Introduction à C

Ken CHEN

Télécom Paris

18/08-01/09 (2021)

Préambule

- C est un langage *compilé*
- Programme *source* : une succession d'instructions (en texte ASCII) dans un fichier (avec l'extension `.c`)
- *Compilateur* (`gcc`, *etc.*) traduit le programme source en instructions machine.
- Environnement "naturel" : UNIX (Linux) : UNIX a été développé en C
- Sous Windows, on peut obtenir un compilateur gcc de diverses façons
  - Le WSL (*Windows Subsystme for Linux*) proposé par Windows 10 permet de faire tourner un système Linux.
  - On peut également installer GCC (Compilateur Gnu) sous l'environnement `MinGW-W64`
    - `MinGW` (Gnu minimal pour Windows) permet de bénéficier des applications Gnu (*open source*) sur Windows
  - On peut également faire tourner un système Linux sur une *machine virtuelle* à l'aide d'un outil de *virtualisation* comme *VMWare* ou *VirtualBox*

Caractéristiques de C

- Programmation *structurée*
  - Modularité, Conception *top-down*
- (relativement) Simple à apprendre
- Génère des codes machines efficaces
- Gérer des activités du niveau matériel
- 1er langage de programmation en termes de popularité
  - Régulièrement confirmée depuis 20 ans (`https://www.tiobe.com/tiobe-index/`)
  - Attention : En train d'être rattrapé par `Python`

Un programme “minimaliste” en C

```c
/*
  Un programme "minimaliste" en C
*/
// Les commandes "preprocessor"
// Au moins #include <stdio.h> (entree/sortie)
#include <stdio.h>
// Tout programme C contient 1 et 1 seule "main"
int main(void)
{
  // toute instruction se termine par un ";"
        printf("Bonjour \n"); // affichage
        return 1;
}
```

Squelette d'un programme en C

- Fichiers d'en-tête (`.h`) qui donnent la déclaration des
  - fonctions standards (bibliothèques)
  - le cas échéant, celle des fonctions issues d'autres modules
- Définition et déclaration des variables (globales)
- Définition des fonctions (dont, obligatoirement, `main`)
- Les commentaires peuvent se rajouter sous 2 formes
  - Entre `/*` et `*/`, s'étale sur 1 ou plusieurs lignes,
  - Derrière `//`, nécessairement la partie finissante d'une ligne

Identifiants, instructions

- Un *identifiant* (*identifier*) est une suite de caractères alphanumériques
  - Exemples : `monEcole, isGood`
  - Convention : `_` peut être utilisé, sauf comme 1er caractère,
- Les séquences réservées ou particulières
  - Les mots-clés : `int, char, return`, *etc.*.
  - Les opérateurs : `>, !=, ==`, *etc.*.
  - les séparateurs : espace, tabulation, *etc.*.
  - les valeurs numériques (`1.23`) et chaîne de caractères (`'A', ''abc''`)
- Une *instruction* (**statement**) est une unité logique d'exécution qui se termine par `;`
  - Exemple : `int i = 10;`

Les variables

- Une **variable** est un symbole correspondant à un segment de mémoire où l'on peut placer, à tout moment, une valeur caractérisée par
  - **pointeur** qui donne l'emplacement de cet espace mémoire
  - **type** qui fixe la taille et l'organisation binaire de l'espace mémoire
- Toute variable **doit être déclarée** avant sa 1ère utilisation
- Exemples
  - `char c; // la variable c est de type` **caractère**
  - `int i; // une variable de type` **entier**
  - `float x; // une variable de type` **flottant** `(réel)`
- Initiaisation
  - Il est recommandé d'initialiser une variable
  - Exemple :`int i = 10;`

Les types

- Types de base
  - Entier : `int, enum, char, short, long`, avec/sans `unsigned`
  - Flottant : `float, double, long double`
- Types dérivés
  - tableau : `int v[6];` est un vecteur d'entiers à 6 dimensions
  - structure (`struct`) est une combinaison de variables **hétérogènes** (ayant chacune son type)
  - pointeur a pour valeur une adresse mémoire.
    - Associé à une variable, elle *pointe* vers l'espace qu'occupe la variable
    - C'est sans doute l'élément le plus *emblématique* du langage C
  - fonction : voir section spécifique
- `void` : ne correspond à aucune valeur
- Remarque : Pour connaître l'occupation mémoire d'un type, utiliser l'opérateur `sizeof`
  - Exemple : taille exacte du type `int` : `sizeof(int)`

