Programmation en C

Martin Quinson

December 12, 2023

Présentation du module SYS1

- Ce cours vise à constituer une première introduction sur les ordinateurs et les systèmes informatiques.
- Il couvre principalement le langage C et le réseau.
  - 6 semaines: Le langage C est une excuse pour regarder comment fonctionne un ordinateur, sans aller dans les détails électroniques.
  - 6 semaines: Les réseaux sont un exemple de grand système informatique qui dure depuis 4 décennies car il était bien pensé à la base.
- Prérequis: prépa MPI, donc base du C. Au besoin, il y a un module de remise à niveau mercredi matin.
- MCC: un projet en binôme en semaine 4 à 6, puis un examen sur table à la fin.
- MCC: rendus de TP chaque semaine. Cela comptera un peu dans la moyenne avec un petit coef.
- Objectifs: introduction à la recherche en informatique appliquée
  - Pas de panique, ce n'est pas un cours d'ingénierie (même si c'est sympa). On veut donner une culture générale et de combattre quelques idées fausses.
  - Objectif1 est de montrer le large spectre de la recherche en informatique (luter contre la spécialisation)
  - Objectif2: mieux comprendre les vrais systèmes informatiques pour que les *problem-solvers* trouvent des problèmes sympas à résoudre. Eviter les affres de la page blanche
  - Et y'a même de la recherche en génie logiciel, en particulier à Rennes (Diverse).
- Différence entre théorie et pratique en informatique
  - La différence, c'est qu'en théorie, il n'y a pas de différence
  - La théorie, c'est quand personne n'y croit sauf l'auteur. La pratique c'est quand tout le monde y croit, sauf l'expérimentateur.
  - Idée reçue 1: Les ordinateurs manipulent des nombres mathématiques (inspiré du cours Intro to computer systems de CMU)
    - Int $\neq$ entiers: $\exists$ n $\in$ `Int` tel que $x^2 < 0$: $50\,000 \times 50\,000 < 0$ (pas de pb avec les flottants)
    - Float $\neq$ réels: (x+y)+z $\neq$ x+(x+z) parfois (pas pb avec les entiers)
      - $\boxed{\text{(1e20 + -1e20) + 3.14 = 3.14}}$ mais $\boxed{\text{1e20 + (-1e20 + 3.14) = 0}}$
    - `Int` n'est pas $\mathbb{N}$ mais c'est un anneau (commutativité, associativité, distributivité)

  - `Double` n'est pas $\mathbb{R}$ mais respecte relation d'ordre (monotonie, signe)
- Idée reçue 2: La mémoire d'un ordinateur est uniforme (RAM)
  - Pas sans limite: taille max mais aussi précision max
  - Source de bugs pénibles (malloc)
  - La vitesse dépend de l'endroit où sont stockées les données
    - ∀ i ∈ [0; 2028]
∀ j ∈ [0; 2028]
dist[i][j] = src[i][j]

      ∀ j ∈ [0; 2028]
∀ i ∈ [0; 2028]
dist[i][j] = src[i][j]
    - La seule différence est l'ordre des boucles: on parcourt les i avant les j ou le contraire
    - L'un met 4ms et l'autre 80ms. On verra plus tard pourquoi :)
  - Ce fait constitue le principal goulot d'étranglement des performances de calcul d'un ordinateur
- Idée reçue 3: Les ordinateurs ne font que calculer On trouve pleins d'autres problèmes de recherche intéressants qui ne sont pas directement liés au calcul
  - Performance dissimétrique des I/O ⇒ algo out of core (moins actif actuellement)
  - Compatibilité
    - Internet (TCP/IP) inventé pour ≈ 100 noeuds et fonctionne pour 4+ milliards de IPv4
    - Démarrage (BIOS, MBR, OS) fonctionne depuis les années 80 pour des matériels très disparates
    - Recherche: Génie log, équipe Diverse à Rennes
  - Concurrence: quand on coordonne différentes entités, il se pose une foule de problèmes intéressants. Recherche: preuve de programme (ariane, voiture), modèle-checking, algo très amusante (wait-free)
  - Usage fiable d'un média non fiable/faire avec ce qu'on a: les réseaux fonctionnent avec des cables de mauvaise qualité, et le logiciel améliore/compense cela. C'est pour ça que les ordis sont binaires (le premier ordinateur russe, le setun, était ternaire, ce qui est préférable d'un point de vue théorique puisqu'il y a une plus grande compacité entre le nombre de positions utilisées et la quantité d'information encodable).

    Recherche: OS (domaine moins actif en OS, pour l'instant), réseau (beaucoup), traitement d'image (énormément)

Chapter 1

Ordinateur

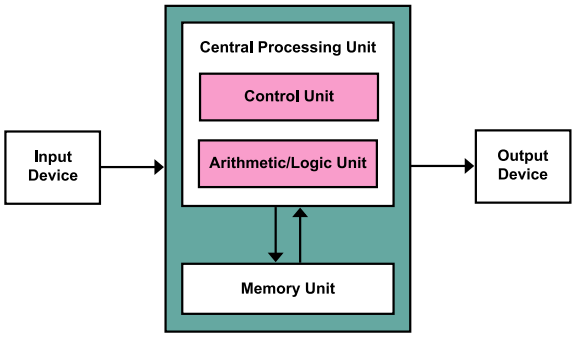
I) Qu'est ce qu'un ordinateur?

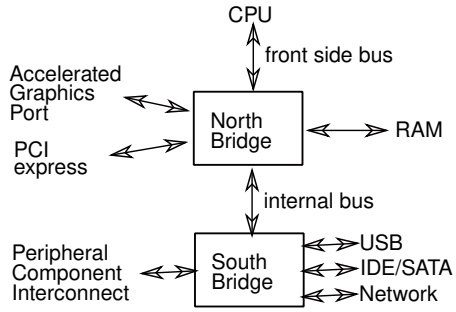
- Machine à calculer universelle.
  - La différence avec un calculateur est qu'il n'y a pas à le modifier physiquement pour lui faire calculer autre chose
  - Le nom FR est un vieux mot FR tombé en désuettude pour désigner Dieu, celui qui range les choses.
- Depuis quand?
  - 1936: Article fondateur de Turing sur la calculabilité et la Machine de Turing
  - 1946: Construction de l'ENIAC.
    - Environ 350 flops, ie 2500 fois plus rapide qu'un humain
    - Changer le programme prend jusqu'à 3 semaines pour tout recabler (c'est donc plutôt un calculateur programmable)
    - Pour info, les supercalculateurs modernes sont exaflopiques, 10^{18} flops. 1 milliard de milliard de milliard. Âge de l'univers en seconde: $\approx 4{,}3 \cdot 10^{17}$
  - 1962: Premier département d'informatique à l'université de Purdue, près de Chicago. Fork du département de génie civil
  - 1886: fondation de Tabulating Machine Company, qui fait des cartes perforées et deviendra IBM par la suite.

II) Contenu d'un ordinateur

- Prise de représentation sur les différentes parties d'un ordinateur. On attend CPU, GPU, disque, écran, clavier, bus, ... poussière.
- Modèle de Von Neumann (1945) suffisament simple pour être indémodable.

`https://en.wikipedia.org/wiki/Von_Neumann_architecture`

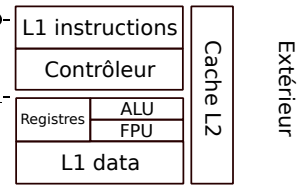




- Les cartes mères actuelles ont des puces qui implémentent grosso modo le modèle de droite
  - Les détails techniques changent beaucoup et souvent, mais l'idée est là.
  - De nos jours, le northbridge est souvent directement sur la puce du CPU pour gagner en vitesse
  - PCI et AGP ont disparu au profit du PCIe, maintenant assez rapide pour tout faire
  - Dans les laptops, tout est soudé directement donc la puce du bus remplace le PCIe

II.1) Le processeur

- Composants
  - Unités fonctionnelles, qui font les calcul.
    - ALU: Arithmetic/Logical Unit: logique booléenne et travail sur les entiers
    - FPU: Floating Point Unit: calculs sur les nombres à virgule flottante
  - Contrôleur, qui coordonne les calculs
  - Zones de stockage, certaines pour les instructions, d'autres pour les données, ou encore mixtes
- Principe général (à 3 Ghz)
  - *Fetch:* Le contrôleur récupère l'instruction suivante du programme
  - *Decode:* Le contrôleur décode l'instruction, sépare les opérandes
  - *Execute:* L'ALU ou FPU exécute l'instruction
  - Le résultat est stocké dans un registre



- Loi de Moore (ex-président d'Intel)
  - Le nombre de transistors double tous les 18 mois.
  - C'est une prédiction autoréalisante puisque Moore est un ex-PDG d'Intel et que la companie a comme business plan que tous les ordinateurs sont remplacés tous les 18 mois.
  - 1971: 2000 transistors sur un 4004; 2010: 2.6 milliards sur xéon; 2020: 40 milliards AMD epyc
- Types de processeurs
  - GPU: unité de calcul spécialisée dans le calcul vectoriel sur nombres flottants. Parallélisme de masse. Contient des ALU mais vraiment pas efficace pour les calculs

génériques. Comparable à une formule 1: très rapide si le calcul est très régulier, nul sinon.
- RISC vs. CISC: Reduced/Complex Instruction Set CPU. Plus ou moins d'instructions (de mots au vocabulaire). Plus de mots: plus de séquences directement inscrites dans le silicium, donc plus rapide et plus gros/complexe
- FPGA: aucune instruction inscrite dans le silicium, tout est interprété
- Little-Big Endian: ordre des bits pour écrire les nombres entiers. Les ingés Intel ont eu des goûts bizarres, mais soit. Le nom vient des voyages de Guillivers. AMD/x86 est little, ARM est (souvent) big.

II.2) La mémoire

- Pyramide d'accès: On peut représenter les types de mémoire par un triangle avec les ordres de grandeurs suivants
  - Registre: <512 octets pour <1ns
  - Caches: ko-Mo pour 5ns
  - RAM: Go pour 10-30 ns
  - Stockage de masse: To-Po pour 10ms
- Memory wall:
  - Latences mises à l'échelle, pour les ordre de grandeur. Imaginons qu'un cycle fasse une seconde

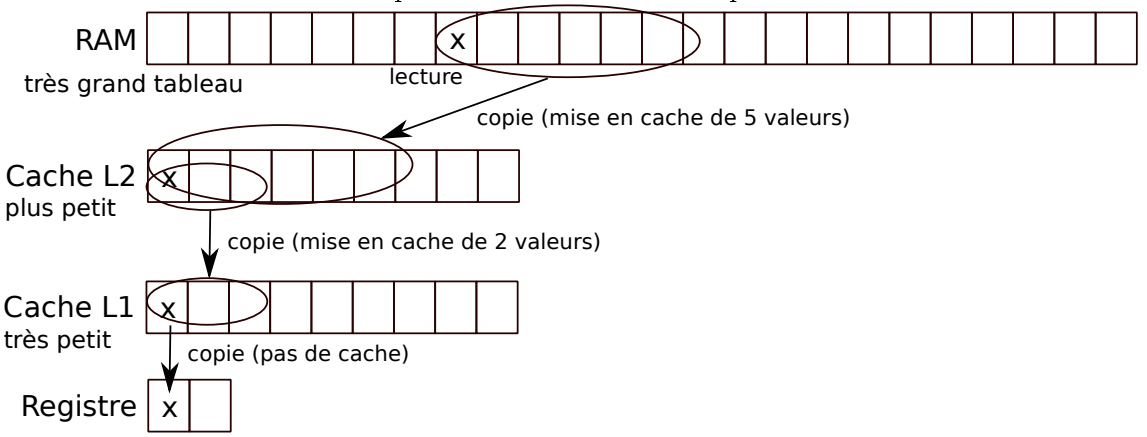
1 cycle CPU	0,3ns	1s
cache L1	1ns	3s
cache L2	3ns	9s
cache L3	13ns	43s
RAM	120ns	6mn
SSD disk	50-150μs	2-6 jours
disque rotationel	1-10ms	1-12 mois
Internet Oslo/Madrid ou SF/NY	40ms	4 ans
Internet transcontinental	180ms	18 ans
TCP retransmit	1-3s	100-300 ans

  - Le CPU va très, très très vite: 0,3ns par cycle, ça fait 10 cycles quand la lumière parcourt 1m. Le défi est donc de nourrir le CPU de données sans pause
  - C'est aussi pour ça que les registres ont une petite capacité: ils doivent être petits en taille pour être proche des unités fonctionnelles. S'ils étaient plus gros, la lumière mettrait plus de temps
  - Le memory wall s'aggrave: Vitesse CPU croit de 55%/an et celle mémoire de 10%/an.

