FIP SIT151 Introduction au langage C

Christophe Lohr *

1 Introduction

Les systèmes électroniques numériques ont pour vocation le traitement automatique de l'information. L'information peut prendre des formes très diverses, alors que la machine ne peut manipuler celle-ci que sous une seule forme, celle d'un nombre.

Les premiers dispositifs de calculateurs automatiques (p.ex. la Pascaline, $17^{\text{ième}}$ siècle) étaient bâtis sur de la mécanique et l'information était codée sous forme décimale (roue dentée). Passons sur les premiers calculateurs analogiques qui ont vu le jour dans les années 20-30.

Les premiers calculateurs «modernes» (années 30, p.ex. le Z-1 de K.Zuse), ont d'abord utilisé la technologie à relais électromagnétiques. Ceux-ci travaillaient en «tout ou rien», c'est-à-dire que la représentation de l'info était nécessairement binaire.

Par la suite (années 40, p.ex. ENIAC de Eckert et Mauchly), quand on a cherché à s'affranchir de la mécanique pour aller plus vite, ce sont les premiers tubes électroniques qui ont été utilisés et on les a fait fonctionner à la manière de relais (bien qu'il pouvaient aussi assurer une fonction d'amplification linéaire). Plus tard encore avec la découverte du transistor (fin des années 40), il s'est confirmé qu'il est plus efficace de faire un calculateur numérique basé sur une électronique de commutation qu'une électronique linéaire.

Enfin, le développement des premiers ordinateurs binaires a grandement bénéficié des travaux réalisés notamment par Boole et Babbage un siècle avant.

*Remerciements : André Thépaut, Bernard Prou

Pourquoi le langage C ? 3/104

Langage incontournable dans de nombreux domaines :
logiciels temps réel, embarqués, système d'exploitation, etc.

But du module :

- maîtriser le langage
- connaître quelques outils associés
- comprendre l'articulation avec la machine
- mise en évidence des points forts et des faiblesses du langage
- insister sur les aspects «délicats»

Historique 4/104

- 1972 - Dennis Ritchie & Ken Thompson (Bell Labs)
  - langage près de la machine, mais généraliste
  - faire confiance au programmeur
  - conserver une syntaxe simple et concise
- Développé à la base pour écrire l'OS Unix
- Norme historique : ANSI C ou C89
- Normes ISO 9899:xxxx (à partir de C90)
- Évolutions (mineures) : C90 C99 C11 C18 C23...
- Bibliographie :
  Le langage C, B. Kernighan, D. Ritchie, Masson, Pratice Hall
  The elements of Programming Style, Kernighan and Plauger's

Caractéristiques du C 5/104

- Langage impératif
  - langage industriel de base
  - utilisé en programmation système
- Langage efficace mais
  - faiblement typé
  - peu lisible
  - laxiste
  - peu sûr, pas de contrôle d'erreur
  - possiblement non portable
- Unités de structuration d'un code C :
  - les fonctions (sous-programmes)
  - les fichiers (modules pouvant être compilés séparément)

Peu lisible, mais fière de l'être ! Voyez "The International Obfuscated C Code Contest" `https://www.ioccc.org/` `https://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest`

Quelques caractéristiques 6/104

- Possibilité d'avoir un contrôle fin sur l'exécution
- Assembleur de haut niveau
  (pour écriture de pilotes, ...)
- Pas de bibliothèque obligatoire
  (un programme contrôlant un lave-vaisselle n'a pas besoin de la fonction printf)
- Pas d'environnement d'exécution *runtime*
  (ce qui est exécuté est le code que l'on a écrit et rien d'autre)
- Mais une bibliothèque C standard POSIX
- Programmes ~ portables
  (en C il n'y a que des programmes portés...)

Une petite liste de langages de programmation de *«bas niveau»* (ou presque), c'est à dire potentiellement dans les mêmes domaines d'utilisation que C : `https://github.com/robertmuth/awesome-low-level-programming-languages`

2 Bases du langange

Les mots réservés

8/104

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Structure d'un programme C

9/104

Un *programme* C est constitué d'un ou plusieurs *fichiers* comprenant des déclarations ou définitions de *fonctions*

- une ou plusieurs fonctions
- structure en râteau
- une et une seule fonction `main()`

Structuration en fichiers

Il est d'usage de respecter l'ordre suivant :

1. inclusion de fichiers d'en-tête (fichiers avec extension .h)
2. définition des directives du préprocesseur
3. déclaration des types
4. déclaration des variables globales
5. déclaration des fonctions
6. définition des fonctions

Déclaration : dire que la fonction existe
Définition : donner le code de la fonction

```
#include <stdio.h>

int main() {
  int i = 10;
  printf("Hello World! %d \n", i);
  return 0;
}
```

Les types de base (ou prédéfinis)

- Les types entiers

```
[ signed | unsigned ] char
[ signed | unsigned ] [ short | long ] int
```

Taille d'occupation mémoire : char $\leq$ short $\leq$ int $\leq$ long

```
short int i = -1 ;
int j = 89 ;
long int n = +123 ;
char key = 'O' ;
char buffer = 0x0A ;
```

Codage des entiers :
dépend de la machine, de l'OS, et du compilateur !
souvent : char 8bits, short 16bits, int 32(|64)bits, long 64(|128)bits
parfois int = long 64bits

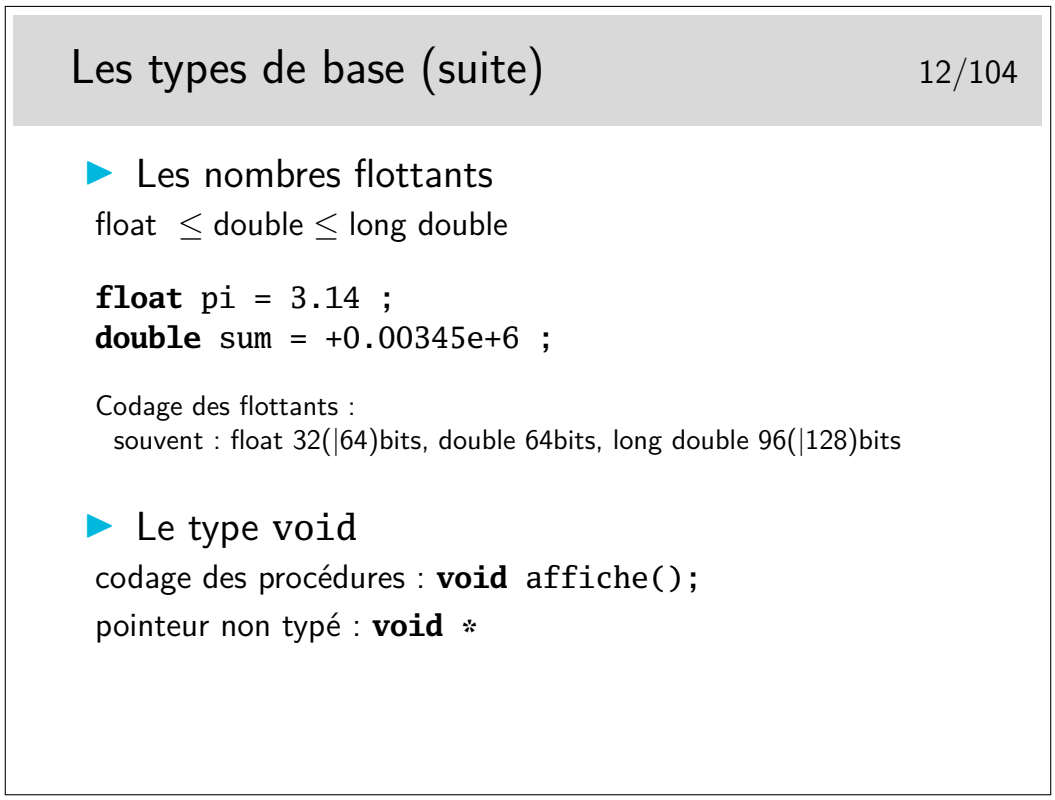
Par défaut (si on ne précise rien) les entiers sont `signed`. Si l'on veut des entiers non signés, il faut le préciser (`unsigned`). Un entier signé prend la même place mémoire qu'un entier non signé (le premier bit indique le signe, et stratégie du complément à 2).

