Formation C Avancé

Gilles Maire

2017

Titre du cours

Pointeurs

Rubriques

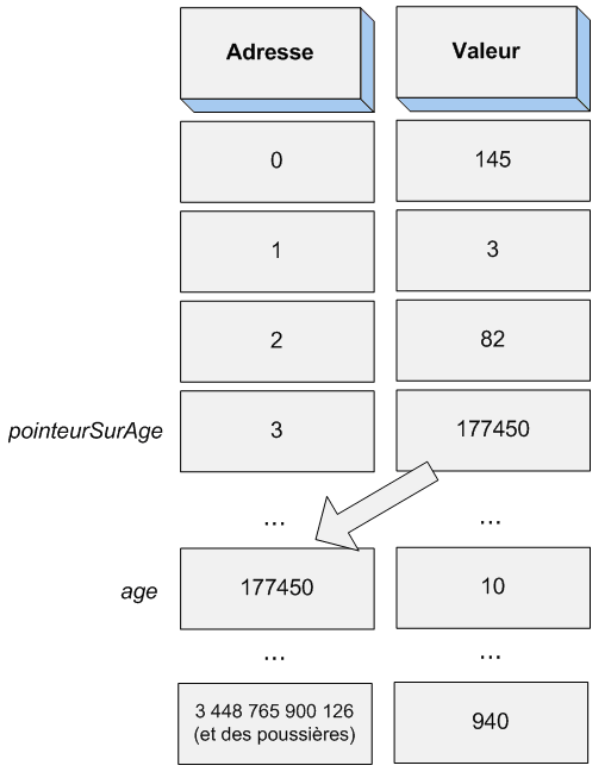
- Présentation
- Utilisations

Présentation

Introduction

- Nous avons accédé jusqu'à présent à un emplacement mémoire par une variable.
- Au lieu d'accéder par le nom d'une variable à cet emplacement mémoire, on peut choisir aussi un chemin d'accès indirect par le biais de l'adresse de la variable. Pour cela, on utilise ce qu'on appelle un pointeur.
- Un pointeur est une donnée constante ou variable qui mémorise l'adresse d'une variable.
- On dit que le pointeur renvoie ou pointe vers la variable concernée.
- Une adresse n'est rien d'autre que la désignation (indépendante du type) d'un certain emplacement mémoire, à savoir celui à partir duquel est rangée la donnée concernée.

Schéma de principe



Avantage des pointeurs

- Un pointeur est une adresse donc il représente quelques octets, ce qui est plus facile à passer en argument d'une fonction
- On peut allouer de la mémoire avec un pointeur, mémoire qui reste allouée quand on sort d'un bloc; il faudra penser à la désallouer cependant.
- On peut fabriquer des listes chaînées comme nous le verrons par la suite.

Notation

- Le pointeur se note avec une étoile comme suit

```
type        *pointeur;
```

- Les pointeurs peuvent référencer des variables de tout type :
  - sauf les variables register
  - sauf les variables champs de bits que nous verrons plus loin
- **Exemples :**

```
int *p;  /* définit un pointeur vers un entier*/
char *x;   /*définit un pointeur vers un caractère*/
float *z;  /*définit un pointeur vers un flottant*/
```

Initialisation d'un pointeur

- Pour initialiser un pointeur on lui donne l'adresse de l'objet sur lequel il pointe

```
char c='a';
char *p;
p=&c ;
```

- La déclaration et l'affectation peuvent se faire en une ligne

```
char c='a';
char *p=&c ;
```

Contenu de la variable pointée

- La déclaration du pointeur se fait par utilisation de l'étoile derrière le type

```
float *pointeur;
```

- Pour atteindre le contenu de la variable pointé on utilise le caractère * après avoir initialisé le pointeur.

```
*pointeur=3.2;
```

Utilisations

Quelques confusions à éviter

- la déclaration se fait par `type *pointeur` ce qui n'a rien à voir avec `*pointeur` qui est le contenu;
- la notation `char *pointeur=3;` est une grosse erreur, cela veut dire qu'on pointe sur la case mémoire 3 de l'adressage. Ce qui est interdit par le plan d'adressage.
- par contre la notation `char *pointeur =0` est admise, elle indique qu'on a un pointeur NULL.

