Institut
Mines-Télécom

Introduction Langage C et Systèmes d'exploitation
Partie 3: Langage C – Chaine de production

Responsable : Etienne Borde (C218-3)

Auteurs : Bertrand Dupouy et Etienne Borde

Plan

1. Généralités :
    - Les outils de production :compilateur, assembleur, éditeur de liens
    - compilation séparée
    - directives : #include, …
    - espace d'adressage d'un programme

2. l'outil make
    - cible, dépendance
    - fichier Makefile de base
    - un peu plus sur le fichier Makefile

Plan

1. *Généralités :*

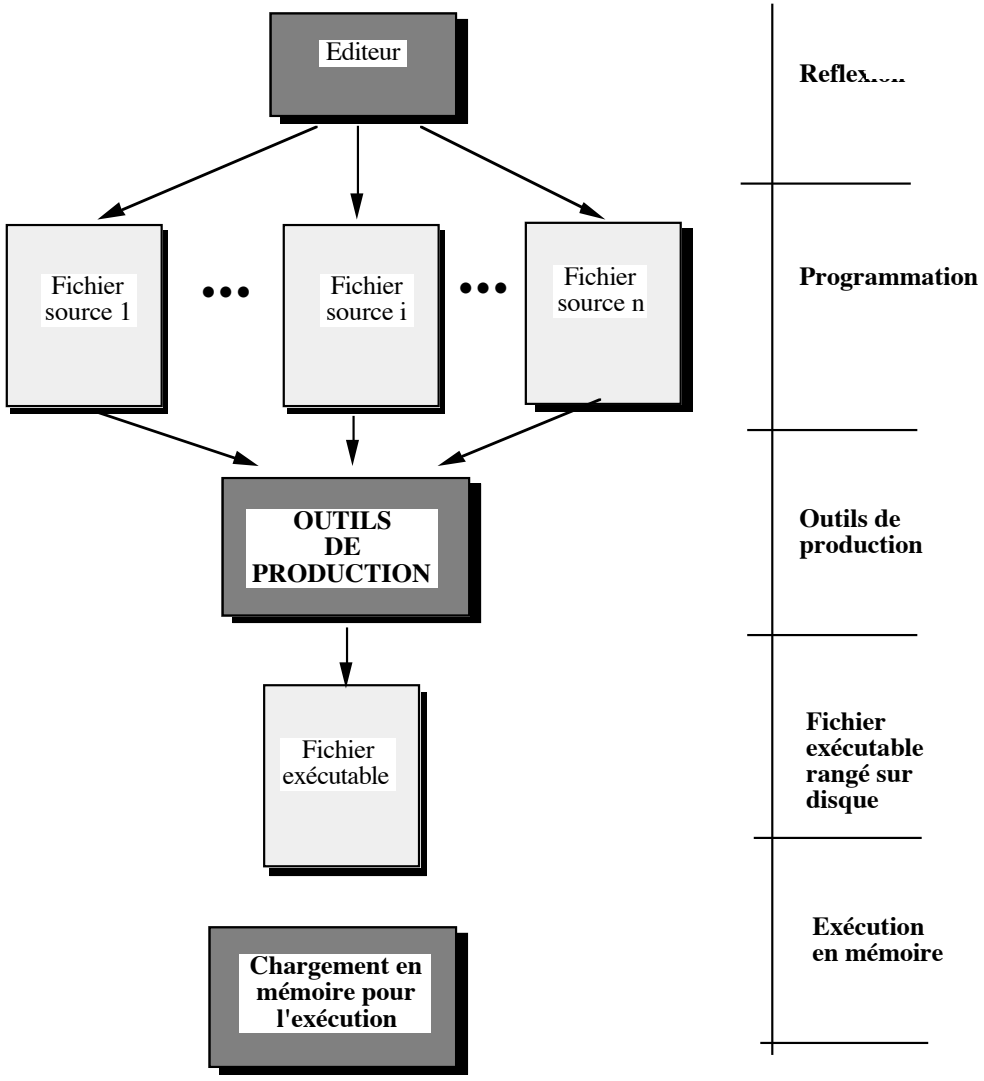
   ☞ **. Les outils de production :compilateur, assembleur, éditeur de liens**

   - *compilation séparée*
   - *directives : #include, …*
   - *espace d'adressage d'un programme*

2. *l'outil make*

   - *cible, dépendance*
   - *fichier Makefile de base*
   - *un peu plus sur le fichier Makefile*

Place des outils de production

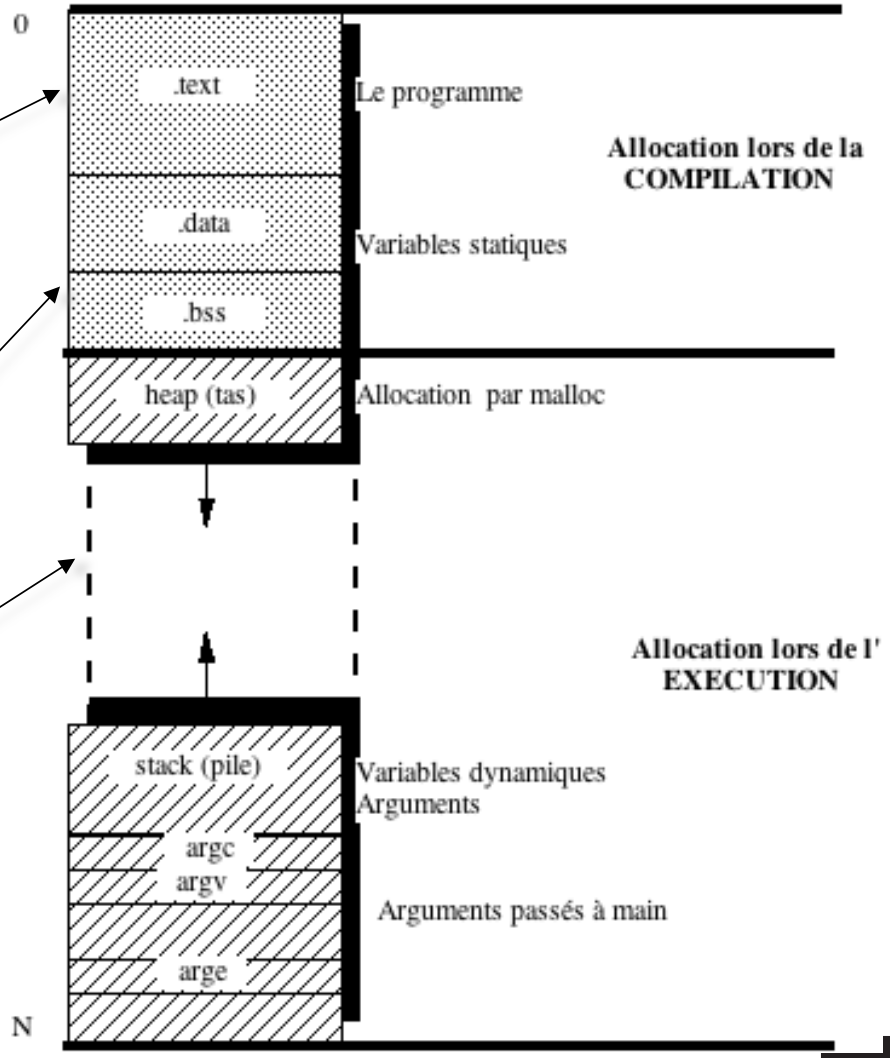


Chargement en mémoire pour exécution

Quand une application est chargée en mémoire, le système lui alloue de l'espace pour :

1. la partie « programme » (les instructions). Ces instructions sont celles du langage machine résultant de la traduction des instructions du langage de haut niveau (C, Java, Ada, etc).

2. les données. On distingue deux partitions :
    1. L'une, créée au moment de la compilation, dite statique.
    2. L'autre, dédiée aux informations allouées dynamiquement, pendant l'exécution du programme.



