Le C, plus proche de la machine

4 décembre 2023

Plan

Administratif

Présentation et utilisation de base du C

Les entiers

Gestion de la mémoire

Structures de données et types construits en C

Autres types en C

Fichiers, entrées, et sorties

Administratif

Résultats

Moyenne	Q1	Médiane	Q3	Max
13, 1	10,7	13,3	16,3	18,8

Le sujet a été assez bien réussi, mais beaucoup de personnes ont encore des difficultés avec la manière dont fonctionne OCaml.

Résultats

Moyenne	Q1	Médiane	Q3	Max
13, 1	10,7	13,3	16,3	18,8

Le sujet a été assez bien réussi, mais beaucoup de personnes ont encore des difficultés avec la manière dont fonctionne OCaml.
Pensez à recompter vos points.

Remarques générale

- Pas assez de phrases de conclusions. Il faut souligner quelque part pour indiquer où est la raison d'avoir des points. Cela fait que vous répondez parfois à côté de la question.

Remarques générale

- Pas assez de phrases de conclusions. Il faut souligner quelque part pour indiquer où est la raison d'avoir des points. Cela fait que vous répondez parfois à côté de la question.
- Il faut des explications sur vos codes.

Remarques générale

- Pas assez de phrases de conclusions. Il faut souligner quelque part pour indiquer où est la raison d'avoir des points. Cela fait que vous répondez parfois à côté de la question.
- Il faut des explications sur vos codes.
- On préfère le français quand le sujet est en français.

Remarques générale

- Pas assez de phrases de conclusions. Il faut souligner quelque part pour indiquer où est la raison d'avoir des points. Cela fait que vous répondez parfois à côté de la question.
- Il faut des explications sur vos codes.
- On préfère le français quand le sujet est en français.
- Beaucoup d'entre vous arrivent à faire des codes courts mais perdent tous leurs moyens quand il s'agit de produire quelque chose de plus de 5 lignes : il faut aussi savoir mettre les mains dans le cambouis,

Remarques générale

- Pas assez de phrases de conclusions. Il faut souligner quelque part pour indiquer où est la raison d'avoir des points. Cela fait que vous répondez parfois à côté de la question.
- Il faut des explications sur vos codes.
- On préfère le français quand le sujet est en français.
- Beaucoup d'entre vous arrivent à faire des codes courts mais perdent tous leurs moyens quand il s'agit de produire quelque chose de plus de 5 lignes : il faut aussi savoir mettre les mains dans le cambouis,
- Par pitié, pas de « Il est évident que...», « C'est logique que ... », « Banalement, on trouve que ...» ou ce genre de phrases qui me font vouloir vous retirer plus de points que nécessaires quand vous vous plantez invariablement juste après.

Remarques générales

Si vous montrez $\mathcal{P}(n+1)$ sans $\mathcal{P}(n)$ dans votre hérédité, c'est probablement que vous avez vu une récurrence là où ça n'était pas nécessaire.

Remarques générales

Si vous montrez $\mathcal{P}(n+1)$ sans $\mathcal{P}(n)$ dans votre hérédité, c'est probablement que vous avez vu une récurrence là où ça n'était pas nécessaire.
Ne passez pas trop de temps à rédiger les petits exemples du début, sauf si ça vous aide vous-même à comprendre ce qui se passe.

Remarques générales

Si vous montrez $\mathcal{P}(n+1)$ sans $\mathcal{P}(n)$ dans votre hérédité, c'est probablement que vous avez vu une récurrence là où ça n'était pas nécessaire.
Ne passez pas trop de temps à rédiger les petits exemples du début, sauf si ça vous aide vous-même à comprendre ce qui se passe.
Attention : un exemple ne suffit pas pour une démonstration. Vous avez la possibilité d'illustrer votre propos, mais il faut rester le plus général possible.

Mutabilité

Il y a encore de nombreuses personnes qui ne savent pas qu'une liste est persistante en OCaml (on ne peut pas les modifier en faisant p :: q), et qui ne font pas la distinction entre les variables persistantes et les références.

Mutabilité

Il y a encore de nombreuses personnes qui ne savent pas qu'une liste est persistante en OCaml (on ne peut pas les modifier en faisant p::q), et qui ne font pas la distinction entre les variables persistantes et les références.

```
let rec f p acc = match f with
|[] -> acc
|p::q -> p::acc ; f q acc
```

Mutabilité

Il y a encore de nombreuses personnes qui ne savent pas qu'une liste est persistante en OCaml (on ne peut pas les modifier en faisant p::q), et qui ne font pas la distinction entre les variables persistantes et les références.

```
let rec f p acc = match f with
|[] -> acc
|p::q -> p::acc ; f q acc
```

```
let rec f p acc = match f with
|[] -> acc
|p::q -> f q (p::acc)
```

Question 7

Peu ont pensé à voir qu'il fallait prouver une équivalence.
Pour montrer $A \Leftrightarrow B \wedge C$, il faut montrer les choses suivantes :

- $A \Rightarrow B$;
- $A \Rightarrow C$;
- $B \wedge C \Rightarrow A$ (qu'on peut montrer avec la contraposée $\neg A \Rightarrow B \vee C$).

Question 27

```
let ou x y =
    renvoyer_nom (
        let val_x = evaluer x in
        if val_x
            then true
            else let val_y = evaluer x in
                 val_x || val_y )
```

Question 27

```
let ou x y =
    renvoyer_nom (
        let val_x = evaluer x in
        if val_x
            then true
            else let val_y = evaluer x in
                 val_x || val_y )
```

On peut simplifier avec

```
let ou x y = renvoyer_nom (
    let val_x = evaluer x in
    if val_x
        then true
        else evaluer y)
```

Question 27

```
let ou x y =
    renvoyer_nom (
        let val_x = evaluer x in
        if val_x
            then true
            else let val_y = evaluer x in
                 val_x || val_y )
```

On peut simplifier avec

```
let ou x y = renvoyer_nom (
    let val_x = evaluer x in
    if val_x
        then true
        else evaluer y)
```

Il s'avère que le code suivant marche :

```
let ou x y = renvoyer_nom (evaluer x || evaluer y)
```

Question 27

J'ai vu le code suivant :

```
let ou x y =
    if evaluer x
        then
            renvoyer
        true else y
```

Il s'avère que le souci est qu'on calcule de toute manière x, mais il y a bien la bonne manière de procéder pour y.

Question 10

Au tableau.

Le Poly enfin !

Nous allons désormais travailler avec un polycopié de cours. Le polycopié de cours va devenir le principal outil de travail.
Il y a quelques blancs dans le polycopié qui seront complétés lors des séances. Par ailleurs les corrections des exercices qui sont dans le cours, et des démonstrations des propositions de cours seront proposées, mais il n'y a pas d'espace pour les faire figurer dans le poly.

Informations sur la classe mobile

L'accès à la classe mobile a effectivement été retardé par les soucis de réseau qui ont été la priorité.
Nous devrions y avoir accès la semaine prochaine, mais il ne faut toujours pas avoir trop d'espoirs.

Présentation et utilisation de base du C

Une brève histoire du C

Le C est un langage apparu au début des années 72 dont le principal objectif était de servir pour développer des outils pour UNIX. Différentes standardisations successives ont élargis et le langage depuis. En particulier un standard, le C99 publié en 1999, sera le standard que nous utiliserons et qui est au programme.

Caractéristiques du C

Le C est un langage de *bas niveau*, il permet d'avoir un contrôle élevé sur les ressources machines et est relativement proche de l'assembleur.
Le C est un langage compilé typé statiquement.

Applications du C

- Programmation système (utilisé massivement dans Linux) ;
- Utilisation dans des langages de plus haut niveau pour servir d'intermédiaire avec la machine (OCaml, CPython, Cython) ;
- En informatique embarquée ;
- Dans les compétitions d'informatique quand il faut un langage performant ;

Applications du C

- Programmation système (utilisé massivement dans Linux) ;
- Utilisation dans des langages de plus haut niveau pour servir d'intermédiaire avec la machine (OCaml, CPython, Cython) ;
- En informatique embarquée ;
- Dans les compétitions d'informatique quand il faut un langage performant ;
- Et de manière pédagogique pour comprendre la manipulation des ressources systèmes.

Premier programme en C

```c
#include <stdio.h>
int main() {
    printf("Bonjour au monde !");
    return 0;
}
```

Processus de compilation

Définition 1 : Compilation

La **compilation** est le processus de transformation d'un programme d'un langage vers un autre, usuellement de plus bas niveau, c'est-à-dire plus proche de la machine et donc plus facile à exécuter.
Le logiciel qui réalise la compilation est appelé **compilateur**.

Pour compiler en C, nous utiliserons gcc.