Arithmétique sur les pointeurs

- On peut ajouter ou retrancher la valeur 1 à un pointeur
  - cela ne veut pas dire qu'on prend l'adresse suivante ou précédente par rapport à sa valeur précédente
  - cela veut dire qu'on prend l'enregistrement suivant ou précédent, autrement dit qu'on va se déplacer de sizeof(l'enregistrement pointé)
- Cela s'applique à toute valeur entière :
  - si je déclare un tableau de `float` de la façon suivante `float tableau[100]` je réserve 100 * 8 octets ;
  - tableau[0] est le contenu de l'adresse de départ du tableau
  - tableau[1] est le contenu de l'adresse de départ augmenté de 8 octets
  - tableau[i] est le contenu de l'adresse de départ augmenté de 8*i octets
  - pointeur+i pointe sur l'adresse de départ augmenté de 8*i octets.

Règle

```
type *pointeur;
pointeur + valeur = adresse du pointeur
                  + sizeof(type)*valeur;
```

Balayage de la mémoire avec un pointeur

- Balayage d'une string

```
char *p="ceci est une chaine" ;
char *q = p;
while (*q) printf ("%c", *q++);
```

- Nous verrons plus loin l'utilisation de listes chaînées.
- **Remarque :** Pour des besoins de débugage ou d'initiation au maniement des pointeurs, il peut être bon de savoir que l'adresse d'un pointeur peut être affichée par la syntaxe :

```
printf ("adresse pointeur %p\n",p);
```

Pointeurs et tableaux

- En C, l'identificateur d'un tableau employé seul , est un pointeur constant sur le premier élément du tableau
- L'accès à un élément quelconque d'un tableau peut se faire non seulement par le nom du tableau accompagné d'un indice, mais aussi par un pointeur manipulé par des opérations spécifiques d'arithmétique de pointeurs

```
int x[10];
int *px;
px=x
```

- px+1 représente alors l'adresse de x[1], px+2 l'adresse de x[2]
- Attention : `char *p="bonjour" ;` => p pointeur sur la constante bonjour qui ne pourra être modifiée.
- Si on veut initialiser un tableau modifiable à partir d'une chaine de caractère on l'initialise par

```
char tableau[]="bonjour";
```

Exercice : palindromes

- Écrire deux fonctions qui renvoient l'inverse d'une chaîne de caractères, l'une en utilisant les tableaux, l'autre en utilisant les pointeurs.
- Exemple : maman => namam

Mémoire

Rubriques

- Les bases
- Extensions

Les bases

Gestion de la mémoire dynamique

- Lorsqu'on définit un tableau dans un programme, on doit lui donner obligatoirement une taille fixe.
- Ceci est un gros inconvénient car souvent, on ne sait pas, à l'avance, de combien d'éléments on aura vraiment besoin.
- On parle alors de gestion statique de la mémoire.
- On peut gérer dynamiquement l'allocation de la mémoire
  - pour allouer la mémoire on utilise `malloc`, `calloc` ou `realloc`
  - on libère la mémoire par la fonction free avec en argument le pointeur sur l'objet alloué `void  free(p)`

malloc

```
#include <stdlib.h>
void *malloc( size_t size);
```

- La fonction `malloc` demande la place mémoire sur le tas (heap).
- Pour cela, on transmet à la fonction, comme paramètre , un nombre entier exprimant la dimension (en octets) du bloc mémoire que l'on désire réserver.
- renvoie un pointeur sur un type void, que l'on caste.
- La fonction `malloc` réserve alors(si possible) un bloc de mémoire contigu ayant la dimension indiquée et retourne son adresse
- Si n'arrive pas à réserver de la mémoire elle renvoie un pointeur `NULL`
- `size_t` est un unsigned integer d'au moins 16 bits.

free

```c
#include <stdlib.h>
void free(void *ptr);
```

- Désalloue la mémoire tenue par le pointeur en argument
- Si le pointeur n'est pas valable un comportement aléatoire peut survenir

Extensions

Cas des chaînes de caractères

- Nous l'avons vu char chaine[30] ; permet de définir une chaîne de 29 caractères au maximum, le caractère 'Ø' faisant office de dernier caractère.
- La manipulation d'une chaîne est donc analogue à celle d'un tableau à une dimension.
- Si on veut modifier la taille de la chaîne en cours de programme en fonction des besoins, il faut, comme nous l'avons vu précédemment opérer de manière dynamique c'est-à-dire définir un pointeur vers un caractère et allouer de l'espace mémoire en cours de programme en utilisant les fonctions d'allocation dynamique.
- **Exemple :**

```
char *nom;
nom=(char*)malloc(20);
```

- Dans ce mode de déclaration, on peut faire varier la place mémoire occupée par la chaîne en cours de programme.

