Retour sur la notion de variable

- **Variable :**
  Désignation, par un nom, d'un emplacement mémoire destiné à contenir des valeurs d'un certain type
- **Définition (création) de variables :**
  i : entier,
  c : caractère

  Attention à la nuance entre les notions de définition et de déclaration de variables !

- **Possibilité de définir des variables contiguës de même type**
  tab : tableau[1..45] de réels
  (bloc de 45 variables contiguës :
  tab[1], tab[2] jusqu'à tab[45])

Mise en œuvre des types primitifs en C++

- Types caractérisés par une taille et par un codage de l'information
- Entier
  - (signed) **char** (-128..127), short ($-2^{15}..2^{15}-1$), int ou long ($-2^{31}..2^{31}-1$)
- Caractère … représentés par des entiers!
  - char (caractères codés par le biais de leur code ascii)
- Booléen
  - bool
- Réel
  - float (4 octets), double (8 octets)

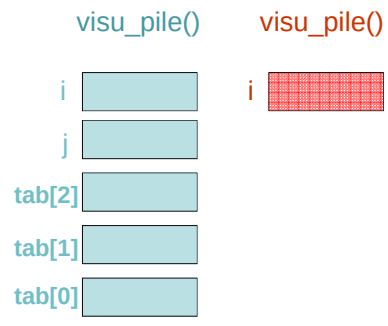
Codage

- Représentation des entiers positifs par décomposition binaire
- Représentation des entiers négatifs par complément à 2 de leur valeur absolue ((complément à 1)+1)
- Représentation des réels en virgule flottante
  $(-1)^{s}*2^{(exposant-décalage)}*1,M$ (norme IEEE)

float	double
s sur 1 bit	**s** sur 1 bit
Exposant sur 8 bits Décalage = 127	**Exposant** sur 11 bits Décalage = $2^{10}-1=1023$
Mantisse sur 23 bits	**Mantisse** sur 52 bits

- **Définition (création) de variables**
  char c;
- Possibilité d'**initialisation** à la définition
  char c=65; // idem char c='a';
- Vie et mort d'une variable :
  Organisation de la mémoire en pile
  - Les variables définies dans un bloc {} sont créées sur le sommet de la pile, dans l'ordre de leurs définitions
  - Les variables sont détruites à la sortie du bloc, dans l'ordre inverse de leur création


```
{
  int i;
  {
      int j;
      int tab[3];
      visu_pile();
  }
  visu_pile();
}
```



Comment modifier le contenu d'une variable?

- **Affectation**
  i ← 25 { i reçoit la valeur 25 }
  i ← j { i reçoit la valeur de la variable j }
  i ← 23+1 {i reçoit la valeur résultat de l'opération}
  c ← 'i' { la variable c prend la valeur 'i'}
- Mise en œuvre en C/C++
  Opérateur =
  i=25;

Les fonctions

- Une fonction est une séquence d'actions qui produit une **valeur de retour** à partir de **valeurs fournies en entrée**.

type **fonction** nom (liste paramètres formels)
Préconditions : Conditions sous lesquelles le comportement de la fonction est garanti
Résultat : Description valeur de retour
variables
séquence de définitions variables locales
début
séquence d'actions sur paramètres formels et variables locales
retourne valeur
fin

Paire_de_réels **fonction** racine(a : **réel**,
b : **réel**,
c : **réel**)
Précondition : aX^2+bX+c polynôme à racines réelles avec b*b-4a*c>=0 et a!=0
Résultat : retourne le couple de racines du polynôme
variables
delta,x1,x2 : **réel**
début
delta ← b*b – 4*a*c
x1 ← (-b + sqrt(delta))/2*a
x2 ← (-b - sqrt(delta))/2*a
retourne (x1,x2)
fin

Appel de fonction

Les paramètres formels sont initialisés avec les valeurs des paramètres effectifs fournis à l'appel

aa,bb,cc : réel
p : paire_de_réels
- - - - - - - -
aa ← 1 bb ← 5 cc ← 2

p ← racine(2,10,3)
p ← racine(aa,bb,cc) { valeurs de aa, bb et cc transmises à a, b et c }

L'appel de la fonction ne modifie pas les variables aa, bb, cc.

On dit que leur **intégrité** est préservée.

Un appel de fonction n'a **pas d'effet de bord** sur les données du programme.

Seules des valeurs sont communiquées à une fonction
(passage par valeur)

Remarque :
p ← racine(2,1,3)

Attention : Comportement de la fonction non garanti! Violation de précondition.

Définition de fonctions en C++

```
float moyenne(float a, float b)
//Précondition : a et b initialisés avec valeurs valides
   //quelconques
//Resultat : retourne la moyenne de a et b
{
   float res;
   res = (a+b)/2;
   return res;
}

A l'appel :
float x=4,d;
 ...
d=moyenne(x,2);
```

Question : Que se passe-t-il en mémoire ?

Evolution de la pile

```
float x=4,d;
 ...
```

x	4
d	

Evolution de la pile (suite)

```
float x=4,d;
 ...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```

x	4
d	

Appel moyenne

Valeur retour	
Paramètres formels a	4
Paramètres formels b	2

Evolution de la pile (suite)

```
float x=4,d;
 ...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```

x	4
d	

Appel moyenne

Valeur retour	
Paramètres formels a	4
Paramètres formels b	2
res	

Evolution de la pile (suite)

```
float x=4,d;
 ...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```

x	4
d	

Appel moyenne

Valeur retour	
Paramètres formels a	4
Paramètres formels b	2
res	3

Evolution de la pile (suite)

```
float x=4,d;
 ...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```

x	4
d	

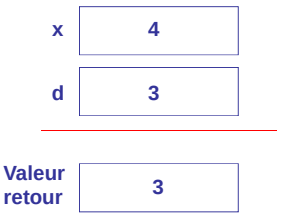
Appel moyenne

Valeur retour	3
Paramètres formels a	4
Paramètres formels b	2
res	3

Evolution de la pile (suite)

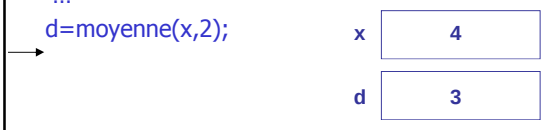
```
float x=4,d;
 ...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```



Evolution de la pile (suite)

```
float x=4,d;
 ...
d=moyenne(x,2);
```




```
int quizz(int & a, int b)
{
   int z=a+b;
   a=a+1;
   return z;
}
```

Est-ce un fonction au sens algorithmique du terme?

