

Une (trop) courte introduction à C

1. Introduction
2. Structure d'un programme C
3. Objets de base du langage
4. Pointeurs et tableaux
5. Fonctions et paramètres
6. Chaînes de caractères et E/S formatées
7. Type énumérés et types structurés
8. Allocation dynamique
9. Structuration des programmes
10. Entrées/Sorties sur les fichiers
11. Chaîne de développement : gcc/make

1. Pourquoi va-t-on utiliser C ?

- JAVA est un langage évolué
 - ▶ Fournit des abstractions éloignées des objets physiques.
 - ▶ Pratique quand on veut s'abstraire de la machine
 - Pour du Génie Logiciel
- Le langage C est proche de la machine
 - ▶ On garde un contrôle relatif sur ce qui se passe en mémoire.
 - ▶ On manipule les objets physiques de la machine
 - Pratique quand on veut faire du « système »
- Les principaux systèmes d'exploitation sont écrits en C
 - ▶ Unix, GNU/Linux, etc.
- Tout programmeur doit avoir des notions de C.
 - ▶ Même si il a beaucoup de défauts, c'est un langage qui n'est pas prêt de disparaître ...

- Plutôt qu'une présentation de C, on va surtout insister sur les différences avec JAVA.
 - ▶ Ceci se fait bien car JAVA et C ont presque la même syntaxe
- La maîtrise d'un langage comme C est longue et difficile
 - ▶ Nécessite une bonne dose de pratique
- Il y a beaucoup de documentation sur C
 - ▶ Polycopié très complet de J.L. Nébut (cours Istic C81)
([accessible sur ndc.istic.univ-rennes1.fr](http://ndc.istic.univ-rennes1.fr))
 - ▶ Les fonctions des bibliothèques sont documentées
 - Exemple : `man fprintf`
 - ▶ Google sera (souvent) votre ami

1. Présentation du langage C

- Originellement conçu pour écrire le système Unix
 - 90-95% du noyau est écrit en C
- Développé par Thompson & Ritchie en 1972



- Normalisé en 1989 par un comité de l'ANSI
- Langage faiblement typé

Chapitre 2 :

Structure d'un programme C

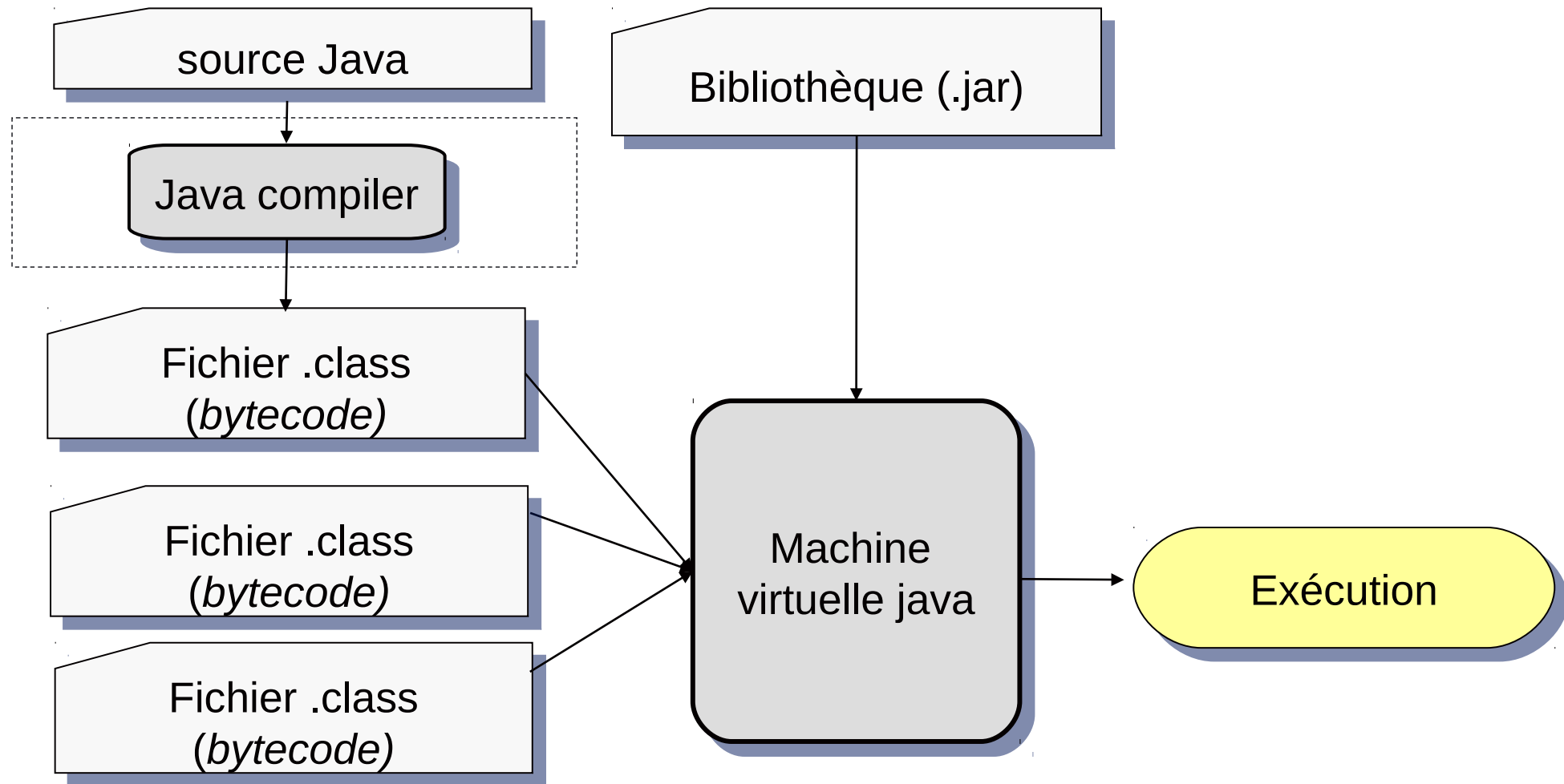
Flot de compilation Java

- ▶ Les fichiers **.java** sont des fichiers contenant du texte qui sont traduits en fichier **.class** par le compilateur **javac**
- ▶ Les fichiers **.class** contiennent du *bytecode* java : un langage proche du langage machine (mais pas en lang. machine).
- ▶ Le *bytecode* peut être exécuté sur n'importe quelle machine disposant d'une machine virtuelle Java (portable mais + lent)

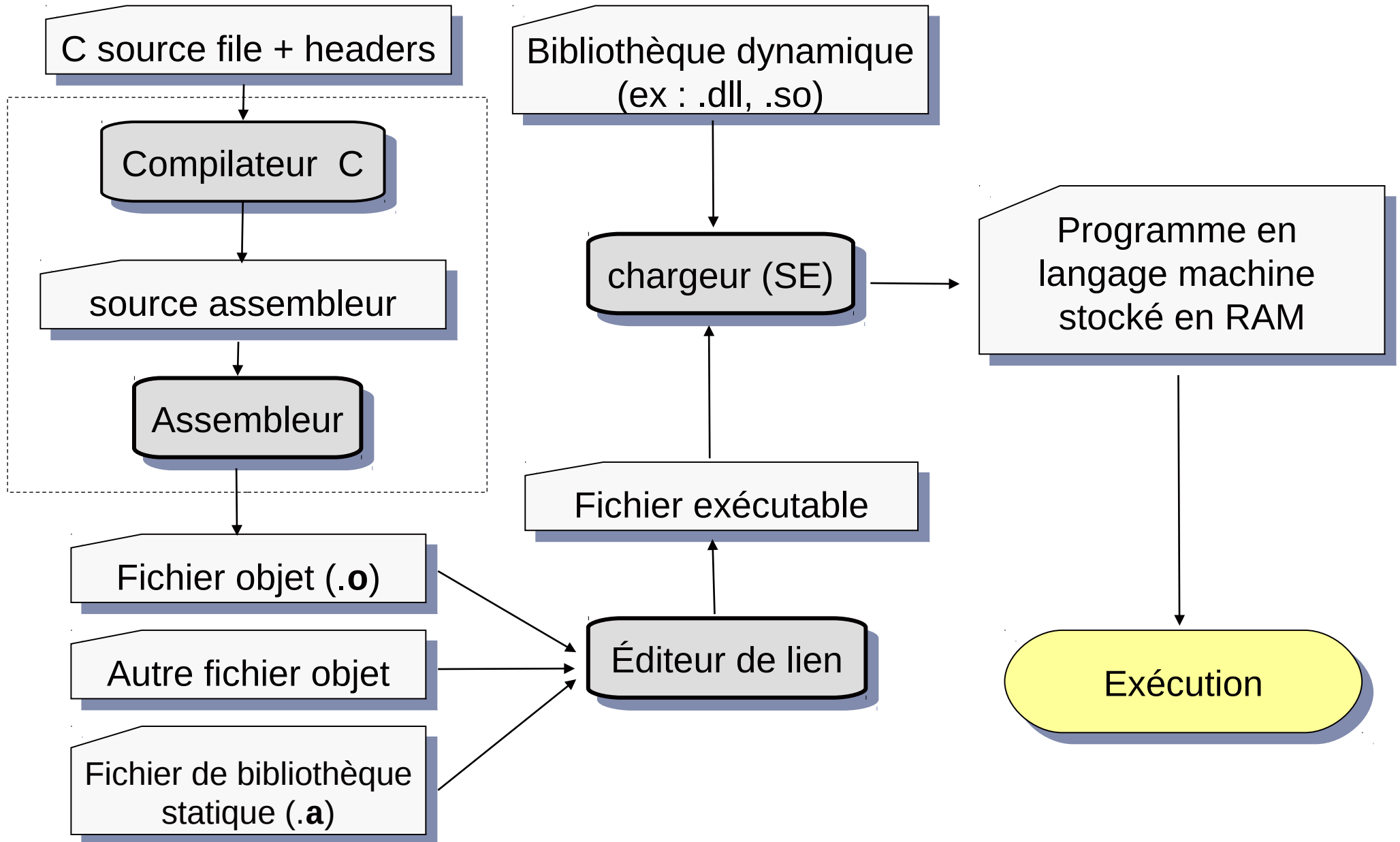
Flot de compilation C

- ▶ Une programme est formé d'un ou plusieurs fichiers C qui est/sont traduit(s) en modules objets (**.o**) par le compilateur C
- ▶ Ces modules objets contiennent des instructions machines et sont fusionnés (reliés) pour former un *fichier exécutable*.
- ▶ Cet exécutable ne peut fonctionner que sur une machine compatible avec la machine hôte (+rapide mais -portable)

