont ceux qui permettent de saisir deux valeurs entières?

- ☒ a.

```
int a, b;
while ( scanf("%d %d", &a, &b) < 2 ) ;
```

 ✓ tordu, mais OK !!!
- ☒ b.

```
int a, b, n;
do
    n = scanf("%d %d", &a, &b);
while (n < 2);
```

 ✓
- ☐ c.

```
int a, b;
a = scanf("%d", &b);
```
- ☒ d.

```
int a, b;
do
    ;
while ( scanf("%d %d", &a, &b) < 2 );
```

 ✓ pas facile !!

On considère le code suivant:

```
void f(void)
{
    printf("Dans f.\n");
}
```

Quelles affirmations sont justes?

- ☒ a. `f` doit être appelée sans paramètre. ✓
- ☒ b. Ce code définit `f`, qui est une fonction. ✓
- ☐ c. `f` peut être appelée avec des paramètres.
- ☒ d. `f` réalise un traitement sans renvoyer de valeur. ✓

```
// A.c
int a = 1;
extern int b;

int f(int n) {
    return a + b + n;
}
```

```
// B.c
#include <stdio.h>
#include <stdlib.h>

extern int f(int);
extern int a;


int b = 2;

int main(void) {
    int c = 3, d = f(a-1);
    printf("%d %d %d %d\n", a, b, c, d);
    return EXIT_SUCCESS;
}
```


On compile avec la commande

```
gcc A.c B.c -o toto
```

Parmi les affirmations suivantes, quelles sont celles qui sont justes ?




- ☒ a. La compilation se déroule sans problème et on peut exécuter le programme avec la commande 

```
./toto
```

- ☐ b. Aucune des autres affirmations n'est juste.
- ☐ c. On obtient une/des erreurs à la compilation.
- ☐ d. La compilation se déroule sans problème mais on a une/des erreurs lors de l'exécution.
- ☒ e. La compilation se déroule sans problème et le programme produit l'affichage suivant ? 

```
1 2 3 3
```

Parmi les affirmations suivantes, quelles sont celles qui sont justes ?

- ☒ a. On peut augmenter la portée d'une variable locale en la déclarant avec le mot clé `static`: dans ce cas sa portée est de niveau fichier. 
- ☒ b. La portée par défaut des variables locales est de niveau bloc. 
- ☐ c. Une variable globale peut être rendue exclusive à un fichier source en la déclarant avec le mot clé `inside`.
- ☒ d. Deux variables de même nom ne peuvent pas être toutes les deux visibles au sein du même bloc. 

Quelles sont les affirmations justes concernant le code suivant?

```
#include <stdio.h>
#include <stdlib.h>

int a;

int main(void) {
    int b;
    b = a + 1;
    printf("%d\n", a+b);
    return EXIT_SUCCESS;
}
```

- ☐ a. La variable `a` est de classe de stockage basique.
- ☐ b. La variable `a` est de classe de stockage automatique.
- ☒ c. Les variables de classe de stockage statique sont initialisées par défaut à 0. ✓
- ☒ d. Pour que la variable `b` soit de classe de stockage statique il aurait fallu la déclarer avec le mot clé `static`. ✓

Quelles sont les affirmations justes concernant le code suivant ?

```
#include <stdio.h>
#include <stdlib.h>

int a;

int main(void) {
    int b;
    b = a + 1;
    printf("%d\n", a+b);
    return EXIT_SUCCESS;
}
```

- ☐ a. Hors du `main`, `b` vaut 0.
- ☐ b. Ce code contient une/des erreur(s).
- ☐ c. `a` n'existe pas dans le `main`.
- ☒ d. `a` contient la valeur 0. ✓

Dans le cadre de la compilation séparée, quelles affirmations sont justes ?