Intepréteur C en ligne

Nous n'aurons pas accès à un compilateur C au début, nous pourrons utiliser des interpréteurs C en lignes.

Entête

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

Exemple 1 : Entêtes pour l'écrit

À l'écrit, on suppose toujours travailler sous les hypothèses que les entêtes suivants :

- assert.h : principalement la macro assert qui permet de faire des assertions à l'exécution du code ;
- stdbool.h : les définitions qui permettent de définir et manipuler les booléens ;
- stddef.h : types size_t et ptrdiff_t qui servent pour la manipulation mémoire ;
- stdint.h : différents types d'entiers (signés et non signés, taille en place mémoire...) ;
- stdio.h : des fonctions liées aux entrées et aux sorties de programme ;
- stdlib.h : la bibliothèque standard avec beaucoup de fonctions utiles dont malloc et free.

Fonction main

```
1 int main () {
2     ...
3 }
```

Une première instruction

```
1 printf("Bonjour au monde !");
```

Une première instruction

```
1 printf("Bonjour au monde !");
```

Définition 2 : Expression

Une **expression** est un morceau de code qui est réduit à une valeur lors de l'exécution du programme.
Une expression est *évaluée* lorsque la valeur est calculée.

Une première instruction

```c
printf("Bonjour au monde !");
```

Définition 2 : Expression

Une **expression** est un morceau de code qui est réduit à une valeur lors de l'exécution du programme.
Une expression est *évaluée* lorsque la valeur est calculée.

Définition 3 : Instruction

Une **instruction** est un morceau de code qui forme une commande, un ordre à effectuer qui modifie l'état actuel de la machine.
Une instruction est *exécutée* lorsque cette ordre est réalisé.

Retour de la fonction main

```
1 return 0;
```

Déclaration

Définition 4 : Déclaration

La **déclaration** est la spécification du nom, du type d'une variable, ainsi que d'autres propriétés selon les cas. Il s'agit d'une information pour le compilateur.

```
1 int a;
```

Définition

Définition 5 : Définition

La **définition** d'une variable est l'affectation initiale d'une variable. On parle parfois d' *initialisation*.

```
1 a = 1;
```

Définition

Définition 5 : Définition

La **définition** d'une variable est l'affectation initiale d'une variable. On parle parfois d' *initialisation*.

```
1 a = 1;
```

C'est aussi la manière dont affecte une valeur à une variable.

Définition

Définition 5 : Définition

La **définition** d'une variable est l'affectation initiale d'une variable. On parle parfois d' *initialisation*.

```
1 a = 1;
```

C'est aussi la manière dont affecte une valeur à une variable.
On peut définir une variable lors de la déclaration.

```
1 int a = 1;
```

Constante

On peut définir une constante qui ne pourra pas être modifiée dans la suite :

```
1 const int a = 1;
```

Exemples

Exemple 2 : Déclaration et définition

- `int i, j;` déclare deux variables `i` et `j` sans les définir ;
- `const int a = 0;` déclare un constante entière égale à 0 ;
- `float a = 0.;` déclare un flottant dont la valeur est 0 initialement.

Portée

Définition 6 : Portée

La **portée** d'une variable est l'étendue du programme au sein de laquelle cette variable est accessible.

Portée

Définition 6 : Portée

La **portée** d'une variable est l'étendue du programme au sein de laquelle cette variable est accessible.

Définition 7 : Variable globale

Une variable est **globale** quand elle est déclarée à l'extérieur de tout bloc ou fonction du programme.
Potentiellement, la portée de cette variable peut-être tous le programme.

Exemple

```
1  const int a = 0;
2  const int b = 0;
3
4  int f(int b){}
5
6  int main(){
7      {
8          int a = 1;
9      }
10     return a;
11 }
```

Exercice

Exercice 1. Quelle est la sortie du code suivant ?

```c
#include <stdio.h>

const int a = 4;
const int b = 0;

int f(int b){
    printf("%d\n", b);
    {
        int b = 2;
        return b;
    }
}

int main() {
    {
        int a = 1;
        a = f(3);
    }
    return a;
}
```

Solution

Affiche 3, et renvoie la valeur de retour 4.

Booléens

Définition 8 : Booléen

Les **booléens** sont un type de variable qui peut prendre deux valeurs, soit Vrai, soit Faux.

Booléens en C

Pour manipuler des booléens en C, il est nécessaire d'utiliser une bibliothèque spécifique :

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

Cela nous donne accès au type `bool` et aux constantes `true` et `false`.

Opérations de comparaisons en C

Exemple 3 : Opération de comparaisons en C

Il existe plusieurs opérations de comparaisons en C.

- a==b vérifie si a est égal à b ;
- a!=b vérifie si a est différent à b ;
- a>b vérifie si a est supérieur strictement à b ;
- a>=b vérifie si a est supérieur ou égal à b ;
- a<b vérifie si a est inférieur strictement à b ;
- a<=b vérifie si a est inférieur ou égal à b.

OU logique

Définition 9 : OU logique

Le OU logique est une opération à deux opérandes dont la valeur est Vrai si et seulement si au moins l'une de ses opérandes l'est.

A	B	A OU B
V	V	V
V	F	V
F	V	V
F	F	F

En C, le OU logique est réalisé par le symbole `||`.

ET logique

Définition 10 : ET logique

Le ET logique est une opération à deux opérandes dont la valeur est Vrai si et seulement si ses deux opérandes le sont.

A	B	A ET B
V	V	V
V	F	F
F	V	F
F	F	F

En C, le ET logique est réalisé par le symbole &&.

Négation

Définition 11 : Négation

Le NON logique est une opération à une opérande dont la valeur est différente.

A	NON A
V	F
F	V

En C, le NON logique est réalisé par le symbole `!`.

Exercice

Exercice 2. Définition du OU exclusif Le OU exclusif est une opération binaire qui vérifie si exactement l'une de ses opérandes vaut Vrai.

A	B	A XOR B
V	V	F
V	F	V
F	V	V
F	F	F

Comment implémenter le XOR à partir des autres opérations logiques ?
Cela peut nous permettre d'introduire une fonction xor :

```
1 bool xor(bool a, bool b) {
2     /* Votre Code */
3 }
```

Exemple

Exemple 4 : Utilisation du OU exclusif

L'utilisation du OU exclusif permet de simplifier quelques expressions logiques.
Ainsi, on peut définir deux fonctions qui calculent la sortie et la retenue d'une addition sur deux bits avec une retenue à l'aide des fonctions suivantes :

```
1 bool sortie(bool a, bool b, bool r)
2 {
3     return xor(a, xor(b, r));
4 }
5 bool retenue(bool a, bool b, bool r)
6 {
7     return (a && b)||(r &&(b || c));
8 }
```

Paresse

```
1 bool f(){
2     printf("Cette fonction s'execute");
3     return true;
4 }
5 ...
6 a = true || f();
7 ...
```

Structure conditionnelle simple

```
1 if (cond) {
2     // Code a executer si la condition cond est
      verifiee
3 }
```

Structure conditionnelle simple

```
if (cond) {
    // Code a executer si la condition cond est
    verifiee
}
```

```
if (cond) {
    // Code a executer si la condition cond est
    verifiee
}
else {
    // Code a executer sinon
}
```

Exemple

```c
#include <stdio.h>
int main()
{
    int date;
    printf("En quelle annee a eu lieu la revolution
    francaise ?\n");
    scanf("%d", &date);
    if (date==1789)
    {
        printf("C'est vrai !");
    }
    else
    {
        printf("C'est faux !");
    }
}
```

Quelques remarques de syntaxe

```
1 if  i == 0
2 {
3       ...
4 }
```

Quelques remarques de syntaxe

```
if i == 0
{
    ...
}
```

```
if (cond)
    printf("Bim");
else
    printf("Bam");
```

Sinon pendant

```
1 if (cond1)
2 if (cond2)
3     {...}
4 else
5     {...}
```

Sinon pendant

```
if (cond1)
if (cond2)
    {...}
else
    {...}
```

```
if (cond1){
    if (cond2)
        {...}
    else
        {...}
}
```

Boucle while

```
while (cond)
    {Code a executer tant que cond est vrai}
```

Boucle while

```
while (cond)
    {Code a executer tant que cond est vrai}
```

Exemple 5 : Exemple de boucle

Voici un code qui affiche tous les entiers de 0 à 99.

```
int i = 0;
while (i<100){
    printf("%d\n", i);
    i = i+1;
}
```

Boucle for

```c
int i = 0;
while (i<100){
    printf("%d\n", i);
}
```

Boucle for

```c
int i = 0;
while (i<100){
    printf("%d\n", i);
}
```

```c
for (init ; cond ; incr)
    {/* Corps de la boucle*/}
```

Exemple

Exemple 6 : Exemple de boucle for

```
1 for (int i = 0 ; i<100; i++){
2     printf("%d\n", i);
3 }
```

Souvent, dans une boucle for, ou bien de manière générale quand on veut incrémenter de 1 un nombre, on utilisera l'instruction `a++;` qui augmente la valeur contenue dans `a` de 1. De même, il existe les instructions de la forme `a--` qui diminue la valeur contenue dans `a` de 1.