2. Chaîne de compilation Java



2. Chaîne de compilation C



- L'appel au compilateur met en jeu 3 logiciels
 - ▶ Un **précompilateur** (CPP) qui joue le rôle de macro-processeur
 - ▶ Le **compilateur/assembleur** C (ici GCC)
 - ▶ L'**éditeur de lien** (ici LD/GCC)
- En Java, on découpe un programme en classes
 - ▶ La liaison entre les classes est faite à l'exécution, on dit que c'est une **liaison dynamique**.
- En C, on découpe le programme en **modules**
 - ▶ La liaison entre les modules est faite lors de l'étape de l'édition de lien qui sera étudiée *au second semestre*, on parle de **liaison statique**

Contenu d'un module

- ▶ Des **directives** pour le pré-processeur
- ▶ Des **déclaration externes** au module
- ▶ Les **variables statiques** gérées par le module
- ▶ Des **sous-programmes** qui agissent sur ces variables

Remarques

- ▶ Toutes les variables non locales et les fonctions sont globales (et donc exportées)
- ▶ Le langage n'impose pas d'ordre dans les déclarations, **mais tout élément doit être déclaré avant son utilisation.**

2. Exemple

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

```
#define PI 3.14
```

```
int varGlob1 = 0 ;
int varGlob2 = 12 ;
```

```
extern int uneFonction (int a)
```

```
int uneAutre (int a, char b) {
    if (b=='\n') {
        return -7 ;
    } else {
        return a-1
    }
}
```

```
int main() {
    int varLoc = 5 ;
    varGlob1 = uneFonction(varLoc) ;
    varGlob2 = uneAutre(varGlob1, 'c');
}
```

Directives pour le préprocesseur

Variables globales du module

Entête d'un sous-programme défini hors du module.

Sous-programme défini dans le module (et donc exporté)

Un des module doit contenir le programme principal **main()**

Chapitre 3 :

Objets de base du langage

3. Types de base du langage

Les mêmes qu'en JAVA

- `char, short, int, long, float, double`

Avec en plus des qualificateurs

- `unsigned / signed` :

- Exemple : `unsigned int a;` (`a` interprété comme un entier non signé)

Attention : pas de type natif booléen en C

- Par contre les expressions booléennes existent.
- Toute valeur entière peut-être considérée comme un booléen :
 - Une expression **fausse est codée par un entier nul**
 - Une expression **vraie est codée par un entier non nul**

- Les notation de constantes : idem Java
 - ▶ Le caractère '`\n`' permet de passer à la ligne
 - ▶ Le caractère '`\b`' permet de faire un « *backspace* »
 - ▶ Constantes de type chaîne "**ceci est une chaîne** `\n`"

- Les constantes nommées
 - ▶ On utilise la directive **#define** du préprocesseur
#define PI 3.1416
#define NOM "toto"

3. Les déclarations de variables

Idem JAVA

- CF. polycopié de langage C page 8

Exemples de déclarations simples :

```
int i,j;  
char c;  
float x,y;
```

Déclaration avec initialisation :

```
int j=3;  
float x=10.235;  
char c='Y';
```

Les commentaires (idem JAVA)

```
/* ceci est un commentaire */  
// ceci est un commentaire (sur une ligne)
```


- Attention : parenthèses souvent indispensables
 - ▶ Règles de priorités identiques à JAVA (cf. p. 12 & 13 du poly.)
- Le piège de l'affectation '='
 - ▶ Comme en Java, une affectation est une expression qui a pour valeur l'expression affectée à l'opérande gauche.
 - ▶ Exemple : **x=3** est interprété comme une expression qui affecte 3 à x, et rend la valeur 3.
 - ▶ Si on utilise dans un test **x=3** au lieu de **x==3**, il n'y aura pas d'erreur de type, car le type entier est utilisé comme booléen.
 - Quelque soit la valeur de **x**, parce que **x=3** rend trois, on considère que le prédicat **x=3** est toujours vrai.
- Les appels de fonction
 - ▶ Idem JAVA (cf. p. 12 du poly.)

3. Les conversions de type

- Fonctionnement proche de celui de JAVA
- Conversions implicites
 - ▶ Voir promotion de type dans le poly. [p11](#)
- Conversions explicites
 - ▶ Utilise le forceur (*cast* en anglais)
 - ▶ Syntaxe : **(nom_type) expression**
 - L'expression est évaluée et convertie dans le type indiqué.
 - Très utile dans le passage de paramètre dans les fonctions
 - ▶ Exemple :
int a = ...;
char b = (char) (a)

- ▣ Idem JAVA
- ▣ Toute expression terminée par ; devient une instruction
- ▣ Instruction composée-bloc (cf poly. p. 19)
- ▣ Autres instructions
 - pages 20 à 23 du poly.

3. Structures de contrôles

- Identiques à celles de JAVA
 - ▶ MAIS pas de type natif booléen, donc un prédicat est considéré comme vrai si son évaluation rend un résultat entier non nul.

■ Exemple :

```
int i=7;
while(i) {
    i--;
    ...
}

int i=7;
while(i!=0) {
    i--;
    ...
}
```

Les deux constructions
sont équivalentes, mais la
seconde est plus lisible

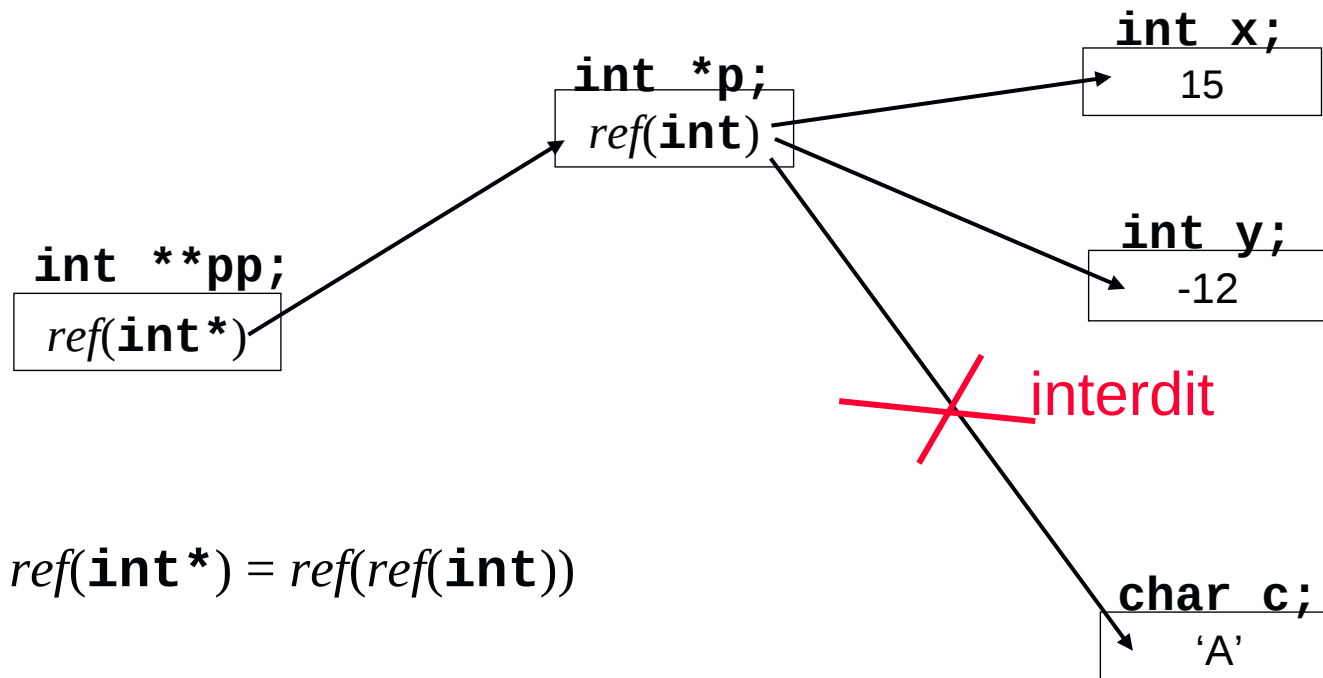
Chapitre 4

Pointeurs et tableaux

- En C on dispose d'un type pointeur
 - ▶ Un pointeur stocke **une référence** à un objet
 - ▶ On dit aussi qu'il contient l'**adresse** de l'objet qu'il référence
- Les pointeurs sont typés
 - ▶ Ils s'appliquent à un **type d'objet** (entier, caractère, pointeur, etc.)
 - ▶ Exception : le type de pointeur (**void***)
- Déclaration d'un pointeur
 - ▶ `int* p1;` déclare **p1** comme un pointeur sur un **int**.
 - ▶ `char* p2;` déclare **p2** comme un pointeur sur un **char**.
 - ▶ `int** pp;` déclare **pp** comme un pointeur de pointeur sur **int**.

