JFA - 1

2023 - 2024

CM 1

Compilation

R 2.04

Jean-François ANNE
jean-francois.anne@unicaen.fr
http://www.jfanne.fr
IUT de CAEN – Campus 3

21/07/2024

JFA 2

Sommaire

- Compilateur
- Interpréteur
- Structure d'un compilateur
- La compilation séparée
- « make » et fichier « Makefile »
- Le débogueur « gdb »
- Autres outils
- La construction de bibliothèques/librairies

R 2.04

JFA - 3

Caractéristiques des compilateurs

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

21/07/2024

Présentation

JFA 4

- Les langages de programmation permettent aux programmeurs d'écrire des programmes avec une notation plus naturelle pour l'humain que le code machine binaire exécuté par les processeurs.
- Les langages assembleurs, premiers langages de programmation développés, n'offrent qu'une simple couche d'abstraction sur le code machine sous-jacent ; ils permettent d'exprimer les instructions avec des codes symboliques et les adresses de mémoire avec des étiquettes.

```
ORG 100h
MOV AH, 09h
MOV DX, message
INT 21h
RET message
db "Bonjour le
monde !", '$'
```

↔

```
01100100100
11101110110111
10110010110001
10000011010110
```

- Évidemment, les langages de programmation ont continué à évoluer, souvent en augmentant la distance entre la notation textuelle utilisée par le programmeur et le code machine. L'avènement du génie logiciel a influencé d'avantage la conception des langages de programmation pour y introduire des notions de robustesse, d'expressivité, d'évolutivité, etc. et protéger les programmeurs d'un certain nombre d'erreurs.

JFA 5

Compilateur Vs Interpréteur

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- L'exécution d'un programme écrit avec un langage de programmation ne peut pas être directement effectuée par le processeur. Une première solution à ce problème consiste à préalablement traduire entièrement le programme source en code machine à l'aide d'un compilateur.
- La seconde solution consiste à traduire le programme source et à en exécuter le code au fur et à mesure à l'aide d'un interpréteur.
- Outre les compilateurs et les interpréteurs, de nombreux programmes doivent pouvoir lire et transformer toute sorte de fichiers.
- Par exemple :
  - Un navigateur Web doit pouvoir analyser l'information d'une page Web codée en langage de balisage HTML.
  - Un extracteur automatique de documentation, comme javadoc ou doxygen, doit pouvoir lire le code source d'un programme, en extraire les commentaires et générer la documentation des classes sous la forme d'une collection de fichiers HTML ou d'un document PDF.
  - Et plus généralement, la majorité des programmes doivent être capable de lire leurs propres fichiers de configuration.

JFA 6

Compilateur

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Compiler, signifie réunir des documents sur un même sujet à partir de diverses sources.

En informatique :

- Un compilateur est un programme qui accepte du code source en entrée et qui produit un code de destination, pour un processeur donné, équivalent en fonctionnement en sortie, tel qu'illustré à la figure 1.

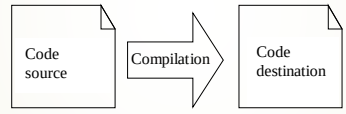


FIGURE 1 – Compilateur

Exemple :

https://info.sio2.be/infobase/6/images/compilation.jpg

JFA 7

Compilateur traditionnel

R 2.04

- Le compilateur de langage de programmation traditionnel est celui qui accepte, un ou plusieurs fichiers de code source, constituant un programme en entrée. Il produit en sortie, un ou plusieurs fichiers binaires formés d'instructions pour un type de processeur donné. Ces fichiers binaires peuvent ensuite être exécutés sur l'ordinateur correspondant. Ceci est illustré à la figure 2.
- gcc est par exemple un compilateur pour le langage C sous Linux. Lorsque le programmeur lance la commande

gcc main.c –o main

On obtient un programme exécutable « main » , pour le type de processeur sur lequel on est, à partir du code source « main.c » écrit en langage C.

Le compilateur lit le fichier main.c ainsi que les autres fichiers sources référencés par celui-ci (même indirectement), et produit un fichier binaire main pouvant être exécuté directement sur la machine.

JFA 8

Compilateur traditionnel

R 2.04

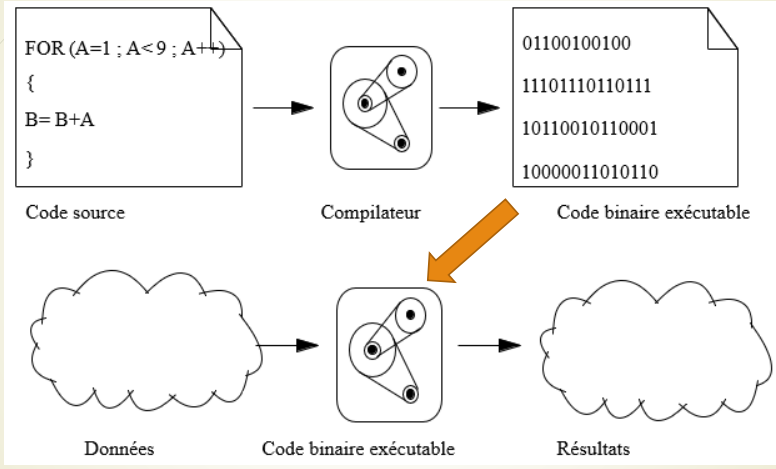


FIGURE 2 – Compilateur traditionnel et exécution.

JFA 9

Etapes de compilation

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Le système de compilation du langage C, est en réalité plus complexe que ne le laisse imaginer la commande précédente. Chaque fichier .c d'un programme C, est en premier, donné en entrée, à un préprocesseur qui traite les directives de pré-traitement contenues dans le fichier source, telles que #include et #define, et produit du code C sans directives en sortie.
- Ce code est ensuite envoyé au compilateur C qui le traduit en code assembleur.
- Par la suite, le code assembleur est donné en entrée à un assembleur qui le traduit en code binaire et le stocke dans un fichier objet .o.
- Finalement, l'éditeur de liens combine les fichiers *.o et produit un fichier exécutable dans un format approprié, tel que EXE sous WINDOWS et ELF sous LINUX.
- Il suffit d'ajouter l'option -v à la ligne de commande de gcc pour visualiser ces différentes étapes illustrées à la figure 3.

JFA 10

Etapes de compilation

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

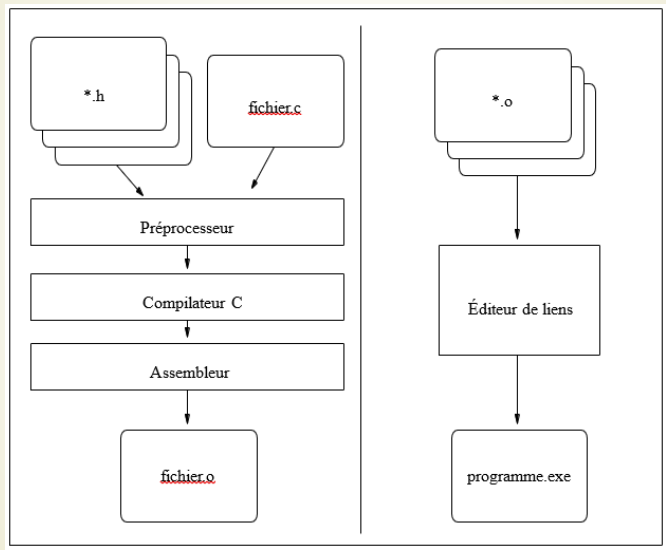


FIGURE 3 – Compilation d'un programme C.

