Les types énumérés

On peut aussi définir des types qui ont un nombre fini de valeurs (ex: jours de la semaine, couleurs primaires, etc.)

```
type nom  = enumeration (valeur1, ... valeurn)
```

Un exemple:

```
type couleur = enumeration (Vert, Rouge, Bleu)
...
variables c: couleur
...
c <- Rouge
```

En C++, cela s'écrit

```
typedef enum nomdetype { valeur1,...,valeurn};
```

Récursion

Dans le pseudolangage et dans C++, il est possible de faire appel, dans le corps d'une procédure ou fonction, à la procédure ou fonction *même* que l'on est en train de définir. Cela s'appelle une définition *recursive*, et elle permet d'écrire des programmes qui sont plus proches des définitions mathématiques par récurrence.

Par exemple, pour calculer la fonction factorielle définie mathématiquement comme:

$$factorielle(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * factorielle(n-1) & \text{sinon} \end{cases}$$

on peut écrire la fonction

```
fonction factorielle (n: un entier): entier
debut factorielle
si (n = 0) alors retourner 1
        sinon retourner (n*factorielle(n-1))
fin factorielle
```


Problème des tours de Hanoi, et récursion

On doit déplacer N disques de la gauche a la droite en passant par le milieu, sans jamais poser un disque plus grand sur un disque plus petit. Par exemple, voila une partie gagnante pour 3 disques.

```
  Gauche    Milieu    Droite       Gauche    Milieu    Droite
     o         |         |            |         |         |
    ooo        |         |            |         o         |
   ooooo       |         |            |        ooo      ooooo
-----------------------------   ----------------------------
     |         |         |            |         |         |
    ooo        |         |            |         |         |
   ooooo       |         o            o        ooo      ooooo
-----------------------------   ----------------------------
     |         |         |            |         |         |
     |         |         |            |         |        ooo
   ooooo      ooo        o            o         |       ooooo
-----------------------------   ----------------------------
     |         |         |            |         |         o
     |         o         |            |         |        ooo
   ooooo      ooo        |            |         |       ooooo
```

Problème des tours de Hanoi: la solution en pseudo

On peut écrire facilement une solution avec une procédure récursive:

```
procedure deplace(valeur n: int, valeur orig,dest,tmp: tour)
            /* deplace une tour de n disques de orig a dest avec pivot tmp */
         debut
            si n = 1
            alors imprime "deplacer disque de" orig "a" dest
            sinon
                   deplace(n-1, orig, tmp, dest)
                   imprime "deplacer disque de " orig " a " dest
                   deplace(n-1, tmp, dest, orig)
         fin
programme Hanoi
type      tour = enumeration (Gauche, Droite, Milieu)
variables ndisques: int
debut Hanoi
        lire ndisques
        deplace(ndisques, Gauche, Droite, Milieu)
fin Hanoi
```

Analyse du coût

Pour déplacer n disques, il faut en déplacer 2 fois $n-1$, puis en déplacer un. Si on appelle $H(n)$ le nombre de déplacement nécessaires pour une tour de n disques, on peut alors écrire l'équation

$$H(n+1) = 2H(n) + 1$$

Le temps est donc sûrement exponentiel dans le nombre de disques, mais comment faire un calcul plus précis?

Dans des cours d'algorithmique, on donnera des outils pour résoudre directement ces genre d'équations définies par récurrence.

Ici, nous allons nous y attaquer par une ruse.
Remarquons d'abord que si on appelle $N(n)$ le nombre de noeuds dans un arbre binaire complet de profondeur n, on obtient la même équation

$$N(n+1) = 2N(n) + 1$$

Mais nous savons compter *directement* le nombres des noeuds d'un arbre binaire complet: $N(n)$ est égale a

$$\sum_{i=0}^{i=n-1} 2^i \quad = \quad (2^{(n-1+1)} - 1)/(2-1) = 2^{n-1}$$

Donc, $H(n) = 2^{n-1}$

Problème des tours de Hanoi: le programme en C–

```
#include <iostream.h>
#include <string>

typedef enum tour {Gauche, Droite, Milieu};

string nom(tour t)
{ if (t==Gauche) {return "Gauche";}
  else if (t==Droite) {return "Droite";} else {return "Milieu";}
}

void deplaceundisque (tour orig, tour dest)
{ cout<<"Deplace un disque de "<<nom(orig)<<" a "<<nom(dest)<<endl;
  return;
}

void deplace(int n, tour orig, tour dest, tour tmp)
{ if (n==1) {deplaceundisque(orig,dest);}
```



```
  else {deplace(n-1, orig,tmp,dest);
        deplaceundisque(orig,dest);
        deplace(n-1, tmp,dest,orig);};
  return;
}

int main()
{int ndisques;
 cout<<"Combien de disques?";
 cin>>ndisques;
 deplace(ndisques,Gauche,Droite,Milieu);
}
```

Arithmetique binaire

Rappel : un nombre binaire est une séquence

$$a_n \ldots a_1\, a_0$$

de *chiffres binaires* (0 ou 1) qui représente l'entier

$$\sum_{k=0}^{k=n} a_k 2^k = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2^1 + a_0$$

On a déjà vu dans le cours sur les itérations comment convertir une représentation binaire en représentation décimale (en évaluant le polynôme), et une représentation décimale en binaire.