4. Les pointeurs : exemple

- L'objet **p** est un pointeur sur des entiers (**int ***)
 - ▶ Il peut contenir une référence soit à **x**, soit à **y** (tous deux **int**), mais pas à **c** (qui est un **char**).
- L'objet **pp** est un pointeur de pointeur sur des entiers (**int ****)
 - ▶ Il ne peut contenir qu'une référence à **p**, car c'est ici le seul objet de type pointeur sur entier (**int ***).



NB : $\text{ref}(\text{int}^*) = \text{ref}(\text{ref}(\text{int}))$

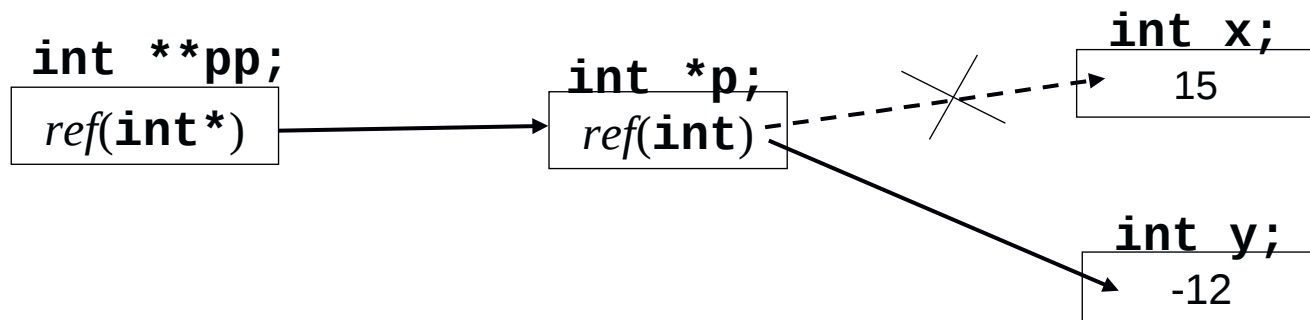
4. Les pointeurs : opérations

Opérations liées aux pointeurs :

- ▶ Le **déréférencement** (noté *****) permet d'accéder à l'information repérée par le pointeur.
- ▶ La **prise d'adresse** (notée **&**) permet de récupérer une référence sur un objet.

Exemples :

- ▶ L'expr. ***p** rend un objet **int** de valeur 15
- ▶ L'expr. ***pp** rend un objet **ref(int)** qui référence l'objet **x**
- ▶ L'expr. **&y** rend un objet **ref(int)** qui référence l'objet **y**



- ▶ L'instruction ***pp=&y;** va modifier le contenu de l'objet référencé par **pp** (c.a.d le pointeur **p**), en lui affectant une référence à l'objet **y**.

■ Ils servent :

- ▶ Dans le passage des paramètres à une fonction
- ▶ Ils font partie intégrante du mécanisme d'indexation
- ▶ Dans l'allocation dynamique
- ▶ Dans la construction des structures de liste
- ▶ etc.

■ On utilise souvent **NULL**

- ▶ **NULL** est une valeur de pointeur ne pointant sur rien, son principe est très proche de la valeur **null**.
- ▶ **Attention à la confusion majuscule/minuscule !**

🖥️ Soit les instructions ci-dessous

```
int x, y ;  
int *p;  
p = &x;  
x=2;  
y=4;  
*p = y;  
x = 5;  
y = *p;  
p = &y;
```

🖥️ Donnez la suite des valeurs prises par **x**, **y** et **p**

- En JAVA la taille d'un tableau est gérée **dynamiquement** lors de sa création par **new()**

```
int[] a ;  
...  
a = new int[10] ;  
...  
System.out.println("a[0] = " + a[10]) ;
```

- En C la taille d'un tableau est définie **statiquement** lors de la déclaration.

```
int a[10] ;  
...  
int main() {  
    a[0] = 12 ;  
    printf("a[0] =%d \n", a[0] ;  
    return 0;  
}
```

🚩 Est-ce que le programme C ci-dessous est correct ?

```
#include <stdio.h>
int taille = 15;
int table[taille];

int main() {
    int i=0;
    for (i=0;i<taille;i++) {
        printf("table[%d]=%d\n", i, table[i]);
    }
    return 0;
}
```

🚩 Est-ce que le programme C ci-dessous est correct ?

```
#include <stdio.h>
int taille = 15;      #define TAILLE 15
int table[taille];    int table[TAILLE] ;

int main() {
    int i=0;      TAILLE
    for (i=0;i<taille;i++) {
        printf("table[%d]=%d\n", i, table[i]);
    }
    return 0;
}
```

- On peut initialiser un tableau lors de sa création

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

```
int table[]={1, 3, -2, 19, -79};
```

```
int main() {  
    int i=0;  
    for (i=0;i<5;i++) {  
        printf("table[%d]=%d\n", i, table[i]);  
    }  
    return 0;  
}
```

- Dans une expression, un nom de tableau a pour valeur la référence à son premier élément:
 - Ex : l'objet **table** a la même valeur que l'objet **&table[0]**

- En C il y a « équivalence » entre tableaux et pointeurs
 - ▶ Les expressions `t[i]` et `*(t+i)` désignent le même objet
 - ▶ La notion de tableau est très proche de celle de pointeur
 - ▶ On peut faire de l'arithmétique sur les pointeurs.
- Alors quelle différence ?
 - ▶ Lorsque l'on **déclare** un tableau, on réserve en même temps de l'espace mémoire (on déclare un tableau en lui affectant une taille donnée)
 - ▶ Lorsque l'on **déclare** un pointeur, on ne réserve de la mémoire que pour stocker une référence (et pas les éléments d'un tableau)

- Les expressions `t[i]` et `*(t+i)` désignent le même objet

```
int a[10];
int* p=a;
...
int main(){
    int i=0;
    for (i=0; i<10; i++) {
        a[i]=i;
    }
    ...
    p=a; // idem p=&(a[0])
    for (i=0; i<10; i++){
        (*p)=i;
        p++;
    }
    ...
}
```


- On peut affecter à un pointeur un objet de type tableau
 - ▶ Ce dernier récupère une référence sur le 1^{er} élément du tableau.
- On **ne peut pas** affecter à un tableau une valeur de type pointeur
 - ▶ L'objet tableau doit toujours représenter la même zone mémoire.
- On **ne peut pas** affecter un tableau à un autre tableau
 - ▶ Si on souhaite copier le contenu d'un tableau dans un autre il faut le faire autrement (copie élément par élément par une boucle).

```
char * a = "toto";
char b [10];
char * p;
...
int main () {
    p=a;
    b=a    -> erreur à la compilation
    ...
}
```

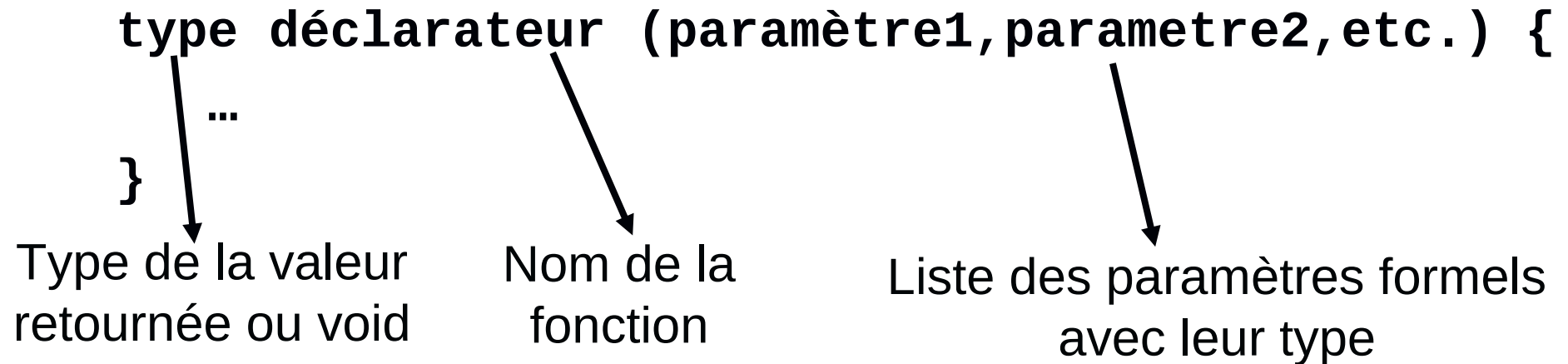
Chapitre 5

Fonctions et passage de paramètres

Pas de notion de **private/public** en C

- ▶ Il n'y a pas de classes, deux **fonctions** ne peuvent avoir le même identificateur.
- ▶ Les fonctions se déclarent au même niveau que le main
- ▶ Une fonction (comme les variables non locales) est globale, donc exportée automatiquement.