II.3) Les mémoires caches

- L'étymologie vient des trappeurs nord-américains. Ils apportaient du matériel sur un chariot jusqu'à une sorte de camp 0, puis ils cachaient ce qu'ils ne pouvaient pas porter pour l'expédition.

- Dans l'ordi, on a un pb de latence entre la mémoire et l'ALU, mais pas de bande passante. On va donc rapatrier plus de données que demandées, au cas où on aurait ensuite besoin de la donnée juste après en mémoire.
- une mémoire cache est une petite réserve de mémoire proche du coeur du CPU



- La fois suivante qu'on accède aux données, on regarde d'abord si elle n'est pas déja stockée dans un niveau de cache intermédiaire. Si oui, on a économisé la latence. Si non, on charge la ligne de cache nécessaire, en virant une autre ligne au besoin. Pleins d'algos différents de gestion des caches, mais LRU (least recently used eviction) marche bien en pratique.
- "Coup de chance", les accès mémoire sont effectivement souvent linéaires. En fait, on programme exprès pour, et le compilo tente de démeller les accès et les données pour fluidifier.
  - En HPC, on cherche même à faire des structures de données *cache oblivious*, cad optimisées pour n'importe quelle taille de cache.
  - C'est ce qui explique la différence de code entre les deux programmes de copie de matrice (4ms vs. 80ms pour une matrice 2048): l'un utilise très efficacement les caches, l'autre passe son temps à les invalider après chaque accès.

III) Performance des ordinateurs

- C'est le sujet du module de HPC l'an prochain en S1

Chapter 2

Système d'exploitation

I) Qu'est ce qu'un système d'exploitation

- Citez moi des exemples d'OS: Linux, Windows, Mac, Android, FreeBSD, blablabla
- Combien d'OS existent? Autant que d'élèves qui ont fait le projet du MIT ou Stanford
- Quel est le lien entre Linux et UNIX? Et entre MacOSX et UNIX
- Quel est l'OS le plus utilisé sur terre ?
  - Desktop: windows sans aucun doute. Serveurs: linux. Au total: Minix qui est embarqué dans la secure zone des processeurs Intel.
- Pourquoi étudier Linux dans cette diversité?
  - Je connais bien
  - C'est ouvert
  - C'est fait en C, ce qui tombe bien dans ce module, avouez
  - le noyau windows va bientôt s'arrêter quand le WSL3 sera encore mieux intégré. Comme Mac OSX mais avec un Unix moderne dessous.

II) Fonctions de l'OS

- Historiquement, c'était juste un ensemble de fonctions mises en commun, mais maintenant, l'OS a une mission bien plus large
- C'est le programme entre les programmes et la machine. Objectifs:
  - **Protéger le matériel des applications** (sous Dos, activité malveillante de la tête de lecture pouvait détruire le disque dur)
  - **Protéger les applications les unes des autres** (sous Dos ou M99, un bug d'un programme = crash généralisé)
  - **Adapte l'interface** du matériel (simplifie/uniformise)
    - les pilotes traduisent les requetes API de façon spécifique au matériel
    - Sous Dos, les jeux spécifiaient les cartes son utilisables
  - **Virtualise les ressources**: multiplexage CPU, swap mémoire

Fin de la séance 1

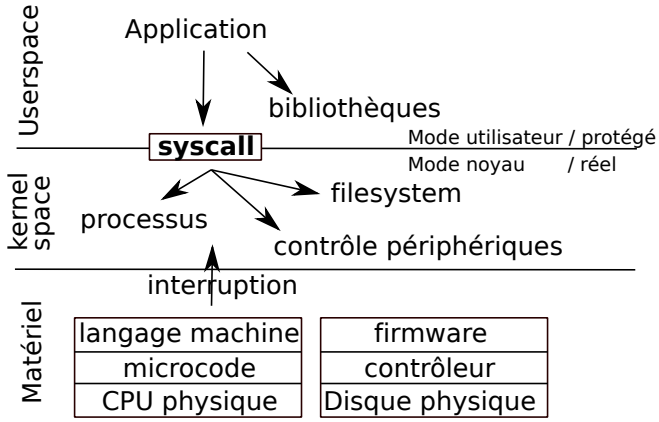
III) Fonctionnement d'un OS (pas vu en cours)

On n'a pas le temps de voir cette partie en cours, même si c'est un peu dommage. Le texte suivant est là pour votre culture. C'est beaucoup plus facile à comprendre après avoir fait le TD sur le M99.

III.1) Protéger les périphériques des applications

- Motivation
  - Sous Dos, on avait des virus détruisant le lecteur de disque.
  - Le malware Stuxnet a été déployé en 2012 pour détruire les centrifugeuses à uranium iraniennes (cf Wikipedia)
- Principe: **interposition et médiation**. Les applications ne peuvent pas accéder directement au matériel, doivent demander à l'OS
  - quand on lit/écrit dans les plages mémoire correspondant aux périphériques, le CPU vérifie qu'il est en mode protégé
  - **mode protégé*** = registre sur un bit ou deux.
    - Valeur 1 => mode protégé, càd confort pour les applications qui peuvent compter sur la mémoire virtuelle et des fonctionnalités pratiques pour faire du multitâche;
    - Valeur 0 => mode "réel", réservé à l'OS qui voit la machine telle qu'elle est vraiment. On parle aussi de Ring0, car en fait, les x86 ont plus de deux modes. Les drivers de l'OS pourraient s'exécuter dans les niveaux intermédiaires pour les sortir du noyau, mais c'est rarement le cas en pratique. Par contre, quand on virtualise des OS, l'hypervisor est en ring0, les OS virtualisés sont dans les ring intermédiaires, et les applications dans le dernier ring 4. Oui, ce paragraphe est parfaitement hors sujet. Il n'y a rien de grave à ne pas le comprendre.
  - Ce n'est pas être super-utilisateur sur linux, c'est encore autre chose (cf. plus loin)
  - Comment on fait pour mettre changer de ring? On peut lâcher des privilèges et monter librement, mais pour baisser le ring et gagner des privilèges, il faut déclencher une interruption
- **interruption**: quand le CPU détecte un evt particulier, il passe en mode réel et exécute le gestionnaire (= la fonction) adéquat, dont l'adresse a été déclarée au moment dqu démarrage
  - En gros, chaque interruption du CPU redonne la main à une fonction du système d'exploitation.
    - D'où l'importance pour l'OS d'être celui qui s'exécute en premier, pour enregistrer les gestionnaires d'interruption. Si un virus arrive avant l'OS (en prennant le contrôle de la séquence de boot avant l'OS), tout est fichu. Le virus fera croire à l'OS que tout va bien tout en gardant le contrôle du CPU
  - Si une appli tente d'écrire sur un périphérique directement, le CPU réveille l'OS après une interruption, et il est probable que l'OS termine l'application
- **appel système**: pour faire proprement, l'application doit demander à l'OS
  - l'application écrit sa requête et les paramètres dans un endroit spécifique de la

mémoire
  - l'application déclenche une interruption signifiant "cher OS, j'ai besoin d'un service"
    * sur x86, y'a une instruction assembleur `INT` qui signifie `interrupt`, et Linux lui passe le paramètre 80
  - le CPU se réveille suite à cette interruption. Il sort du mode protégé, puis exécute la fonction de l'OS correspondant à cette interruption.
  - l'OS vérifie la validité des paramètres (l'application a le droit de faire cet accès)
  - l'OS fait le travail demandé par l'appli. Au besoin, écrit le résultat dans la mémoire de l'application
  - l'OS repasse en mode protégé, et retourne l'exécution à l'application.
- **schéma récapitulatif**



III.2) Protéger CPU contre le blocage par app

- Problème: accaparement du CPU par un programme donné qui empêche de rendre le contrôle à l'OS.
  - Soit à cause d'un bug (boucle infinie)
  - Soit programme malicieux, au sens anglais, i.e. qui cherche à nuire au système
- Principe: **préemption*** Je ne donne que ce que je peux reprendre.
- Solution: une interruption donnée est déclenchée 1000 fois par seconde. Cela redonne la main à l'OS
- Donc 1000 fois par seconde, l'OS a l'occasion de changer le programme en cours d'exécution si nécessaire
  - Pb de performance maîtrisé si le gestionnaire chargé est suffisament rapide. Qques pourcents de perte de perf.
- multiplexage CPU: une seule chose à la fois, mais suffisament vite pour que plusieurs applis s'exécutant en alternance donnent l'impression de s'exécuter en parallèle. Comme un film où 60 images fixes par seconde donne une impression de fluidité
  - Le scheduling est un pb de recherche amusant, même s'il est un peu limité au coeur du CPU car faut aller vite sans bcp contexte
  - les tâches interactives / tâches de fond traitées très différent par linux.

  - On schedule les bouts d'applis distribuées, aussi, et c'est un pb scientifique plus riche (même si random reste difficile à battre en pratique). Notez que gagner 2% d'efficacité fait une énorme différence en conso électrique
- Changement de contexte: remplacement du context d'exec par un autre.
  - Sauvegarde de tous les registres (= du contexte) qque part en mémoire
  - Lecture de l'autre contexte ailleurs en mémoire
  - Changement de PC (le compteur de programme, ou PC, qui désigne là où on en est de l'exec d'un programme) pour changer d'activité

III.3) Protéger les applications les unes des autres

- Pb: protéger la mémoire : on veut empêcher une appli de lire/écrire dans la mémoire des autres
  - vol de mot de passe, ou crash de l'autre appli
  - Vient en plus de la protection contre l'accaparement du CPU, bien sûr
- Principe: **interposition*** pas d'accès direct à la mémoire. **médiation** généralisée (mode protégé. En Ring0, on voit la mémoire telle quelle)
  - Chaque application ne manipule que des adresses virtuelles [0; MAXINT) et n'a aucune idée des adresses mémoire physiques
  - C'est une protection efficace de la mémoire des autres processus (programme en cours d'exec), puisque l'appli ne peut même pas nommer la mémoire des autres
  - C'est d'ailleurs aussi une protection du matériel, puisqu'on ne peut pas accéder aux adresses physiques servant d'interface aux périphériques
  - Il y a quelque part en mémoire une table d'association (processus; plage mémoire virtuelle) -> (plage mémoire physique)
- Gros problème de perf en vue, si chaque accès mémoire demande encore plus de temps
  - La solution est d'ajouter du matériel dédié. Le MMU (mem managmt unit) est une zone du silicium du CPU en charge de faire ces translations à toute vitesse
- Ceci explique d'ailleurs un message d'erreur classique en C: le SEGFAULT (erreur de segmentation)
  - cela veut dire que le programme a accédé à une zone mémoire virtuelle qui ne correspondait à aucune entrée dans la table du MMU.
  - Pour l'OS, l'accès est invalide puisqu'il sait pas quoi faire. Donc l'appli a fait une faute, donc peine de mort: l'appli est terminée...