Langage de haut niveau et langage machine

LHN

```
int glob =255;
int main(){
  static int j=4;
  if(j == 4)
glob=2;
  return 0;
}
```

Langage d'assemblage

```
main:
    ...
    cmpl    $4, j
    jne       L2
    movl    $2, glob
L2:
    ...
```

Codage binaire en mémoire

```
...

11: 83 f8 04                (cmp    $4, j)
14: 75 0a                   (jne    20)
16: c7 05 00 00 00 00 02    (movl   $2,glob)
1d: 00 00 00
20:
```

- Cet exemple montre la traduction d'un fichier source écrit en langage de haut niveau (LHN), ici le C, en langage d'assemblage, ou encore langage machine (LM).
- Le langage d'assemblage est lui-même une **représentation symbolique** du codage binaire des instructions, seule représentation compréhensible par le processeur
- Remarque sur le langage machine: l'instruction `if( == )` du LHN correspond à deux instructions du LM (`cmpl` et `jne`).

Les outils de production (du LHN vers le LM)

1. **Compilateur**
   - Traduit le fichier source (*source file*), écrit en langage de haut niveau, en langage d'assemblage et affecte des adresses aux variables, il construit ainsi un fichier appelé fichier objet (*object file*),
   - Dans ces fichiers objets, certaines références ne sont pas satisfaites : par exemple le compilateur rencontre le symbole `printf`, mais ne trouve pas la fonction en question,

2 **Editeur de liens (*linker*)**

   - Rassemble les fichiers objets produits par le(s) traducteur(s), pour construire un fichier exécutable, il essaie de résoudre toutes les références non satisfaites en consultant des bibliothèques,
   - Pour fusionner les espaces d'adressage séparés des modules objet en un seul espace linéaire, l'éditeur de liens doit, par exemple :
     - Modifier les instructions qui contiennent des références à la mémoire,
     - Résoudre les références non satisfaites en insérant l'adresse des fonctions à l'endroit où elles sont appelées,

Un outil de production : gcc

- `gcc` n'est pas seulement un compilateur. Cette commande, en fonction du type de fichier passé en argument, enchaine différentes tâches. Par exemple, voici ce que fait la commande `gcc fic.c` :

  1. appel du préprocesseur C (`cpp`) qui crée un fichier `fic.i`,
  2. appel du compilateur C (`cc1`) qui crée un fichier assembleur `fic.s`,
  3. appel de l'assembleur (`as`) pour générer le fichier objet `fic.o` à partir de `fic.s`,
  4. appel de l'éditeur de liens (`ld`), la bibliothèque C standard est incluse par défaut, un fichier exécutable `a.out` est produit, si possible. Les fichiers intermédiaires sont détruits.

- Deux options permettent de suivre ces étapes :

`gcc -v` (une trace de toutes les étapes intermédiaire est affichée à l'écran)

`gcc -save-temps` (sauvegarde des fichiers intermédiaires, cf. page suivante)

gcc : les fichiers temporaires

```
$gcc -save-temps  fic.c

$ ls -ltr
-rw-r--r-- 1 domas s3   882  9 déc.  18:45 fic.c
-rw------- 1 domas s3 17128 10 déc.  14:53 fic.i
-rw------- 1 domas s3   568 10 déc.  14:53 fic.s
-rw------- 1 domas s3  1304 10 déc.  14:53 fic.o
-rwx------ 1 domas s3  6519 10 déc.  14:53 a.out
```

Commentaires :

- `fic.i` : résultat des effets des commandes commençant par # sur le fichier `fic.c`. Créé par `cpp`, entrée de `cc1`
- `fic.s` : fichier assembleur temporaire créé par `cc1`, entrée de `as`,
- `fic.o` : fichier objet temporaire créé par `as`, entrée de `ld`
- `a.out`: fichier créé par `ld`, le fichier exécutable

Structure d'un fichier objet

Un fichier objet contient les informations suivantes :

1. un **en-tête**, qui indique la taille et l'emplacement en mémoire de la zone objet, le nom du fichier source à partir duquel il a été créé, la date de création;
2. l'espace objet proprement dit divisé en (au moins) trois parties :
   - la **table des symboles** qui contient des informations sur les symboles locaux, les symboles de bibliothèque et les références non satisfaites;
   - le code binaire des instructions (section **text**),
   - les sections de données : **data** (données initialisées par le programme) et **bss**,
3. des informations pour la mise au point si l'option –g a été spécifiée,

Fichier objet : commandes `nm` et `objdump -s`

```
$gcc -c  fic.c
$nm fic.o
 0000000000000000 D glob
 0000000000000004 d j.2167
 0000000000000000 T main
                  U printf

$objdump -s fic.o
 Contents of section .text:
 0000 554889e5 4883ec10 897dfc48 8975f08b
 0010 15000000 008b0500 00000089 c6bf0000
 0020 0000b800 000000e8 00000000 b8000000
 0030 00c9c3
 Contents of section .data:
 0000 16000000 04000000
 Contents of section .rodata:
 0000 6a203d25 642c2067 6c6f6220 3d25640a  j =%d, glob =%d.
 0010 00
```

```
$cat fic.c

#include <stdio.h>
int glob =22;
int main (int argc, char * argv[]){
    static int j =4;
    printf("j =%d, glob =%d\n", j, glob);
    return 0;
}
```

Commentaires (les sorties des commandes ont été simplifiées) :

- `nm` : affichage des symboles : glob et j sont aux adresses 0 et 4 en section .data, main en 0 dans .text, le symbole **printf est inconnu.**
- `objdump -s` : contenu de la table des symboles. Dans `.text` le code binaire; dans `.data` glob (22->16 en base 16) et j (4). Dans la partie read only de data, on trouve la chaine de caractères correspondant au format.
- Dans chaque section les adresses sont calculées à partir de zéro

Editeur de liens

- Objectif : créer un fichier exécutable en rassemblant de fichiers objets et des bibliothèques,
- Etapes importantes :
  - résoudre les références non satisfaites en insérant l'adresse des fonctions à l'endroit où elles sont appelées,
  - ajouter des informations cachées (*glue*) pour permettre le démarrage du programme par le système (fichier `crt0.o` pour le système Unix)
  - éventuellement : ajouter des informations telles que celles qui sont nécessaires à l'utilisation d'un *debugger*
- Le fichier obtenu peut être très volumineux. Pour diminuer sa taille, on remplace le code objet d'une fonction de bibliothèque par un pointeur. Ce pointeur donne l'adresse d'un module exécutable qui sera utilisé lors de l'exécution du programme. Ce module sera partagé par tous les programmes qui font appel à cette fonction. On parle alors d'édition de liens dynamique et de bibliothèques partagées(*shared libraries*).

Fichier exécutable : commandes `nm` et `objdump -s`

```
$gcc   fic.c
$nm a.out
00000000006008c4 D glob
00000000006008c8 d j.2167
00000000004004dc T main
                 U printf@@GLIBC_2.2.5

$objdump -s a.out
 Contents of section .text:
4003d0 31ed4989 d15e4889 e24883e4 f0505449
...
400580 4c8b6424 184c8b6c 24204c8b 7424284c
400590 8b7c2430 4883c438 c30f1f80 00000000
4005a0 f3c36690
Contents of section .data:
 6008c0 00000000 16000000 04000000
 Contents of section .rodata:
 4005c0 6a203d25 642c2067 6c6f6220 3d25640a  j =%d, glob =%d.
 4005d0 00
```

Les adresses ont été recalculées, la section **`.text`** est plus volumineuse (ajout de crt0, etc)

Rappel du résultat obtenu pour le fichier objet :

```
$nm fic.o
0000000000000000 D glob
0000000000000004 d j.2167
0000000000000000 T main
                 U printf
```

Commentaires (les sorties des commandes ont été simplifiées) :

- `nm` : le symbole printf est une référence vers une bibliothèque dite « dynamique ». printf ne sera chargée qu'à l'exécution, si nécessaire.
- `objdump -s` : chaque section a une **adresse**