Déclaration de fonction



- Soit les fonctions :

```
void proc1() { ... }  
void proc2(int n, float x, char t[]){ ... }
```

- Avec les déclarations

```
float r, char t1[10];
```

- On peut avoir l'appel

```
proc2 (2, r, t1);
```

- Les objets visibles dans une fonction sont :
 - ▶ Ses paramètres formels.
 - ▶ Ses variables locales (celles déclarées dans son bloc).
 - ▶ Les variables globales au module.
 - ▶ Les variables globales importées (spécifiée **extern**).
 - Les autres fonctions déclarées **avant** dans le module ou externes
- Les règles classiques de durée de vie s'appliquent
- Le langage C ne tolère pas l'homonymie
 - ▶ Les mots clés du langage sont réservés.
 - ▶ Ex : on ne peut pas nommer une variable **for** ou **while**, etc.

- ▣ Idem qu'en JAVA
- ▣ Exprimée par l'exécution de l'instruction **return**.
 - Si aucune instruction return n'est exécutée avant la rencontre de l'accolade fermante de son bloc, la fonction délivre une valeur indéterminée (ceci est normal si le type rendu est **void**).
- ▣ Syntaxe de **return**
 - **return** *expression*
 - Les types possibles rendus par l'évaluation de *expression* sont :
 - ▣ Les types de base
 - ▣ Les pointeurs
 - ▣ Les structures
 - Attention : on ne peut pas retourner de tableaux !
 - ▣ On retourne un pointeur **char***, un **int***, ...

- En Java le mode de passage dépend du type de l'objet
 - ▶ Quand on passe un objet de type **class** on envoie une **référence** à un objet (adresse de cet objet)
 - ▶ Quand on passe un objet de type primitif (**char**, **int**, ...) on envoie un **valeur** (qui est une copie de l'objet)
- En C, le mode de passage se fait uniquement par **valeur**
 - ▶ Si on souhaite que le sous-programme modifie le contenu d'un objet, il faut passer en paramètre un pointeur sur cet objet.

Le paramètre formel est vu comme une variable locale initialisée avec la valeur du paramètre effectif.

5. Pas de paramètres à modifier : exemples

Exemples de fonction

```
int max (int a, int b) {  
    if(a>b) return a; else return b;  
}
```

```
void afficher_vecteur (float v[], int n) {  
    /*n composants*/  
    int i;  
    for (i=0;i<n;++i) printf ("%f ", v[i]);  
}
```

Exemples d'appel

```
int x, y;  
float t[10];  
x=5; y=2;  
printf("%d ", max(x,y));  
afficher_vecteur(t,10);
```


- ▣ Lorsque le paramètre effectif est un tableau **t [...]**.
 - Le paramètre formel reçoit l'adresse du 1^{er} élément (&t[0]).
 - Puisque le paramètre formel est initialisé avec l'adresse du paramètre effectif, on peut modifier les éléments du tableau

▣ Exemple :

- Fonction de mise à zéro des **n** valeurs d'un vecteur **v**

```
void raz_vecteur (float v [], int n) {  
    int i;  
    for (i=0;i<n;i++)  
        v[i]=0;  
}
```

5. Modification d'un objet simple

- On passe un pointeur sur cet objet à la fonction
 - Le paramètre formel est donc un pointeur

Exemple :

```
/* affecte v à l'objet pointé par x */
```

```
void affecter(int *x, int v){
```

```
    *x = v
```

```
}
```

```
/* permute x et y */
```

```
void permuter(int *x, int *y){
```

```
    int aux;
```

```
    aux=*x ;
```

```
    *x=*y;
```

```
    *y = aux;
```

```
}
```

```
int a;
```

```
affecter ( &a, 3);
```

```
int b;  b=10
```

```
permuter ( &a, &b);
```

affecter(&a, 3)

```
int *x;
```

```
ref(int)
```

```
int v;
```

```
3
```

permuter(&a, &b)

```
int *x;
```

```
ref(int)
```

```
int *y;
```

```
ref(int)
```

Var. globales

```
int a;
```

```
3
```

```
int b;
```

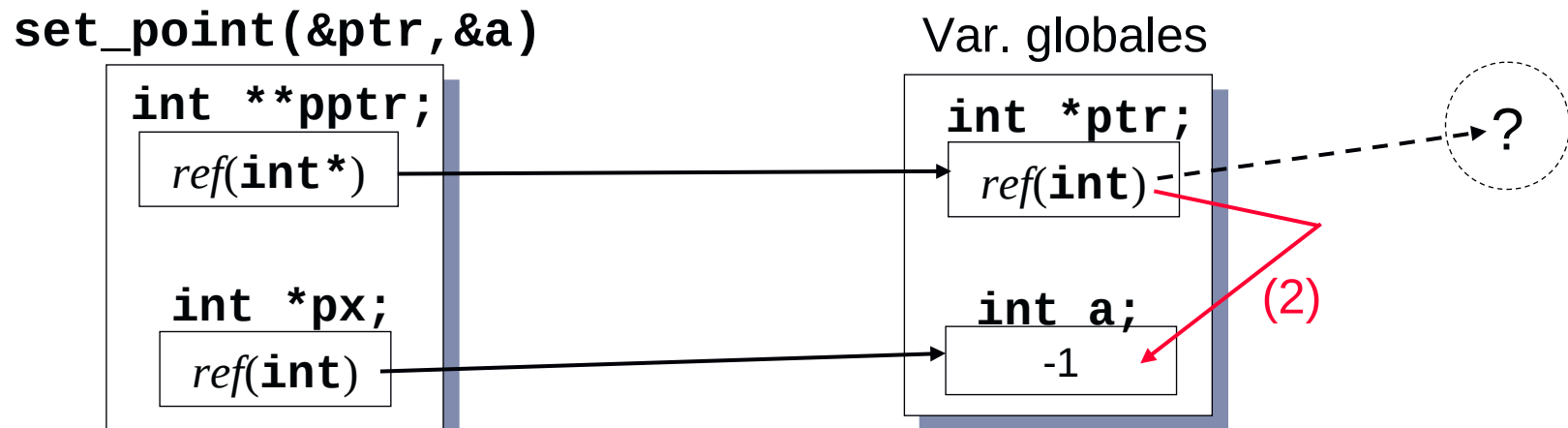
```
10
```

5. Modification d'une var. pointeur

- Affecter un pointeur à partir d'un autre pointeur
 - Il faut passer un pointeur sur l'objet à modifier, dans notre cas on doit donc utiliser une paramètre de type **pointeur de pointeur**.

Exemple :

```
void set_point(int **pptr, int *px) {  
    *pptr = px; (2)  
}  
  
int *ptr; int a=-1; (1)  
set_point(&ptr, &a); (3)
```



```
#include <stdio.h>
void exchange(int * a,int b){
    int sauve; sauve = *a; *a = b; b = sauve;
}
void exchange2(int * a,int ** b){
    int sauve; sauve = *a; *a = **b; **b = sauve;
}
void exchange3(int * a,int * b){
    int sauve; sauve = *a; *a = *b; *b = sauve;
}
int main(){
    int a,b,d; int* c=&d; a = 1; b = 2; (*c) = 3;
    exchange(&a,b);      printf("a= %d, b= %d\n",a,b);
    exchange2(&a,&c);     printf("a= %d, c= %d\n",a,*c);
    exchange3(&a,&b);     printf("a= %d, b= %d\n",a,b);
    exchange3(&b,c);     printf("b= %d, c= %d\n",b,*c);
    exchange(&a,&b);     printf("a= %d, b= %d\n",a,b);
    return 0;
}
```

Qu'affiche cette fonction ?

- Une fonction peut-être déclarée au même niveau que le **main()**
 - ▶ On peut **vouloir** appeler une fonction avant sa déclaration, d'où un problème pour le compilateur (pour le choix du type)
 - ▶ Solution : déclaration de prototype
- Un prototype de fonction = déclaration de fonction
 - ▶ Le bloc y est remplacé par un point virgule.
 - ▶ Le nom des paramètres peut être omis
- Exemples:
 - ▶ **void f1(int, char);**
 - ▶ **int maxi(float[], float *);**

- Un exécutable peut être appelé avec des arguments
 - ▶ L'appel **prog arg1 arg2 ... argn** exécute le programme **prog** en lui passant en argument les chaînes **arg1 arg2 ... argn**
- Ces arguments sont comme des paramètres de **main()**
 - ▶ Le prototype s'écrit alors **int main(int argc, char* argv[])**
 - ▶ Le paramètre **argc** contient la taille de **argv** (c-à-d. le nombre d'arguments + 1)
 - ▶ Le paramètre **argv** est un tableau de chaînes de caractères
 - **argv[0]** est un **char*** pointant sur le nom de la commande (idem **\$0** en sh)
 - **argv[1]** est un **char*** pointant sur le 1^{er} argument (idem **\$1** en sh)
 - **argv[argc-1]** est un **char*** pointant sur le dernier argument.
- Remarque :
 - ▶ Si on souhaite récupérer des valeurs entières en argument, il faut effectuer une conversion de **char*** -> **int** à l'aide de la fonction de conversion **int atoi(char* ch)**

