AGREGAT

Les agrégats sont des collections de données de types différents (structures et classes) ou toutes de même type (tableaux).

STRUCTURE

définition

Une définition de structure se fait en deux étapes :
- définition du type
- définition des variables

La définition du type se fait de la façon suivante :

struct** nom_du_type **{** liste des données ou fonctions membres **};

exemple :

struct tdate {unsigned jour, mois, annee}; définit un type *tdate* constitué de trois **données membres** de nom *jour*, *mois*, *annee*.

A la suite de la définition du type, nous pouvons définir des variables de ce type de la façon habituelle : *tdate d1, d2;*
Les variables définies peuvent être initialisées : *tdate d1={1, 2, 3}, d2({1, 2, 3});*

On peut également déclarer des variables de type structure sans définir le **sélecteur de structure**, c'est à dire sans définir de type : *struct {unsigned jour, mois, annee} d1, d2;*

Un membre de structure peut avoir n'importe quel type sauf le type de la structure qui est en train d'être déclarée.

Les fonctions peuvent renvoyer des valeurs de type structure, et avoir des paramètres par valeur, ou par référence de type structure.

accès aux membres

L'accès aux membres se fait en écrivant le nom de la variable, suivi d'un point et le nom de la donnée membre :
d1.jour=1; d1.mois=1; d1.annee=93;

surcharge des opérateurs

Soit la structure *TDuree* définie comme suit :

```
struct TDuree{ unsigned int heure, minute, seconde;};
```

Nous pouvons alors définir des fonctions de conversion d'une durée en un long entier et inversement de la façon suivante :

```
unsigned long conv ( TDuree d)
{ return d.heure*3600L+d.minute*60+d.seconde; }

TDuree invconv( unsigned long d)
{TDuree l;
    l.heure = d/3600; d=d%3600;
    l.minute = d/60;  l.seconde = d%60; return l;
}
```

Nous pouvons alors surcharger les opérateurs +, -, ++, <, ==, et != de la façon suivante :

```
TDuree operator + (TDuree d1, TDuree d2)
{ return invconv( conv(d1)+conv(d2)); }

TDuree operator - (TDuree d1, TDuree d2)
{ unsigned long ud1, ud2, ud;
  ud1=conv(d1); ud2=conv(d2); if(ud1<ud2) ud=0;else ud= ud1-ud2;
  return invconv(ud);
}
TDuree operator ++ (TDuree d1)
{  return invconv(conv(d1)+1); }
int operator < (TDuree d1, TDuree d2)
{ return conv(d1)<conv(d2);}

int operator == (TDuree d1, TDuree d2)
{ return conv(d1)==conv(d2);}

int operator != (TDuree d1, TDuree d2)
{ return conv(d1)!=conv(d2);}
```

De la même façon, nous pouvons surcharger les opérateurs << et >> afin de pouvoir afficher directement des durées :

```
istream& operator >>(istream& in, TDuree &d)
{ in>>d.heure>>d.minute>>d.seconde; return in;}

ostream& operator <<(ostream& out, TDuree d)
{cout<<d.heure<<"h "<<d.minute<<"mn "<<d.seconde<<"s\n"; return out;}
```

Un programme utilisant cette structure et les différents opérateurs pourrait être :

```
void main ()
{TDuree d1, d2, d3;
     cin>>d1>>d2;
     d3=d1+d2;   cout<<d3;
}
```

CLASSE

Les classes sont étudiées dans le chapître suivant.

TABLEAU

Nous distinguerons deux sortes de tableaux :

- les tableaux *statiques* pour lesquels la place occupée par leur représentation est réservée par le compilateur au moment de leurs définitions.
- les tableaux *dynamiques* pour lesquels le programmeur doit lui même gérer l'espace mémoire qui leur est destiné.

tableau statique

définitions

La notation algorithmique : **X un tableau sur [binf...bsup] d'éléments de type T** s'écrit en C++ : **T x [bsup-binf+1]**

- L'expression entre [] doit être une expression constante, ou une constante, qui doit pouvoir être évaluée par le compilateur, et non pendant l'exécution du programme.
- Sa valeur doit être supérieure ou égale à 1.
- Les éléments de *x* sont numérotés à partir de 0, jusqu'au nombre d'éléments moins 1.

