Programmation impérative en C

Valisp 1. Présentation et allocateur

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 2 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- On va implémenter un interpréteur Lisp
  - un des plus vieux langage informatique
  - Premier langage haut niveau
  - Simple à implémenter.
  - Étonnement expressif (plus que Python par certains aspects !).
- Langage hautement dynamique
  - Premier language récursif
  - Premier language fonctionnel
  - Premier language interprété
  - Premier language avec conditionnel (if, then, else)
  - Premier language au typage dynamique
  - Premier language avec rammasse-miettes
- Bref, du Python mais en 1957.
- Ceci n'est malheureusment pas un cours sur Lisp … désolé !

Objectif

- La problématique centrale sera la gestion mémoire.
- On vise six objectifs :
  - Pour tous :
    1. Écrire un vrai programme C (plusieurs fichiers, non trivial, etc).
    2. Découvrir les structures de données dans leurs environnements naturels.
    3. Implémenter un ramasse-miettes (comment gérer la mémoire)
  - Pour ceux qui aime prendre du recul :
    4. Découvrir comment fonctionnent les langages dynamiques.
    5. Découvrir Lisp (culture général)
    6. ???
- Le dernier et véritable objectif est pour les meilleurs et les *happy-few*
  - Vous n'êtes pas encore prêt pour que je vous le révèle !
  - Au dernier cours, peut-être, si certains s'en montrent dignes.

Sommaire

- En 6 séances :
  - Écrire un allocateur (malloc/free)
  - Encoder les données (types)
  - Écrire un parseur (traduire un texte en données)
  - Écrire un environnement (gestion des variables) et l'interpréteur
  - Écrire les primitives (fonction prédéfinies) et gérer les erreurs
  - Écrire le ramasse-miettes
- La septième séance, voir que tout cela est bon et se reposer.
- Malheureusement, la vie est trop courte et l'on a que 5 séances.

- La problématique du projet étant la gestion mémoire.
  - on va gérer entièrement la mémoire.
  - pas de `malloc` ni de `free`
  - c'est nous qui allons les écrire
- `malloc` sera remplacé par `allocateur_malloc`
- `free` sera remplacé par `ramasse_miettes`
- **Objectifs de la semaine 1**
  - Création de la mémoire
  - Écriture de notre `malloc`
  - Écriture d'une partie du ramasse-miettes
  - Écriture de `free` (optionnelle)

- On va créer les différents types de notre langage
  - entiers, chaînes, fonctions, listes, etc.
- Pour chaque type
  - Constructeur
  - Prédicat
  - Accesseur
  - Afficheur
- Création des erreurs (erreurs fatales ou récupérables) ▸ Création des premières primitives (opérations de bases)
- **Objectifs de la semaine 2** (exemple)
  - Créer deux entiers.
  - Appeler la primitives qui les ajoute
  - Afficher le résultat

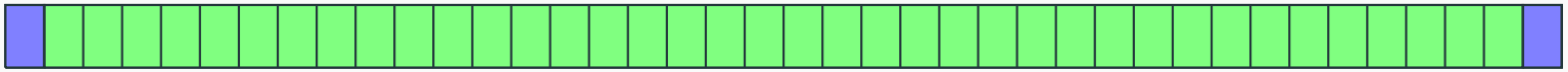
- Trois grosses étapes :
  - Création de l'environnement globale (les variables)
  - Écriture du ramasse-miette statique
  - On branche le tout et on fait tourner.
    - Je fournis `parseur.o` `parseur.h`
    - Je fournis `interpreteur.o` `interpreteur.c`
    - Je fournis `main.c`
- **Objectifs**
    - Je crée une variable `x` initialisée à 5
    - J'incrémente `x` (calcul de la somme)
    - La mémoire se libère automatiquement (plus de 5, ni de 1, seulement 6)

- Pour l'instant l'interpréteur fourni a deux gros défauts :
  - On ne peut pas créer de nouvelles fonctions (uniquement les primitives)
  - La ramasse-miettes se lance uniquement entre les calculs
- Vous écrirez l'interpréteur
  - C'est un des codes les plus mythique de l'histoire de l'informatique.
  - Toute la simplicité et l'élégance de Lisp se cache ici.
- On ajoutera l'allocateur dynamique qui se lance lorsque nécessaire.
- **Objectifs**
  - Obtenir le langage finale !

- On finit les TP précédents
- On ajoute des primitives
- On améliore et optimise ce qui aura été fait avant.
- Et le *parseur* ! Il faut l'écrire vous même !
- À ce moment là, vous aurez entièrement créé votre langage
  - de la lecture du fichier texte à l'affichage du résultat
  - de l'encodage des données à leurs libérations automatiques
  - de la gestion des variables (locales et globales) à la gestion des erreurs.
  - Très peu de lignes de code venant de moi.