Binaire objet et binaire exécutable

```
$objdump -d fic.o
 0:    55                          push   %rbp
   1:  48 89 e5                    mov    %rsp,%rbp
   4:  48 83 ec 10                 sub    $0x10,%rsp
   ...
   f:  8b 15 00 00 00 00           mov    0x0(%rip),%edx        # 15 <main+0x15>
  15:  8b 05 00 00 00 00           mov    0x0(%rip),%eax        # 1b <main+0x1b>
  1b:  89 c6                       mov    %eax,%esi
  1d:  bf 00 00 00 00              mov    $0x0,%edi
  22:  b8 00 00 00 00              mov    $0x0,%eax
  27:  e8 00 00 00 00              callq  2c <main+0x2c>
```

```
$objdump -d a.out
 00000000004004dc <main>:
  4004dc:     55                          push   %rbp
  4004dd:     48 89 e5                    mov    %rsp,%rbp
  4004e0:     48 83 ec 10                 sub    $0x10,%rsp
  4004e4:     89 7d fc                    mov    %edi,-0x4(%rbp)
  4004e7:     48 89 75 f0                 mov    %rsi,-0x10(%rbp)
  4004eb:     8b 15 d3 03 20 00           mov    0x2003d3(%rip),%edx        # 6008c4 <glob>
  4004f1:     8b 05 d1 03 20 00           mov    0x2003d1(%rip),%eax        # 6008c8 <j.2167>
  4004f7:     89 c6                       mov    %eax,%esi
  4004f9:     bf c0 05 40 00              mov    $0x4005c0,%edi
  4004fe:     b8 00 00 00 00              mov    $0x0,%eax
  400503:     e8 a8 fe ff ff              callq  4003b0 <printf@plt>
  400508:     b8 00 00 00 00              mov    $0x0,%eax
  40050d:     c9                          leaveq
  40050e:     c3                          retq
  40050f:     90                          nop
```

Dans le binaire **exécutable** les adresses sont recalculées et
la référence vers printf est résolue par appel à la fonction de la bibliothèque dynamique.

Fichiers exécutables : remarques

- **a.out :** pourquoi ce nom ? il s'agit du nom du format originel des fichiers objets et exécutables sous Unix. Les fichiers actuels sont au format ELF (Executable and Linking Format), parmi les différences : des sections ont été ajoutées aux .text, .data et .bss originelles.

```
$ file fic.o
fic2.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
$file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.32,not stripped
```

- **`relocatable`**: adresses relatives, ce fichier pourra être rassemblé avec d'autres objets pour construire un fichier **`executable`**

- **`dynamically linked :`** ce fichier exécutable contient des références vers des bibliothèques dites partagée ou dynamiques

Bibliothèques partagées

• Les fonctions de ces bibliothèques sont chargées en mémoire à **l'exécution** du programme. Si la fonction est déjà chargée, ce binaire sera partagé par tous les exécutables qui l'utilisent,

• Avantages : les fichiers exécutables sont plus petits, en effet les fichiers objets manquants ne sont pas ajoutés lors de l'édition de liens, mais remplacés par des références vers des fichiers **exécutables**.

• Inconvénients :
  - les modifications faites dans les procédures contenues dans ces bibliothèques, leur déplacement d'un répertoire à un autre ou le changement de leur nom peuvent perturber le fonctionnement de l'application (cf. le message *DLL missing* de Windows),
  - temps d'exécution (encore plus) difficile à estimer,

Java et le *bytecode*

Le compilateur java (`javac`) produit un fichier contenant du *bytecode* :

- ce code n'est exécutable sur aucun processeur,
- il sera interprété par la JVM(*Java Virtual Machine*) : chaque instruction *bytecode* sera traduite en langage machine.
- la JVM commence l'exécution en chargeant la classe contenant la méthode main.

Exemple :

`$javac HelloWorld.java` (production de `HelloWorld.class`)

`$java HelloWorld` (appel à la JVM qui charge `HelloWorld.class`)

Java : du source au bytecode

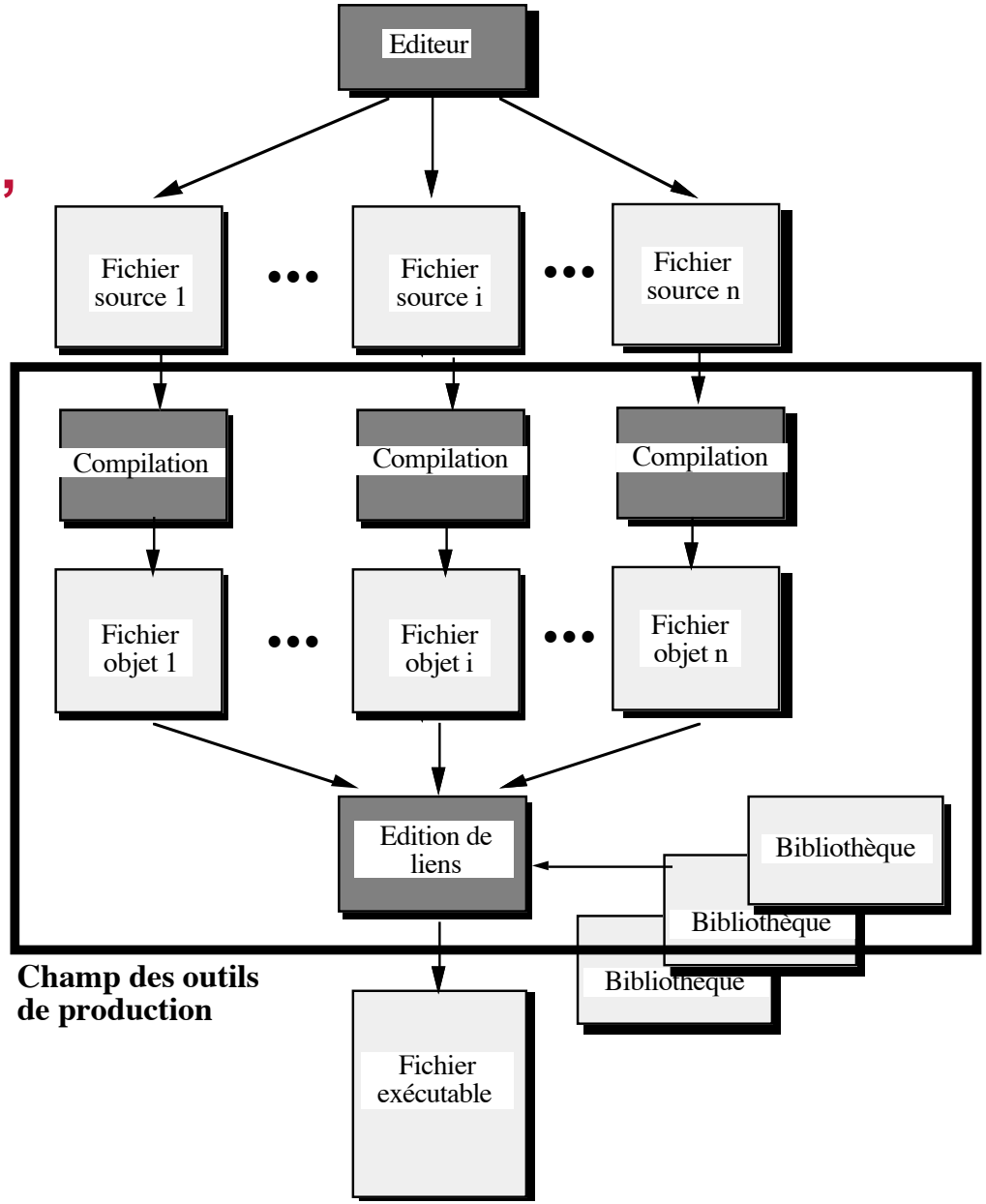
Les fichiers contenant du *bytecode* peuvent être désassemblés grâce à l'utilitaire `javap`.
Par exemple, pour le fichier bytecode créé à partir du fichier source ci-contre, on obtient (commentaires ajoutés en français…) :

```
public class HelloWorld{
public static void main(String args[]){
   int i;
   for(i=0 ; i<10 ; i++);
   System.out.println(" Hello world,  i = " + i);
   }
}
```

```
$javac  HelloWorld.java
$javap -c HelloWorld.class

 0: iconst_0 0          # mettre 0 sur la pile
 1: istore_1            # vider la pile dans la première variable
 2: iload_1             # empiler la premiere variable
 3: bipush        10    # mettre 10 sur la pile
 5: if_icmpge     14    # comparer les deux valeurs en sommet de pile
                        # si >= aller à l'adresse 14
 8: iinc          1,1   # ajouter 1 à la première variable
11: goto          2     # boucler : aller à l'adresse 2
14: ...
    ...
39: return
}
```