III.4) Adapter l'interface

- Puisqu'on intercepte toutes les requêtes de l'appli aux périphériques, on peut facilement offrir une interface commune à chaque catégorie de matériel
- l'OS a besoin d'un bout de code qui sait traduire les requêtes haut niveau en accès direct à la mémoire, bas niveau. C'est le pilote (driver en anglais)
  - Distribué avec le matériel y'a 10 ans (sur un beau ptit CD inutile)
  - Inclu dans le noyau Linux (c'est la majorité des lignes de code de Linux)

  - Téléchargé sur internet au besoin, mais attention à la sécurité: ce code s'exécute DANS le noyau, en ring0. D'ailleurs, on a inventé la notion de micro-noyau pour éviter ce genre de pb. Il y a pas mal de recherches dans ce domaine de nos jours (même si c'était encore plus vif il y a une décennie). Regardez le projet Pistachio si vous êtes curieux.

III.5) Virtualiser les ressources

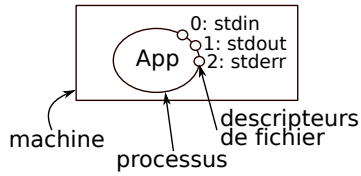
- C'est l'art de faire croire aux applis qu'on a plus de ressources qu'en réalité.
- Le multiplexage CPU est une forme de virtualisation puisque les applis ont l'impression de s'exécuter en //
- L'interposition mémoire permet de virtualiser la mémoire, aussi. Chaque appli peut nommer 2^{64} octets en mémoire virtuelle, mais l'ordi est bien plus limité.
  - Quand la mémoire physique commence à manquer, certaines pages mémoires d'application sont déplacées sur disque et retirées de la mémoire vive.
  - Cela fait de la place pour plus de mémoire
  - Quand on a besoin des pages passées sur disque, c'est très très lent. Il faut donc choisir avec précaution.
  - **swap**: L'espace disque qui stocke des pages qui débordent de la RAM. **swapping*** action de déplacer des choses sur disque.
- Machine virtuelle: un programme qui fait croire aux autres programmes qu'ils sont sur une vraie machine alors que pas du tout. Bcp de technos.
  - Réécriture de binaire: la plus simple.
    * lourde : tout est réécrit, ce qui permet d'exécuter du code ARM sur un CPU x86.
      · Si on le fait à l'avance c'est du Binary Translation. Si on le fait juste au chargement du programme (et non longtemps à l'avance), c'est du Dynamic Binary Translation (sujet de recherche de Simon Rokicki entre autres). Quand Apple est passé d'architecture PowerPC à Intel, le programme Rosetta était une machine virtuelle faisant du DBT.
      · Si on le fait à l'exécution de chaque instruction, cela revient à interpréter l'ISA de la machine cible. Quand la machine virtuelle QEMU suit cette approche, on parle d'émulation logicielle, par opposition à la virtualisation matérielle (cf plus bas) qu'il sait aussi faire
    * légère : On change juste les INT dans assembleur (les appel système) pour mettre un appel de fonction à l'OS virtualisé. Il a été modifié pour faire des INT non interceptées quand il a besoin d'accéder aux ressources matérielles, qui sont gérées par l'hypevisor en ring0, un OS qui gère les OS au dessus de lui. s'exécuter
  - Accélération matérielle: le code s'exécute sans modification. En cas d'interruption, on utilise le bon vecteur de gestionnaires d'evt en fonction du contexte du programme qui s'exécutait. Ca demande un matériel doté de plusieurs vecteurs de gestionnaires (banal de nos jours), mais ça va bcp plus vite.

Début séance 2

IV) Design d'UNIX

- UNIX est un OS dominant depuis 1970, grâce à qques bonnes idées de design
- C'est aussi parce que C et UNIX ont été inventés ensemble, par les mêmes personnes. Un éventuel replaçant devrait peut-être remplacer les deux à la fois.
  - Ca sera pas Windows/C++.
  - Peut-être que ça sera être Rust (qui est un C bien plus extrêmement moderne) et Redox (qui vise à implémenter les dernières avancées en OS, au besoin en s'éloignant un peu d'UNIX – mais pas tant que ça)
  - Mais on s'égare, revenons déjà au design d'UNIX avant de vouloir le dépasser

1. Tout est un fichier (ou y ressemble, au moins)
   - Utilisateur : stdin /stdout /stderr
     - Dans tout le cours, on utilisera la même représentation simple ci-dessous. Les machines (ordinateurs) sont des boites, les programmes qui s'exécutent dedans sont des ellipses, et les descripteurs de fichier sont des petits ronds à la surface de l'ellipse.



   - Autres programmes: socket réseau, tubes
   - Le noyau: `/proc` et `/sys`
   - Le matériel: `/dev`
   - Au passage, une chose agréable sous Unix est que la place des choses sur disque est standardisé (FHS – Filesystem Hierarchy Standard), et je ne comprend pas qu'Apple mette autant d'efforts à ne plus le respecter.
   - Notez que le réseau n'est pas parfaitement intégré vu que c'est arrivé en 1983 seulement.
     - Plan 9 est un UNIX distribué, mais Unix est good enough. Plan 9 (et Inferno) sont quasi abandonnés maintenant. Ceci étant dit, il demeure que Plan 9 contenait de très bonnes idées qui n'ont pas encore été reprises ailleurs. Ceux qui cherchent des idées novatrices (ainsi que les collectionneurs bizarres) devraient se pencher sur le cas de cet OS.
       * Plan 9 is not a product, it is an experimental investigation into a different way of computing. The developers started from several basic assumptions: that CPUs are very cheap but that we don't really know how to combine them effectively; that **good*** networking is very important; that an intelligent user interface is a Right Decision; that existing systems are not the correct way to do things, and in particular that today's workstations are not the way to go. [from `http://wiki.xxiivv.com/site/plan9.html`]

2. Assembler les outils simples interchangeables

- Ca aide à l'établissement d'un écosystème aux briques évolutives
- Mécanismes dans le shell:
  - Redirection des E/S avec des tubes: `<`, `>`, `»`, `2>`, `|`
  - grep, sed, cat
  - voir les pages de manuel
- C'est dommage que la même chose n'ai jamais vu le jour pour l'UI, malgré les nombreuses tentatives
  - DOM-X windows a tourné au cauchemard de sécurité
  - AppleScript avait l'air sympa mais ca n'a pas marché
  - Gnome pousse l'usage du javascript (et c'est donc une mauvaise idée)

3. Préférer les textes en clair (vs. binaire)

   - Ca participe à simplifier l'assemblage de briques hétérogènes
     - Interopérabilité et debug.
     - Plus généralement, UNIX applique souvent le KISS
   - C'est pour la config (vs. système de registres de windows) et pour les logs sur disque.
   - Mais c'est aussi pour les protocoles du web (http and co)
   - Dans l'histoire récente, cet aspect tend à disparaitre avec l'avènement de System D qui fait le contraire
     - Grosse crise dans Debian vers 2015, à deux doigts de la scission. Je n'adore pas ce qui est advenu

4. Repères historiques pour UNIX et C (pas vu en cours)

   - Ils ont été inventés ensemble, par les mêmes personnes
   - UNIX
     - 1965: ambitieux projet MULTICS aux Bell Labs
     - 1969: Abandon de MULTICS, le projet UNICS commence
     - 1970: début officiel du projet UNIX
     - 1973: réécriture en C
     - 1982: AT&T perd son procès anti-monopole. UNIX est donné aux universités, dont Berkeley, et à diverses entreprises qui vont commercialiser leurs versions.
     - 1983: TCP/IP arrive, bien après le design d'UNIX. Le résultat ressemble à une verrue.
     - 80-90: UNIX War. c'est BSD contre System V (V=5 pas la lettre). Développer des applis portable entre toutes ces différences subtiles devient difficile
     - 90-00: normalisation. Les différentes normes POSIX posent un socle commun
     - 00-10: fin des UNIX propriétaires, abandonnés les uns après les autres par leurs entreprises. AIX (IBM), IRIX, HP-UX. Microsoft Windows règne sans partage. La légende dit que Microsoft s'assure de la survie d'Apple (en les finançant secrètement) Pour éviter le procès antitrust.

  - 10-20: internet et les webapps font passer windows au second plan. On parle peut-être moins des OS installés.
- Langage C:
  - 1967: BCPL aux Bell Labs
  - 1968: B (simplifié mais tjs très illisible)
  - 1971: C version K&R. Brian Kernighan et Dennis Ritchies (+ Thompson pour UNIX)
  - 1983: C++ Bjarne Stroustrup
  - 1989: ANSI C
  - 1990 puis 1999 puis 2011: versions de ISO C
  - Le C++ a bcp plus de variantes, l'institut de normalisation en ajoute tjs plus. C++11, C++14, C++17, C++20. Et c'est pas fini.

V) Interfaces de l'OS

- Interface graphique vs. shell: même usage. Le mieux est celui que l'on connait le mieux.
  - Pour moi c'est le shell car interactions plus riches (paramètres), et c'est utilisable à distance (accès ssh)
  - Je rêve d'une interface mélangeant les deux, comme caja-terminal si ce dernier fonctionnait vraiment
- Interface de programmation: En C la plupart du temps, même s'il faudrait arrêter ça et passer au Rust maintenant.
  - Importance de la notion de standardisation: avant POSIX, nombreuses variantes d'Unix incompatibles, ce qui rendait l'écriture de prog impossible.

1. Le shell en pratique
   - Syntaxe fondamentale: `cmd options paramètres`
   - commandes de base: ls, cd, gcc, emacs
   - CtrlC CtrlZ & bg fg: lancer emacs.
   - Historique des commandes, édition de commande, Complétion,
   - Tubage de commandes
   - Interaction avec les processus: `ps aux`, `kill 42`, `killall -STOP firefox`
   - micro scripts de conversion d'images ou de concaténation de fichiers mp3: `convert -rotate 90`
   - Lien vers les exos: `https://www.github.com/mquinson/shutorial`

VI) Sécurité des systèmes d'exploitation

1. Les systèmes réels ont des failles
   - Parfois exprès pour ne pas brider l'implémentation
     - `while (1) fork()`

  - `while (1) {int*a = malloc(512); *a = '1'; }`
  - `while true ; do mkdir toto ; cd toto ; done`
  - Y'a pleins de *undefined behavior* en langage C aussi. Après un tel comportement, tout est possible (overflow sur les entiers => parfois $2000 < -2000$)
- Souvent pas exprès. Très nombreux problèmes de sécurité
  - dans les applis, motivation pour le Rust, donc. Faites du Rust je vous dis
  - mais aussi dans le matériel: meltdown
    * C'est une fuite de données (canal caché) car mise en commun des caches entre les processus.
    * Attaque RowHammer (!= Meltdown): Lecture de la mémoire des autres depuis un script javascript s'exécutant dans le sandbox du navigateur
    * Pour s'en protéger, Linux vide les caches à chaque changement de contexte. Perte de presque 30% de performances.

