MACS1 – Sup Galilée

Langage C avancé : Séance 1

Marc TAJCHMAN
e-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Les supports de ce cours sont disponibles sur Github

Ouvrir la page web : https://github.com/tajchman/MACS1_LC

Ce support de cours et plusieurs exemples vus dans cette séance sont disponibles dans la page internet:

https://github.com/tajchman/MACS1_LC/Seance_1

Avant chaque séance du cours, les supports seront ajoutés dans

https://github.com/tajchman/MACS1_LC

Objectifs du cours

- Rappels de notions de base de programmation en langage C
- Quelques notions : pointeurs, structures, mémoire dynamique, etc.
- Ajouts récents dans le langage
- Maîtrise des outils de développement
- Test du code, corrections des erreurs
- Conseils pour du code maintenable, robuste, évolutif
- Utilisation comme langage de base

Prérequis

- On suppose que vous savez ce qu'est un répertoire, un fichier;
- Que vous pouvez ouvrir un terminal, dans lequel vous pouvez taper des commandes;
- Que vous pouvez créer, détruire un répertoire, et positionner un terminal dans un répertoire;
- Que vous savez utiliser un éditeur de texte, pour créer et écrire un fichier ou modifier un fichier existant.
- On supposera que vous avez déjà une première expérience avec le langage C

Un peu d'histoire :

- C est un des plus anciens langages de programmation (créé en 1972)
- C est encore aujourd'hui un des langages les plus utilisés (2ème position début 2026)
- Plusieurs normes du langage (C89, C99, C11, C17, C23) au cours des années

Philosophie du C :

- *faire confiance au programmeur*
- *obtenir un code le plus efficace possible (même au détriment de la portabilité)*

Les codes écrits en C sont souvent plus rapides que des codes équivalents écrits dans d'autres langages. Mais C est un langage « dangereux » .

Exemple de code en 2 versions

Dans le répertoire exemple2, on trouvera 2 versions d'un code qui copie le contenu d'un fichier dans un autre.

Les 2 versions sont équivalentes mais la seconde est plus rapide que la première.

Expliquer pourquoi.

Code source C

Le code source est un texte écrit dans un langage de programmation (par exemple C), lisible par un humain

Fichier `exemple1.c` dans `exemples/exemple1`

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("hello world\n");
    return 0;
}
```

Dans le code source, on décrit l'organisation des données et des résultats et les opérations à effectuer.

C est un langage impératif

Les instructions sont exécutées dans un ordre décidé par le programmeur et modifient des zones mémoires réservées par le programmeur (accessibles par des variables).
Elles sont regroupées dans un ensemble de fonctions, exemple :

```c
int F1(double y)
{
    return (int) y*1.5;
}

int F2(double x)
{
    double a = x*3.4;
    int b = F1(a) + F1(a*2.5);
    return b;
}
```

```c
int main()
{
    double u = 2.4;
    int v = F2(u);
    return 0;
}
```

Une (et une seule) de ces fonctions doit avoir le nom « main »,
l'exécution commencera par la 1ère instruction dans « main »

C est un langage typé

Types en C

- 4 types de base permettant de définir des valeurs et variables simples :
  char (caractère), int, float, double

par exemple : double u;
u = 2.4;

u est une variable de type double définie par le programmeur
2.4 est une valeur de type double

et des variantes : unsigned int, long, wchar_t, …
(différents intervalles de valeurs possibles, avec ou sans signe)

Une variable occupe une zone mémoire, une valeur n'occupe pas de place en mémoire, mais peut être rangée dans une variable

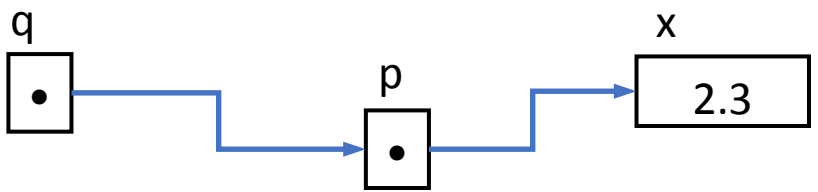
Types en C (2)

D'autres types sont définis à partir des types précédents

- Pointeur qui contient l'adresse d'une zone mémoire, exemples :