Entiers

- La nature de certains types d'entiers dépend du matériel/compilateur
- `char` : 1 octet, plage $-128, \ldots, 127$ **ou** $0, \ldots, 255$
  - `signed char` : plage $-128, \ldots, 127$
  - `unsigned char` : plage $0, \ldots, 255$
  - Remarque : En pratique, c'est pour un caractère (code ASCII). Mais ça s'utilise plus souvent avec un *pointeur* pour une *chaîne de caractères* avec (`char*`)
- `int` : peut être codé comme `short` (2 octets) **ou** `long` (4 octets)
- `short` : 2 octets, $-32\,768, \ldots, 32\,767$
  - `unsigned short` : $0, \ldots, 65\,535$
- `long` : 4 octets, $-2\,147\,483\,648, \ldots, 2\,147\,483\,647$
  - `unsigned long` : $0, \ldots, 4\,294\,967\,295$

Flottant (Réel)

- Les réels se déclarent avec `float, double, long double`
- Exemple : `float r=1.2; double r=3.456;`
- Codage (en bits) et précision minimale (en **digits**)

Type	Taille	Signe ($\pm$)	Mantisse	Exposant	Précision
`float`	32	1	23	8	6
`double`	64	1	52	11	15
`long double`	80	1	64	15	17

- Attention : Il est préférable d'utiliser `double` au lieu de `float`

Entrée et Sortie

- C traite tout flot d'information comme un *fichier* (**file**)
- En particulier pour les flots d'entrée et de sortie
  - Entrée par défaut (par défaut clavier) : `stdin`
  - Sortie par défaut (par défaut écran) : `stdout`
  - Affichage d'erreur par défaut (par défaut écran) : `stderr`
- Fonction principale pour la sortie : `printf`
- Fonction principale pour la saisie : `scanf`
- Il s'agit des fonctions d'écriture et de lecture avec *formats*

la fonction `printf`

- Fonction d'écriture avec formats
- Forme générique : `printf(format, expr1, expr2, ...)`
  - `expr1, expr2, ...` : l'ensemble des valeurs à afficher
  - `format` : une chaîne de caractères indiquant les formats d'affichage, dans l'ordre, des expressions
- Exemple : avec l'instantiation `int myInt=1;`,
  - `printf("myInt vaut %d\n", myInt)`
  - `%` indique un format, `d` pour entier, `\n` à la ligne
  - `printf("myInt= %8d, codé sur %2d octets \n", myInt, sizeof(int) );`
  - Afficher la valeur de `myInt` sur 8 digits et donne sa taille sur 2 digits

Format

- les (principaux) éléments d'un format sont donnés dans l'ordre suivant :
  1. Obligatoire : `%` (début d'un format)
  2. *Facultatif* : un nombre donnant la largeur du champ
     - Pour un flottant, `l.d` avec `d` nombre de digits après virgule (ou significative pour `g`)
  3. *Facultatif* : une lettre indiquant la précision du nombre
     - `h` (devant `d, i, o, u, x`) : `short`
     - `l` (devant `d, i, o, u, x`) : `long`
     - `l` (devant `e, f, g`) : `double`
     - `L` (devant `e, f, g`) : `long double`
  4. Obligatoire : indication du format
     - `d` pour les `int` en notation décimale
     - `x` pour les `int` en notation hexadécimale
     - `c` pour les `char`
     - `s` pour les chaînes de caractères (`char*`)
     - `e` pour les réels en notation scientifique (mantisse, exposant)
     - `f` pour les réels sous forme "flottant" avec par défaut 6 digits après "`,`"
     - `g` donne un affichage optimal des grands réels (flottant vs scientifique)
     - `p` pour un pointeur (adresse mémoire)

La fonction scanf

- Fonction de lecture avec formats
- Forme générique : `scanf(format, &var1, &var2, ...)`
  - `&var1, &var2, ...` : l'ensemble des variables à lire, `&var1` indiquant l'adresse de `var1`
  - `format` : une chaîne de caractères indiquant les formats de lecture, dans l'ordre, des variables, *idem* `printf`.
- Exemple : `"%lf"` pour une variable `double myDouble;`

```
printf("Entry a double \n");
scanf ("%lf", &myDouble);
printf("myDouble= %f\n", myDouble );
```