2. Solutions classiques en sécurité
   - Réponses techniques:
     - les mécanismes de protection mis en place par l'OS
     - Utilisateurs UNIX, droit d'accès (`rwx r-x r-x` : 755) et délégation de droit (sudo et setuid)
       * répertoires: r=ls; w=création éléments; x=cd
       * `chmod` et `ls -lh`
     - Windows offre des Acces Control List plus fins
   - Réponses sociales:
     - quota disque $\rightsquigarrow$ publication des stats d'usage de chacun
     - Parfois, c'est la seule solution. Il est interdit juridiquement de reverse-engineerer certains matériels.
       * Portail Skylander sur Wii = simple lecteur RFID, mais service juridique interdit de regarder comment ça marche. Grosses menaces.
       * Wii turing complete, donc le constructeur ne peut pas les brider pour n'exécuter que du code maison. Mais l'installation de certains jeux efface tout code non certifié Nitendo, y compris les créations personnelles.

3. Quelques propriétés de sécurité
   - confidentialité: accessible aux seuls autorisés.
     - basé sur le secret et les maths. chiffrement asymétrique
     - confidentialité persistante (forward secrecy): si qqun sauvegarde le canal chiffré puis fini par trouver la clé dans le futur, il peut tjs pas lire. Génération de nouvelles clés à intervales réguliers, et échange des nouvelles clés sous la protection des précédentes. Donc on peut pas remonter le flux.
   - intégrité: pas de modif indésirée
   - authentification: garantie utilisateur est qui il prétend.
     - Attention à la biométrie car changer de mdp quand qqun sait le reproduire est

plus facile que changer d'oeil ou d'empreinte digitale.
  - Un ministre allemand a perdu l'exclusivité de ses empreintes suite à une photo HD en conférence de presse.
- Anonymat: /!\ pseudonymat pas suffisant; TOR: on noie son flot dans la masse
  - On veut aussi du secret futur (par exemple si PGP est cassé dans 15 ans)
- non-répudiation: celui qui a envoyé ça ne peut pas nier l'avoir fait. Banque, contrat.
  - Centralisé (notaire), basé sur la possession d'un secret (carte bleu ou clé PGP), ou décentralisée (contrat bitcoin)

4. Règles à suivre
   - Protéger ses données (c'est la loi)
     - Il faut offrir une protection raisonnable au regard des données protégées. Certains systèmes ne sont pas connectés à internet (mais stuxnet a fonctionné quand même)
     - Hadopi: charge de preuve inversée sur le défaut de sécurisation
   - Ne pas contourner les protection (c'est la loi)
   - Avoir la possibilité technique ne donne pas le droit
     - c'est comme avec le sac des petites vieilles dans la rue
   - La loi actuelle ne plaisante pas
     - Loi programmation militaire: internet = circonstance agravante, passage en antiterrorisme
     - jurisprudence blue touf: notion d'effraction n'existe pas. Interdit d'entrer quand tu sais que t'es pas sensé être là, même si c'est ouvert
   - Vous êtes informaticiens, vous ne pourrez plus plaider l'incompétence technique, alors faites attention. Surtout vues la législation et la jurisprudence françaises

Fin de la séance 2

Chapter 3

Le langage C

I) Pourquoi étudier le C?

- Intérêt historique et culturel
  - Très répandu (tous les OS, 50% de Debian)
  - Bcp de langages s'en inspirent (C++, Java, C#, Python, Ruby, Rust, PHP)
  - Permet de faire des programmes efficaces (mais c'est pas automatique: on peut tout à fait écrire un code C inefficace)
- Intérêt pédagogique
  - Comprendre le C permet de comprendre la machine
    - C'est le langage évolué le plus proche de l'architecture
    - Devenu un modèle opérationel concret de la machine, par co-design: c'est la lingua franca entre les programmeurs les plus bas niveau et les ingénieurs concevant les ordinateurs, donc le matériel se conforme au modèle puisque c'est ce à quoi les programmeurs s'attendent.
  - Comprendre C et machine pour mieux programmer en <insert language name>
    - Les VM cachent plus ou moins bien les détails (CAML vs. Perl)
    - Faut connaître les pointeurs C pour bien comprendre les références Python/Java
- Avantages et inconvénients du C
  - (+) C'est un langage très simple, très KISS. Syntaxe en une seule page A4
  - (-) AUCUN garde fou, les outils font une confiance aveugle aux humains, c'est fatiguant
  - (+) On contrôle tout (optimisation) (-) il faut tout contrôler
  - (+) On peut comprendre les méandres (-) 1001 façons de se tirer dans le pied
  - Presque backward compatible avec 1970: (+) stable et pérène (-) qques héritages regrettables
  - Pas de bibliothèque standard: (-) il faut tout refaire soi-même
  - $\Rightarrow$ C'est un langage à connaître, mais à n'utiliser qu'au besoin

II) Les bases du langage (sous-section pas vue en cours)

Si vous venez de prépa MPI, vous avez probablement déjà vu le contenu de cette partie (qui prenait 45mn de cours avant). À l'ENS Rennes, tout ceci est maintenant vu dans le module de rattrapage MPI, et ce qui suit n'est pas détaillé en cours.

II.1) Historique

- Inventé dans les années 1970 par Brian Kernighan et Dennis Ritchie, du laboratoire Bells de AT&T, pour abstraire un peu l'écriture de programmes de l'ordinateur cible. Compilateur pour traduire le code source des humains en langage machine des ordis $\rightsquigarrow$ ne pas tout réécrire à chaque fois.
- Mais ça reste un langage très proche de la machine (compilos très limités à l'époque)
- La première machine cible est le PDP-11: 24ko de mémoire $\rightsquigarrow$ KISS
  - La mémoire est un tableau d'octets sans structure
- (le nom est une mauvaise blague car c'est une variante du langage B de laboratoires Bells)
- Le C++ est une extension du C, semblable en apparence mais bien différente (on ne confond pas)

II.2) Syntaxe de base du langage C

- **On utilise le pense bête, distribué à chacun.e**
- C'est très similaire au Java ou C++ (forcément), et assez similaire au Python où on marque les blocs
- Il faut déclarer les variables à l'avance, qui sont typées pour aider le compilo (pas pour la sémantique).
  - `int` et `double` pour tout faire (booléen : entiers — vrai $\neq 0$)
- Opérateurs arith (+ - ** % ++ +=) logiques (&& || !) relationnels (< <= != ==) binaires (& | ^ «)
- `if () {} else {}`

  `while () {}`

  `for (init; cond; inc) {}`

  `switch () {case:  break; }`
- Préprocesseur: les lignes commençant par `#` passent dans une machine à cherche-remplace améliorée
- Lecture ensemble du code "Premier programme" du pense bête. Include, fonction, scanf/printf.
- Définition de fonctions et appels de fonction
- Tout le reste (ou presque) c'est en option, pas dans le langage
  - conçu en 1970, c'est à l'humain de s'adapter aux outils
  - messages d'erreur parfois cryptique

II.3) Écrire du code C

- Workflow attendu
  - On écrit son source dans un fichier `toto.c` (avec geany)
  - On compile ce fichier (ie, traduction en binaire exécutable) `gcc toto.c -o binaire`
  - On exécute ce binaire `./binaire`
- Compiler ? C'est quoi en vrai ?
  - Phase 1: préprocesseur (define, include, ifdef)
  - Phase 2: compilation = traduction en assembleur
  - Phase 3: édition de liens = puzzle des ptits bouts d'assembleur (si +ieurs fichiers)

III) Pratique du langage C

- Premières bétises possibles:
  - Exécuter le source : impossible, le C est pas interprété (le CPU ne comprend que le langage machine – cf. M99)
  - Compiler sans resauvegarder (un IDE aide, mais il faut savoir faire sans – e.g. à l'agreg)
  - Exécuter sans recompiler (idem)
  - Ne pas lire les erreurs ou warning (clang plus compréhensible)
- Erreurs classiques qu'il faut savoir reconnaître
  - syntax error: facile
  - missing main function: car oui, il faut un main. On ne peut pas écrire de code hors des fonctions comme en python
  - redefinition d'une fonction
  - segfault: c'est le grand problème
- Comment réagir: don't panic; lire le message d'erreur; ne pas lire le second message d'erreur (le compilo est aux fraises)
- Il faut faire preuve d'auto discipline et utiliser les rares outils à notre disposition.
  - `-Wall -Wextra -Werror -g -fsanitize=undefined`
  - Écrire du C moderne (éviter certaines choses autorisées par le compilo – place des déclarations)
  - gdb et valgrind pour chasser les bugs (y'a un TP spécifique)
  - KISS (simplicité algorithmique), car un algorithme embrouillé donne rarement un beau code
  - Le code doit être lisible car c'est vous-même qui allez le lire le plus souvent. Et puis, on n'a pas le niveau de l'IOCCC (Intl Obfuscated C Code Contest). On regardera ensemble les 3 premières entrées de la liste suivantes (en fonction du temps dispo), sur le code distribué.
    * 2012 endoh1 (simulation fluide) avec les entrées pour-out, column et clock
    * 2000 dhyang (quine affichant le visage de Saitou Hajime puis aku soku san, ie sin swift slay)

  * 2004 arachnid (simulateur labyrinthe déplacement: adws)
  * 2014 deak (calcul de fractale)
  * 2014 maffido (mario)
  * 2015 mills2: un programme de compression très très court
  * 2015 mills1: un flappy bird ascii

IV) Lire et écrire dans des fichiers

- Deux interfaces disponibles dans Unix
  - Haut niveau (printf/scanf): performances et interface raisonnables, moins de contrôle. C'est ce qu'on va voir ensemble
  - Bas niveau (read/write): contrôle parfait, donc meilleures perfs si bon usage (et bien pire si mal utilisé), et interface inamicale. Par exemple, il faut savoir convertir les nombres en suite de lettre-chiffres
- Interface printf et scanf
  - Relire le premier code de la feuille à ce sujet; le reste est bien expliqué sur la page man
  - Formatages:
    * %d: int
    * %f: double (pour scanf, il faut %lf)
    * %c: char
    * %s: string (char*)
    * %%: caractère %
    * %03.2f -> 003.14 (3 caractères à gch, complété par des 0, et 2 à droite)
- Interface fprintf/fscanf:
  - Comme printf/scanf mais on précise sur quel descripteur de fichier on veut lire ou écrire
  - Tester si le fichier est arrivé à la fin: feof()
  - Lire un caractère est piégeux, faut bien vider le retour chariot que l'humain a mis en plus. Cf code proposé.
  - Lire ligne à ligne est carrément difficile sans `getline`, même si on ne comprend pas tout dans le code proposé.
  - Ouvrir / fermer un fichier avec fopen, comme sur la feuille de pompe

V) Paramètres en ligne de commande

- argv est un tableau de chaines de caractères, une cellule pour chaque paramètre de la ligne.
- argv[0] = nom du binaire
- argv[1] = premier paramètre (if any)
- argv[argc-1] = dernier paramètre

