

# Production et Exécution des Programmes

Présentation: **Stéphane Lavirotte**

Auteurs: ... et al\*



(\*) Cours réalisé grâce aux documents de :  
Sacha Krakowiak, Stéphane Lavirotte, Jean-Paul Rigault

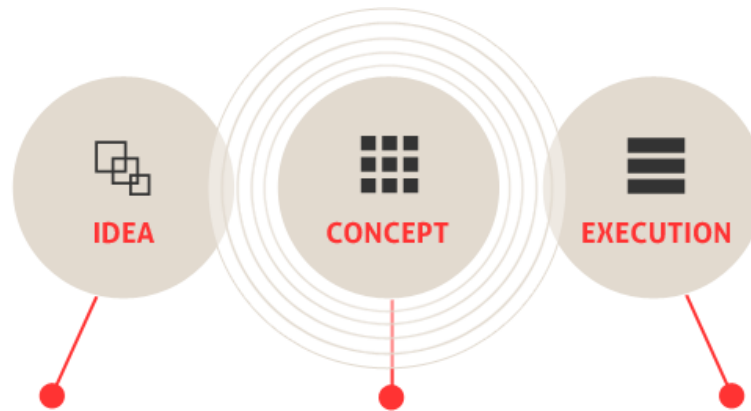
Mail: [Stephane.Lavirotte@unice.fr](mailto:Stephane.Lavirotte@unice.fr)

Web: <http://stephane.lavirotte.com/>

Université de Nice - Sophia Antipolis

# Positionnement de ce Cours

- ✓ **Cours de liaison avec les cours du 1<sup>er</sup> semestre**
  - **Programmation Procédurale**
    - Introduction au langage C (types, fonctions, préprocesseur, pointeurs, programmation modulaire...)
  - **Principe d'exécution des programmes**
    - Représentation des nombres, opérateur de base, logique et circuit combinatoire, unité arithmétique et logique, assembleur
  - **Programmation Orientée Objet**
    - Classes, objets, héritage, design patterns, machine virtuelle
  
- ✓ **Pour aller vers un point essentiel**
  - La couche entre vos programmes et la machine
  - Pourquoi, comment « un Système d'Exploitation »



# Concepts de base de l'exécution des programmes

Au niveau du processeur

# Modèle de von Neumann

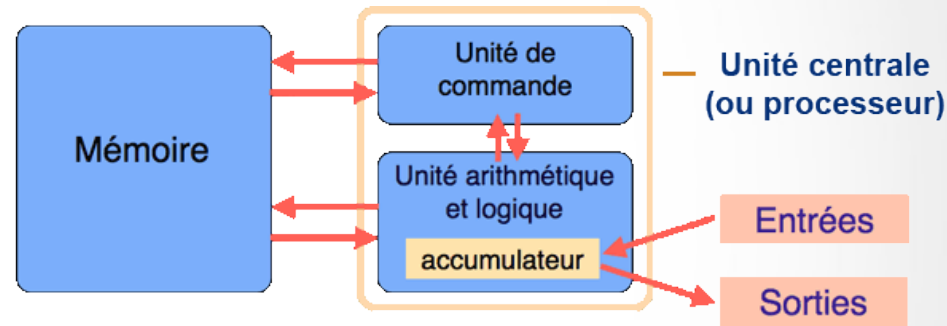
- ✓ **A la base, machine de Turing (1936)**
  - **Modèle *abstrait* du fonctionnement des appareils mécaniques de calcul**
    - Un ruban infini
    - Une tête de lecture/écriture
    - Un registre d'état
    - Une table d'actions
- ✓ **Puis, Modèle de von Neumann (étendu) (1945)**
  - **Mémoire pour programmes et données**
    - Programme non inscriptible (non automodifiable)
  - **Unité centrale avec registres de travail**
  - **Mot d'état du processeur**
    - Modes utilisateur/système, instructions privilégiées
  - **Exécution séquentielle**
    - Compteur ordinal
    - Ruptures de séquence (branchement)

# Une Architecture Novatrice

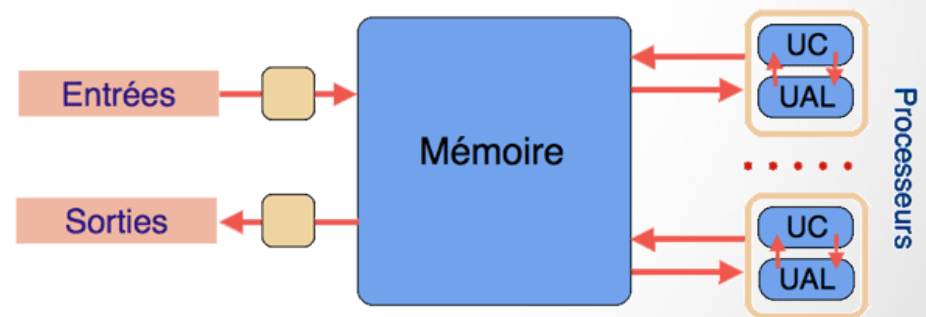
## ✓ “First Draft of a Report on EDVAC”

– Document de 101 pages décrivant:

- un schéma d'architecture de calculateur: 3 éléments
  - unité arithmétique
  - unité de commande
  - mémoire contenant programme et données
- des principes de réalisation pour ces éléments
  - les opérations arithmétiques



Modèle originel



Modèle actuel

# Modèle de von Neumann

## Concepts de base (1/2)

- ✓ **Atomicité des instructions élémentaires (?)**
- ✓ **Procédure et activité**
  - Procédure : élément structurant d'un programme
  - Activité : exécution d'une procédure
- ✓ **Programme et processus**
  - Programme : ensemble de procédures, le résultat statique d'une compilation/édition de liens (binaire exécutable)
    - Ensemble de procédures
  - Processus : l'activité dynamique résultant de l'exécution d'un programme
    - Enchaînement (séquentiel ou concurrent) d'activités

# Modèle de von Neumann

## Concepts de base (2/2)

### ✓ Contexte

- Information caractérisant l'état courant d'un processus
  - Sauvegardé lors de la suspension du processus et rechargé lors de sa reprise
- Contexte matériel
  - Mot d'état, compteur ordinal, registres de l'unité centrale
- Contexte logiciel (ou système)
  - Segments de texte (instructions), de données, de pile...
  - Attributs : identification utilisateur, droits d'accès, priorité...
  - Ressources utilisées : fichiers ouverts...

