conception modulaire - principe

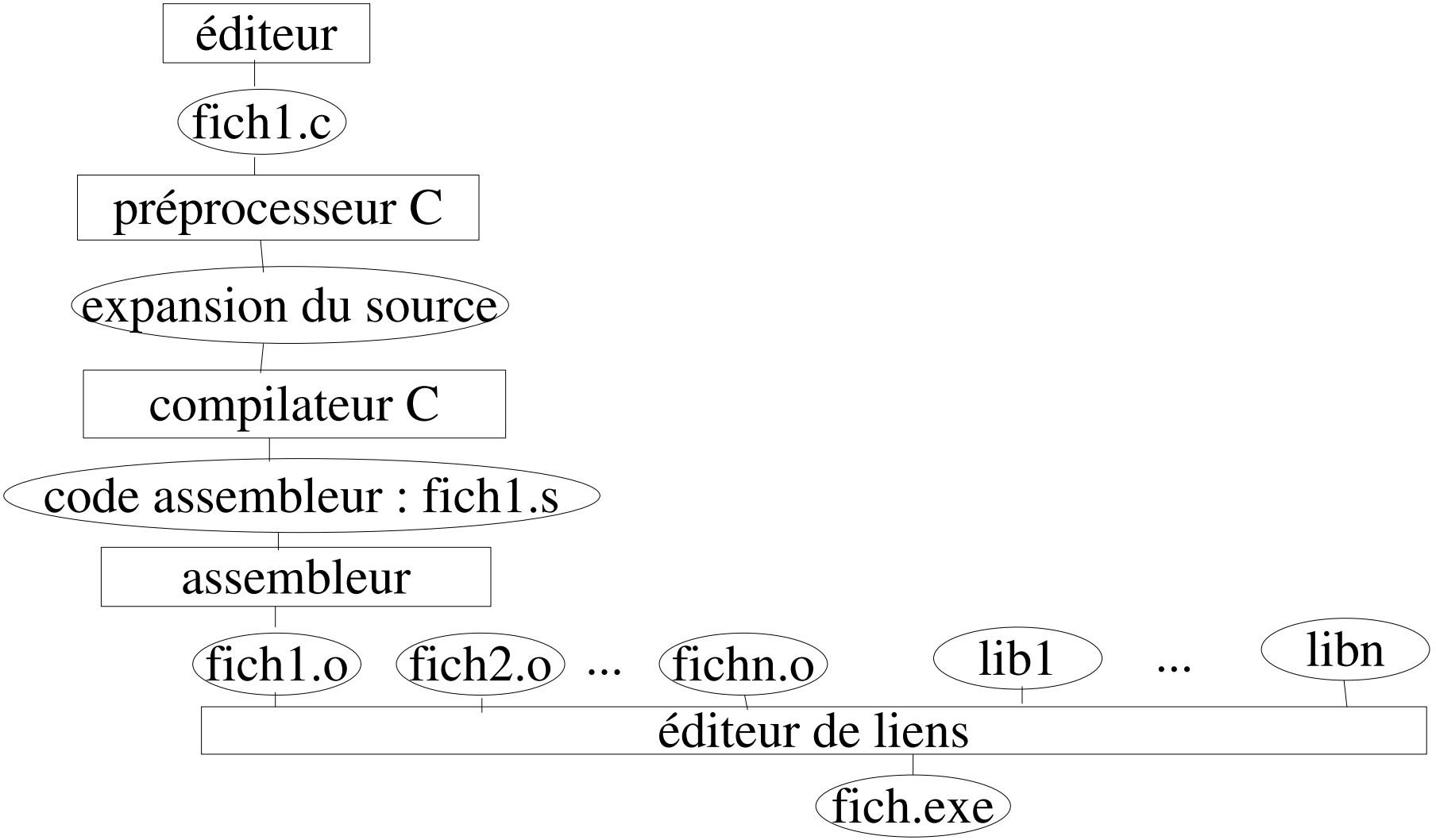
- concevoir une application comme
  - un ensemble d'unités programmatiques
  - un ensemble de sous systèmes plus simples
- but : diminuer la complexité
- une entité programmatique = un module