Break et continue

Il existe deux instructions spéciales pour les boucles break et continue. break met fin à la boucle, et continue revient au début de la boucle.

Exercice

Exercice 3. Proposer un programme C qui affiche tous les nombres de 0 à 100 qui sont des carré parfait, c'est-à-dire les nombres de la forme n^2 où n est un entier.

Usage des booléens

Comme beaucoup de choses en C, les booléens sont en réalité des entiers. En fait, le type `bool` est le type `unsigned int`, `true` est égal à `1` et `false` est égal à `0`.
On essaye cependant de faire le possible pour distinguer les entiers des booléens dans la mesure du possible :

```
1 if (k) {...}
```

Usage des booléens

Comme beaucoup de choses en C, les booléens sont en réalité des entiers. En fait, le type `bool` est le type `unsigned int`, `true` est égal à `1` et `false` est égal à `0`.
On essaye cependant de faire le possible pour distinguer les entiers des booléens dans la mesure du possible :

```
1 if (k) {...}
```

```
1 if (k!=0) {...}
```

Fonctions

Définition 12 : Fonction

Une **fonction** est un élément syntaxique qui regroupe des instructions qui peuvent être exécutées d'un seul bloc.
Les fonctions peuvent avoir des **arguments** nécessaire à l'appel.
Le plus souvent, les fonctions ont une **valeur de retour** qui est le résultat de la fonction.
Lorsque que ces instructions sont exécutées, on parle d'un **appel** à la fonction.

Fonctions en C

On peut définir des fonctions en C à l'aide de la syntaxe suivante :

```
1 type_retour nom_fonction(type_argument1
      nom_argument1, type_argument2 nom_argument2, ...)
      {
2      //Corps de la fonction
3 }
```

Les arguments dans la déclaration et dans l'appel sont séparés par des virgules. L'ordre des arguments est important.
Par ailleurs il n'y a *pas* de point-virgule à la fin de la définition.

Fonctions en C

On peut définir des fonctions en C à l'aide de la syntaxe suivante :

```
type_retour nom_fonction(type_argument1
    nom_argument1, type_argument2 nom_argument2, ...)
    {
    //Corps de la fonction
}
```

Les arguments dans la déclaration et dans l'appel sont séparés par des virgules. L'ordre des arguments est important.
Par ailleurs il n'y a *pas* de point-virgule à la fin de la définition.
On peut réaliser un appel à une fonction à l'aide de la syntaxe suivante :

```
nom_fonction(argument1, argument,2, ...)
```

La syntaxe est proche de celle de la définition, mais on fera attention à ne pas repréciser les types des arguments.

Prototype

Définition 13 : Prototype

Le **prototype** d'une fonction sont les informations concernant le type de retour et des arguments de la fonction.
On parle parfois de *signature*. Par ailleurs, on rajoute souvent les noms des arguments dans le prototype.

Déclarer sans définir une fonction

Tout comme une variable, on peut déclarer une fonction sans la définir pour pouvoir la définir plus tard.

```
type_retour nom_fonction(type_argument1 nom_argument1, type_argument2 nom_argument2, ...);

/*...*/

type_retour nom_fonction(type_argument1 nom_argument1, type_argument2 nom_argument2, ...)
{
    /*...*/
}
```

Valeur de retour

On dispose du mot-clef return qui permet de renvoyer un résultat dans une fonction. Cela permet d'interrompre un calcul.

```
1 int racine_entiere(int n){
2     int k = 0;
3     while (true){
4         if (k*k >= n)
5             return k;
6         k++;
7     }
8 }
```

Exercice

Exercice 4. Proposer une fonction `bool est_premier(int n)` qui renvoie si un entier est premier.

Pas de points virgules partout

```
1 if () {
2     ...
3 };
```

Pas de points virgules partout

```
1 if () {
2     ...
3 };
```

```
1 int f () {
2     ...
3 };
```

Toutes les fonctions doivent être avant le main

```
1 int main()
2 {
3     int f() {}
4 }
5 int g() {}
```

Toutes les fonctions doivent être avant le main

```
1 int main()
2 {
3     int f() {}
4 }
5 int g() {}
```

```
1 int f() {}
2 int g() {}
3 int main()
4 {
5
6 }
```

On n'a pas besoin de la racine carrée

```
1  bool est_premier(int n){
2      if (i<2){
3          return false;
4      }
5      for (int i = 2; i<sqrt(n); i++){
6          if (n%i==0){
7              return false;
8          }
9      }
10     return true;
11 }
```

On n'a pas besoin de la racine carrée

```
1  bool est_premier(int n){
2      if (i<2){
3          return false;
4      }
5      for (int i = 2; i<sqrt(n); i++){
6          if (n%i==0){
7              return false;
8          }
9      }
10     return true;
11 }
```

```
1  bool est_premier(int n){
2      if (i<2){
3          return false;
4      }
5      for (int i = 2; i*i <n; i++){
6          if (n%i==0){
7              return false;
8          }
9      }
10     return true;
```

Pas de points virgules dans des endroits saugrenus

```
1 if (...) {...};
2 while (...) {...};
3 for (...) {...};
4 int f(...) {...};
```

Pas de points virgules dans des endroits saugrenus

```
1 if (...) {...};
2 while (...) {...};
3 for (...) {...};
4 int f(...) {...};
```

```
1
2 if (...) {...}
3 while (...) {...}
4 for (...) {...}
5 int f(...) {...}
```

Attention à votre type de retour

```
1 int f(){
2     int i;
3     ...
4     printf("%d\n", i);
5 }
```

Attention à votre type de retour

```
1 int f(){
2     int i;
3     ...
4     printf("%d\n", i);
5 }
```

```
1 int f(){
2     int i;
3     ...
4     return i;
5 }
```

Récursivité en C

Il est possible d'avoir des fonctions récursives en C. Il n'est pas nécessaire de mettre de mots-clefs particuliers :

```
1 int factorielle(int n)
2 {
3     if (i==0)
4     {
5         return 1;
6     }
7     return n * factorielle(n-1);
8 }
```

Récursivité en C

Il est possible d'avoir des fonctions récursives en C. Il n'est pas nécessaire de mettre de mots-clefs particuliers :

```
1 int factorielle(int n)
2 {
3     if (i==0)
4     {
5         return 1;
6     }
7     return n * factorielle(n-1);
8 }
```

Il n'y a pas toujours d'optimisation des fonctions récursive terminale en C (même si gcc fait parfois de la magie). Dans la mesure du possible, il est préférable d'éviter les fonctions récursives en C.

Type void

On peut utiliser le type void pour définir le type de retour de fonctions qui ne renvoient rien.

```
1 void affichier_entiers_inferieurs(int n){
2     for (int i = 0; i<n; i++)
3     {
4         printf("%d\n", i);
5     }
6 }
```

Exercice

Exercice 5. Proposer une fonction à 3 arguments qui renvoie le plus grand des entiers en argument. La fonction aura le prototype `int plus_grand(int a, int b, int c)`.

Les entiers

Décomposition en base 2

Proposition 1 : Décomposition en base deux

Pour tout $n \in \mathbb{N}*$, il existe un unique $N \in \mathbb{N}$, et un unique $N+1$-uplet d'entiers a_0, a_1, ..., a_N tels que :

$$\forall k \leqslant N, a_j \in \{0,1\}$$

$$a_k = 1$$

$$\sum_{k=0}^{N} a_k 2^k = n$$

On écrit alors :

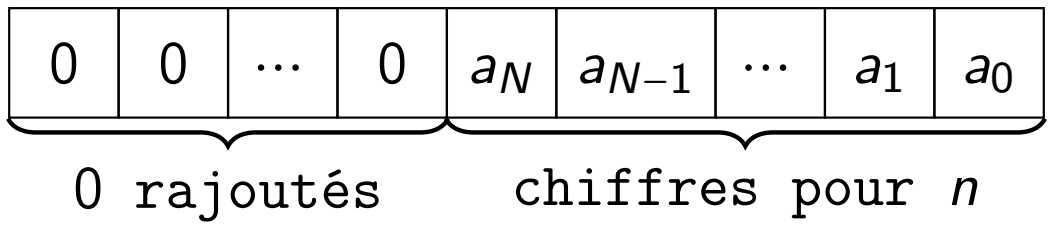
$$n = \overline{a_N a_{N-1} \ldots a_1 a_0}^2$$

Bit de poid fort, bit de poid faible

Définition 14 : Bit de poid fort, bit de poid faible

On parle de **bit de poid fort** pour le bit qui correspond au coefficient devant la plus grande puissance de 2.
On parle de **bit de poid faible** pour le bit qui correspond au coefficient devant la plus faible puissance de 2.