- ☐ a. On inclut les fichiers d'en-tête de nos développements entre chevrons doubles (`<<` et `>>`), avec `#include <<FICHIER>>`
- ☒ b. On inclut les fichiers d'en-tête de la bibliothèque standard entre ✓ chevrons simples (`<` et `>`), avec `#include <FICHIER>`
- ☐ c. On utilise `#include` pour inclure les fichiers objets dans l'exécutable.
- ☒ d. Le plus souvent on écrit un fichier d'en-tête par fichier source, à ✓ part pour le fichier source principal (celui contenant le `main`).
- ☒ e. On inclut les fichiers d'en-tête de nos développements entre ✓ guillemets, avec `#include "FICHIER"`.

On considère un programme basé sur les fonctions suivantes :

- `f1()` et `f2()`, définies dans «A.c», avec `f1()` appelée dans `f2()`
- `g()`, définie dans «B.c», qui utilise `f1()`
- `main`, définie dans «principal.c», qui utilise uniquement `g()` en plus des fonctions de la bibliothèque standard

Parmi ces affirmations, quelles sont celles qui sont justes ?

- ☐ a. On a besoin de fichiers «A.h», «B.h» et «principal.h».
- ☐ b. Si on a écrit les fichiers d'en-tête «A.h», «B.h», contenant les prototypes des fonctions définies dans «A.c», «B.c», alors il faut au minimum les inclusions suivantes :
 - «A.h» inclus dans «A.c», «B.c» et «principal.c»
 - «B.h» inclus dans «A.c», «B.c» et «principal.c»
 - «B.h» inclus dans «A.h»
- ☐ c. Aucune des autres affirmations n'est juste.
- ☐ d. On doit écrire un fichier «Header.h», qui contient les définitions des trois fonctions `f1()`, `f2()` et `g()`, et on doit l'inclure dans «principal.c».
- ☒ e. Si on a écrit les fichiers d'en-tête «A.h», «B.h», contenant les prototypes des fonctions définies dans «A.c», «B.c», alors il faut au minimum les inclusions suivantes :
 - «A.h» inclus dans «A.c» et «B.c»
 - «B.h» inclus dans «B.c» et «principal.c»

Dans le cadre de la compilation séparée avec la commande `make` ...

- ☐ a. l'édition des liaisons consiste à préciser le(s) fichier(s) d'en-tête qui doit/doivent être inclus dans les différents fichiers sources.
- ☒ b. on ajoute souvent une règle `clean` pour nettoyer le répertoire courant (enlever les fichiers temporaires et les fichiers objets, voire l'exécutable). ✓
- ☒ c. on fait figurer les fichiers d'en-tête parmi les dépendances. ✓
- ☐ d. on génère un fichier objet pour chaque fichier source, avec l'option `-obj` de `gcc`.

Les règles d'un fichier de makefile sont de la forme :

- ☐ a. `DEP1, DEP2, DEP3 => CIBLE`
`:: COMMANDE1`
`:: COMMANDE2`
- ☒ b. `CIBLE: DEP1 DEP2 DEP3` ✓
`[TAB] COMMANDE1`
`[TAB] COMMANDE2`
- ☐ c. `CIBLE: DEP1, DEP2, DEP3`
`[TAB] COMMANDE1`
`[TAB] COMMANDE2`

Dans cette dernière partie du test, on va considérer une application réelle. Cette question n'en est pas une: elle présente l'application considérée.

Il s'agit de simuler un grand nombre de lancers de dés, et de compter le pourcentage de fois où on obtient la valeur 6.

Mais en fait on va tricher, en utilisant un dé un peu particulier: il se relance si la valeur est trop petite (3 et moins).