# Événements, Interruptions et Déroutements (1/2)

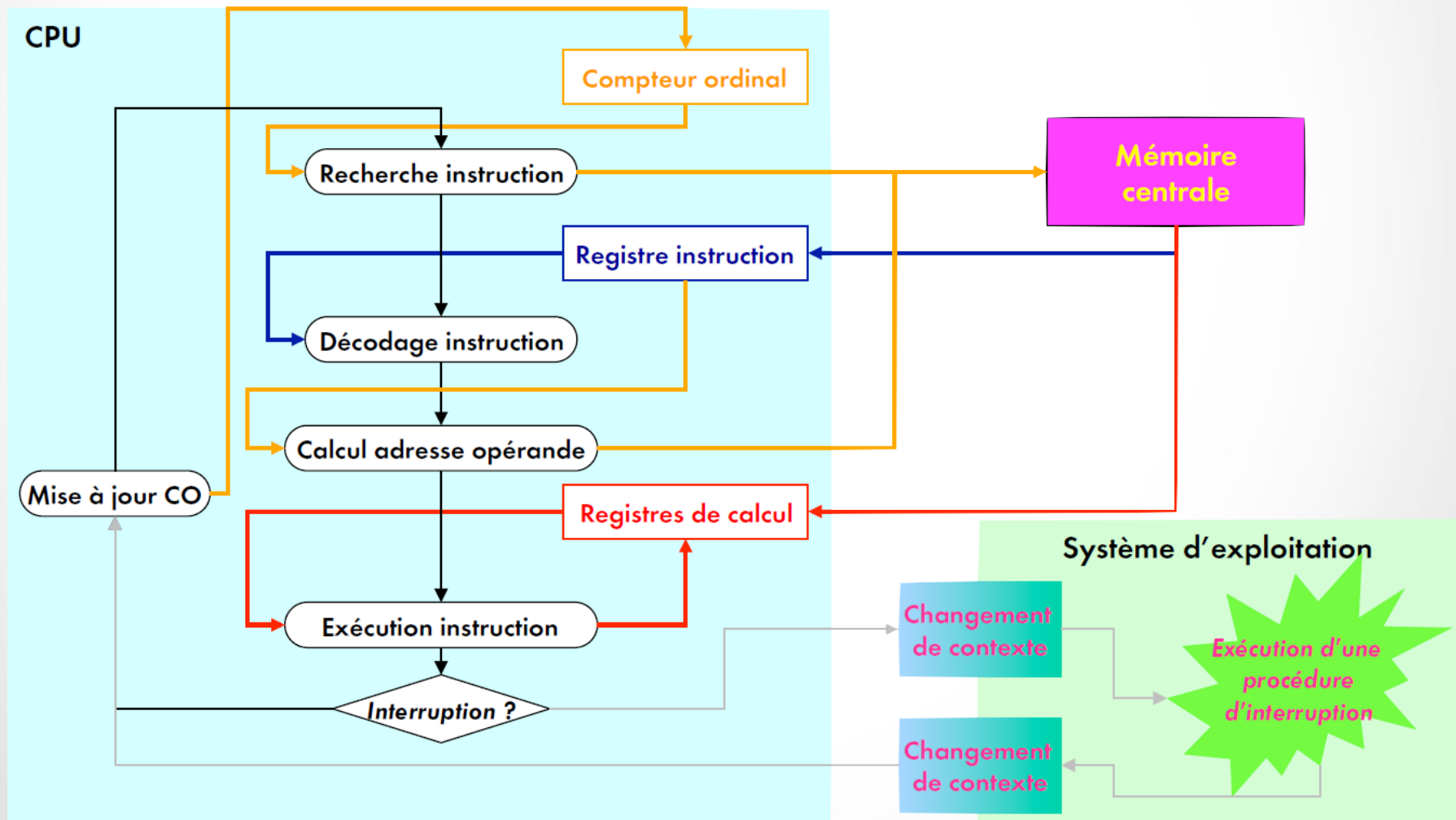
- ✓ **Événements : interruption et déroutement**
  - **Interruption** : événement asynchrone (inopiné)
    - Fin d'entrée-sortie
    - Condition « anormale » d'un processus extérieur (temps réel)
  - **Déroutement (ou exception)** : événement provoqué par l'exécution même du programme, correspondant à une situation « exceptionnelle »
    - Exception arithmétique, violation de protection mémoire...
    - Passage du mode utilisateur au mode système (TRAP)
  - **Traitement uniforme par la plupart des architectures**



# Événements, Interruptions et Déroulements (2/2)

- ✓ Événements : interruption et déroulement (suite)
  - Hiérarchie des interruptions (priorité)
  - Masquage/démasquage
  - Armement/désarmement
  - Procédure (de traitement) d'interruption
    - Déclenchement (ordonnancement) par le matériel
    - Notion de réentrance
      - Réexécution d'une procédure actuellement suspendue (par suite d'une IT)
      - Conditions nécessaires : invariance du code, séparation des données
      - La récursivité implique la réentrance

# Cycle élémentaire d'Exécution d'une Instruction





# Production et Exécution de Programmes

# Langage de Programmation (de Haut Niveau)

- ✓ **Modèle de calcul plus proche des applications**
  - Types de données et opérations
  - Mécanismes d'abstraction
  
- ✓ **Vérification et renforcement de la sécurité de programmation**
  - Vérifications statiques
  - Vérifications dynamiques
  - Utilisation des données
  
- ✓ **Nécessité d'un mécanisme de traduction ou d'interprétation en langage-machine**

# Langage de Programmation

## *Modèle de calcul*

- ✓ **Types de données et opérations**
  - Structuration des données de base de la machine sous-jacente
  - Nouveaux types de données et opérations
    - définis par le langage : `int`, `long`, `long long`, `complex`...
    - définis par l'utilisateur : `struct`, `union`, `class`
  
- ✓ **Mécanismes d'abstraction**
  - Structuration du flot de contrôle, des données
    - Sous-programmes, fonctions...
    - Modules, classes...
  - Séparation spécification/implémentation

# Langage de Programmation

## *Sécurité de Programmation*

### ✓ Vérifications statiques

- Avant toute exécution
- Lors du processus de traduction
- Exemples :
  - Analyse de type (en C/C++, Java)
  - Contrôle d'accès (en C++, Java)