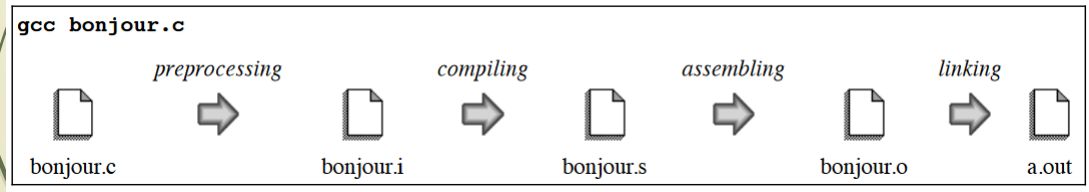
JFA 11

Etapes de compilation

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Un utilisateur peut choisir de n'effectuer que les premières phases du système de compilation :
  - L'option -E de gcc n'invoque que le préprocesseur et génère un gros fichier .c pur (gros car chaque directive #include est remplacée par le contenu complet du fichier inclus),
  - L'option -S invoque le préprocesseur et le compilateur C et génère un fichier assembleur (.s)
  - et l'option -c invoque toutes les phases sauf l'édition de lien et génère un fichier en langage machine (.o) non exécutable en soit.

Il est intéressant de noter que le préprocesseur et l'assembleur sont des compilateurs ; ils traduisent du code source en code destination équivalent.



http://perso.univ-lemans.fr/~cpiau/L2-Outils%20de%20Programmation/C1_L2SPI-Outils%20Avances%20de%20Prog-Compilation%20-etu.pdf

JFA 12

Conclusions

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Dans gcc, le préprocesseur et l'assembleur sont même des programmes séparés (respectivement cpp et as) qui sont utilisés par le programme gcc mais qui pourraient tout aussi bien être exécutés directement par l'utilisateur. Le compilateur C ne constitue qu'un élément du système de compilation d'un programme C. Il traduit du code C pur en code assembleur.
- Un autre exemple de compilateur est le compilateur javac du langage JAVA. Le système de compilation d'un programme Java est bien plus simple que celui d'un programme C.
- Il n'y a ni préprocesseur, ni assembleur, ni éditeur de liens. Les fichiers *.class générés par javac sont des fichiers binaires formés d'instructions pour un processeur virtuel. L'exécution d'un programme JAVA compilé n'est donc pas effectuée directement par le processeur de l'ordinateur, mais plutôt par une machine virtuelle, un programme qui réalise l'exécution des instructions du processeur virtuel.

JFA 13

Compilation source à source

IUT GRAND OUEST NORMANDIE iNFO

R 2.04

- Un compilateur source à source est un programme qui accepte du code source en entrée et qui génère du code source équivalent (dans le même langage ou dans un autre) en sortie. Ce type de compilateur est d'une grande utilité pour les programmeurs.
- Un exemple type est un compilateur qui lit une description de schéma de données en XML et qui produit un ensemble de classes (fichiers sources) en langage JAVA pour encapsuler l'accès à une base de données avec un modèle objet.
- Un autre exemple est le préprocesseur C qui transforme du code C avec directives de pré-traitement en code C sans directive de pré-traitement.
- Il existe même des compilateurs optimisants qui font de la compilation source à source. Le compilateur SMARTEIFFEL, par exemple, traduit les programmes EIFFEL en code source C portable (ou en code-octet JAVA). Ainsi, SMARTEIFFEL peut cibler tous les systèmes qui possèdent un compilateur C sans nécessiter le développement de générateurs de code distincts pour les divers processeurs et systèmes d'exploitation. C'est le système de compilation C, sur chaque système, qui prend en charge les lourdes tâches de génération de code machine selon les exigences architecturales et d'édition de liens.

JFA 14

Compilation source à source

IUT GRAND OUEST NORMANDIE iNFO

R 2.04

- Un compilateur source à source est un extraordinaire outil de génie logiciel. Les avantages d'utiliser un tel compilateur plutôt que d'écrire un code répétitif à la main sont nombreux :

  1. Il est très facile de faire des erreurs dans l'écriture de code répétitif et, pire, il est difficile de trouver ces erreurs à la lecture du code.
  2. En cas d'évolution ou de correction d'erreur, il suffit d'intervenir à un seul endroit dans le compilateur source à source plutôt qu'à chaque apparition du patron dans le code. Ceci peut réduire considérablement le temps de développement requis pour appliquer et tester l'évolution ou la correction.
  3. Finalement, il est généralement plus stimulant de développer un petit compilateur à l'aide d'outils que d'écrire du code répétitif.
- Lors de la conception d'un nouveau langage de programmation, la compilation source à source peut servir à créer rapidement un compilateur prototype ciblant un langage de programmation existant.

JFA 15

Décompilateur

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Un décompilateur est un compilateur particulier qui fait le travail inverse d'un compilateur. Il prend en entrée un ou plusieurs fichiers binaires et produit en sortie un programme équivalent dans le langage de programmation original.
- En général le résultat de la décompilation n'est pas identique au programme original. En effet la compilation traditionnelle n'est pas réversible car lors de la compilation traditionnelle de l'information non nécessaire au fonctionnement du programme est perdue. Par exemple les commentaires et les noms des variables locales ont disparus. De plus, plus le compilateur est optimisant, plus celui-ci risque d'avoir transformé des morceaux du programme en des équivalents plus efficaces. Par exemple certaines structures de contrôles complexes peuvent avoir été transformées en des structures plus simples (de la même façon qu'une boucle for peut être transformée en une boucle while, voire juste en quelques goto).
- Les décompilateurs sont généralement utilisés dans un cadre de rétroingénierie pour étudier ou comprendre un programme dont le code source est indisponible. L'objectif peut être, par exemple, de trouver ou d'analyser des bogues, de réaliser un portage sur une architecture différente, de vérifier son fonctionnement ou d'en analyser la sécurité comme, par exemple, chercher la présence de portes dérobées (backdoors).

JFA 16

Décompilateur

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

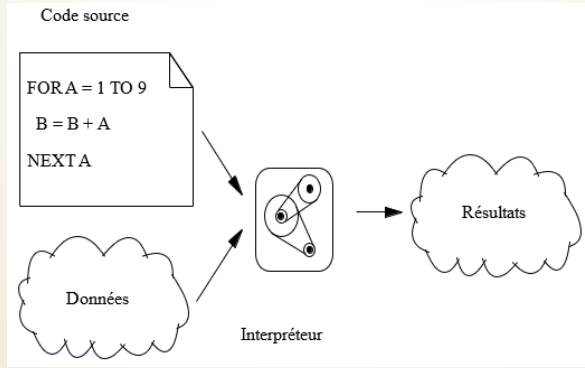
- L'option --disassemble de la commande objdump Unix permet de transformer un programme machine en code assembleur (on parle alors de désassembleur). Plusieurs décompilateurs existent pour d'autres langages comme le C ou le Java.
- En pratique, la décompilation est un problème difficile car en plus du travail habituel d'un compilateur, le décompilateur doit trouver le moyen de retrouver (ou de fabriquer) un maximum d'information non nécessaire au fonctionnement du programmes mais nécessaire à la compréhension du programme par un humain.
- Au niveau légal, la plupart des licences de logiciels propriétaires interdisent la décompilation. Dans certains pays, elle est illégale ou seulement autorisée à des fins d'interopérabilité. Certains logiciels sont même parfois compilés avec des compilateurs obfuscateurs de code dont l'objectif est de rendre plus difficile la décompilation du code produit : un décompilateur arrivera à produire un programme équivalent dans le langage d'origine mais il sera relativement incompréhensible pour un humain.