- `scanf` étant une fonction, elle renvoie son résultat (un `int`, nombre de variables correctement lues)
- Remarque : Il peut être utile de contrôler le résultat d'exécution de `scanf`

Les opérateurs

- Arithmétiques : addition, soustraction, *etc.*.
  - `x = 2 +3.0;`
- Relationnels : comparaison (supérieur, égale, *etc.*.)
  - `(x != y)`
- Logiques : ET, OU, *etc.*
  - Opération AND : `(x && y)`
- Binaires : Et, ou, ou exclusive au niveau binaire *etc.*
  - AND binaire (de tous les bits) entre `a` et `b` : `(a & b)`
- Affectation (`=`) et *opération/affectation combinées*
  - Rajouter 2 : `(x+=2)`
- Particuliers
  - Pointeur (`*`) et adresse (`&`)
  - `sizeof()`
  - Expression conditionnelle (`? :`) : équivalent à `if ... else`

Opérations arithmétiques

- les 4 opérations de base : `+, -, *, /`, ordre d'exécution conventionnel.
  - pour les entiers et les flottants (et même les pointeurs)
  - Attention : Opération sur les entiers donnent résultats en entiers
  - `5/2` donne `2`, `5-2*3` donne `-1`,
  - `5/2.0` donne `2.5`, `5-2.0*3` donne `-1.0`,
- Modulo (`%`) : `a % m`
  - pour les entiers,`5 % 2` donne `1`
- les opérateurs `++` et `--`
  - il s'agit d'une expression (`x+1`) qui produit une *valeur*.
  - `x++` incrémente `x` **après** l'évaluation de l'expression
    - Pour `x=5`, `x++` vaut `5` et `x` devient `6`
  - `++x` incrémente `x` **avant la fin de** l'évaluation de l'expression
    - Pour `x=5`, `++x` vaut `6` et `x` devient `6`
  - *idem* pour `--`

Comparaison

- Evaluation d'une expression du type `(x operator y)` qui donne `true` (1) ou `false` (0)
- Operateurs avec notation "habituelles" : `>, >=, <, <=`
  - `(x >= y)`
- Operateur "égalité" : `==`
  - `(x == y)`
  - Attention : Surtout ne pas le confondre avec `=` qui réalise une **affectation de valeur**.
- Operateur "différent de" : `!=`
  - `(x != y)`
- mélange possible d'entiers et flottants

Opérations logiques

- Opération AND (`&&`) :
  - Forme : `expr1 && expr2`, `(5>2) && (5<1)`
  - Vaut 1 si ni `expr1` ni `expr2` n'est nulle
- Opération OR (`||`) :
  - Forme : `expr1 || expr2`, `(5>2) || (5<1)`
  - Vaut 1 si `expr1` **ou** `expr2` n'est nulle
- Operation NEGATION (`!`) :
  - Forme : `! expr`, `!(5>2)`
  - **Inverser** (0 vs 1) la valeur de `expr`
- Remarque : L'expression à gauche est évaluée d'abord et permet de **ne pas** évaluer la 2nde expression
  - Pratique : efficace et protecteur
- Remarque : Il n'y a pas d'opératteur "ou exclusive" (XOR) prédéfini
  - Un XOR nécessiterait l'évaluation des 2 expressions

Les structures de contrôle

- Une **structure de contrôle** (ou structure conditionnelle) permet l'exécution *conditionnelle* d'un bloc d'instructions :
- En C, il y a deux catégories de structures de contrôle.
  - Décision séquentielle, du genre "IF .. THEN .... ELSE"
  - Boucle, du genre "FOR" ou "WHILE"
- Décision séquentielle :
  - `if` ... `else`
  - variantes : `if`, `switch`, l'opérateur `?` ... `:`
- Boucles :
  - `for`
  - `while`
    - variantes : `do` ... `while`
  - instructions particulières : `break`, `goto`, `continue`

Formes génériques

- Instruction `if` (condition) {bloc} `else` {bloc}
- Instruction `switch` avec `case`
- Instruction `while` (condition) {bloc}
  - variante : `do` {bloc} `while` (condition)
- Instruction `for` (condition) {bloc}