5. Exemple d'arg. passés à la commande

Soit le programme **prog** dont le source C est le suivant

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int i;
    for (i=1;i<argc;i++) {
        printf("l'argument n°%d vaut %s ",i,argv[i]);
    }
}
```

L'exécution de la commande **prog toto 12 titi**
affiche

L'argument n°1 vaut toto
L'argument n°2 vaut 12
L'argument n°3 vaut titi

Chapitre 6

Manipulation des chaînes de caractères & Entrées/Sorties formatées

- En Java on dispose de la classe **String**
- En C, une chaîne = tableau de **char** terminé par '`\0`'
 - ▶ C'est à l'utilisateur de s'assurer que le contenu de la chaîne ne déborde pas de l'espace alloué au tableau.
 - ▶ Beaucoup plus fastidieux à manipuler qu'en JAVA
 - Source de beaucoup d'erreurs d'exécutions
- Il existe des fonctions de manipulation de chaînes
 - ▶ Pour les utiliser il faut ajouter la directive `#include<string.h>` dans la partie entête du module.
 - ▶ Longueur d'une chaîne : **strlen(...)**
 - ▶ Comparaison de chaînes : **strcmp(...)**
 - ▶ Copie de chaînes : **strcpy(...)**
 - ▶ Formatage de chaînes : **sprintf(...)**

Synopsis :

- ▶ `#include <string.h>`
- ▶ `int strlen(char *s);`

Valeur retournée :

- ▶ La fonction `strlen()` retourne la longueur de la chaîne de caractère (terminée par le marqueur `'\0'` qui n'est pas compté comme un caractère de la chaîne).

Erreurs

- ▶ Pas d'erreurs définies.

Exemple

```
char ch[]="toto";  
printf("%d\n",strlen(ch));
```

Synopsis :

- ▶ `#include <string.h>`
- ▶ `char *strcpy(char *s1, char *s2);`

Description

- ▶ La fonction **`strcpy()`** copie le contenu de la chaîne désignée par `s2` (le caractère `'\0'` inclus) dans le tableau désigné par `s1`. Les chaînes désignées par `s1` et `s2` ne doivent pas se recouvrir.

Valeur retournée :

- ▶ La fonction **`strcpy()`** retourne normalement `s1`; il n'y a pas de valeur de retour pour signaler une erreur d'exécution.

Erreurs

- ▶ Pas d'erreurs définies.

6. Exemple pour strcpy()

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

int main(){
    char text1[10] = "Hyppolite";
    char text2[10] = "Hector";

    printf("Le contenu initial de text1 est :%s \n", text1);
    /* Copie de text2 dans text1, si la taille de text1 est
    insuffisante, on risque d'écraser des données à la
    suite de text2 ou un « Incident de segmentation » ! */
    strcpy(text1,text2) ;
    printf("text1 contient maintenant :%s \n", text1);

    strcpy(text1,"Hyppolite, Barnabé & Cie") ;
    printf("text2 contient maintenant :%s \n", text2);
}
```

Attention, `strcpy()` ne vérifie pas les tailles des chaînes avant de faire la copie !

Synopsis

- ▶ `#include <string.h>`
- ▶ `int strcmp(char *s1, char *s2);`

Description

- ▶ La fonction `strcmp()` compare deux chaînes de caractères `s1` et `s2` toutes deux terminées par le caractère `'\0'`..

Valeur de retour

- ▶ Une fois terminée, `strcmp()` retourne un entier de valeur respectivement supérieure, égale ou inférieure à 0, dans le cas la chaîne désignée par `s1` (resp.) succède à, est égale à, ou précède la chaîne `s2`..

Erreurs

- ▶ Pas d'erreurs définies.

6. Exemple pour strcmp ()

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

char nom1[100] ;
char nom2[100] ;
int res ;

int main(){
    printf("Entrez un mot :");
    scanf("%s", nom1);
    printf("Entrez un mot :");
    scanf("%s", nom2);

    res = strcmp(nom1,nom2) ;
    if (res == 0) {
        printf("%s et %s sont identiques\n", nom1, nom2);
    } else if (res<0) {
        printf("%s précède %s \n", nom1, nom2);
    } else { // res>0
        printf("%s précède %s \n", nom2, nom1);
    }
}
```

- Tout module faisant appel aux fonctions d'E/S doit contenir la directive **#include<stdio.h>**
- La gestion des E/S en C est différente de celle de Java
 - ▶ En Java, un objet « sait » s'afficher si sa classe dispose d'une méthode **toString()**
 - ▶ En C, on doit fournir une chaîne qui indique le **type des objets** sur lesquels on souhaite faire une E/S.
- On va se limiter pour l'instant à des E/S simples sur
 - ▶ L'*entrée standard* (désignée par l'identificateur **stdin**)
 - ▶ La *sortie standard* (désignée par l'identificateur **stdout**)

- Lecture d'un caractère : **getchar()**
 - ▶ C'est une fonction de type int qui délivre le caractère suivant lu sur l'entrée standard
- Écriture d'un caractère : **putchar(c)**
 - ▶ S'emploie comme une procédure pour ajouter le caractère c sur la sortie standard
- Exemple

```
char c;  
c = getchar();  
putchar( '\n' ); /*affichage retour de ligne*/
```


On dispose pour cela de deux fonctions :

- ▶ `int printf(format, e1, e2, ..., en);`
- ▶ `int scanf(format, a1, a2, ..., an);`

Paramètres

- ▶ **format** est une chaîne décrivant le format de lecture/écriture
- ▶ Les expressions à formater sont notées **e_i**
- ▶ Les adresses des variables à initialiser sont notées **a_i**

Fonctionnement

- ▶ Les lectures (ou écritures) sont réalisées en fonction du format indiqué dans la chaîne **format**.
- ▶ A chaque paramètre correspond dans **format** une spécification de format commençant par le caractère %

- Syntaxe d'une spécification de format:
 - ▶ **%d** : indique que l'E/S concerne un entier signé (**int**)
 - ▶ **%c** : indique que l'E/S concerne un caractère ASCII (**char**)
 - ▶ **%s** : indique que l'E/S concerne une chaîne terminée par '**\0**'
 - ▶ **%lf** : indique que l'E/S concerne un double (**double**).
 - ▶ **%x** : indique que l'E/S concerne un entier non signé à afficher en sous sa représentation hexadécimal.
 - ▶ Pour plus d'info **man format** ou *google printf*
- Remarques :
 - ▶ Une chaîne de format peut également contenir du texte normal
 - ▶ On peut exprimer plusieurs E/S dans une chaîne de format
 - ▶ Le mieux : tester sur des exemples

Synopsis

- ▶ `int printf (format, e1, e2, ..., en);`

Description

- ▶ Effectue une écriture formatée des paramètres e_1, \dots, e_n sur la sortie standard , à partir du paramètre **format**.
- ▶ C'est une fonction qui accepte un **nombre variable** de paramètres
- ▶ Faites attention à ce que le nombre d'arguments corresponde au nombre d'E/S spécifiés dans le format !

Résultat

- ▶ Nombre de caractères affichés (valeur < 0 en cas d'erreur)

Erreurs

- ▶ Pas de code d'erreur

6. Exemple : printf(...)

🚧 Qu'affiche ce programme ?

```
#include<stdio.h>

int main(){
    char nom[20] = "Balthazar";
    double solde = 1000.25;
    int age = 21;

    printf("Bonjour %s, vous avez %d ans\n", nom,age);
    printf("Votre age s'écrit %x en hexadécimal\n", age);
    printf("Votre nom commence par la lettre %c\n", nom[0]);
    printf("Votre solde bancaire est de %lf\n", solde);
    printf("L'adresse du tableau nom est %x\n", nom);
}
```

🚧 Réponse :

Synopsis

- ▶ `int scanf(format, a1, a2, ..., an);`

Description

- ▶ Lit sur l'entrée standard des valeurs selon le format spécifié et les affecte aux variables dont les adresses sont **a₁, a₂, ..., a_n**
- ▶ Faites attention à ce que le nombre d'arguments corresponde au nombre d'E/S spécifiés dans le format.

Résultat

- ▶ Renvoie le nombre d'éléments entrés correctement

Erreurs

- ▶ Pas de code d'erreur

Rappel: notation de l'adresse d'une variable: **&ident**

6. Exemple : scanf (...)

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

```
int main(){
```

```
    int i;
```

```
    float f;
```

```
    char chaine[100];
```

```
    while (1) {
```

```
        printf("Entrez un entier :");
```

```
        scanf("%d",&i);
```

```
        printf("Entrez un nombre réel :");
```

```
        scanf("%f",&f);
```

```
        printf("Entrez une chaîne :");
```

```
        scanf("%s",chaine);
```

```
        printf("Vous avez saisi %d, %f, et %s \n", i, f, chaine);
```

```
    }
```

```
}
```

On passe l'adresse de **i** pour permettre sa modification par la fonction **scanf()**.

On passe directement l'identificateur de la variable **chaine**, car il s'agit ici d'une **adresse** (idem **&chaine[0]**)

- ▣ Que se passe-t-il si je me trompe de format ?
 - Ça dépend (en général ça plante ...)
- ▣ Question : qu'affiche le programme suivant :

```
#include <stdio.h>

void main() {
    int a = 1654 ;
    printf("%s\n", a);
}
```

- ▣ Réponse : **Segmentation fault (core dumped)**
 - Essaie d'afficher la chaîne de caractère stockée à partir de l'adresse 1654, et dont la fin est indiquée par le caractère '\0'.
 - ▣ Le programme va finir par rencontrer une adresse « illégale »
 - C'est une erreur d'exécution que vous allez souvent rencontrer !

Chapitre 7

Types énumérés et structures

- Le type **void** (type indéfini), utilisé
 - ▶ Pour des fonctions qui ne rendent pas de valeur
 - ▶ Pour les pointeurs d'un type non déterminé
- Le type booléen (déjà vu)
 - ▶ Il n'existe pas en C.
 - ▶ Par convention toute valeur entière peut être considérée comme une valeur booléenne:
 - Le **false** de Java équivaut en C à un entier nul
 - Le **true** de Java équivaut en C à un entier différent de 0

7. Type énuméré

- On peut définir un nouveau type (d'entiers) par :

```
enum nomdtype { liste_valeurs }
```

- Exemple :