Exemple



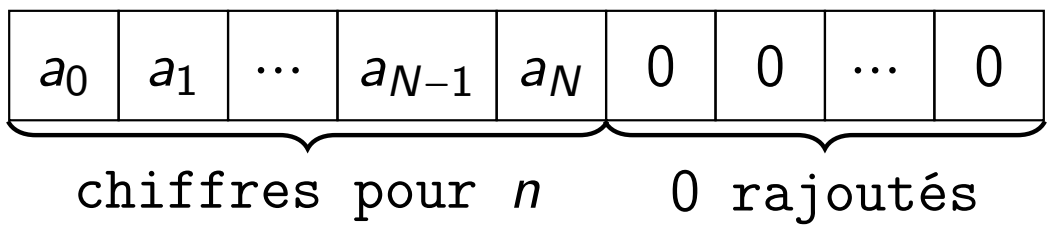
Boutisme

Définition 15 : Boutisme

On parle de **boutisme** pour décrire l'ordre dans lequel les bits sont représentés.
On parle de **grand-boutisme** quand on représente les bits de poids fort en premier.
On parle de **petit-boutisme** quand on représente les bits de poids faible en premier.

Exemple

Ainsi, avec l'orientation petit-boutiste, on obtient la représentation suivante :



Calcul de la représentation binaire d'un entier

Algorithme 1 : Calcul de la représentation binaire d'un entier

Pour calculer la représentation binaire d'un nombre, on peut utiliser l'algorithme suivant :

Entrée : n un entier, N le nombre de bit

Sortie : t le tableau représentant en binaire le nombre

$t \leftarrow$ un tableau de taille N rempli de 0

Pour Chaque $0 \leqslant i \leqslant N-1$ **Faire**

$q, r \leftarrow$ les entiers q et r le quotient et le reste de n par 2

$t[N-1-i] \leftarrow r$

$n \leftarrow q$

Renvoyer t

On a bien utilisé l'ordre grand-boutiste.

Calcul de la représentation binaire d'un entier

Algorithme 1 : Calcul de la représentation binaire d'un entier

Pour calculer la représentation binaire d'un nombre, on peut utiliser l'algorithme suivant :
Entrée : n un entier, N le nombre de bit
Sortie : t le tableau représentant en binaire le nombre

$t \leftarrow$ un tableau de taille N rempli de 0
Pour Chaque $0 \leqslant i \leqslant N-1$ **Faire**
    $q, r \leftarrow$ les entiers q et r le quotient et le reste de n par 2
    $t[N-1-i] \leftarrow r$
    $n \leftarrow q$
Renvoyer t

On a bien utilisé l'ordre grand-boutiste.

Exercice 6. Comment représenter l'entier 75 sur 8 bits ?

Exercice

Exercice 7. Que représente l'écriture binaire suivante ?

0	1	1	0	0	1	1	0

Gamme des entiers représentables

Proposition 2 : Gamme des entiers représentables

Sur N bits, on peut représenter les entiers de 0 à $2^N - 1$.

Entiers positifs en C

Exemple 7 : Entiers positifs en C

En C, les entiers ne sont pas nécessairement positifs en C, et il faut donc utiliser différents types.
Par défaut, on dispose du type `unsigned int` qui représente un entier de même nombre de bit que `int` mais qui est toujours positif. Le nombre de bits précis n'est pas une contrainte du langage, mais une contrainte de la machine.
Par ailleurs, l'entête <stdint.h> rajoute les types suivants :

- uint8_t : entiers positifs sur 8 bits ;
- uint16_t : entiers positifs sur 16 bits ;
- uint32_t : entiers positifs sur 32 bits ;
- uint64_t : entiers positifs sur 64 bits sur les machines qui le supporte.

Nombre en complément à deux

Définition 16 : Nombres en complément à deux

La **représentation par complément à deux** d'un nombre s'obtient de la manière suivante :

- Si le nombre est positif, on utilise la représentation binaire ;
- si le nombre est strictement négatif, on utilise la représentation binaire de l'opposé, on inverse les 0 et les 1 dans cette représentation, et puis on ajoute 1 à cette représentation comme s'il s'agissait d'un entier naturel.

Exemple

Ainsi, pour représenter le nombre -99 sur 8 bits, on commence par représenter le nombre 99, ce qui nous donne la représentation suivante :

0	1	1	0	0	0	1	1

Exemple

Ainsi, pour représenter le nombre -99 sur 8 bits, on commence par représenter le nombre 99, ce qui nous donne la représentation suivante :

0	1	1	0	0	0	1	1

On inverse les 0 et les 1 :

1	0	0	1	1	1	0	0

Exemple

Ainsi, pour représenter le nombre -99 sur 8 bits, on commence par représenter le nombre 99, ce qui nous donne la représentation suivante :

0	1	1	0	0	0	1	1

On inverse les 0 et les 1 :

1	0	0	1	1	1	0	0

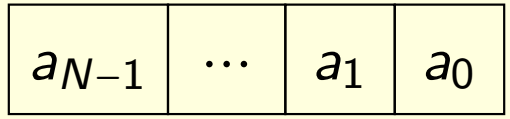
Enfin, on rajoute 1 en incrémentant le résultat :

1	0	0	1	1	1	0	1

Une définition bien pratique

Proposition 3

En complément à deux, un nombre représenté par les bits suivant :



est égal à :

$$-a_{N-1}2^{N-1} + \sum_{k=0}^{N-2} a_k 2^k$$

Réciproquement, si un nombre peut s'écrire de cette manière, alors sa représentation sera celle qui est donnée au dessus.

Gamme des entiers relatifs

Proposition 4 : Gamme des entiers relatifs sur N bits

Sur N bits, on peut représenter les entiers relatifs de -2^{N-1} à $2^{N-1}-1$ sans ambiguïté.

Entiers relatifs en C

Exemple 8 : Entiers relatifs en C

Le type `int` par défaut correspond à des entiers qui peuvent être à la fois positifs ou négatifs.
De la même manière que pour les positifs, l'entête <stdint.h> rajoute les types suivants :

- `int8_t` : entiers relatifs sur 8 bits ;
- `int16_t` : entiers relatifs sur 16 bits ;
- `int32_t` : entiers relatifs sur 32 bits ;
- `int64_t` : entiers relatifs sur 64 bits sur les machines qui le supporte.

Incrémenter sur les entiers naturels

Algorithme 2 : Incrémenter en base deux sur les entiers naturels

Pour incrémenter un nombre en base deux, on pars du bit de poids faibles, et on ajoute un en propageant l'éventuelle retenue.

Entrée : La représentation t d'un entier n sur N bits

Sortie : La représentation de $n+1$ sur N bits.

$k \leftarrow N-1$
Tant Que $k \geqslant 0$ **Faire**
    Si $t[k] = 0$ **Alors**
        $t[k] \leftarrow 1$
        Renvoyer t
    Sinon
        $t[k] \leftarrow 0$
    $k \leftarrow k-1$
Renvoyer t

Exercice

Exercice 8. Comment réaliser l'addition de deux nombres naturels ?

Maintenant sur les entiers relatifs

Proposition 5 : Compatibilité des additions et des incrémentations sur la représentation des entiers naturels et relatifs

Pour ajouter n et m deux entiers relatifs, on peut considérer leur représentation comme une représentation d'entiers positifs, faire l'addition de ces entiers avec les algorithmes précédent, et considérer le résultat en tant qu'entier relatif.

Opérations sur les entiers

Exemple * : Opérations sur les entiers en C

- Les opérateurs classiques $+$, $-$ et $*$;
- La division entière /, le modulo %.

Opérations de décalage logique

Définition 17 : Opérations de décalage logique

Un **décalage** est une opération qui décale à gauche ou à droite les bits d'une représentation binaire en complétant par des 0.

Opérations de décalage arithmétique

Définition 18 : Opérations de décalage arithmétique

Un **décalage** est une opération qui décale à gauche ou à droite les bits d'une représentation binaire en complétant par par des 0 dans le décalage à gauche, ou par un bit de signe dans le cas du décalage à droite.

Opérations de décalage en C

Exemple 9 : Opérations de décalage en C

Les opérations de décalage en C sont << et >>.