```
double x = 2.3;
double *p = &x;
double **q = &p;
char *s;
```

x est une variable de type double, p un pointeur sur un double, q un pointeur sur un pointeur sur double, s un pointeur sur caractère




```
void * p;
```

Type spécial de pointeur quand on ne connaît pas le type de la zone mémoire pointée par p

Types en C (3)

- Vecteurs (ensemble de valeurs de même type)

  Plusieurs façons de définir des vecteurs:

```
double x[4];

int n = 5;
int k[n];

double *y;
y = (double *) malloc(n * sizeof(double));
free(y);
```

`x` est un vecteur statique de 4 doubles,
`k` un vecteur de 5 entiers (forme non conseillée)
`y` un vecteur dynamique de 5 doubles

Types en C (4)

- Structure : agrégation d'un nombre fixe de composantes nommées et de type différents ou non

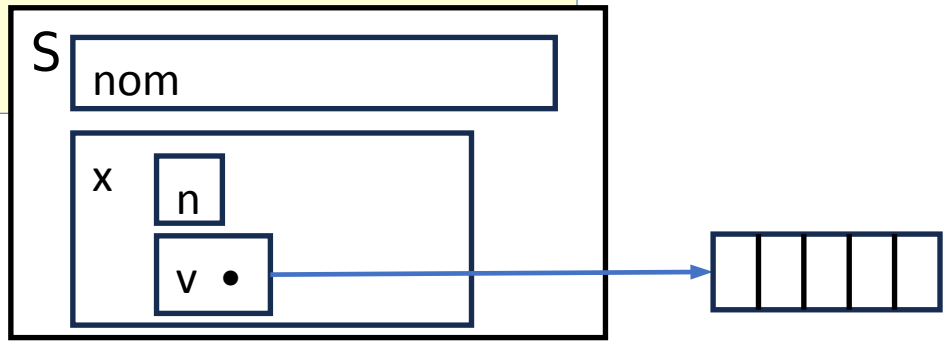
Les composantes d'une structure peuvent être des types simples, ou des pointeurs ou des vecteurs ou d'autres structures, exemple:

```
struct V {
 int n;
 double * v;
};

struct S {
  char nom[10];
  struct V x;
};
```

```
struct S z;

