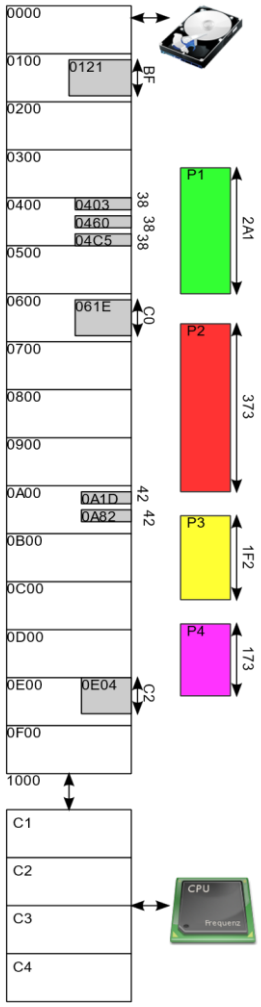
L'objectif du TD est de comparer différentes gestions de mémoire virtuelle.

Nous partons du schéma suivant : un disque dur échange avec une mémoire vive qui échange avec une mémoire cache qui échange avec le processeur. Il ne serait pas raisonnable de travailler des Go de données. Nous partirons sur une taille fictive de mémoire vive de 4Ko. La mémoire cache est constituée de 4 pages indépendantes de 256 octets. Nous noterons les adresses et taille en hexadécimal (c'est plus facile). Il y a déjà des processus chargés en mémoire. Voici la liste des zones occupées :

début	fin	taille	(libre)
0121	01DF	BF	(01E0)
0403	043A	38	(043B)
0460	0497	38	(0498)
04C5	04FC	38	(04FD)
061E	06DD	C0	(06DE)
0A1D	0A5E	42	(0A5F)
0A82	0AC3	42	(0AC4)
0E04	0EC5	C2	(0EC6)



1 Placement de processus

Nous voulons placer 4 processus :

id processus	taille
P1	2A1
P2	373
P3	1F2
P4	173

Le 1er exercice consiste à placer des processus

✓ Placez les nouveaux processus dans une mémoire non virtuelle suivant le format :

id processus	@ mémoire	taille
P1		2A1
P2		373
P3		1F2
P4		173

✓ Que se passe-t-il ?

✓ Placez les nouveaux processus dans une mémoire segmentée et donnez la liste des segments suivant le format :

id processus	@ processus	@ mémoire	taille
P1	0000	0000	121
P1	0121		

id processus	@ processus	@ mémoire	taille

✓ Placez les nouveaux processus dans une mémoire paginée et donnez la liste des pages suivant le format

id processus	@ processus	n° page	taille
P1	**00**00	**0**	100
P1	**01**00		

2 Calcul d'adresse mémoire

✓ Donnez l'adresse mémoire non-virtuelle correspondant à l'adresse processus.

id proc.	P1	P1	P2	P2	P2	P3	P4
@ proc.	**0000**	**0140**	**0000**	**0040**	**00C0**	**0000**	**0000**
@ seg.							
@ off.							
@ mem.							

✓ Donnez l'adresse mémoire segmentée correspondant à l'adresse processus (avec le décalage dans le segment).

id proc.	P1	P1	P2	P2	P2	P3	P4
@ proc.	**0000**	**0140**	**0000**	**0040**	**00C0**	**0000**	**0000**
@ mem.							

3 Calcul d'adresse processus

✓ Donnez le nom et l'adresse processus correspondant à une mémoire non-virtuelle.

@ mem.	0100	0300	0500	0900	0B00	0F00
id proc.						
@ proc.						

Donnez le nom et l'adresse processus correspondant à l'adresse mémoire segmentée.

@ mem.	0100	0300	0500	0900	0B00	0F00
@ seg.						
@ off.						
id proc.						
@ proc.						

✓ Donnez le nom et l'adresse processus correspondant à l'adresse mémoire paginée

@ mem.	0100	0300	0500	0900	0B00	0F00
id proc.						
@ proc.						

4 Comparaison

✓ Combien de mémoire reste-t-il pour chaque type de mémoire ?

- No-virtuelle
- segmentée
- paginée

5 Présentation des langages C/C++

Le langage C a été développé par Denis Ritchie en 1972 dans les laboratoires Bell pour

permettre l'écriture du système Unix. C'est un langage impératif de bas niveau. Il permet de faire le lien avec les adresses mémoires, les registres du processeur, les circuits d'entrée/sortie entre autres. Sa syntaxe à été reprise par de nombreux langage comme C++, Java ou PHP.