Dans la syntaxe C, le mot clef `int` peut être omis si l'un de ses qualificatif est présent.

Notons qu'il existe aussi le `long long int`, mais souvent de même taille qu'un `long`.

Le fichier d'entête `limits.h` définit un certain nombre de macro standards qui donnent les valeur min et max des différents types, spécifiquement à la machine. Le développeur qui veut un code portable, évitera donc de faire des présupposition sur les min et max, et utilisera ces macros.

Sur certaines machines, les pointeurs avait la même taille qu'un `int` (souvent 32 bits). Ce fut vrai, cela ne l'est plus depuis longtemps : de plus en plus de machines ont un espace d'addresses (et donc des pointeurs) sur 64 bits, la taille de `long`. La norme ne spécifiant rien à ce propos, il serait mal avisé de convertir un pointeur en un `int` car on perd alors la moitié des bits significatifs sans aucun espoir de les voir revenir.



Les fichiers `limits.h` et `float.h` définissent quelques macros intéressantes pour des programmes portables.

Poit de vocabulaire : certains langage de programation font la distinction dans leur syntaxe entre une *fonction* (qui retourne quelque chose) et une *procédure* (qui contient du code mais ne retourne rien). En C on n'a que des fonction, mais certaines retournent un void, c'est à dire rien, et s'apparentent donc à des procédures.

Déclaration de «variables» 13/104

- Syntaxe :
  [const] type var1, var2, ..., varn;

```
int aNumber, myInteger, i;
int count = 100;
char reply;
float a_real, x, y;
const int max = 20;
```

- Définition d'un nouveau type (à partir d'un autre) :
  `typedef int year;`
  Introduit le nouveau type 'year' codé par un entier

Déclaration et définition de fonctions 14/104

- *Déclaration* d'une fonction :

type_retourné nom(déclaration des paramètres ou type);

```
int add(int i, int j);
int add(int, int);
```

- *Définition* d'une fonction :

type_retourné nom(déclaration des paramètres)
bloc de code

bloc de code :

```
{
  déclarations de variables locales
  instructions simples
  bloc de code
}
```

instruction simple :

```
expression;
```

— Déclaration : donner la spéciations d'une fonction pour que l'on sache comment l'appeler.
— Définition : donner les instructions qui composent la fonction.

Un premier programme simple

simple.c

```
int main( ) {
    printf("notre premier programme C \n");
}
```

- Compilation pour générer un fichier exécutable (`a.out`) :
  `~$ cc simple.c`
  ou
  `~$ gcc -Wall -o simple simple.c`
- À l'exécution nous obtenons :
  `~$ ./simple`
  `notre premier programme C`
  `~$`

```
/* Commentez vos fonctions: rôle, arguments attendus, valeur rendue, etc. */
```

Les sorties formatées

- Syntaxe :
  printf(" *formats* ", *var1*, ..., *varn*);
- Le format d'une variable s'indique derrière un %

%d	en décimal signé
%u	en décimal non signé
%o	en octal
%x	en hexadécimal
%f	réel sans exposant
%e	réel avec exposant
%c	un caractère
%s	chaîne de caractères

```
int i = 10;
float x = 3.5;
printf("i = %d, x = %f \n", i, x);
```

donnera à l'exécution : `i = 10, x = 3.5`

Programme avec appel de fonction 17/104

```
void display(int number, float real) {
    printf("number:%d real:%f \n", number, real);
}

int main() {
    int my_number;
    float my_real;

    my_number = 10;
    my_real = 2.7;
    display(my_number, my_real);
}
```

- À l'exécution nous obtenons :
  `number:10 real:2.7`

Programme avec appel de fonction 18/104

```
void display(int number, float real);

int main() {
    int my_number;
    float my_real;

    my_number = 10;
    my_real = 2.7;
    display(my_number, my_real);
}

void display(int number, float real) {
    printf("number:%d real:%f \n", number, real);
}
```

L'instruction `return` 19/104

- `return;`
  interrompt la fonction et rend la main à l'appelant
- `return` *expression*`;`
  retourne la valeur de l'expression
  et rend la main à l'appelant

```
int add(int i, int j) {
    return i + j ;
}

int main() {
    .......
    sum = add(7, 16);
    .......
}
```

3 Structures de contrôle

L'instruction conditionnelle `if` 21/104

- Syntaxe :
  if (*expression*)
  instruction simple ou composée
  ou
  if (*expression*)
  instruction simple ou composée
  else
  instruction simple ou composée
- Exemple :

```
if (count == 0)
    printf("There is no one yet.\n");
else {
    printf("Congratulation!\n");
    printf("There are %d person.\n", count);
}
```

Conseils d'écriture

22/104

```
if (foo)
  if (bar)
    win();
  else
    lose();
```

```
if (foo)
  ...
else if (bar)
  ...
```

- Mettez des accolades explicites :

```
if (foo) {
  if (bar)
    win();
  else
    lose();
}
```

```
if (foo)
  ...
else {
  if (bar)
    ...
}
```

L'instruction d'aiguillage `switch`

23/104

- Syntaxe :

```
switch (expression) {
case constante1:
      instructions simples ou composée
      break;
case constante2:
      instructions simples ou composée
      break;
case constante3:
      instructions simples ou composée
      break;
case ...
default:
      instruction simple ou composée
}
```

`switch` (exemple) 24/104

```
/* Indique si un caractere est une voyelle */
int voyelle(char caractere) {
  switch (caractere) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'y':
      return 1;
    default:
      return 0;
  }
}
```

En Java la constante doit être de type entier.
En C n'importe quelle constante de type numérique (ou dénombrable) est acceptée.