strcpy(z.nom, "exemple");
z.x.n = 5;
z.x.v = (double *)malloc(z.x.n *
                         sizeof(double));
z.x.v[2] = 3.2;
```



Types en C (5)

Un octet est une zone mémoire utilisable de la plus petite taille (8 bits, chaque bit peut prendre la valeurs 0 ou 1). Un octet peut être vu comme contenant un entier entre 0 et 255.

Pour tous les types, une fonction standard : **sizeof**, calcule la taille (en nombre d'octets) occupée directement en mémoire par une variable de ce type

Exemples: voir exemple3/exemple3.c

sizeof(int)	: taille d'un entier
sizeof(double)	: taille d'un double
sizeof(double *)	: taille d'un pointeur de double
sizeof(double [10])	: taille de 10 doubles
sizeof(struct V)	: taille >= somme des tailles d'un entier et d'un pointeur de double

Types en C (6)

Toutes les variables utilisées dans le code source C doivent avoir un type défini et ne peuvent pas changer de type au cours de l'exécution.

Toutes les valeurs utilisées dans le code source doivent être représentées par un des types définis ci-dessus.

C est un langage compilé

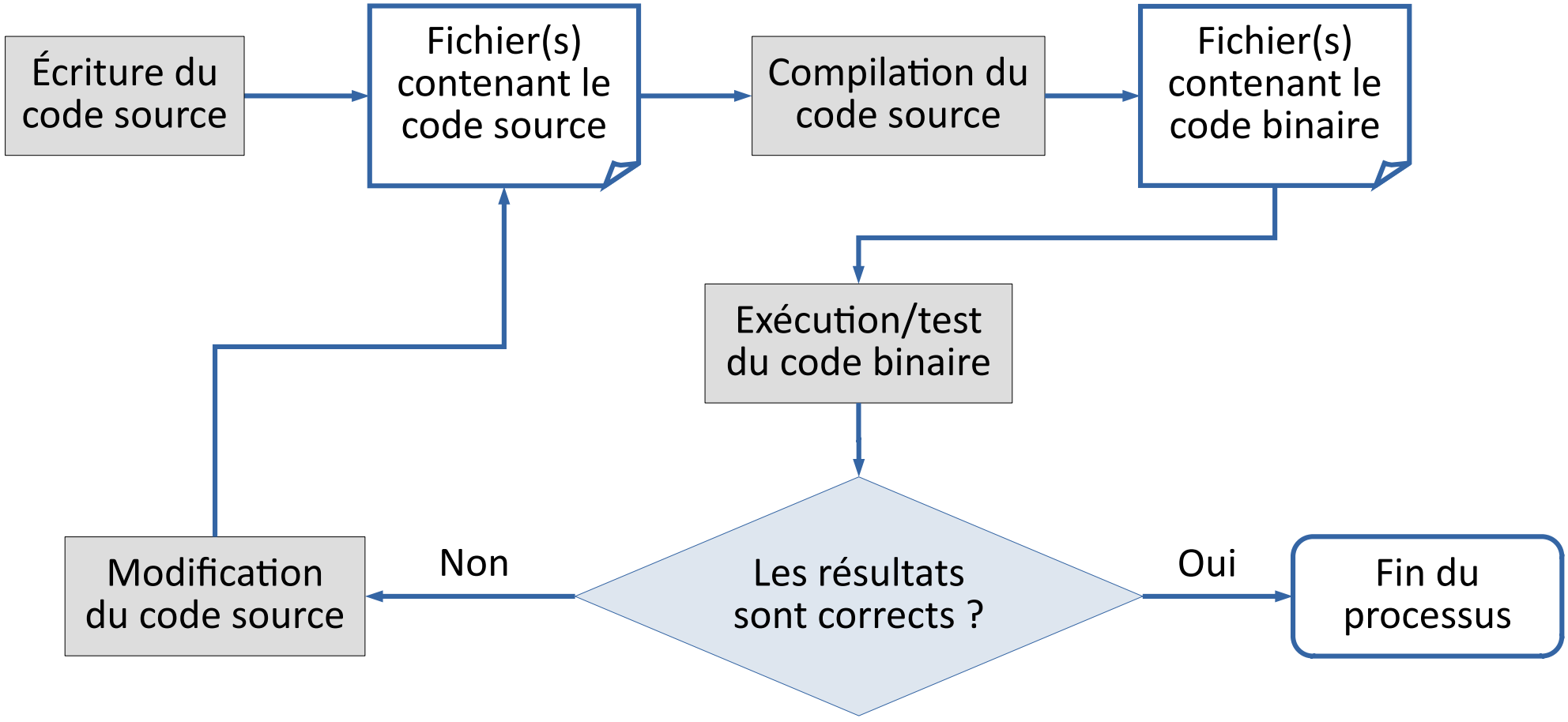
Compilateur

Pour traduire le **code source** (écrit par le programmeur humain) en **code binaire** : instructions bas niveau (compréhensibles par l'ordinateur), un logiciel appelé compilateur est utilisé.

Il existe plusieurs compilateurs : gcc, clang, icx (Intel), xlc (IBM), aocc (AMD), cl.exe (Microsoft), etc.

Leur disponibilité dépend du système qui gère l'ordinateur (Linux, Windows, MacOS, …) que vous utilisez.