calloc

```
#include <stdlib.h>
void * calloc(size_t nombre,size_t dimension  );
```

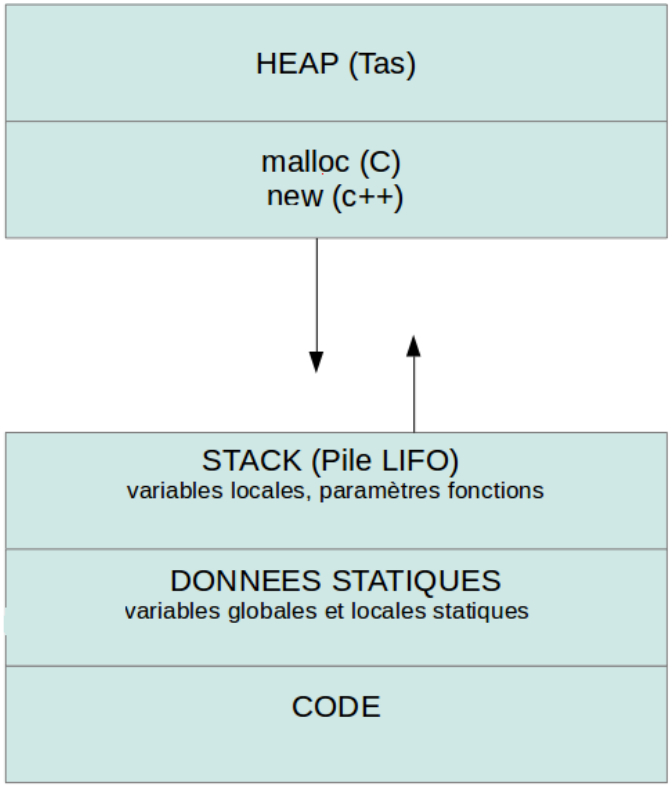
- calloc tente d'allouer un tableau de nombre d'élément et de dimension dimension.
- Les valeurs entières `nombre` et `dimension` désignent respectivement le nombre d'objets désirés et la taille (en octets) de chaque objet.
- On sauvegarde l'adresse pointée en effectuant un cast sur le type de pointeur
- réservera donc (`nombre * dimension`) octets
- initialise la mémoire à 0 ce qui peut être utile

realloc

```
void *realloc (void *base,size_t t)
```

- La fonction realloc() tente de redimensionner un bloc mémoire donné à la fonction via son adresse base avec une nouvelle taille t
- Si la tentative échoue elle renvoie un pointeur NULL et le pointeur en argument garde son allocation
- La mémoire n'est pas remise à zéro
- si le pointeur passé en argument est NULL, la fonction est équivalente à un `malloc`
- si la taille passée en argument est nulle, la fonction est équivalente à un `free`.
- Si la tentative est honorée le pointeur en argument est réalloué et c'est le pointeur retourné qui servira pour la désallocation.

Organisation de la mémoire



- Les variables statiques non initialisées sont initialisées à 0
- Elles sont déclarées précédées du mot static

Exercice : Adresses des variables

- Écrire un programme qui indique l'adresse mémoire :
  - d'une variable locale
  - d'une variable globale
  - d'une variable statique
  - d'un tableau local
  - d'un tableau global
  - d'une constante
  - d'une procédure
- Que remarque-t-on?

Fonctions

Rubriques

- Définitions
- Plus loin

Définitions

Fonctions

- Lorsqu'on veut qu'une fonction puisse modifier la valeur d'une donnée passée comme paramètre, il faut transmettre à la fonction non pas la valeur de l'objet concerné, mais son adresse.
- La fonction appelée ne travaillera plus sur une copie de l'objet transmis, mais sur l'objet lui-même (car la fonction en connaît l'adresse).
- La fonction appelée range l'adresse transmise dans un paramètre formel approprié, donc dans un pointeur.
- Bien que ce paramètre formel ne soit qu'une variable locale à la fonction appelée, la fonction a maintenant accès, via ce paramètre, à l'objet de la fonction appelante dont l'adresse a été passée comme paramètre effectif.