La déclaration algorithmique : **X un tableau de T = [, , ,]** s'écrit en C++ : **T x [nombre d'éléments]={ , , , }**

Le nombre d'éléments doit être supérieur ou égal au nombre d'éléments de la liste qui est alors complétée par des 0 (quelque soit le type des éléments du tableau), mais ne peut en aucun cas être inférieur.

Lorsqu'un tableau est initialisé sa taille peut être omise, et il s'agit alors d'un tableau ayant pour taille le nombre d'éléments donnés dans la liste d'initialisation. En revanche il faut conserver les crochets [ ].

La notation algorithmique $\mathbf{T_i}$ se traduit en C++ par **t[i]** ou **i[t]**. Il va sans dire que la première forme est plus habituelle que la seconde.

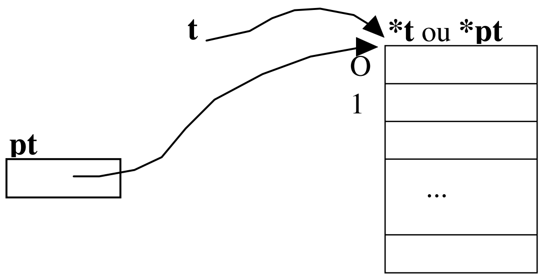
remarques :

1. *char t []={'C','+', '+'}* est un tableau de trois caractères initialisé avec les caractères C++, alors que *char t[]="C++"* est un tableau de 4 caractères contenant les caractères C++ et le caractère 0.

2. La notation *t* n'est pas une L-Value, elle est équivalente à *&t[0]* et par conséquent ***L'affectation entre tableau n'est pas légale :*** si les deux tableaux *t1* et *t2* sont de même type, l'affectation *t1 = t2* provoque le message d'erreur de compilation suivant : ***Lvalue required***. En effet un tableau est une adresse, mais une adresse qui n'est par représentée : ce n'est pas un pointeur.
En revanche on peut écrire l'expression **t1* qui a pour valeur le premier élément de *t1* : **t1* est équivalent à *t1[0]*. De façon plus générale *&t[i]* ≡ *t+i* et *t[i]* ≡ **(t+i)*.
exemple :

```
int t [10];
```

```
int *pt=(int*)&t;     // le transtypage est obligatoire pour le compilateur
int *pt=t;            // c'est plus simple
...
pt[i] et t[i] sont alors équivalents...
```



les tableaux en argument

C++ ne transmet jamais les tableaux en paramètre par valeur : *type x (T a[...])* ou *type x (T (&a)[...])*. dans les deux cas c'est l'adresse du tableau qui est transmise à la fonction.Pour transmettre un paramètre tableau consulté on utilisera le mot *const*, et dans ce cas le compilateur interdit les modifications des éléments du tableau.

Les séquences suivantes ont le même effet : dans les deux cas l'élément de rang 1 du tableau paramètre effectif sera modifié.

```
void x (char a[])
{ a[1]='b' ; }
```

```
void x ( char (&a)[])
{ a[1]='b'; }
```

La valeur de la première dimension d'un tableau paramètre peut n'être pas spécifiée : dans ce cas on parle de **tableau ouvert**, sinon de **tableau fixe**.

paramètre	sémantique
T t []	Le nombre d'éléments du paramètre n'est pas spécifié : dans la fonction **sizeof t** vaut 1(taille des éléments du tableau). Le paramètre effectif correspondant a un nombre quelconque d'éléments, et donc le programmeur doit gérer ce nombre en ajoutant un paramètre.
T t[N]	N est une constante. l'appel de la fonction **sizeof t** vaut 1(taille des éléments du tableau).. Le paramètre effectif correspondant a un nombre quelconque d'éléments, et donc le programmeur doit gérer ce nombre en ajoutant un paramètre.
T (&t)[]	l'usage de **sizeof** dans la fonction est illégal : *not allowed type*. Le paramètre effectif correspondant a un nombre quelconque d'éléments
T (&t)[N]	N est une constante. l'usage de **sizeof** renvoie le nombre d'éléments du tableau multiplié par la taille des éléments. Le paramètre effectif correspondant doit avoir le même nombre d'éléments.