Boucle `while` 25/104

- Syntaxe :
  `while (`*condition***`)`**
  instruction simple ou composée

- Exemples :

```
while (i > 0) {
  i--;
  ...
}
```

```
while (i--) {
  ...
}
```

Notez que ces deux exemples sont équivalents

Boucle do while

- Syntaxe :
  do
    instruction simple ou composée
  while (*condition***)**

- Exemple :

```
int main() {
  int number;
  do {
    printf("Choose a number divisible by 3\n");
    scanf(" %d ", &number);
    if ( (number % 3) == 0)
      printf("You win!\n");
    else
      printf("You lose... try again\n");
  }
  while ( (number % 3) != 0);
}
```

Boucle for

- Syntaxe :
  for (*initialisation***;** *condition***;** *progression***)**
    instruction simple ou composée

- Exemple :

```
for ( i = 1 ; i < 100 ; i++)
  printf("i = %d, i^2 = %d \n", i, i*i);
```

Les entrées formatées

28/104

- Syntaxe :
  scanf(" *formats* ", **&***var1*, ..., **&***varn***);**
- Exemple :

```
int i;
float x;
printf("enter an integer and a real\n ");
scanf(" %d%f ", &i, &x);
```

Exercice

29/104

1. demandez à l'utilisateur d'entrer un nombre positif
2. boucler en (1) tant qu'il ne saisit pas zéro
3. affichez le plus petit et le plus grand de la série

Exercice à faire à la maison : écrire le code (au moins sur le papier) d'un programme qui affiche les caractères ASCII affichables (de la valeur 32 à 126), dix par ligne.

```
  ! " # $ % & ' ( )
* + , - . / 0 1 2 3
4 5 6 7 8 9 : ; < =
> ? @ A B C D E F G
H I J K L M N O P Q
R S T U V W X Y Z [
```

```
\ ] ^ _ ‘ a b c d e
f g h i j k l m n o
p q r s t u v w x y
z { | } ~
```

Les opérateurs 30/104

- Arithmétiques + - * / %
- Unaires ++ -- & *
- Logiques ! && ||
- Bit à bit ~ | & ^ << >>
- Relationnels == != < <= > >=

Écriture compacte :

- Arithmétique et affectation
  += -= *= /= %=
  x+=10; équivalent à x=x+10;
- Bit à bit et affectation x<<=2; équivalent à x=x<<2;

Attention à ne pas confondre l'affectation = et la comparaison ==.

Notez qu'en C il n'y a pas à proprement parler de booléens, les opérations logiques portent sur des entiers. Cependant, le fichier d'en-tête `stdbool.h`, standardisé dans C99, introduit un type `bool` et les constantes `true` (qui vaut 1) et `false` (qui vaut 0). Avant cela, il était courant que chaque développeur le définisse lui-même de manière ad-hoc sous forme de macro, constante, ou type énuméré. Aujourd'hui, on ajoute "`#define <stdbool.h>`" en préambule de son code.

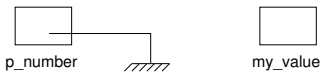
& «et» sur suites de bits (and)
| «ou» sur suites de bits (or)
^ «ou exclusif» sur suites de bits (xor)
~ «non» sur suites de bits
<< décalage de suites de bits vers la droite
>> décalage de suites de bits vers la gauche

4 Pointeurs

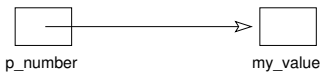
Les pointeurs 32/104

- Pointeur, une variable qui contient l'adresse mémoire d'une autre variable *(principe d'indirection)*
- Déclaration : la marque "`*`", et le type de l'objet pointé
  `int * p_i;` *(on s'intéresse au type de la variable pointée)*
- L'opérateur `*` placé devant un pointeur permet d'accéder à la variable pointée

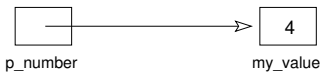
```
int *p_number, my_value ;
```




```
p_number = &my_value ;
```




```
*p_number = 4 ;
```



  - Note : l'opérateur **&** donne l'adresse d'une variable
    `&i` représente l'adresse de la variable `i`

Les pointeurs (suite) 33/104

- Le type de l'objet pointé est important

```
int *px; /* px est un pointeur d'entiers */
char *pchar; /* pchar est un pointeur de caractères*/
```

`px` et `pchar` sont tous les deux des pointeurs, mais des pointeurs de types différents, ils ne pourront donc pas être directement affectés l'un à l'autre

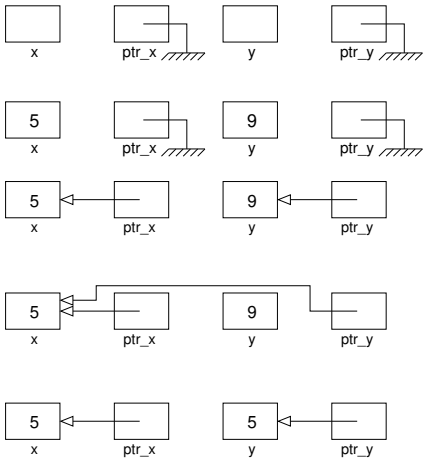
```
pchar = px; /* interdit */
```

- Conversion explicite (*cast*) :

```
pchar = (char *) px; /* le pointeur px est converti
                        en un pointeur de char */
```

Affectation des pointeurs

- Soit :
```
int x,y,*ptr_x,*ptr_y;
```
```
x = 5;
y = 9;
ptr_x = &x;
ptr_y = &y;
```
- Ne pas confondre :
```
ptr_y = ptr_x;
```
- Avec :
```
*ptr_y = *ptr_x;
```



Les tableaux

Un tableau en C est caractérisé par :

- son nom
- le type des éléments qu'il contient
- sa dimension
- Contrairement à d'autres langages de programmation, un tableau en C ne connaît pas sa taille !
  Ce qui peut occasionner des problèmes de débordement et des comportements imprévisibles...
  Charge au programmeur de gérer sa taille séparément (variable, constante, define, etc.)

Déclaration et création d'un tableau :

```
int tabint[10];
char tabcar[20];
```

Initialisation d'un tableau

- Initialisation :

```
int tabpair[5] = { 2, 4, 6, 8, 10 };
char tabcar[] = { 'a', 'b', 'c', 'd', 'e' };
```

- Accès à un élément d'un tableau :

```
tabpair[2] = 22;
tabcar[0] = 'x';
if (tabpair[3] == 8) ...
```

(les indices sont des entiers débutant à 0)

Remplissage d'un tableau


```
int main() {
  int tab[20], i;

  printf ("Enter 20 numbers\n");
  for (i = 0; i < 20; i++)
    scanf("%d", &tab[i]);

  printf("Here are the numbers you entered\n");
  for (i = 0; i < 20; i++)
    printf(" tab[%d]=%d\n", i, tab[i]);
}
```

Les tableaux multidimensionnels 38/104

- Un tableau de tableaux
  `int b[5][3];`
  `int c[10][10][10];`

- d'abord en lignes, puis en colonnes :

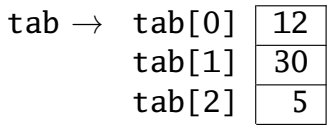
	colonne 0	colonne 1	colonne 2
ligne 0	b[0][0]	b[0][1]	b[0][2]
ligne 1	b[1][0]	b[1][1]	b[1][2]
ligne 2	b[2][0]	b[2][1]	b[2][2]
ligne 3	b[3][0]	b[3][1]	b[3][2]
ligne 4	b[4][0]	b[4][1]	b[4][2]

Les tableaux et les pointeurs 39/104

`int tab[3] = {12,30,5};`

- le *nom* du tableau est un *pointeur constant*
- pointant sur le premier élément du tableau

`tab` est équivalent à `&tab[0]`
`tab[i]` est équivalent à `*(tab + i)`



Attention à ne pas confondre :

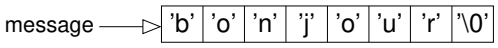
— un *pointeur constant* : il ne pointera pas sur autre chose, c-à-d. il contient une adresse (d'un objet en mémoire), et cette adresse ne changera pas, même si la valeur de l'objet pointé peut changer ;

— et un *pointeur sur une constante* : à l'instant t il pointe sur un objet, dont la valeur ne changera pas (constante), mais peut être que plus tard dans le code ce pointeur pourra pointer sur autre chose.