Exemple arguments

- Déclaration

```
void retourneDeuxArguments ( int *a, double *b)
{
 *a=1;
 *b=2;
}
void retouneDeuxArgumentsAutre( int a, int*b, int *c);
```

- Appel :

```
int val1, val2;
retourneDeuxArguments ( &val1, &val2);
retouneDeuxArgumentsAutre( 2, &val1, &val2);
```

Exercice : Date

- Écrire une fonction qui prend en premier argument une date sous la forme “23/03/2014” et qui renvoie trois arguments entiers qui sont le jour, le mois et l’année

Plus loin

Tableaux en argument d'une fonction

- Soit T un tableau d'entiers.
- Si on veut transmettre l'adresse du tableau T à une fonction fonc, il suffira d'écrire :

```
fonc(T);
```

ou

```
fonc(&T[0]).
```

- Le paramètre formel pourra s'écrire : `fonc (int *x)` ou `fonc(int x[])`

Les pointeurs sur les fonctions

- On peut déclarer un pointeur sur une fonction comme suit :

```
type_retour (*pointeur_fonction)(liste_paramètres);
```

- La parenthèse du milieu sert à ne pas confondre le * avec un pointeur sur le type en paramètre de retour
- Le nom du pointeur `pointeur_fonction` est aussi libre que celui de tout pointeur.
- On peut ainsi choisir dynamiquement la fonction à appeler parmi un ensemble de fonctions possédant les mêmes paramètres par une assignation du type

```
pointeur= &fonctionvoulue;
```

- Le caractère & est facultatif.

Arguments d'une application

```
void main ( int argc, char *argv[])
{
}
```

- Le paramètre argv est en un tableau de pointeurs.
  - Chacun de ces pointeurs pointe sur des chaînes de caractères
- argc est le nombre d'arguments >0
- argv[0] est le nom du programme
- les autres sont les arguments du programme
- Ainsi si argc vaut 2 nous aurons argv[0] nom du programme et argv[1] l'argument unique du programme qui peut être vide

Exercice : argument de main

- Afficher le nom du programme appelé ainsi que les éventuels arguments appelés

Fonctions avec arguments variables

- **Exemple :**

```
#include <stdarg.h>
void mafonction(int n,...)
```

- Le dernier argument est ...
- Le compilateur ne vérifiera pas le nombre d'arguments et le type des arguments, on peut en revanche les récupérer à l'exécution.
- on dispose du type va_list et de trois macros: `v_arg`, `va_end`, `va_start` permettant de traiter les arguments variables
- L'appel à la fonction se fera via la forme mafonction(1,"chaine") o mafonction (2,"chaine1", "chaine2") si le type char* est retenu.

Utilisation de va_list

- mafonction va utiliser va_list comme suit :

```
void mafonction (int n , ...){
    int i;  /* message  argument supplémentaire*/
    va_list local;
    va_start (local,n); /* n est le nombre d'arguments */
    for (i=0;i<n;i++) printf ("%lf",va_arg(local,double));
    va_end(local);
}
```

- **va_start** initialise local qui peut être de type entier, char * etc en fonction du type de n
- **va_arg** renvoie l'argument suivant, suivant le type donné en deuxième argument
- **va_end** déclenche un retour normal de la fonction

Exercice : va_list

- Écrire une fonction de debug qui peut prendre de une à n chaînes de caractères en argument afin de les afficher.

Les fonctions récursives

- Une fonction C peut être récursive c'est à dire s'appeler elle-même.
- **Exemple :**

```
unsigned long factoriel (int n){
   if (n < 0) { return (0);}
   else if (n == 1 || n == 0) {
      return 1L;
   }   return n * factoriel (n - 1);
}
```

- **Attention :**
  - à bien sortir de la boucle de récursivité
  - aux stacks overflows

Typedef

Rubriques

- Définition de type
- Les unions
- Présentation
- Divers

Définition de type

Création de nouveaux types

- La définition d'un nouveau type nécessite sa définition en utilisant le mot-clé `typedef` suivant la séquence

```
typedef typeexistant nouveautype;
```

- **Exemple :**

```
typedef unsigned short ushort;
```

- ushort devient un alias (synonyme) de unsigned short.
- Il est ensuite possible de déclarer ou définir une variable x par

```
ushort x;
```

Les énumérations

- **Exemple :**

```
enum jours {lundi,mardi,mercredi,jeudi,vendredi,
            samedi,dimanche};
enum jours j;
```

- ou encore avec une déclaration de type

```
typedef enum {lundi,mardi,mercredi,jeudi,vendredi,
                  samedi,dimanche} jours;
jours j;
```

Initialisation des énumérations

- On peut forcer les valeurs des énumérations
  - `enum jours {mardi=2, mercredi=3,lundi,jeudi=6}` => lundi vaut 4
  - `enum jours {mardi=4, mercredi=3,lundi,jeudi=6}` => lundi vaut 4
  - on accède aux éléments d'une énumération par

```
jours j;
if (j == lundi)
ou (j ==0)
jours j; j++;
```