JFA 17

Interpréteur

R 2.04

- Un interpréteur est un programme qui accepte du code source en entrée et qui exécute ce qui est spécifié dans le code source, tel qu'illustré à la figure 4.
- L'interpréteur de langage de programmation traditionnel fonctionne en mode interactif où le programmeur écrit le code source et l'interpréteur exécute ce code dès qu'il est entré.



JFA 18

Interpréteur

R 2.04

- Plusieurs langages de programmation sont interprétés. Les programmes RUBY, par exemple, sont lus et directement exécutés par l'interpréteur de ce langage. Typiquement, l'exécution d'un programme RUBY est lancée avec la commande ruby fichier.rb. Le programme ruby est l'interpréteur de ce langage. Il lit le fichier fichier.rb et exécute le programme sur le champ (immédiatement)[2] .
- La plupart des interpréteurs peuvent être aussi interactifs. C'est-à-dire qu'ils exécutent le programme au fur et à mesure que le programmeur le saisit et non pas d'un coup à la fin de la lecture. Cela permet au programmeur de choisir l'instruction suivante à écrire en fonction du résultat (ou de l'erreur) qui a produit l'instruction précédente. Le mode interactif d'un interpréteur est un outil pratique pour apprendre à utiliser un nouveau langage ou pour tester le comportement de petites séquences d'instructions.

[2] Sur un système de type UNIX correctement configuré, la commande ./fichier.rb lancera automatiquement l'interpréteur RUBY.

JFA 19

Interpréteur

IUT GRAND OUEST NORMANDIE iNFO

R 2.04

- Ils sont également réflexifs. C'est-à-dire qu'ils peuvent dynamiquement manipuler et exécuter des morceaux de code source. Ainsi, dans de nombreux langages, l'instruction eval(s) permet d'évaluer la chaîne de caractères s en tant que code source sachant que la valeur de s peut avoir été calculée dynamiquement.
- Habituellement les interprètes de commandes des systèmes d'exploitation (Shells) sont considérées comme des interpréteurs à part entière. Ils sont généralement utilisés en mode interactif mais il est habituel d'écrire ou d'exécuter des programmes pour ces interpréteurs (script Shells).

JFA 20

Structure d'un compilateur

IUT GRAND OUEST NORMANDIE iNFO

R 2.04

- Pour accomplir son travail, un compilateur doit :
  - lire le code source,
  - vérifier sa validité et, enfin,
  - générer le code de destination lorsqu'il n'y a pas d'erreur ou afficher les erreurs détectées.

La figure 5 donne une vue générale de la structure interne d'un compilateur.

- Lorsque le code source est textuel, le compilateur doit analyser la séquence des caractères du code source pour extraire l'information structurée qu'elle contient[3]. L'utilisation de la théorie des langages, pour s'attaquer à ce problème, a été amenée à diviser cette tâche en deux étapes :
  - L'analyse lexicale et
  - L'analyse syntaxique.
- L'analyse de l'information structurée (reconstituée ou non) permet de détecter des erreurs, s'il y a lieu, et de calculer des informations dérivées nécessaires à la génération de code. C'est ce qu'on appelle l'analyse sémantique[4].

[3] Les compilateurs qui acceptent du code source déjà structuré sont dispensés du travail de reconstitution.

[4] La sémantique du code, c'est le sens ou la signification du code.

JFA 21

Structure d'un compilateur

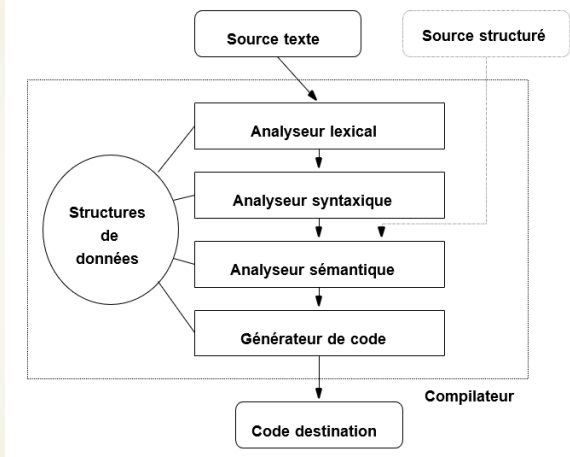


FIGURE 5 – Structure d'un compilateur

Finalement, l'étape de génération de code se sert des informations extraites et calculées précédemment pour émettre du code destination équivalent au code source lu.

JFA 22

Analyse lexicale

- L'analyse lexicale consiste à diviser la longue séquence de caractères du fichier source en ***jetons*** (ou ***token***). Un jeton est une petite sous-séquence significative de caractères. Cela peut être un mot clé, une ponctuation, une espace, un identifiant, un nombre ou toute autre entité significative.
- Généralement, les langages utilisent deux types de jetons : les jetons utiles à l'analyse syntaxique (tels les identifiants et les nombres) et les jetons ignorés (tels les espaces et les commentaires). Les jetons ignorés sont éliminés par l'analyseur lexical. Seule la séquence des jetons utiles est envoyée à l'analyseur syntaxique.
- Par exemple, la séquence d'octets :

```
« for i = 1 to 10 loop »
```

peut produire la séquence de jetons suivante :

```
mot clé « for »,
identifiant « i »,
symbole d'égalité,
entier littéral 1,
mot clé « to »,
entier littéral 10, etc…
```

JFA 23

Analyse syntaxique

R 2.04

- Le rôle de l'analyse syntaxique est de reconstituer la structure syntaxique de la séquence des jetons utiles reçue de l'analyseur lexical.
- Le plus simple est d'expliquer l'analyse syntaxique à l'aide d'une analogie. La reconstitution de la structure syntaxique d'une longue séquence de mots en langue française identifie le verbe et les autres éléments des phrases, l'organisation des phrases en paragraphes, l'organisation des paragraphes en sections, l'organisation des sections en chapitres et l'organisation des chapitres en livre.
- La structure obtenue par analyse syntaxique est naturellement arborescente et est appelée *arbre syntaxique ;* elle est envoyée à l'analyseur sémantique.

La figure 6 donne l'exemple de l'analyse syntaxique de la phrase « le chat dort. ».