C++ peut être vu comme une extension au C en apportant la notion d'objet. Toute la syntaxe C est utilisable en C++.

Le matériel que nous utiliserons peut-être programmé en C. Aussi nous profitions de cette occasion pour une courte introduction à ce langage.

L'objectif de ce TP n'est pas d'apprendre le langage, mais de pouvoir le comprendre. On pourrait dire que l'objectif et de pouvoir lire du C++ sans forcément être capable de l'écrire.

Les limitations de l'environnement de développement est que l'on ne peut pas bénéficier de toute la richesse des bibliothèques C/C++. En particulier, il n'y a pas de classe : string, vector, map… Nous présenteront des substituts au prochain TP sur lesquels nous feront des exercices évalués.

Donc, il n'y aura pas de rendu cette semaine, car ce TP3 présente les rudiments du C. Profitez-en pour prendre de l'avance sur ce langage. Le TP4 montrera quelques structures de données pratiques String, R204Vector, R204Map.

Aujourd'hui, nous commençons par un « hello word » puis la présentation de la syntaxe et des exercices de compréhension. Nous utiliserons C++ (pour sa simplicité de manipulation des chaînes et des entrée/sortie) mais nous ne définirons pas de classe nous mêmes.

6 Hello Word

Il faut commencer par installer un compilateur (celui de GNU).

```
apt-get install g++
```

Voici un exemple de programme que l'on enregistre dans helloWorld.cpp :

```
#include<iostream>

// using namespace std; // pour supprimer le préfixe std::
int main() {
    std::cout << "Hello, new world!"<< std::endl;
}
```

La compilation s'effectue comme suit :

```
$ g++ helloWorld.cpp
```

L'exécution du programme se fait :

```
$ ./a.out
Hello, new world!
```

7 Documentation

Commençons par indiquer quelques liens de confiance :

```
https://fr.wikipedia.org/wiki/C_(langage)
https://fr.wikipedia.org/wiki/C++
https://cplusplus.com/
https://en.cppreference.com/
```

Pensez à consulter ces sites lorsque par exemple vous hésitez sur les paramètres d'une fonction.

8 Les instructions

Pour vous rassurer nous allons commencer par les instructions. Elles vous seront familières puisqu'à l'origine de nombreux autres langage.

8.1 Les blocs

Ils sont mis entre accolades. La portée des variables commence au moment où elles sont déclarées jusqu'à la fin du bloc. Les variables sont également utilisables dans les blocs imbriqués (sauf si elles sont masquées par une nouvelle définition).

```
{
  /* instructions */
  cout << "bonjour" << endl;

  /* déclarations */
  int i = 0;
  {
```

```
    double i = 0;
    cout << "i: " << i << endl;
  }
  cout << "i: " << i << endl;
}
```

8.2 Les conditions

8.2.1 « if », « else », « else if »

Il ne peut y avoir qu'une instruction dernière une condition. Mais comme un bloc est une instruction, nous pouvons regrouper plusieurs actions suivant une condition.

```
{
  if (!a)
    x = 1;
  else if (b) {
    y = 2;
    z = tan (x);
  } else if (c >= 8)
    x = y = 6;
  else {
    affiche ("je cherche\n");
    x = cos (z);
  }
}
```

8.2.2 « switch », « case » et « default »

Attention, l'exécution se poursuit dans le « switch » tant qu'il n'y a pas de « break ». Aujourd'hui les compilateurs avertissent de ces situations.

```
{
  switch (c) {
  case '0' :
  case '1' :
    /* ... */
  case '9' :
    printf ("un chiffre\n");
    break;
  case 'a' :
    /* ... */
  case 'z' :
    printf ("un caractère\n");
    break;
  case '\n' :
  case '\t' : {
    int i = c;
    cout << "ponctuation de valeur " << int (i) << endl;
    break;
  }
  default :
    cout << "Je ne sais pas" << endl;
  }
}
```

8.3 Les boucles

Si l'on considère que le « goto » n'existe pas (non, je ne viens pas d'évoquer son existence) et que l'on mette de côté la récursivité, il n'existe que trois manière de faire des boucles.