Les chaînes de caractères 40/104

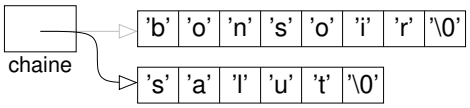
- Une chaîne de caractères est un tableau de caractères dont le dernier caractère est '\0'

```
char message[8] = "bonjour";   ou :
char message[] = "bonjour";   ou :
char message[8] = {'b','o','n','j','o','u','r','\0'};
```



- Ou un pointeur sur le premier caractère d'une suite

```
char *chaine = "bonsoir";
chaine = "salut";
```

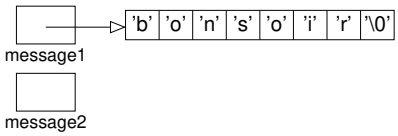


- Une chaîne peut être lue par scanf ou écrite par printf

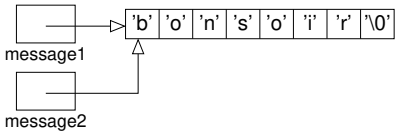
```
scanf("%s", message);
printf("%s", message);
```

Affectation de chaînes de caractères 41/104

-
```
char *message1 = "bonjour";
char *message2;
```




```
message2 = message1;
```



-
```
char chaine1[8] = "bonjour";
char chaine2[8];
```

~~`chaine2 = chaine1;`~~ refusée car chaine1 et chaine2 sont constants

Pour une *copie profonde*, utiliser la fonction strcpy :

```
char *strcpy(char *arrivee, char * depart);
```

Copie d'une chaîne de caractères 42/104

- Utilisation de tableaux :

```
void strcpy(char d[], const char s[]) {
  int i = 0;
  while ((d[i] = s[i]) != '\0')
    i++;
}
```

- Utilisation de pointeurs :

```
void strcpy(char *d, const char *s) {
  while ((*d++ = *s++) != '\0');
}
```

- En C, le passage des paramètres s'effectue par valeur,
  et l'adresse est une valeur ! (un pointeur, le nom d'un tableau)

On peut parfois préférer à `strcpy` la fonction `memcpy()` qui copie le contenu de zone mémoire, mais en précisant le nombre d'octets.

Notez que ces fonctions ont un comportement indéfinit si les zones mémoire se recouvrent (devinez pourquoi).

D'autres fonctions de la bibliothèque <string.h> 43/104

- Concatène la chaîne `src` à la suite de la chaîne `dst`
  et retourne `dst` :

```
char *strcat(char *dst, const char *src);
```

- Compare les deux chaînes `s1` et `s2` :

```
int strcmp(const char *s1, const char *s2);
```

retourne 0 si chaînes identiques
< 0 si `s1` plus court que `s2`
> 0 si `s1` plus long que `s2`

L'allocation dynamique de mémoire

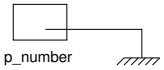
- La fonction C malloc *réserve* un espace mémoire de size octets et *retourne l'adresse* de la zone mémoire

```
void * malloc(size_t size);
```

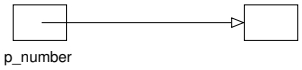
Le type `void *` est le type pointeur générique, c'est à dire capable de pointer vers n'importe quel type d'objet

Il sera convertit implicitement vers un type spécifique de pointeur en cas de besoin

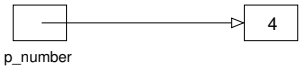
```
int *p_number;
```




```
p_number = malloc(sizeof(int));
```




```
*p_number = 4;
```



Arithmétique sur les pointeurs

- Tient compte de la taille de l'objet pointé

- Addition avec un entier :

```
p_number = p_number + i;
```

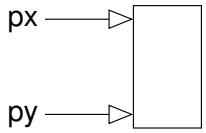
Incrémente `p_number` non pas de `i` octets,
mais de `i * sizeof(*p_number)`,
`p_number` pointe sur le $i^{ème}$ entier suivant

```
p_number = malloc( 3 * sizeof(int) );
*pentier = 4;
*(pentier + 1) = 5;
*(pentier + 2) = 6;
```

Arithmétique sur les pointeurs (suite) 46/104

- La soustraction entre deux pointeurs de même type donne un résultat entier : nombre d'éléments possibles entre les deux adresses

```
taille = py - px;
```



- L'addition de pointeurs est interdite
- La comparaison entre pointeurs est possible avec
  `== != < <= > >=`

Les tableaux et les pointeurs 47/104

- Allocation dynamique d'un tableau avec calloc :
  `void *calloc(size_t nb, size_t size);`

```
int *ptab, i;
ptab = calloc(10, sizeof(int));
for (i = 0; i < 10; i++)
  ptab[i] = 2;
```

- On libère l'espace alloué par malloc ou calloc avec :
  `void free(void *ptr);`

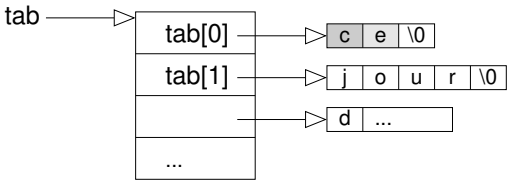
```
int *p;
p = malloc(sizeof(int));
...
free(p);
```

`free()` ne s'applique que sur les zones mémoire obtenus par malloc ou calloc, car le programme "sait" quelle taille il faut libérer (noté dans une table qui gère les allocations mémoire). Par ailleurs, il ne faut libérer la mémoire que une seule fois, sous peine de plantage garanti...

Les tableaux de pointeurs 48/104

- On peut stocker des pointeurs dans un tableau :

```
char *tab[10];
```



`tab[0]` est un pointeur sur le caractère `'c'`
`*tab[0]` vaut `'c'`
`*(tab[0] + 1)` vaut `'e'`

- Attention : ne pas confondre avec une matrice !
  à l'usage, cela peut ressembler `(tab[0][0] == *tab[0] == 'c')`
  mais ce n'est pas la même organisation en mémoire...

Attention à ne pas confondre :

— un tableau de pointeurs sur des chaînes de caractères, et

— un tableau de tableau de caractères (une matrice).

Certes, à l'usage cela peut ressembler un peu (testez le code ci-dessous), mais la gestion de la mémoire est très différente. Faites un petit dessin.

```
#include <stdio.h>
int main() {
  char * tab[] = { "ce", "jour", "de", "mai", "2011" };
  char mat[5][5] = { "ce", "jour", "de", "mai", "2011" };
  printf("\%c\n", tab[0][1]);
  printf("\%c\n", mat[0][1]);
}
```

Passage de paramètres à une fonction 49/104

- En C tous les paramètres des fonctions sont passés par valeur.
  La fonction appelée ne possède pas de moyen direct de modifier les variables de la fonction appelante.

```
void exchange(int i, int j) {     // Attention:
  int tmp = i;                    //   i j sont des
  i = j;                          //   variables locales
  j = tmp;                        //   content les valeurs
}                                 //   des paramètres
```

```
int i=4, j=8;
exchange(i, j);       // équivaut à exchange(4,8)
printf("i=%d j=%d\n", i, j);   →   affiche: i=4 j=8
```

- La fonction exchange ne peut échanger les arguments i et j du programme appelant car elle travaille sur des copies de i et j.
  Pour obtenir le fonctionnement souhaité, la fonction appelante doit passer l'adresse des variables devant être modifiées.