Qu'un tableau paramètre soit fixe ou ouvert, on a en général besoin dans la fonction du nombre effectif d'éléments du tableau. Dans le cas ou le tableau est ouvert il faut aussi souvent transmettre la taille effective du tableau transmis en paramètre.

exemple :

```
void x (int a [], int taille)
{...}
void main()
{ int t [10];
  ...
  x(t, sizeof(t)/sizeof(t[0]));  // si le tableau est plein
  x(t, 3)                        // si le tableau ne contient que trois éléments.
}
```

Une fonction ne peut retourner une valeur de type tableau (c'est une conséquence de la non possibilité d'affectation), mais en revanche elle peut retourner une référence à un tableau, sous la forme d'un pointeur.

tableau multidimensionnel

exemple : *int a [3] [4]* définit un tableau à trois éléments qui sont des tableaux de 4 entiers. Pour initialiser un tel tableau on écrirait :

int a [3] [4] = {
{0, 1, 2, 3},
{4, 5, 6, 7},
{8, 9, 10, 11}
}

a[0] désigne *{0, 1, 2, 3}*, et *a[2][1]* désigne *9*.

La définition précédente est équivalente à : *int a[3][4]={0, 1, 2, 3 4, 5, 6, 7, 8, 9, 10, 11}*

L'initialisation : *int a[3][4]={{0}, {4}, {8}};* est équivalente à *int a[3][4]={0, 0, 0, 0, 4, 0, 0, 0, 8, 0, 0, 0}* alors que *int a[3][4]={0, 4, 8};* est équivalente à *int a[3][4]={0, 4, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0}*.

Dans les langages traditionnels, un tableau multidimensionnel est indexé par une suite d'index séparés par des virgules, et entre crochets.

En C++ une telle expression n'a pas le même sens : ***t[i, j]*** $\Leftrightarrow$ ***i, j*** est une expression virgule : chaque terme est évalué et la valeur de l'expression est la valeur de *j*. Pour indexer un tableau multidimensionnel, il faut donc écrire chaque indice entre crochets : *t[i][j]*.

Dans un tableau à plusieurs dimensions transmis en paramètre, seule la première dimension peut être laissée non affectée : en effet le compilateur doit pouvoir calculer les adresses des différents éléments

t[i][j] est un élément qui se trouve à l'adresse *t+i*(taille de la deuxième dimension)+j*

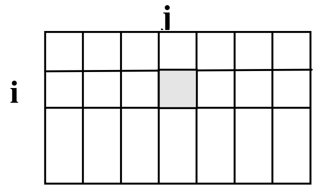


tableau dynamique

pointeur

Il existe une seule constante de type pointeur, c'est le pointeur **NULL** qui vaut 0. NULL est défini entre autres dans le fichier d'en-tête **stdlib.h**.

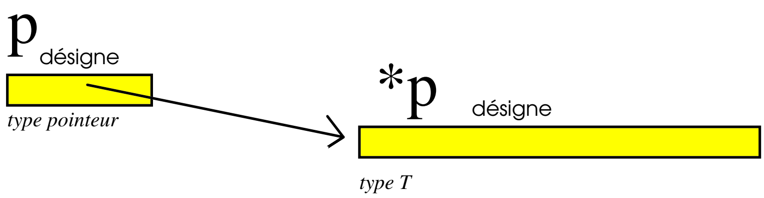
Un pointeur *p* sur un objet de type *T* est déclaré par :

```
T * p;
```

La création de l'objet pointé par *p* se fait de la façon suivante :

```
p = new T;
```

Nous pouvons représenter les relations entre pointeur et variable pointée par :



***** est l'opérateur de **déréférencement**.

Pour libérer l'espace occupé par la variable poitée par *p*, nous écrirons :

```
delete p;
```

La valeur du pointeur n'est pas modifiée, en revanche un accès à la variable pointée n'est plus légal.