JFA 24

Analyse syntaxique

R 2.04

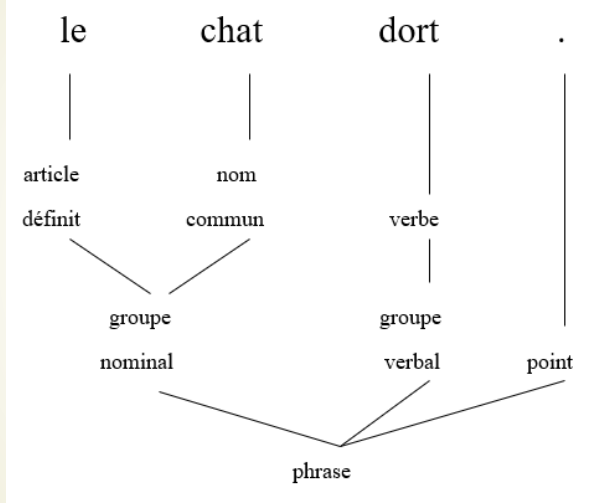


FIGURE 6 – Arbre syntaxique de la phrase « le chat dort ». Les feuilles sont des mots ou des symboles de ponctuation, les autres nœuds sont des concepts grammaticaux.

JFA 25

Analyse sémantique

IUT GRAND OUEST NORMANDIE iNFO

R 2.04

- L'analyse sémantique consiste à vérifier la cohérence de l'information structurée obtenue de l'analyseur syntaxique (ou directement du code source structuré). L'analyse sémantique peut également inclure le calcul d'informations sémantiques additionnelles (tel que le calcul de la quantité de mémoire nécessaire à stocker une variable).
- Voici un exemple qui illustre une phrase syntaxiquement correcte mais sémantiquement erronée : « Le crayon mange une pomme ». Syntaxiquement, tout va bien ; on trouve un groupe sujet, un verbe et un complément. Sur le plan sémantique, il y a un problème : un crayon, ça ne mange pas !
- Par exemple, lors de la compilation d'un programme, c'est l'analyse sémantique qui détermine qu'une affectation est illégale car les types ne correspondent pas ou qui détecte qu'une variable locale est utilisée avant d'être initialisée.

JFA 26

Génération de code

IUT GRAND OUEST NORMANDIE iNFO

R 2.04

- Le générateur de code se sert des informations et des structures obtenues et créées par les analyseurs lexical, syntaxique et sémantique pour émettre le code destination.
- Quand le langage à compiler est de haut niveau, la génération de code peut se faire en plusieurs étapes. Chacune traduisant le programme vers un langage de moindre niveau d'abstraction. C'est particulièrement vrai pour les compilateurs optimisants dans lesquels chacune des optimisations est généralement spécifique à une étape particulière.
- D'une certaine façon, dans le cadre d'un système de compilation C, les phases de pré-traitement et d'assemblage peuvent être considérée comme des étapes intermédiaires entre le langage de haut niveau, qui est le C avec des directives de précompilation, et le langage de bas niveau qui est le code machine du processeur. Toutefois, même l'étape proprement dite de compilation C, qui traduit du code C pur en assembleur, se fait à l'interne en utilisant des langages intermédiaires.

JFA 27

Structure d'un interpréteur

R 2.04

- La structure interne d'un interpréteur est presque identique à la structure d'un compilateur expliquée précédemment. La seule différence, c'est que le générateur de code est remplacé par un exécuteur de code comme le montre la figure 7.
- L'interpréteur est réflexif si l'exécuteur de code peut interpréter directement du code source nouveau, qui peut être, soit extrait des données du programme, soit dynamiquement construit sous une forme manipulable par le langage interprété, comme une chaîne de caractères par exemple. Ceci est représenté sur la figure par la flèche entre l'exécuteur de code et l'analyseur lexical.
- Dans le cadre des interpréteurs interactifs, la structure principale est une boucle, appelée read-eval-print, où l'exécution d'une instruction est suivie par la lecture, l'analyse et l'exécution de l'instruction suivante.

JFA 28

Structure d'un interpréteur

R 2.04

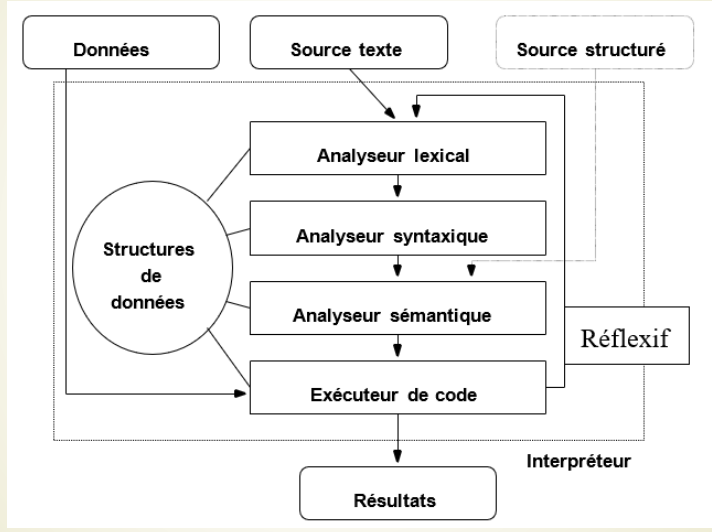


FIGURE 7 – Structure d'un interpréteur

JFA 29

Structure d'un interpréteur

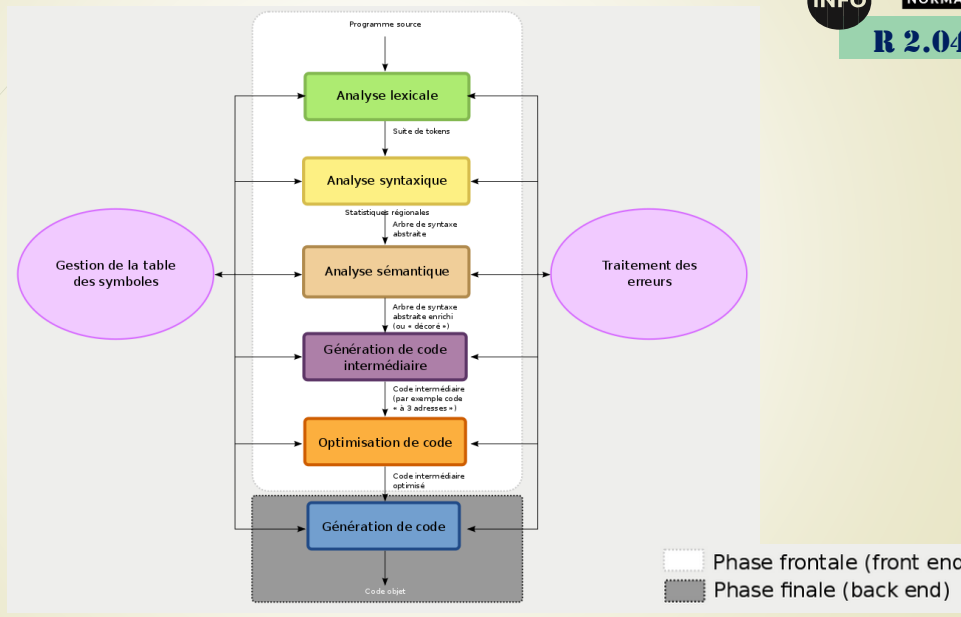
IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Tout comme le générateur de code dans le cadre d'un compilateur, l'exécuteur de code se sert des informations et des structures obtenues et créées par les analyseurs lexical, syntaxique et sémantique pour réaliser les tâches décrites dans le code source. Toutefois, l'interpréteur doit également maintenir des structures de données additionnelles permettant de représenter l'état d'exécution du programme.
- Dans le cas des interpréteur mixtes, l'interprétation peut se faire non pas sur le code source originel mais sur une représentation intermédiaire. C'est par exemple le cas des interpréteurs PERL et PYTHON qui compilent à l'interne le code source en un code binaire puis interprètent ce code binaire. Toutefois, ce travail de compilation est à recommencer à chaque nouvelle exécution d'un programme (sauf si l'interpréteur peut sérialiser ce code binaire et le réutiliser lors des exécutions suivantes, ce que propose l'interpréteur PYTHON avec les fichiers *.pyc).
- De plus, un interpréteur mixte, disposant d'un compilateur juste-à-temps, peut choisir de compiler des morceaux de code source (ou plus vraisemblablement de code binaire intermédiaire) en code binaire machine directement exécutables par le processeur physique.