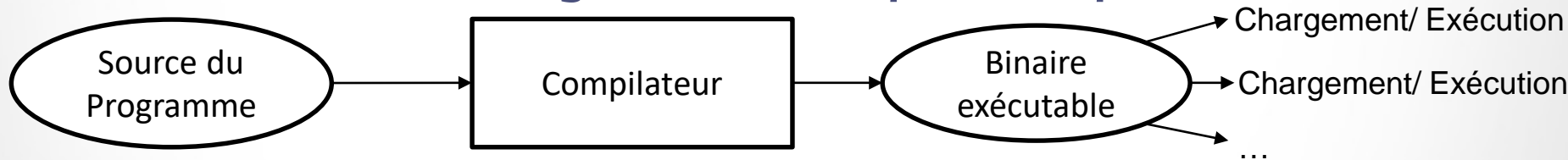
### ✓ Vérifications dynamiques

- Lors de l'exécution
- Exemple :
  - Dépassement de borne d'indice
  - Paramètres incorrects

# Compilation et Interprétation

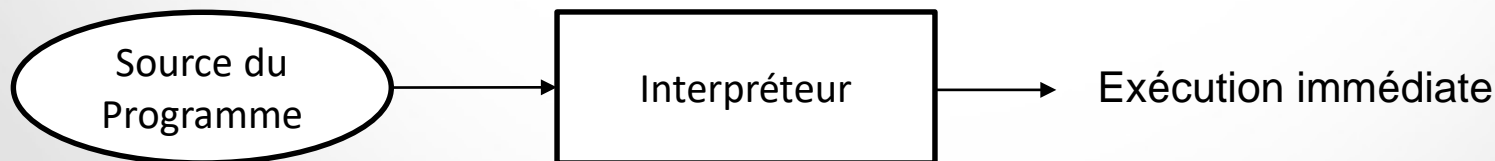
## ✓ Compilation

- Le source du programme est traduit, une fois pour toutes (?) en langage-machine
- Le résultat est un fichier binaire exécutable
- Ce fichier est chargé en mémoire pour chaque exécution



## ✓ Interprétation

- Le source du programme est traduit « à la volée », instruction
- par instruction, lors de chaque exécution



# Compilation et Interprétation

## *Avantages et Inconvénients*

### ✓ Compilation

- Possibilité de vérifications et d'optimisations statiques et globales
- Renforcement de la sécurité de programmation
- Programme figé
- Cycle de développement lourd

### ✓ Interprétation

- Le programme peut évoluer (et même se faire évoluer lui-même)
- Grande puissance d'expression
- Mise au point interactive
- Vérifications dynamiques seulement
- Détection d'erreurs à l'exécution et optimisation délicate



# Compilation et Interprétation

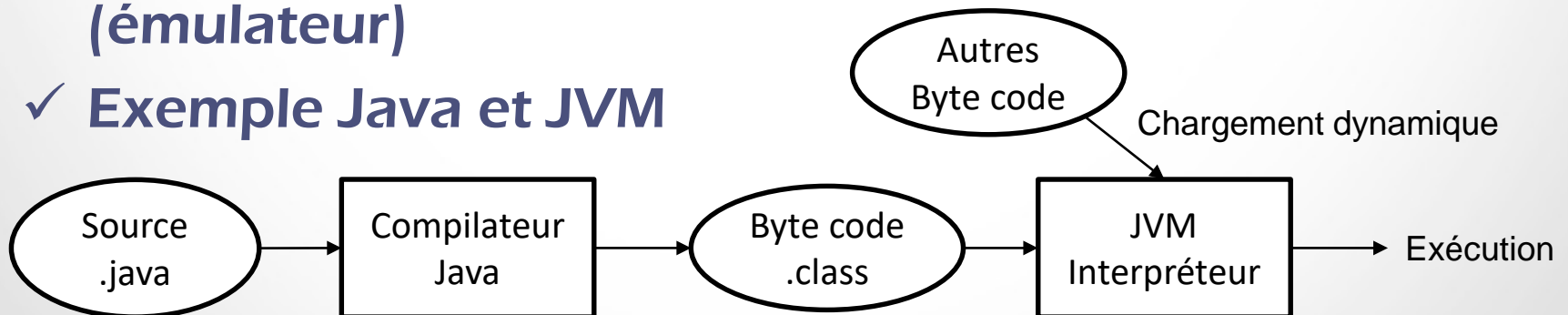
## *Combinaison des 2 Approches*

- ✓ **Langages de haut niveau et matériel**
  - Des langages comme C/C++, Ada... sont traduits en langage machine (compilation)
  - Le matériel (processeur) interprète ce langage-machine
  - Les fichiers exécutables sont dépendants du matériel utilisé
  
- ✓ **Langages de haut niveau et machine virtuelle**
  - Un langage comme Java est compilé en un langage-machine virtuel (byte-code)
  - Ce langage est lui-même interprété par un programme appelé machine virtuelle (voir plus loin)
  - Les fichiers byte-code sont indépendants du matériel

# Machine Virtuelle

## *Principe*

- ✓ Inventer une architecture de processeur adaptée à l'interprétation du langage de haut niveau
  - Registres
  - Jeux d'instruction
  - Structure de données manipulées ...
- ✓ Compiler le langage dans le jeu d'instructions de la machine
- ✓ Fournir une réalisation logicielle de cette machine (émulateur)
- ✓ Exemple Java et JVM