Si le type *T* est une structure :

```
typedef struct { int a; int b; int c;} T;
// ou encore identique à la ligne précédente
struct T { int a; int b; int c;};

T * x;
x = new T;
```

Dans ce cas l'accès aux champs *a, b* et *c* se fait de la façon suivante :

`(*x).a   (*x).b   (*x).c` ou `x->a   x->b   x->c`

remarque :

On peut réaliser des additions sur des pointeurs :

```
short * pl; int * pi;

pl=pl+1; // la valeur de pl est augmentée de 2 octets
```

```
pi=pi+1; // la valeur de pi est augmentée de 4 octets
```

const et les pointeurs

- *X * const a* déclare un pointeur constant, c'est à dire qui ne peut pas être modifié, sur un objet de la classe *X*. *a* doit donc être initialisé.
- *const X * a* ou *X const * a* déclare un pointeur *a* variable sur un objet constant.
- *const X *const a* déclare un pointeur *a* constant sur un objet constant. *a* doit donc être initialisé.

new et delete

Ces opérateurs permettent une allocation et une désallocation dynamiques de la mémoire. La syntaxe est :

pointeur-sur-T = new T {(T-initialisateur)}; (facultatif entre {})

pointeur-sur-T = new T [nombre]; pour nombre élément de type T

delete pointeur-sur-T

delete [] pointeur sur T si T est un tableau[1].

new essaie de créer un objet en allouant si possible **sizeof(T)** octets dans le tas. S'il y réussit il renvoie l'adresse de la zone de mémoire allouée, sinon il renvoie **NULL**. S'il y a présence d'un initialisateur, la variable pointée est initialisée à sa création :

exemple :

```
int * pi;
pi = new int (3) // *pi a pour valeur 3
```

remarques :

1. `pi = new int [0]` ⇒ `pi` vaut un pointeur différent de NULL. Ce n'est pas le même à chaque fois.

2. `delete pi` si `pi` vaut NULL est correct et ne fait rien.

3. `pi = new int [10];` `pi` est un pointeur sur un tableau de 10 entiers. pour récupérer l'espace, il faut écrire `delete [] pi`.

Les fonctions tri et recherche dans un tableau

Ces fonctions ont leurs prototypes définis dans *search.h*.

La fonction *qsort* permet de trier un tableau à l'aide de l'algorithme de tri rapide (Hoare) version médian de trois pour le choix du pivot. Le prototype de cette fonction est le suivant :

void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));
base désigne le premier élément, *nelem* le nombre d'éléments, *width* la taille des éléments du tableau exprimée en octets et *fcmp* est la fonction de comparaison appliquée au éléments. Cette fonction doit

[1]Sinon tout le tableau est supprimé, mais le destructeur d'éléments n'est appliqué qu'au premier élément.

être définie par l'appelant de *qsort* et doit retourner une valeur < 0si *a<*b , =0 si *a==*b et >0 si *a>*b.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int compare( const void *a, const void *b);
char liste[5][4] = { "cat", "car", "cab", "cap", "can" };

int main(void)
{
   int  x;

   qsort((void *)liste, 5, sizeof(list[0]), compare);
   for (x = 0; x < 5; x++)
      printf("%s\n", list[x]);
   return 0;
}

int compare( const void *a, const void *b)
{
   return( strcmp((char *)a,(char *)b) );
}
```

void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (*fcmp)(const void*, const void*));

Cette fonction retourne l'adresse d'une entrée du tableau pour laquelle *fcmp* renvoie une valeur 0 quand elle est comparée avec *key*. Elle retourne NULL s'il n'y a pas d'entrée correspondante. Il s'agit d'une recherche dichotomoique, et donc le tableau doit évidemment être ordonné.

```
#include <stdlib.h>
#include <stdio.h>

typedef int (*fptr)(const void*, const void*);

#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))

int numarray[] = {123, 145, 512, 627, 800, 933};

int numeric (const int *p1, const int *p2)
{
   return(*p1 - *p2);
}


int recherche(int key)
{
   int  *itemptr;
   itemptr = (int *) bsearch (&key, numarray, NELEMS(numarray),
      sizeof(int), (fptr)numeric);
   return (itemptr != NULL);

}

int main(void)
{
   if (recherche(512))
      printf("512 trouvé\n");
   else
```

```
      printf("512 pas trouvé\n");

   return 0;
}
```

void *lfind(const void *key, const void *base, size_t *num, size_t width, int (*fcmp)(const void *, const void*));
void *lsearch(const void *key, void *base, size_t *num, size_t width, int (*fcmp)(const void *, const void *));

Recherche d'une clé dans une table. ***lfind*** revoie un pointeur sur l'élément cherché s'il se trouve dans la table, et NULL sinon. ***lsearch*** trouve toujours l'élément puisqu'il est ajouté dans le cas où il ne se trouve pas dans la table.