- argv[argc] = NULL

Chapter 4

La mémoire en C

I) Identifiants C, portée et durée de vie

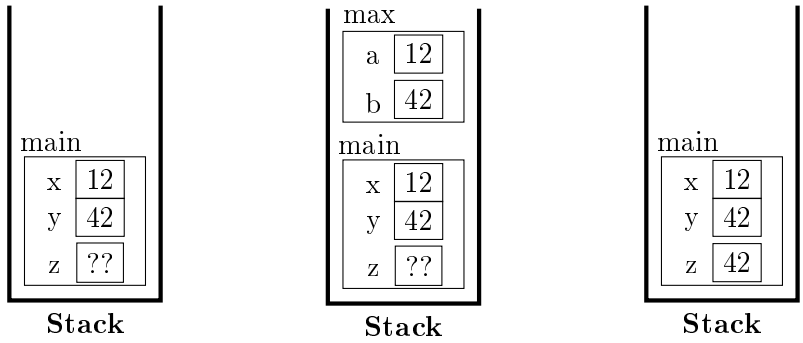
1. Portée des identifiants locaux et globaux
   - Pas de différence entre variables et fonctions (application un peu extrême des idées de Von Neumann)
   - Les identifiants sont globaux ou locaux. Cela leur donne une visibilité ou portée (scope) et une durée de vie spécifique.
     - Les variables locales sont dans une fonction.
       * Scope: visible depuis le bloc appelant
       * Durée de vie: jusqu'à la fin du bloc
       * Faire des fonctions locales n'est pas très C-ish, même si les compilos modernes acceptent.
     - Identifiants globaux (variables et fonctions)
       * Scope: partout ; Durée de vie : toujours (jusqu'à la fin du programme)
   - Étude du programme 1 sur la visibilité
   - Oui, on peut faire des locales à un sous-bloc, aussi.
   - Non, faire des variables de même nom qui se masquent ainsi n'est pas une bonne idée.
   - Pour comprendre, il faut imaginer des variables qui s'empilent, et indicer a par la ligne de déclaration (a_2 pour la globale)

```
(l5)  a:0  b:0
(l9)  a:?  b:0
(l14) a:10 b:10
(l18) a:10 b:0
(l20) a:10 b:15
(l22) a:10 b:0
```

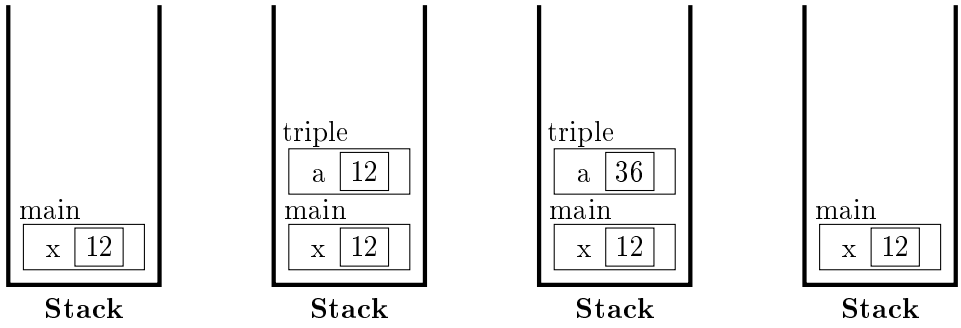
2. Paramètres et pile d'appel
   - Quel est l'affichage du **programme 2** (fonctions et paramètres) ?
     - Cela inverse les valeurs de `a` et `b` car les paramètres ligne 4 sont inversés. haha, très drôle.
     - En fait, non. C'est pas drôle, il ne faut pas écrire des pièges à soi-même comme ça.
   - La pile d'appel est un endroit de la mémoire où sont créés des **cadres de pile**, un contexte d'exécution pour chaque appel de fonction (récursif ou non). C'est

là que vivent les variables locales. Créé à la demande lors de l'appel, puis détruit lors du `return`.

- Les paramètres sont des variables locales (que la bienséance seule interdit de modifier même si le **programme 3*** montre que le compilo laisse faire)
- En C, (tous) les paramètres sont passés par copie.
- Le M99 ne fonctionne pas exactement comme ça, puisque les paramètres sont consommés. Il faudrait améliorer le M99 pour corriger.
- Étudions le **programme 4a**



- Attention, le passage par valeur pose parfois des pièges, comme on va voir dans le **programme 4b**
  - On calcule bien 36 dans la fonction, mais cette zone mémoire est effacée en sortie de fonction



3. Mot-clé `static`

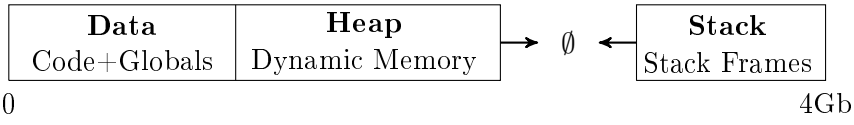
- Quel est l'affichage du **programme 5*** ?
  - La colonne des a est remplacée par des zéros; l'incrément de a en fin de fonction est sans effet puisque la variable est détruite après chaque appel
- Le mot-clé `static` a deux significations:
  - Associé à une variable locale, cela augmente sa durée de vie tandis que sa portée est inchangée. Cf. **programme 6**, qui fait ce à quoi on s'attend: la fonction `nextInt` énumère les nombres entiers
  - Associé à une variable globale, cela réduit sa visibilité au fichier courant. Sa durée de vie est inchangée. C'est l'équivalent Java ou C++ de `private`

	portée	durée de vie
Fonction	projet	∞
Variable globale	projet	∞
globale `static`	$\approx$ fichier	∞
locale `static`	bloc	∞
locale	bloc	bloc

- Plus précisément, une globale statique est limitée à l'unité de compilation, c'est à dire au .o constitué.
- Il est important ici de comprendre que la portée est définie à la compilation, tandis que la durée de vie est à l'exécution.
- Ces nouvelles connaissances posent la question de *où* sont stockées les locales statiques en mémoire.
  - on dirait une globale : initialisé une seule fois et persistant entre les appels
  - Il faut parler système pour comprendre ce mystère

II) Organisation logique de la mémoire

1. Memory layout d'un processus dans l'OS
   - Les locales `static` sont troublantes
     - On dirait une globale (initialisé une seule fois, presistante) mais déguisée en locale (visible ici seulement)
     - Pour comprendre, il faut savoir comment est organisée la mémoire du processus.
     - (un processus est un programme en cours d'exécution dans l'ordinateur)
   - Il y a trois segments mémoire, de 0 à 2^{32} en 32bits
     - **Data**: Là où se trouvent le code compilé + les variables globales. C'est ce qu'on avait en M99
     - On reparle du **tas*** bientôt: c'est là où on fait les `malloc` pour ceux qui connaissent. Il n'y en a pas en M99 car le tas est une fonctionnalité de l'OS.
     - **Pile**: Contient les cadres de piles comme vu la semaine dernière, et comme dans la M99
     - Il y a un trou (une zone inutilisée) entre la pile et le tas, qui peuvent grandir tous les deux. Si collision, alors le processus a épuisé la mémoire disponible.

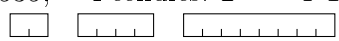


   - C'est de l'**adressage virtuel**, car l'OS fait de la médiation sur les cases mémoires lues.
     - A chaque accès mémoire, le programme lit une adresse virtuelle sans savoir où les données sont physiquement en mémoire. Le CPU a une table associative

(TLB – translation lookaside buffer) qui à chaque couple (*pid*, *adresse virtuelle*) associe une adresse physique.
  - Il faut faire efficace pour les performances générales donc fait par matériel: MMU = zone du CPU.
    * De plus la TLB groupe la mémoire par pages, pas des adresses individuelles
  - C'est fondamental pour assurer les fonctions de l'OS vu la semaine dernière: protection des applis; virtualisation mémoire; uniformisation des accès mémoire depuis l'appli
- Cela explique les locales `static`: à la compilation, ce sont des locales, mais le compilateur les place dans le segment Data, pas dans les cadres de pile.

2. Espace d'adressage virtuel

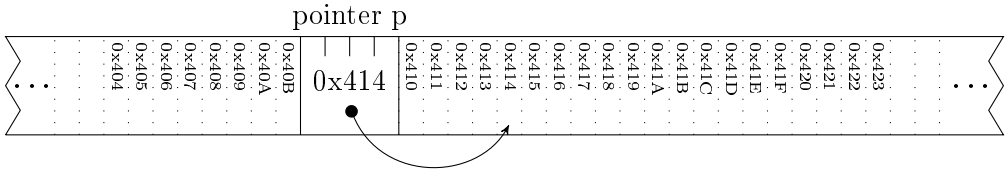
- C'est un autre nom pour la mémoire d'un processus. Elle peut donc être vue comme un grand tableau de case mémoire successives. Chaque case fait un octet sur tous les systèmes (hard et OS) que je connais.
- **Adresse mémoire**: numéro de la case dont on parle, tout simplement. En adressage virtuel, hein.
- *Pourquoi la pile commence à 4Go ?* Car je fais mes dessins en 32bits et que MAXINT=2^{32}=4Go sur cette architecture.
  - En fait, sous linux, la pile commence à 3Go car le dernier Go de l'espace d'adressage est un mapping direct d'une zone de l'OS pour rendre les context switch entre l'appli et l'OS plus efficaces. Mais c'est hors programme.
- *Et si l'ordinateur n'a pas tant de mémoire?* $\rightsquigarrow$ virtualisation: des pages swappées sur disque au besoin
- *Contenu d'une cellule*
  - Avec 8 bits on code 256 valeurs. Par exemple $[0, 255]$ ou $[-127, 128]$
  - Pour manipuler des plus grandes données, on agrege plusieurs cases (lues et interprétées ensemble)
  - 2 cellules: $2^{16} = 65535$; 4 cellules: $2^{32} \approx 4 \cdot 10^{10}$; 8 cellules: $2^{64} \approx 1.8 \cdot 10^{19}$



- *Organisation interne*
  - Il n'y a aucune meta-donnée, ce qui est cohérent avec le langage C: si tu l'as pas fait, y'en a pas.
  - On ne sait donc pas interpréter les données qui se trouvent dans la mémoire
    * comme en M99: on sait pas si 110 est `LDA 10`, ou si c'est la valeur numérique 110.
    * En C, on ne sait même pas si c'est une valeur ou un morceau (l'une des cases) d'une valeur

III) Pointeur

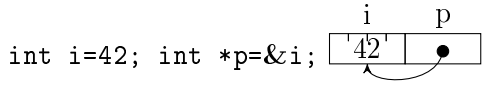
- C'est un mot qui fait peur quand on apprend le C, et c'est une source d'erreur très importante. Peut-être la principale
- Mais à la base, c'est juste une variable numérique contenant une adresse mémoire
- En 32bits, il faut 4 cases pour stocker une adresse



- Pour savoir interpréter une case pointée, il faut connaître le type de donnée stockée à cet endroit

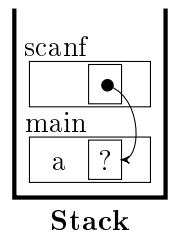
- On peut pointer vers une case contenant un pointeur. On obtient par exemple `int**`

1. Utiliser les pointeurs
   - La valeur numérique des pointeurs n'a aucune valeur sémantique (valeur `0x414` sans importance).
     - Ce qui compte, c'est ce vers quoi ça pointe (c'est un pointeur vers là où la variable `cpt` est stoquée)
     - C'est à quoi sert l'opérateur `&` qui se lit *adresse de*