```
enum jour {  
    lundi, mardi, mercredi,  
    jeudi, vendredi, samedi,  
    dimanche  
};  
/* lundi vaut 0, mardi vaut 1 ....*/
```

- On a créé un nouveau type de nom **enum jour**.

- toute variable de type **enum jour** pourra valoir soit **lundi**, soit **mardi**, etc...

- exemple : **enum jour hier, demain;**

- Crée deux variables de type **enum jour**;

7. Définition de type : typedef

- En utilisant **typedef**, on peut définir un nouveau type
 - Plus besoin de répéter **enum** dans la déclaration de variables :

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

```
typedef enum {  
    faux, vrai // attention à l'ordre !!!  
} BOOLEEN;
```

- On peut alors directement déclarer:

```
JOUR hier, demain;  
BOOLEEN salarié;
```

- En C on définit une structure à l'aide du mot clé **struct**

```
struct date {  
    int j,m;  
    double a;  
};
```

- Si on souhaite déclarer un variable de ce type :

```
struct date une_date;
```

- Pour pouvoir l'utiliser comme un type :

```
typedef struct {  
    int j,m;  
    double a;  
} date;
```

- Les déclarations de variables, se font alors par :

```
date hier, demain;
```

- Possibilité d'imbriquer des structures les unes dans les autres.

- Exemple:

```
typedef struct {  
    date naissance;  
    boolean etudiant;  
} renseignement;
```

- Déclaration avec initialisation

```
renseignement toto= {  
    { 13, 10, 1960}, vrai  
};
```

- Pointeur sur une structure

```
date* queljour;
```

- Pour un objet de type **struct**, l'accès se fait par la notation pointée **'.'**

```
renseignement toto= {{13,10,1960},vrai};  
toto.etudiant = vrai;  
toto.naissance.j = 13;
```

- Pour un objet de type pointeur sur **struct**, l'accès se fait par la notation **'->'**

```
date* queljour;  
queljour->j = toto.naissance.j;
```

- Remarque :

- La notation **queljour->j** est équivalente à **(*queljour).j**

- Une structure peut faire référence à elle-même dans sa définition.
 - On parle alors de définition récursive.

■ Exemple de la liste chaînée

```
typedef struct list_elem{  
    int data;  
    struct list_elem *next;  
} elem;
```

(...)

```
elem *liste1;  
liste1->data=4;  
liste1->next=NULL;
```

- On peut obtenir la taille d'un type à l'aide de `sizeof()` qui délivre la taille en octet occupée
 - Soit par une variable
 - Soit par une expression
 - Soit par les objets d'un type

Exemples

```
printf("taille du type date %zu", sizeof (date));  
printf("taille de la var. toto %zu", sizeof  
    (toto));
```


7. Opérateur sizeof : exemple

- Le programme C ci-dessous

```
#include <stdio.h>
int taille = 15;
int table[15];
int* p;
int main() {
    p= table;
    printf("%d\n", sizeof(table));
    printf("%i\n", sizeof(p)); return 1;
}
```

- Affiche sur la S.S. les valeurs 60 (15x4), puis 4
 - Attention, cela dépend des machines ! Par exemple sur les vieux processeurs *Intel*, les **short int** sont codés sur 8 bits.

Chapitre 8 :

Allocation dynamique

En Java :

- ▶ On fait de l'allocation dynamique lorsque l'on crée un nouvel objet en utilisant l'opérateur **new**.
- ▶ Le ramasse miette se charge de libérer l'espace mémoire lorsque l'objet n'est plus utilisé.

En C :

- ▶ L'allocation dynamique se fait par un appel à la fonction système **malloc()** :

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

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

- ▶ C'est au programmeur de se charger de la libération de la mémoire en utilisant la fonction système **free(...)**.

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

```
void free(void *pointer);
```

Synopsis

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

Description

- ▶ La fonction malloc() alloue *size* octets, et renvoie l'adresse du premier octet alloué .
- ▶ Le contenu de la zone de mémoire n'est pas initialisé.

8. La fonction `malloc()`

- ✚ Pour calculer la mémoire nécessaire pour stocker un type d'objet, on utilise la fonction `int sizeof(type)`.
 - Par ex : `sizeof(int)` sur un x86 retourne 4, car les entiers sont codés sur 32 bits (à partir du i386)
- ✚ Il faut faire une conversion de type car `malloc()` retourne un type `void*` alors qu'on veut souvent un `int*` ou `char*`

Synopsis

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

Description

- ▶ **`free()`** libère l'espace mémoire pointé par *ptr*, qui a été obtenu lors d'un appel à **`malloc()`**, **`calloc()`** ou **`realloc()`**.
- ▶ Si le pointeur **`ptr`** n'a pas été obtenu par l'un de ces appels, ou si il a déjà été libéré avec **`free()`**, le comportement est indéterminé.
- ▶ Si *ptr* est **`NULL`**, aucune tentative de libération n'a lieu.

8. Allocation dynamique dans le TAS

Exemple :

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

int main(){
    int* p; int i;
    int taille = 100;

    p = (int*)(malloc (taille * sizeof(int)));
    if (p == NULL) {
        printf("Allocation impossible:");
        exit(EXIT_FAILURE);
    }

    for (i=0; i<taille;i++)
        p[i]=0;

    free(p);
}
```

Chapitre 9 :

Entrées/Sorties sur les fichiers




- En C les entrées/sorties se font sur des fichiers
- Il existe plusieurs types de fichiers prédéfinis :
 - L'entrée standard notée **stdin** (flux de lecture)
 - La sortie standard notée **stdout** (flux d'écriture)
 - La sortie d'erreur notée **stderr** (flux d'écriture)
- Pour l'instant on a vu des E/S formatées simples
 - **printf(...)**: écriture sur le fichier de S.S.
 - **scanf(...)** : lecture dans le fichier d' E.S.
- On va étudier des E/S plus générales
 - E/S non formatées sur de **vrais fichiers**

- On accède aux fichiers au travers de **fichiers logiques**.
 - ▶ Pour les utiliser, il faut ajouter la directive **#include<stdio.h>**
 - ▶ Défini comme un type **FILE**
 - ▶ On utilise plutôt des pointeurs **FILE***
- Exemple
 - ▶ **FILE* file;** *// décl. d'un fichier logique*
- Remarque :
 - ▶ Pour être utilisable, un fichier logique d'un programme doit être lié à un fichier physique du système de gestion fichier.
 - ▶ Lien effectué au travers d'opérations d'ouverture et de fermeture de fichiers logiques

Synopsys

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

Description

- ▶ Ouverture du fichier physique de nom externe **name** dans le mode défini par la chaîne **mode** :
 -  **"r"** ouverture d'un fichier en mode lecture.
 -  **"w"** ouverture d'un fichier en mode écriture/création.
 -  **"a"** ouverture d'un fichier en mode écriture à la fin.

Paramètres

- ▶ **name** : nom externe du fichier à ouvrir
- ▶ **mode** : mode d'ouverture.

Valeur de retour

- ▶ Un pointeur sur le fichier logique est retourné. Si erreur, on retourne la valeur de pointeur **NULL**.

Synopsys

```
int fclose(FILE *file);
```

Description

- Fermeture du fichier logique passé en paramètre

Paramètres

- **file** : fichier logique à fermer

Valeur de retour

- La fonction rend la valeur zéro si le fichier a été fermé,
- Elle rend la constante **EOF** si il y a eu une erreur.

ATTENTION : un fichier ouvert en écriture (en construction) doit obligatoirement être fermé (sinon son contenu n'est pas sauvé)

- `int putc (int c, FILE * file);`
 - Écriture de la valeur **c** (convertie en **char**) dans le fichier **file**, retourne la valeur constante **EOF** en cas d'erreur.
- `int fwrite(void *buf, int size, int nb, FILE *file);`
 - Écriture des paquets de données situés dans la zone tampon repérée par **buf** dans le fichier **file** , le paramètre **size** donne la taille de chaque paquet, **nb** indique le nombre de paquets à écrire.
 - Retourne le nombre de paquets réellement écrits
- `int fprintf(FILE *file, char *format, liste_args);`
 - Fonctionne comme **printf()**, mais effectue l'écriture sur le fichier **file** plutôt que sur la sortie standard.

📄 **int getc (FILE * file);**

- ▶ Lecture d'un caractère dans le fichier **file**, retourne la valeur (convertie en **int**) du caractère, ou la constante **EOF** en cas d'erreur.

📄 **int fread(void *buf, int size, int nb, FILE *file);**

- ▶ Lecture des paquets de données à partir du fichier **file** dans le tampon mémoire repéré par **buf**, le paramètre **size** donne la taille de chaque paquet, nb indique le nombre de paquets à écrire.
- ▶ Retourne le nombre de paquets réellement lus, 0 si fin de fichier atteinte
- ▶ **Attention** : le tampon **buf** doit être assez grand pour contenir tous les paquets lus

📄 **int fscanf(FILE *file, char *format, liste_args);**

- ▶ Fonctionne comme **scanf()**, mais effectue la lecture sur le fichier **file** plutôt que sur l'entrée standard.