Exercice énumération

- Contruire une énumération des jours de la semaine de Lundi à Dimanche
- Afficher les numéros des jours

Les unions

Présentation

Les unions

- permettent de laisser le choix d'interprétation d'une zone mémoire

```
union MonUnion
{
    int entier;
    double reel;
} u;
```

- ici on utilise soit `u.entier` soit `u.reel` en fonction de la façon dont on veut traiter l'information.
- Une union est donc un choix entre plusieurs représentations d'une zone mémoire.
- Une union peut être incluse dans une structure

Structures

Variables structurées

```
 struct livre{
        char auteur[30];
        char titre[20];
        short an;
};
```

- Un élément liv sera déclaré comme suit :

```
struct livre liv
```

- On peut uniquement à la déclaration utiliser : struct livre liv={“AGATHA CHRISTIE”,“MORT SUR LE NIL”,1965};
- On accède au champ auteur par liv.auteur

Tableau de structures

- on peut également bâtir un tableau de variables structurées

```
struct livre livres[200];
```

- l'auteur du premier livre sera alors livres[0].auteur

Typedef et Structure

- on utilise souvent un typedef pour définir une structure

```
typedef struct  {
        char titre[10];
        int ISBN;
        }  livres;
```

- On peut utiliser `livres` comme un type et déclarer `livres livre ;`

Pointeurs et structures

```
struct article
{
    char nom[20];
    long numero;
};
```

- `struct article *px;` définit un pointeur sur la structure article
- on accède au nom par `(*px).nom` ou par la notation simplifiée `px->nom`
- De même avec la variable `struct article a` on aura :
  - `a.nom`
  - `&a->nom`

Divers

Exercice Structure

- Créer une bibliothèque de 5 livres
- balayer la bibliothèque avec un pointeur pour afficher le titre de chaque livre

Alignement en mémoire

- Pour augmenter leurs performances, les processeurs sont souvent reliés à la mémoire vive par un bus plus large que la granularité de leur adressage :
  - un processeur capable d'adresser des octets est relié à la mémoire par un bus de 32 bits, soit 4 octets.
  - si une donnée de 4 octets ne se trouve pas à une adresse divisible par 4, alors il faut deux accès à la mémoire pour l'atteindre, ce qui est plus lent.
- Pour éviter les pertes de performance et les problèmes, les données sont donc alignées avec des multiples de 2, 4, 8... selon les caractéristiques du processeur cible.

Exemple

```
typedef struct _noalign {
    char c;      /* 1 octet */
    double d;    /* 8 octets */
    int i;       /* 4 octets */
    char c2[3]; /* 3 octets */
}noalign; /* 24 octets car padding */

typedef struct _align{
    double d;    /* 8 octets */
    int i;       /* 4 octets */
    char c2[3] ;/* 3 octets */
    char c;      /* 1 octet */
 }align;  /* 16 octets */
```

L'option -Wpadded permet de donner un warning si la structure n'est pas alignée

Exercice : Alignement structure

- Afficher la taille des deux structures précédemment définies

Les champs de bits

- Le langage C permet de créer, à l'intérieur d'une structure, des données (champs) dont la taille est spécifiée en bits.
- Ces champs doivent être de type entier.
- **Exemple :**

```
struct {
   unsigned a : 1;
   unsigned b : 1;
   unsigned c : 4;
   unsigned d : 3;
} x;
```

Remarques sur les champs de bits

- Ici, on a une structure x composée de 4 champs tous de type unsigned int
  - a et b de 1 bit
  - c de 4 bits
  - d de 3 bits
- Cela ne signifie pas que la taille de x est de 9 bits mais au moins 9 bits et multiple d'un octet
- On pourra définir des champs de bits sans nom pour assurer le cadrage sur un nombre pair d'octets

Listes chaînées

Rubriques

- Généralités

Généralités

Présentation

- Il est souvent très commode d'utiliser des listes chaînées.
- Chaque élément de la liste est composé
  - d'une ou plusieurs valeurs informatives
  - de l'adresse de l'élément suivant
  - parfois de l'adresse de l'élément précédent.
- On peut étendre ce mécanisme aux arbres avec feuilles gauches et feuilles droites