```
float moyenne(float & a, float & b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```

Même question?

Pour éviter toute ambiguïté :

```
float moyenne(const float & a, const float & b)
{
   float res;
   res = (a+b)/2;
   return res;
}
```


Les procédures

- Morceau de programme qui porte un nom
- Pour mettre en place des traitements pouvant avoir un **effet de bord** sur des variables existant en dehors de l'appel
- 3 modes de passage de paramètres :
  - **Donnée**
  - **Résultat**
  - **Donnée-Résultat**

procédure nom(**mode passage** liste paramètres formels)

Préconditions : Conditions sous lesquelles le comportement de la procédure est garanti

Postcondition : Décrit l'effet sur les données de l'application de la procédure

variables
séquence de définition variables locales

début
séquence d'actions sur variables locales et paramètres formels

fin

Remarque : Une procédure ne retourne rien!

procédure doubler(données-résultat x : réel)
Préconditions : x variable initialisée quelconque
Postcondition : x^+ est le double de x^-
début
x ← 2*x
fin

Appel de procédure :
- On communique les **valeurs** initialisant les paramètres formels donnée,
- On donne le nom des **variables** sur lesquels se feront *effectivement* les traitements réalisés sur les paramètres résultat et donnée-résultat :
paramètres effectifs fournis à la procédure.

Variable
dédé : **réel**
...
doubler(dédé)
doubler(3) - - NON! Cet appel n'a pas de sens!

- **Une procédure connue : l'affectation!**

procédure affectation(résultat x : réel,
donnée val : réel)
Préconditions : val initialisée avec valeur réelle quelconque
Postcondition : x^+ a pour valeur val
début
...
fin

Définition de procédure en C++

```
void exemple(int & a,     //paramètre donnée-résultat
                          //ou résultat
             int x, const int & z  //paramètre donnée
            )
//Précondition : a variable initialisée quelconque
//Postcondition : a+ a pour valeur x*z
{
   a=x*z;
}
```

Attention : En C++, on peut s'offrir la possibilité d'avoir des procédures retournant des valeurs…
A manipuler avec précaution (exemple de l'affectation)

Visibilité des variables

- (et généralement durée de vie)
- Dans leur bloc de définition
- On examinera plus tard le cas de variables dites globales

Allocation dynamique de mémoire

- Il est possible de réserver des emplacement mémoire ailleurs que dans la pile

Qu'est ce que le tas?

Qu'est ce que l'allocation dynamique de mémoire?

Qu'est ce que la libération de la mémoire allouée?

Allocation libération

- Variable
  Y : **pointeur sur** *TYPE*

  (ex: Y : **pointeur sur** entier)
- Allocation dynamique :

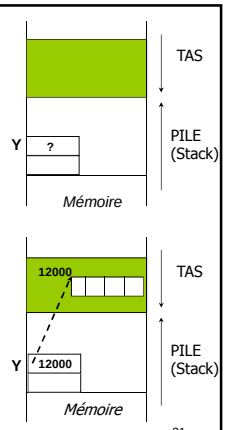
  Y ← **reserve** *TYPE*

  ou

  Y ← **reserve** tableau[1..4] de *TYPE*

Si allocation d'un tableau de 4 éléments
Alors :
Il y a réservation de 4 éléments dans le **tas**.
L'adresse du premier élément est renvoyée dans le pointeur Y

- Libération : **libère** Y



Implantation en C/C++

- En C++
  - Utilisation des opérateurs **new** et **delete**

    Ex : int *pi= **new** int;
    ....
    delete pi;
    pi=**new** int[4];
    ....
    delete [ ] pi;
- En C et en C++
  - Utilisation des fonctions **malloc** et **free** (stdlib)

    int *pi= (int *)**malloc**(4*sizeof(int));
    ....
    free(pi);

Avantage de **new**/**delete** sur **malloc**/**free** :

1. Couplage possible de **new**/**delete** avec une initialisation dépendant du type alloué
2. **new** et **delete** sont des opérateurs (pas d'édition de lien avec une bibliothèque stdlib comme pour **malloc** et **free**)

Que pensez-vous du fragment de programme suivant ?

```
{
  int j=4;
  int *pi1=new int;
  int *pi2=new int;
 *pi1=j;
 *pi2=6;
  pi2=&j;
 *pi2=*pi1+1;
}
```


Attention : Bien penser à restituer un espace alloué dans le tas avant qu'il ne soit plus pointé.

Pas de ramasse-miette *(garbage-collector)* en C ni en C++

```
{
  int j=4;
  int *pi1=new int;
  int *pi2=new int;
  *pi1=j;
  *pi2=6;
   delete pi2;
   pi2=&j;
   *pi2=*pi1;
   delete pi1;
}
```


Et que pensez-vous de ce fragment de programme là ?

```
{
   int *pi1;
   int i=5;
   {
     int *pi2=new int;
     *pi2=i;
      pi1=pi2;
      pi2=&i;
   }
   *pi1=6;
   delete pi1;
}
```