= un ensemble de fonctions (et éventuellement de déclarations) ayant entre elles une relation logique

ex : - module d'utilisation d'une bibliothèque

- Un module → un fichier source
- Une application → plusieurs fichiers source

étapes requises pour l'exécution



l'interface d'un module

- rendre visible (public) les fonctionnalité du module : "le quoi"
- cacher (rendre privée) l'implantation : "le comment"
- conséquence : chaque module peut disposer de ses propres noms de fonctions et de variables (locales au module)
  - prévient double emploi des noms de variable ou fonctions
  - attention particulière à l'utilisation des variables
  - encapsulation

Les classes de fonction (1)

- 2 classes de fonctions : static extern (extern par défaut)
- Fonction extern : Fonction pouvant être définie dans un module et utilisée dans un autre module, moyennant la déclaration externe correspondante
- Exemple :

Dans moduleA

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

Dans moduleB

```
extern int max(int, int);
main() { int a= 2, b=3; printf(« %d\n », max(a,b));}
```

Les classes de fonction (2)

- Fonction static : fonction utilisable uniquement dans le module où elle est définie (prévient double emploi des noms de fonction)
- Exemple :

Dans ModuleA :

```
static int fff(char c, int b) { ......}

void ggg() { char c; int b; ..... fff(c, b) .....}
```

Dans ModuleB :

```
static void fff() { ..............}

void hhh() { ..... fff() .....}
```

les classes de variables en C (1)

- une variable C possède une classe, un type, se déclare et s'alloue
- la classe :
  - recouvre deux notions selon le lieu de la déclaration
    - ✓ variable d'un fichier (en dehors de toute fonction)

      ⇒ fixe les droits d'accès (visibilité)
    - ✓ variable d'un bloc (à l'intérieur d'une fonction)

      ⇒ fixe
      - le lieu d'allocation (zone de données ou pile)
      - le moment d'allocation (compilation ou exécution)
      - sa durée de vie

les classes de variables en C (2)

- le type :
  - précise l'ensemble dans lequel la variable prend ses valeurs
  - détermine
    - ✓ taille mémoire nécessaire à l'allocation
    - ✓ règles d'utilisation
- la déclaration
  - donne au compilateur le type et la classe de la variable
  - lieu déclaration + classe définissent les droits d'accès

les classes de variables en C (3)

- l'allocation :
  - ouverture d'une zone mémoire
  - peut être faite pendant l'exécution ou à la compilation
- *Attention* : déclaration ≠ allocation
- 4 classes de variables en C : auto, static, extern, register
- classe non précisée ⇒ déterminée par l'endroit où l'identificateur est déclaré
  - à l'intérieur d'une fonction ⇒ auto
  - à l'extérieur d'une fonction ⇒ extern

la classe auto (1)

- seules variables pouvant être déclarées de classe auto : les variables d'un bloc ou les paramètres de fonctions (auto par défaut)
- allocation sur la pile : espace réservé à l'entrée du bloc (ou de la fonction) et libéré à sa sortie

  ⇒ allocation pendant l'exécution
- initialisation possible lors de la définition
- variable non initialisée ⇒ contenu = ?????
- ⇒ <u>initialiser les variables auto</u>

la classe auto (2)

- paramètres d'une fonction ou variables d'un bloc :
  - type précisé ⇒ classe auto par défaut
  - classe précisée ⇒ type int par défaut

⇒ dans une fonction :

```
        auto i = 0;
⇔         int i = 0;
⇔         auto int i = 0;
```

la classe extern (1)

- les variables d'un fichier sont par défaut de classe extern
  - ⇒utilisables dans tout le fichier (à partir de la définition → fin de fichier)
  - + accessibles dans tout le programme (⇒ à partir d'un autre fichier)
- ATTENTION : classe extern par défaut mais

  extern int i; ≠ int i;

la classe extern (2)