Sommaire

- Je commence avec une mémoire vide (40 cases dont 38 libres).
  - La mémoire est constituée de deux blocs
  - Le premier à gauche : une case de métadonnées et 38 cases libres
  - Le second à droite : une case finale de métadonnées.



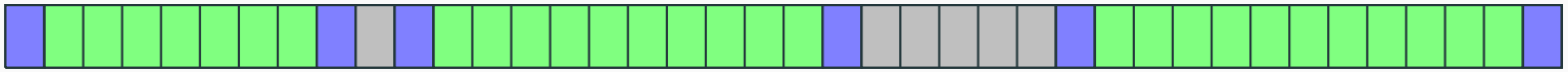
- Je réserve 7 cases :
  - Premier bloc occupée : 1 case → 7 cases
  - Second bloc libre : 1 case → 30 cases
  - Dernier bloc : 1 case

- On va noter cela [7] → [30] → [0]
  - en rouge les blocs occupés
  - en noir les blocs libres

Fonctionnement 2/2

- On commence avec
  - [7] → [1] → [10] → [5] → [11] → [0]

- On libère les blocs 1 et 3
  - [7] → [1] → [10] → [5] → [11] → [0]



- On alloue un bloc de 8 (dans le premier bloc libre et assez grand)
  - [7] → [1] → [8] → [1] → [5] → [11] → [0]

Fusion des blocs

- On libère l'avant dernier bloc
  - [7] → [1] → [8] → [1] → [5] → [11] → [0]

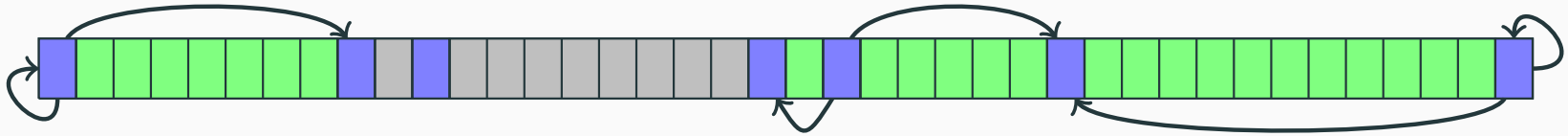
- Je veux allouer un bloc de 15. Est-ce possible ?
- Lors de la libération des blocs, il faut fusionner les blocs libres
  - [7] → [1] → [8] → [19] → [0]

- Pour cette raison, une liste chaînée ne suffit pas.
  - Il faut regarder si le blocs d'avant et d'après sont libres
  - Il faut une liste doublement chaînée.

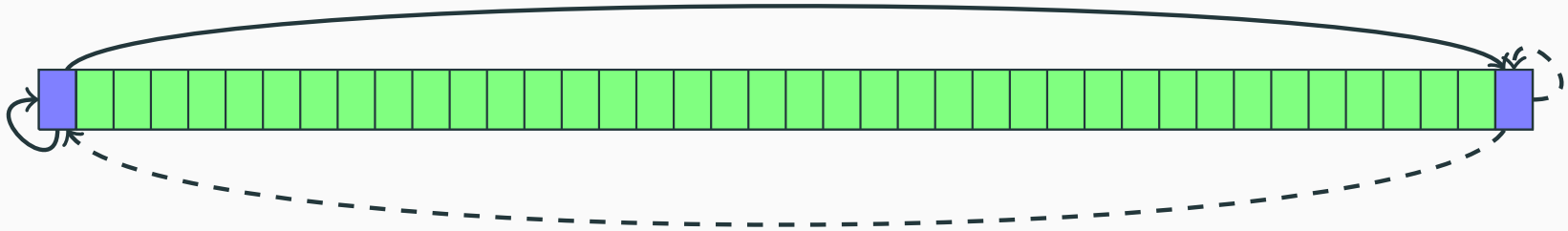
- Que mettre dans nos métadonnées ?
  - Le bloc précédent
  - le bloc suivant
  - l'état du bloc : libre ou occupé ?
  - une marque pour le ramasse-miettes : à conserver ou non ?

- Par abus de langage, on appelle bloc la case (bleu) des métadonnées.

- Chaque bloc a ainsi un précédent et un suivant.
- Le premier bloc est son propre prédécesseur.
- Le dernier bloc est son propre successeur.