Utilisation des listes chaînées

- On utilise un pointeurs début de liste (ou d'arbre) et des pointeurs pour parcourir les listes
- **Exemple :**

```
struct Liste {
   int valeur;
   struct Liste *suivant;
   }

 struct Liste *premier= malloc(sizeof(struct Liste));
 premier->valeur=45;
 premier->suivant=NULL;
}
```

Structures chaînées avec typedef

- Si on utilise un typedef avec une liste chaînée il faut prendre conscience que le typedef n'existe pas pendant la création de la structure ainsi on doit explicitement faire

```
typedef struct TypeLivre {
char titre[20];
int annee;
struct TypeLivre * suivant;
} livre
```

- ensuite on peut déclarer :

```
livre * elem;
```

Exercice : Structure

- Refaire l'exercice 13 en utilisant cette fois une liste chaînée de pointeurs pour décrire la bibliothèque. On insérera les livres par ordre alphabétique d'auteur

Structures et fonctions

- Comme pour les tableaux, les structures peuvent servir de paramètres aux fonctions.
- Ici, cependant, contrairement aux tableaux, on a de nouveau le choix entre le passage d'une structure par valeur ou par adresse.
- Pour `struct livre a ;` on peut appeler
  - une fonction fonc(a) si la fonction est déclarée `fonc ( struct livre a)`
  - fonc(&a) si la fonction est déclarée `fonc ( struct livre *a)`
- Il vaut mieux passer une adresse qu'une structure car quand celle si correspond à une grosse zone de mémoire le temps de copie peut devenir significatif.

Fichiers

Rubriques

- Gestion de haut niveau
- Classes de variables
- C99

Gestion de haut niveau

Les fichiers en haut niveau

```
#include <stdio.h>
FILE *fp;
```

- La structure FILE contient
  - un pointeur sur le début du buffer
  - un pointeur donnant l'adresse du prochain byte
  - le nombre de caractères dans le buffer
  - l'état du fichier
- le descripteur du fichier

Utilisation

```
FILE *fopen(char *nom de fichier,char * modeacces);
```

- **Exemple :**

```
fp=fopen("client.dat","r");
```

- mode d'accès peut être
  - "r" pour read,
  - "w" pour write,
  - "a" pour append,
  - signe + le fichier est créé s'il n'existe pas
- `fclose(fp)` ferme le fichier ou `fcloseall()` pour fermer tous les fichiers

Lecture écriture fichier mode bloc

```
unsigned size_t fwrite( const void *adresse,
    size_t taillebloc, size_t nombreblocs,FILE *fp);
```

- Écriture
  - adresse du bloc à écrire, taille du bloc, nombre de bloc et descripteur de fichier
  - `feof (fp)` 0 si la fin du fichier n'est pas atteinte - retourne le nombre d'éléments écrits

```
unsigned size_t fread(void *adresse,size_t taillebloc,
                      size_t nombreblocs,FILE *fp);
```

- Lecture
  - idem
  - `feof (fp)` 0 si la fin du fichier n'est pas atteinte - retourne le nombre d'éléments lus

Écriture lecture fichiers mode caractères

```
int fputc(int c,FILE *p);
```

- Écriture
  - La valeur de retour de cette fonction est un nombre entier correspondant au caractère écrit si pas de problème, ou à EOF (-1) en cas de problème.

```
int fgetc (FILE *p);
```

- Lecture
  - Retourne le caractère lu sous la forme d'une valeur entière.

Lecture écriture des chaînes de caractères

```
int fputs(char *s,FILE * p);
```

- Écriture
  - écrit une chaîne de caractères dans un fichier, à la position courante.
  - Le caractère nul de fin de chaîne n'est pas recopié.
  - retourne une valeur non négative si l'écriture s'est déroulée sans encombre.

```
char * fgets(char *s int nombre, FILE *p);
```

- Lecture
  - Retourne un pointeur vers le début de la zone mémoire contenant la chaîne de caractères lue ou bien le pointeur NULL en cas de fin de fichier ou d'erreur.

Exercice : Sauvegarde bibliothèque

- Sauvegarder votre bibliothèque dans un fichier texte
- Récupérer votre bibliothèque depuis le fichier

Classes de variables

Classes de variables

- Globales
- Locales

```
int valeur1 ;
auto int valeur2;
static int valeur3;
extern int valeur4;
volatile int valeur5;
register int valeur6;
```

Variables globales

- Une variable globale est définie hors de toute fonction (même du main()).
- Elle est connue de chaque bloc et de chaque fonction qui suit sa définition.
- Sa place est réservée lors de la compilation.
- Les variables globales peuvent être utilisées et modifiées par toutes les fonctions, elles peuvent servir de moyen d'échange d'informations entre fonctions.
- Mais prenez garde aux accès multiples.

