In [4]:

```
from common import utils
import random
u=utils("PDF") # PDF+SOL / PROF / PDF / INTER
```

```
Ce cours a été régénéré le 2018-11-06 19:55:51.513545. Mode sans corrigé. Mode statique.
```

Les données

La manipulation des données par le processeur

Les données d'un système de calcul finissent toujours par être stockées dans un système de *mémoire adressée*. Chaque élément d'information va être stocké dans un système (électronique de nos jours, mais on peut imaginer d'autres choses) qui est *neutre* vis-à-vis de la nature de la donnée stockée. Cela veut dire en particulier que si on lit la mémoire qui va servir au stockage sans connaissance spécifique sur le système qui s'occupe du stockage, on ne va pas y reconnaître des données d'un *type* particulier, mais uniquement des nombres, avec des valeurs uniformes.

De nos jours, toutes les mémoires sont organisées de la même manière : les éléments de stockage sont tous uniformes, de taille 1 octet (soit 256 valeurs possibles), et repérés par leur adresse.

Donc toute donnée stockée doit être convertie, à un moment, en un ou plusieurs octets. Lorsqu'un octet ne suffit pas à stocker une donnée, on lui affectera plusieurs adresses consécutives, et on ne retiendra que la première comme étant l'adresse de la donnée.

Un processeur manipule des données avec une taille fixe. Il y a parfois plusieurs tailles manipulables.

Historique :

- Ordinateurs 8 bits : 1972–1985
- Ordinateurs 16 bits : 1975–1990
- Ordinateurs 32 bits : 1986–maintenant
- Ordinateurs 64 bits : 1992/2003–maintenant

Pour les flottants ou les données graphiques, il y a des circuits spécialisés qui ont des tailles différentes (plus grandes). Les données de taille supérieure doivent être manipulées en plusieurs opérations et ne sont pas des données simples. C'est la taille du *mot-machine*.

Structuration des données

Types de base, types composés

Tous les processeurs sont capables de manier des données répondant à un certain *type*, c'est-à-dire des données assorties d'opérations et de contraintes de taille. Quasiment les seuls types gérés au niveau des processeurs sont des nombres, de différentes tailles et de précision variable. On a de nos jours essentiellement les nombres codés en C2 (sur 1, 2, 4 octets, parfois 8), les nombres codés en NAT (mêmes tailles), les adresses (qui sont aussi des nombres, donc sont gérés comme eux) et les nombres codés en virgule flottante. Et c'est à peu près tout.

Les langages de programmation ont évidemment besoin de gérer d'autres types de données. Certains types de données sont ainsi gérées dans la construction du langage, par des équivalences avec des opérations simples que le processeur est capable de faire. C'est le cas, par exemple, des caractères. L'ensemble de tous les types de données que sont capable de gérer un langage de cette manière, avec une correspondance directe sur des opérations du processeur, est appellé l'ensemble des *types simples*. Outre les nombres proposés par le processeur, on trouve donc souvent les *booléens* et les *caractères*. Ils sont peu nombreux.

Les programmes, eux, nécessitent des types beaucoup plus variés que ce que peuvent offrir les langages de programmation. Ils sont capables donc de créer des structures beaucoup plus complexes par *assemblage* de types, que ce soit des *types simples* ou des types plus complexes, eux-même le résultat d'un assemblage.

On distingue essentiellement trois formes d'assemblage :

- **L'assemblage homogène** de taille fixe (appelé aussi vecteur, ou tableau) ou taille arbitraire (liste)
- **L'assemblage hétérogène** de taille fixe (appelé aussi structure) ou arbitraire (ensemble de propriétés, dictionnaire)
- **L'assemblage par superposition** ou union qui permet de mettre dans une donnée plusieurs types qui occupent la même place (et donc pas plusieurs types séparés — ce serait alors juste une structure).