- a << k décale de k bits vers la gauche la représentation de a ;
- a >> k décale de k bits vers la droite la représentation de a.

Ces décalages sont arithmétiques : à la compilation, le compilateur se sers des informations de typage pour savoir comment compléter dans le cas d'un décalage à droite (0 dans le cas d'un nombre positif, 1 dans l'autre cas).

Opération bit à bit

Définition 19 : Opération bit à bit

Une **opération bit à bit** est une opération qui opère directement sur les bits de la mémoire.

Par exemple si on se donne une opération $\odot$ et deux nombres A et B dont les représentations sont les suivantes :

a_{N-1}	a_{N-2}	$\cdots$	a_1	a_0

b_{N-1}	b_{N-2}	$\cdots$	b_1	b_0

On obtient le résultat suivant :

$a_{N-1} \odot b_{N-1}$	$a_{N-2} \odot b_{N-2}$	$\cdots$	$a_1 \odot b_1$	$a_0 \odot b_0$

Bien sûr, dans le cas d'une opération unaire, on n'aura besoin que d'un seul opérande.

En C

Exemple 10 : Opérations bit à bit en C

On a les opérations bits à bits suivantes en C :

- | le ou bit à bit ;
- & le et bit à bit ;
- ~ la négation bit à bit ;
- ^ le ou exclusif bit à bit.

Exercice

Exercice 9. Combien vaut (234|125)&127 ?

Gestion de la mémoire

Mémoire dans un programme

Définition 20 : Organisation de la mémoire d'un programme

La mémoire dont dispose un programme est organisé en plusieurs zones :

- Le **texte** : correspond à l'espace mémoire où le programme est chargé en binaire ;
- La **pile** : qui sers à réaliser les différents appels de fonctions ;
- Le **tas** : qui sers à stocker les données dynamiquement indépendamment de la pile ;
- Le **segment de données** : qui contient les variables globales, statiques et certaines chaînes de caractère.

Allocation mémoire

Définition 21 : Allocation

L' **allocation mémoire** est le procédé qui permet de réserver des parties de la mémoire.

```c
# include <stdio.h>

void f(int n){
    n++;
    printf("n vaut %d dans la fonction f\n", n);
}

int main(){
    int n = 0
    f(n);
    printf("n vaut toujours %d dans la fonction main
    \n", n);
    return 0;
}
```

Passage par valeur

Définition 22 : Passage par valeur

On dit qu'il y a **passage par valeur** lorsque le code appelé dispose d'une copie de ses paramètres.

Adresse mémoire et Pointeur

Définition 23 : Adresse mémoire

Une **adresse mémoire** est un entier qui désigne une zone particulière de la mémoire.

Définition 24 : Pointeur

Un **pointeur** est une variable qui contient une adresse mémoire.

Utilisation des pointeurs en C

Exemple 11

On peut indiquer le type d'un pointeur à l'aide du symbole * dans le type :

```
1 int *aptr; // aptr est un pointeur vers un
    entier
```

On peut accéder à l'adresse d'une variable à l'aide du symbole & :

```
1 int a = 0;
2     aptr = &a;// aptr contient l'adresse vers
    la variable a
```

Utilisation des pointeurs en C

Exemple *

Par ailleurs, on peut utiliser le symbole $*$ pour modifier valeur contenue dans un pointeur selon le contexte :

```
1 *aptr = 1; // On mets 1 dans la case memoire
      pointee par aptr
```

On peut aussi s'en servir pour accéder à la valeur contenue dans un pointeur.

```
1 printf("%d\n", *aptr); // On affiche la valeur
      contenue dans l'adresse aptr
```

On dit alors que $*$ est un opérateur de *déréférencement*.

Exemple

Exemple 12 : Exemple d'utilisation des pointeurs

```c
#include <stdio.h>

void f(int *n){
    *n++;
    printf("n vaut %d dans la fonction f\n", *n
    );
}

int main(){
    int n = 0;
    f(&n);
    printf("n vaut maintenant %d dans la
    fonction main\n", n);
    return 0;
}
```

Pointeur NULL

Exemple 13 : Pointeur NULL en C

Le pointeur NULL est défini dans plusieurs entêtes classiques : stddef.h, stdio.h, stdlib.h et string.h.

```
1 int *a = NULL;
```

Exercice

Exercice 10. Proposer une fonction de prototype `void echanger(int *a, int *b)` qui échange la valeur contenue dans deux pointeurs, et proposer un programme qui illustre ce code :

```
1  # include <stdio.h
2  void echanger(int *aptr, int *bptr){
3      ...
4  }
5
6  int main(){
7      int a = 87;
8      int b = 65;
9
10     printf("a vaut %d et b vaut %d", a, b);
11     echanger(...);
12     printf("a vaut %d et b vaut %d", a, b);
13 }
```

Définition 25 : Bloc d'activation d'un appel

Un **bloc d'activation** d'une fonction est un segment de donnée qui se trouve dans la pile et qui est découpé de la manière suivante :

- Paramètres de la fonction : les paramètres de la fonctions ;
- Adresse de retour : l'adresse qui pointe vers un élément du texte pour savoir où revenir dans le code après l'exécution de la fonction ;
- Variables locale à la fonction : les différentes variables locales qui sont calculées et modifiés dans la pile.

Exemple 14 : Exemple de fonctions imbriquées

```
1  int f(int a, int b){
2      int c = 2;
3      return 0;
4  }
5
6  int main(){
7      int d = 0;
8      int e = 1;
9      f(d, e);
10     return 0;
11 }
```

Durée de vie

Définition 26 : Durée de vie d'une variable

La **durée de vie** d'une variable est la période temporelle durant laquelle elle est allouée dans la mémoire.

Exemple

```
1 int x;
2 {
3     int x;
4     /* Ici, nous ne sommes plus dans la portee de la
       premiere variable x
5     mais celle-ci est encore en vie. */
6 }
```

Problème

```c
int *f(){
    /* Cree une variable entiere et en renvoie l'
    adresse */
    int a = 0;
    return &a;
}

int main(){
    int *b;
    b = f();
    // L'adresse contenue dans b n'est pas valide :
    il s'agissait d'une variable locale a la fonction
     f
    printf("%d\n", *b);
    return 0;
}
```

Définition 27 : Séquence d'utilisation de l'allocation dynamique de la mémoire sur le tas

1. Le programme demande au système d'exploitation de lui attribuer un espace mémoire ;
2. Si possible, le système d'exploitation renvoie l'adresse de début d'un bloc de cette taille là et la réserve ;
3. Le programme utilise l'espace alloué ;
4. Le programme informe le système d'exploitation que l'espace mémoire peut être libéré ;
5. Le système d'exploitation libére cet espace mémoire.

Taille d'une donnée

Définition 28 : Taille d'une donnée

La **taille d'une donnée** est la quantité de mémoire que la donnée occupe.

Exemple 15 : Obtenir la taille d'une donnée en C

En C, la fonction sizeof donne la taille en nombre d'octet du type passé en argument.
Par exemple :

```
1 sizeof(int8) (* 1 *)
```

Cela nous permet de connaître le nombre d'octet utilisé par un entier par défaut. Cette valeur peut dépendre du support, mais en pratique, il est souvent de 4.

```
1 sizeof(int) (* 4 *)
```

Définition 29 : Conversion

Une **conversion de type** est la transformation d'une donnée d'un type pour qu'elle devienne une donnée d'un autre type.

Définition 29 : Conversion

Une **conversion de type** est la transformation d'une donnée d'un type pour qu'elle devienne une donnée d'un autre type.

Exemple 16 : Conversion en C

Pour convertir en C, on utilise le type entre parenthèse devant l'objet à convertir :

```
1 int a = -1;
2 unsigned int b;
3 b = (unsigned int) a;
```

Malloc et Free

Exemple 17 : Fonctions malloc et free en C

La fonction malloc prend en argument une taille, et renvoie soit un pointeur vers le début d'une zone de la mémoire du tas qui devient réservée au moment de l'appel, soit le pointeur NULL si l'allocation est un échec.
La fonction free prend en argument un pointeur vers le début d'une zone réservée du tas et la libère.

Forme générale

Exemple 18 : Utilisation de malloc et free

```c
#include <stdlib.h>

int main(){
    int * ptr;

    ptr = (int *) (malloc(sizeof(int)));
    if (ptr==NULL){
        return 1;
    }
    /* ... */

    free(ptr);
    return 0;
}
```

Un entiers qu'on manipule sur le tas

Exemple 19 : Exemple d'utilisation

```
1  int *allouer_entier(){
2      int * ptr;
3
4      ptr = (int *) (malloc(sizeof(int)));
5      if (ptr==NULL){
6          abort();
7      }
8      return ptr
9  }
10 void incrementer_entier(int * ptr){
11     *ptr++;
12 }
13 void liberer_entier(int * ptr){
14     free ptr;
15 }
```

Structures de données et types construits en C

Structures de données

Définition 30 : Structure de donnée

Une **structure de donnée** est une manière d'organiser les données pour les traiter plus facilement.