Cette application s'appuie sur trois fichiers sources:

- «gen.c» contient au moins la fonction `tirage1()`, dont la signature est disponible dans le fichier d'en-tête «gen.h»
- «triche.c» contient les fonctions nécessaires pour tricher, et on a aussi un fichier d'en-tête «triche.h»
- «principal.c» contient la fonction `main()` de l'application

Voici les codes de certains des fichiers. Ils seront répétés dans les questions portant sur cette application.

```
// gen.h

/* .... */

// génère un entier aléatoire compris entre 1 et 6 (pour un la
int tirage1(void);

/* ... */
```



```
// triche.c

/* INCLUSIONS */

int valeur; // variable globale (pour la valeur générée)

_Bool ok(void) {
    valeur = tirage1();
    return (valeur >= 4);
}

int tirage2(void) {
    // si la valeur générée est suffisante on la garde ; sinon
    return ok() ? valeur : tirage1();
}
```

```
// principal.c

/* INCLUSIONS */

int main(void) {
    const int essais = 10000;
    int nb = 0;
    for (int i = 1 ; i <= essais ; i++)
        if ( tirage2() == 6 )
            nb++;
    printf("Sur %d essais, on a eu %.2f %% de 6.\n", essais, nb*100/essais);

    return EXIT_SUCCESS;
}
```

L'application estime le pourcentage de 6 pour un grand nombre de lancers de dé (dé pipé par le module **triche**).

Fichiers sources :

- «gen.c» contient au moins la fonction **tirage1()**, dont la signature est disponible dans le fichier d'en-tête «gen.h»
- «triche.c» contient les fonctions nécessaires pour tricher, et on a aussi un fichier d'en-tête «triche.h»
- «principal.c» contient la fonction **main()** de l'application

Extraits de codes :

```
// gen.h

/* .... */

// génère un entier aléatoire compris entre 1 et 6 (pour un la
int tirage1(void);

/* ... */
```

```
// triche.c

/* INCLUSIONS */

int valeur; // variable globale (pour la valeur générée)

_Bool ok(void) {
    valeur = tirage1();
    return (valeur >= 4);
}

int tirage2(int n) {
    // si la valeur générée est suffisante on la garde ; sinon
    return ok() ? valeur : tirage1();
}
```

```
// principal.c

/* INCLUSIONS */

int main(void) {
    const unsigned essais = 10000;
    int nb = 0;
    for (int i = 1 ; i <= essais ; i++)
        if ( tirage2() == 6 )
            nb++;
    printf("Sur %d essais, on a eu %.2f %% de 6.\n", essais, n

    return EXIT_SUCCESS;
}
```

Dans `triche.h`,

- ☐ a. on peut se contenter de déclarer la fonction `ok()`.
- ☐ b. il faut déclarer les deux fonctions `ok()` et `tirage2()`.
- ☒ c. on peut se contenter de déclarer la fonction `tirage2()`. ✓

Dans `triche.c`,

- ☐ a. il faut inclure les deux fichiers d'en-tête de la bibliothèque standard `stdio.h` et `stdlib.h`.
- ☐ b. on peut inclure `triche.h` sans `gen.h`.
- ☐ c. on peut inclure `gen.h` sans `triche.h`.
- ☒ d. il faut inclure `gen.h` et `triche.h`. ✓

Pour les pointeurs, l'opérateur &...

- ☒ a. est l'opérateur d'adressage. ✓
- ☒ b. est un opérateur unaire. ✓
- ☐ c. est l'opérateur de récupération.
- ☐ d. est l'opérateur de déréférencement.

Avec le code suivant

```
int *a, b;  
int* c, d;
```

- ☒ a. a est un pointeur d'int ✓
- ☒ b. b est un int ✓
- ☒ c. c est un pointeur d'int ✓
- ☐ d. d est un pointeur d'int

On considère le code suivant:

```
#include <stdio.h>
#include <stdlib.h>

void f(int *p) {
    int x = *p;
    x++;
    printf("%d ", x);
}

int main(void) {
    int n = 5;
    f(&n);
    printf("%d", n);
    return EXIT_SUCCESS;
}
```

Quel est l'affichage réalisé par ce programme?