Bloc d'instruction ({ ...}) et conditions

- un bloc d'instructions est encadré par les accolades ({ ...})
  - En cas d'instruction simple, les accolades peuvent être omises
  - En cas d'instructions multples, les accolades sont obligatoires
  - Conseil : garder les systématiquement
- (expression donnant une) Condition
  - Dans le monde réel : VRAI (*true*) ou FAUX (*false*)
  - Dans C : FAUX (*false*) : **0**, VRAI (*true*) toute valeur (entière) non nulle
  - Expression d'une condition
    - Opérations arithemétiques, relatioonnelles, logiques,
    - Une expression plus complexe

if ... else

- `if` (condition) {bloc} `else` {bloc}
- Exemple

```
if (a>0) {
  i=12;
}
else {
  i=23;
}
```

switch

- Syntaxe (par un exemple)

```
switch(niveau) {
       case 'A' :
          printf("Bravo!\n");
          break;
       case 'B' :
       case 'C' :
          printf("Bien!\n" );
          break;
       default :
          printf("Cas non prevus\n" );
    }
```

Boucle `for`

- Forme générique

```
for (init; condition; modif) {
   bloc d'expressions;
}
```

  - `init` est une expression qui s'exécute **au début et une seule fois**.
  - `condition` est l'expression à vérifier
    - Si `condition` N'est PAS vérifiée (0), la boucle s'arrête
    - Sinon, on rentre dans la boucle
  - La boucle se déroule comme suit :
    1. bloc d'expressions, **puis**
    2. l'expression `modif`, **puis**
    3. vérification de la `condition`

- Exemple

```
for (int i=0; i<3; i++) {
  printf(''Hello\n'');
}
```

Boucle while

- `while (...) {...}` (tant que ... (faire) ...)
- Exemple :

```
int i=9;
while (i>0) {
  i--;
}
```

- Attention : Tant que la condition reste inchangée, ça boucle toujours sur le même bloc
  - Il faut toujours s'assurer que la condition est modifiée dans la boucle avec la certitude de pouvoir sortir de la boucle.

Boucle do while

- `do { ... } while (...)`
- Exemple

```
int i=-1;
do  {
  i=1;
}
while (i<0);
```

- Différence avec `while` : le bloc d'instructions est exécuté au moins une fois.
- Attention : Tant que la condition reste inchangée, ça boucle toujours sur le même bloc

Fonction

- La forme générique d'une fonction

```
return_type fname(lparameter) {
    bloc d'expressions
}
```

  - `return_type` : type de la valeur (résultat) renvoyée par la fonction
    - valeur renvoyée par `return`
    - Si pas de valeur : type est `void` (une "procédure")
  - `fname` : identifiant de la fonction
  - `lparameter` : liste des paramètres (**reçus**)
    - Si pas de paramètres, mettre `void` (ou vide)

- Exemple :

```
int main() {
    printf(''Bonjour\n'');
    return 1;
}
```

Fonction (exemple)

- Exemple : La fonction ci-dessous réalise en fait l'addition de deux nombres.

```
int myShift(int a, int d) {
    a = a + d;
    return a;
}
```

Appel d'une fonction

- Utilisation (**appel**) d'une fonction :
  - Par son nom (identifiant) avec la liste des paramètres
  - affectation de la valeur (le cas échéant)
- Exemple :

```
...
int myShift(int a, d) {
    a = a + d;
    return a;
}
...
int main() {
    int j = myShift(6, 2);
    printf("shift de 6 par 2=%d\n", j);
    printf("%d\n", myShift(1, 2));
    return 1;
}
```

Variable globale, Variable locale

- Une **variable locale** est définie et valable au sein de la fonction où elle est définie
- Une **variable globale** est définie et valable au sein de tout le programme
  - Elle peut être définie partout, en dehors des **fonctions**
  - Généralement, elle est définie au début du programme
- Une variable locale peut posséder le même identifiant qu'une variable globale.
  - Dans un tel cas, c'est la variable locale qui prévaut (c'est-à-dire la valeur de l'identifiant sera celle de la variable locale).
- Remarque : Il est préférable de réduire le nombre de variables **globales** au strict minimum

Tableau

- Un tableau est une suite de variables du même type, indexées par `[i]`.
- Forme générique : `type nom_tableau[dimension]`
- Exemple `int v[3];` définit un tableau (vecteur) composé de 3 éléments entiers
- Initialisation d'un tableau :
  - Forme générique : avec un bloc (entre `{...}`) de valeurs constantes
  - `int v[3] = {3, 5, 8}`
  - Remarque : `int v[] = {3, 5, 8}` **crée** puis *initialise* un table à 3 dimensions.
- Elements d'un tableau :
  - Les éléments se manipulent individuellement par leur index `[i]`.
  - Le 1er élément est indexé par `0` (donc le dernier par `dimension-1`)
  - Exemple : Les 3 éléments de `int v[3]` sont `v[0]`, `v[1]` et `v[2]`