Tous les langages n'ont pas la même expressivité de ce point de vue (par exemple en Python la structure de dictionnaire fait partie du langage, pas en C), mais de toute façon, toutes les opérations sur les structures doivent pouvoir être transcrites dans le langage restreint du processeur — et donc, dès qu'on peut utiliser les possibilités du processeur, on peut être aussi expressif que dans n'importe quel autre langage.

L'assemblage homogène

L'assemblage homogène de données permet de regrouper des donnés qui sont sémantiquement sur la même valeur. Par exemple, on peut regrouper les points d'un ensemble de données statistiques, les moyennes d'un élève, les tailles des membres d'un groupe... On peut même parfois attribuer des valeurs particulières si ça peut avoir un sens. Par exemple, le premier élément représente un but à atteindre, et les valeurs suivantes des essais successifs ; mais dans l'absolu, il serait meilleur de séparer les données qui n'ont pas la même valeur (l'ordre à un sens, spécialiser certaines valeurs et pas d'autres est plus problématique).

On distingue principalement deux sortes d'assemblage homogène : l'assemblage de taille fixe, appelé souvent tableau (en anglais *array*) ou vecteur (lorsque l'on a des nombres), et celui de taille variable, la liste (anglais *list*). Les deux présentent des caractéristiques bien distinctes : il est notamment plus facile d'accéder à un élément quelconque du tableau, alors que la liste se parcourt de proche en proche. Il est souvent plus facile d'avoir la taille d'un tableau, ce qui va faciliter sa copie en mémoire. On distinguera parfois les tableaux de taille arbitraire, mais fixé à l'écriture du programme, des tableaux de taille fixe mais dont la taille n'est connue qu'à l'exécution. Par exemple, la différence entre un tableau de dimensions de paquet (longueur, largeur, hauteur) est toujours 3, alors que l'on peut faire un tableau de notes pour un contrôle (taille fixe, mais dépendant du nombre d'élèves au départ), ce qui est encore différent d'un tableau contenant tous les élèves qui passent par un même endroit (on ne connaît absolument pas le nombre au départ). Bon nombre de bugs sont d'ailleurs dus à l'utilisation d'un tableau du deuxième type à la place d'une liste, en supposant que la taille prévue au départ sera assez grande.

L'assemblage homogène le plus répandu est la chaîne de caractères. Vu comme un assemblage de type liste de caractères, il permet de manipuler des données textuelles, qui revêtent une grande importance.

Dès que l'on fait un tableau, se pose la question de la détection de la longueur et de la terminaison des données. Si les listes sans adressage direct d'un élément sont souvent composés de petits éléments liés entre eux, les tableaux ou vecteurs sont en effet des blocs de mémoire contigus. La copie d'une donnée (qui est une opération fréquente) nécessite donc de bien savoir où commence (facile) et où se termine (parfois plus compliqué) la donnée à copier.

Il est à noter qu'on indexe les données du tableau ou de la liste (quand il y a un accès direct) par un entier. La plupart des langages suivent l'exemple du C et commencent les indices à 0, mais certains commencent à 1 (le 1er élément est alors le numéro 1).

La structure de dictionnaire, qui indexe par des mots, se rapproche plus de la structure suivante (notamment parce que le parcours du dictionnaire peut être rendu plus complexe).

L'assemblage hétérogène

Il s'agit de pouvoir décrire les propriétés attachées à un même concept, chacune de ses propriétés étant potentiellement d'un type différent. Par exemple, pour un individu, on pourrait vouloir donner son nom (chaîne de caractères), son numéro INSEE (un nombre), son salaire moyen (un flottant), le dernier trajet qu'il a effectué à vélo (une liste de points GPS+temps), etc. Ces données, structurées, forment ce qu'on appelle une *structure*. Il y a plusieurs champs qui portent tous des noms. Lorsque le nombre de champs est fixe et connu à l'avance, on a bien la *structure* proprement dite. Lorsque le nombre de champs est arbitraire, on a alors affaire a un *dictionnaire*.