N.B.: en anglais, chiffre binaire s'écrit "**b**inary dig**it**", abrégé en **bit** (pluriel: bits).
N.B.: il y a 2^n séquences différentes de chiffres binaires (ou bits) de longueur n.
N.B.: il y a $2^8 = 256$ séquences de 8 bits (appelés *octets* ou *bytes*).

Ici, nous allons nous intéresser aux algorithmes de somme sur les représentations binaires.

Opérations sur les binaires

Sur les binaires on peut réaliser des sommes, soustractions, multiplication et divisions en utilisant les mêmes algorithmes appris pour les décimales, mais en se rappelant que l'on doit travailler exclusivement avec des 0 et 1.

Par exemple, on sait calculer la somme

```
 10101101 +
 01011100 =
---------
100001001
```

ou mieux, en explicitant les retenues:

```
111111100 retenue

 10101101 +
 01011100 =
---------
100001001
```

L'addition de deux entiers binaires

L'algorithme d'addition avec retenue que l'on connaît depuis la primaire fonctionne sur les binaires aussi. Dans le programme suivant, on supposera par simplicité que les nombres binaires à additionner soient fournis sous la forme d'un tableaux de 8 entiers, chaque entier valant 0 ou 1.

```
constantes taille=8
type       octet = tableau de taille entiers
programme AddOctets
variables a, b: deux octets
                   r,tmp: deux entier
debut AddOctets
  lire a, b
  r <-0
  ecrire le resultat est
  pour i<-0 a taille faire
    tmp <- a[i]+b[i]+r
    imprime tmp mod 2
    r <- tmp div 2
  fin pour
  imprime la retenue vaut r
fin AddOctets
```

Complément

Sur les binaires, on définit aussi les opérations suivantes:

complément à 1 : étant donné un binaire i, par exemple 0011010, on appelle complément à 1 de i le binaire obtenu en remplaçant dans i chaque 0 par 1 et chaque 1 par 0; cela donne 1100101 sur notre exemple.

complément à 2 : étant donné un binaire i, par exemple 0011010, on appelle complément à 2 de i le résultat de la somme de 1 au complément à 1 de i. Toujours sur l'exemple:

- $i = 0011010$
- complément à 1 de i: 1100101
- complément à 2 de i: $1100101 + 1 = 1100110$

ASTUCE: on peut calculer le complément à deux en complémentant à 1 toutes les chiffres de droite à gauche en commençant à partir du premier 1 exclu.

Représentation des négatifs en complément à deux: addition, soustraction, débordement

Nous introduisons maintenant la représentation des entiers dite *en complément à deux*, qui a l'avantage de

- représenter des entiers *relatifs* (positifs et négatifs)
  plus précise-ment, sur n bit, on représentera les entiers entre -2^{n-1} et $2^{n-1} - 1$
- permettre de savoir facilement si l'entier représenté est positif ou négatif (le premier bit sur n donne le "signe": positif si 0 et négatif si 1)
- permettre d'utiliser l'algorithme d'addition que l'on a écrit plus haut pour toutes les combinaisons entre entiers positifs et négatifs
  Autrement dit, cela rende possible de effectuer une soustraction par une addition.
- permettre de détecter facilement les débordements (règle: il y a débordement ssi les 2 dernière retenues ne sont pas égales).

Représentation des négatifs en complément à deux

Voici la règle pour construire la représentation en complément à deux sur n bits d'un entier i compris entre -2^{n-1} et $2^{n-1} - 1$.

- si i est positif, écrire la représentation binaire habituelle, puis ajouter des 0 à gauche si nécessaire pour obtenir une chaîne de n bits.
- si i est négatifs, alors écrire le complément à 2 de la représentation de i sur n bits

Quelques exemples sur 8 bits:

entier	représentation en complément à 2
0	00000000
1	00000001
-1	11111111
-128	10000000
127	01111111

Représentation des négatifs en complément à deux: addition, soustraction, débordement

En complément à deux, les opérations arithmétiques sont particulièrement simples:

changement de signe pour changer de signe à une représentation, on calcul son complément à deux, MAIS cela ne fonctionne pas pour le cas particulier -2^{n-1}, parce-que 2^{n-1} n'est pas représentable

addition algorithme d'addition

soustraction la soustraction de deux binaires se réalise comme addition entre un binaire e le complément à deux de l'autre

Dans l'addition comme dans la soustraction, il peut y avoir débordement. La règle pour le détecter est la suivante:

- dans une addition entre représentations en complément à deux, il y a débordement ssi les deux retenues le plus à gauche sont différentes.

Addition de binaires avec les opérations sur les booléens.