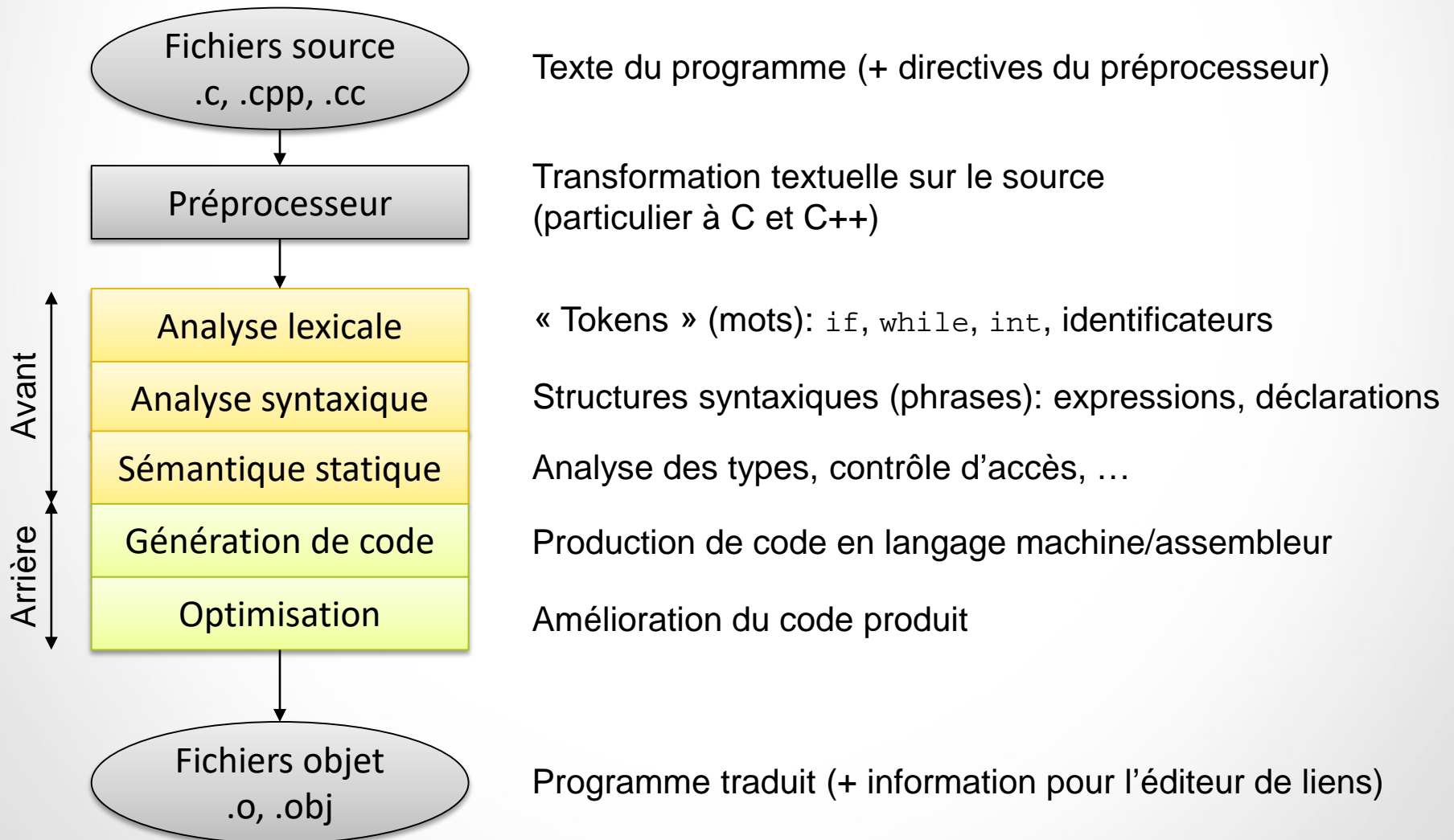


# Machine Virtuelle

## *Avantages Inconvénients*

- ✓ **Les programmes sont portables**
  - Seule la machine virtuelle doit être réécrite ...
  - et encore, elle peut elle-même être largement portable
- ✓ **Avantages de l'interprétation et de la compilation réunis**
  - Mécanismes dynamiques et vérifications statiques
- ✓ **Performance**
  - Interprétation logicielle
  - Double traduction
- ✓ **Maintenance, évolution**
  - Un changement du langage implique la mise à jour du compilateur et de la machine virtuelle

# Processus de Compilation (cas de C / C++)



# Compilation séparée ?

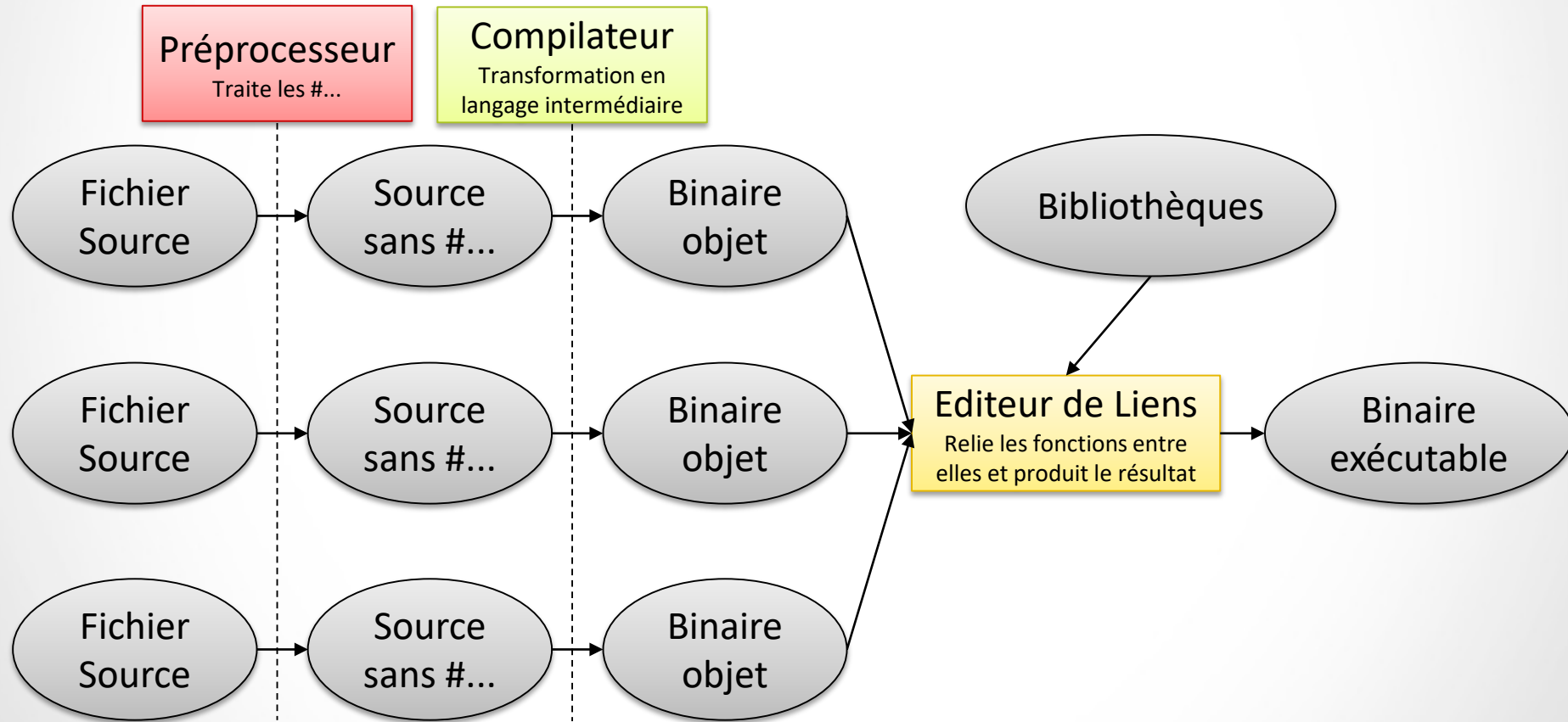
## ✓ Fichier source unique ?