Il faut remarquer que l'assemblage est *a priori* hétérogène, mais qu'on peut très bien avoir le même type partout. Par exemple un nombre complexe est formé a priori de deux nombres flottants, qui ne jouent pas le même rôle, donc on le mettra plutôt sous forme d'un assemblage hétérogène que d'un tableau à deux éléments... même si on peut décider le contraire.

Chacune des propriétés de la structure est appelé un *champ*.

La superposition

Bien que beaucoup de langages n'aiment pas ce genre d'assemblage, ça consiste à réserver une place... pour quelque chose. C'est souvent source de confusion, et ne se présente que dans des structures plus complexes.

Par exemple, on pourrait définir deux types de complexe : une structure `(re,im)` d'une part avec partie réelle et imaginaire, et une structure `(module,angle)` pour la représentation polaire des complexes. Ces deux représentations sont équivalentes, et on peut très bien ne convertir (pour les opérations) qu'au moment où on en a besoin, et pas dès qu'on saisit un complexe. Du coup, on pourrait avoir un type

```
COMPLEXE = SUPERPOSITION(COMPLEXE_RECTANGULAIRE,COMPLEXE_POLAIRE)
COMPLEXE_RECTANGULAIRE = STRUCTURE(REELLE,IMAGINAIRE)
COMPLEXE_POLAIRE = STRUCTURE(MODULE,ANGLE)
```

Mais dans ce cas, comment fait l'ordinateur pour savoir si les deux nombres stockés dans l'union représentent des coordonnées rectangulaires ou polaires ? Il faut ajouter un déterminant à ça :

```
COMPLEXE = SUPERPOSITION(STRUCTURE(SOUS-TYPE,REELLE,IMAGINAIRE),STRUCTURE(SOUS-TYPE,MODULE,ANGLE))
```

ou même en moins fragile :

```
COMPLEXE = STRUCTURE(SOUS-TYPE,SUPERPOSITION(STRUCTURE(REELLE,IMAGINAIRE),STRUCTURE(MODULE,ANGLE)))
```

avec `SOUS-TYPE` qui est un champ qui indique quelle est la nature des nombres stockés.

Lorsque les deux membres d'une superposition sont de taille différentes, il faut réserver la taille suffisante pour la plus grande des deux.

Contraintes techniques

La principale contrainte technique est l'*alignement*. La deuxième est la taille du *mot-machine*. La troisième est la taille du *mot-mémoire*. Le mot-mémoire est la taille de ce qui circule entre la mémoire principale et la mémoire cache (proche du processeur, plus rapide), et est souvent identique au mot-machine.

Alignement

Les processeurs présentent des contraintes techniques pour l'adressage des données. Une opération sur une donnée de type simple en mémoire doit se faire avec une adresse multiple de sa taille. Les types de taille supérieure au mot-machine sont de toute façon manipulée en morceaux indépendants, et doivent donc être alignés sur la taille du mot machine.

> **Exemple :** Sur une machine 32 bits, un `int` (4 octets) ne peut pas commencer à l'adresse 0x00000002. Il commence soit à l'adresse 0x00000000, soit 0x00000004. Par contre, un `short int` de 16 bits pourra commencer à cette adresse.

Alignement et données composées

L'alignement doit être respecté pour toutes les composantes de la donnée composée, quel que soit le type d'assemblage. L'adresse de la donnée composée est l'adresse de la première composante, et chaque composante doit être alignée correctement vis-à-vis de sa propre taille.

> **Exemple :** Sur la même machine, une structure composé d'un `int` et de deux `short int` (dans ce sens) est correctement alignée si elle commence sur une adresse multiple de 4. Par contre, la même structure organisé en un `short int`, un `int` et un `short int` ne sera pas convenablement alignée pour le `int` du milieu si l'adresse initiale est un multiple de 4.

Pour résoudre le problème de l'alignement dans les structures, on rajoute aux structures des blancs (octets inutilisés) qui permettent de recaler toujours les alignements.