Processus pour aboutir à un code binaire



Compilation, exécution d'un code

Ouvrir un terminal dans le répertoire exemples/exemple1.
Dans ce répertoire, il y a un fichier exemple1.c contenant un exemple de code source C

- Pour compiler, tapez la ligne (exemple avec gcc) :

  gcc exemple1.c -o exemple1

  qui génère l'exécutable exemple1.

  exemple1 contient les instructions bas niveau pour produire le résultat des instructions haut niveau contenues dans exemple1.c.

- Pour exécuter, tapez la ligne :

  ./exemple1

En interne la compilation se fait en plusieurs étapes

La compilation, exemple : `gcc exemple1.c -o exemple1` est faite en interne en plusieurs étapes :

1. Pré-traitement : **preprocessing**
   ```
   gcc -E exemple1.c -o exemple1b.c
   ```
2. Génération d'un code source intermédiaire en langage **assembleur**
   ```
   gcc -S exemple1b.c -o exemple1b.s
   ```
3. Génération d'instructions binaires correspondant au code source
   ```
   gcc -c exemple1b.s -o exemple1b.o
   ```
4. Un fichier exécutable (code binaire) est créé avec le(s) fichier(s) créés par (3) et des instructions binaires système : **édition de liens**
   ```
   gcc exemple1b.o -o exemple1
   ```

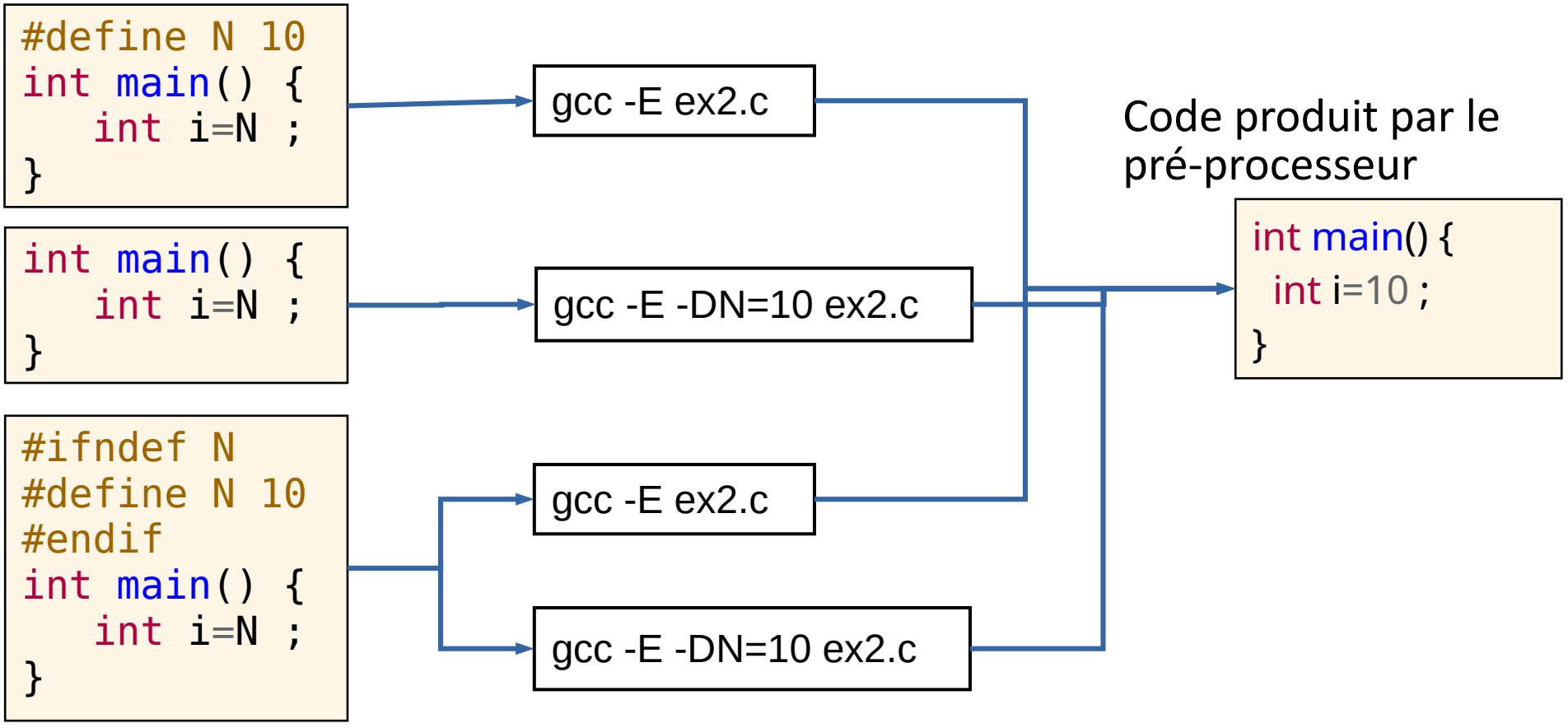
Première étape : preprocessing

Dans la 1ère étape, un **préprocesseur** génère à partir du code source, un **second code source** où les lignes qui commencent par # sont traitées.
Par exemple :