- Difficile à maintenir
- Inefficace
- Besoin de bibliothèques précompilées

## ✓ Compilation séparée

- Le source du programme est découpé en unités de compilation
  - En C/C++, unité de compilation = un fichier .c (ou .cc, .cpp) et tous les fichiers .h (ou .hpp...) qu'il inclut
  - En Java, unité de compilation = un fichier .java
- Chaque unité est compilée séparément
- Un éditeur de lien colle les morceaux et produit l'exécutable
  - (sous unix/linux, l'éditeur de liens est ld)

# Etapes de la Compilation de Code Source



# Editeur de Liens

- ✓ **Rôle : lier ensemble les différentes parties du programme**
  - Fichiers-objets produits par la compilation séparée
  - Bibliothèques (fichiers-objets précompilés)
  
- ✓ **Résolution de références**
  - Utilisation d'un objet (constante, variable, fonction) dans une unité de compilation alors qu'il est défini dans une autre unité
  - Ne concerne a priori que les fonctions et les variables ou constantes statiques externes
    - Les variables allouées dynamiquement (tas ou pile) sont exclues
    - Les autres variables statiques (privées à un fichier ou à un bloc) et les fonctions statiques sont également exclues

# Exemple

fichier1.c

```
double g = 1.0;
```

**Définition** d'une variable externe g

```
int f(int);
```

**Déclaration** d'une fonction externe f

```
main() {
```

```
    int i;
```

```
    i = f(3);
```

**Utilisation** de la fonction externe f

```
}
```

fichier2.c

```
extern int g;
```

**Déclaration** d'une variable externe g

```
void f(int x) {
```

**Définition** d'une fonction externe f

```
    return 2 * g;
```

**Utilisation** de la variable externe g

```
}
```



# Résolution de Référence

- ✓ Chaque fichier-objet contient une table des symboles externes afin de faciliter le travail de l'éditeur de liens
- ✓ Cette table des symboles est en principe inutile dans le fichier binaire exécutable
  - On l'y conserve cependant pour permettre le debugging
  - La commande `strip` permet de la supprimer
- ✓ Commandes (unix/linux) de lecture des fichiers-objets (`.o`)
  - `nm` : liste des symboles
  - `objdump`, `readelf` : examine le contenu sous forme « lisible »

# Editeur de Liens et Typage

- ✓ Le typage est réalisé par le compilateur
- ✓ En cas de compilation séparée
  - chaque fichier-source doit contenir toute l'information nécessaire aux vérifications statiques
  - les fichiers-objets ne contiennent plus (a priori) d'informations liées au typage



