



UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en C

Valisp I. 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

 Partie I. Présentation de Valisp

 Partie II. Programme futur

 Partie III. Allocateur

 Partie IV. Algorithmes

 Partie V. Codage

 Partie VI. Notre projet

 Partie VII. Table des matières

- ▶ On va implémenter un interpréteur Lisp
 - ▶ Un des plus vieux langages informatiques
 - ▶ Premier langage de haut niveau
 - ▶ Simple à implémenter
 - ▶ Étonnement expressif (plus que Python par certains aspects!).
- ▶ Langage hautement dynamique
 - ▶ Premier langage récursif
 - ▶ Premier langage fonctionnel
 - ▶ Premier langage interprété
 - ▶ Premier langage avec conditionnel (if, then, else)
 - ▶ Premier langage au typage dynamique
 - ▶ Premier langage avec ramasse-miettes
 - ▶ Premier langage avec de la méta-programmation (macro)
- ▶ Bref, du Python mais en 1957.
- ▶ Ceci n'est malheureusement pas un cours sur Lisp ... désolé!

- ▶ Écrire un lisp revient à résoudre de nombreux problèmes fondamentaux
 - Un des langages les plus simples et les plus expressifs
 - *Greenspun's Tenth Rule of Programming*

« *Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.* »

« Tout programme C ou Fortran suffisamment complexe contient une implémentation *ad hoc*, mal définie, lente et buggée de la moitié de Common Lisp »

- ▶ On vise trois grands objectifs :
 - Améliorer votre niveau en C :
 1. Écrire un vrai programme C (plusieurs fichiers, non trivial, etc).
 2. Réviser tout ce qu'on a vu en C
 3. Découvrir les structures de données dans leurs environnements naturels.
 - Concevoir concrètement une mémoire :
 4. Allocation
 5. Encodage
 6. Gestion dynamique des variables
 7. Ramasse-miettes
 - Prendre du recul sur les langages de programmation :
 8. Écrire un vrai langage de programmation
 9. Découvrir comment fonctionnent les langages dynamiques.
 10. Découvrir Lisp (culture général)

 Partie I. Présentation de Valisp

 Partie II. Programme futur

 Partie III. Allocateur

 Partie IV. Algorithmes

 Partie V. Codage

 Partie VI. Notre projet

 Partie VII. Table des matières

- ▶ En 6 séances :
 - Séance 1 : écrire un allocateur (malloc/free)
 - Séance 2 : encoder les données (types) et gérer les erreurs
 - Séance 3 : écrire un environnement et le ramasse-miettes
 - Séance 4 : écrire l'interpréteur
 - Séance 5 : améliorer le code (ramasse-miettes dynamique, closure)
 - Séance 6 : écrire un parseur (traduire un texte en données)

- ▶ La septième séance, voir que tout cela est bon et se reposer.
 - ▶ Malheureusement, il n'y a que 6 séances ; vous vous reposerez chez vous.

- ▶ L'objectif pour tous est les trois premières 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 **allocateur_free**
- ▶ Objectifs de la semaine 1
 - ▶ Création de la mémoire
 - ▶ Écriture de notre **malloc**
 - ▶ 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

- ▶ Deux étapes :
 - Création de l'environnement globale (les variables)
 - Écriture du ramasse-miette statique
- ▶ Objectifs
 - ▶ Je crée une variable x initialisée à 5
 - ▶ Je modifie x pour qu'il soit égale à 10.
 - ▶ La mémoire se libère automatiquement (plus de 5, seulement 10)

- ▶ 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.
- ▶ **Objectifs**
 - ▶ Obtenir le langage final!

- ▶ On améliore le projet
 - ▶ ajout des closures
 - ▶ amélioration du ramasse-miettes
- ▶ Le *parseur* qui transforme une chaîne de caractère en code Valisp.
- ▶ À ce moment là, vous aurez entièrement créé votre langage
 - ▶ de la lecture du 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 (uniquement le REPL).