> **Exemple :** Toujours avec le même exemple, si on insère un blanc de 2 octets après chaque `short int`, on aura un alignement automatiquement correct de l'`int` au milieu. La structure sera alignée sur un multiple de 4 ainsi que tous les membres.

In [2]:

Activité : Type simple ou composé ?

Répondez sur chacun de ces éléments par :

- **b** pour un type simple (basique)
- **s** pour un type composé (structure)
- **v** pour un type composé (vecteur, tableau)
- **l** pour un type composé (liste)
- **sv** pour un type composé, qui peut être vu comme un tableau ou comme une structure

In [ ]:

Description de structures de données en C

Dans le langage C (un langage très proche des possibilités proposées par le processeur), il est possible de représenter un certain nombre de ces types d'assemblage facilement.

En C, la déclaration d'une variable se fait de la façon suivante : `typeDeLaVariable nomDeLaVariable = valeurInitiale ;` (les points-virgules séparent les instructions en C, et non des retours à la ligne comme en Python). L'initialisation est facultative — dans le langage ; elle reste obligatoire pour assurer un résultat correct. On peut déclarer plusiers variables portant sur un même type en les séparant par des virgules (mais ce n'est pas toujours plus clair).

Les nombres

Les nombres sont représentés par des entiers `int` (codage C2) ou des entiers non-signés `unsigned int` (codage NAT). Ils sont représentés sur plusieurs octets comme déjà illustré dans ce cours. Le mot clé `int` peut-être remplacé par `short` ou `short int` pour des données sur 2 octets et `char` sur un octet, ainsi que `long` ou `long int` pour des données plus grandes sur les processeurs 64 bits. `float` (4 octets) et `double` (8 octets) sont utilisés pour les nombres en virgule flottante.

Exemple :

```
int a = 120000;
short int b=-1800;
char c=-10;
unsigned int d;
unsigned char e = 0xE3;
float f = -200.0;
double g = 10000000000000000000.0;
```

L'initialisation doit se faire sur une valeur fixe, pas une valeur calculée (par une fonction par exemple).

Le nombre d'octets dépend du modèle de programmation employé par le processeur :

Modèle	char	short	int	long	adresse	long long	Qui ?
32 bits	8 bits	16 bits	32 bits	32 bits	32 bits	—	Processeurs 32 bits
LP64	8 bits	16 bits	32 bits	64 bits	64 bits	—	Processeurs 64 bits (sauf ci-dessous)
ILP64	8 bits	16 bits	64 bits	64 bits	64 bits	—	Quelques exceptions
LLP64	8 bits	16 bits	32 bits	32 bits	64 bits	64 bits	Microsoft

Les caractères

Les caractères sont simplement des nombres, en C, mais il est possible d'utiliser la table ASCII pour initialiser un caractère.

```
char c='A';
c=c+1; /* c vaut maintenant 'B' */
```

Les chaînes de caractères sont un type construit par assemblage, mais tellement fréquent qu'il existe une syntaxe spécifique pour les rentrer :

char *chaine = "Toto et ses parents";

Nous reviendrons sur cette présentation plus tard, car beaucoup d'éléments sont cachés.

Les tableaux (taille fixe)

Un tableau est un assemblage *homogène* par répétition de plusieurs éléments de même type. Il est déclaré par `type nomDeVariable[quantité] = { valeur_0, valeur_1, ..., valeur_(n-1) };` .

Exemple:

```
int tab[6]={1,2,4,8,16,32};
float point[]={0.0,1.0,3.0}; /* Initialisation, on peut omettre la quantité */
double notes[2]; /* Attention, pas initialisé */
double error[]; /* Déclaration illégale : ni initialisation ni quantité */
```

La raison d'être de la quantité est de pouvoir réserver le bon nombre d'octets ! La mémoire est allouée au moment de la déclaration, et réutilisée lorsque la déclaration n'a plus cours.