```
#include "toto.h" /* ligne remplacée par le contenu du
fichier toto.h */
#define N 10 /* remplace N par 10 dans toute la suite
                du fichier, */
#ifdef condition
...          /* ne compile les lignes entre ifdef et endif */
#endif       /* que si "condition" à la valeur vraie */
```

Exemple de traitement du code par le pré-processeur

Codes source ex2.c (plusieurs versions)



Seconde étape : le code source écrit par le programmeur est traduit en langage assembleur

Le compilateur traduit ensuite le code source dans un langage intermédiaire (assembleur)

```
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    subq  $16, %rsp
    movl  %edi, -4(%rbp)
    movq  %rsi, -16(%rbp)
    leaq  .LC0(%rip), %rax
    movq  %rax, %rdi
    call  puts@PLT
    movl  $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Pour voir le code assembleur, tapez gcc -S exemple1.c et examiner le fichier exemple1.s

Troisième étape : l'assembleur (code intermédiaire) est traduit en liste de hexadécimaux (entiers en base 16)

Certains hexadécimaux représentent des instructions bas niveau, d'autres des valeurs numériques ou des lettres, etc

```
...
0003010 4347 3a43 2820 6255 6e75 7574 3120 2e33
0003020 2e33 2d30 7536 7562 746e 3275 327e 2e34
0003030 3430 2029 3331 332e 302e 0000 732e 7368
0003040 7274 6174 0062 692e 746e 7265 0070 6e2e
0003050 746f 2e65 6e67 2e75 7270 706f 7265 7974
0003060 2e00 6f6e 6574 672e 756e 622e 6975 646c
0003070 692d 0064 6e2e 746f 2e65 4241 2d49 6174
0003080 0067 672e 756e 682e 7361 0068 642e 6e79
0003090 7973 006d 642e 6e79 7473 0072 672e 756e
00030a0 762e 7265 6973 6e6f 2e00 6e67 2e75 6576
```

Pour produire ces valeurs binaires à partir du code assembleur, tapez :

gcc -c exemple1.s -o exemple1.o ou gcc -c exemple1.c -o exemple1.o

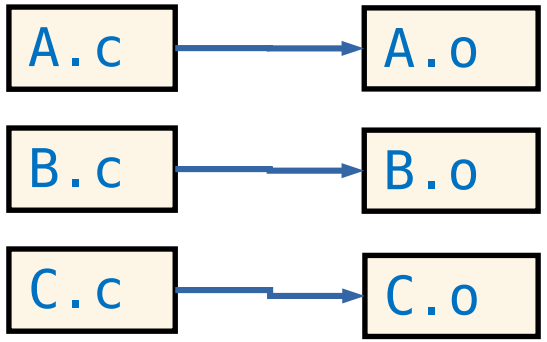
Pour les afficher à l'écran, tapez :

od -c exemple1.o

Si le code source est contenu dans plusieurs fichiers

Le compilateur traduit en instructions bas niveau chaque fichier source C **séparément** :

Exemple :
dans le répertoire exemple2, les fichiers A.c, B.c et C.c contiennent, chacun, une partie du code source