```
#include <stdio.h>
#include <stdlib.h>

int compare(int *x, int *y)
{
   return( *x - *y );
}

int main(void)
{
   int array[5] = {35, 87, 46, 99, 12};
   size_t nelem = 5;
   int key;
   int *result;

   key = 99;
   result = (int *) lfind(&key, array, &nelem,
      sizeof(int), (int(*)(const void *,const void *))compare);
   if (result)
      printf("%d pas trouvé\n",key);
   else
      printf("%d pas trouvé\n",key);

   return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


char *colors[10] = { "rouge", "bleu", "vert" };
int ncolors = 3;

int colorscmp(char **arg1, char **arg2)
{
   return(strcmp(*arg1, *arg2));
}

int addelem(char *key)
{
   int oldn = ncolors;
   lsearch(key, colors, (size_t *)&ncolors, sizeof(char *),
     (int(*)(const void *,const void *))colorscmp);
   return(ncolors == oldn);
}
```

```
int main(void)
{
   int i;
   char *key = "Purple";

   if (addelem(key))
      printf("%s déjà dans la table\n", key);

   else
   {
      strcpy(colors[ncolors-1],key);
      printf("%s ajouté à la table\n", key);
   }

   printf("Les couleurs:\n");
   for (i = 0; i < ncolors; i++)
      printf("%s\n", colors[i]);
   return 0;
}
```

New, delete et les classes.

1. Si *T* est une classe, lorsqu'on écrit *x= new T;* il y a appel du constructeur implicite de la nouvelle instance de classe s'il n'y a pas d'initialisateur, ou du constructeur correspondant au type des arguments. De la même façon, *delete* appelle le destructeur de la classe avant de libérer l'espace.

exemple :

```
class chainon
{
  private : int info; chainon * lien;
  public : chainon():info(0),lien(NULL){} ;
          chainon(int i):info(i), lien(NULL){} ;
          chainon(int i, chainon * l):info(i),lien(l){} ;
          ~chainon(){ ;if (lien!=NULL) delete lien;};
};
```

```
void main ()
{
  chainon * x, * y, *z;
    x = new chainon;       // constructeur implicite
    y = new chainon(1);    // deuxième constructeur
    z = new chainon(3, new chainon(4,new chainon(5, NULL)));
                           // troisième constructeur

    delete x;
    delete y;
    delete z;
}
```

2. Les opérateurs **new** et **delete** peuvent être surchargés pour une classe donnée. Ceci peut se faire pour assurer soi-même la gestion de la mémoire.

La surcharge peut ajouter des arguments à **new** ou à **delete**. Ces arguments figurent, au moment de l'appel entre parenthèses juste après **new** ou **delete**.

exemple :

```
class x
{ private : char * c;
  public :
        x(){};
       x(unsigned int n) : c(new char[n]){};
       ~x(){ delete c ;};
};
```

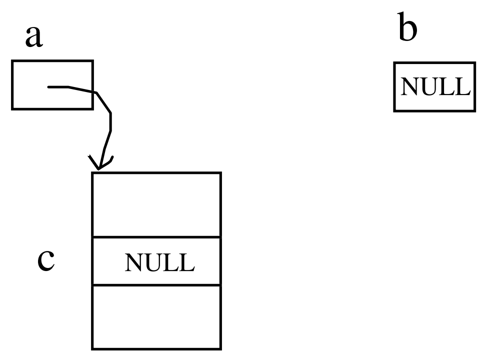
```
class y
{ private : char * c;
  public :
       y(){};
       ~y(){ delete c ;};
       void * operator new (size_t t, unsigned int n)
       { y * temp;
         temp = (y*)new char [t];
         if(temp==NULL) return NULL;
         temp->c=new char[n];
         if (temp->c==NULL) {delete temp; return NULL;}
         return temp;
        }
};
```

`(y*)` est un **transtypage** c'est à dire une conversion de type : en effet *new char* délivre un pointeur de type *char ** qu'il faut convertir en *y **.

La ligne ne peut évidemment pas être remplacée par : *temp = new y ...* car la surcharge de **new** masque la définition de l'ancien **new** a un argument qu'il faudrait donc redéfinir pour la classe **y**.

```
void main()
{
  int n; cout<<"n=? ";cin>>n;
  x * a; y * b;
  a = new x(n); // n : argument du constructeur de x
  if(a==NULL) cout<<" pas d'espace pour a";
  b = new(n) y; // n : deuxième argument du new de y;
  if(b==NULL) cout<<" pas d'espace pour b";
}
```

Dans le programme précédent, lorsqu'il n'y a pas assez de mémoire pour représenter l'objet **a** de la classe **x**, le **new** renvoie une valeur différente de **NULL**, alors que pour l'instance de la classe **y** il renvoie **NULL :** il se peut qu'on ait pas réussi à réserver tout l'espace nécessaire pour représenter **a**, mais **a** lui même a pu être alloué : dans le programme suivant, si l'espace est insuffisant pour allouer **a** ou **b** le résultat sera le suivant :



Pour traiter les réservations de mémoire qui n'ont pas marché du fait de l'insuffisance de mémoire disponible, on peut être amené à écrire une fonction de gestion de cette erreur. Pour indiquer à **new** l'existence de cette nouvelle fonction, on appelle **set_new_handler** en fournissant en argument un pointeur sur la nouvelle fonction.

Le prototype de **new** doit être le suivant :

```
void * operator new (size_t taille à allouer, liste d'arguments)
```

`size_t` est équivalent au type `unsigned`

Le prototype de **delete** doit être l'un des suivants :

```
void operator delete (void * adresse à libérer);
void operator delete (    void * adresse à liberer,
                          size_t nombre d'octets)
```

remarques :

1. On peut avoir autant d'opérateurs **new** qu'on veut pour chaque classe, en revanche il ne doit y avoir qu'un seul opérateur **delete** par classe.
2. **new** et **delete** sont considérées comme des fonctions statiques et sont donc appelables indépendamment de l'existence préalable d'une instance.
3. L'appel de **new** avec des arguments supplémentaires s'effectue en plaçant ceux-ci entre **new** et la spécification du type à allouer, pour distinguer ceux-ci des arguments du constructeur du type à allouer.
4. La surcharge de l'opérateur **delete** est plus rarement utilisée que la surcharge de l'opérateur **new**, sauf si le programmeur désire gérer lui-même la mémoire.

pointeur sur une fonction

Pour déclarer un pointeur sur une fonction, nous écrivons :

*type-retour (*nom-pointeur) (arguments)*

Nous pouvons alors affecter à une telle variable l'adresse d'une fonction ayant le même prototype. Exemple :

```
void A (int a)
{ cout<<"A : "<<a<<"\n"; }

void B (int a)
{ cout<<"B : "<<a<<"\n"; }

void (*pf) ( int ) ;      //déclaration d'une variable pf pointeur sur une fonction

void main ()
{   pf=&A; pf(1);
    pf=&B; pf(2);
}
```

Un autre exemple d'utilisation des pointeurs sur fonction :

```
typedef void(*TAction)(int &);// declaration d'un type Taction pointeur sur une fonction

void Affiche( int &a){cout<<a<<" ";}
void Plus1( int &a){++a;}
void PourTous( int t[], int n, TAction x)
{      int i;
       for(i=0; i<n;++i) x(t[i]);
}

void main()
{  int i;
       int t[]={1,2,3};
       PourTous(t,3,Affiche);cout<<"\n";
       PourTous(t,3,Plus1);  cout<<"\n";
       PourTous(t,3,Affiche);cout<<"\n";
}
```

On peut également définir un tableau de pointeurs sur des fonctions de la façon suivante :

```
void (*ptf[10])(int);

ptf[0] = f;         // affectation de f à ptf[0]

ptf[0](30);         // appel de ptf[0]
```

pointeur sur une fonction membre.

La manipulation de pointeurs sur des fonctions membres implique l'emploi de l'opérateur de portée. La déclaration d'un pointeur sur une fonction membre de la classe *x* se fait de la façon suivante :

*type-retour (x::*nom-pointeur) (arguments)*

On peut affecter l'adresse d'une fonction membre à un tel pointeur de la façon suivante : *pfm=&X::A;*

Pour appeler une fonction via un pointeur sur une fonction membre, il faut qu'il existe une instance de la classe, et de plus utiliser l'opérateur .* (déréférencement de pointeur sur membre de classe) ou ->* (déréférencement de pointeur sur pointeur de membre de classe)[2] :

```
class X
{ public :
     void A (int a){ cout<<"A : "<<a<<"\n"; }

     void B (int a){ cout<<"B : "<<a<<"\n"; }
};

void (X::*pfm) ( int ) ;

void main ()
{   X y; X *py;
```

[2]Ces deux opérateurs font partie des opérateurs qui ne peuvent pas être surchargés.

```
    pv=&v;
    pfm=&X::A;  (y.*pfm)(1);(py->*pfm)(1);
    pfm=&X::B;  (y.*pfm)(2);(py->*pfm)(2);
}
```

Gestion de la mémoire en C

Le tas en C est géré par les fonctions suivantes qui ont leur prototype dans *alloc.h* :

- **void * malloc (unsigned int n)** alloue un espace de n octets dans le tas et renvoie l'adresse de ces n octets si l'opération réussit et NULL sinon. Le pointeur renvoyé est long ou court suivant le modèle de mémoire utilisé. La valeur de n doit être inférieure à 64KO.
- **void * calloc (unsigned int n, unsigned int taille)** alloue dans le tas n blocs de taille octets initialisé à 0. renvoie NULL si l'opération n'a pas réussi.
- **void * realloc (void * p, unsigned taille)** modifie la taille d'une région mémoire préalablement allouée. le contenu est conservé, même s'il y a recopie.
- **free (void * p)** libère l'espace pointé par l'argument p.
- **long coreleft**() donne la taille du plus gand espace disponible dans le tas.
- **int heapcheck ()** (dos ou win32) teste la validité du tas : renvoie _HEAPOK si ça va, _HEAPCORRUPT si le tas est altéré, _HEAPEMPTY, s'il n'y a pas de tas.

tableau d’éléments de type T

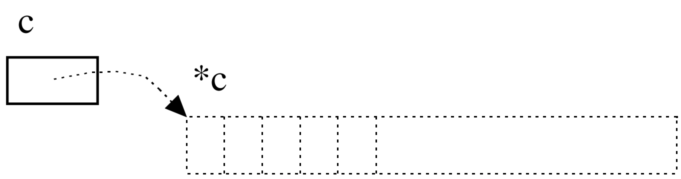
Pour définir un tableau dynamique d’éléments de type *T* qui contient *n* éléments nous écrirons :

```
T * tableau;
tableau = new T[n];        // réservation d’espace

...// utilisation du tableau

delete [] tableau;         // libération de l’espace réservé
```

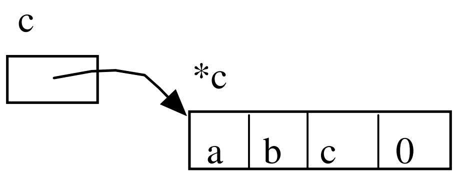
char *

Ce type permet de représenter les chaînes de caractères. Les chaînes de caractères sont représentées comme des tableaux de caractères.

La déclaration *char * c* déclare un pointeur susceptible de pointer sur un caractère :



Lorsqu'on écrit *c="abc";* il y a réservation, par borland®, d'un espace mémoire de 4 caractères et *c* pointe sur cet espace mémoire :



Avant d'écrire *cin>>c;* il faut réserver la place où seront rangés les caractères lus au clavier. Il faut évidemment réserver suffisamment de mémoire pour ranger tous les caractères. La réservation se fait de la façon suivante :

```
c = new char [10];
```

Il y a réservation d'un espace de 10 caractères consécutifs, et *c* contient un pointeur sur cet espace.

Lorsqu'on n'utilise plus la mémoire réservée, il faut la libérer pour qu'elle soit disponible à nouveau. Cet libération se fait de la façon suivante :

```
delete [] c;
```

On peut accéder à chacun des caractères de la chaîne pointée par *c* à l'aide de l'opérateur [] : *c[i]* est le $i^{ème}$ caractère de la chaîne c. le premier caractère est le caractère de rang 0.

Exemple :

```
for (i=0; c[i]!=0; ++i) cout<<c[i]<<" ";
```

Permet d'écrire les caractères de la chaîne *c* séparés par des espaces. L'instruction suivante est équivalente à la précédente :

```
for (i=0; c[i]!=0; ++i) cout<<i[c]<<" ";
```

remarques

1. Dans les séquences suivantes :