Réponse : 6 5



Les principaux cas d'utilisation des pointeurs en C sont :

- ☒ a. le passage de références à des fonctions ✓
- ☐ b. le suivi de l'évolution des zones mémoire utilisées par nos programmes: pile, tas, segment de données, code exécutable
- ☐ c. l'organisation des variables des différentes classes de stockage (statique / automatique)
- ☒ d. l'allocation dynamique de mémoire ✓
- ☒ e. la manipulation de données complexes, stockées dans des structures ou des tableaux ✓

On considère le code suivant, qui demande deux valeurs entières à l'utilisateur et qui identifie laquelle est le minimum des deux : ça se fait par un appel de la fonction `adr_min()`, qui renvoie l'adresse de celle des deux variables qui contient la valeur la plus petite.

```
#include <stdio.h>
#include <stdlib.h>

int * adr_min(int, int) {
    /* ... */
}

int main(void) {
    int x, y;
    scanf("%d %d", &x, &y);
    int *ptr = adr_min(x,y);
    printf("%d\n", *ptr);
    return EXIT_SUCCESS;
}
```

Identifiez les formes correctes pour la fonction `adr_min()`:

☐ a.

```
int * adr_min(int a, int b) {  
    return a < b ? &a : &b;  
}
```

☐ b.

```
int * adr_min(int a, int b) {  
    if ( a < b )  
        return &a;  
    return &b;  
}
```

☒ c. Aucune des autres propositions n'est correcte: il y a une erreur ou un warning à la compilation, ou la valeur renvoyée n'est pas cohérente. ✓

☐ d.

```
int * adr_min(int a, int b) {  
    int c = a < b ? a : b;  
    return &c;  
}
```

☐ e.

```
int * adr_min(int a, int b) {  
    return &( a < b ? a : b );  
}
```

Considérons ce code:

```
int a;  
int *p;  
p = &a;
```

- ☒ a. `p` permet d'obtenir l'adresse de `a`: `printf("adresse de a : %p\n", p);` ✓
- ☐ b. `p` permet de modifier l'adresse de `a` par `int b`: `p = &b;`
- ☒ c. `p` permet de modifier la valeur de `a`: `*p = 3;` ✓
- ☒ d. `p` permet d'obtenir la valeur de `a`: `int n = *p;` ✓

```
#include <stdio.h>
#include <stdlib.h>

void modif(double *p) {
    /* ... */
}

int main(void) {
    double x;
    scanf("%lf", &x);
    /* appel de modif() */
    printf("%f\n", x);
    return EXIT_SUCCESS;
}
```

Identifiez les formes correctes pour l'appel de la fonction `modif()`:

- ☐ a. `double *p = &x;`
`modif(*p);`
- ☒ b. `modif(&x);` ✓
- ☐ c. `modif(x);`
- ☒ d. `double *p = &x;`
`modif(p);` ✓

Un pointeur sur `int`...

- ☒ a. est une variable pouvant stocker l'adresse d'un objet de type `int`. ✓
- ☐ b. contient la valeur par défaut 0 (s'il n'a pas été initialisé explicitement).
- ☐ c. doit être initialisé lors de sa déclaration.
- ☒ d. est une valeur, qui correspond à l'adresse d'un objet de type `int`. ✗
- ☒ e. doit être initialisé avant d'être utilisé, comme toutes les variables de classe de stockage automatique. ✓

Avec le code suivant,

```
int a;  
int *p = &a;
```

- ☒ a. `*(&a)` et `a` désignent la même chose. ✓
- ☒ b. `&(*p)` et `&a` désignent la même chose. ✓
- ☒ c. `*p` désigne la variable `a`. ✓
- ☐ d. `p*&` correspond à `p`.
- ☒ e. `&a` et `&p` sont des valeurs. ✓
- ☒ f. `a` et `p` sont des variables. ✓

Pour les pointeurs, l'opérateur `&`...

- ☒ a. est l'opérateur de référencement. ✓
- ☐ b. est un opérateur binaire.
- ☐ c. est l'opérateur d'indirection.
- ☐ d. peut être remplacé par `[0]` : par exemple `t[0]`.