9. Exemple 2

```
#include <stdio.h>
#include <string.h>    /* strlen() */
#include <stdlib.h>    /* exit() */

main() {
    FILE *fichier;
    char *msg = "123456789012345";
    int n;
    fichier = fopen("/tmp/toto", "w")
    if (fichier!= NULL) {
        n = strlen(msg);
        int res = fwrite(msg, sizeof(char), n, fichier);
        if (res!=n) {
            /* écriture dans le flux stderr */
            fprintf(stderr,"Erreur a l'écriture\n");
            exit(2);
        }
        fclose(fichier);
    }
}
```



Écrire un programme C qui :

1. Lit un nom de fichier (nom1) au clavier
2. Lit un nom de fichier (nom2) au clavier
3. Copie (caractère par caractère) le contenu du fichier de nom1 dans le fichier nom2, en remplaçant tous les 'X' par des 'Y'.
4. Si la copie est impossible, affiche un message d'erreur

Chapitre 10 : Structuration des programmes

- On découpe le programme en modules
 - ▶ Chaque module utilise des données et fonctions **définies** localement ou dans un autre module.
 - ▶ Chaque module contient des données et fonctions **utilisées** localement ou dans un autre module.

- Pour faciliter la structuration du programme
 - ▶ On va écrire un fichier d'entête (**module.h**) par module
 - ▶ Ce fichier d'entête contient les types et prototypes des données et fonctions exportées par le module.
 - ▶ Quand un module doit utiliser une donnée/fonction d'un autre module il inclut le fichier entête (directive **#include<...>**)

Syntaxe :

`#include <nom_fichier>` (1)

`#include "nom_fichier"` (2)

Description

1. Incorpore un fichier de nom **nom_fichier** se trouvant dans le répertoire **/usr/include**
2. Recherche le fichier désigné dans le répertoire courant (si le fichier est dans un autre répertoire, préciser son chemin absolu)

Remarques :

- ▶ Permet d'incorporer, avant compilation, le texte figurant dans un fichier, ce texte est traité comme s'il se trouvait dans le fichier courant.
- ▶ Ces fichiers contiennent en général des déclarations, et ont pour extension `.h` (header), on les désigne sous le terme de fichier entête.

- Les déclarations de type ont comme portée le module:
 - ▶ Elles ne sont pas exportables (de même que les constantes).
- Si on souhaite les exporter :
 - ▶ On crée un fichier d'entête comportant ces déclarations
 - ▶ On l'inclus dans le module qui en a besoin par une directive **#include** au pré-compilateur
- NB : Les déclarations de variables globales et de fonctions sont automatiquement exportés

Chapitre 11 :

Chaîne de développement

Compileur gcc et outil make

- GCC est un compilateur C libre (GNU)
 - ▶ Conçu pour pouvoir être reciblé d'une machine vers une autre
 - ▶ Il est disponible par défaut sur les machines Linux
 - ▶ Il existe des versions Windows (cygwin et mingw32)
- Synopsys
 - ▶ **gcc [-c] [-g] [-I dir] [-o nom] f1 ... fn**
- Options
 - ▶ **-c** : indique que l'on souhaite produire un fichier objet **.o**
 - ▶ **-I dir** : ajoute dir à la liste des répertoires contenant des **.h**
 - ▶ **-o nom** : spécifie le nom de fichier de sortie (**a.out** par défaut)
 - ▶ **-g** : ajoute des informations pour le déboguage

- 📄 Produire un fichier objet à partir d'un module C
`gcc -o module.o -c module.c`
- 📄 Produire un exécutable **prog** à partir de module(s) objet(s)
`gcc -o prog module1.o ... modulen.o`

Principe

- ▶ Utilisé pour gérer des fichiers dépendants les uns des autres
- ▶ Permet d'automatiser la mise à jour des fichiers
- ▶ Utilisé pour automatiser la compilation des programmes
 - mais peut être utilisé pour autre chose

Fonctionnement

- ▶ On définit des dépendances entre des **cibles** (target), et des fichiers dont dépend chaque cible
- ▶ On donne des commandes à exécuter pour créer la cible à partir de ces fichiers.

Utilisation

- ▶ On regroupe les règles dans un fichier dont le nom est **makefile** (ou **Makefile**)
- ▶ Les règles ont le format suivant

```
cible : fichier1 fichier2 ...  
(TAB) unecommande
```

Signification d'une règle :

Si la date de dernière modification d'un des fichiers **fichier1**, **fichiers2**, etc. est plus récente que le fichier **cible**, alors l'outil **make** exécute la commande **unecommande**.

Appel de make

- ▶ L'appel se fait par la commande **make**, le programme essaie alors de réaliser la première cible du fichier **makefile** du répertoire de travail.

11. L'outil make : exemple

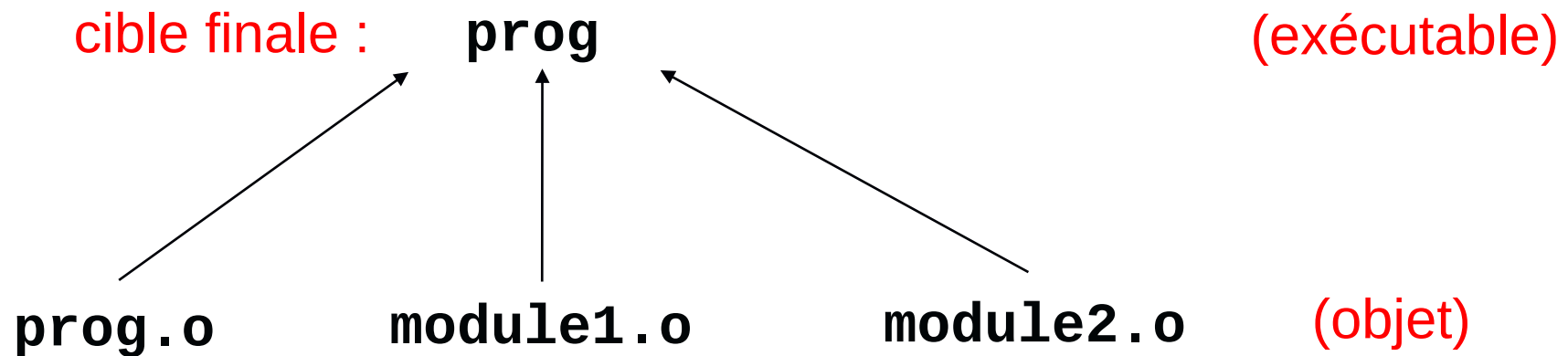
- Un programme C composé de 3 modules
 - `prog.c`, `module1.c`, `module2.c`
- Graphe de dépendance des fichiers

cible finale : **prog**

(exécutable)

11. L'outil make : exemple

- Un programme C composé de 3 modules
 - `prog.c`, `module1.c`, `module2.c`
- Graphe de dépendance des fichiers



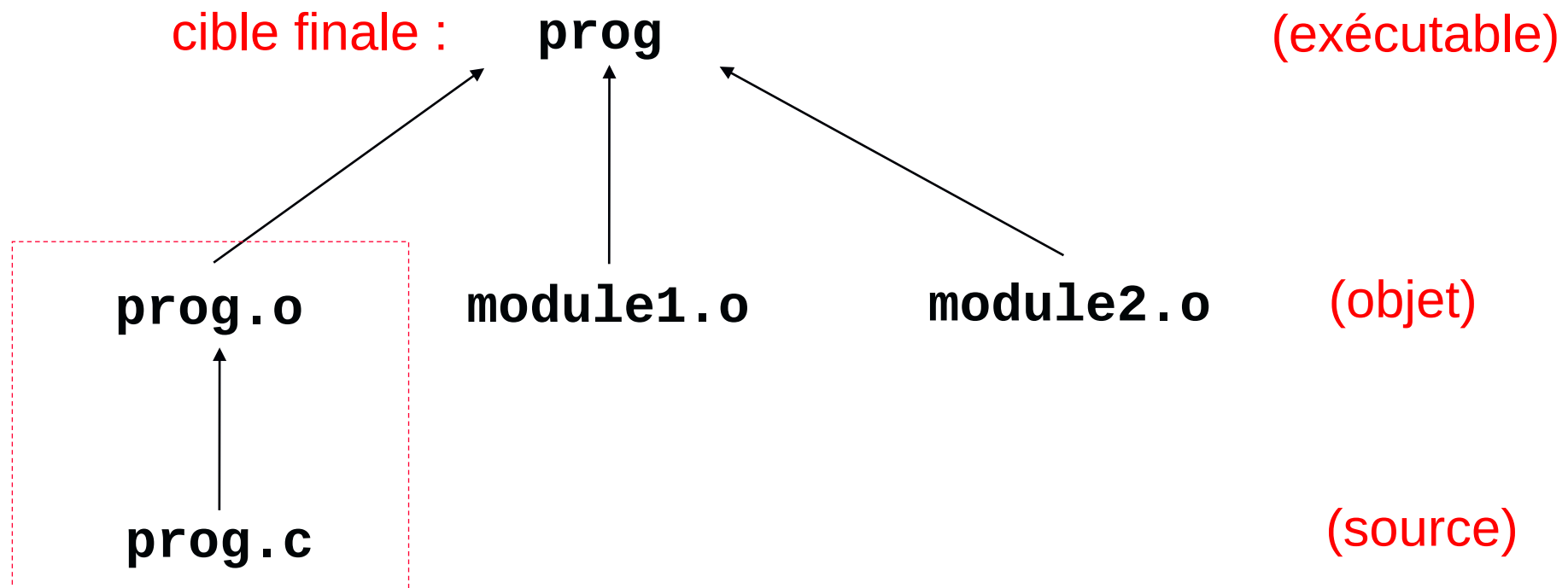
Pour produire le fichier **prog**, on doit faire la commande :

```
gcc -o prog prog.o module1.o module2.o
```

Le fichier **prog** **dépend** donc des fichiers objets **prog.o**, **module1.o** et **module2.o**