Variables locales

- Une variable locale est définie à l'intérieur d'une fonction ou d'un bloc.
- Une définition d'une variable locale doit toujours se trouver en début de bloc.
- La portée d'une variable locale est limitée au bloc dans lequel elle est déclarée, sa durée de vie dépendra de sa classe de mémorisation.
- Une variable locale peut être automatique `auto`, `static`, `register` ou `extern`

Variables auto

- Les variables de classe auto sont définies au sein d'une fonction ou d'un bloc.
- Toute variable locale est donc par défaut auto.
- La portée d'une variable auto est la fonction dans laquelle elle est définie.
- La variable n'existe en mémoire que durant l'exécution de la fonction ou du bloc dans lequel elle est définie.
- Lorsque toutes les instructions du bloc sont exécutées, la variable disparaît de la mémoire et sa valeur est automatiquement perdue pour le programme.
- Si le bloc est de nouveau exécuté, la variable est recréée.
- Une variable auto n'a pas de valeur initiale par défaut.

Variables statiques

- Les variables statiques sont définies avec le mot clé static.
- Elles ont la même portée que les variables auto mais elles existent en mémoire pendant toute l'exécution du programme.
- Une variable statique existe à partir du moment où le bloc dans lequel elle a été définie a été exécuté une fois.
- Une telle variable est créée une seule fois en mémoire.
- Si elle est initialisée lors de sa définition, elle ne sera plus réinitialisée lors d'un appel ultérieur.
- Une telle variable est initialisée automatiquement à zéro

variable externes

- Indique au compilateur qu'il ne s'agit pas ici de la définition d'une variable locale, mais de la déclaration d'une variable qui a été définie ailleurs dans le programme.
- Cette déclaration ne crée pas de nouvel objet, et en particulier n'entraîne aucune allocation de mémoire pour quelque donnée que ce soit.
- Elle sert simplement à déclarer au compilateur qu'il doit utiliser une variable globale définie ailleurs dans le programme. On parle alors d'importation de variable globale.

variables register

- Obéissent aux mêmes règles que les variables automatiques, mais elles ne sont pas toujours rangées en mémoire de travail.
- Si le compilateur le peut, il les stocke dans des registres c'est-à-dire dans des zones de mémoire incluses dans le processeur.
- Si aucun registre n'est disponible, la variable recevra la classe auto.
- L'opérateur & ne peut pas être utilisé sur des variables registre.
- L'avantage d'avoir une variable conservée dans un registre réside avant tout dans la diminution du temps d'accès à cette variable en comparaison du temps d'accès à une variable située dans la mémoire RAM.
- Ceci peut être intéressant lorsqu'une variable est souvent demandée.

Variables volatiles

- sert lors de la programmation système et indique qu'une variable peut être modifiée en arrière-plan par un autre programme (par exemple par une interruption, par un thread, par un autre processus, par le système d'exploitation ou par un autre processeur dans une machine parallèle).
- Cela nécessite donc de recharger cette variable à chaque fois qu'on y fait référence dans un registre du processeur, et ce même si elle se trouve déjà dans un de ces registres (ce qui peut arriver si on a demandé au compilateur d'optimiser le programme)

Exercice : Incrément statique

- Implémenter une fonction qui renvoie un nombre incrémenté de un à chaque appel

C99

Apports de C 99

- Les déclarations comme instruction : une déclaration de variable peut être fait ailleurs qu'en début de bloc.
- Ceci implique que les tableaux peuvent être dimensionnés avec une variable.
- Les commentaires // : si les commentaires /* */ sont toujours acceptés on peut utiliser les commentaires ligne par ligne // qui commentent le reste de la ligne
- Les const : le mot clé `const` permet d'indiquer qu'une fonction ne va pas modifier un paramètre void affiche(const Objet x)
- type long long : $>=$ 64 bits
- les tableaux peuvent être déclarés avec une valeur variable (int tab[n]) et pas seulement avec une valeur constante.
- Fonctions Inline

Fonctions inline

- Placé devant la déclaration d'une fonction `inline` demande au compilateur de recopier le code de la fonction à l'emplacement de l'appel
- Si la fonction est volumineuse ou si elle est appelée souvent, le programme devient plus gros, puisque la fonction est réécrite à chaque fois qu'elle est appelée.
- En revanche, il devient nettement plus rapide, puisque les mécanismes d'appel de fonctions, de passage des paramètres et de la valeur de retour sont ainsi évités.
- Le mot clé inline est une indication au compilateur, mais celui ci peut de pas effectuer cette directive.
- Les fonctions inline ne peuvent pas être récursives et on ne peut avoir de pointeur dessus
- Il faut que ces fonctions soient déclarées avant leur appel dans un fichier header

Exercice : C99

- Mettre en exergue les commentaires // , une fonction inline et la taille du type long long.