Chaine de production, récapitulatif :



Plan

1. *Généralités :*
2. *. Les outils de production :compilateur, assembleur, éditeur de liens*

   ☞ **• compilation séparée**

   • *directives : #include, …*

   • *espace d'adressage d'un programme*

2. l'outil make

   • *cible, dépendance*

   • *fichier Makefile de base*

   • *un peu plus sur le fichier Makefile*

Compilation séparée

Avantages de la compilation séparée :

- Maintenance plus facile,
- Modules réutilisables, partageables,

Si on utilise deux fichiers, `main.c` qui contient la fonction `main`, et `foncs.c` qui contient les fonctions utilisées par `main` :

```
gcc -c main.c
gcc -c foncs.c
```

Compilations séparées des fichiers source et productions de deux fichiers objets.

```
gcc main.o foncs.o -o appli
```

Edition de liens à partir des deux fichiers objets et production d'un fichier exécutable appelé `appli`.

Compilation séparée, exemple (1/2)

Fichier `f1.c`

```
int main (int argc, char *argv){
   int i = 0;
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}
```

Fichier `f2.c`

```
float carre (float var){
  static compteur =1;
  compteur = compteur +1;
   return(var * var);
}
```

- Compilation séparée :

Commandes	Effet de la commande
`gcc -c f1.c`	Production du fichier OBJET `f1.o`
`gcc -c f2.c`	Production du fichier OBJET `f2.o`
`gcc f1.o f2.o -o appli`	Changement du nom du fichier de sortie produit, le défaut étant `a.out` pour un fichier exécutable.

Compilation séparée, exemple (2/2): les fichiers objets (extraits)

$ nm f1.o

```
                 U carre
0000000000000000 T main
                 U printf
```

$ objdump -s f1.o

```
Contents of section .rodata:
 0000 69203d20 25640a00                    i = %d..
```

$ nm f2.o

```
0000000000000000 T carre
0000000000000000 d compteur.1708
```

$ objdump -s f2.o

```
Contents of section .data:
 0000 01000000
```

$ objdump -h f1.o

```
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000043  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000084  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000084  2**2
                  ALLOC
  3 .rodata       00000008  0000000000000000  0000000000000000  00000084  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

$ objdump -h f2.o

```
  0 .text         00000034  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Compilation séparée et variables globales

En C, deux variables globales ne peuvent pas avoir le même nom. Sinon, il y a ambiguïté:

- Visibilité d'une variable globale = toutes les sections de code.
- Durée de vie = durée de vie du programme.

Problème: comment utiliser la compilation séparée et utiliser dans fic.c une variable globale déclarée dans main.c?

C définit pour cela le mot clé **extern,** qui précise qu'une variable est déclarée dans un autre fichier source:

- Une variable « extern » résulte en un symbol non-défini à l'issue de la compilation.
- Le symbole devra être résolu lors de l'édition de lien.

Compilation séparée et variables globales (exemple)

Fichier `f1.c`

```
int i = 0;
int main (int argc, char *argv){
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}
```

Fichier `f2.c`

```
int i = 5;

float carre (float var){
  static compteur =1;
  compteur = compteur +1;
  i = i+1;
  return(var * var);
}
```

Commandes	Effet de la commande
`gcc -c f1.c`	Production du fichier OBJET `f1.o`
`gcc -c f2.c`	Production du fichier OBJET `f2.o`
`gcc f1.o f2.o -o appli`	Erreur, redéfinition du symbole i: le symbole i est définit dans f1.o et f2.o; le compilateur ne sais pas quelle symbole utiliser (ni quelle valeur initiale lui attribuer).

Compilation séparée et variables globales (exemple)

Fichier `f1.c`

```
int i = 0;
int main (int argc, char *argv){
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}
```

Fichier `f2.c`

```
extern int i;

float carre (float var){
  static compteur =1;
  compteur = compteur +1;
  i = i+1;
  return(var * var);
}
```

Commandes	Effet de la commande
`gcc -c f1.c`	Production du fichier OBJET `f1.o`, le symbol i est défini.
`gcc -c f2.c`	Production du fichier OBJET `f2.o`, le symbol i est non défini.
`gcc f1.o f2.o -o appli`	Production de l'application appelé `appli`. L'éditeur de lien procède à la résolution de symbole et associe à i dans f2.o l'adresse de i dans f1.o

Compilation séparée et variables globales (exemple)

Fichier `f1.c`

```
extern int i;
int main (int argc, char *argv){
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}
```

Fichier `f2.c`

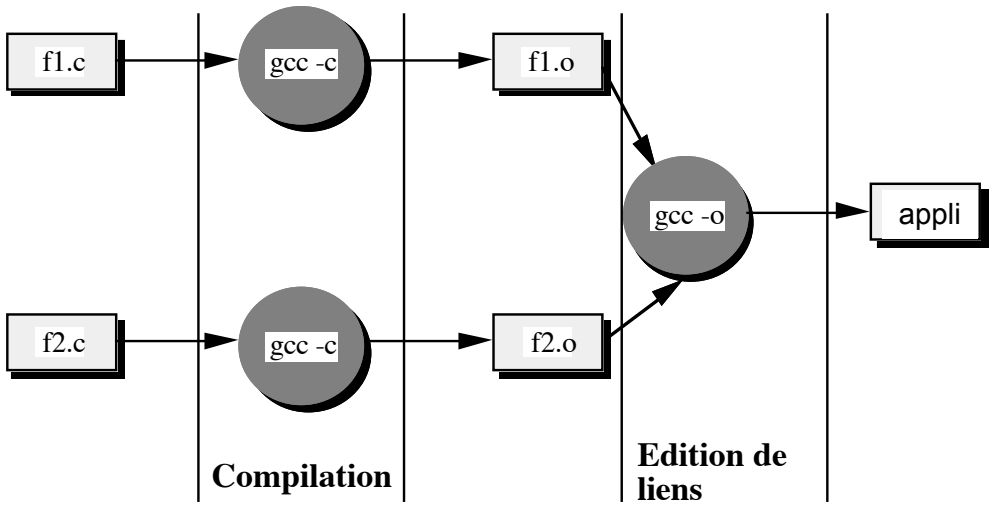
```
extern int i;

float carre (float var){
  static compteur =1;
  compteur = compteur +1;
  i = i+1;
  return(var * var);
}
```

Commandes	Effet de la commande
`gcc -c f1.c`	Production du fichier OBJET `f1.o`
`gcc -c f2.c`	Production du fichier OBJET `f2.o`
`gcc f1.o f2.o -o appli`	Erreur, le symbole i n'est pas défini: il est non-défini dans f1.o et dans f2.o il est donc défini nulle part.