JFA 30

Synthèse : Chaine de compilation

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04



JFA - 31

Les différentes étapes du compilateur C

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

JFA 32

Les différentes étapes du compilateur C : cc ou gcc (Gnu Compiler Collection)

- Pour compiler un programme C avec gcc vers un exécutable sous linux, on part d'un
  - **fichier.c** : le fichier source écrit en langage « C » avec des indications (instruction #) pour le préprocesseur (cpp)

Le lancement de la chaîne de compilation :

```
prompt> gcc fichier.c -o fichier
```

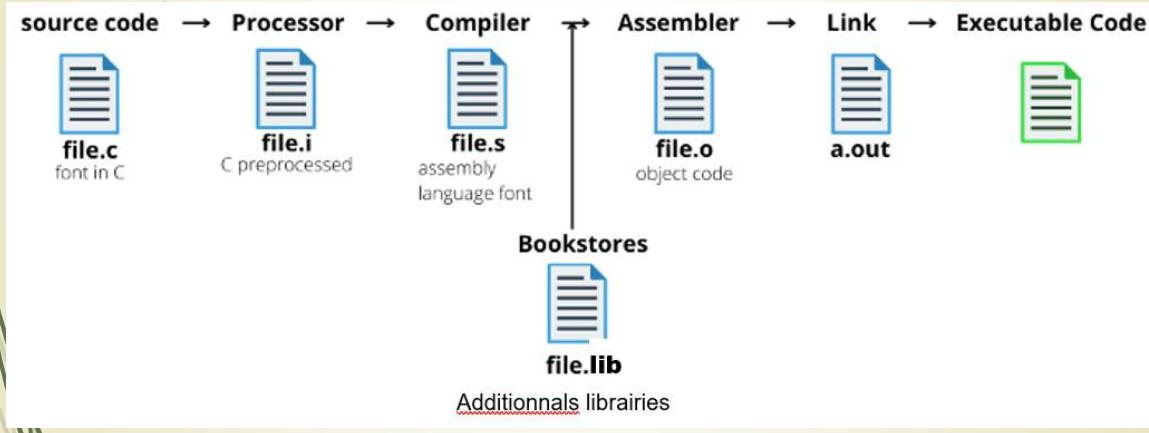
  - **fichier a.out :** le fichier binaire exécutable avec le nom par défaut.

- La chaîne de compilation comprend 4 étapes principales découpables :
  1. préprocesseur (cpp → fichier.i → option –E)
  2. compilateur (cc1 → fichier.s → option –S)
  3. assembleur (as → fichier.o → option –c)
  4. éditeur de liens (ld → fichier a.out)

JFA 33

Les différentes étapes du compilateur C

R 2.04



JFA 34

Les différentes étapes du compilateur C

R 2.04

- Les deux premières étapes travaillent avec des fichiers ***éditables*** (compréhensibles par l'homme), les deux suivantes travaillent avec des fichiers ***binaires***. Le fichier produit, si tout se passe bien est un fichier ***binaire exécutable***.

```
prompt> gcc fichier.c -o fichier_executable
```

est « presque » équivalent à

```
prompt> cpp fichier.c > fichier.i
prompt> cc1 fichier.i > fichier.s
prompt> as fichier.s > fichier.o
prompt> ld fichier.o > fichier_executable
```

Il faudra en plus détruire les fichiers intermédiaires !

JFA 35

Les options de GCC

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Les options de la chaîne de compilation de GCC sont donc :
  - **-E** : permet de stopper le processus de construction après la phase du pré-processeur.
  - **-S** : permet de stopper le processus de construction après l'étape de production du code assembleur.
  - **-c** : permet de stopper le processus de construction après la phase de compilation (ie, production du code assembleur, puis du code machine).
  - **-o** : permet de spécifier le nom du fichier produit, quelle que soit sa nature (code assembleur, code machine ou exécutable).

https://koor.fr/C/Tutorial/gcc.wp

JFA 36

Options relatives à la gestion des warnings

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- **-Wall** : affiche tous les types de warnings. Par exemple, avec cette option, les variables déclarées mais non utilisées produiront des warnings.
- **-Wconversion** : affiche des warnings en cas de cast implicit et de perte de précision. Je vous recommande l'emploi systématique de cette option. Attention, elle ne fait pas partie du périmètre de l'option -Wall.
  - **-Werror** : demande au compilateur de traiter les warnings affichés en tant qu'erreurs. D'un certain point de vue, on peut juger cette possibilité comme étant très contraignante, mais d'un autre point de vue, cette option permet de mieux maîtriser la qualité des codes produits. Je vous recommande aussi l'emploi systématique de cette option.

https://koor.fr/C/Tutorial/gcc.wp

Compiler en mode « Debug » et optimisations

Compiler en mode « Debug »

- **-g** : pour compiler votre programme en mode « Debug » et ainsi pouvoir y faire du pas à pas (via gdb, pourquoi pas).

Options relatives à des demandes d'optimisations

Par défaut le compilateur ne cherche pas à jouer toutes les optimisations possibles. Si vous souhaitez contrôler l'optimisation de votre programme, il faudra utiliser l'une des options suivantes.

- **-O0** : aucune optimisation demandée.
- **-O1** : un premier niveau d'optimisation qui conserve une taille de l'exécutable minimale.
- **-O2** : un niveau d'optimisation intermédiaire avec un bon compromis sur la taille du fichier.
- **-O3** : toutes les optimisations sont demandées, quelles qu'en soient les conséquences sur la taille de l'exécutable.

Arrêt après la phase du pré-processeur.

Pour demander un arrêt de gcc après la phase de pré-processeur, il faut utiliser l'option -E. :

prompt> gcc -E essai.c > essai.i

Note : le pré-processeur ne génère pas de fichier physique sur le disque dur : l'affichage se fait sur la console. Vous pouvez obtenir en obtenir un, en utilisant l'opérateur de redirection de flux **> essai.i**

Le fichier produit étant très conséquent (tous les fichiers inclus via la directive **#include** ont été injectés dans le fichier final), je me permets de vous en afficher que les dernières lignes.

```
$> cat essai.c
... Début du fichier produit ...
# 8 "essai.c"
int main() {
    int result = ( (10)<(20) ? (10) : (20) ) + 2;
    printf( "Hello World with %d\n", result );
    return 0
# 13 "essai.c" 3 4
# 13 "essai.c";
}
```

vous devriez y remarquer le remplacement des macros par le code résultant.