Tableau (suite)

- Il n'y a pas, formellement, de tableau *multidimensionnel* dans C
- Un tableau *multidimensionnel* est obtenu comme un tableau de tableaux :
- Exemples : `int m[3][3];` donne une matrice carrée 3x3 $M = [m_{i,j}]_{i \in \{1,2,3\}, j \in \{1,2,3\}}$.
  - L'élément $m(1,3)$ est donné par `m[0][2]`
- Tableau et fonction
  - Un tableau peut être "passé" à une fonction comme paramètre
  - Une fonction peut renvoyer un tableau comme résultat
  - Nécessite la manipulation des `pointeurs`

Pointeurs

- le concept de **pointeur** est sans doute le concept le plus important et emblématique de C
- Un pointeur est une adresse qui donne accès à **tout**
- Rappel : Principe des ordinateurs (Von Neumann)
  - Tous les éléments d'un programme résident dans l'espace mémoire
- C'est en particulier vrai pour toute variable
- A chaque variable, on sait lui associer une adresse mémoire
- Dans C, l'adresse est indiquée par `&`
  - Soit `int i;`, `&i` donné l'adresse de `i`
  - On sait donc exactement que le contenu de `i` se trouve dans les `sizeof(int)` octets à partir de `&i`.
- Question : comment y accéder ?
- Réponse : à l'aide de cette adresse (`&i`)
- D'où le concept de **pointeur**

Pointeurs (suite)

- Un **pointeur** (`p`) est donc une **variable** qui donne **l'adresse** d'une **autre variable** (`v`).
- Question : Combien d'octets derniers `p` correspondent réellement à `v` ?
- Réponse : il faut préciser le **type** associé à tout pointeur.
- Forme générique : `type *nom;` ou `type* nom;`
  - `nom` est l'identifiant de ce pointeur (une *variable* dont la valeur est une adresse)
  - `*nom` "pointe" sur un espace mémoire qui débute avec `nom` et qui a pour taille et format ceux de `type`
- Exemple : avec `int *ip;`, `ip` est une adresse et `*ip` est un `int` .

```
int  i=20;
int  *ip; //pointeur sur int
ip=&i; // ip pointe sur i
// On aura 20 et 20
printf("i=%d, *ip=%d\n", i, *ip);
```

Pointeurs (suite)

- Pointeur `NULL` : constante valant 0
  - Permet de neutraliser un pointeur : aucune adresse physique n'est nulle (0)
  - éviter l'affectation au hasard d'une valeur à un pointeur (donc pointage vers une zone mémoire)
  - `Usage : int* ip=NULL;`
- Opérations sur les pointeurs
  - Un pointeur est une variable, certaines opérations sont possibles
  - Arithmétique : `++, --, +, -`
  - Comparaison : les mêmes que pour `int`
- Sélection d'applications du concept du pointeur
  - Chaînes de caractères
  - Tableaux
  - Transmission de paramètres à travers une fonction

Opérations arithmétiques sur pointeurs

- Opérations : `++, --, +, -`
- Particularité : **l'unité** est `sizeof(type)` où `type` est le type du data associé au pointeur
- Exemple
  - avec `int v[5]={1, 2, 3, 4, 5};`
  - et `int *ip;`
  - `ip = &v[0];` rend `ip` pointer sur `v[0]`, c-à-d `*ip` vaut 1
  - Après `ip = ip+2;`, le pointeur `ip` est "déplacé" de 2 "cases" (ici 2 `int`)
  - `ip` pointe sur `v[2]`, c-à-d `*ip` vaut 3

Chaînes de caractère (*string*)

- Une chaîne de caractère est un tableau de caractères
- Un tel tableau se termine par le caractère nul (\0) noté `null`.
- Exemple :
  - `char msg[]="TPT";` déclare une chaîne de caractères `msg[]` et l'initialise avec la valeur `TPT`
  - qui équivaut à `char msg[4] = {'T', 'P', 'T', \0};`
  - C'est un vecteur à 4 éléments dont l'élément terminal `null` (\0)
  - Remarque : Une chaîne est comprise entre `"..."`, un caractère seul est cadré par `'.'`.