   - On a déjà rencontré ce `&`: dans les paramètres de la fonction scanf. Cela explique comment elle peut modifier les variables dans la fonction appelante:
     - Et c'est pour ça que la fonction a besoin du `%d`: c'est pour savoir comment interpréter ce pointeur.
     - Oui, les pointeurs permettent de modifier la mémoire hors du scope. Mémoire magma informe.

```
int main() {
  int a;
  scanf("%d",&a);
}
```



   - On peut maintenant corriger le code de la fonction triple sur la feuille
     - Le paramètre est de type `int*`, on lit et modifie avec `*a`, on appelle avec `&a`

2. Pièges classiques avec les pointeurs

- **#1: L'étoile * a une sémantique extrêmement lourde.**
  - une étoile en trop ou pas assez dans un programme, et c'est le SEGFAULT (mort du processus)
- **#2: L'étoile a deux sens très différents**
  - `=int *p=` declares a **pointer variable*** `p` which is a pointer to an integer value
  - `=*p=` is then the **pointed value**, interpreted according to the pointer type
  - (that's actually three meanings when counting ×, the multiplication)
  - `=int *p; p=12;=` selects where it points in memory
  - `=int *p; *p=12;=` changes the memory in the pointed area
  - Pascal was a bit more reasonable: `INTEGER ^p` vs. `p^` (pas le même ordre au moins)
  - In Java, there is no pointers, but reference to objects are close to that concept

3. Le pointeur NULL
   - Utile pour dire non initialisé, ou invalide (comme dans la fonction `fopen`)
   - Par convention, c'est la valeur numérique 0 (même ça, ça n'est pas dans le langage. Un simple define)
   - Pour s'assurer que l'OS fait bien le SEGFAULT attendu, on a qques pages sans permission de r/w au début

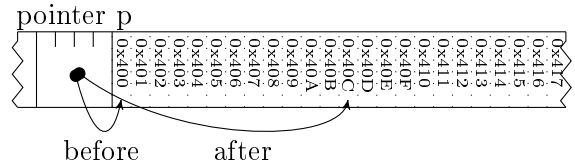
Fin de la séance 3

4. Arithmétique des pointeurs
   - Addition: pointeur + entier : valide.
     - C'est un décalage en case, pas en octets.
     - C'est déroutant, mais c'est parce que les pointeurs ressemblent aux tableaux en C

       `p[i]` $\triangleq$ `*(p+i)`

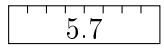
     - Donc `after = before + sizeof(int)*3` car ce sont des entiers.

```
int *pi=0x400;
pi=pi+3;
printf("pi:%x\n",pi);
```



   - Soustraction: pointeur - entier : valide
     - C'est la même chose, mais en décalant vers la gauche.
   - Pointeur - pointeur : Valide s'ils pointent sur la même chose. C'est le calcul du nombre de cases entre eux
   - Toutes les autres opérations entre pointeurs et entiers sont invalides (division, multiplication, etc)

5. Transtypage (cast en anglais)
   - c'est l'art de convertir un type dans un autre. Par exemple `int a = (int)b`.
     - Quel que soit le type de `b`, le compilateur fait de son mieux pour en faire un entier
   - On retrouve des transtypages dans à peu près tous les langages.
   - Deux types très differents de transtypages en C.
   - Dans tous les cas, le type en C ne dénote pas de la sémantique, mais taille et représentation en mémoire

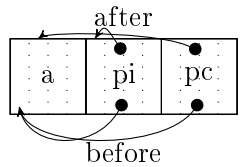
   (a) Transtypage de scalaires
   - un scalaire est une valeur normale: un entier, une lettre un double.
   - Lors d'un transtypage de scalaire, on change la valeur.
   - `double d = 5.7;` 

     `int i = (int)d;` 

   - Cela change la représentation en mémoire.
     - Dans l'exemple, double représentés en IEEE 754 (partout), sur 8 octets; entier sur 4 octets en 32bits.
   - Cela peut induire une perte irréversible de précision
     - Dans l'exemple, on ne retrouvera jamais le 0.4 depuis la variable `i`

   (b) Transtypage de pointeurs
   - Mémoire inchangée (donc valeur inchangée), mais change la sémantique.
     - Càd, change la façon d'interpréter les pointeurs (arithmétique)

```
int a;
int *pi=&a;
char *pc=pi;
pi++;
pc++;
```



   - Dans l'exemple: `pi` se décale de 4 octets car il pointe sur des entiers; `pc` se décale d'un seul octet.
   - C'est cohérent avec l'idée que les pointeurs peuvent être utilisés comme des tableaux en C

6. Pointeurs génériques: `void*`
   - Ce sont des pointeurs dont on ne connaît pas le type.
     - Exemple: les paramètres de `printf` ou `scanf`
     - Exemple: manipulation directe de la mémoire `memcmp`, `memcpy` (si pas overlap), `memmove` (si overlap)
   - A l'origine on ne pouvait pas faire d'arithmétique des pointeurs sur ces pointeurs, et c'est assez logique
     - Mais `gcc` l'autorise quand même, en supposant que `sizeof(void)=1` car c'est trop pratique

    * C'est une extension GNU, donc `clang` le fait aussi mais pas `icc` ni `visualC`
  - Oui, la taille du vide n'est pas nulle, mais le vide est de taille 1 :)

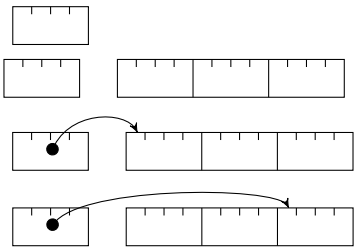
7. Pointeurs sur fonction

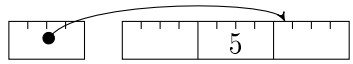
- C'est utile pour passer des callbacks (préciser un comportement à exécuter quand un événement arrive) ou pour faire du *dynamic dispatch* (associer un code à exécuter à un type de structure, pour faire un pas de plus vers la POO en C. Pas forcément une bonne idée: C++ préférable)
- Définir une variable de type pointeur: il faut des parenthèses.
  - `int (*fun)(void); //`:
    * La variable est nommée `fun`; son type est `int(*)(void)`.
  - Si on oublie les parenthèses, `int*` est plus prioritaire, et le résultat n'a pas de sens.
- Affecter une valeur à un pointeur sur fonction: pas besoin d'esperluette
  - `int mafonction(){ return 42; }` // Déclaration d'une fonction
  - `fun =&mafonction;` // La variable `fun` contient maintenant un pointeur vers `mafonction`.
  - `fun = mafonction;` // exactement pareil : le `&` est optionnel
  - car aucun autre sens possible pour cette expression : une fonction en rvalue ne peut être qu'une prise d'adresse, donc le compilateur est permissif
- Invoquer la fonction pointée: naturellement
  - `x = (*fun)()` // La variable `x` vaut maintenant 42
  - `x = fun()` // Exactement comme avant, même si c'est très moche. C'est autorisé car non-ambigu.

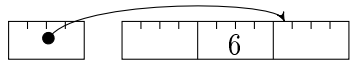
IV) Tableaux

1. Similarités et différences avec les pointeurs

- Tableaux et pointeurs se ressemblent beaucoup en C, sans être la même chose: conversions automatiques
  - C'est une source de complexité très avancée. Les plus curieux liront le tuto "la vérité sur les pointeurs et les tableaux en C" (sur la page du cours) pour comprendre. Les concepteurs du langage ont un regret pour ce point précis : ils ont fait trop compliqué.
- Exemples de code pour comprendre les points communs et différences
  - `int *p;`
  - `int tab[3];`
  - `p = tab;`
  - `p += 2;`



- `tab[1] = 5;` 

- `p[-1] = 6;` 

(aucune vérification des bornes en C, jamais)

- Dans cet exemple, `tab = p` n'aurait aucun sens: `tab` n'est pas une case mémoire modifiable.
  - Il faut voir `tab` comme une valeur numérique, comme 42. On ne change pas la valeur de 42.
  - Chaque fois que le compilateur voit `tab`, il écrit une valeur numérique, l'adresse du début de `tab` dans le segment *Data* qu'il construit (ce qui est joyeux à calculer avec l'ASLR address space layout randomization, mais c'est vraiment hors sujet).
  - C'est aussi pour ça qu'on ne peut pas écrire `tab1 = tab2` Le compilateur ne voit que des adresses. Ce serait comme lui dire d'exécuter l'affectation `32 = 14`. Nonsense.
- Pour bien comprendre la différence entre pointeur et tableau, réfléchissons à la taille mémoire occupée.
  - `p` est un pointeur $\rightsquigarrow$ 4 octets en 32bits; `tab` est un tableau de 3 cases $\rightsquigarrow$ 12 octets en 32 bits
  - Mais quand `sizeof` compte la place occupée par le tableau, il compte toutes les cases à la fois. Donc un `int t[3]` occupe 12 octets (3 cases de 4 octets chaque).
  - D'ailleurs, on compte les cases avec `sizeof(tab)/sizeof(tab[0])`.
- Donc les pointeurs et les tableaux sont différents.
  - Sauf dans les types de paramètres: `fun(char *p)` $\triangleq$ `fun(char p[3])` $\triangleq$ `fun(char p[])`
  - Là, les pointeurs et les tableaux sont rigoureusement identiques, et la taille du tableau est ignorée.
  - Plus précisément, le tableau est dégradé en pointeur et on ne peut plus utiliser la ruse précédente pour connaître la taille du tableau, car le pointeur n'a plus la taille du tableau pointé.
  - C'est historique, mon ami.
- Les pointeurs de tableaux ne sont pas des tableaux de tableaux (ni des pointeurs de pointeurs)
  - si `int tab[3] = { 0, 1, 2};` alors `&tab` est de type `int (*)[3]`, ce qui n'est pas `int **`
  - mais c'est compliqué, vous irez voir le doc sur le site si ça vous intéresse vraiment
  - La fonction main accepte les deux prototypes, entre autres.

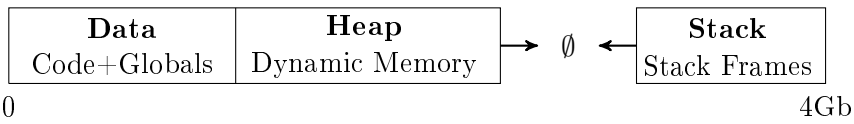
2. Les chaînes de caractères

- Il n'y a pas de type `String`, on fait des tableaux de caractères.
- Pas de métadonnées où ranger la taille. Convention: chaînes terminées par le caractère zéro, noté `'\0'`
- La convention du langage Pascal était de préfixer la taille dans un entier short au début de la chaîne, mais ça limite la taille de la plus grande chaîne (65535 lettres max) et ça consomme un octet de plus par chaîne.
- Fonctions à connaître: `strlen`, `strcpy`, `strcat`, `strcmp`
- Vous comprenez maintenant pourquoi il n'y a pas besoin d'esperluette pour lire une chaîne de caractères avec scanf, et aussi pourquoi on risque les dépassements mémoire à faire cela.
- Une initialisation devrait être `char str[]={'h', 'e', 'l', 'l', 'o', 0};` mais c'est un peu lourd, donc on peut faire `char str[] = "hello";`. Attention à bien garder la place pour le `0` final si on précise la taille.