On suppose que les types simples ont les tailles et les contraintes d'alignement suivantes:

```
_Bool : 1      char : 1
short : 2      int : 4      long : 8      long long : 16
float : 4      double : 8   long double : 16
```

On considère le type structuré

```
struct A {
    double d;
    long double ld;
    int i;
};
```

Quelle est la taille, en octets, des variables de ce type?

La réponse correcte est : 40

On définit le type structuré A par

```
struct A {  
    double a;  
    int n;  
};
```

On souhaite pouvoir échanger les contenus de deux variables de type struct A à l'aide de la fonction echA().

Quels sont les codes corrects pour définir la fonction echA() ?

☒ a.

```
void echA(struct A *p1, struct A *p2) {  
    struct A x;  
    x = *p1;          *p1 = *p2;          *p2 = x;  
}
```



☒ b.

```
void echA(struct A *p1, struct A *p2) {  
    struct A x;  
    x.a = p1->a;      p1->a = p2->a;      p2->a = x.a;  
    x.n = p1->n;      p1->n = p2->n;      p2->n = x.n;  
}
```



☐ c.

```
void echA(struct A *p1, struct A *p2) {  
    struct A *p;  
    p = p1;          p1 = p2;          p2 = p;  
}
```

☐ d.

```
void echA(struct A *p1, struct A *p2) {  
    struct A x;  
    x.a = p1.a;      p1.a = p2.a;      p2.a = x.a;  
    x.n = p1.n;      p1.n = p2.n;      p2.n = x.n;  
}
```

Avec

```
struct a { double x, int n; };  
struct a s;  
struct a *p = &s;
```

On peut accéder au champ `x` de `s` par

- ☒ a. `*(&s.x)` ✓
- ☐ b. `s->x`
- ☒ c. `p->x` ✓
- ☒ d. `(*p).x` ✓
- ☐ e. `*p.x`
- ☒ f. `s.x` ✓

Avec

```
struct { double x, y; } a, *b;  
b = &a;
```

- ☐ a. `b->y` est une écriture simplifiée pour `(&a).y`.
- ☒ b. `b->y` est une écriture simplifiée pour `(*b).y`. ✓
- ☐ c. `b->y` est une écriture simplifiée pour `*b.y`.
- ☐ d. `a->y` est une écriture simplifiée pour `(*b).y`.

Les structures en langage C :

- ☐ a. Les variables d'un type de structure doivent être initialisées au moment de leur déclaration.
- ☒ b. Définir une structure c'est donner son étiquette et la liste des champs, chacun avec son type. ✓
- ☐ c. Dans une structure, les éléments agrégés sont tous de types différents.
- ☒ d. Lorsqu'on définit une structure, le nouveau type créé contient le mot-clé `struct`. ✓
- ☐ e. Les variables d'un type de structure sont forcément de classe de stockage automatique.

Avec

```
struct { int a, b; } x,y;
```

- ☐ a. `y[a]` permet d'accéder au champ `a` de la variable `y`.
- ☐ b. On déclare le type structuré `x` et une variable `y` de ce type.
- ☐ c. `y(a)` permet d'accéder au champ `a` de la variable `y`.
- ☒ d. On déclare deux variables `x` et `y` d'un type structuré anonyme. ✓

Avec

```
struct a { double x, int n; };  
struct a s;  
struct a *p = &s;
```

Pour obtenir l'adresse du champ `x` de `s` on peut utiliser

- ☐ a. `p->x`
- ☐ b. `(&s).x`
- ☒ c. `&(p->x)` ✓
- ☒ d. `&s.x` ✓

Avec

```
struct { double x, y; } a, *b;  
b = &a;
```

- ☐ a. On peut écrire indifféramment `a->x` et `b.x`.
- ☒ b. On peut écrire indifféramment `a.x` et `b->x`. ✓
- ☐ c. `a->y` est une écriture simplifiée pour `(*b).y`.
- ☐ d. `b->y` est une écriture simplifiée pour `&a.y`.