Pour voir le résultat de cette phase, tapez (pour gcc) :

```
gcc -c A.c -o A.o
gcc -c B.c -o B.o
gcc -c C.c -o C.o
```

attention : A.o, B.o et C.o sont des binaires non exécutables (ils sont incomplets)

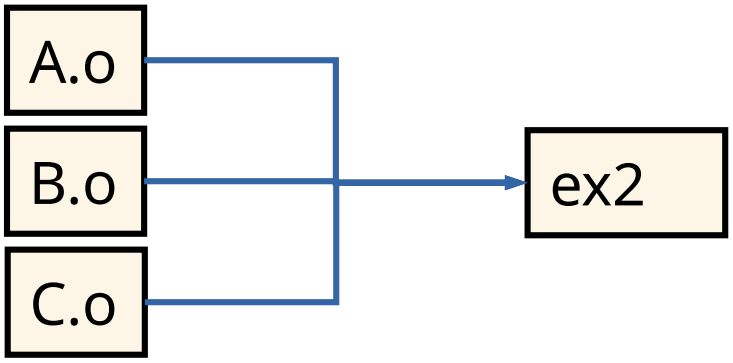
Quatrième étape : Édition de liens (link)

Dans cette étape, l'éditeur de liens :

- Examine les fichiers binaires générés à l'étape précédente
- Vérifie que, si une fonction/variable globale est utilisée dans un des fichiers binaire, cette fonction/variable globale est définie quelque part dans les différents fichiers binaires
- Vérifie que toutes les fonctions sont définies une seule fois
- Vérifie qu'il existe une et une seule fonction « int main(...) » (indique où l'exécution doit commencer)
- Supprime (éventuellement) les fonctions définies mais non utilisées
- Rassemble les fichiers binaires dans un seul fichier qui peut être exécuté

A la fin de cette étape, si le compilateur n'a pas trouvé d'erreur, il crée un fichier binaire prêt à être exécuté sur l'ordinateur.

Quatrième étape : Édition de liens (2)



Pour la compilation complète, tapez (cas de gcc) :

```
gcc -c A.c -o A.o
gcc -c B.c -o B.o
gcc -c C.c -o C.o
gcc A.o B.o C.o -o ex2
```

On peut faire toutes les étapes en une seule commande :

```
gcc A.c B.c C.c -o ex2
```

C’est plus rapide écrire mais c’est déconseillé si le code source de grande taille et réparti dans de nombreux fichiers (voir séance sur les outils type make, cmake, ...)

Variables

Pour s'exécuter un code doit réserver une zone mémoire qui va contenir les données utilisées et résultats produits.
L'utilisateur doit donc utiliser des variables qui sont des emplacements mémoire de taille suffisante pour contenir des valeurs modifiables dont le type est défini dès la réservation.

Exemples :

```
int i ;    /* déclaration d'une variable entière
              (réservation d'un espace mémoire assez grand
               pour un entier) */
double x ; /* déclaration d'une variable décimale */

i = 3 ;    /* modification de la valeur dans la variable i */
x = 4.4 ;  /* modification de la valeur dans la variable x */
```

Variables

Toutes les variables utilisées dans un code source doivent avoir un type défini et non modifiable, et on peut seulement les modifier avec des valeurs du même type.

La seule dérogation est s'il existe une conversion implicite entre le type d'une valeur et le type de la variable qui doit contenir cette valeur.
Il y a des conversions de type dangereuses ou non.

Si une variable est utilisée mais non déclarée, le compilateur suppose qu'elle est de type entier.

(voir conseils de bonne pratique, 2 pages plus loin)

Variables (2)

Exemples avec les variables définies précédemment :

```
x = 4 ;   /* x contiendra le double 4.0 */
```

Il y a une conversion implicite entier → double. Pas de problèmes, une variable double est capable de représenter un entier sans perte de précision.

```
i = 3.5 ; /* i contiendra l'entier 3 */
```

Il y a une conversion implicite double → entier. La partie décimale de 3.5 (0.5) est perdue. Les compilateurs acceptent cette instruction, certains préviennent le programmeur.