Compilation séparée : résumé

Production d'un fichier exécutable appli à partir des fichiers sources f1.c et f2.c :



• Utilisation de gcc dans le schéma ci-dessus :

`gcc -c f1.c` (produit l'objet `f1.o`)
`gcc -c f2.c` (produit l'objet `f2.o`)
`gcc f1.o f2.o -o appli` (l'exécutable s'appelle `appli` au lieu de `a.out`)

Plan

1. *Généralités :*

   Les outils de production :compilateur, assembleur, éditeur de liens

   • *compilation séparée*

   ☞ . • **directives du préprocesseur : #include, …**

   • *espace d'adressage d'un programme*

2. *l'outil make*

   • *cible, dépendance*

   • *fichier Makefile de base*

   • *un peu plus sur le fichier Makefile*

Le préprocesseur : fonctionnement

- Les lignes commençant par #, appelées directives, sont traitées par le préprocesseur AVANT la compilation
- Nous allons voir quelques unes de ces directives :
  - `#define`
  - `#include`
  - `#ifdef et #endif`
- Remarque :
  - l'option `E` de `gcc` permet de voir l'effet du préprocesseur sur le fichier source

```
/* essai.c */
#define DIM 128
int main(int argc, char*argv[]){
  int i;
  short tab [DIM ] ;
  for (i=0; i<DIM ; i++)tab[i]=0;
  return 0;
}
```

`gcc -E essai.c`

```
int main(int argc, char*argv[]){
  int i;
  short tab [128] ;
  for (i=0; i<128; i++)tab[i]=0;
  return 0;
}
```

Le préprocesseur : #define

- `#define` effectue des **substitutions** de chaines de caractères.
  - Par exemple, dans l'exemple suivant, les occurrences des caractères MAXIMUM sont remplacées par les caractères 100
- Attention aux effets de bord, ici, les parenthèses qui encadrent le symbole « x « les évitent :

```
#define double1(x) (2*x)

#define double2(x) (2*(x))
```

  - `double1(1+2)` donne 4 (* est prioritaire sur +),
  - `double2(1+2)` donne 6,

```
#define MAXIMUM 100
#define double1(x) (2*x)
#define double2(x) (2*(x))

int main(int argc, char*argv[]){
   long int i,j, k;
   i = MAXIMUM ;
   j = 0;
   k = double1(1+2);
   k = double2(1+2);
   return 0;
}
```

Effet du préprocesseur

```
int main(int argc, char*argv[]){
   long int i, j ,k;
   i = 100;
   j = 0 ;
   k = (2 * 1 + 2 );
   k = (2 *( 1 + 2 ));
   return 0 ;
}
```

Le préprocesseur : #ifdef, #endif

- Si la variable suivant `ifdef` n'est pas définie, les instructions comprises entre ce `ifdef` et le `endif` suivant ne sont pas incluses dans le fichier à compiler.
- On donne ci-dessous un exemple (utilisation avec gcc et l'option –D) :

```
/*  essai.c */
#include <stdio.h>
int main(int argc, char*argv[])
     printf("Debut de main\n");
#ifdef DEBUG
     printf("Fin de main\n");
#endif
     return 0;
}
```

```
$ gcc -Wall  essai.c -o essai
$ essai
Debut de main


$ gcc -Wall -DDEBUG  essai.c -o essai
$ essai
Debut de main
Fin de main
```

Le préprocesseur : #include

- Cette directive demande au préprocesseur d'ajouter le contenu d'un fichier dans le fichier courant,
  - Elle sert en général à inclure les prototypes des fonctions utilisées, mais non définies, dans un programme :

```
/*** main.c ***/
#include <stdio.h>
#include "carre.h"

int main(int argc, char*argv[]){
     double  x;
     x = carre(4);
     printf("x = %f\n", x);
     return 0;
}
```

```
/* carre.c */
double carre(float var){
   return (var * var);
}
```

```
/*** carre.h ***/
double carre (float);
```

Effet du préprocesseur

```
...
double carre (float);

int main(int argc, char*argv[])
{
 double x;
 x = carre(4);
 printf("x = %f\n", x);
 return 0;
}
```

Dépendences cycliques: En-tête des fichiers .h (1/2)

- Considérons le code suivant… Il ne compile pas. Pourquoi?

```
/*****************
      main.c
 *****************/
#include <stdio.h>
#include "types.h"
#include "functions.h"

int main (int argc, char * argv[]){
   person_t dupont = {"Dupont", 44};
   person_t dupond = {"Dupond", 44};

  char same = are_same(dupont,dupond);
  …
}

char are_same(person_t p1,
              person_t p2) {
  …
}
```

```
/*****************
      functions.h
 *****************/
#include "types.h"

char are_same(person_t p1,
              person_t p2);
```

```
/*****************
      types.h
 *****************/
typedef struct {
  char* name;
  char age;
} person_t;
```

```
./types.h:7:3: error: typedef redefinition with different types
    ('struct person_t' vs 'struct person_t')
} person_t;
  ^
./types.h:7:3: note: previous definition is here
} person_t;
  ^
```

- Solution?

Dépendences cycliques: En-tête des fichiers .h (2/2)

- En-tête classique (et assez systématique) des fichiers .h

```
/*****************
      main.c
 *****************/
#include <stdio.h>
#include "types.h"
#include "functions.h"

int main (int argc, char * argv[]){
   person_t dupont = {"Dupont", 44};
   person_t dupond = {"Dupond", 44};

  char same = are_same(dupont,dupond);
  …
}

char are_same(person_t p1,
              person_t p2) {
  …
}
```

```
/*****************
      functions.h
 *****************/
#ifndef _FUNCTIONS_H_
#define _FUNCTIONS_H_

#include "types.h"

char are_same(person_t p1,
              person_t p2);

#endif
```

```
/*****************
      types.h
 *****************/
#ifndef _TYPES_H_
#define _TYPES_H_

typedef struct {
  char* name;
  char age;
} person_t;

#endif
```

Le préprocesseur : #include

- Effet d'une exécution avec et sans inclusion du fichier `carre.h` :

Sans `#include "carre.h"` dans le fichier essai.c :

```
$gcc  main.c carre.c -o essai

$essai

x = 0.000000 (la conversion de 4 au format réel n'a pas été
   faite)
```

Avec `#include "carre.h "` dans le fichier essai.c :

```
$gcc  main.c carre.c -o essai

$essai

x = 16.000000 (conversion de 4 au format réel grâce au
   prototype)
```

Compilation : pourquoi prototyper

- Soit l'exemple :

```
/*****************
      f1.c
 *****************/
int main (argc, argv){
   int i = 0;
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}
```

```
/*****************
          f2.c
 *****************/
float carre(float var){

    return (float* var);
}
```

- 1 - L'exécution de : `gcc -Wall -c f1.c`
  donne le message suivant :
  `warning` `implicit declarations: carre, printf`

  Pourquoi ? **prototypages** mal faits dans f1.c, en effet il manque le prototype (ou signature) de carre et #include <stdio.h> pour printf.

- Cependant le fichier objet f1.o est créé et on peut fabriquer un exécutable, mais l'exécution du programme peut donner ce résultat (le résultat va dépendre des machines):

  `i = 0`

Du bon usage des fonctions

• Comme on l'a vu un programme peut ne pas donner un résultat correct parce que l'appel d'une fonction ne correspond pas à sa définition (son implémentation).

Solution :

1.Il faut toujours utiliser l'option `-Wall` du compilateur `gcc` qui permet de détecter ces erreurs,
2.Il faut prototyper les fonctions. Pour ce faire, on peut associer à chaque fichier source (type .c) un fichier (type .h) qui contient les prototypes des fonctions décrites dans ce fichier source,
3.Le programmeur qui utilise des fonctions définies dans ce fichier source doit inclure le fichier contenant les prototypes pour éviter toute utilisation erronée de ces fonctions.
4.Le fichier contenant les prototypes doit être complété par des commentaires indiquant comment utiliser les fonctions. Le fichier contenant les implémentations doit être complété par des commentaires indiquant comment les fonctions sont réalisées.