 Partie I. Présentation de Valisp

 Partie II. Programme futur

 Partie III. Allocateur

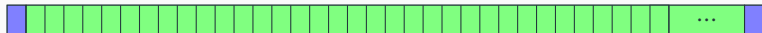
 Partie IV. Algorithmes

 Partie V. Codage

 Partie VI. Notre projet

 Partie VII. Table des matières

- ▶ Je commence avec une mémoire vide de 32 768 cases ($= 2^{15}$)
 - ▶ La mémoire est constituée de deux régions
 - ▶ La première à gauche : un bloc de métadonnées et 32 766 cases libres
 - ▶ La seconde à droite : un bloc finale de métadonnées.



- ▶ Si on demande à Valisp d'afficher la mémoire :
 - ▶ entre crochets [] et [x] l'occupation de la région
 - ▶ entre crochets [0] et [32767] les indices des blocs
 - ▶ entre parenthèses (32767) et (0) les tailles des régions

```
0 → [ 0] → 32767 [ ] 0x55b0ad0fa980 [ 0] (32766)
0 → [32767] → 32767 [x] 0x55b0ad11a97c [32767] (0)
```

- ▶ On commence par allouer quatre régions de taille 7, 1, 10 et 5

0	→	[0]	→	8		[x]	0x55b0ad0fa980	[0]	(7)
0	→	[8]	→	10		[x]	0x55b0ad0fa9a0	[8]	(1)
8	→	[10]	→	21		[x]	0x55b0ad0fa9a8	[10]	(10)
10	→	[21]	→	27		[x]	0x55b0ad0fa9d4	[21]	(5)
21	→	[27]	→	32767		[]	0x55b0ad0fa9ec	[27]	(32739)
27	→	[32767]	→	32767		[x]	0x55b0ad11a97c	[32767]	(0)



- ▶ On alloue les blocs là où trouve l'espace suffisant.
 - ▶ au début, les régions occupées se suivent
 - ▶ la première région est de taille 7 mais occupe 8 cases.
 - ▶ 8 cases = 1 bloc de métadonnée + 7 cases

- ▶ Je libère maintenant le bloc d'indice 8.

0	→	[0]	→	10		[]	0x55b0ad0fa980		[0]	(9)
0	→	[10]	→	19		[x]	0x55b0ad0fa9a8		[10]	(8)
10	→	[19]	→	21		[]	0x55b0ad0fa9cc		[19]	(1)
19	→	[21]	→	27		[x]	0x55b0ad0fa9d4		[21]	(5)
21	→	[27]	→	32767		[]	0x55b0ad0fa9ec		[27]	(32739)
27	→	[32767]	→	32767		[x]	0x55b0ad11a97c		[32767]	(0)



- ▶ Problème : j'ai deux régions libres consécutives.
 - ▶ l'allocation découpe des régions
 - ▶ si on les fusionne pas en libérant, il y a un risque de fragmentation
 - ▶ on s'interdit d'avoir deux régions libres consécutives.

- ▶ Que mettre dans nos métadonnées ?
 - Le bloc précédent (nécessaire pour la fusion)
 - le bloc suivant (nécessaire pour le parcours)
 - l'état du bloc : libre ou occupé ?
 - une marque pour le ramasse-miettes : à conserver ou non ?
- ▶ Il faut faire tenir ces quatre info dans un bloc (une case mémoire)

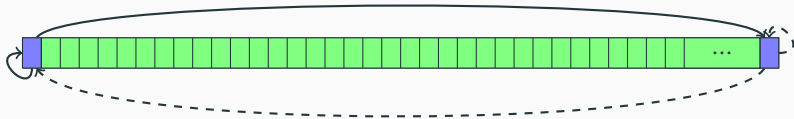
0	→	[0]	→	10		[]	0x55b0ad0fa980		[0]	(9)
0	→	[10]	→	19		[x]	0x55b0ad0fa9a8		[10]	(8)
10	→	[19]	→	21		[]	0x55b0ad0fa9cc		[19]	(1)
19	→	[21]	→	27		[x]	0x55b0ad0fa9d4		[21]	(5)
21	→	[27]	→	32767		[]	0x55b0ad0fa9ec		[27]	(32739)
27	→	[32767]	→	32767		[x]	0x55b0ad11a97c		[32767]	(0)