On peut aussi faire des tableaux de types plus complexes, qui auraient été définies. Par exemple, on peut déclarer :

```
struct complex { float re; float im } points_du_plan[6];
```

Le calcul de l'adresse d'un élément est simple dans un tableau. Un tableau est en fait assimilé à l'adresse de son premier élément (numéroté 0), et pour obtenir l'adresse de l'élément suivant, il suffit d'ajouter la taille d'un élément. Dans l'exemple ci-dessus, si un `int` fait 4 octets, et que `tab` est rangé à l'adresse 0x8000, `tab` vaut alors 0x8000, tab+1 vaut 0x8004, tab+2 vaut 0x8008.

Si on veut accéder aux données d'un tableau, il suffit d'écrire `tab[n]` (n un entier) qui demande au processeur à récupérer la valeur qui est stocké à l'adresse `tab` +n fois la taille d'un entier.

Les structures (assemblage hétérogène de taille fixe)

Une structure permet de juxtaposer des données, éventuellement de type différent. Chaque sous-donnée a un nom et s'appelle un *champ* de la structure. Par exemple, un nombre complexe peut être défini comme ayant deux champs : partie réelle, et partie imaginaire, qui sont ici tous les deux des nombres flottants.

```
struct complexe {
    float re;
    float im;
} monComplexe;
```

Toutefois, cette définition devrait être répété à chaque fois qu'on utilise la structure. Du coup, il est possible de *nommer* un nouveau type avec l'instruction `typedef` . Cela permet ensuite d'utiliser le nouveau nom plutôt que de répéter la structure à chaque fois.

```
typedef struct complexe {
    float re;
    float im;
} Complexe;
/* Ensuite on peut utiliser la déclaration */
Complexe monComplexe;
```

Exemples cumulés

Regardez à l'URL https://goo.gl/56G6Do (https://goo.gl/56G6Do) comment les valeurs en C sont représentées dans la mémoire.

> **Note pour plus tard:** sur comment lire toute déclaration faite en C, on trouve des renseignements utiles à l'adresse http://www.unixwiz.net/techtips/reading-cdecl.html (http://www.unixwiz.net/techtips/reading-cdecl.html). Pour mémoire : on part de la variable, on va à droite quand on peut, on va à gauche sinon, jusqu'à arriver au type (qui est tout à gauche). On a alors construit une phrase qui permet de décrire le type complet, y compris les morceaux les plus compliqués.

Les relations entre structures

Des fois, on veut indiquer comme une partie d'une structure une autre structure. Il y a deux solutions possibles :

- l'inclusion
- la référence

La première peut se faire facilement, à condition qu'on obtienne à la fin une structure de taille fixe (toujours dans le but de pouvoir la copier facilement), ce qui exclut les listes, par exemple.

```
typedef struct complexe {
    float re;
    float im;
} Complexe;
struct transformationAffine {
    Complexe affine;
    Complexe transform;
}
```

Mais des fois, on veut pouvoir faire une référence à une structure de taille variable. Dans ce cas, on ne va pas inclure la donnée, on va inclure une référence à la donnée sous la forme de son adresse. On pourra accéder à la deuxième structure en connaissant son adresse.

Pour indiquer une adresse, on rajoute devant la variable le symbole `*` qui se lit `typeC *var est var est un pointeur sur une donnée de type typeC` . On peut cumuler les structurations comme ça pour empiler les définitions.

```
int *a; /* a est une adresse, et à cette adresse on trouve une donnée de type int */
float *b; /* b est une adresse, et à cette adresse on trouve une donnée de type float */
int *c[6]; /* c est un tableau de six adresses, et à chacune de ces adresses il y a un entier */
int **d; /* d est une adresse, et à cette adresse on trouve une donnée qu'on appelle (*d)
            Cette donnée (*d) est une adresse et à cette adresse, il y a un entier */
```

In [ ]:

Les problèmes d'ordre des octets

Lorsqu'on affecte une donnée à un paquet de plusieurs octets, il n'a pas encore été précisé quel octet allait dans quel adresse. La raison, c'est que des choix différents ont été faits historiquement, et perdurent encore. C'est le choix de l'*endianness*.