Utilisation des string

- Déclaration : `char mystr[80];` crée la chaîne `mystr` pouvant accueillir 79 caractères non `null`
- Initialisation : `char mystr[80] = "TPT";`
- Input/output : format `%s`,
  - `printf("le contenu de mystr est %s \n", mystr)` ;
  - `scanf("%s", mystr)` ; (Attention : il n'y pas de `&` devant `mystr`)
- Quelques fonctions utilitaires (avec `#include <string.h>`) pour les strings (`s1, s2`)
  - `strcpy(s1, s2)` le contenu de `s2` est mis dans `s1`
  - `strcat(s1, s2)` le contenu de `s2` est *rajouté* à celui de `s1`
  - `strcmp(s1, s2)` donne (selon l'ordre ASCII)
    - `0` si `s1` et `s2` sont identiques
    - une valeur **négative** (valeur type `-1`) si le 1er caractère non concordant de `s1` est inférieur à celui de `s2`
    - une valeur **positive** (valeur type `1`) si le 1er caractère non concordant de `s1` est supérieur à celui de `s2`

Structures

- Les variables vues jusqu'à présent sont de type
  - soit un type élémentaire (`char`, `int`, `double`, *etc.*)
  - soit une suite de variables du même type : les tableaux
- Le monde physique réclame une modélisation des informations aux *attributs* multiples
- Exemple : Un individu peut être modélisé par
  - son nom (chaîne de caractères) et
  - son age (un entier)
- Le concept de **structure** (`struct`) permet de créer, selon les besoins, de nouveaux **types** de données aux *champs* multiples et *hétérogènes*
- Forme générique

```
struct nom_structure {
type champ1;
type champ2;
...
};
```

Structures

- Exemple : définir une structure (`Usager`) puis déclarer 2 variables (`u1, u2`)

```
struct Usager {
char     nom[40];
int      age;
double   taux;
} u1;
struct Usager u2;
```

- **Accès aux champs** : via le séparateur **point** `.`
  - Exemple : u1.age=23 ; strcpy(u1.nom, "Dupont") ; ;
- Tableau de `struct` : tout à faire possible, Exemple :
  - Déclaration : `struct Usager ut[2];`
  - Accès : `ut[0].age=23;`
- Un pointeur sur une `struct` : tout à faire possible, Exemple :
  - Déclaration : `struct Usager *pusg;`
  - Accès aux champ (**se fait via** `->`) : `pusg->age=23;` ;

typedef

- Avec `struct`, il est possible de définir de nouveaux types de données
- Dans la déclaration, nous avons dû mettre systématiquement
  - `struct nom_structure var;`
- `typedef` permet de donner un nom à tout type
  - En pratique, il permet de ne plus mentionner `struct`
- Exemple : En reprenant l'exemple précédent

```
typedef struct  Usager {
char    nom[40];
int     age;
double  taux;
} Usager;
Usager u1, u2;
```

Allocation dynamique de mémoire (`malloc`)

- La déclaration de chaque variable permet au compilateur de connaitre l'espace mémoire requis
  - Avec `int i;`, `i` occupe `sizeof(int)` octets.
  - Avec `char s[6];`, `s` occupe 6*`sizeof(char)` octets.
  - avec `struct {int i; char s[60];} st`, `st` occupe `sizeof(int)`+6*`sizeof(char)` octets.
- Cet espace est **réservé** à la variable
- Pour un tableau, en particulier un *string*, quel est la "bonne" taille **?**.
- Remède : allocation dynamique :
  - **malloc** pour la réservation
  - **free** pour la libération
- Exemples d'applications
  - gestion d'une **liste chaînée**
  - gestion d'un **arbre**