On considère le code suivant:

```
struct x { int a, b; } y, z;
```

- ☒ a. On définit ainsi le type `struct x` et deux variables `y` et `z` de ce type. ✓
- ☐ b. Il y a une erreur de syntaxe: on aurait dû écrire
- ☐ c. On définit ainsi le type `x` et deux variables `y` et `z` de ce type.
- ☒ d. Ce code est équivalent à ✓

```
struct x { int a, b; };  
struct x y, z;
```

On définit le type structuré `A` par

```
struct A {  
    double a;  
    int n;  
};
```

On souhaite pouvoir initialiser les deux champs d'une variable de ce type à l'aide d'une fonction `initA()`.

Quels sont les codes corrects pour définir la fonction `initA()` ?

- ☒ a. `void initA(struct A *p, double a, int n) {
 p->a = a; p->n = n;
}` ✓
- ☐ b. `void initA(struct A x, double a, int n) {
 &x.a = a; &x.n = n;
}`
- ☐ c. `void initA(struct A x, double a, int n) {
 x.a = a; x.n = n;
}`
- ☒ d. `void initA(struct A *p, double a, int n) {
 p.a = a; p.n = n;
}` ✗

On considère un tableau d'entiers de dimension 2, défini par:

```
int tab2[2][3] = { {1,2} , {3} };
```

- ☐ a. On suppose que $0 \leq L < 2$ et $0 \leq C < 3$.
L'adresse de la case de ce tableau située dans la colonne C et la ligne L est donnée par la formule mathématique suivante: $\text{tab2} + (3*L + C) * \text{sizeof}(\text{int})$.
- ☒ b. La 5ème case de ce tableau 2D est `tab2[1][1]`. ✓
- ☒ c. `tab2[1][2]` contient la valeur 0. ✓
- ☒ d. En consultant `sizeof tab2` aussitôt après la définition de `tab2`, on obtient la valeur 24. ✓

Avec

```
int t1[] = { 0, 1, 2 };  
int t2[3] = { [1]=1, 2 };
```

le test

```
t2 == t1
```

- ☐ a. provoque une erreur, à la compilation.
- ☐ b. permet de tester si les contenus des deux tableaux sont identiques.
- ☒ c. permet de tester s'il s'agit du même tableau... ce qui est faux ici. ✓
- ☐ d. provoque un warning à la compilation et une erreur à l'exécution.

`tab` désigne un tableau de 4 entiers.

Parmi les affirmations suivantes, quelles sont celles qui sont justes ?

- ☐ a. `tab + 1` contient l'adresse du premier élément du tableau.
- ☒ b. `tab + 1` contient l'adresse du deuxième élément du tableau. ✓
- ☒ c. `tab` contient l'adresse du premier élément du tableau. ✓
- ☐ d. `tab + 4` contient l'adresse du deuxième élément du tableau (un `int` fait 4 octets).

On considère le code suivant:

```
#include <stdio.h>
#include <stdlib.h>

/* définition de saisieT() */
/* définition de affichageT() */

int main(void) {
    int tab[100];
    /* appel de saisieT() */
    /* appel de affichageT() */

    return EXIT_SUCCESS;
}
```

Parmi les appels suivants, lesquels sont corrects?

Votre réponse est partiellement correcte.

Vous en avez sélectionné correctement 2.

Les réponses correctes sont :

```
saisieT(tab, 100);
```

```
saisieT(&tab, 100);
```

```
affichageT(tab, 100);
```

Avec

```
int tab[3] = {0, 1, 2, 3};  
tab[3] = 0;
```

- ☐ a. le tableau a 4 cases.
- ☒ b. le tableau a 3 cases. ✓
- ☐ c. on obtient une erreur au moment de la compilation.
- ☐ d. on obtient un warning à la compilation et une erreur à l'exécution.