- Situation initiale :

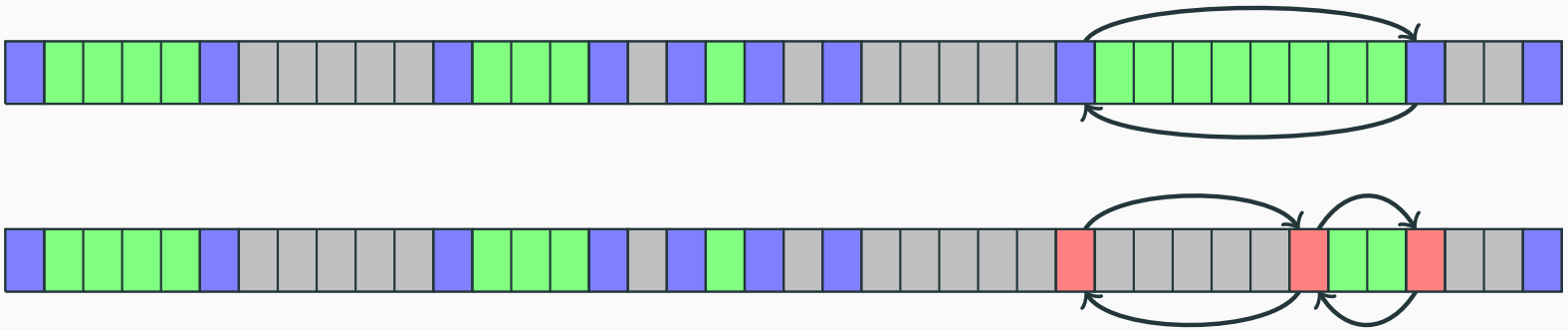


Sommaire

- On parcours la mémoire bloc par bloc
  - On commence avec `b` initialisé au premier bloc
  - À chaque passage de boucle on remplace `b` par son successeur
  - On s'arrête lorsque `b` est égale à son sucesseur

- Si ça vous inspire une boucle `for`, c'est normal...

- On parcours la mémoire bloc par bloc.
- Lorsqu'on trouve un bloc dont la taille est plus grande que ce l'on recherche :
  - Si c'est exactement la taille
    - on alloue le bloc (on le note comme étant occupé)
    - on renvoie un pointeur vers la première case
  - Sinon on modifie le bloc et on en crée un suivant
    - Il faut alors modifier trois blocs
    - Celui que l'on réserve ; le nouveau ; le successeur du nouveau

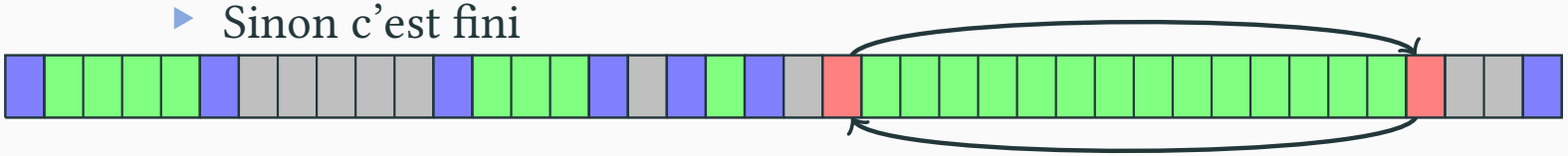


- On veut libérer le bloc rouge (remarque blanc = vert ou gris)

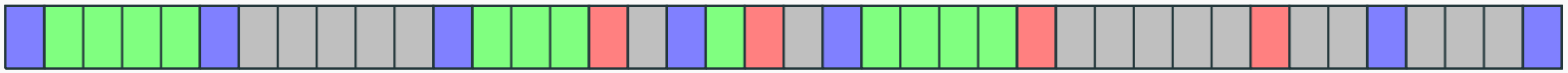
- Étape 1 : on marque le bloc comme libre

- Étape 2 : si le bloc suivant est libre
  - On fusionne en reliant le courant avec le suivant du suivant.
  - Sinon on passe à l'étape 3

- Étape 3 : si le bloc précédant est libre
  - On fusionne en reliant le précédant au suivant
  - Sinon c'est fini

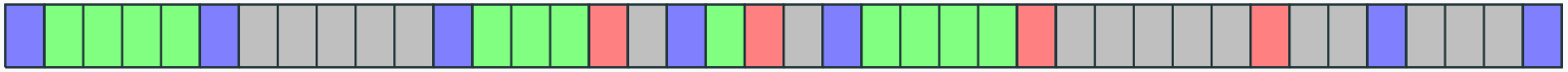


- Pour nettoyer la mémoire, on marquera les blocs à sauvegarder
  - *cf.* TP 4 et 5
  - Ici les blocs marqués seront représentés en rouge
- On suppose les blocs marqués, on cherche à nettoyer.
  - On veut passer de :