Le choix de mettre les octets dans l'ordre où ils sont rencontrés dans la donnée dans des adresses croissantes (donc premier octet de la donnée = plus petite adresse) s'appelle *big-endian*. Le choix inverse (donc pour un nombre, de mettre les petits poids dans les petites adresses), s'appelle *little-endian*.

Le choix le plus fréquent était de faire des processeurs *big-endian*, mais Intel (leader du marché des processeurs) a choisi pour ses modèles emblématiques (le 286, puis le 386, puis les Pentium, et tous les autres maintenant) le *little-endian*. Les processeurs ARM, PowerPC (sauf G5) et MIPS sont capables de fonctionner dans l'un ou l'autre mode (avec un mode par défaut).

Le problème de place des octets d'une donnée à l'intérieur de la zone réservée n'est pas un problème tant qu'on n'y accède que par un seul processeur (qui, de fait, va les ranger dans le bon ordre pour lui, et les lire dans le bon ordre). Le problème existe dès qu'on commence à faire de la communication avec d'autres processeurs (où il faut alors décider d'un sens). C'est le cas notamment d'un bon nombre de protocoles réseaux qui fixent le sens de transport des octets (en plus, à plus bas niveau, de fixer le sens des bits à l'intérieur d'un octet).

In [ ]:

In [ ]:

In [ ]:

In [ ]:

La compression

La compression permet d'économiser de la place sur les disques et en mémoire. Elle pénalise l'accès direct aux données. Elle se paye par des calculs de décompression pour y accéder. En gagnant de la place sur le disque dur, elle permet d'accélérer le traitement des données, la lecture depuis le disque étant beaucoup plus lente que les calculs pour décompresser en général.

L'archivage

L'archivage consiste à transformer toute une hiérarchie de fichiers en un fichier unique. Ce n'est pas de la compression mais c'est souvent couplé à une ou plusieurs méthodes de compression.

- Utilitaire tar sous Unix, programmes commerciaux zip ou rar.

Les formats classiques d'archivage

- gzip utilise le codage de Lempel-Ziv (extension usuelle : gz)
- zip utilise le codage de Lempel-Ziv et de Huffman par dessus (extension usuelle : zip, mais aussi d'autres formats : jar, odt)
- rar (logiciel commercial) utilise le codage de Lempel-Ziv avec des techniques de prédiction PPM par dessus (extension usuelle : rar)

La compression RLE

La compression RLE (pour Run Length Encoding) est une des compressions les plus simples : on transforme une suite de symboles en une suite de paires (nombre de symboles à répéter – symbole). Par exemple 00000011110011 se code 6(0)4(1)2(0)2(1).

Si on alterne uniquement entre des 0 et des 1 (codage d'une suite binaire), on peut alors omettre les 0 et les 1 : 6422. Si ensuite on note chaque longueur en NAT3, on obtient donc 101100010010, qui ne fait plus que 12 bits, au lieu des 16 initiaux.

Il peut mener à une chaîne plus grande que l'originale (par exemple pour 010101 : 1(0)1(1)1(0)1(1)1(0)1(1), soit au final 001001001001001001 qui est plus long que l'original). Le compte est souvent codé sur une taille fixe (un octet si on a des chaînes assez longues, quelques bits sinon). Lorsqu'on dépasse, on peut découper en plusieurs *runs* de longueur maximale (sauf le dernier). Exemple : 255(0)45(0) pour un chaîne de 300 fois 0. En binaire, on autorise des *runs* de longueur nulle pour marquer qu'il n'y a pas d'alternance : 255(0)0(1)45(0), soit au final 111111110000000000101101 (ou encore en hexadécimal `FF 00 2D` pour compresser une chaîne qui au départ était `0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00` . C'est plus court (mais ce n'est pas une chaîne très intéressante).

In [ ]:

In [ ]: