Classes d'allocation de la mémoire

Automatique (auto):

- la durée d'existence est le bloc (les entités sont allouées dans la pile d'exécution),
- est utilisée par défaut.

Statique (static):

- la durée d'existence est celle du programme,
- peut être utilisée pour des entités locales, globales ou spécifiques à la classe.

Dynamique:

- le début d'existence est lié à la fonction **malloc** (ou **calloc**),
- la fin d'existence est liée à la fonction **free**.

Temporaire:

- entités créées pour le besoin de l'évaluation d'une expression.

Opérateurs de gestion de la mémoire

En C++ : **new** et **delete**, opérateurs du langage.

En C, CE NE SONT PAS des opérateurs MAIS des fonctions **malloc** et **free**, définies dans stdlib.h.

```
/* en C les prototypes sont: void * malloc (size_t) et void free (void *) */
int * a = (int *) malloc (10 * sizeof(int));
…
free ( a );

/* en C++ */
int *a = new int[10];
int *b = new int;
…
delete [ ] a;
delete b;
```

Gestion dynamique de la mémoire

```
#include <stdlib.h> /* --> fonctions malloc () et free() */

#include <assert.h> /* --> fonction de vérification assert () */

...

int *p;

p = (int *) malloc (100*sizeof ( int )); /* allocation de la mémoire */

/* comparable à un int p[100], mais décidé à l'exécution */


/* Remarque: malloc() retourne NULL si l'allocation échoue */

assert (p! = 0); /* vérifie si l'allocation a été bien faite */

...

p[0] = 15; /* utilisation de la mémoire réservée */

p[99] = 33; /* utilisation de la mémoire réservée */

...

free (p); /* libération de la mémoire réservée */

p = 0; /* Noter le " =0 " */
```

Pointeurs sur fonctions

Un nom de fonction est implicitement un pointeur (constant) sur cette fonction.

Exemples:

int (*pf) (**void**); /* pf est un pointeur retournant un entier */

int *f (**void**); /* f est une fonction retournant un pointeur sur un entier */

double (*tf[]) (**double**) = {sin, cos, tan}; /* tf est un tableau de pointeurs sur des fonctions retournant un nombre réel */

Applications:

- transmettre une fonction à une autre fonction,
- grouper données et fonctions correspondantes dans une structure.

Attention: ne pas confondre f( ) avec f:

f( ) est un appel de fonction,

f est une adresse de fonction !!!

Exemple d'utilisation de pointeurs sur fonctions

```
main()
{       double x, y;
        x = integrate(P, 0., 3.14159);
        y = integrate(sin, 0., 3.14159);
}

double P (double x)   /* Calcule P[x] = x^2 – 2*x + 3 */
{        return (x*x - 2*x + 3);
}

double integrate (double (*f)(double x), double a, double b)
{       double k, s;
        k =1.0 + a*a;
        s += (*f)(a + k*b);
        return s;
}
```

Énumérations

Énumération est un ensemble de valeurs entières représentées par des identificateurs.

Exemples:

```
enum feux {vert, orange, rouge};

enum feux f1, f2;

f1 = vert;
```

/* Énumérations sont souvent utilisées avec l'instruction switch */

```
switch      (f1) {

case vert : passer(); break;

case orange : accelerer(); break;

case rouge : freiner(); break;

}
```

Remarques:

Il faut penser à enum feux comme à un nouveau type. Le premier identificateur reçoit la valeur 0, le suivant la valeur 1, et ainsi de suite, SAUF SI:

```
enum taille {petit = 7, moyen, grand, geant};
```

Structures

Structures sont des agrégations d'entités pouvant être de type différent.

Exemples:

```
struct personne {
                    char Nom[20];
                    short Age;
                  };
struct personne p1, p2;
struct personne p3 = {"Jean", 36}; /* initialisation */
short i = p3.Age; /* accès aux champs par l'opérateur point */
char c = p3.Nom[3]; /* accès aux champs par l'opérateur point */
```

Remarques:

Penser à struct personne comme à un nouveau type.

Taille mémoire = la somme des tailles des différents champs.

Structures: utilisation (1)

```
struct point {
        int x;
        int y;
    } ;

struct rect {
        struct point pt1;
        struct point pt2;
    } ;

struct point maxpt = { 420, 300 };
struct rect ecran;

/* fabpoint: fabrique un point à partir de ses composantes x et y */
        struct point  fabpoint (int x, int y)
        {
            struct point temp;

            temp.x = x;
            temp.y = y;
            return temp;
        }
```

Structures: utilisation (2)

```
/* addpoint: additionne deux points */
struct point  addpoint (struct point  p1, struct point  p2)
{       p1.x + = p2.x;
        p1.y + = p2.y;
        return p1;
}
/* pt_dans_rect: retourne 1 si p est dans r, 0 sinon */
int  pt_dans_rect (struct point  p, struct rect  r)
{ return p.x >= r.pt1.x && p.x < r.pt2.x && p.y >= r.pt1.y && p.y < r.pt2.y;
}

# define min (a, b) ((a) < (b) ? (a) : (b))
# define max (a, b) ((a) > (b) ? (a) : (b))
/*  canonrect: met les coordonnées d'un rectangle sous forme canonique */
struct rect  canonrect (struct rect  r)
{       struct rect temp;
        temp.pt1.x = min(r.pt1.x, r.pt2.x);
        temp.pt1.y = min(r.pt1.y, r.pt2.y);
        temp.pt2.x = max(r.pt1.x, r.pt2.x);
        temp.pt2.y = max(r.pt1.y, r.pt2.y);
        return temp;
}
```

Unions

Similaire à une structure, sauf qu'une seule variable est stockée à un instant donné.