0	→	[0]	→	8	[]	0x55b0ad0fa980	[0]	(7)
0	→	[8]	→	10	[x]	0x55b0ad0fa9a0	[8]	(1)
8	→	[10]	→	19	[x]	0x55b0ad0fa9a8	[10]	(8)
10	→	[19]	→	21	[]	0x55b0ad0fa9cc	[19]	(1)
19	→	[21]	→	27	[x]	0x55b0ad0fa9d4	[21]	(5)
21	→	[27]	→	32767	[]	0x55b0ad0fa9ec	[27]	(32739)
27	→	[32767]	→	32767	[x]	0x55b0ad11a97c	[32767]	(0)



- ▶ Chaque bloc a ainsi un précédent et un suivant : $19 \rightarrow [21] \rightarrow 27$
- ▶ Le premier bloc est son propre prédécesseur : $0 \rightarrow [0] \rightarrow 8$
- ▶ Le dernier bloc est son propre successeur. $27 \rightarrow [32767] \rightarrow 32767$

- ▶ On initialise la mémoire avec deux régions



0	→	[0]	→	32767	[]	0x55b0ad0fa980	[0]	(32766)
0	→	[32767]	→	32767	[x]	0x55b0ad11a97c	[32767]	(0)		

 Partie I. Présentation de Valisp

 Partie II. Programme futur

 Partie III. Allocateur

 Partie IV. Algorithmes

 Partie V. Codage

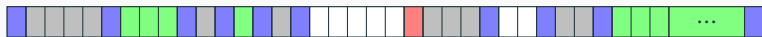
 Partie VI. Notre projet

 Partie VII. Table des matières

- ▶ On **ne** parcourt **jamais** la mémoire comme un tableau!
- ▶ On **parcours** la mémoire **bloc par bloc**
 - On commence avec b initialisé au premier bloc $b = 0$
 - À chaque passage de boucle on remplace b par son successeur
 - On s'arrête lorsque b est égale à son successeur

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

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

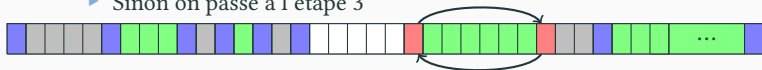


- ▶ Étape 1 : on marque le bloc comme libre



- ▶ 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 précédent est libre

- ▶ On fusionne en reliant le précédent au suivant



 Partie I. Présentation de Valisp

 Partie II. Programme futur

 Partie III. Allocateur

 Partie IV. Algorithmes

 **Partie V. Codage**

 Partie VI. Notre projet

 Partie VII. Table des matières

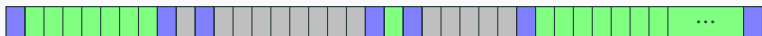
- ▶ Va-t-on coder les blocs avec des structures et des pointeurs ?
 - ▶ trop facile
 - ▶ trop coûteux en taille (≈ 4 pointeurs = 4×64)
- ▶ La mémoire sera un « grand » tableau de 2^{15} cases de 32 bits.
 - ▶ La mémoire sera allouée une fois au début du programme de manière statique.
- ▶ On va coder 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



- ▶ Pour éviter le flou autour des types entiers : `stdint.h`
- ▶ Dans ce qui suit, vous pouvez remplacer 8 par n'importe quelle valeur N parmi : 8, 16, 32, 64
 - entier d'exactly 8 bits : `int8_t`
 - ▶ sur certaines architectures et pour certains N peut être non défini.
 - entier d'au moins 8 bits : `int_least8_t`
 - ▶ toujours défini : optimise la taille (plus petite taille valide).
 - entier d'au moins 8 bits : `int_fast8_t`
 - ▶ toujours défini : optimise la vitesse
 - ▶ il peut être parfois plus rapide de travailler sur 32 bits que 8 bits
- ▶ Existe en version non signée :
 - ▶ `uint8_t`, `uint_least8_t` et `uint_fast8_t`.

- ▶ Notre mémoire n'est qu'un tableau contenant des entiers de 32 bits.
- ▶ Supposons que la case 20 correspond à un bloc et contienne 524312
 - On décompose 0 000000000001000 0 000000000011000
 - ▶ $0_2 = 0$
 - ▶ $000000000001000_2 = 8$
 - ▶ $0_2 = 0$
 - ▶ $000000000011000_2 = 24$
 - On en déduit :
 - ▶ que le bloc précédent est d'indice 8 et la suivante d'indice 24
 - ▶ que la région est libre et de longueur 3.
- ▶ La double liste chaînée est cachée dans certains entiers du tableau.