Les réponses correctes sont : on obtient un warning à la compilation et une erreur à l'exécution., le tableau a 3 cases.

Avec

```
int t1[] = { 1, 2, 3 };  
int t2[3];
```

l'instruction

```
t2 = t1;
```

- ☐ a. permet de tester si les contenus des deux tableaux sont identiques.
- ☐ b. provoque un warning à la compilation et une erreur à l'exécution.
- ☒ c. provoque une erreur, à la compilation. ✓
- ☐ d. permet de recopier intégralement le contenu des cases de **t1** dans celles de **t2** (ça n'est possible que parce que les deux tableaux ont la même taille).

```

/* définition de saisieT() */
/* définition de affichageT() */

int main(void) {
    int tab[100];
    /* appel de saisieT() */
    /* appel de affichageT() */

    return EXIT_SUCCESS;
}

```

le prototype de la fonction `affichageT()` peut être

- ☒ a. `void affichageT(int *t, int nb);` ✓
- ☐ b. `void affichageT(int t*);`
- ☒ c. `void affichageT(int t[], int nb);` ✓
- ☐ d. `void affichageT(int t[5]);`

`tab` désigne un tableau d'entiers.

L'affirmation suivante est-elle vraie ?

`tab` et `&tab` ont la même valeur.

- ☒ Vrai ✓
- ☐ Faux

Un tableau d'entiers ...

- ☐ a. a au plus un million de cases.
- ☒ b. a toutes ses cases rangées côte à côte en mémoire. ✓
- ☐ c. peut contenir des réels.
- ☐ d. doit être forcément déclaré en précisant son nombre de cases.

Avec

```
int tab[] = {3, 4, [3]=5, 6};
```

- ☐ a. le tableau a 4 cases.
- ☐ b. `tab[4]` vaut 6.
- ☒ c. `tab[2]` vaut 0. ✓
- ☒ d. le tableau a 5 cases. ✓

Votre réponse est partiellement correcte.

Vous en avez sélectionné correctement 2.

Les réponses correctes sont : `tab[4]` vaut 6., `tab[2]` vaut 0., le tableau a 5 cases.

L'allocation dynamique d'un tableau multidimensionnel en un bloc...

- ☐ a. ne nécessite pas de connaître les tailles dans chaque dimension mais le nombre de cases global.
- ☐ b. ne peut pas être réalisée par VLA.
- ☒ c. oblige à une "gymnastique" pour simuler un tableau 2D (par exemple) comme un tableau 1D. ✗
- ☐ d. nécessite de connaître les tailles dans chaque dimension plutôt que le nombre de cases total.

Votre réponse est incorrecte.

La réponse correcte est : ne nécessite pas de connaître les tailles dans chaque dimension mais le nombre de cases global.

Avec `malloc()`...

- ☒ a. la zone mémoire allouée est d'un seul tenant en mémoire. ✔
- ☐ b. la zone mémoire allouée est directement initialisée à 0.
- ☐ c. on a besoin d'autant de mémoire pour 50 objets `double` que pour 100 objets `int` (avec les taille habituelles pour les types primitifs).
- ☒ d. on doit transtyper l'adresse obtenue en fonction du type de l'objet: par exemple la caster en `double *` pour un tableau de `double`. ✗

Votre réponse est incorrecte.

Les réponses correctes sont : la zone mémoire allouée est d'un seul tenant en mémoire., on a besoin d'autant de mémoire pour 50 objets `double` que pour 100 objets `int` (avec les taille habituelles pour les types primitifs).

Pour allouer un tableau 2D `t` de taille 3x5 par la stratégie "en peigne"...