- pour les variables d'un fichier , extern explicite ⇒
  - pas d'allocation mémoire (réservation d'une place dans la table globale des noms du compilateur pour que l'éditeur de liens puisse unir toutes les références au même nom)
  - spécifie que zone de stockage (espace mémoire) allouée ailleurs dans le programme
  - fait référence à une variable définie ailleurs (en dehors de tout bloc, sans le mot clé extern)
  - ⇒ pas d'initialisation (faîte à la déclaration/définition)

la classe extern (3)

- extern implicite ⇒ variable globale
  - allocation mémoire à la déclaration par le compilateur + éventuellement initialisation (par le compilateur)
  - ⇒ une seule déclaration sans mot clé extern /var /prog
- ⇒ permet de définir un ensemble de variables globales partagées par différents fichiers
  - déclarer les variables une seule fois (i.e. : sans le mot clé extern) dans le fichier main.c
  - déclarations externes correspondantes dans tous les fichiers qui utilisent ces variables globales

La classe extern (4)

- Exemple :

Dans Module A :

```
int a=2;
extern void fff();
main() {  fff()  ; printf(« %d », a); ........}
```

Dans Module B :

```
extern int a;
void fff() { a=a+1; }
```

- attention : l'usage des variables globales doit être limité
  - pas de protection des variables
  - variables accessibles partout
  - ne favorise pas la modularité

la classe static (1)

- variables d'un bloc
  - espace mémoire alloué au début de l'exécution d'un programme et libéré en fin d'exécution (allocation dans zone de données)
  - stockage permanent
  - variables permanentes
  - ex : compter le nombre de fois que l'on passe dans une fonction

```
void fff() { static cpt=0; cpt++ ; ..............}
```

la classe static (2)

- variable d'un fichier :
  - accessible uniquement dans ce fichier
  - inconnu dans un autre
  - zone de stockage existe pendant toute la durée d'exécution du programme
  - si zone de stockage définie au niveau global alors étendue et visibilité restreinte au fichier source
  - initialisation des variables statiques faîte à la compilation (0 par défaut)
  - Prévient double emploi des noms de variables

La classe static (3)

- Exemple :

Dans module A :

```
static int a;
extern void fff();
void ggg() { ..... scanf(« %d », &a); .....}
main() { .....ggg();  printf(« %d », a); .....fff(); .....}
```

Dans module B :

```
static int a[MAX];
void fff() { ..... prinf(« %d », a[0]); .....}
```

la classe register

- peut être spécifiée pour les paramètres d'une fonction ou les variables d'un bloc pour une zone de stockage à taux d'utilisation très élevé
- ⇒ réduction du temps d'accès (utilisation des registres du processeur)
- problèmes :
  - nombre de registres disponibles ??
  - type de l'identificateur compatible avec taille registres ?
- impossibilité de stockage dans un registre ⇒ déclarateur considéré comme de classe auto et conservé en mémoire

Modularité : limites du C

- aucune structure syntaxique pour un module
- aucune déclaration particulière indiquant quels objets sont partagés par quels modules
- peu d'outils pour rendre fiable le partage des noms
  - seul moyen : utilisation des fichiers entête

les fichiers en-tête (1)

- seul moyen pour l'auteur d'un module A de s'assurer que les autres modules utilisent correctement les identificateurs publics :
  - écrire un fichier en-tête (A.h) contenant toutes les déclarations publiques de A
  - inclure ce fichier par la directive #include dans
    - ✓ le module qui implante ces objets publics (A.c)
    - ✓ chaque module qui les utilise
- tous ces fichiers "voient" la même définition
- toute modification de définition est reportée automatiquement dans tous les fichiers qui l'utilisent

les fichiers en-tête (2)

- que met-on dans A.h ?
  - groupes de constantes partagées par plusieurs modules
  - déclarations des variables externes définies dans A.c
  - déclarations des fonctions externes de A.c
  - types de données nécessaires dans plusieurs modules
- remarque : possibilité d'inclure un fichier en-tête dans un autre par #include