JFA 39

Arrêt après l'étape de génération du code assembleur

- Les plus curieux d'entre vous pourraient jeter un coup d'oeil sur le code assembleur produit. Pour ce faire, il faut utiliser l'option -S. Naturellement, le code ne sera pas assemblé pour produire le code machine. :

Prompt>gcc -S essai.c -o essai.s

- Et voici le contenu du fichier de code assembleur ainsi produit.

```
$> cat essai.s
    .file   "essai.c"
    .text
    .section    .rodata
.LC0:
    .string "Hello World with %d\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB6:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $12, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE6:
    .size   main, .-main
    .ident      "GCC:   (GNU)   9.2.1
    20190827 (Red Hat 9.2.1-1)"
    .section            .note.GNU-
stack,"",@progbits
$>
```


JFA 40

Arrêt après la phase d'assemblage

- C'est une étape que, normalement, vous connaissez bien : il suffit d'ajouter l'option -c pour s'arrêter à ce niveau. On se situe juste avant la phase d'édition des liens. Un fichier d'extension .o (pour objet) sera produit.

Prompt>gcc -c essai.c

JFA 41

Lancement de la phase d'édition des liens

R 2.04

- Pour lancer la dernière phase du « build » de votre programme, il ne faut pas mettre d'option d'arrêt du processus de construction. (-E, -S, -c, ...). Il faut alors utiliser l'option -o si vous souhaitez contrôler le nom du fichier finalement produit. On peut envisager deux cas classiques.
- Soit on lance un build complet à partir du code source original :

```
Prompt>gcc essai.c -o essai
```

- Soit on ne lance que la phase d'édition des liens à partir de fichiers .o déjà produits. :

```
Prompt>gcc essai.o -o essai
```

Au terme de cette étape, vous avez normalement produit un programme exécutable !

https://koor.fr/C/Tutorial/gcc.wp

JFA 42

Les principales utilisations de GCC

R 2.04

Les principales utilisations seront les suivantes :

- Compilation séparée de deux fichiers binaires en fichier objets

```
prompt>gcc -Wall -g -c fichier1.c
prompt>gcc -Wall -g -c fichier2.c
# Regroupe de deux fichiers objets dans un fichier binaire exécutable
prompt>gcc fichier1.o fichier2.o -o resultat
```

- compilation en une seule passe

```
prompt>gcc -Wall -g fichier1.c fichier2.c -o resultat
```

- compilation mixte fichier source et objet

```
prompt>gcc -Wall -g –c fichier1.c
prompt>gcc -Wall -g  fichier1.o fichier2.c -o resultat
```

- compilation pour production

```
prompt>gcc -Wall -DNDEBUG -O2 fichier1.c fichier2.c -o resultat
```

JFA 43

Le préprocesseur CPP

R 2.04

- Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives. Les différentes directives au préprocesseur, introduites par le caractère #, ont pour but de :
  - Définir des pseudo-constantes (***#define***) ;
  - Définir des *pseudo-fonctions* ou *macros* (***#define***) ;
  - Inclure des fichiers (***#include***) ;
  - Faire de la *compilation conditionnelle* (***#if, #else, #endif***).

JFA 44

Le préprocesseur CPP

R 2.04

- **La directive #include**
  - Elle permet d'incorporer dans le fichier source le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (stdio.h, math.h,...) ou n'importe quel autre fichier.
  - La directive #include possède deux syntaxes voisines :

    #include <nom-de-fichier>

    recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, /usr/include/) ;

    #include "nom-de-fichier"

    recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option -I du compilateur.

  La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

JFA 45

Le préprocesseur CPP

R 2.04

- **La directive #define**

La directive #define permet de définir :

- des constantes symboliques,
- des macros avec paramètres.

JFA 46

Le préprocesseur CPP

R 2.04

- **Définition de constantes symboliques**

La directive

```
#define nom reste-de-la-ligne
```

demande au préprocesseur de substituer toute occurrence de nom, par la chaîne de caractères reste-de-la-ligne, dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée.

Par exemple :

```
#define NB_LIGNES 10
#define NB_COLONNES 33
#define TAILLE_MATRICE NB_LIGNES * NB_COLONNES
```

Il n'y a toutefois aucune contrainte sur la chaîne de caractères reste-de-la-ligne. On peut écrire

```
#define BEGIN {
#define END }
```

JFA 47

Le préprocesseur CPP

R 2.04

- **Définition de macros**

Une macro avec paramètres se définit de la manière suivante :

#define nom(liste-de-paramètres) corps-de-la-macro

où liste-de-paramètres est une liste d'identificateurs séparés par des virgules. Par exemple, avec la directive

#define MAX(a,b) (a > b ? a : b)

le processeur remplacera dans la suite du code toutes les occurrences du type

MAX(x,y)

où x et y sont des symboles quelconques par

(x > y ? x : y)

- Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.

JFA 48

Attention :

R 2.04

- La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante. Ainsi, si l'on écrit par erreur

#define CARRE (a) a * a

la chaîne de caractères **CARRE(2)** sera remplacée par **(a) a * a (2)**

Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères. En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Si l'on écrit sans parenthèses :

#define CARRE(a) a * a

le préprocesseur remplacera **CARRE(a + b)** par **a + b * a + b** et non par **(a + b) * (a + b)**.

De même, **!CARRE(x)** sera remplacé par **!x * x** et non par **!(x * x)**.

Enfin, il faut être attentif aux éventuels effets de bord que peut entraîner l'usage de macros. Par exemple, **CARRE(x++)** aura pour expansion **(x++) * (x++)**. L'opérateur d'incrémentation sera donc appliqué deux fois au lieu d'une.

JFA 49

La compilation conditionnelle

R 2.04

- La *compilation conditionnelle* a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- La valeur d'une expression
- L'existence ou l'inexistence de symboles.

JFA 50

Compilation en fonction de la valeur d'une expression

R 2.04

- Sa syntaxe la plus générale est :

```
#if condition-1
 partie-du-programme-1
#elif condition-2
 partie-du-programme-2
 ...
#elif condition-n
 partie-du-programme-n
#else
 partie-du-programme-∞
#endif
```

Par exemple, on peut écrire

```
#define PROCESSEUR PC

#if PROCESSEUR == X86-64
 taille_long = 64;
#elif PROCESSEUR == X86
 taille_long = 32;
#endif
```

- Le nombre de #elif est quelconque et le #else est facultatif. Chaque condition-i doit être une expression constante.
- Une seule partie-du-programme sera compilée : celle qui correspond à la première condition-i non nulle, ou bien la partie-du-programme-∞ si toutes les conditions sont nulles.

JFA 51

Compilation liée à l'existence d'un symbole

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- L'objectif est de conserver ou non dans le code source une partie des instructions C, suivant l'existence ou la non existence de certains identificateurs.