- ☒ a. on alloue un tableau *extérieur* `t` de trois `int *`. ✓
- ☐ b. on alloue un tableau *extérieur* `t` de cinq `int *`.
- ☐ c. les tableaux *intérieurs* à rattacher au tableau *extérieur* `t` sont de trois `int`.
- ☒ d. les tableaux *intérieurs* à rattacher au tableau *extérieur* `t` sont de cinq `int`. ✓
- ☐ e. on peut faire la libération mémoire en une seule fois par `free(f)`.
- ☒ f. on doit faire la libération mémoire en deux fois, en commençant par la libération du tableau *extérieur* puis en libérant les tableaux *intérieurs*. ✗

Votre réponse est incorrecte.

Les réponses correctes sont : on alloue un tableau *extérieur* `t` de trois `int *`., les tableaux *intérieurs* à rattacher au tableau *extérieur* `t` sont de cinq `int`.

Un VLA...

- ☐ a. est alloué sur le tas mais doit être géré comme s'il était sur la pile.
- ☒ b. ne peut pas avoir une taille aussi grande que si on l'allouait dynamiquement. ✓
- ☒ c. ne peut pas être initialisé au moment de sa définition par une instruction du type `int t[n] = { 1, [2]=3 };` (si `n > 2`). ✗
- ☐ d. est alloué sur la pile mais doit être géré comme s'il était sur le tas.

Votre réponse est incorrecte.

Les réponses correctes sont : est alloué sur le tas mais doit être géré comme s'il était sur la pile., ne peut pas avoir une taille aussi grande que si on l'allouait dynamiquement.

L'allocation dynamique de mémoire...

- ☐ a. nécessite de libérer la mémoire avec `reef()`.
- ☒ b. permet d'adapter la quantité de mémoire utilisée aux besoins réels de l'application. ✓
- ☐ c. s'effectue sur la pile.
- ☒ d. s'effectue sur le tas. ✓

Pour l'allocation dynamique de mémoire réalisée explicitement...

- ☒ a. il y a deux fonctions d'allocation: `malloc()` et `realloc()`. ✗
- ☐ b. `realloc()` permet de changer la taille d'un tableau, en le gardant au même emplacement mémoire.
- ☐ c. on doit préciser à `malloc()` le type des éléments et leur nombre.
- ☒ d. on doit préciser à `malloc()` le nombre d'octets à allouer. ✓

Votre réponse est incorrecte.

La réponse correcte est : on doit préciser à `malloc()` le nombre d'octets à allouer.

Lorsqu'on simule un tableau 3D d'`int` de taille 2x3x4 par un tableau 1D `t` alloué en un seul bloc, la case `t[1][2][3]` est en fait `t[indice]` avec `indice` qui vaut...

Réponse : ✓

$1 \times 12 + 2 \times 4 + 3$ (ou $2 \times 3 \times 4 - 1$)

La réponse correcte est : 23

Lorsqu'on simule un tableau 3D d'int de taille 5x2x4 par un tableau 1D t alloué en un seul bloc, l'allocation dynamique peut être réalisée par...

- ☐ a. `t = malloc(4 * sizeof(int[10]));`
- ☐ b. `t = malloc(40 * sizeof(int));`
- ☒ c. `t = malloc(5*2*4 * sizeof(int));` ✓
- ☒ d. `t = malloc(sizeof(int[5][2][4]));` ✓

La notion d'objet en C...

- ☒ a. est relative à la quantité de mémoire des variables et à leur représentation. ✓
- ☐ b. permet de faire de la programmation orientée objet.
- ☐ c. permet de relier données et fonctions associées.
- ☐ d. est ce qui a permis d'évoluer vers le C++.

Un tableau de longueur variable...

- ☐ a. a besoin d'un appel à `free()` pour être libéré.
- ☐ b. est un tableau dont on peut changer la taille durant l'exécution.
- ☒ c. est un tableau dont la taille est renseignée par une variable * plutôt que par une expression constante. ✓
* variable ou expression calculée à partir d'une ou plusieurs variables
- ☒ d. est alloué sans faire appel à `malloc()` ou `calloc()`. ✓