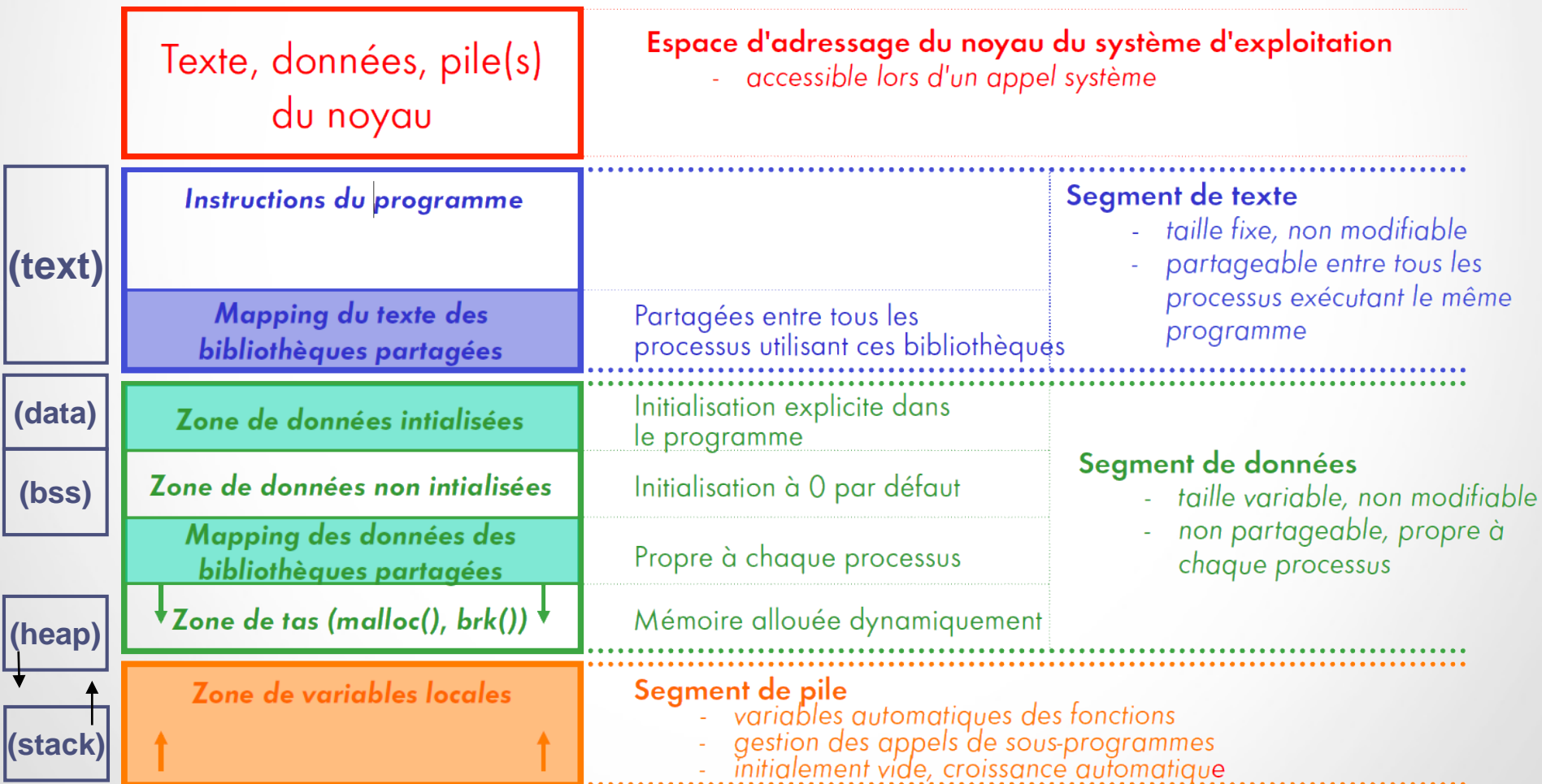


# Modèle d'Exécution

Pile d'exécution (call stack)

# Espace d'Adressage d'un Processus

## Exemple d'Unix



# Exemple Concret de Régions Mémoires

- ✓ Seules les variables globales ou statiques locales sont initialisées à 0 par défaut
- ✓ Une variable non-initialisée locale ou créée sur le tas a une valeur non-définie
- ✓ Dans quelles zones mémoires ces différentes variables sont-elles localisées ?

<code>int var1;</code>	←	bss
<code>char var2[] = "buf1";</code>	←	data
<code>main() {</code>		
<code>int var3;</code>	←	stack
<code>static int var4;</code>	←	bss
<code>static char var5[] = "buf2";</code>	←	data
<code>char * var6;</code>	←	stack
<code>var6 = malloc(512);</code>	←	heap
<code>}</code>		

# « Run Time »

- ✓ Informations nécessaires à l'implémentation de la sémantique d'exécution
- ✓ Techniques
  - Code généré par le compilateur
  - Bibliothèques (partagées ou non)
  - Machines virtuelles
- ✓ Exemples
  - Code généré
    - Structure de contrôle, appels de fonction
    - Évaluation des expressions...
  - Bibliothèques
    - Déclenchement des appels-système
    - Gestion mémoire (e.g., `malloc()`, `new`)

## Gestion Mémoire Dynamique

- ✓ **Gestion mémoire dynamique**
  - **Segment de données**
    - Allocation de mémoire dans le tas (heap)
    - Mémoire partagée entre processus
    - Données des bibliothèques dynamiques
    - Entrées-sorties « mappées »
  - **Segment de texte**
    - Code des bibliothèques dynamiques
- ✓ **Partage possible grâce à la mémoire virtuelle**
- ✓ **Mécanismes fondamentalement identiques**

# « Run Time »

## Gestion de la Pile

- ✓ Gestion de la chaîne dynamique d'appels de fonctions
- ✓ Gestion des variables locales des fonctions
- ✓ Gestion des exceptions
- ✓ Exemple de l'exécution d'un programme



## Exemple de Gestion de la Pile

```
void f() {  
    int a, b;  
    g(a, b);  
}  
void g(int x, int y) {  
    double z;  
    h(z);  
}  
void h(double a) {  
    ...  
}
```

sp →  
fp →

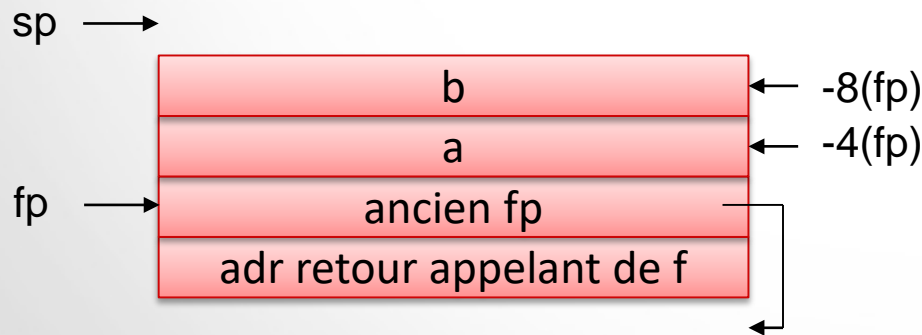
adr retour appelant de f

## Exemple de Gestion de la Pile

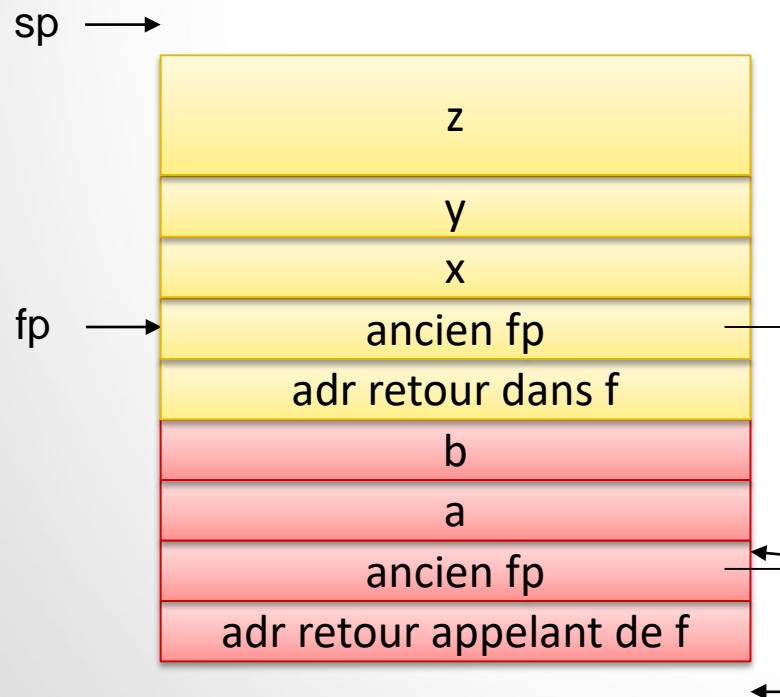
```
void f() {  
    int a, b;  
    g(a, b);  
}
```

```
void g(int x, int y) {  
    double z;  
    h(z);  
}
```

```
void h(double a) {  
    ...  
}
```