Adresse d'une variable en paramètre 50/104

- Pour pouvoir modifier la valeur d'une variable dans une fonction appelante, il faut passer l'adresse de cette variable en paramètre à la fonction appelée.

```
void exchange(int *pi, int *pj) {
  int tmp = *pi;
  *pi = *pj;
  *pj = tmp;
}

int main() {
  int i=4, j=8;
  exchange(&i, &j);
  printf("i=%d j=%d\n", i, j);
}
```

```
  i=8 j=4
```

5 Constructions de types

Les définitions de type 52/104

- Syntaxe :
  typedef *type déclarateur*;

- `typedef long Color;`

  déclare que le mot `Color`
  est un nouveau type
  et qu'il est construit à partir de `long`

  ```
  Color pixel, palette[128];
  ```

Le langage C étant *faiblement* typé, le compilateur de relève pas d'erreur si le code mélange les types d'origine et les types redéfinis. C'est une facilité offerte au programmeur pour organiser sa pensée et le moyen de l'exprimer...

Les structures 53/104

- Définir un *nouveau type*
  composé de *plusieurs champs* de types différents

```
struct address_t {       // Le nom de ce nouveau type
  int number;            //   ...
  char street[15];       //   ... les différents champs
  long zipcode;          //   ...
  char town[15];         //   ...
};

struct account_t {       // Un autre
  long id;
  char name[15];
  char forename[15];
  struct address_t address;
} the_account1;   // déclaration de variable

struct account_t the_account2, *p_account;
```

Les habitués des concepts de *programmation objet* verront le parallèle entre les *classes* et les *structures*. En première approximation, on peut dire qu'une classe est une structure incluant des méthodes en plus, ou qu'une structure est une classe qui n'a que des attributs.

Initialisation d'une structure 54/104

```
struct family_t {
  char name[20];
  char dad[15];
  char mum[15];
  unsigned childs;
};

struct family_t fam_dupont = {"dupont", "luc", "odile", 3};

typedef struct family_t family;

typedef struct {   // structure sans nom!
  float real_part;
  float imaginary_part;
} complex;

complex c1 = { 4.0, 5.7};
```

Notons que l'on aurait pu avoir le même nom pour la structure et pour la redéfinition de type :

```
struct family {
  ...
};
typedef struct family family;
```

En effet, le nom de la structure est `struct family` (le mot struct fait partie intégrante du nom), que le compilateur de confondra pas avec le mot `family` tout seul, introduit par le typedef...

Accès aux membres d'une structure 55/104

- L'opérateur **"."** (point) à partir d'une variable de type structure :

```
struct family_t fam;   /* ou: 'family fam;' */
strcpy(fam.nom, "durand");
strcpy(fam.dad, "benoit");
strcpy(fam.mum, "sylvie");
...
if (fam.childs > 3)
  printf("large family\n");
```

- L'opérateur **->** (flèche) à partir d'un pointeur sur une structure :

```
struct account_t *p_account;
p_account = malloc( sizeof(struct account_t) );
p_account->id = 450004;
p_account->adresse.number = 5;
```

Opérations sur les structures 56/104

- Obtenir l'adresse d'une structure **&**
- Accéder à un membre d'une structure **.** ou **->**
- Affecter une structure à une autre structure de même type :

```
struct family_t fam1, fam2;
...
fam2 = fam1;
```

Par contre il n'est pas possible de comparer deux structures :
`if (fam1 == fam2)`
ou `if (fam1 != fam2)` sont des tests interdits
Pour réaliser ces tests il faut connaître la topologie exacte de la structure, puis écrire la fonction désirée

Les énumérations 57/104

- Une énumération en C est une suite de constantes symboliques automatiquement transformées en `int`

```
enum color { RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW};
enum color pixel, border;
pixel = GREEN; /* équivalent à pixel=1 */
```

  - `enum color` est un nouveau type
  - Les éléments de la liste ont des valeurs entières affectées par défaut à partir de 0 (`RED=0 GREEN=1 ...`)

- Intérêt de l'énumération :
  - une manière de définir des constantes et de leur donner un sens
  - les programmes sont plus lisibles
  - permet au compilateur de relever des bizarreries (p.ex. `switch`)
  - souvent on choisit d'écrire ces noms de constante sont en majuscule (comme les `#define`)

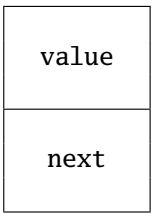
Petite parenthèse à propos des convention d'écriture dans un code : l'auteur d'un code peut choisir le style qui lui plaît, mais il est important de rester cohérent et de s'en tenir au choix que l'on a fait tout au long de son code. Par style, on entend le faite de mettre des espace à certain endroit (p.ex. dans les tests des if, autour du =, dans des listes de paramètres), le fait de revenir à la ligne avant ou après des accolades, la façon de nommer ses variables et fonctions, etc.

6 Listes chaînées

Les listes 59/104

- Caractéristiques :
  - Une suite ordonnée d'éléments de même type
  - Taille dynamique : le nombre d'éléments peut varier
  - Une structure de donnée de base en informatique
- Une liste chaînée : ensemble de cellules
  - chaque cellule comprenant l'information à stocker
  - mais aussi l'adresse de la cellule suivante

```
struct cell_t {
  int value;
  struct cell_t * next;
};
typedef struct cell_t cell;
```

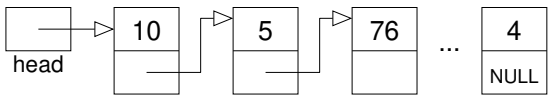


Dynamique : on va pouvoir ajouter ou supprimer des éléments de la liste au début, à la fin, au milieu. Ce ne serait pas possible avec un tableau : il est de taille fixe, et modifier sa structure implique des recopies.

Une liste d'entiers 60/104

- Une variable de type liste contient l'adresse de la première cellule de la liste (tête)

```
typedef cell * list;
list head;
```



Multiples variantes possibles :

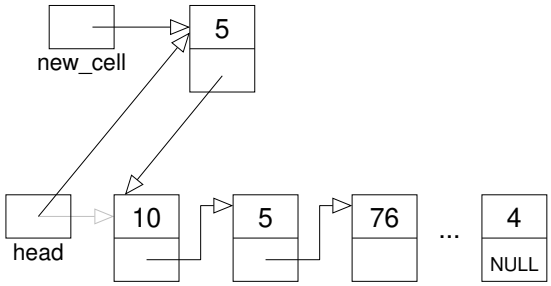
```
typedef struct cell_t * list;
cell * head;
...
```

Insertion d'un entier en tête de liste 61/104

```
cell * new_cell;

new_cell = malloc( sizeof(cell) );
new_cell->value = 25;
new_cell->next = head;

head = new_cell;
```



En toute rigueur, la liste ci-dessus est une liste *simplement* chaînée. Mais, en informatique on utilise également des listes *doublement chaînées* (chaque cellule a en plus un pointeur vers la cellule précédente), cela facilite certaines opération (insérer avant une cellule donnée).

Pour insérer en queue d'une liste simplement chaînée, il faut d'abord la parcourir pour trouver la fin. Aussi, dans certain cas (si c'est un besoin fréquent dans le code) on défini alors la liste non pas uniquement comme le pointeur sur le début, mais on conserve une paire de pointeurs : l'un sur le début, l'autre sur la queue.

Devinez comment on réalise une structure de type *fifo*, *pile*, *liste circulaire*...