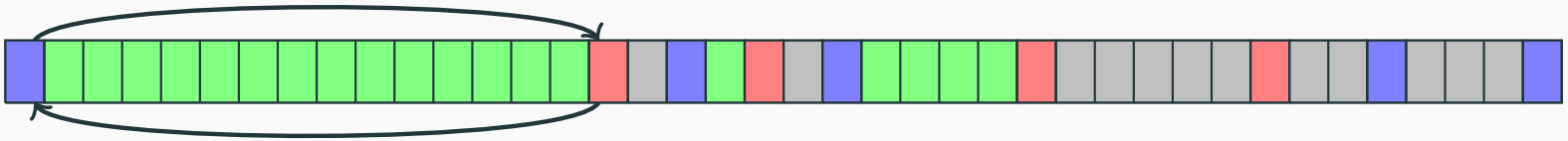


  - à :

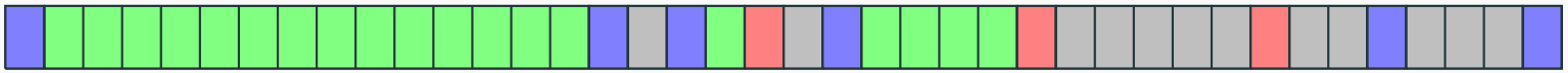
  - concrètement
    - on libère tous les blocs non marqués
    - deux blocs consécutifs ne peuvent pas être libre (sinon, on fusionne)
    - on enlève toutes les marques



- On parcours les blocs.
  - Si on tombe sur un bloc libre (non marqué = ici en bleu)
    - on cherche le prochain bloc à sauvegarder.
    - on relie les deux blocs ensembles.
    - on reprend le parcours à partir du bloc suivant



  - Si on tombe sur un bloc à conserver : on enlève sa marque



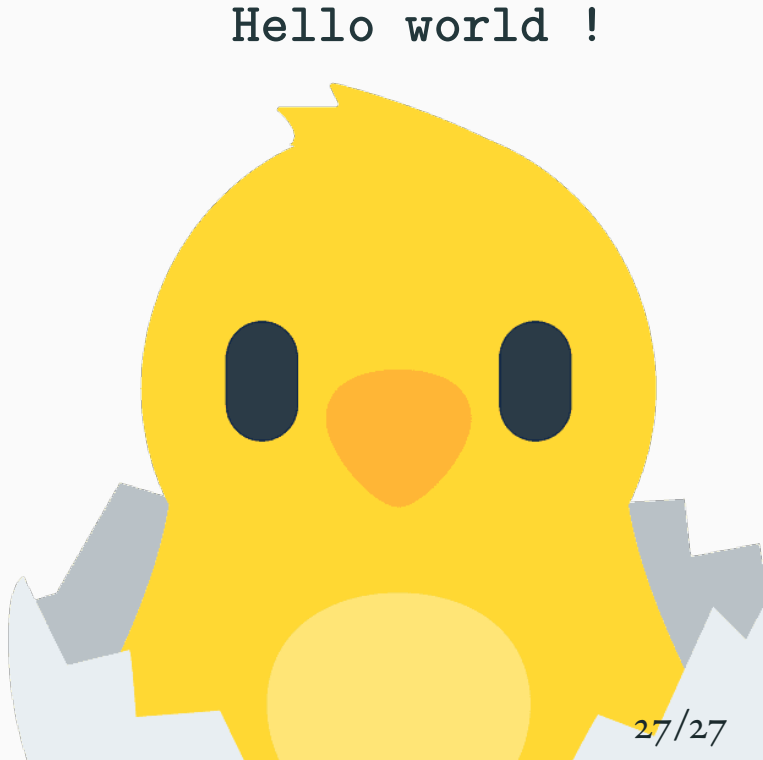
Sommaire

- Va-t'on coder les blocs avec des structures et des pointeurs ?
  - trop facile
  - trop coûteux en taille ($\approx$ 4 pointeurs = 4×64)
- La mémoire sera un « grand » tableau de 2^{15} cases de 32 bits.
  - La mémoire sera allouée une fois au début du programme de manière statique.
- On va coder chaque bloc sur 32 bits (`uint32_t`)
  - un bit pour la marque du ramasse-miettes
  - 15 bits pour coder l'indice du précédent
  - un bit pour indiquer si le bloc est libre
  - 15 bits pour coder l'indice du suivant

- on travaille sur un tableau `MEMOIRE` de `bloc` (alias pour `uint32_t`)
- Problème :
  - Malloc renvoie des pointeurs
  - nos algorithmes fonctionnent sur des tableaux
- Comment passer de `MEMOIRE[i]` à `*ptr` ?

- Question 1 : conversion pointeur/indice
  - Notre `malloc` nous dit que le bloc d'indice i est de taille suffisante
  - Quel adresse `allocateur_malloc` doit-il renvoyer ?
  - Réciproquement, quel est l'indice du bloc correspondant à l'adresse `ptr` ?
- Question 2 : conversion octet/`bloc`.
  - Si on demande de faire un `malloc` de n octets,
  - combien de cases doit on réserver ?

Merci pour votre attention
Questions



Cours 1 — Présentation et allocateur