Un bit a deux valeurs, zéro ou un, comme un booléen. Il est donc naturel d'essayer d'utiliser les booléens comme représentation des chiffres binaires dans nos programmes.
Or, sur les booleens on a des opérations logiques pré-définies *et*, *ou*, *non*, et non pas des opérations "arithmétiques".
Comme on a vu avec les tables de vérité, à partir de *et*, *ou* et *non*, on peut définir d'autres opérations binaires, comme par exemple le *ou exclusif*:


```
fonction ouexcl(a,b: deux booleens): booleen
debut ouexcl
     retourner ((a ou b) et non (a et b))
fin ouexcl
```

En utilisant seulement les opérations logiques, on peut réaliser les opérations arithmétiques sur les binaires, comme nous montrons dans le programme suivant, où un octet sera un tableau de 8 booléens.

Addition de binaires avec les opérations sur les booléens.

```
constantes taille=8
type       octet = tableau de taille booleens

procedure AddOctets (a, b, res: trois octets, reference retenue: booleen)
variables
       r: un booleen /* la retenue */
       i: un entier
debut AddOctets

  r <-0

  pour i<-0 a taille faire
           res[i] <- ouexcl(ouexcl(a[i],b[i]),r)
           r<-(a[i] et b[i]) ou (a[i] et r) ou (b[i] et r)
  fin pour

retenue <- r
fin AddOctets
```

Exercice: modifiez ce programme pour qu'il travaille en complément à deux (il faudra simplement tester les deux dernières retenues pour détecter un possible débordement).

Circuits logiques

Ils existent des circuits qui réalisent les fonctions logiques sur les binaires (vrai=présence de courant, faux=absence de courant), et le programme de plus en haut peut-être alors "réalisé" physiquement par un bout de silicium.
C'est comme cela que l'on réalise la UAL des ordinateurs d'aujourd'hui.

Représentation des flottants selon le standard IEEE

Sur les ordinateurs modernes, on dispose d'une représentation des réels qui est donnée par le standard IEEE 754, qui prévoit, entre autre:

simple précision : flottants sur 4 octets (32 bits), dont 1 de signe, 8 d'exposant et ... 24 de mantisse normalisée
Cela donne des flottants qui vont approximativement entre 10^{-38} et 10^{+38}, et qui ont une précision legérement supérieure à 6 chiffres décimales.

double précision : flottants sur 8 octets (64 bits), dont 1 de signe, 11 d'exposant et ... 53 de mantisse normalisée
Cela donne des flottants qui vont approximativement entre 10^{-308} et 10^{+308} et qui ont une précision legérement supérieure à 15 chiffres décimales.

En C++, le type `float` est en simple précision, et le type `double` est en double précision.

Représentation des flottants selon le standard IEEE: simple précision

31	30 23	22 0
1	11111111	11111111111111111111111
signe	*exposant*	*mantisse*

Question: on avait dit 24 bits de mantisse, mais on ne mets en mémoire que 23! que se passe-t-il avec le 24eme bit?

Réponse: la mantisse est toujours "normalisée", c'est à dire, un flottant binaire

$$b_1 \dots b_k . b_k + 1 \dots b_n 2^e$$

est réécrit commme

$$b_1 . b_2 \dots b_n 2^{e+k-1}$$

avec b_1 égal à 1 avant d'être mis en mémoire.
Or, comme b_1 est *toujours* 1, on ne le mémorise pas, donc une mantisse normalisée sur 24 bits sera mémorisée sur 23 bits (les bits après le point)

Représentation des flottants selon le standard IEEE: addition, soustraction

Les opération d'addition et soustraction entre flottants en machine se font de la façon suivante:

- on *aligne* les deux flottants, en les mettant sur la même base

- on additionne (ou soustrait) les mantisse
- on renormalise le résultat

Malheureusement, l'opération d'alignement peut faire disparaître des chiffres de la mantisse si la différence entre les deux exposants est grande.

Exemple:

$$1E5 + 1 = 100001$$

mais

$$1E8 + 1 = 1E08$$

Flottants binaires et chiffres décimales

Avec 24 bits de mantisse, on pourrait représenter une peu plus de 7 chiffres décimales, mais on tronque la représentation de la sortie décimale a 6 chiffres pour garder un peu de marge pour les calculs.

C'est ainsi que dans le fragment de programme

```
cout<<"Test 2: on additionne 1 a 1E6, puis 100000 fois 1 a 1E6"<<endl;
x=1E6;
cout<<x+1<<endl;
for(i=0;i<100000;i++){x=x+1.0;};
cout<<x<<endl;;
```

on obtient d'abord 1E6 comme résultat de 1E6+1, mais si on continue d'additionner 1 on retrouve à la fin 1.1E6.

Ce que l'on n'a pas vu

Dans ce cours, on n'a pas pu voir les aspects plus complexes d'un langage de programmation qui sont nécessaires pour traiter des vrais problèmes de la vie courante d'un informaticien. En voici quelques uns.

- Allocation de mémoire et structures dynamiques (vecteurs de taille arbitraire, arbres, listes, graphes etc.).
- Entrée/sortie sur le disque (fichiers).
- Classes/objets pour structurer les gros programmes.
- Compilation séparée.

Vous verrez quelquesuns de ces aspects dans les cours des semestres suivants.