Dans les fichiers .h	Dans les fichiers .c
Prototype, Signature, Déclaration,	Implémentation, Définition,

Préprocesseur : #include

Ici, on crée le fichier `f2.h` qui contient les prototypes des fonctions décrites dans `f2.c` :

```
/*****************
      f1.c
 *****************/
#include <stdio.h>
#include "f2.h"
int main (argc, argv){
   int i= 0;
   i = carre (4);
   printf("i=%d\n",i);
return 0 ;
}
```

```
/*****************
      f2.c
 *****************/
#include "f2.h"

float carre(float var){

return (var * var);
}
```

```
/*****************
      f2.h
 *****************/
float carre(float);
```

- Lors de l'exécution de `gcc -Wall -c f1.c` :

  1. le préprocesseur inclut `f2.h` et `stdio.h` en tête de `f2.c`,
  2. le compilateur connaît donc les types des différents paramètres et applique les conversions de type dans le fichier généré,

- A l'exécution, on obtient :

```
i =  16
```

Bibliothèques et fichiers #include

Si le programme contenu dans `f1.c` est à nouveau modifié comme indiqué ci-contre :

```
#include <stdio.h>
#include "f2.h »
int main (argc, argv){
   int i;
   i = carre (4);
   printf ("i = %d\n", i);
   i = sqrt (i);
   printf ("i = %d\n", i);
   return 0 ;
}
```

- alors, le résultat des commandes :

```
gcc -c -Wall f1.c
gcc -c -Wall f2.c
gcc f1.o f2.o
```

sera :

```
warning: type mismatch
ld : fatal : Undefined Symbol sqrt
```

- pour remédier au premier problème (`warning`) il faut inclure le fichier qui contient les PROTOTYPES des fonctions mathématiques : `math.h`

- pour remédier au second problème (`ld ...`) : il faut charger la BIBLIOTHEQUE mathématique.

On indique à l'éditeur de liens d'aller chercher une bibliothèque avec l'option `-l`.
Pour la bibliothèque mathématique l'option est : `-lm`

Bibliothèques et fichiers #include

Le programme contenu dans `f1.c` est à nouveau modifié :

```
#include <stdio.h>
#include <math.h>
#include "f2.h"
int main (argc, argv){
   int i;
   i = carre (4);
   printf ("i = %d\n", i);
   i = sqrt (i);
   printf ("i = %d\n", i);
   return 0 ;
}
```

- alors, l'exécution des commandes :

```
gcc -c -Wall f1.c
gcc -c -Wall f2.c
gcc f1.o f2.o
```

Donnera le message d'erreur :

```
ld : fatal : Undefined Symbol sqrt
```

Pas de création de fichier exécutable : il manque un fichier objet.
Il faut demander le chargement de la bibliothèque mathématique :

```
gcc f1.o f2.o -lm
```

Erreurs liées à la gestion des #include et des bibliothèques : récapitulatif

	Effet
▪ pas de `#include` ▪ pas de chargement de bibliothèque	pas d'exécutable
▪ pas de `#include` ▪ chargement de bibliothèque	production d'un exécutable mais **problèmes possibles à l'exécution**
▪ avec `#include` ▪ pas de chargement de bibliothèque	pas d'exécutable

Erreurs liées à la gestion des #include et des bibliothèques : récapitulatif

Mise au point d'une application (*debugging*, débogage) :

Les programmes ne fonctionnent généralement pas correctement dès la première exécution. Lors de la mise au point d'une application volumineuse, la trace avec `printf` s'avère vite fastidieuse et inefficace.

Pour une mise au point efficace, on utilise l'option "debug" qui permet d'exécuter un programme en mode pas à pas, de visualiser le contenu des variables, leur adresse, etc.

Cette option "debug" (option `-g`) doit être indiquée à la compilation et à l'édition de liens :

```
gcc -g -c -Wall f1.c
gcc -g -c -Wall f2.c
gcc -g f1.o f2.o -lm
```

Le préprocesseur : la directive #define

Cette directive effectue des **substitutions** de chaînes de caractères, elle ne définit pas de constantes.

On peut constater son effet en utilisant une des options de gcc :
gcc –E
Dans ce cas, seul est activé le préprocesseur, il traite toutes les lignes commençant par #

Fichier `titi.c` :

```
#define MAX 100
#define double1(x) (2*x)
#define double2(x) (2*(x))
int main (argc, argv){
   long int i, j, k;
   i = MAX;
   j = 0;
   k= double1(1+2);
   k= double2(1+2);
   return 0;
}
```

Effet de `gcc -E titi.c`

```
int main (argc, argv){
   long int i , j , k;
   i = 100;
   j = 0 ;
   k = (2 * 1 + 2 );
   k = (2 *( 1 + 2 ));
   return 0 ;
}
```

Attention aux effets de bord : ici, les parenthèses qui encadrent le symbole « x » les évitent

Le préprocesseur : la directive #ifdef

Les traces sont utiles lors de la mise au point, mais peuvent être indésirables au moment d'une présentation. Pour les conserver et les exécuter seulement si nécessaire, on peut utiliser la directive **`#ifdef`**.

• Exemple :

Fichier `essai.c`

```
#include <stdio.h>
int main (void){
   int i;

   i = 4;
   #ifdef DEBUG
   printf("i=%d\n", i);
   #endif

   return 0;
}
```

Exemple d'exécutions

Exécution 1 :

```
$ gcc -Wall essai.c -o essai
$ essai
```

Exécution 2 :

```
$ gcc -Wall -DDEBUG essai.c -o essai
$ essai
i=4
```

Le préprocesseur : la directive #ifndef

Si on ne peut éviter d'inclure plusieurs fois un fichier du type « .h », le compilateur va détecter des redéfinitions. Pour les éviter, on utilise la directive : **`#ifndef`**

• Exemple :

Fichier `fonctions.h`

```
void fonc1(int i, int j);
int   fonc2(float r);
```

Fichier `fonctions.h` modifié

```
#ifndef FONC1_H
#define FONC1_H
void fonc1(int i, int j);
int   fonc2(float r);
#endif
```

Lors de le première inclusion de `fonctions.h`, `FONC1_H` n'est pas définie, donc le préprocesseur traite les lignes comprises entre la directive `#ifndef` et le `#endif` associé.

Lors des inclusions suivantes, `FONC1_H` est définie, le préprocesseur ignore les lignes comprises entre `#ifndef` et le `#endif` associé.

Le préprocesseur : la directive #ifndef

Les différences d'environnement (système+compilateur) peuvent rendre les sources non portables, pour éviter des réécritures fastidieuses, on peut utiliser **`#ifdef`** ou **`#ifndef`** combinés avec l'option `–D` de `gcc`.

Fichier `essai.c` créé, par exemple, avec devcpp sous Windows

```
int main (void) {
 …
#ifndef GCC
     system("PAUSE");
#endif
…
     return 0;
}
```

Exemple d'exécutions sous UNIX. (Utilisation de l'option **`–D`** de **`gcc`**)

Exécution 1 :

```
gcc -Wall essai.c –o essai
essai
sh: PAUSE: command not found
```

Exécution 2 :

```
gcc -Wall essai.c –DGCC –o essai
essai
```

Plan

1. *Généralités :*
   - *compilateur, assembleur, éditeur de liens*
   - *compilation séparée*
   - *directives #include, …*
   - ☞ **Espace d'adressage d'un programme**

2. *l'outil make*
   - *cible, dépendance*
   - *fichier Makefile de base*
   - *un peu plus sur le fichier Makefile*