Tableaux

Définition 31 : Tableau

Un **tableau** de données est une structure de données linéaire à taille fixe dans laquelle il est rapide d'accéder à un élément quelconque du tableau.

Exemple 20 : Tableaux en C

Pour définir un tableau, on doit en préciser la taille et le type des éléments à l'aide de la syntaxe suivante :

```
1 int tableau[10];
```

La taille entre crochet doit être un littéral (c'est-à-dire un nombre en toute lettre) ou une constante.
On peut ensuite accéder à un élément du tableau à l'aide de l'expression suivante :

```
1 tableau[k]
```

Enfin, on peut modifier un élément du tableau à l'aide de l'instruction suivante :

```
1 tableau[k] = nouveau;
```

Définition 32 : *Pointer Decay*

Quand nous manipulons le tableau en tant que valeur (par exemple en affectant le tableau à une variable, ou surtout en l'utilisant comme argument d'une fonction), celui-ci est transformé en pointeur vers son premier élément.
On appelle ce procédé ***Pointer Decay*** en anglais (que l'on pourrait traduire par *dégradation en pointeur*).
Les deux exceptions notables sont l'opérateur & et la fonction sizeof

Problème

```c
#include <stdio.h>

int main(){
    int t[10];
    printf("%d\n", t[10]); // Le compilateur accepte, mais le resultat est faux.
    return 0;
}
```

Solution

```c
void afficher_tout(int * t, int taille){
    for(int i = 0; i<taille; i++)
    {
        printf("%d\n", t[i]);
    }
}
```

Tableau sur le tas

Exemple 21 : Allocation d'un tableau sur le tas

Pour allouer un tableau sur le tas de taille n, nous devons allouer une taille n multiplié par la taille d'un élément.

```
1 int *t = (int*) malloc(n * sizeof(int));
```

Exercice

Exercice 11. Proposer une fonction de prototype `int * recopier(int * t, int n)` qui, à partir d'un tableau (et donc d'un pointeur vers son premier élément) ainsi que sa taille, alloue un tableau de même taille sur le tas et recopie les valeurs du premier tableaux dans le second.

Calloc

Exemple 22 : Fonction calloc

La fonction `calloc` prend en argument le nombre d'éléments à allouer, et la taille d'un élément, et renvoie un pointeur vers le premier élément d'une zone dans la mémoire initialisée à zéro réservée pour ces éléments.
Pour allouer un tableau de taille n d'entiers, il faut donc utiliser :

```
1 (int*) calloc(n, sizeof(int))
```

Un peu d'administratif

- Je ramasse les DM ;
- DS la semaine prochaine ;
- Pas de nouvelles des Ordinateurs sous Linux.

Programme du DS

- Représentation des entiers en mémoire ;
- Pointeurs, gestions de la mémoire ;
- Tableaux ;
- Structures en C, listes chaînées.

Type Produit

Un **type produit** de plusieurs types t_1, t_2, ..., t_n est un type dont chaque élément dispose de n composantes de types respectifs t_1, t_2, ..., t_n.

En C

Exemple 23 : Type structuré en C

Une **type structuré** en C est une implémentation d'un type produit. Un tel type est composé de **champs** qui ont chacun un type.
Nous pouvons en définir un avec la syntaxe suivante :

```
1 struct nom_de_la_struct {
2             type1 champs1 ;
3             type2 champs2 ;
4             ...
5             typen champsn;
6         };
```

Exemple

Exemple 24 : Un type structuré

```
1 struct couple {
2     int x;
3     int y;
4 };
```

Listes Chaînées

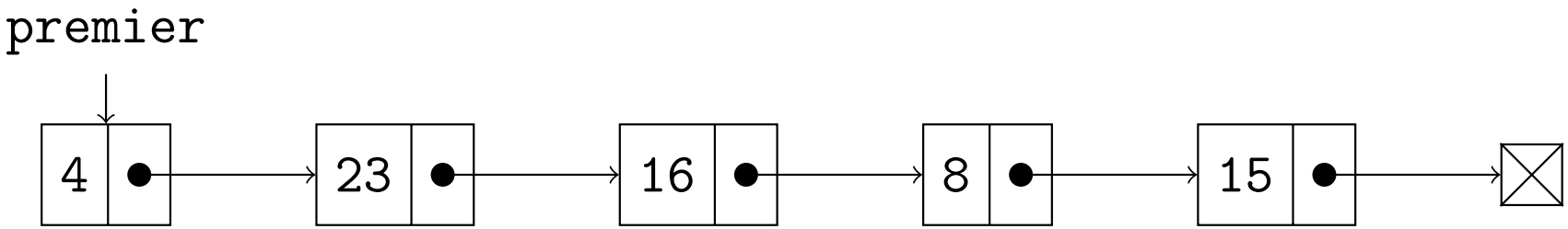
Définition 33 : Liste Chaînée

Une **liste chaînée** est une structure de données linéaire dont la représentation mémoire est une succession de **cellules** faites d'un contenu et d'un pointeur vers la cellule suivante si elle existe.

Listes Chaînées

Définition 33 : Liste Chaînée

Une **liste chaînée** est une structure de données linéaire dont la représentation mémoire est une succession de **cellules** faites d'un contenu et d'un pointeur vers la cellule suivante si elle existe.



Listes Chaînées en C

```c
typedef struct _cell {
    int val;
    struct _cell *suiv;
} cellule;
```

Listes Chaînées en C

```c
typedef struct _cell {
    int val;
    struct _cell *suiv;
} cellule;
```

```c
typedef cellule *chainee;
```

Listes Chaînées en C

```c
typedef struct _cell {
    int val;
    struct _cell *suiv;
} cellule;
```

```c
typedef cellule *chainee;
```

Comment construire une liste chaînée avec les éléments 16, 8 et 23 ?

Exemple

Exemple 25 : Calculer la longueur d'une liste chaînée

```
1  int longueur(chainee lc){
2      int reponse = 0;
3      cellule * actuel;
4      actuel = lc;
5      while (actuel != NULL){
6          reponse++;
7          actuel = (*actuel).suiv;
8      }
9      return reponse;
10 }
```

Un problème des listes chaînées

Proposition 6 : Temps d'accès dans une liste chaînée

Pour accéder à l'élément d'indice k, c'est-à-dire le $(k+1)$-ième élément en prenant les éléments dans l'ordre, il faut parcourir $k+1$ cellules.

Exercice

Exercice 12. Proposer une fonction de prototype `int indice(chainee lc, int n)` qui renvoie l'indice de l'élément de valeur `n` dans la liste chaînée `lc`.

Exemple

Exemple 26 : Ajouter une cellule en tête d'une liste chaînée

```
1  void ajouter_debut(chainee *lc, int valeur){
2      cellule *nouvelle;
3      nouvelle = (cellule *) (malloc(sizeof(
       cellule)));
4      if (nouvelle == NULL){
5          abort();
6      }
7      nouvelle->suiv = *lc;
8      nouvelle->val = valeur;
9      *lc = nouvelle;
10 }
```

Administratif

DS en fin de semaine, je ne sais pas si ils seront corrigés lundi prochain.

Rendu des DM

Moyenne	Écart type	Q1	Médiane	Q3	Max
16,15	-	15	16,7	18,3	20

Remarques générales

- Quelques soucis de présentations, mais le code reste lisible et bien présenté dans la plupart des copies ;
- Vous pourrez supposer que vous avez les modules classiques (mais pas <math.h>) ;
- J'attendais un programme complet dans la question 2.2, avec les entêtes, une fonction main (mais vous n'aurez pas de questions supplémentaires de ce type à l'écrit) ;
- Pas besoin de fonction main à l'écrit sauf quand cela est demandé explicitiment ;
- Ne me donnez pas votre sujet avec votre copie, je l'ai déjà.

Au sujet des preuves

- Beaucoup de soucis dans les démonstrations : on veut un argument clair et précis ;
- Un argument percutant souligné vaut souvent mieux que beaucoup de texte inutile : *« le nombre de zéro à droite dans la retenue augmente d'au moins 1 à chaque étape »*, *« $a + b$ est constant »*, *« l'addition dans les entiers naturelles est compatible avec l'addition des entiers en complément à deux »*...
- Vous ne pouvez pas vous passer de la partie preuve en informatique : réfléchir à pourquoi les choses fonctionnent est au cœur du programme. De manière générale, si vous comprenez comment un algo fonctionne, la preuve de terminaison et de correction est facile à donner.