V) Mémoire dynamique

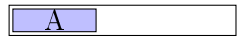
1. Motivation

   - Les tableaux sont de taille statique en C (canal historique)
     - L'écriture `int n; scanf("%d", &n); int tab[n];` est interdite
     - Donc il faut connaître la taille de tous les tableaux à la compilation
     - C'est très pénible, on voudrait pouvoir faire des tableaux de taille connue seulement à l'exécution
     - En fait, les tableaux de taille variable sont autorisés dès C99, mais on a quand même besoin de la solution ci-dessous dans d'autres contextes plus complexes, comme des structures imbriquées
   - Solution: on va le faire à la main, en utilisant le tas:
     - On demande des blocs mémoire quand on en a besoin
     - On les rend après usage



2. L'interface malloc

   - c'est l'approche standard. c'est une bibliothèque; emacs plus hardcore et fait sans ça, `brk()` direc
     - `#include <stdlib.h>`
     - `void** malloc(int size)`: réserve un bloc de `size` octets en mémoire et retourne adresse début
     - `void free(void** p)`: libère (rend) un bloc précédemment alloué

  - `void** realloc(void* p, int size)`: modifie la taille d'un bloc; `/!\` cela peut le déplacer
- Exemple simple.
  - `void *A=malloc(12);` 

  - `void *B=malloc(5);` 

  - `free(A);` 

  - `void *C=malloc(6);` 

  - `C=realloc(C,13);` 

3. Survivre à `malloc`
   - Comme d'habitude en C, il n'y a aucun garde-fou et la moindre erreur mène au SEGFAULT
   - Solution 1: **bonnes pratiques** pour éviter les problèmes (et métaphore du notaire: mémoire = terrain)
     - **Règle #1:** on n'accède qu'à des zones réservées
       * usage avant malloc: on achète le terrain avant de construire, svp
       * usage après free: on arrête d'utiliser ce qu'on a vendu, svp
       * symptômes sinon: SEGFAULT si chanceux, corruption mémoire quelque part sinon
         · On ne peut pas être chanceux sur un *use after free*
     - **Règle #2:** à chaque `malloc` correspond un `free`
       * s'il manque un `free`, alors c'est une fuite mémoire *(memory leak)*
         · Le système pense que la mémoire est encore prise et ne peut la réutiliser
         · Quand y'a qu'un leak, ça va. C'est quand il y en a beaucoup que ça pose problème.
         · Ralentissement *(swapping)*, puis malloc retourne `NULL`, puis l'OOM killer de linux intervient
       * Double `free` du même malloc: on vend deux fois
         · Si le bloc n'avait pas été réalloué, c'est une no-op
         · S'il avait été réalloué [dans un autre module], ça le libère dans le dos de l'autre. Pas cool.
         · `int *A = malloc(12); free(A);`
         · `int *B = malloc(12); free(A);` $\rightsquigarrow$ libère `B`.
         · Les mallocs modernes se suicident quand ils détectent ça. Car oui, y'a plusieurs implem de malloc, et même de la recherche en R&D là dedans. La détection se base sur les métadonnées autour. Implémenter son propre malloc est un exercice instructif, même s'il est dur de prendre les vrais de vitesse
   - Solution 2: il faut **connaître les outils** pour soigner le mal:
     - `valgrind` pour détecter les erreurs à leur source, et pas seulement les défaillances: un outil formidable, simple d'accès et voit 90% des problèmes (seul le

tas est surveillé, pas la pile)

- `gdb` pour explorer l'état de la mémoire: un outil formidable. Vieux mais parfaitement robuste. Pratique comme un tank soviétique.
- C'est l'occasion d'introduire un peu de vocabulaire classique
  - faute: bêtise du programmeur
  - erreur: comportement incorrect (ie, comportement proscrit, ou qui ne fait pas partie de la spec)
  - défaillance: comportement observé $\neq$ comportement attendu (là, ça se voit)

Chapter 5

Maîtriser la programmation C

- En C, on est libre de tout, y compris de coder proprement
- Il faut tout faire, car le compilateur n'aidera en rien (il ne se plaint pas des fautes de goût)

I) Constructions C pour organiser ses données

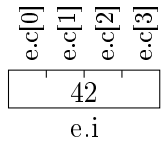
I.1) Déclarer des structures

- Base de la propreté en informatique: ranger ensemble ce qui va ensemble. En C, c'est des structures
- Voir le code de la structure `point` sur la feuille de pompe. `/!\` au ; final. C'est qu'on peut définir une variable de ce type entre l'accolade fermante et le point-virgule
- Exemple d'usage avec la notation pointée
- Initialisation en place: `struct point p2 = {2, 4.2, 3.7};`
- Usage en paramètre ou en retour de fonction $\rightsquigarrow$ copie complète, comme d'habitude
- Possibilité de faire des structures imbriquées, voire récursives.
- Pas d'opérateurs globaux. `memset` et `memcpy` pour mettre à zéro ou copier l'un dans l'autre.

I.2) Déclarer des unions

- C'est exactement comme une structure, sauf que tous les champs d'un union sont au même endroit en mémoire

```
union entier {
  int i;
  char c[4];
};
union entier e;
e.i = 42;
printf("%d %d %d %d",
        e.c[0], e.c[1],
        e.c[2], e.c[3]);
```



- Cela affiche `42 0 0 0` sur mon ordinateur, little endian 64bits.
- **sizeof** de l'ensemble: 4 (comme un int)
- C'est donc assez pratique pour aller explorer la mémoire à l'échelle de l'octet

- Autre usage des unions C: Faire une variable générique, comme en python. Mais il faut stocker le type de l'objet à coté de sa valeur.

I.3) Explorer l'intérieur des octets

- Il est impossible en C de faire une variable scalaire dont la taille soit inférieure à un octet. Le plus petit type est `char` sur un octet.
- Au sein d'une structure, on peut déclarer un champ de bits. C'est un type entier dont on précise la taille.

```c
struct A {
  int bool1;
  int bool2;
};
```

```c
struct B {
  int bool1 : 1;
  int bool2 : 1;
};
```

- La structure A pèse deux fois plus lourd que la structure B: A:8 B:4 sur ma machine. Ca ne fait pas un seul octet pour autant, car le compilateur utilise un entier pour porter le tout.
- Le type de base du champ de bit (celui de `bool1` par exemple) peut être `int`, `signed int` ou `unsigned int`. Cela change l'interprétation des bits du champ
- Le plus grand champ de bit que je peux créer sur mon ordinateur a une largeur de 32 bits. C'est un 64bits, `sizeof(int)=4` ici. Les big ints ne sont pas offerts par le compilateur (sauf `uint128_t`). Il faut utiliser une bibliothèque.
- Si l'un des éléments n'a pas de nom mais seulement une largeur, c'est du remplissage. Mais c'est dangereux de faire ça car la norme n'impose pas au compilateur de faire quelque chose de logique en mémoire. Il est libre de représenter les champs de bits comme il veut.
- Impossible de faire un pointeur vers un champ de bits, bien sûr

I.4) Nommer ses types avec `typedef`

- Nommer ses types est une bonne idée pour rendre le code plus lisible.
- L'habitude est de terminer le nom des types par `_t` comme dans `point_t`
- La sémantique du `typedef` est: Le dernier mot de la ligne devient un synonyme de tout le reste de la ligne. Cette règle admet des exceptions quand on définit un type de fonction ou de pointeur sur fonction, puisque le nom de l'identificateur est avant les parenthèses

I.5) Ranger ses valeurs avec `enum`

- `enum` permet de définir un type n'admettant que certaines valeurs nommées
- L'exemple classique est `enum couleur { pique, coeur, carreau, trefle };`
- Cela va naturellement avec les `switch` cases: Si on fait des switch sur une valeur énumérée (eg le type de la valeur dans l'union), alors on a envie de faire un enum.
- Plus besoin d'un cas `default` qui ne peut pas avoir lieu, et le compilateur nous prévient si on oublie l'un des cas dans le switch (par exemple quand on ajoute des cas)

II) Modèles d'organisation d'un code C

II.1) Procédural

- Programmer procédural en C
  - C'est la façon historique
  - On organise des modules où toutes les fonctions qui vont ensemble sont dans le même fichier
  - L'état est dans des globales cachées (déclarées globales static dans un module)
  - L'interface est dans un fichier d'entête
  - cette approche est *has been* pour de bonnes raisons (mauvaise encapsulation, impossible d'avoir deux instances du module, état caché difficile à tester)

II.2) Orienté objet

- Programmer orienté objet en C
  - État placé dans une structure. Fonctions prennent un pointeur vers l'instance en premier argument
  - Pas si éloigné du python avec son `self`, si on regarde pas de trop près
  - On peut faire de l'encapsulation et du dispatch (avec des pointeurs sur fonction) assez facilement
  - On peut aussi faire de l'héritage, mais faut savoir arrêter et passer au C++ (ou Rust!)
- L'habitude en C est de faire de l'OO à la sauce C
  - La gestion mémoire reste à la charge de l'appelant, qui peut soit faire malloc soit les garder sur la pile.
  - La différence est de savoir si on fait une fonction `new` qui malloc, ou bien `init` qui met la bonne valeur dans les champs
  - Vous allez sans doute voir les deux variantes dans les codes C que vous allez lire

II.3) Le module point en détail

- Alias de type avec `typedef` pour raccourcir par rapport au très valide `struct point*`
- Constructeur, destructeur, copy constructeur; Des fonctions de manipulation
- Le contenu de la structure et l'implem des méthodes est caché, dans un fichier à part
  - Le compilo n'a pas besoin de connaître la structure car on manipule un *pointeur vers* de taille connue
- Le fichier d'entête donne le prototype de ces méthodes
  - Inclus dans le fichier d'implem pour vérifier que le prototype correspond à la vraie définition
  - Inclus dans les fichiers appelants pour annoncer le prototype au compilo
  - Attention, il faut protéger contre la double inclusion (crétin de compilo)

II.4) L'habitude en C

III) Code lisible et propre

- L'IOCCC est l'exemple à ne pas suivre.
- La programmation n'est pas une question de technologie. Il s'agit d'exprimer vos idées de façon précise et efficace. [`http://prog21.dadgum.com/`]
- Dans une large mesure, l'acte de coder est un acte d'organisation. Refactorisation. Simplifier. Comprendre comment supprimer les manipulations superflues ici et là. [`http://prog21.dadgum.com/177.html`]

III.1) Code lisible

1. Commentaires
   - Mauvais commentaires:
     - Les commentaires ne doivent pas paraphraser le code. D'autant que les commentaires sont rarement mis à jour en même temps que le code, donc risque d'obsolescence.
     - Si le code est très mauvais, il faut le réécrire, pas le commenter
   - Les commentaires doivent expliquer le code
     - donner l'intention du code, les présupposés et les invariants
     - surprises lors de l'implémentation et tentatives précédentes (par exemple sur les perfs)
     - expliciter les choix sous-jacents, par exemple sur la valeur des constantes
   - reconnaître les problèmes en donnant des pistes d'amélioration
     - TODO, HACK, FIXME (qqch cassé), XXX (gros problème)
   - Se mettre à la place du lecteur
     - Si le code fait qqch étonnant, il faut documenter la raison de ce choix
     - Un commentaire de fichier doit donner une vue d'ensemble, expliquer la raison de ce découpage du projet.
     - Résumer des blocs de code un peu longs
   - Les commentaires doivent être compacts et informatifs, sans ambiguité
     - Des exemples de paramètres / résultats sont précieux
2. Identificateurs
   - Les identificateurs doivent être bien choisis. S'il faut expliciter dans les commentaires ce que fait un paramètre, peut-être qu'il faudrait plutôt le renommer.
   - Les booléens devraient avoir `is` ou `has` dans leur nom pour être explicite. `disable_x=0` est moins lisible que `use_x=1`
   - Une fonction est un outil pour l'utilisateur. Il faut donc que son nom parle à l'utilisateur, pas seulement à son auteur. GrahamAlgorithm est un nom moins explicite que EnveloppeConvexe
3. Mise en page