7 Pointeurs de fonctions

Les pointeurs de fonctions 63/104

- Pointe non pas sur une variable, mais sur une fonction
- on peut alors le passer en argument à des fonctions

```
int add(int a, int b) { return a+b; }
int sub(int a, int b) { return a-b; }

int apply(int a, int b, int (*calculus)(int,int) ) {
  return calculus(a, b);
}
```

Ce troisième paramètre attend une fonction
- qui retourne un `int`
- et qui exige deux paramètres `int`

Les pointeurs de fonctions 64/104

```
int add(int a, int b) { return a+b; }
int sub(int a, int b) { return a-b; }

int apply(int a, int b, int (*calculus)(int,int) ) {
  return calculus(a, b);
}

int main() {
  int i, j, sum, diff;
  printf("Enter two numbers: ");
  scanf("%d %d", &i, &j);
  sum = apply(i, j, add);
  diff = apply(i, j, sub);
  printf("sum=%d diff=%d\n", sum, diff);
}
```

Note : en C, le nom d'une fonction est un pointeur constant

8 Constructions de types avancées

Les unions 66/104

- Même syntaxe que les structures (à part le mot clé union)
- Permet de réaliser des enregistrements variants
  (peut contenir, à des moments différents, des types différents)

```
union length {
  int n;
  float x;
};

int main() {
  union length width;
  width.n = 10;
  printf("as an int:%d   as a float:%f\n", width.n, width.x);
  width.x = 10.0;
  printf("as an int:%d   as a float:%f\n", width.n, width.x);
}
```

Affiche :
```
as an int:10           as a float:0.000000
as an int:1092616192   as a float:10.000000
```

C'est bien la même zone mémoire qui est occupée par notre varible, quelque soit la manière dont on la regarde (ici, soit comme un int, soit comme un float). C'est au programmeur de savoir ce qu'il fait.

Que fait le programme suivant ?

```
#include <stdio.h>

union unfold_int {
  int i;
  unsigned char buf[4];
};

int main() {
  union unfold_int a;
  int n;

  printf("Give me number: ");
  scanf("%d", &a.i);

  for (n=0; n < 4; n++)
    printf("%02x ", a.buf[n]);
  printf("\n");
}
```

Champs de bits 67/104

- Manipuler des tranches de bits dans des octets

```
struct float16 {
  int sign:1;
  int exponent:11;
  unsigned fraction:4;
}
```

A manier avec beaucoup de précautions...
Ces constructions ne sont pas portables...

Cela offre des facilités pour manipuler des structures de données binaires. Pour autant, il faut se méfier des questions du genre *big endian* vs. *little endian* de la machine.

Pour les différents codages standardisés de nombres à virgule flottante : `https://fr.wikipedia.org/wiki/IEEE_754`

9 Organisation de la mémoire

Gestion des variables en mémoire 69/104

- Visibilité : **locale** à la fonction, **globale** au programme
- Implantation des variables en mémoire :
  - **pile** : variables locales ordinaires, paramètres d'appel des fonctions, créés au début et détruites à la fin de la fonction, sauf celles qualifiées de `static`
  - **tas** : les `malloc() free()`, cycle de vie explicite
  - registres : certaines variables locales très utilisées, à placer sur les registres du processeur, si possible (et pas de &)
- Qualifieurs :
  - `auto` (implicite) : variables locales ordinaires
  - `extern` : *déclare* que la variable est *définie* ailleurs
  - `static` : variable locale permanante,
    ou variable globale mais visible que dans le fichier

Notons qu'il y a également le qualifieur `volatile` qui prévient le compilateur que la variable

locale peut être modifiée par autre chose que le programme lui-même (p.ex. sur périphérique, pour des entrées sorties), et qu'il de doit pas faire d'optimisations dessus.

Les variables externes

- *déclaration* vs. *définition* d'une variable globale
- La variable globale est définie dans un autre fichier et déclare qu'elle est visible ici

main.c

```
double lambda;

...

int main() {
  lambda = 1.5;
  ...
}
```

module1.c

```
extern double lambda;

int func(double f) {
  ... lambda *= f;
}
...
```

(sauf si le *main.c* l'avait définit comme `static double lambda;`)

Les variables statiques locales

- Une variable statique locale garde sa valeur d'un appel sur l'autre

```
double mean(double x) {
  static double sum = 0; /* initialisée qu'une seule fois */
  static int count = 0;
  sum += x;
  count++;
  return sum / count;
}
```

Les variables statiques globales 72/104

- Une variable statique globale est spécifique au fichier où elle est définie ; elle n'est pas exportable

main.c

```
int buffer;

...

int main() {
  ...
}
```

éventuellement accessible dans un autre fichier par un `extern`

module.c

```
static int buffer;

int func(int i) {
  ...
}
...
```

globale dans ce fichier mais inaccessible dans un autre

Il n'y a pas de conflit entre ces deux variables

10 La ligne de commande

Paramètres de la ligne de commande 74/104

- Possibilité de passer des arguments de la ligne de commande à la fonction principale

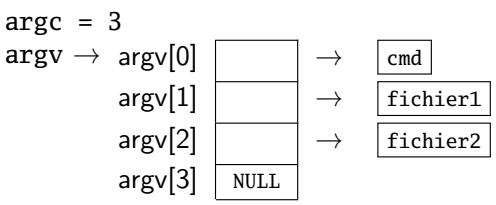
```
int main (int argc, char *argv[])
```

  - Les noms `argc` et `argv` sont des noms traditionnels qui peuvent être remplacés par n'importe quels autres noms (seuls les types doivent être respectés)
  - `argc` (*argument count*), le nombre de mots sur la ligne de commande
  - `argv` (*argument values*), une table de pointeurs pointent sur les chaînes de caractères de chaque mot

Exemple

75/104

- Par exemple le lancement du programme :
  `~$ cmd fichier1 fichier2`



Les habitués de Java remarquerons une différence : en C `argv[0]` désigne le nom du programme lui-même tel qu'il a été appelé sur la ligne de commande, alors qu'en Java c'est le nom du premier paramètre.

Commande "echo"

76/104

```
main (int argc, char *argv[]) {
  int i;
  for (i = 1; i < argc; i++)
    printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n' );
}
```

- Opérateur Conditionnel :

  expression1 **?** *expression2* **:** *expression3*

  si *expression1* vrai on renvoie *expression2*
  sinon on renvoie *expression3*

11 Compilation

Les phases de la compilation 78/104

- Sur chaque fichier `.c`
  - Prétraitement (`gcc -E` →`.e`) :
    purement textuel, traîte les include et defines
  - Compilation (`gcc -S` →`.s`) :
    traduit le text en mnémoniques assembleur
  - Assembleur (`gcc -c` →`.o`) :
    traduit les mnémoniques en fichier objet
- Avec tous les fichiers `.o` :
  Édition de liens (`ld` →`a.out`) : regroupe les fichiers objet et librairies, résout les référence croisées, adresses des fonctions et variables, créé un exécutable

Habituellement, le compilateur enchaîne toutes ces étapes (lorsque l'on appelle juste `gcc`). Mais il peut s'arrêter à n'importe laquelle (cf. option indiquée), et démarrer à n'importe laquelle précédente (analyse du suffixe du fichier donné). On peut visualiser le travail effectué avec l'option `-v`.

Le prétraitement 79/104

Le préprocesseur (`cpp`) va traiter les directives (#) :

- definition et expansion des macros
  `#define MAX 10`
- inclusion de fichiers
  `#include <math.h>`
- compilation conditionnelle
  `#ifdef MAX`

L'option `gcc -E file.c` permet de visualiser le résultat du prétraitement

Traitement des macros

- Macros sans paramètre définies à l'appel du compilateur (-D) :
  `cc -DNB_LIGNES=48 fichier.c`

- Macros sans paramètres :

```
#define NB_LIGNES 24
#define NB_COLONNES 80
#define TAILLE_TAB NB_LIGNES * NB_COLONNES
```

Macros avec paramètre


```
#define abs(x) (( (x) > 0) ? (x) : -(x))

#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

int main() {
  int int i,j,k;
  i = min(j,k); //remplacé par i = (j < k)? j : k
  i = max(j,k); //remplacé par i = (j > k)? j : k
}
```

Substitution purement textuelle
Penser à bien parenthéser les arguments dans la macro

Usage délicat

- Attention aux erreurs difficiles à déceler :

  Si parenthèse juste après le nom
  alors macro avec paramètres
  Sinon
  macro sans paramètre

  ... un espace mal placé peut tout changer...