8.3.1 « while »

Boucle dans que la condition est vraie, mais réalise l'opération au moins une fois. Exemple d'utilisation : lire un fichier tant qu'il n'est pas vide.

```
  char cstr[] = "Hello";
  int k = 0;
  while (char c = cstr[k++])
    std::cout << c;
  std::cout << '\n';
```

8.3.2 « do »

Exemple d'utilisation : lecture d'une réponse et boucle tant que la réponse n'est pas correcte.

Pour des questions de lisibilité, ne placez pas des branchements « case » dans des boucles.

Le code suivant fonctionne, mais n'est pas

```
void
```

```
mauvaisExemple (char *s, char *t, int nb) {
  int i = (nb + 7) / 8;
  switch (nb % 8) {
  case 0:  do { *t++ = *s++;
  case 7:       *t++ = *s++;
  case 6:       *t++ = *s++;
  case 5:       *t++ = *s++;
  case 4:       *t++ = *s++;
  case 3:       *t++ = *s++;
  case 2:       *t++ = *s++;
  case 1:       *t++ = *s++;
          } while (--i > 0);
  }
}
```

8.3.3 « for »

Cette boucle permet d'expliciter les conditions initiale et d'incrément.

```
  for (i = 0; i < 1000 && getchar () != EOF; i++)
    /* rien */;

  /* boucle infinie */
  for (;;)
    ;
```

8.4 Les interruptions de séquence

8.4.1 « break [tag] »

L'instruction permet d'interrompre une boucle ou un « switch ».

```
 sentence: for (; readline (); ) {
  word: for (; nextWord; ) {
      switch (token) {
      case x: break;
      case y: break word;
      case z: break sentence;
      case t: return;
      }
    }
  }
```

8.4.2 « continue [tag] »

L'instruction ne peut être utilisée que dans une boucle.

8.4.3 « return [value] »

L'instruction ne peut être utilisée que dans une fonction C, une méthode C++ ou une fonction lambda.

8.5 Les déclarations

Le type est en début de ligne et elle se termine par un point-virgule. Les identifiant sont séparés par des virgules.

```
int x,  y; // x et y sont des entiers
bool a,b;  // a et b sont des booléens
```

« * » signifie pointeur, « & » référence, « [] » tableau et des « () » après l'identifiant veut dire fonctions. L'ordre de priorité est « () » puis « [] », « & » et « * ». Il est possible de parenthéser pour lever une ambiguïté.

```
int *i;           // pointeur sur un entier
int *f ();        // fonction qui retourne
                  //   un pointeur sur un entier
int (*pf) ();     // pointeur sur une fonction qui retourne
                  //   un entier
int *(*tf[]) (void (*pg) (int t[]);
                  // tableau de pointeur sur des fonctions
                  // qui retourne des pointeurs d'entier et
                  // qui prennent en paramètre de pointeurs
                  // de procédures prenant
                  // des tableaux d'entier en paramètre
```

Ne feront pas aussi compliqué. C'est simplement pour vous montrer que l'on peut tout exprimer.

9 La définition de fonctions

La définition d'une fonction ressemble à une déclaration, mais au lieu du point-virgule de fin, on place le corps de la fonction entre accolades « {} ».

10 Les directives de compilation

Il existe une sorte de « traitement de texte » avant la compilation qui permet la substitution de chaîne de caractère modèle ainsi que le masquage ou non de portion de code. On appelle le ce traitement le préprocesseur.

La ligne suivant défini une constante. Les 3 lettres VAR seront remplacées par le texte qui suit (il n'y contrôle de syntaxe à ce niveau)

```
#define VAR val
```

On peut définir des macros. Les parenthèses doivent être collées au nom de la macro (c'est le seul cas dans le langage où les espaces comptent). Comme vous le voyez dans l'exemple suivant (à ne pas suivre) la syntaxe sera vérifié après remplacement.

```
#define M(x,y) x + y /
```

Il est possible de mettre des directives de compilation optionnel avec « # » suivit des instructions de conditions.

```
#ifdef DEBUG
int debug = 0;
#elif defined (SCREEN)
int width (100);
#else
int draw () {}
#ifndef
```

L'inclusion de fichier de déclaration se fait par

```
#include <fich1> // recherché dans les bibliothèques standards
#include "fich2" // recherché dans le code local
```

Pour éviter les définitions multiples, il faut toujours encadrer des fichiers en-têtes avec une condition d'unicité.

```
#ifndef __mon_nom_de_fichier_h_
#define __mon_nom_de_fichier_h_
/*
      ici toutes les déclarations du fichier en-tête
*/
#endif
```