malloc et free

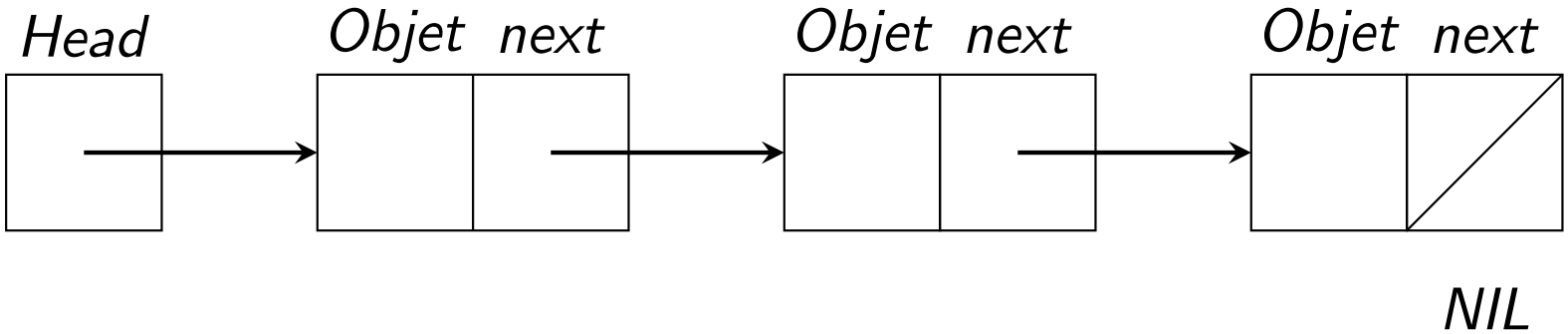
- `malloc` : `void *malloc(int n);`
  - Réservation d'un espace mémoire de `n` octets
  - Renvoie l'adresse de l'espace en cas de succès
  - Renvoie **NULL** (0) si **échec**.
- `realloc` : `void *realloc(void *address, int new);`
  - Modifier l'occupation de l'espace mémoire avec la nouvelle taille `new`
  - Peut augmenter ou diminuer.
  - Renvoie **NULL** (0) si **échec**.
- `free` : `void free(void *address);`
  - libère l'espace mémoire spécifié par `address`

Listes chaînées

- Besoin : Mémorisation d'une collection d'objets (souvent du type `struct`) dont le nombre varie dynamiquement.
- Un `tableau` (des objets) n'est souvent pas approprié, en raison de l'évolution dynamique de cette collection :
  - Prévoir un tableau de grande dimension : risque de gaspillage d'espace mémoire
  - Prévoir un tableau de petite taille : risque de ne pas pouvoir récupérer de nouveaux objets si leur nombre dépasse la taille prévue
- Une **liste chaînée** (*linked list*) est un procédé idéal pour gérer de telles situation
  - Pour chaque nouvel objet, allocation d'un espace mémoire (via `malloc`)
  - Pour chaque objet sortant de la collection, libération de l'espace mémoire occupé (via `free`)
- Liste chaînée versus Tableau
  - Avantage : Gestion souple de l'évolution de manière économique en termes d'espace mémoire consommé
  - Inconvenient : opération plus complexe et accès non direct (cf transparents suivants)

Cas d'une liste avec chaînage simple

- Chaque maillon de la chaîne contient
  - Le composant principal (**Objet**)
    - En pratique, **Objet** peut être une structure de donnée quelconque
  - Un pointeur (**next**) qui donne accès au maillon suivant
- La liste se repère par un pointeur (**Head**) qui donne accès au **premier maillon**
- Il faut également neutraliser (avec `NULL`) le pointeur du **dernier** maillon



Codage en C

- Les maillons se définissent à l'aide de `struct`

```
typedef struct nom_structure {
type Objet;
struct  nom_structure* next;
} nom_structure;
```

- Exemple : Un maillon dont l'objet contient deux éléments nom et age

```
typedef struct  Usager {
    char     nom[40];
    int      age;
    struct   Usager* next;
} Usager;
```

- Chaque maillon est créé avec `malloc`
  - Exemple : `up=malloc(sizeof(Usager))` crée un maillon repéré par son pointeur `up`. Bien entendu, il faut contrôler le résultat de `malloc` .
- L'adresse du premier maillon doit être mémorisée à part
  - Exemple : Avec l'exemple précédent, après `up=malloc(sizeof(Usager))` effectuer `1stUser = up` pour mémoriser ce 1er maillon

Arguments de `main`

- Appel d'un programme avec des **paramètres**
- Via les paramètres de la **fonction** `main`
- Syntaxe : `int main(int argc, char **argv)`
  - `int argc` : nombre d'arguments (paramètres) **dont** le nom du programme
  - `char **argv` donne en réalité `argc` *string*
    - `argv[0]` : nom du programme
    - `argv[1]` à `argv[argc-1]` : paramètres (si `argc`>1)
  - Autre syntaxe : `int main(int argc, char *argv[])`
- Exemple : avec `test coucou`
  - `argc` vaut 2
  - `argv[0]` vaut `"test"`, `argv[1]` vaut `"coucou"`