- ▶ Comment obtenir le nombre **1111111** (7 bits)
- ▶ J'ai un nombre de 9 bits, comment obtenir les trois derniers?
 - ▶ 111111**111** → **111**
 - ▶ 011010**101** → **101**
 - ▶ 110011**001** → **001**
- ▶ J'ai un nombre de 9 bits, comment supprimer les trois derniers?
 - ▶ **111111111** → **111111**
 - ▶ **011010101** → **011010**
 - ▶ **110011001** → **110011**
- ▶ J'ai un nombre de 9 bits, comment obtenir les trois du milieu?
 - ▶ 111**111**1111 → **111**
 - ▶ 011**010**101 → **010**
 - ▶ 110**011**001 → **011**



- ▶ `malloc` et `free` fonctionnent avec des tailles en octets et des pointeurs
- ▶ nos algorithmes fonctionnent avec des blocs de 32 bits et des indices
 - 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?

 Partie I. Présentation de Valisp

 Partie II. Programme futur

 Partie III. Allocateur

 Partie IV. Algorithmes

 Partie V. Codage

 Partie VI. Notre projet

 Partie VII. Table des matières

- ▶ Des fichiers divers :
 - ▶ Un `Makefile`
 - ▶ un script `maj-tests.sh` pour re-télécharger les fichiers de tests.
 - ▶ un dossier `img/` avec les images mémoires nécessaires pour les tests
 - ▶ un fichier `parseur.o` (incompatible avec macOS, merci d'installer GNU/Linux)
- ▶ Des fichiers C (au format `.c` pour les sources et `.h` pour les *headers*)
 - Les bibliothèques fournies par le prof (non faites pour être comprises)
 - ▶ `lib_memoire` : permet d'afficher la mémoire et de gérer des images
 - ▶ `lib_tests` : permet de tester chacune de vos fonctions
 - ▶ `lib_repl` : pour gérer le repl (=shell) Valisp
 - Les fichiers fournis par le prof (que vous pouvez comprendre)
 - ▶ `main` : définit le point d'entrée du programme
 - ▶ `tp1` : définit le REPL du tp1
 - ▶ `couleurs` : parce que M. Baldellon ❤ la couleur
 - Le fichier `allocateur` que vous devrez créer (90% du tp1).

- ▶ Le REPL utilise la bibliothèque GNU/Readline

```
olivier@valrose:~ $ sudo apt install libreadline-dev
```

SHELL

- ▶ Cette bibliothèque est sous licence GPL 3
 - ▶ si vous souhaitez publier votre code sous github
 - ▶ vous êtes obligé d'utiliser la même licence

Merci pour votre attention

Questions



Hello world !



Cours I — Présentation et allocateur

Partie I. Présentation de Valisp

Qu'est ce que Valisp

Pourquoi Lisp

Objectif

Partie II. Programme futur

Programme

Semaine 1

Semaine 2

Semaine 3

Semaine 4

Semaine 5 et 6

Partie III. Allocateur

La mémoire

Allocation basique

Libération basique

Allocation

Libération naïve

Libération fusion

Métadonnées

Une liste doublement chaînée

Situation initiale

Partie IV. Algorithmes

Parcours de la mémoire

Malloc

Free

Partie V. Codage

Codage

Types entiers explicites : `stdint.h`

Concrètement

Jouons avec les bits


Pointeur d'octets et tableau de blocs

Partie VI. Notre projet

Les fichiers

Readline et la licence

Partie VII. Table des matières

- ▶ © 2026 — Olivier Baldellon
- ▶ Ce document est publié sous licence **CC-BY Attribution 4.0** 
 - Vous êtes autorisé à :
 - ▶ **Partager** — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats pour toute utilisation, y compris commerciale.
 - ▶ **Adapter** — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.
 - Selon les conditions suivantes :
 - ▶ **Attribution** — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.
- ▶ <https://creativecommons.org/licenses/by/4.0/deed.fr>