11. L'outil make : exemple

- Un programme C composé de 3 modules
 - `prog.c`, `module1.c`, `module2.c`
- Graphe de dépendance des fichiers

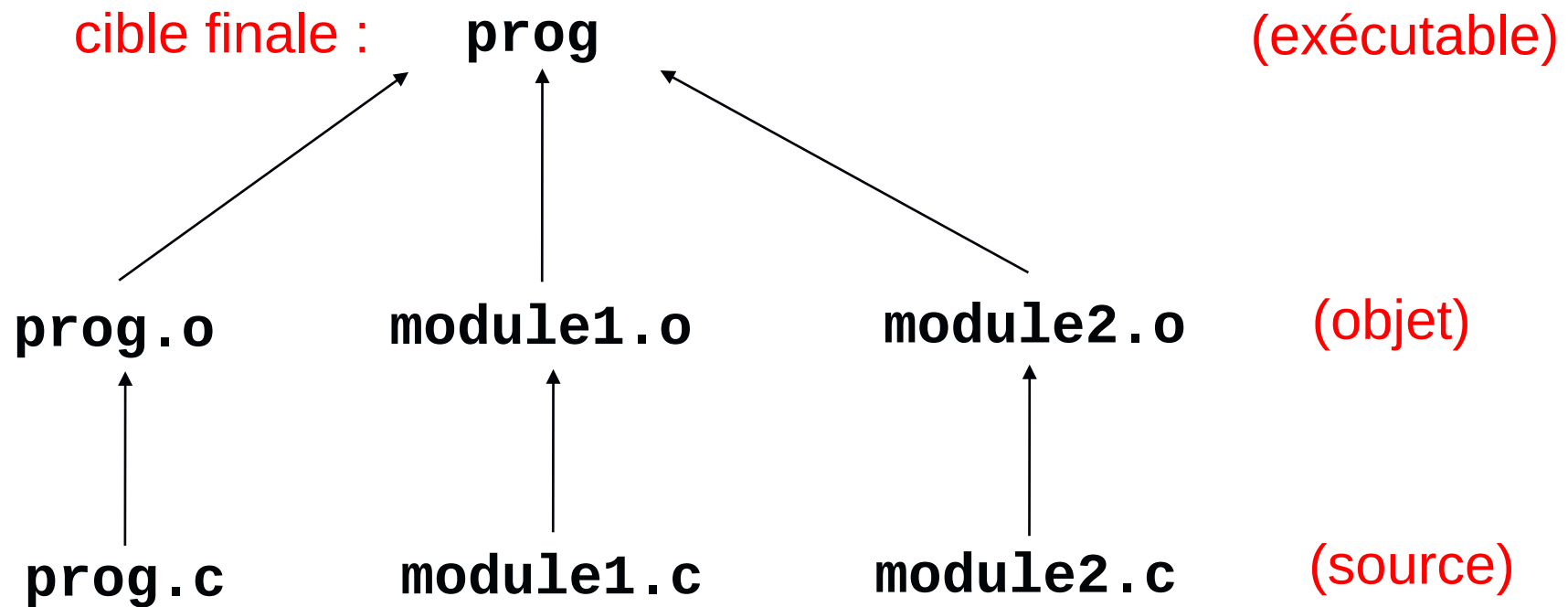


Pour produire **prog.o**, on doit faire `gcc -c prog.c`

Le fichier **prog.o** **dépend** donc du fichier source **prog.c**

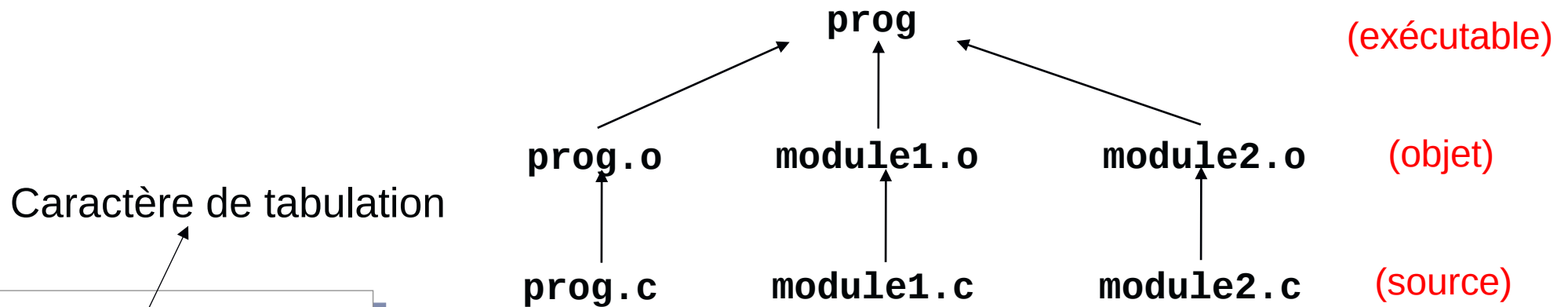
11. L'outil make : exemple

- Un programme C composé de 3 modules
 - `prog.c`, `module1.c`, `module2.c`
- Graphe de dépendance des fichiers



Du graphe de dépendance, on va déduire les définitions (cibles, dépendances et commandes) du **makefile**.

11. L'outil make : exemple



Caractère de tabulation

makefile

```
prog : prog.o module1.o module2.o
    gcc -o prog prog.o module1.o module2.o

prog.o : prog.c
    gcc -c prog.c

module1.o : module1.c
    gcc -c module1.c

module2.o : module2.c
    gcc -c module2.c
```

11. L'outil make : exemple

On a modifié **prog.c**, que se passe-t-il si on lance la commande **make** ?

La date de modification de **prog.c** est plus récente que celle de **prog.o**, l'outil make exécute donc la commande **gcc -c prog.c**

makefile

```
prog : prog.o module1.o module2.o
    gcc -o prog prog.o module1.o module2.o

prog.o : prog.c
    gcc -c prog.c

module1.o : module1.c
    gcc -c module1.c

module2.o : module2.c
    gcc -c module2.c
```

11. L'outil make : exemple

On a modifié **prog.c**, que se passe-t-il si on lance la commande **make** ?

La date de modification de **prog.o** est plus récente que celle de **prog**, l'outil make exécute donc la commande **gcc -o prog prog.o ...**

makefile

```
prog : prog.o module1.o module2.o
    gcc -o prog prog.o module1.o module2.o

prog.o : prog.c
    gcc -c prog.c

module1.o : module1.c
    gcc -c module1.c

module2.o : module2.c
    gcc -c module2.c
```


11. L'outil make : ajout d'une cible `clean`

Il est également possible d'ajouter des commandes au makefile, par exemple, lorsque l'on exécute

make clean,

le programme **make** va exécuter la commande

rm *.o

makefile

```
prog : prog.o module1.o module2.o
      gcc -o prog prog.o module1.o module2.o
```

...

```
module2.o : module2.c
          gcc -c module2
```

```
clean :
      rm *.o
```

Poubelle

- Intérêt du pointeur par rapport au tableau?
 - ▶ On utilise un objet pointeur quand on ne connaît pas la taille du tableau que l'on doit manipuler.
 - ▶ Par exemple, lorsque l'on souhaite passer un tableau en paramètre d'une fonction.

■ Exemple :

```
void razTableau (int* tab, int taille){  
    int i=0;  
    for (i=0; i<taille; i++)  
        tab[i]=0;  
}  
  
int t1[15];  
int t2[36];  
  
int main(int argc, char* argv[]){  
    razTableau(t1,15);  
    razTableau(t2,35);  
}
```

- Les déclarations ci-dessous définissent une variable pointeur sur un caractère :

```
char t[] ; char *p ;  
T = "une " ; p = "deux"
```

- Pour référencer un élément on peut procéder:

- Soit par indexation

- `t[1]` vaut 'n', `t[2]` vaut 'e'

- `p[1]` vaut 'e', `p[2]` vaut 'u'

- Soit par indirection

- `*t` vaut 'u', `*(t+2)` vaut 'e'

- `*p` vaut 'd', `*(p+1)` vaut 'e'

9. Exemple 2

```
/* nom du module : occ_let.c
 *   Imprime le nombre de lettres dans un texte lu sur
 *   l'entrée standard
 */
#include <stdio.h>
/*définition de type */
typedef enum { FAUX, VRAI } boolean;
/* *****/
/* les prototypes de fonctions */
/******/
boolean estuneLettreMinuscule (char);
boolean estuneLettreMajuscule (char);

/* le programme principal */
void main ( )
    int nbl [26];
    int c, i;
    for ( i=0; i<26, i+= 1) nbl [ i ] = 0;
```

Exemple 2 (suite)

```
printf ("tapez votre texte");
c = getchar ();
while (c!= EOF){
    if (estuneLettreMinuscule(c))
        nbl[c-'a']+=1;
    else if (estuneLettreMajuscule(c))
        nbl[c-'A']+=1;
    c = getchar ();
}

for(i=0;i<26;i+=1) {
    if (nbl[i]>0)
        printf ("la lettre %c est apparue %d fois\n",
                i+'a', nbl[i]);
}
printf (" \n ");
} /* du main */
```

Exemple 2 (suite)

```
/*  
déclaration des fonctions  
***/  
int estuneLettreMinuscule (char c){  
    if (c <= 'z'  && c>= 'a')  
        return (VRAI);  
    else  
        return (FAUX);  
}  
  
int estuneLettreMajuscule (char c) {  
    if (c <= 'Z'  && c>= 'A')  
        return (VRAI);  
    else  
        return (FAUX);  
}
```