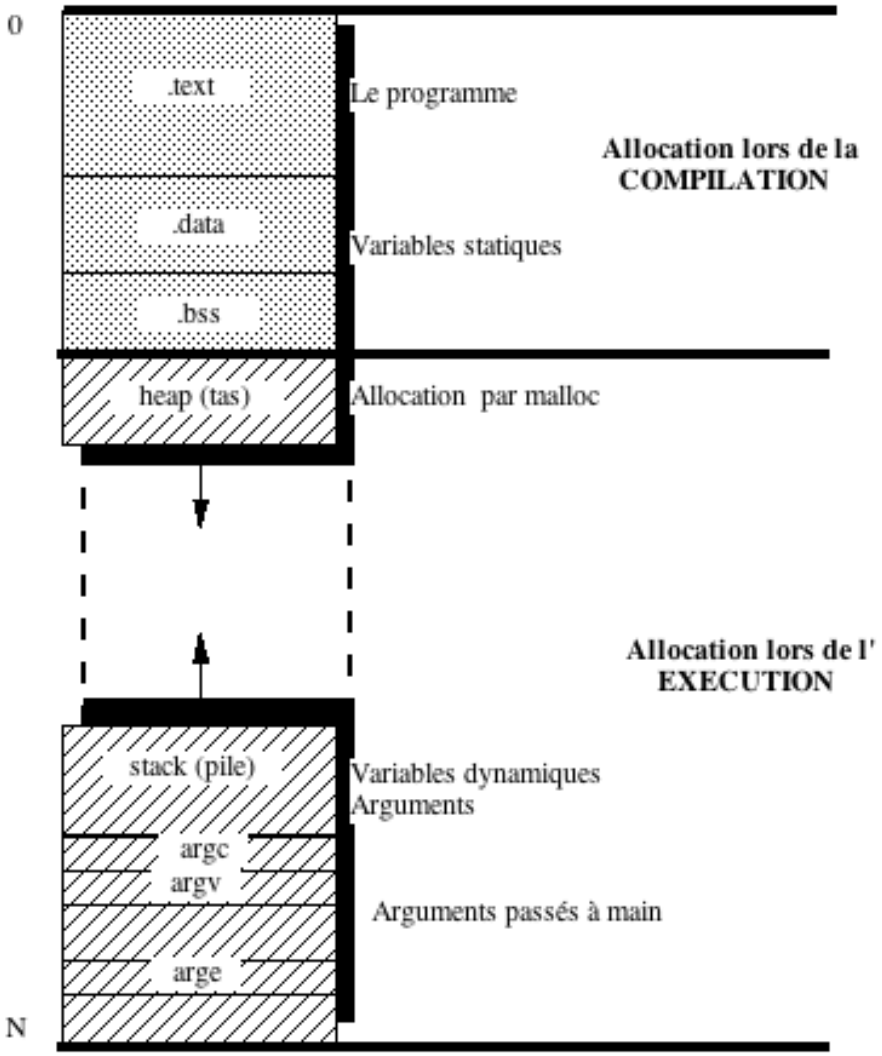
Espace d'adressage d'un programme (rappel)

Allocation de la mémoire pour un programme.

On distingue deux zones :

1. l'une dédiée aux informations allouées dynamiquement, pendant l'exécution du programme.

2.l'autre, statique, allouée à la compilation.



Visibilité (portée) et durée de vie des variables

- Variables locales (déclarées dans une fonction) :
  - par défaut : allocation sur la pile lors de **l'exécution**, la variable disparaît à la sortie de la fonction,
  - déclarée static, variable rémanente : allocation en zone « data » ou « bss » lors de la **compilation** (adresse fixe).

- Variables globales (déclarées en dehors de toute fonction) :
  - en C, par défaut, une variable déclarée globale dans un fichier est
  - visible par l'ensemble des fonctions réunies au moment de l'édition de liens,
  - static indique que la variable n'est visible que par les fonctions qui se trouvent dans le même fichier,

- Et aussi :
  - allocation dynamique sur le « tas » (heap) avec **malloc**,
  - volatile : indique au compilateur qu'il ne faut pas optimiser l'allocation (registres d'entrées-sorties, …)

Exemple de mauvaise utilisation de la mémoire

```
/**********   main *********************/
#include <stdio.h>
int main(int argc, char* argv[]){
   int *  fonc(void);
   int * buf;
   int i;

   buf = fonc();
   printf ("Dans main : \n");
   for (i=0; i<10; i++){
      printf (" buf[%d] = %d", i , buf[i]);
      if ((i+1)%5 == 0) printf ("\n");
   }
   printf ("\n");
   return 0;
}
```

```
/*******   fonc   *************/
int *  fonc(void){
   int i ,tab[10];
   printf ("Dans fonc : \n");
   for (i=0; i<10; i++){
      tab[i]=i;
      printf(" tab[%d] = %d", i , tab[i]);
       if ((i+1)%5 == 0) printf ("\n");
   }
   printf ("\n");
   return tab ;
}
```

On obtient le résultat suivant :

```
Dans fonc :
 tab[0] = 0 tab[1] = 1 tab[2] = 2 tab[3] = 3 tab[4] = 4
 tab[5] = 5 tab[6] = 6 tab[7] = 7 tab[8] = 8 tab[9] = 9

Dans main :
 buf[0] = -2062736424 buf[1] = 32767 buf[2] = 1 buf[3] = 0 buf[4] = 1606416880
 buf[5] = 32767 buf[6] = -2063526796 buf[7] = 32767 buf[8] = 8 buf[9] = 9
```

En effet, `tab` est alloué sur la pile, et perdu lors de le sortie de la fonction `fonc` .

Allocation lors de la compilation ou allocation lors de l'exécution

```
#include <stdio.h>
int main (void){
 int i;
 void fonc (void);
 for (i = 1; i < 4; i++) {
  printf ("Appel numero %d\n", i);
  fonc();
  }
 return 0;
}
```

```
void fonc(){
  static int Compteur1 = 0;
  int Compteur2 = 0;
  Compteur1 = Compteur1 + 1 ;
  Compteur2 = Compteur2 + 1 ;
  printf ("Compteur1(adr.: %p)=%d,
           Compteur2(adr.:%p)=%d\n",
           &Compteur1, Compteur1,
           &Compteur2, Compteur2);
  if (Compteur1 < 3) fonc();
}
```

On obtient le résultat suivant :

```
Appel numero 1
Compteur1(adr.: 0x100001068)=1, Compteur2(adr.:0x7fff5fbffa7c)=1
Compteur1(adr.: 0x100001068)=2, Compteur2(adr.:0x7fff5fbffa5c)=1
Compteur1(adr.: 0x100001068)=3, Compteur2(adr.:0x7fff5fbffa3c)=1
Appel numero 2
Compteur1(adr.: 0x100001068)=4, Compteur2(adr.:0x7fff5fbffa7c)=1
Appel numero 3
Compteur1(adr.: 0x100001068)=5, Compteur2(adr.:0x7fff5fbffa7c)=1
```

- La variable Compteur1 est allouée une fois pour toutes en mémoire (dans .data) au moment de la **compilation**. Elle conserve la valeur qui lui a été affectée au dernier appel à la fonction (variable rémanente).
- Compteur2 est **réallouée sur la pile à chaque appel** à la fonction, donc réinitialisée à chaque appel.

Plan

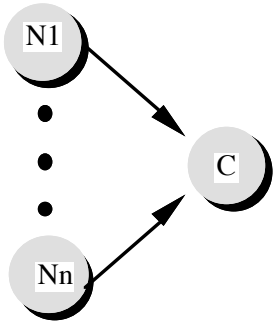
1. *Généralités :*
   - *compilateur, assembleur, éditeur de liens*
   - *compilation séparée*
   - *directives #include, …*
   - *Espace d'adressage d'un programme*

☞ **. l'outil make**
   - *cible, dépendance*
   - *fichier Makefile de base*
   - *un peu plus sur le fichier Makefile*

Graphe de dépendance

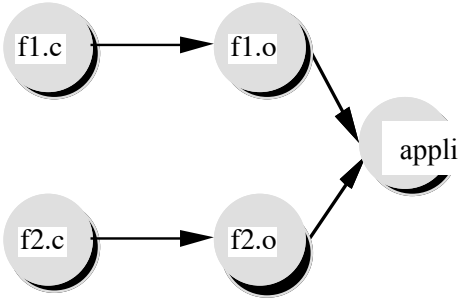
1- Rappel : graphe de dépendance :

Si le fichier A dépend du fichier B, il y aura un arc de B vers A. Par exemple, si C dépend de N1, N2, ... Nn :



2 -Traçons le graphe de dépendances de l'application précédente construite à partir des fichiers source `f1.c` et `f2.c` :

- La cible `appli` dépend de `f1.o` et `f2.o`,
- `f1.o` et `f2.o` sont eux-mêmes des cibles dépendant de`f1.c` et `f2.c`,
- L'ensemble des sommets précédant `appli` s'appelle la règle de dépendance de `appli`



Plan

1. *Généralités :*
   - *compilateur, assembleur, éditeur de liens*
   - *compilation séparée*
   - *directives #include, …*
   - *Espace d'adressage d'un programme*