  - Exemple 1

    ```
    #define CARRE (a) a*a
    CARRE(2)    donne : (a) a*a(2)
    ```

  - Exemple 2

    ```
    #define CARRE(a) a*a
    CARRE(2)    donne : 2*2
    ```

Bien parenthéser

- l'exemple précédent a encore un problème :

  ```
  CARRE(a+b)       donne : a+b*a+b
  1.0/CARRE(a+b)   donne : 1.0/a+b*a+b
  ```

- Règle 1 : parenthéser les occurrences des paramètres formels
- Règle 2 : parenthéser le corps complet de la macro
  - Une définition de CARRE deviendrait :

    ```
    #define CARRE(a) ((a) * (a))
    CARRE(2)    donne : ((2) * (2))
    CARRE(a+b)  donne : ((a+b) * (a+b))
    ```

Inclusion de fichiers


```
#include "file.h"
#include <file.h>
```

- cpp remplace cette ligne par le contenu du fichier *file.h*
- Si on utilise " ", le fichier à inclure est un fichier utilisateur, il sera donc recherché dans le répertoire courant
- Si on utilise < >, le fichier sera recherché dans les répertoires standards du compilateur (`/usr/include`), et ceux indiqués par l'option `-I`

Compilation conditionnelle

- Directives de compilation conditionnelle :
  `#if #ifdef #ifndef #else #endif`
  les tests portent sur la valeur ou l'existence de macros
- Le préprocesseur analysera le résultat de ces tests pour conserver ou supprimer des lignes du fichier source

  Usage : éviter les inclusions multiples, tester l'existence des macros...
  - Éviter les multiples définitions :
    ```
    #ifndef VRAI
    #define VRAI 1
    #define FAUX 0
    #endif
    ```
  - Test et annulation d'une définition

    ```
    #ifdef MAX
    #undef MAX
    #endif
    #define MAX 20
    ```

    ```
    #if MAX != 20
    #define MAX 20
    #endif
    ```

Mise au point de programmes 86/104

- Usage classique : aider à outiller des instructions de débuggage

```
#ifdef DEBUG
  printf("taille du tableau: %d\n",taille);
#endif /*DEBUG*/
```

  - Le symbole DEBUG sera défini soit dans un fichier d'entête
  - ou au moment de la compilation grâce à l'option -D
    `gcc -DDEBUG principale.c module1.c`

Autre usage classique : pour gérer les risques d'inclusions multiples de fichier `.h` lors de gros projets composés de nombreux `.c` et `.h` et pour lesquels il n'est pas facile de déterminer le "bon" ordre dans lequel faire les `#include`. On laisse alors le préprocesseur se débrouiller. Pour cela, on protège chaque fichier `.h` par un test préalable.

Par exemple, si mon fichier s'appelle `module1.h`, j'ajouterais en début et fin de mon fichier :

```
#ifndef MODULE1_H
#define MODULE1_H
...
... (le contenu effectif)
...
#endif
```

Ainsi, dans mes fichiers de code, je peux faire des `#include "module1.h"` sans me préoccuper de savoir si il n'aurait pas déjà été inclus par un autre include précédent, le préprocesseur ne l'icluera qu'une seule fois.

12 Compilation séparée

Compilation séparée 88/104

- Afin de structurer une application il ne faut pas hésiter à la partitionner en plusieurs fichiers (fichier1.c fichier2.c ...)
- Ces fichiers seront compilés indépendamment afin de ne pas recompiler toute l'application à chaque fois.
- L'option `-c` du compilateur génère des fichiers objet `.o`
- Les fichiers `.o` seront liés par l'éditeur de liens qui complétera les références manquantes

```
gcc -c fichier1.c
  création de fichier1.o
gcc -c fichier2.c
  création de fichier2.o
....
gcc -o appli fichier1.o fichier2.o ...
  édition de liens et création d'un fichier exécutable nommé appli
```

L'outil `make` 89/104

- Permet la gestion des divers fichiers en évitant les re-compilations inutiles : gère les dépendances
- La commande `make` exécute les directives décrites dans le fichier `Makefile`
- Ces directives décrivent quoi faire pour obtenir la *cible* à partir de ses *dépendances*

```
fichier_cible: liste_des_fichiers_de_dépendance
 <TAB> commandes de mise à jour
```

- `make` compare les dates des fichiers et n'exécute que ce qui doit être mis à jour

Le fichier Makefile


```
# tp2.c ------------------+-> tp2.o -------------+-> tp2
#                        |                      |
# ./include/liste-note.h -'                      |
#                                               |
# ./src/liste-note.c -----+-> ./src/liste-note.o-'
#                        |
# ./include/liste-note.h -'

tp2: ./src/liste-note.o tp2.o
        gcc -g -o tp2 ./src/liste-note.o tp2.o

tp2.o: ./include/liste-note.h tp2.c
        gcc -I ./include -c -g tp2.c

./src/liste-note.o: ./include/liste-notes.h ./src/liste-note.c
        gcc -I ./include -c -g ./src/liste-note.c

clean:
        -rm -f *.o a.out core ./src/*.o
```

Noter le tiret devant la commande rm qui permet d'ignorer le code de retour ces commandes. Sans tiret et après un nettoyage complet du répertoire, demander à nouveau nettoyage conduirait à une erreur puisque sh ne saurait expanser `*.o` puisqu'il n'y a plus de tels fichiers. L'instruction `rm *.o` serait donc erronée, make s'arrêterait et n'exécuterait donc pas la seconde instruction `rm core`.

La commande make

- Exécute les commandes écrites dans le fichier `Makefile`
- La commande `make` peut comporter un nom de cible :
  ```
  ~$ make stp2.o
  ~$ make clean
  ```
- Si aucune cible n'est précisée, c'est la première du fichier `Makefile` qui sera prise en compte
  ```
  ~$ make
  ~$ make -f autre_fichier_Makefile
  ```

13 Les bibliothèques

Les bibliothèques 93/104

- Une bibliothèque vient avec un fichier `.so` ou `.a` (le code compilé) et un ou plusieurs fichiers `.h` (les includes)
- La bibliothèque standard, quelques fichiers d'entête :

  stdio.h : entrée / sorties, flux,
  errno.h : manipulation des erreurs (perror ...),
  math.h : fonctions mathématiques,
  signal.h : signaux Unix,
  string.h : manipulation de chaînes (strcat ...),
  time.h : manipulation de date et temps,
  ...