- Sa syntaxe est :

```
#ifdef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

Par exemple, on peut écrire

```
#define DEBUG
 ....
#ifdef DEBUG
 for (i = 0; i < N; i++)
 printf("%d\n",i);
#endif /* DEBUG */
```

- Si symbole est défini au moment où l'on rencontre la directive #ifdef, alors partie-du-programme-1 sera compilée et partie-du-programme-2 sera ignorée. Dans le cas contraire, c'est partie-du-programme-2 qui sera compilée. La directive #else est évidemment facultative.

JFA 52

Compilation liée à l'existence d'un symbole

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme.

  Il suffit alors de supprimer la directive #define DEBUG pour que les instructions liées au débogage ne soient pas compilées. Cette dernière directive peut être remplacée par l'option de compilation -Dsymbole, qui permet de définir un symbole. On peut remplacer

```
#define DEBUG
```

- en compilant le programme par

```
gcc -DDEBUG fichier.c
```

Ce qui permet de ne pas modifier tous les programmes pour enlever le #define DEBUG, quand on ne souhaite plus le debug !

JFA 53

Compilation liée à l'existence d'un symbole

- Exemple dans le cas de débogage de programme :

➢ La syntaxe du .h :

```
prompt> cat def.h
#ifdef DEBUG
   #define trace(s) printf s
#else
   #define trace(s)
#endif
```

```
prompt> gcc source.c -o source
```

ou si on ne veut pas mettre de

#Define DEBUG dans notre programme

```
prompt>  gcc  -DDEBUG  source.c  -o
source
```

Par exemple, on peut écrire

```
prompt> cat source.c
#define DEBUG
#include "def.h"
#include <stdio.h>
 int main(void){
float r;
printf("entrez un nombre réel :");
scanf("%f",&r);
trace(("valeur du nombre entré :
%f\n",r));
printf("fin de programme\n");
return 0 ;
}
```

JFA 54

Compilation liée à la non existence d'un symbole

- De façon similaire, on peut tester la non-existence d'un symbole par :

➢ Sa syntaxe est :

```
#ifndef symbole
   partie-du-programme-1
#else condition-2
   partie-du-programme-2
#endif
```

Par exemple, on peut écrire

```
#include <stdio.h>
#include <stdlib.h>
 int main() {
   #ifndef NDEBUG
      printf( "Mode DEBUG activé\n" );
   #endif
printf( "Code de votre application\n" );
return EXIT_SUCCESS; }
```

```
$> gcc -o Sample -Wall Sample.c
$> ./Sample
Mode DEBUG activé
Code de votre application
$> gcc -o Sample -Wall -DNDEBUG=1 Sample.c
$> ./Sample
Code de votre application
$>
```

- Voici maintenant un exemple montrant le même programme, mais compilé de deux manières différentes en fonction de si la macro NDEBUG est définie ou non.

https://koor.fr/C/Tutorial/Preprocesseur.wp#ifdef

JFA - 55

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

21/07/2024

JFA 56

Le débogueur « gdb »

- Le logiciel ***gdb*** est un logiciel GNU permettant de déboguer les programmes C (et C++). Il permet de répondre aux questions suivantes :
  - À quel endroit s'arrête le programme en cas de terminaison incorrecte, notamment en cas d'erreur de segmentation ?
  - Quelles sont les valeurs des variables du programme à un moment donné de l'exécution ?
  - Quelle est la valeur d'une expression donnée à un moment précis de l'exécution ?
- ***Gdb*** permet donc de lancer le programme, d'arrêter l'exécution à un endroit précis, d'examiner et de modifier les variables au cours de l'exécution et aussi d'exécuter le programme pas-à-pas.
- En compilant une application avec l'option -g, on peut exécuter cette application sous le contrôle de ***gdb***.

➢ **Conditions pour utiliser GDB** :

```
prompt> gcc -g mon_prog.c -o mon_prog
prompt> gdb mon_prog
gdb>
```

JFA 57

Les commandes du débogueur « gdb »

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Compilation et lancement de GDB

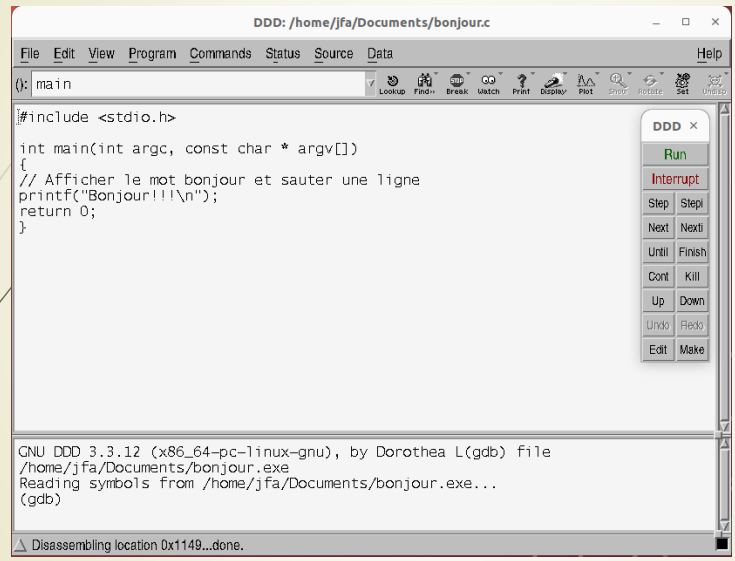
```
prompt> gcc -g -Wall mon_prog.c -o mon-prog
prompt> gdb mon_prog
```

- Fonctionnalités de GDB
  - `list` // liste le programme
  - `(Entrée)` // liste les lignes suivantes
  - `break main` // place un point d'arrêt à la 1ère ligne
  - `set args un deux trois` // donne les arguments pour le main
  - `run` // lance le programme, précisez ici les arguments
  - `print argv[1]` // on examine les variables en utilisant les opérateurs du C
  - `break 17` // place un point d'arrêt ligne 17
  - `cont` // le programme reprend son exécution
  - `next` // arrivé au point d'arrêt on avance pas-à-pas
  - `step` // idem next, mais on rentre dans les fonctions si elles ont été compilé avec l'option -g
  - `watch` // place un watchpoint sur une variable qui a pour effet d'arrêter l'exécution si la valeur de cette variable change
  - `backtrace` // affiche la pile d'exécution du programme
  - `quit` // le programme reprend son exécution

JFA 58

Le débogueur « ddd »

IUT GRAND OUEST NORMANDIE – INFO – R 2.04



Documentation en français :
http://www.linuxfocus.org/Francais/January1998/article20.html

Le programme ddd propose une interface graphique très riche pour gdb. On peut visualiser un très grand nombre d'informations concernant le binaire choisi

JFA 59

Le débogueur « gdb » et les cores

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Le débogueur *gdb* peut aussi examiner le fichier **core** produit lors de la fin ***brutale*** d'un processus.
- Ce fichier **core** est une image exacte du processus au moment même où il a été stoppé pour une erreur incorrigible, par exemple une division par zéro.
- Dans le cas d'un accès mémoire illégal, le fichier **core** indiquera la ligne du code source où l'erreur s'est produite, et nous permettra d'inspecter les pointeurs et variables pour déceler l'accès erroné.
- La commande ***ulimit*** du Shell permet de fixer des limitations de ressources (usage de la mémoire, du temps CPU, du nombre de fichiers ouverts simultanément...) pour les commandes lancées à partir du Shell. Par défaut la taille maximale du fichier **core** est fixée à 0 !