Exercice

Exercice 13. Comment retirer une cellule à la fin d'une liste chaînée ? La fonction aura le prototype :

```
1 void retirer_fin(chainee * lc)
```

Proposition de correction

```
1  void retirer_dernier(chainee * plc){
2      cellule ** actuelle;
3      if(*plc==NULL)
4          abort();
5      actuelle = plc;
6      while((*actuelle)->suiv!=NULL){
7          actuelle = &((*actuelle)->suiv);
8      }
9      free((*actuelle));
10     *actuelle = NULL;
11 }
```

Autres types en C

Proposition 7 : Écriture d'un réel en base deux

Pour tout réel positif x, il existe N, ainsi qu'un $N+1$-uplet a_N, a_{N-1}, ..., a_0, et une suite $(d_k)_{k\geqslant 1}$ tels que :

$$\forall 0 \leqslant k \leqslant N,\ a_k \in \{0,1\}$$

$$\forall k \in \mathbb{N},\ d_k \in \{0,1\}$$

$$x = \sum_{k=0}^{N} a_k 2^k + \sum_{k=1}^{+\infty} d_k 2^{-k}$$

On dit que $\sum_{k=1}^{+\infty} d_k 2^{-k}$ est le **développement décimale en base 2 de** x.

Algorithme 3 : Développement décimal en binaire d'une partie décimale

Entrée : $0 \leqslant x \leqslant 1$ un nombre à représenter, N une précision voulue

Sortie : d_1, ..., d_N les chiffres du développement décimale de x.

resultat $\leftarrow$ liste vide

Pour Chaque k de 1 à N **Faire**

- $x \leftarrow x \times 2$
- $d_k \leftarrow$ la partie entière de x
- $x \leftarrow$ la partie décimale de x
- ajouter d_k à la fin de resultat

Renvoyer resultat

Algorithme 3 : Développement décimal en binaire d'une partie décimale

Entrée : $0 \leqslant x \leqslant 1$ un nombre à représenter, N une précision voulue

Sortie : $d_1, ..., d_N$ les chiffres du développement décimale de x.

resultat $\leftarrow$ liste vide

Pour Chaque k de 1 à N **Faire**

- $x \leftarrow x \times 2$
- $d_k \leftarrow$ la partie entière de x
- $x \leftarrow$ la partie décimale de x
- ajouter d_k à la fin de resultat

Renvoyer resultat

Exercice 14. Quelle est le développement décimale de 0.375 ? De $\frac{1}{3}$?

Nombre flottant

Définition 34 : Nombre à Virgule Flottante

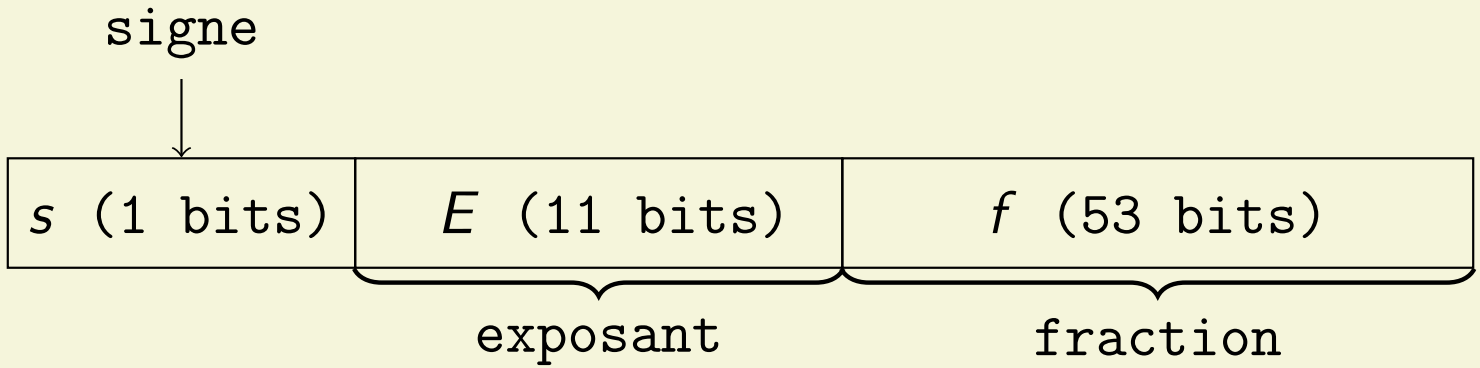
Un **nombre à virgule flottante**, ou souvent un **nombre flottant** et parfois un **flottant**, est un nombre x pour lequel il existe trois nombres :

- son signe s égal à -1 ou 1 ;
- sa mantisse $m \in \mathbb{N}$;
- son exposant $e \in \mathbb{Z}$

Et ce, avec :

$$x = s \times m \times 2^e$$

Exemple 27 : Valeur d'un flottant sur 64 bits dans la norme IEEE 754



E est un entier positif sur 11 bits qui correspond à l'exposant, f est la fraction sur 53 bits, et s est le signe du flottant, sur 1 bit.
Si $1 \leqslant E < 2^{11} - 1$, le nombre est dit normal et sa valeur est :

$$(-1)^s \times 2^{E-1023} \times 1.f$$

Si $E = 0$, le nombre est dit subnormal et sa valeur est

$$(-1)^s \times 2^{1-1023} \times 0.f$$

Un grande gamme de valeurs

Proposition 8 : Valeurs extrêmes

Le plus grand nombre que l'on peut représenter avec un tel flottant est :

$$2^{2^{11}-2-1023}1.1.....1 \approx 2^{1023}$$

Le plus petit nombre strictement positif que l'on peut représenter avec un tel flottant est :

$$2^{-1074}$$

Exemple 28 : Les flottants en C

On dispose des types float et double qui permettent de manipuler des flottants en C.
Le premier de ces types est un flottant sur 32 bits, tandis que le second est un flottant sur 64 bits.

Caractère

Définition 35 : Caractère

Le **caractère** est un type de donnée qui permet de représenter une lettre, un chiffre, un symbole, un blanc (espaces, tabulations, retours à la ligne...), ou une opération spéciale de contrôle.

En C

Exemple 29 : Caractères en C

En C, le type char encode un caractère sur un entier sur 8 bits en ASCII[a] .
Les caractères sont représentés par des guillemets simples.

a. https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_

Chaînes de Caractères

Définition 36 : Chaîne de caractère

La **chaîne de caractère** est un type qui correspond à une suite ordonnée de caractères.

Exemple 30 : Chaîne de caractères en C

En C, une chaîne de caractère est un tableau de caractères de sentinelle nulle, c'est-à-dire un tableau de caractères dont le dernier élément est un caractère spécial de valeur `'\0'`.
Les chaînes peuvent être définie à l'aide de guillemets droits doubles :

```
1 char chaine[] = "Bonjour !"
```

Cela est équivalent à initialiser le tableau (en n'oubliant pas la sentinelle) :

```
1 char chaine[] = {'B', 'o', 'n', 'j', 'o', 'u',
      'r', ' ', '!', '\0'}
```

Exemple 31 : Calcul de la longueur d'une chaîne de caractère

On nomme **longueur** d'une chaîne de caractère le nombre d'éléments dans cette chaîne de caractère. Par convention, en C, le caractère '\0' ne compte pas dans la longueur de la chaîne de caractère.
Pour trouver la longueur, on peut parcourir les éléments du tableau jusqu'à trouver le caractère nul.

```
1 int longueur(char * chaine){
2     int indice = 0;
3     while (chaine[indice]!='\0'){
4         indice++;
5     }
6     return indice;
7 }
```

Exercice

Exercice 15. Proposer une fonction de prototype `int compter(const char * chaine, char c)` qui renvoie le nombre d'occurence de `c` dans `chaine`.

Exemple 32 : Fonction printf

La fonction printf du module stdio.hpermet d'afficher une chaîne de caractère sur la sortie standard.

```
1 printf("Bonjour !");
```

On peut se servir de cette fonction pour afficher des entiers à l'aide de %d dans la chaîne de caractère et un entier en arguemnt :

```
1 int a = 2;
2 printf("L'entier a vaut %d", a);
```

On peut rajouter des arguments et utiliser des types différents à l'aide d'autres lettres après le symbole % et d'autres arguments :

```
1 printf("Voici un entier %d et voici un flottant
      %f", 2, 1.0);
```

(Les suites de caractères %... sont remplacés dans l'ordre par les arguments en plus du premier dans l'ordre)

Caractères	%d, %i	%u	%f, %F
Type	Entier signé	Entier non signé	Flottant de type double

Caractères	%c	%s
Type	Caractère	Chaîne de caractère

Caractères	%d, %i	%u	%f, %F
Type	Entier signé	Entier non signé	Flottant de type double

Caractères	%c	%s
Type	Caractère	Chaîne de caractère

Exercice 16. Proposer une fonction de prototype
`void afficher_espace(const char * chaine)` qui affiche les caractères de `chaine` en les séparant par des espaces.