Fichiers

- Rappel : C gère tous les flots avec son environnement comme des fichiers (**file**)
  - Périphériques comme **file** : écran=`stdout`, clavier=`stdin`
- Manipulation de "vrais" fichiers :
- le type **FILE** définit un fichier
- Exemple : Déclaration d'un fichier (par pointeur) : `FILE *fp;`
- Ouverture (**fopen()**), fermeture (**fclose()**)
- Plusieurs fonctions pour lecture et écriture
  - Par unité de **char**, lecture (**fgetc()**) et écriture (**fputc()**)
  - Par "string", lecture (**fgets()**) et écriture (**fputs()**)
  - Généralisation de **printf()** et **scanf()**
    - **fprintf()** et **fscanf()** pour tout fichier

Ouverture, fermeture

- Déclarer le fichier à manipuler (sous forme de pointeur) : `FILE *fp`
- Ouverture : `FILE *fopen(char *filename, char *mode);`
  - Divers modes d'ouverture (lecture seule, lecture/écriture *etc.*)
- Fermeture : `int fclose(FILE *fp);` tout simplement.
- Modes d'ouverture
  - `r` : lecture, `r+` : lecture/écriture
  - `w` : écriture **depuis le début**, crée le fichier le cas échéant.
  - `a` : similaire à `w`, écriture **à partir de la fin actuelle**.
  - `r+, w+, a+` : lecture/écriture

sprintf

- `sprintf` est une fonction utile car elle permet la **création dynamique** des chaînes de caractères.
- Fonctionnellement : comme `printf` (impression **formaté**).
- Syntaxe : comme `fprintf`, un **string** à la place d'un **file**
- Exemple : Les 2 approches donnent le même résultat

```
// 1: via printf
printf("i=%d, x=%g\n", i, x);
// 2: via sprintf
sprintf(s, "i=%d, x=%g\n", i, x);
printf("%s", s);
};
```

- Remarque : On pourrait voir `sprintf` et `printf` comme 2 cas particuliers de `fprintf` correspondant respectivement au `string` et `stdout`.

Modulariser (*Link*)

- Il est ni pratique ni souhaitable de gérer **un seul** fichier source qui contient tout le code
- Il est une bonne pratique de répartir les lignes de codes dans plusieurs fichiers source, Chacun de ces fichiers :
  - contient une ou plusieurs identificants (fonctions, variables, types) qui sont définis spécifiquement dans le fichier
  - peut être compilé *individuellement* (création d'un ficher (`*.o`)
- Pour qu'un identifiant (une fonction en particulier) défini dans un fichier source (`fa.c`) puisse être utilisé dans un autre (`fb.c`) :
  - il faut qu'il soit *déclaré* (c-à-d *connu*) de `fb.c`
  - c'est le rôle du fichier `*.h`, en l'occurrence, `fa.h`, qui contient la déclaration de tous les identifiants définis dans `fa.c`
  - Le fichier `fa.h` doit être **inclus** (via `#include "fa.h"`) dans `fb.c`
- Il suffit, *in fine*, d'effectuer une compilation *ultime*, qui réalise le **lien** (*link*) entre tous les fichiers `*.o` pour obtenir le code **exécutable**

Etapes et syntaxe

- Situation : codes répartis dans deux fichiers : `fa.c` et `fb.c`, avec leur fichiers `*.h` associés
- Compilation individuelle :
  - `gcc -c fa.c` qui produit `fa.o`
  - `gcc -c fb.c` qui produit `fb.o`
- Compilation finale (*link*) :
  - `gcc fa.o fb.o -o nomprg` qui produit un **exécutable** avec le nom `nomprg`

Modulariser : remarques

- La modularité est un élément essentiel dans tout développement logiciel
- Au delà de son efficacité, c'est également un reflet de la vision sémantique et du découpage fonctionnel du programme
- Il existe de véritables environnement dit *IDE* (Integrated developpement environement), comme par exemple `Eclipse`
- De manière plus directe, il y a aussi l'outil `makefile`
- La présente présentation a pour (très humble) objectif de vous montrer
  - Le principe et procédé de la modularisation (`*.c, *.h`), ainsi que
  - un moyen élémentaire de sa réalisation via `gcc`