## Exemple de Gestion de la Pile

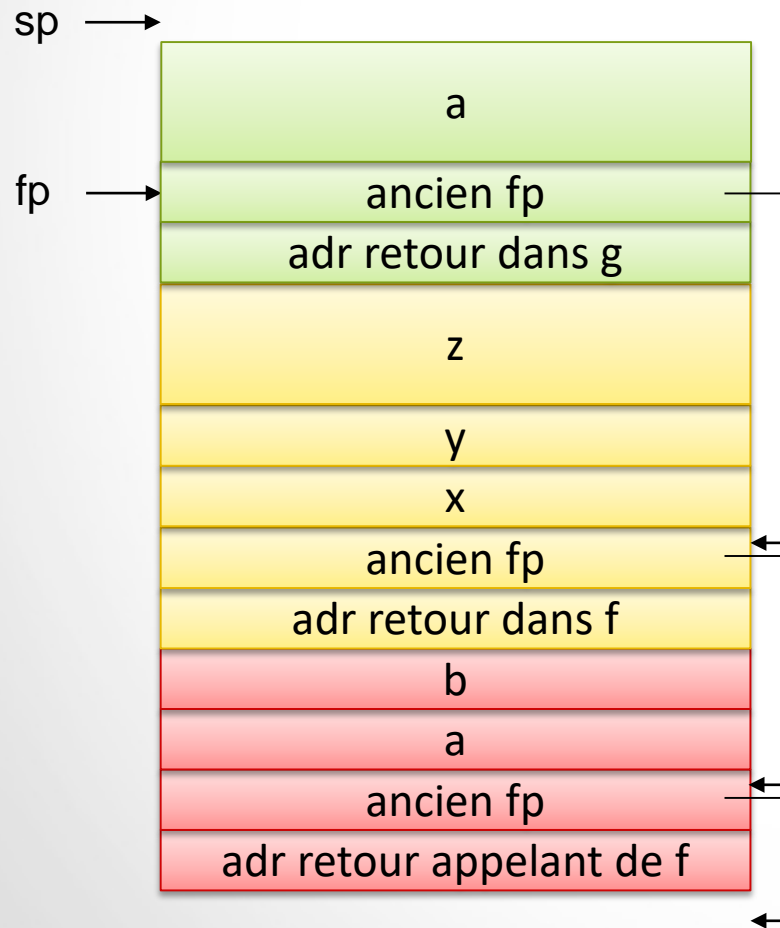


```
void f() {  
    int a, b;  
    g(a, b);  
}
```

```
void g(int x, int y) {  
    double z;  
    h(z);  
}
```

```
void h(double a) {  
    ...  
}
```

## Exemple de Gestion de la Pile



```

void f() {
    int a, b;
    g(a, b);
}

void g(int x, int y) {
    double z;
    h(z);
}

void h(double a) {
    ...
}
  
```

# Exemple Génération de Code

## <https://godbolt.org/>

### Code C

```

void h(double a) {
}

void g(int x, int y) {
    double z;
    h(z);
}

void f() {
    int a, b;
    g(a, b);
}
  
```

### Code machine

```

h(double):
    push    rbp
    mov     rbp, rsp
    movsd   QWORD PTR [rbp-8], xmm0
    nop
    pop     rbp
    ret

g(int, int):
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    mov     DWORD PTR [rbp-20], edi
    mov     DWORD PTR [rbp-24], esi
    mov     rax, QWORD PTR [rbp-8]
    mov     QWORD PTR [rbp-32], rax
    movsd   xmm0, QWORD PTR [rbp-32]
    call    h(double)
    nop
    leave
    ret

f():
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     edx, DWORD PTR [rbp-8]
    mov     eax, DWORD PTR [rbp-4]
    mov     esi, edx
    mov     edi, eax
    call    g(int, int)
    nop
    leave
    ret
  
```

# Quizz: Exemple 1

✓ **Quel est le problème dans ce code ?**

```
#include <stdio.h>
#include <stdlib.h>
int * foo(void){
    int t[] = {1, 2, 3};
    return t;
}

int main(void) {
    int * t = foo();
    for (int i = 0; i < 3; i++) {
        printf("%d ", t[i]);
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```

# Quizz: Exemple 2

✓ **Que se passe-t-il avec ce code ?**

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

```
int * foo(void){
    int v[] = {1, 2, 3};
    int *u = v;
    return u;
}
```

```
int main(void) {
    int i;
    int * t = foo();

    for (i = 0; i < 3; i++) {
        printf("%d ", t[i]);
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```

# Quizz: Exemple 3

✓ **Que se passe-t-il avec ce code ?**

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

```
int * foo(void){
    int v[] = {1, 2, 3};
    int *u = v;
    return u;
}

void bar(void) {
    int w[] = {4, 8, 16, 32};
}
```

```
int main(void) {
    int * t = foo();
    bar();

    int i;
    for (i = 0; i < 3; i++) {
        printf("%d ", t[i]);
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```





# Bibliothèques

Edition de liens statique et dynamique

# Bibliothèques (« *Library* »)

- ✓ **Fichier unique contenant un ensemble de fichiers-objets précompilés**
- ✓ **L'éditeur de liens traite les bibliothèques de manière spéciale**
  - Seul le code nécessaire au programme (les définitions d'objets utilisés par le programme) iront dans le binaire exécutable
- ✓ **Commandes unix/linux**
  - `ar` : crée une bibliothèque par regroupement de `.o`
  - `ld` : crée une bibliothèque (partagée)
  - `ranlib` : dote la bibliothèque d'un index (aide pour `ld`)
  - `ar, nm, readelf` : lisent le contenu d'une bibliothèque

# Bibliothèques Partagées

- ✓ **Eviter la duplication du code des bibliothèques courantes (e.g., libc, libstdc++)**
  - dans le fichier binaire exécutable
  - en mémoire
- ✓ **Permettre de modifier l'implémentation d'une bibliothèque (mais pas son interface !)**
  - sans refaire l'édition de liens des applications
  - sans « rebooter »
- ✓ **Permettre le chargement à la demande de bibliothèques**
- ✓ **Nommage et versions**
  - **Sous Windows**
    - Nom avec extension .dll
      - Version : cauchemar ! (amélioration avec .Net)
  - **Sous Unix/Linux (principe !)**
    - libstdc++.so.3.5

Nom de base. **extension so (shared object)**, **numéro majeur (version interface)**, **numéro mineur (implem interface)**

# Edition de Liens Statique

## ✓ Création de la bibliothèque

```
gcc options -c libobj1.c
```

```
gcc options -c libobj2.c
```

```
...
```

```
ar c libobj.a libobj1.o libobj2.o ...
```

```
ranlib libobj.a
```

## ✓ Utilisation de la bibliothèque

```
gcc options -o myprog prog1.o prog2.o libobj.a
```

ou

```
gcc options -o myprog prog1.o prog2.o -L dir -lobj
```

# Edition de Liens Dynamique

## ✓ Création de la bibliothèque

```
gcc options -fPIC -c libobj1.c
```

```
gcc options -fPIC -c libobj2.c
```

...