Fonction scanf

Exemple 33 : Fonction scanf

La fonction scanf en C permet de récupérer des données depuis l'entrée standard, le plus souvent depuis une console.
Il faut lui spécifier un format attendu et des adresses pour les différentes valeurs attendus. Le format est similaire à celui de printf .

```c
int a;
scanf("%d", &a);
```

Fichiers, entrées, et sorties

Commandes linux

Exemple 34 : Quelques commandes utiles en shell UNIX

- `ls` permet d'afficher les répertoires et fichiers dans le répertoire courant ;
- `cd nom` permet de se rendre dans le répertoire `nom` ;
- `cd ..` permet de remonter au répertoire parent ;
- `mkdir nom` crée un répertoire `nom`.

Organisation des fichiers

Définition 37 : Racine

La **racine** est le répertoire dont tous les fichiers sont un descendant.

Organisation des fichiers

Définition 37 : Racine

La **racine** est le répertoire dont tous les fichiers sont un descendant.

Définition 38 : Chemin relatif, chemin absolu

Un **chemin absolu** est un chemin vers un fichier ou un répertoire depuis la racine du système.
Un **chemin relatif** est un chemin vers un fichier ou un répertoire depuis le répertoire courant.

Arguments de la fonction main

Exemple 35 : Arguments de la fonction main en C

On peut écrire la fonction main avec le prototype suivant :

```
1 int main(int argc, char *argv[])
```

agrc correspond au nombre d'arguments utilisés lors de l'appel du programme ;
argv est un tableau de chaînes de caractères dont dont chaque élément est un argument du programme.
Le nom du programme lui-même est le premier argument. Il y a donc toujours au moins une chaîne dans agrv et agrc est toujours au moins égal à 1.

Un programme somme

Exemple 36 : Un programme somme

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

int main(int argc, char *argv[]){
    int resultat = 0;
    for (int i = 1; i<argc; i++){
        resultat = resultat + atoi(argv[i]);
    }
    printf("%d", resultat);
    return 0;
}
```

Exercice

Exercice 17. Proposer un programme qui prend en argument des chaînes de caractères, et renvoie sur la sortie standard leurs tailles séparés par des espaces.

Rendu des DS

Moyenne	Écart type	Q1	Médiane	Q3	Max
9,77	4,17	6,7	8,8	12	19,8

DS plutôt raté dans l'ensemble. Il y a eu des soucis algorithmiques mais surtout des problèmes liés à une mauvaise compréhension du cours.
Il y avait 150 points au maximum, la réduction vers une note sur 20 a été fait sur une base de 125 points.

Remarques générales

- Toujours des erreurs de présentation, des codes sur plusieurs pages, des choses mal indentées ;
- Pareillement, il y a beaucoup de copies où rien n'est souligné ou encadré ;
- Manques d'explications dans les codes ;
- Des copies qui ne sont pas numérottées, des noms qui ne sont pas présents sur toutes les feuilles ;
- Réutilisez les fonctions de sorte à éviter de propager des erreurs recopiées ;
- Ne mettez jamais plusieurs solutions, vous choisissez la meilleur ou vous ne faite que celle-là, mais *bien*.

Des erreurs récurrentes en C

- `n>>k;` ne modifie pas n, c'est un calcul, vous devez faire `n = n>>k;` ;
- Vous ne pouvez PAS accéder aux bits d'un entiers n avec la syntaxe $n[k]$, c'est réservé au tableau ;
- Vous devez savoir faire des manipulations sur les représentation mémoire sans passer par des propriétés arithmétiques ;
- Vous de DEVEZ PAS accéder aux éléments d'un tableau avec la syntaxe d'arithmétique des pointeurs $*(t+k)$;
- Quelques soucis dans les conversions ;
- De nombreuses copies avec des erreurs de pointeurs de base ;
- Fermez vos fonctions ;
- Opérateurs ternaires ne sont pas toujours clairs.

Erreurs de compréhension dans le sujet

Il y a eu de nombreuses erreurs de compréhension dans le sujet :

- Le nombre de bloc reste le même après une allocation, le nombre de bloc augment de 1 exactement à chaque libération ;
- Confusion entre les blocs mémoires, et les cases mémoires ;
- Les blocs sont par ordre croissant d'adresse ;
- Les adresses de type `adresse_t` ne sont pas des adresses au sens classique en C et ne peuvent pas être manipulées comme telles.

Des erreurs algorithmiques

- Dans l'exercice 1, être plus grand en petit-boutiste ne correspond pas à être plus petit en grand-boutiste.
- Dans l'exercice 2, i_0 ne doit pas contenir le minimum du tableau,mais le minimum qui est utilisé au moment où vous mettez à jour j_0.
- Dans le problème, le cas où c'est le dernier entête qui est l'entête précédent n'est pas toujours traité correctement.

Erreurs dans les démonstrations

- Si on veut montrer qu'un programme est en $O(n^2)$, on peut montrer qu'on fait deux boucles imbriquées de $O(n)$ étapes, mais il faut aussi préciser que l'on ne fait que $O(1)$ opérations ;
- Quelques manques de rigueur dans les démonstrations qui n'ont pas été punis outre-mesure cette fois-ci.

Exemple d'utilisation d'un fichier

Exemple 37 : Écriture dans un fichier en C

```c
FILE * fptr;
fptr = fopen("nom_du_fichier", "w");
fprintf(fptr, "Bonjour !\n");
fprintf(fptr, "Bonjour sur une nouvelle ligne !
    ");
fclose(fptr);
```

Ici, l'option w indique que l'on désire écrire (Write en anglais) dans ce fichier. Si le fichier existe déjà, son contenu sera effacé pour laisser place à notre texte.
On peut utiliser l'option a (Append en anglais) pour ajouter à la fin d'un fichier au lieu de le remplacer.

Exemple de lecture dans un fichier

Exemple 38 : Lecture dans un fichier en C

```c
FILE * fptr;
char tampon[64];
fptr = fopen("nom_du_fichier", "r");
fscanf(fptr, "%s", tampon);
printf("Voici la premiere chaine que nous
    pouvons extraire du fichier : %s", tampon);
fclose(fptr);
```

Flux standard

Définition 39 : Des fichiers spéciaux

Le **flux de sortie standard** correspond aux données sortantes du programme qui sont souvent dirigées vers un terminal.
Le **flux d'entrée standard** correspond aux données entrantes du programme qui proviennent souvent des entrées fournies dans un terminal.
Le **flux d'erreur standard** correspond aux sorties liées aux erreurs qui sont souvent envoyés dans un terminal.

Redirection

Exemple 39 : Redirection

On peut rediriger l'entrée standard depuis un fichier à l'aide du symbole <.
On peut rediriger la sortie standard vers un fichier à l'aide des symboles > et >>.
On peut rediriger la sortie d'erreur standard vers un fichier à l'aides des symboles 2> et 2>>.

Exemple

Exemple 40 : Exemple de redirection

La commande suivante exécute le programme prog et envoie sa sortie dans un fichier sortie .out :

```
1 ./prog > sortie.out
```

Si le fichier existait déjà, son contenu est remplacé par la sortie de prog, sinon, il est crée et son contenu devient la sortie de prog.

Tube

Définition 40 : Tube

Un **tube** est un mécanisme de communication inter-processus dont les données sont organisés sous la forme d'une file.
Un tube redirige la sortie standard d'un premier processus vers le l'entrée standard d'un deuxième processus.

Tubes sous Linux

Exemple 41 : Tubes en shell UNIX

On peut utiliser le symbole | pour créer un tube entre deux programmes invoqués par des commandes.

```
1 history | grep cd
```

Nœud d'index

Définition 41 : Nœud d'index

Un **Nœud d'index** (ou inode) est une structure de données contenant des informations à propos d'un fichier.
L'inode contient des informations sur le contenu du fichier, ainsi que sur des métadonnées sur le fichier (droits d'accès, propriétaire, ...)

Norme EXT2

Exemple 42 : Norme EXT2

EXT2 est un système de fichier historique de Linux. Il a cédé sa place à d'autres variantes (EXT3, EXT4).
L'idée derrière l'organisation des fichiers en EXT2 est d'utiliser l'inode pour sauvegarder des informations spécifiques aux caractéristiques du fichier, ainsi que des pointeurs vers des blocs de données directement, ou indirectement.