- La mise en page est importante pour la lisibilité
- Indentation consistante. Utilisez `clang-format` pour cela
- Faites des petits blocs comme on fait des paragraphes.
  - Réduisez le scope des variables
  - Documentez le bloc

III.2) Code propre

- Si le code est compliqué, il n'est pas digne de confiance et il faut le remplacer par qqch de plus simple si possible. On parle de changer la logique.
- Le découpage du projet en un ensemble bien pensé de fonctions et modules
  - Les ingénieurs informaticiens n'emboutissent pas du métal, mais ils forment des concepts-outils
  - On a tous une taille d'espace de stockage dans le cerveau et on ne peut pas rentrer plus gros. On veut donc comprendre chaque brique en entier, puis comprendre l'assemblage des briques prises comme éléments atomiques
- Dupliquer les choses est la pire idée.
  - Moto classique de programmeur: DRY/SPOT. Don't repeat yourself / Single point of truth
  - DRY = ne jamais dupliquer du code. Si le code doit être corrigé ou modifié, il faudra le faire dans chacune des copies
    * Ne jamais agraver avec CtrlC/CtrlV; tjs chercher à mettre le code en facteur commun pour réduire la duplication de code.
    * Le pire, c'est un code qui a été créé par CtrlC/CtrlV puis les copies ont divergé légèrement, de façon difficile à comprendre mais incompatibles entre elles. L'enfer à relire / comprendre / faire évoluer
  - SPOT = on se débrouille pour que les valeurs aient des noms. Par exemple, au lieu d'écrire 640 partout où on a besoin de la taille de l'écran, on fait une constante nommée en début de programme avec cette valeur, et on l'utilise ensuite.
    * Avantage1: le code est plus lisible ainsi, on comprend le pourquoi des calculs numériques.
    * Avantage2: s'il faut modifier la taille de l'écran, on va avoir un moment très difficile si les constantes numériques n'ont pas été nommées correctement
    * Anecdote: Quand nvidia a voulu faire un driver open source pour rentrer dans le monde linux, mais sans perdre la main sur le code (s'assurer qu'ils étaient les seuls à pouvoir le faire évoluer), ils ont diffusé une version obfusquée où les constantes étaient inlinées (nom remplacé par la valeur) et les premiers calculs déjà faits pour cacher les valeurs. Cela faisait un code profondément difficile à lire/modifier. Ils ont longtemps prétendu qu'il s'agissait effectivement de la "preferred form for modification", jusqu'à ce que la communauté développe un autre pilote par rétroengineering.
- L'objectif est de regrouper ensemble les traitements qui vont ensemble sans fragmenter les fonctionnalités partout dans code (shotgun design)

- Simplifier le flow exécutif
  - C'est pour ça qu'on n'utilise plus de labels et goto en C (même si le compilo nous laisserait faire)
  - On évite les pièges idiots et les surprises pour lecteurs inattentifs
  - Préférer `if (a < b)` à la version à l'envers `if (b > a)`
  - S'il y a if/then, la condition du if ne doit pas avoir de négation
  - Préférez `if (!cond) return;` à un niveau d'indentation supplémentaire
  - Évitez les indentations trop profondes, qui demandent d'avoir bcp de code en tête
- Simplifier les expressions, en nommant les variables intermédiaires
- Eliminer les choses inutiles. Si supprimer quelque chose rend le code plus compact et facile à lire, il faut le faire
  - Les variables et fonctions inutilisées sont vraiment à désherber
  - Supprimer les variables temporaires, les variables de flot (`done`)
  - Inliner les fonctions utilisées à un seul endroit (sauf si le découpage aide à la lecture aide)
  - De toute façon, le compilateur va redécouper notre code en fonction de l'architecture. Rien ne sert d'essayer de l'aider.
- Les variables constantes sont préférables: le lecteur peut avoir confiance dans leur valeur sans vérifier. Mutability seen as an antipattern
- Éviter l'overengineering en n'écrivant que le code immédiatement nécessaire
  - Pas besoin de couvrir un cas qui ne peut pas se produire dans ce projet
  - Il faut supprimer du code devenu inutile, comme on désherbe, car un petit codebase est moins complexe
  - Il faut utiliser les bibliothèques existantes au lieu de réinventer la roue

III.3) Pièges

- Optimisation = mauvaise idée, ca complique et faut bien comprendre pour être sûr que ça gagne. Donc on n'optimise que un code déjà écrit, et après profiling. Jamais à priori.
  - Rule 1: Optimization: don't do it
  - Rule 2 (experts only): Optimization: don't do it yet.
  - Premature optimization is the root of all evil. Knuth.
- Code golf: vouloir le faire en moins de caractères. Cette performance n'aide pas le lecteur
  - On ne veut pas réduire le nombre de lignes, mais le temps pour qu'un lecteur comprenne
- Garder la première version. Il est souvent nécessaire de refactorer son code pour qu'il devienne lisible.
  - On refactore SimGrid depuis 20 ans, et c'est pour ça qu'on arrive à faire autant
  - Refactorer, c'est comme désherber son jardin ou remettre ses idées au jardin
  - Mais on risque de casser des trucs en changeant. D'où l'intérêt de tester.

III.4) Tests

- Un code bien testé est facile à modifier, par exemple pour le refactorer et le rendre plus propre
  - En plus, un code facile à tester est souvent un code plus simple à comprendre car il n'a pas d'état caché
  - Mais attention, les tests sont du code eux aussi. Il faut qu'ils soient lisibles et de qualité. Bien nommés, documentés, etc.
- Vocabulaire:
  - Erreur: comportement incorrect; Faute: cause de l'erreur
  - Tests en boite blanche (écrit après lecture du code source), boite noire (écrit à l'aveugle)
  - Tests unitaire (= par fonction); tests d'intégration (tout le projet en même temps)
  - Tests fonctionnel, tests de performance, tests de régression (éviter que les bugs reviennent lors des évolutions futures)
  - Mocking de composants (abstraction du composant pour tester le reste en isolation), stub (développement d'un étayage temporaire pour faire fonctionner le reste le temps que le vrai code soit développé)
  - Alpha testing (en cours de dev), beta testing (validation après dev mais avant release)
- Générer les bons tests est difficile, demande de l'expérience voire une expertise spécifique
  - Stratégie pour aider les développeurs débutants: Right + BICEPS
    - Est-ce que le résultat est correct? Right
    - B: boundary. Conditions aux limites, la base du test en boite blanche.
      - Valeurs à la limite de l'intervalle possible pour chasser les off-by-one, ou bien au delà pour chasser les dépassements de capacité.
      - Valeurs mal formées. Ou pas de valeur du tout dans la liste.
      - Duplicata, Ordre invalide,
    - I: inverse relationship. Si j'ajoute un élément, je le retrouve.
    - C: cross-check. Vérifier le résultat du test avec une autre méthode, même si l'autre méthode est trop inefficace pour être utilisée en prod.
    - E: Force error conditions. Écrire des torture tests pour le code, comme le Chaos Monkey d'Amazon. Difficile de faire des torture tests permettant non seulement de détecter mais en plus de diagnostiquer / comprendre le pb. Souvent, il y a trop de hasard et de bruit pour comprendre la cause.
    - P: performance, à tester aussi.
    - S: Specific language. Faire un petit langage dédié (DSL) documentant ce que le test fait dans un formalisme compact peut être utile. Réfléchir à ce langage demande de réfléchir à ce qu'il faut tester, et ensuite il est plus facile de tester tous les cas de façon combinatoire.
- Test Driven Development: stratégie très efficace pour écrire le code qui a l'interface dont on a besoin, en écrivant d'abord les tests qui ressemblent au code appelant. Les outils permettent de générer les stubs de code à écrire ensuite.

- Design by contract vs. defensive programming. Il faut vérifier les présupposés, mais pas tout le temps car cela prend du temps. En plus c'est du code écrit, et on pourrait se tromper dans les vérifications de présupposés, ou bien il faudra les maintenir.
- Quelques framework de tests:
  - En C, j'ai fait un petit framework simple pour vous
  - En C++, nous utiliserons Catch2
- Beaucoup de recherche en tests
  - QuickCheck: assertions are written about logical properties that a function should fulfill. Then QuickCheck attempts to generate a test case that falsifies such assertions.
  - fuzzing: faire varier les données en entrée en respectant une syntaxe
  - model checking: faire varier l'ordre des opérations concurrentes

IV) Compilation séparée

- Un seul fichier de 500 000 lignes, c'est dur à lire, naviguer; long à compiler; dur à collaborer
- Mais 5 fichiers séparés, c'est dur à compiler. Il faut savoir quoi recompiler quand. Transitif inclusions.
- Il faut pas essayer de compiler à la main, mais il faut utiliser un outil pour ça
- `make` est parfait pour compiler: il a un arbre et gère tout bien. Mais dur d'écrire un `Makefile` sans oublier la moindre dépendance. Cela demande de lire tous les sources pour ne rien oublier, c'est un travail de robot, pas d'humain.
- `cmake` converti un fichier `CMakeList.txt` à peu près lisible en `Makefile` très bien fait. Mais interdit à l'agreg, donc il faut maîtriser les deux formalismes

V) Conclusion sur le module

- L'objectif était surtout de vous faire comprendre comment fonctionne la machine au plus bas niveau, quand on a meme pas de libc pour nous aider
  - Mais vous avez maintenant une bonne compréhension du langage C, en prime
- Il y a énormément de ressources sur C et UNIX pour aller plus loin. Le plus difficile est de trouver une ressource qui vous convient. La liste suivante est volontairement courte, pour vous permettre d'aller droit au but.
  - Apprendre en C sur OpenClassroom. Particulièrement intéressant pour les débutants. `https://openclassrooms.com/courses/apprenez-a-programmer-en-c`
  - Modern C par Jens Gustedt. Particulièrement intéressant pour les programmeurs confirmés. `https://gustedt.gitlabpages.inria.fr/modern-c/`
  - Polycopié de C de l'ENSIMAG `http://matthieu-moy.fr/cours/poly-c/`.
  - Les wikibooks {Programmation C} et C programming. Il s'agit de livres de référence incomplets, c'est assez frustrant. N'hésitez pas à les améliorer si vous pouvez. `https://en.wikibooks.org/wiki/C_Programming`

- Voici quelques liens supplémentaires, plus amusants qu'indispensables :
  - Clean code Présentation amusante sur pourquoi et comment programmer proprement. `http://www.jeremybytes.com/downloads/slides-cleancode.pdf`
  - Wat, une présentation sarcastique et hors sujet sur les horreurs du javascript. Juste pour rire. `https://www.destroyallsoftware.com/talks/wat`
- A partir de là, on peut partir dans plusieurs directions
  - le réseau, à partir de la semaine prochaine, pour découvrir un gros système cohérent, cf. le poly spécifique
  - le C++, le semestre prochain, pour avoir de l'aide du compilateur pour organiser de gros codes
  - l'architecture, le semestre prochain, pour mieux comprendre comment fonctionne un vrai processeur
  - le système, l'an prochain, pour voir comment fonctionne l'OS, et comment sont implémentés les OS
  - le module HPC, pour parler des performances des ordinateurs