- Si l'on souhaite utiliser les fonctions de ces bibliothèques il faut le préciser en début de programme
  `#include <stdio.h>` va inclure le fichier `/usr/include/stdio.h`
- Consulter le manuel ! p.ex. : `~$ man stdio.h`

Exemples de bibliothèques :

— `string.h` apporte des fonctions de manipulation de chaînes :
  — comparaison : `strcmp() strncmp()`
  — recopie : `strcpy() memcpy() strcat()`
  — calcul longueur : `strlen()`
  — recherche occurrence : `strchr()`
— `ctype.h` apporte des fonctions de manipulation de caractères telles que :
  — `isalpha() isprint() isupper() tolower()` etc.
— `math.h` (compiler avec l'option `-lm`) apporte des fonctions telles que :
  — `exp() log() power() sqrt() sin( ) atan() fabs()`

14 Les fichiers

Les Entrées-Sorties 95/104

- La librairie standard offre des moyens d'effectuer des opérations d'E/S sur un *flot* (*stream*)
- Un fichier est toujours un flot d'octets, muni d'un accès séquentiel et d'un accès direct (au niveau octet)
- Association du flot à un fichier en l'ouvrant
- La fermeture du flot annule ce lien par un vidage de la mémoire tampon et une fermeture de la communication
- Les fonctions, types, et macros sont définis dans le fichier `stdio.h`

À bas niveau, on gère les E/S via un *descripteur* de fichier, un numéro qui référence un contexte dans le système d'exploitation. Ce contexte précise la nature de ce flux d'E/S (fichier ordinaire, périphérique, socket, pipe, etc.), le mode d'accès (lecture et/ou écriture), les permissions Unix, la position, etc.

La librairie standard définie une sur-couche, les flots, en gérant notemment des mémoires tampons bien pratiques.

En toute rigueur, les fichiers ne sont pas *typés*. Tout dépend de ce qu'en feront les applications. Cependant, si les octets qui composent le fichier sont tous des caractères imprimables, alors on a coutume d'appeler cela un fichier *texte*, et si non un fichier *binaire*.

Ouverture d'un fichier 96/104

- Avant d'accéder à un fichier il faut l'*ouvrir*

```
FILE *fopen(const char *name, const char *mode);
```

  - `name` : nom du fichier
  - `mode` : mode d'ouverture du fichier :
    - `"r"` pour la lecture
    - `"w"` pour l'écriture
    - `"a"` pour ajout
    - `"r+"` pour lecture et mise à jour

    Si on ouvre un fichier qui n'existe pas en "w" ou en "a", il est crée
    Si on ouvre en "w" un fichier qui existe déjà, son ancien contenu est perdu
- Retourne un pointeur sur un `FILE`
  ou `NULL` en cas d'erreur (pb. permission, fichier inexistant, ...)

Le type `FILE` est un type *opaque* : on ne sait pas de quoi il est fait, mais on sait comment l'utiliser avec les fonctions de la librairies. (Bon, il suffirait d'aller regarder à l'intérieur des fichiers include standards pour savoir, mais en fait on n'a pas besoin.)

Utilisation 97/104

- Avant d'accéder à un fichier il faut l'ouvrir

```
#include <stdio.h>

FILE *fp;
if ( (fp = fopen("myfile.data", "r")) == NULL) {
  perror("myfile.data");
  return ... ;
}
...
...
fclose(fp);
```

- Lorsque les écritures / lectures sont terminées sur un fichier on en informe l'OS par la fonction `fclose()` qui vide la mémoire tampon et ferme la communication

La fontion standard `perror()` (*print error*) affiche le message d'erreur éventuellement généré par une erreur système (ici la tentative d'accès incorrecte au fichier).

Lecture, écriture de fichiers texte 98/104

- Lecture caractère par caractère

  `int getc(FILE *fp);`

  lit un caractère dans le fichier et le retourne dans un int (cas du `EOF`)

  `getchar()` est identique à `getc(stdin)`

- Ecriture caractère par caractère

  `int putc(char c, FILE *fp);`

  `putchar(c)` est identique à `putc(c, stdout)`

```
int c;
FILE *fi, *fo;
...
while ( (c = getc(fi)) != EOF)
  putc(c, fo);
```

Entrées / sorties formatées dans un fichier 99/104

- Lecture

  `int fscanf(FILE *fp, const char *format, ...);`

  retourne le nombre d'items effectivement lus

  `fscanf(file, "%d %d ", &i, &j);`

  si le fichier contient 12 et 35, i et j receveront 12 et 35

- Écriture

  `int fprintf(FILE *fp, const char *format, ...);`

  retourne le nombre d'items effectivement écris

  `fprintf(stdout,...)` équivalent à `fprintf(...)`

Lecture, écriture de chaînes de caractères 100/104

- Écriture de chaînes de caractères

```
int puts(const char *s);
int fputs(const char *s, FILE *stream);
```

`puts()` écrit à l'écran
`fputs()` écrit dans un fichier
Les deux retournent le nombre de caractères écrits

- Lecture de chaînes de caractères

```
char * gets(char *s);
char * fgets(char *s, int n, FILE *stream);
```

`gets()` lit au clavier
`fgets()` lit dans un fichier au plus n-1 caractères
Lit jusqu'au caractère *newline* et range dans le tableau pointé par s auquel on rajoute `'\0'`

Pour les lectures, charge au programmeur de réserver suffisamment de place mémoire pour stocker les caractères lu.

Voir également des fonctions plus sophistiquées comme **`getline()`** (standard), ou encore **`readline()`** (encore plus riche, mais spécifique à GNU).

Copie d'un fichier texte 101/104

```
#include <stdio.h>
int main() {
  FILE *infp, *outfp;
  char c;
  if ( (infp=fopen("source.txt", "r")) != NULL ) {
    if ( (outfp=fopen("dest.txt", "w")) != NULL ) {
      while ( (c=getc(infp)) != EOF ) {
        putc(c, outfp);    // write to file
        putchar(c);        // write to screen
      }
      fclose(outfp);
      fclose(infp);
    } else {
      perror("dest.txt");
      fclose(infp);
    }
  } else
    perror("source.txt");
}
```

Lecture écriture de fichiers non texte

- Lecture de n'importe quel type de données

```
size_t fread(void *ptr,
             size_t size, size_t nitems,
             FILE *fp);
```

Transfère, depuis `fp`, n^{items} éléments de `size` octets vers une zone mémoire débutant à l'adresse `ptr`
Retourne le nombre d'objets réellement lus

- Ecriture de n'importe quel type de données

```
size_t fwrite(void *ptr,
              size_t size, size_t nitems,
              FILE *fp);
```

Transfère, vers `fp`, n^{items} éléments de `size` octets depuis une zone mémoire débutant à l'adresse `ptr`
Retourne le nombre d'objets réellement écrits

Copie d'un fichier binaire


```
#include <stdio.h>

main() {
  struct user_t {
    char name[15];
    unsigned age:
  } user;

  int size = sizeof(struct user_t);

  FILE *outfp, *infp;

  infp = fopen("users_in.dat", "r");
  outfp = fopen("users_out.dat", "w");

  while (fread(&user, size, 1, infp)) {
    fwrite(&user, size, 1, outfp);
    printf("name:%s age:%d\n", user.name, user.age);
  }

  fclose(outfp);
  fclose(infp);
}
```

Positionnement dans un fichier 104/104

- Déplacement direct à une position quelconque :

```
int fseek(FILE *stream, long offset, int origin);
```

`stream` : le fichier
`offset` : valeur du déplacement en nombre d'octets
`origin` : une macro parmis `SEEK_SET` (début), `SEEK_CUR` `SEEK_END`

Les fonctions citée précedemment travaillent en mode séquentiel. Chaque lecture ou écriture s'effectue à partir d'une position courante, et incrémente cette position du nombre de caractères lus ou écrits.

La position courante est initialisée en fonction du mode d'ouverture (typiquement soit au début, soit à la fin en mode ajout).