2 l'outil make
   - ☞ **• cible, dépendance**
   - *fichier Makefile de base*
   - *un peu plus sur le fichier Makefile*

Cible, dépendance

- Un fichier `Makefile` contient la représentation du graphe de dépendance d'une application sous **forme de texte.**
- La notation est la suivante :

```
cible : dépendance 1 ... dépendance n
```

- Il contient aussi, à la suite de chaque règle de dépendance, les actions à entreprendre pour passer des dépendances à la cible.
- Un fichier `Makefile` est donc une succession de lignes de dépendance et de lignes d'action :

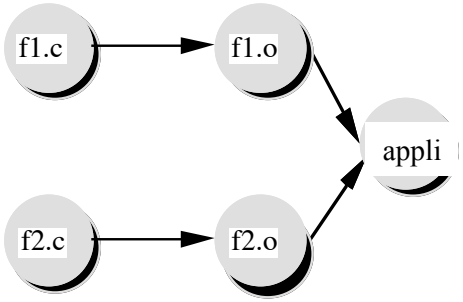
ligne de dépendance	`cible : dépendance1 … dépendancen`
ligne d'action	commande(s) à exécuter pour maintenir cible si une `dépendancei` est modifiée

Plan

1. *Généralités :*
   - *compilateur, assembleur, éditeur de liens*
   - *compilation séparée*
   - *directives #include, …*
   - *Espace d'adressage d'un programme*

2 l'outil make
   - *cible, dépendance*
   - ☞ **• fichier Makefile de base**
   - *un peu plus sur le fichier Makefile*

Exemple de fichier Makefile



- Voici un fichier Makefile qui effectue le travail représenté sur le schéma précédent. *(<TAB> indique qu'il FAUT une tabulation en début de ligne d'action) :*

```
########### appli dépend de f1.o et f2.o:
appli : f1.o f2.o
<TAB>  gcc f1.o f2.o  -o appli

############ f1.o dépend de f1.c
f1.o : f1.c
<TAB> gcc -c -Wall f1.c

########### f2.o dépend de f2.c
f2.o : f2.c
<TAB> gcc -c -Wall f2.c
```

Execution de *make*

- Après une modification de l'un des fichiers impliqués dans le Makefile, il suffit de donner la commande :

```
make appli
ou
make
```

- **make parcourt récursivement les dépendances en fonction des dates de modification** : si une cible est plus ancienne qu'une dépendance, les actions pour reconstruire la cible sont effectuées.

- make sans paramètre traite la première cible

Les variables de makefile

Exemple de variables :

Rôle	Nom	Exemple d'initialisation
option de compilation	`CFLAGS`	`CFLAGS=-c -g -Wall`
option d'éd. de liens	`LDFLAGS`	`LDFLAGS= -g -lm`
fichiers objets	`OFILES`	`OFILES= f1.o f2.o`
fichiers sources	`CFILES`	`CFILES= f1.c f2.c`
nom du compilateur	`CC`	`CC= gcc`
nom de l'éd. de liens	`LD`	`LD= gcc`
commande rm	`RM`	`RM= /bin/rm`
nom du programme	`PROG`	`PROG= appli`

la cible n'est pas forcément un fichier, ici `make clean` va nettoyer le répertoire courant :

```
...
clean :
     $(RM) $(OFILES)
...
```

Makefile et variables

Nouvelle version,
paramétrée,
du fichier Makefile :

```
########################################
BINDIR   =    /usr/local/bin
CFLAGS   =    -c -g -Wall
LDFLAGS  =    -g -lm
OFILES   =    f1.o f2.o
CC       =    $(BINDIR)/gcc
LD       =    $(BINDIR)/gcc
RM       =    /bin/rm -f
PROG     =    appli

########################################
appli: $(OFILES)
     $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)

f1.o: f1.c
     $(CC) $(CFLAGS) f1.c

f2.o: f2.c
     $(CC) $(CFLAGS) f2.c

########################################
clean:
     $(RM) $(OFILES) core
```

Plan

1. *Généralités :*
   - *compilateur, assembleur, éditeur de liens*
   - *compilation séparée*
   - *directives #include, …*
   - *Espace d'adressage d'un programme*

2 l'outil make
   - *cible, dépendance*
   - *fichier Makefile de base*
   - ☞ **• un peu plus sur le fichier Makefile**

Makefile et variables

Remarques :

- La commande **makedepend** permet d'ajouter automatiquement les dépendances dues aux fichiers d'inclusion.
- On peut utiliser des raccourcis, citons :

`$@`	La liste des cibles.
`$^`	La liste des dépendances
`$<`	La première dépendance

http://www.gnu.org/software/make/manual/make.html#Automatic-Variables

Règle de suffixe

Si on ne lui donne pas d'indications, make utilise des "règles de suffixe" pour construire la cible.

Par exemple, si on donne cette règle de suffixe pour passer d'un fichier source à un fichier objet :

```
.c.o:
	$(CC) $(CFLAGS) $<
```

Ainsi, dans le fichier makefile précédent, les règles suivantes deviennent inutiles :

```
f1.o : f1.c
	gcc -c -Wall f1.c

f2.o : f2.c
	gcc -c -Wall f2.c
```

Exemple de *makefile*

Première partie

```
#  Cree l'executable 'myprog' a partir des fichiers 'personne.c' et 'main.c'
#  Auteur: Reda Dehak

# Nom de l'executable a creer
PROG= myprog

# Fichiers source (NE PAS METTRE les .h ni les .o,
SEULEMENT les .c)
SOURCES= personne.c main.c

# Fichiers objets (ne pas modifier, sauf si l'extension n'est pas .c)
OBJETS=${SOURCES:%.c=%.o}

# Compilateur C
CC= gcc
# Options du compilateur C
# -g permet de debugger, -O optimise, -Wall affiche les erreurs
CFLAGS= -g -Wall

# Editeur de lien
LD=gcc

# Options de l'editeur de liens
LDFLAGS=

# Librairies a utiliser
# Exemple pour Qt: LDLIBS = -L/usr/local/qt/lib -lqt
LDLIBS=
```

Deuxième partie

```
############################################
# Regles de construction/destruction des .o et de l'executable
# (depend sera un fichier contenant les dependances)

all: ${PROG}

${PROG}: depend ${OBJETS}
		${LD} -o ${PROG} ${LDFLAGS} ${OBJETS} ${LDLIBS}

clean:
		$(RM) $(PROG) *.o depend

# Gestion des dependances : creation automatique des dependances en
utilisant
# l'option -MM de gcc (attention certains compilateurs n'ont pas cette option)

depend:
		${CC} -MM ${SOURCES} > depend

############################################
# Regles implicites

.SUFFIXES: .c
.c.o:
		$(CC) -c $(CFLAGS) $(INCPATH) -o $@ $<

############################################
# Inclusion du fichier des dependances
-include depend
```

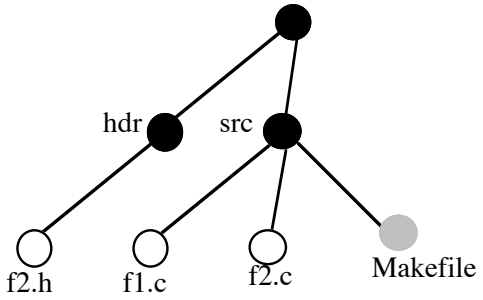
Organisation de l'application

Lorsque les fichiers constituant une application deviennent nombreux, il est souhaitable de les organiser en plusieurs répertoires.

On peut vouloir séparer les fichiers sources et objets des fichiers `include`.
On pourrait adopter l'organisation suivante :

Type de fichiers	Répertoire de stockage
`sources et objets` correspondants	`src`
`"include"`	`hdr`

Organisation



La seule modification à faire est d'indiquer à `gcc` de chercher les fichiers de type .h dans le répertoire `hdr` en utilisant l'option `-I` .
On ne modifie pas les sources.

```
CFLAGS= -c -g -Wall -I../hdr
```