Si c'est vraiment ce que veut le programmeur, il est préférable d'écrire

```
i = (int) 3.5 ;
```

(on utilise une conversion explicite avant de ranger la valeur dans la variable)

Variables : bonnes pratiques

Même si un compilateur accepte de ne pas l'écrire dans le code source:

- Définir explicitement le type de toutes les variables utilisées
- Rechercher les conversions implicites entre variables de type différents et y mettre une conversion explicite

Pour compiler, surtout dans la phase de développement d'un code source, utiliser les options de vérification du compilateur et tenir compte des messages d'avertissement (warnings)

- Pour gcc et clang, utiliser les options

  –Wall –Werror –Wextra –pedantic

- Pour les autres compilateurs, regarder leur documentation

Les vecteurs

Qu’est ce qu’un vecteur ?

- **Un vecteur est un ensemble de n éléments de même type** (int, double ou autre type C)

- Pour utiliser un vecteur de type T (T = int, double ou autre type C), on passe par un pointeur : `T * v;`

  (explicitement ou implicitement)

- En mémoire:

Composantes d'un vecteur

Les composantes d'un vecteur v de taille n sont accessibles sous la forme:

```
v[i] = …     /* écriture (modification) */
x = … v[j] … /* lecture  (utilisation) */
```

où i et j sont des entiers entre 0 et n-1.

Toute utilisation de v[i] où

- i est négatif ou
- i est égal ou supérieur à n

est une erreur (on verra dans une autre séance des outils pour détecter ce type d'erreur)

Plusieurs façons de définir un vecteur (1)

Vecteur de taille fixe définie dans le code source

```
double v[3];
```

La taille de v est constante (ici 3), mais chaque composante peut être modifiée:

```
v[2] = v[1] + 1.4;
```

Attention, à la déclaration, les composantes d'un vecteur n'ont pas de valeur prédéfinie (en particulier, pas 0)

Il y a un pointeur (fixe) invisible pour l'utilisateur vers le début de la zone mémoire occupée par le vecteur.

Plusieurs façons de définir un vecteur (2)

Vecteur de taille fixe et initialisé à la définition (la taille du vecteur est celle du nombre de valeurs initiales)

```
double v[] = {1.2, 3.4};
```

La taille de v est constante (ici 2), mais chaque composante peut être modifiée:

```
v[0] = v[1] + 1.4;
```

Plusieurs façons de définir un vecteur (3)

Vecteur de taille fixe mais dont la taille n'est pas connue dans le code source (seulement à l'exécution)

```
int n = …
double v[n];
```

Cette façon de définir un vecteur (VLA : variable length array) **est déconseillée** : cela ne marche pas avec tous les compilateurs et/ou pour des tailles de vecteur (n) assez grandes.

Utiliser plutôt l'allocation dynamique de la mémoire (page suivante).

Plusieurs façons de définir un vecteur (4)

Vecteur dont la taille peut être spécifiée à l'exécution et dont la mémoire est gérée dynamiquement.

Par exemple, pour un vecteur de taille n et de type entier :

- On déclare d'abord un pointeur sur entier: `int * v;`
- On réserve (alloue) la mémoire suffisante pour n entiers (v désigne le début de la mémoire réservé par le système):

```
v = (int *) malloc(n * sizeof(int));
```

- Le vecteur peut être utilisé (n'oubliez pas de l'initialiser)

```
v[0] = 1;
V[1] = v[0] + 3;
```

Plusieurs façons de définir un vecteur (4)

- Quand le vecteur n'est plus utilisé, on signale au système que sa mémoire est disponible (on libère la mémoire)

```
free(v);
```

- Le pointeur est à nouveau disponible pour un autre vecteur

```
v = (int *) malloc(m * sizeof(int));
```

Il n'y a pas de restriction sur la taille du vecteur (sauf la taille totale disponible dans le système).
Si malloc ne parvient pas à réserver la mémoire, il a comme résultat un pointeur null (NULL), pour plus de sécurité, ajouter un test sur le pointeur:

```
if (v == NULL)
   exit(-1); /* ou autre traitement d'erreur */
```