```
char * a; ... a="abcdef";   ... delete [] a;
```

ou

```
char * a="abcdef"; ... delete [] a;
```

La réservation de place pour représenter la chaîne *"abcdef"* a été faite par le système, c'est donc à lui de faire la libération de mémoire.

2. Dans la séquence suivante :

```
char *a, *b, *c;
a="abcd";
b="efgh";
c=strcat(a, b);
```

La réservation de mémoire faite pour *a* est de 5 caractères (4 caractères plus le zéro terminal), et donc on ne peut concaténer les caractères de b.

la librairie STRING.H

La librairie STRING.H contient un certain nombre de fonctions de manipulation des chaînes à zéro terminal :

- **char * strcpy (char * dest, const char * source)** copie la chaîne contenue dans *source* dans *dest*. Il faut avoir réservé suffisamment d'espace dans *dest* pour contenir la chaîne *source*. La valeur retournée est un pointeur sur *dest*.
- **char * dup(char * dest, const char * source)** copie la chaîne contenue dans *source* dans *dest*.L'espace nécessaire à la copie est réservé par la fonction. La valeur retournée est un pointeur sur *dest*. Le programmeur doit restituer l'espace réservé par strdup lorsqu'il n'est plus utilisé.
- **char * strcat (char * dest, const char * source)** copie la chaîne *source* à la fin de la chaîne *dest*. La valeur retournée est un pointeur sur *dest*. Il faut également s'assurer qu'il y a assez de place pour la copie de la chaîne *source*.
- **char * strchr(const char * chaine, int c)** renvoie un pointeur sur la première occurrence du caractère *c* dans la chaîne. Si *c=0* on aura un pointeur sur le 0 terminal de la chaîne. Si le caractère ne se trouve pas dans la chaîne on récupère un pointeur qui vaut NULL.
- **int strcmp(const char * chaine1, const char * chaine2)** renvoie une valeur :

- ◊ <0 si chaine1 < chaine2
- ◊ =0 si chaine1 = chaine2
- ◊ >0 si chaine1 > chaine2

L'ordre de comparaison est l'ordre lexicographique.

- **int strcmpi(const char * chaine1, const char * chaine2)** renvoie les mêmes valeurs que *strcmp*, sans distinguer les majuscules des minuscules.
- **unsigned strlen(const char * chaine)** a pour valeur la longueur de la chaîne.
- **char * strlwr(char * chaine)** convertit en minuscules (LOWER) les lettres majuscules de la chaîne en paramètre. La valeur retournée est un pointeur sur cette chaîne.
- **char * strupr(char * chaine)** convertit en majuscules (UPPER) les lettres minuscules de la chaîne en paramètre. Les lettres accentuées sont converties en leur équivalent accentuées. La valeur retournée est un pointeur sur cette chaîne.
- **char * strrev(char * chaine)** inverse la chaîne donnée en argument. C'est l'argument lui-même qui est modifié.
- **char * strset(char * chaine, int car)** remplace tous les caractères de chaîne par *car*.
- **char * strstr(const char * chaine1, const char * chaine2)** a pour valeur un pointeur sur la première occurrence de *chaine2* dans *chaine1*. S'il n'y a pas d'occurrence de *chaine2* dans *chaine1*, le pointeur vaut NULL.
- **char * strdup (const char * ch)** copie une chaîne en allouant automatiquement l'espace nécessaire.

strcpy, strcat et *strcmp*, ont une version avec *str**n**...(char *, char*, int lmax)* qui permet de ne traiter que les *lmax* premiers caractères.

La librairie *stdlib* possède de nombreuses fonctions de conversions de chaîne :

- **int atoi(char *ch), long atol(char *ch), double atof(char *ch) long double atold(char *ch)** permettent de convartir une chaîne dans l'un de ces types.
- **char * itoa(int i, char * ch, int base)**, **char *ltoa(long i, char * ch, int base), char * ultoa(unsigned long i, char * ch, int base)** permettent de convertir i en une chaîne de caractères en notation dans la base *base*.

sprintf(char * ch, char * format, arguments) permet de préparer une chaîne de caractères qui serait celle affichée si on avait utilisé *printf.*

Il y a bien d'autres fonctions définies dans la librairie *string.h.*