```
gcc -shared -Wl,soname,libobj.so.3 -o libobj.so.3.2  
libobj1.o libobj2.o ...
```

## ✓ Utilisation de la bibliothèque

```
gcc options -o myprog prog1.o prog2.o libobj.so
```

ou

```
gcc options -o myprog prog1.o prog2.o -L dir -lobj
```

(on suppose qu'il n'existe pas de libobj.a, sinon utiliser option -Bdynamic)

## ✓ *La bibliothèque devra être accessible à l'exécution (voir plus loin)*

# Chargement Dynamique de Bibliothèque

- ✓ Possibilité de chargement/déchargement à la demande d'une bibliothèque sans avoir effectué au préalable une édition de liens avec elle
- ✓ Sous Unix/Linux, utilisation d'une bibliothèque de chargement dynamique, `libdl.so` (en-tête `<dlfcn.h>`)
  - `dlopen()` ouverture d'une bibliothèque partagée
  - `dlclose()` fermeture de la bibliothèque
  - `dlsym()` recherche d'un symbole dans la bibliothèque
  - `dlerror()` traitement des erreurs
- ✓ Mécanisme similaire au chargement dynamique de classes dans Java

# Commandes de Manipulation des Bibliothèques partagées

- ✓ **Lecture du contenu des bibliothèques partagées**
  - `readelf`
- ✓ **Bibliothèques partagées utilisées par un exécutable ou une autre bibliothèque**
  - `ldd`
- ✓ **Recherche des bibliothèques partagées à l'exécution**
  - **Chargeur dynamique** : `ld.so`
  - **Variable d'environnement** `LD_LIBRARY_PATH`
  - **Commande** `ldconfig` **et fichier de configuration** `ld.so.conf` (réservé au super-utilisateur)





# Organisation du code

Format ELF



# Format des objets, Bibliothèques et Exécutables

- ✓ Ancien format `a.out` **obsolete**
- ✓ ELF : Executable and Linking Format
  - Format unique pour fichiers exécutables, objets, et bibliothèques (partagées)
  - Format en principe portable
  - Fichier exécutable (*executable file*)
    - Information nécessaire au SE pour créer l'image mémoire d'un processus
  - Fichier relogeable (*relocatable file*)
    - Fichier destiné à subir une édition de liens avec d'autres fichiers objets pour créer un exécutable ou une bibliothèque partagée
  - Fichier objet partagé (*shared object file*)
    - Information pour l'édition de liens statique ou dynamique
  - Lecture des informations d'un fichier ELF
    - `readelf`
    - bibliothèque `libelf`

# Format ELF

## 1/3

### Link view

<b>ELF header</b>	Description du reste du fichier
<b>Program header table (optional)</b>	Description des segments (utilisée au chargement)
<b>Section 1</b>	
...	
<b>Section n</b>	
<b>Section header table</b>	Description des sections (utilisée à l'édition de liens)

### Execution view

<b>ELF header</b>
<b>Program header table</b>
<b>Segment 1</b>
...
<b>Segment p</b>
<b>Section header table (optional)</b>

# Format ELF

## 2/3

### ✓ Sections

- Code, données
- Tables de chaînes de caractères
- Information d'édition de liens, de relogement
- Tables de symboles, informations de « debugging »
- Commentaires, notes...

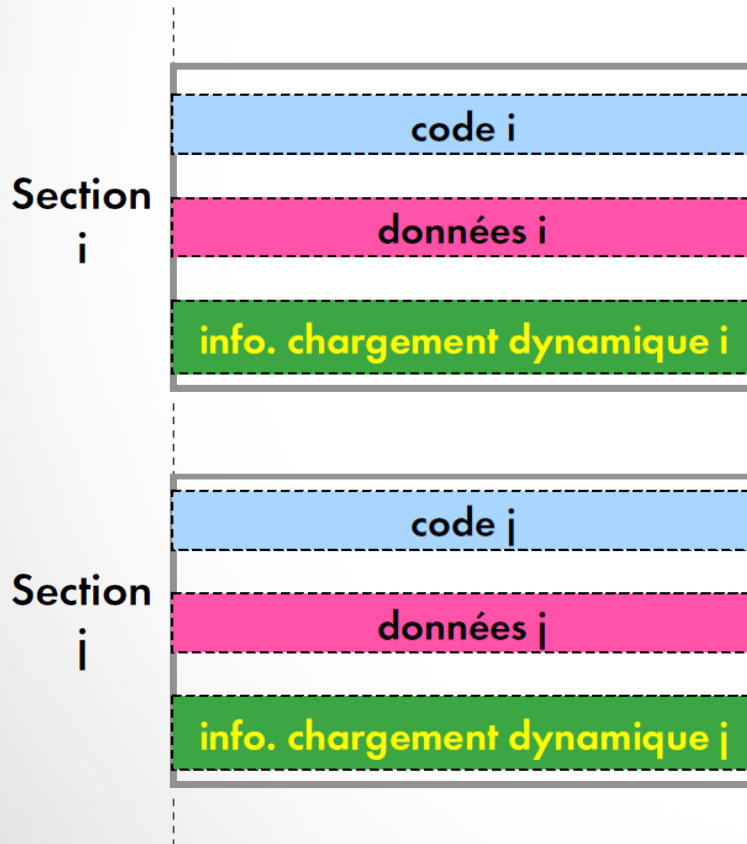
### ✓ Segments

- Groupement des informations éparpillées dans diverses sections afin de construire l'image mémoire du processus

# Format ELF

## 3/3

Link view



Execution view