Compilation

Rubriques

- Présentation
- Les librairies

Présentation

La compilation

- Mécanismes de compilation
- Makefile
- Options de compilation et édition de liens
- Compilation et librairie

Les principales options de compilation

- -o program : définit le nom du programme sinon a.out
- -g génère les informations de débogage
- -0n : niveau d'optimisation de taille des binaires
- -mcpu : code pour le cpu (-mx86)
- -march=cpu : type de processeur par exemple -marm
- -W : active de Warning supplémentaires
- -Wall : active tous les Warning possibles

Les étapes de compilation

- Quatre étapes
- Preprocessing
  - gcc -E programme.c > programme.i
- Compilation vers l'assembleur
  - gcc -S programme.i produit le fichier programme.s
- Assembleur vers le code machine
  - gcc -c programme.s produit le fichier programme.o
- Édition des liens
  - gcc -o programme programme.o produit l'exécutable programme
- Mais en général on fait les 4 en une étape raccourcie :
  - gcc -o programme programme.c

Makefile

- Un est un fichier constitué de plusieurs règles de la forme :
- cible: dépendance
- commandes
- Les dépendances sont analysées, si une dépendance est la cible d'une autre règle du , cette règle est à son tour évaluée.
- Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier dépendance est plus récent que la règle, les différentes commandes sont exécutées.

Exemple d'un fichier Makefile

```
all: hello
hello: hello.o main.o
    gcc -o hello hello.o main.o
hello.o: hello.c
    gcc -o hello.o -c hello.c
main.o: main.c hello.h
    gcc -o main.o -c main.c
clean:
    rm *.o
```

Les librairies

Pour aller plus loin

- les makefiles sont dotés de variables personnalisées
  - généralement CC, CFLAGS, LDFLAGS, EXEC
- de variables internes
  - $@ : nom de la cible
  - $< : nom de la première dépendance
  - $^ : liste des dépendances
  - $? : liste des dépendances plus récentes que la cible
  - $* : nom du fichier sans suffixe
- les commandes silencieuses commencent par un @
- voir plus d'info sur la page gnu make

Définitions

- Les librairies dynamiques : le code de la librairie est présent sur le disque au moment de l'exécution et le programme va chercher les fonctions dont il a besoin
  - extension .so (sharing object) sous Unix
  - ou dll (dynamic link library) sous window
- Les librairies statiques : le code de la librairie est inclus dans l'exécutable
  - extension .a
- Dans le premier cas la librairie peut être utilisée par plusieurs programme => réduction de place
- Dans le cas de librairies rares on préfère les librairies statiques

Librairie statique

- Fabrication librairie statique
  - gcc -c fonctions.c -o fonctions.o
  - ar -q libmachin.a machin.o
- utilisation de la librairie statique au moment du lien
  - gcc program.o libmachin.a -o programme

Librairie dynamique

- Création
  - gcc -c -fPIC truc -o truc.o
  - gcc -shared -fPIC truc.o -o libtruc.so
  - L'option -fPIC (Position Independent Code) compile sans indiquer d'adresse mémoire dans le code
- Utilisation cp libtruc.so /usr/local/lib
- lancer ldconfig : il inspecte les bibliothèques dans les emplacements /lib,/usr/lib,/usr/local/lib, les chemins indiqués dans /etc/ld.so.conf , les chemins de LD_LIBRARY_PATH.

Règles pour la portabilité

- Redéfinissez les types car : char est toujours sur 8 bits, mais en fonction de architectures short, int peuvent aller de 16 à 64 bits,long de 32 à 64 bits.
- Idem pour les pointeurs sur entier ou long
- Pensez que les chaînes de caractères sont constituées de charactères UTF8 sur 16 bits
- Pensez aux problèmes d'alignements de données (voir struct)
- Pensez que certaines machines sont little endian d'autre big endian
- Méfiez vous des manipulations de bits
- Utilisez des librairies portées dans tous les environnements Qt par exemple fonctionne sous Windows, Mac, Unix