Exemple:

```
union realint {
        double Real;
        int Int;
        long int LongInt;
    } x ;
union realint x;
x.Real = 3.14;
x.Int = 30;
x.LongInt = 35000;
```

Problème: se souvenir du champ qui est utilisé !!!

Remarques:

Penser à union realint comme à un nouveau type.

Taille mémoire = Max taille des différents champs.

Structures et pointeurs

```
struct noeud {  /* le noeud d'arbre */
                int Valeur; /* valeur associée au noeud */
                noeud *FilsGauche; /* fils gauche */
                noeud *FilsDroit; /* fils droit */
            }

noeud arbre1, arbre2, *arbre3;
...
arbre3 = &arbre1;
arbre2.Valeur = 99;
arbre1.FilsDroit = &arbre2;
arbre3 -> FilsGauche = &arbre2;
...
```

La structure ci-dessus définit une structure autoréférentielle d'un arbre binaire.

Définition de nouveaux types

Syntaxe: **typedef** *type* *nouveau_nom* ;

Définition d'un nouveau nom à un type existant.

Exemples:

typedef char Message[50] ;

typedef int Longueur;

typedef int Poids;

typedef struct personne Personne;

typedef union realint realint;

typedef enum feux feux;

Remarque:

Possibilité d'utiliser le même identificateur que celui utilisé pour déclarer la structure (ou l'union ou l'enum). Possibilité aussi de grouper typedef avec la définition du type:

typedef enum Note {a, b, c, d, e} Note;

Tableau de structures contenant un pointeur sur fonction

```
struct mfunct {
        char *name;
        double (*funct)(double);
    };
...
struct mfunct funct_tab[] = {
                { "cos", cos },
                { "sin", sin },
                ...
                { "log", log }
            };

struct mfunct *functP;
double x, val;
functP = funct_tab;
x = (*functP->funct)(val);
```

Renverser une liste chaînée

```
/* Le but de cet exercice est d'écrire une fonction qui renverse le sens d'une
liste chaînée, sans effectuer d'allocation dynamique de mémoire. */
#include <stdio.h>
#include <stdlib.h>
struct maillon                    /* définition d'un élément de la liste chaînée */
{  int valeur;
  struct maillon * suivant;
};
/* les prototypes des fonctions utilisées */
typedef struct maillon * VersMaillon ;
VersMaillon construire(void);
void ecrire(VersMaillon);
VersMaillon renverser(VersMaillon);

void main()
{ VersMaillon debut;
   debut = construire();
   printf("Voici la liste de vos entiers dans l'ordre de la liste chaînée construite: \n");
   ecrire(debut);
   debut = renverser(debut);
   printf("Voici la liste de vos entiers dans l'ordre de la liste chaînée renversée: ");
   ecrire(debut);
}
```

```
VersMaillon construire() /* fonction permettant de construire une liste chaînée */
{  VersMaillon deb, p;
   int donnee;
   deb = NULL;
   printf ("Introduisez vos données, tapez -1 pour terminer : \n");
   scanf ("%d",&donnee);
   while (donnee! = -1)
   {                              /* remplissage de la liste chaînée */
      p = (VersMaillon) malloc(sizeof(struct maillon));
      p->valeur = donnee;
      p->suivant = deb;
      deb = p;
      scanf("%d",&donnee);
    }
   return deb;
}

void ecrire (VersMaillon deb) /* imprime les données de la liste chaînée */
{  VersMaillon p = deb;
while (p! = NULL)
   { printf("%d ",p->valeur);
      p = p->suivant;
   }
   printf("\n");
}
```

```
VersMaillon renverser (VersMaillon deb) /* renverse la liste chaînée */
{  VersMaillon p, q , r;

   if ((deb == NULL)||(deb->suivant == NULL)) return deb;
   q = deb->suivant;
   deb->suivant = NULL;
   r = q->suivant;
   q->suivant = deb;
   while (r! = NULL)
   {
      p = q;
      q = r;
      r = r->suivant;
      q->suivant = p;
   }
   deb = q;
   return deb;
}
```

Entrées/Sorties en C: ouverture/fermeture de fichiers

Déclaration d'un pointeur sur fichier: FILE *pf;

Ouverture d'un fichier: pf = fopen (*nom_fichier*, *mode*);

Avec *mode* pouvant prendre plusieurs valeurs dont:

la lecture (read): r

l'écriture (write): w

l'ajout en fin de fichier (append): a

Exemples:

```
pf = fopen ("test", "w"); /* ouvre le fichier test pour l'écriture */
if ((pf = fopen ("test", "w")) == NULL)
   exit (1);   /* on quite le programme si le fichier test n'existe pas */
```

Fermeture d'un fichier: fclose(pf);

Test de fin de fichier: feof(pf);

Lecture/écriture d'un caractère: fgetc(pf); fputc(c, pf);

Lecture/écriture d'une chaîne: fgets(s, n, pf); fputs(s, pf);

Lecture et écriture formatée:

fprintf(pf, *format*, *valeurs*); fscanf(pf, *format*, *adresses*);