```
prompt> ulimit -a
core file size (blocks, -c) 0
…
prompt> ulimit -c unlimited
prompt> ulimit -a
core file size (blocks, -c) unlimited
…
```

JFA 60

Le débogueur « gdb » et les cores

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- *prompt>cat exemple-crash.c*

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

int main (void)
{
char * pointeur=NULL;
fprintf(stderr,"Avant écriture dans le pointeur\n");
pointeur[0]='X';
printf(stderr,"Apres écriture dans le pointeur\n");
return EXIT_SUCCESS;
}
```

- *prompt>gcc -g exemple-crash.c -o exemple-crash*

```
prompt> exemple-crash
Avant écriture dans le pointeur
Segmentation fault (core dumped)
prompt>ls -lh core
-rw------- 1 cpb cpb 253952 févr. 10 22:04 core
prompt> gdb exemple-crash core
GNU gdb (GDB) 7.6.1-ubuntu […]
Core was generated by `solution-02-04'.
Program terminated with signal 8, Arithmetic exception. #0 0x00000000004005b7 in main () at exemple-crash.c:8
8 fprintf(stderr,"Apres ecriture dans le pointeur\n");
(gdb) print pointeur
$1 = 0x0
(gdb) quit
```

JFA 61

Le débogueur « gdb » et les cores

R 2.04

- Ici le signal 8 correspond à SIGPFE (Page Fault Exception), car on accède à une adresse de la mémoire virtuelle où aucune adresse physique n'a été rattachée.
- On aurait pu avoir le signal 11 (SIGSEGV – Signal Segmentation Violation).
- Suivant la configuration du système (systemd) les fichiers « core » peuvent ne pas être visible dans le système de fichier, car ils peuvent être stockés dans une base de données.
- Dans ce cas il faut utiliser l'utilitaire « coredumpctl » pour les examiner.

```
prompt> sudo apt-get install systemd-coredump
prompt>
prompt>coredumpctl list // affiche la liste des processus qui
ont crashé avec un fichier «core»
prompt>coredumpctl gdb 1234 // lance « gdb » avec le «core»
 correspondant
prompt>gdb coredumpctl  // lance «gdb» avec le dernier «core»
qui a été produit
```

JFA 62

Le débogueur « gdb » et la mémoire

R 2.04

Le débogueur symbolique **gdb** permet aussi d'examiner la mémoire utilisée par le programme ainsi que les registres utilisés par le processeur.

- Dans l'exemple suivant, **gdb** est utilisé pour voir l'état des registres avant le démarrage de l'exécution du programme.

```
Prompt>gdb -q ./a.out  // -q évite les informations affichées avant gdb
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb) break main    // placement d'un point d'arrêt sur le «main»
Breakpoint 1 at 0x400531   // première instruction effective du programme main
(gdb) run // lancement de l'exécution du programme jusqu'au prochain point d'arrêt
Starting program: /home/a.out
Breakpoint 1, 0x0000000000400531 in main ()
(gdb) info register    // visualisation des registres d'exécution du processus
rax 0x40052d   4195629  // adresse du «main» dans le processus «rax» contient la
valeur de retour des fonctions
rbx 0x0     0   // registre général
rcx 0x0     0   // registre général, compteur de boucle
rsp 0x7fffffffde30 0x7fffffffde30     //  pointeur  de  pile  (stack  pointer),
contient l'adresse de la donnée au sommet de la pile
r12 0x400440    4195392     // adresse de la section « text » dans le processus
rip 0x400531    0x400531 <main+4> // première instruction effective du programme,
le registre « rip » est le compteur du programme
(gdb) quit // sortie du débogueur
```

JFA 63

Désassembler avec « gdb »

Le débogueur symbolique **gdb** permet aussi de désassembler le code C d'un programme en son équivalent en langage d'assemblage.

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

```
prompt>gdb -q firstprog
// lancement de gdb avec le programme en paramètre
 Reading symbols from firstprog...done.
(gdb) list   // permet d'afficher les lignes du programme en C
#include <stdio.h> int main()
{
    int i;
    for(i=0; i < 10; i++)
    {
      printf("Hello World!\n");
    }
}
```

```
(gdb) disassemble main
// permet d'afficher le code du programme en assembleur
Dump of assembler code for function main:
0x000000000040052d <+0>:   push    rbp
0x000000000040052e <+1>:   mov     rbp,rsp
0x0000000000400531 <+4>:   sub     rsp,0x10
0x0000000000400535 <+8>:   mov DWORD PTR [rbp-0x4],0x0
0x000000000040053c <+15>:  jmp     0x40054c <main+31>
0x000000000040053e <+17>:  mov     edi,0x4005e4
0x0000000000400543 <+22>:  call    0x400410 <puts@plt>
0x0000000000400548 <+27>:  add     DWORD PTR [rbp-0x4],0x1
0x000000000040054c <+31>:  cmp     DWORD PTR [rbp-0x4],0x9
0x0000000000400550 <+35>:  jle     0x40053e <main+17>
0x0000000000400552 <+37>:  leave
0x0000000000400553 <+38>:  ret
End of assembler dump.
(gdb)
```

DWORD signifie des mots de 4 octets (32 bits)

JFA - 64

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

JFA 65

La compilation séparée

R 2.04

La compilation séparée permet de compiler ses différentes parties de programmes séparément, et de compiler le programme final en linkant tous les programmes objet des différentes parties avec le programme principal :

➢ fichier : Principal.c :

```
void main (void )
{
Printf("programme
principal\n");
fct1 ( );
}
```

Le programme principal affiche une chaîne de caractères puis appelle la fonction ***fct1*** qui appelle à son tour la fonction ***fct2***. Ces deux fonctions affichent une chaîne de caractères.

➢ fichier : fct1.c :

```
void fct1 (void)
{
printf ("fonction1\n");
fct2 ( );
}
```

➢ fichier : fct2.c :

```
void fct2 (void)
{
printf ("fonction2\n");
}
```

JFA 66

La compilation séparée

R 2.04

On compile les fichiers fct1.c et fct2.c avec l'option -c pour obtenir deux fichiers objets.

➢ Compilation :

```
gcc fct1.c -c
gcc fct2.c -c
gcc principal.c fct1.o fct2.o -
o principal
```

La commande « nm » visualise la table des symboles contenue dans un fichier binaire (objet ou exécutable). Pour chaque symbole, la commande fournit :

- **son nom**
- **sa valeur (son adresse absolue ou relative)**
- **sa classe**
- **son type (donnée, programme)**

La compilation séparée

JFA 67

- La commande **nm** va alors donner le résultat suivant :

➢ **Table des symboles de fct1** :

```
prompt>nm fct1.o
00000000 t ___gnu_compiled_c
0000000c T _fct1
U _fct2
U _printf
00000000 t gcc2_compiled.
```

- « t » pour du texte (code)
- « T » pour défini (Translated)
- « U » pour non défini (Untranslated)

- Ce listing contient deux variables qui indiquent que l'objet a été créé avec le compilateur **gcc** (*gnu*). Ce programme contient une fonction qui a été définie (***T***). Il s'agit de ***fct1*** et deux fonctions qui ont été utilisées sans être définies (***U***), il s'agit de ***fct2*** et de ***printf***.

La compilation séparée

JFA 68

- Le fichier objet **fct2.o** est à peu près identique :

➢ **Table des symboles de fct2** :

```
Prompt>nm fct2.o
00000000 t ___gnu_compiled_c
0000000c T _fct2
U _printf
00000000 t gcc2_compiled.
```

- « t » pour du texte (code)
- « T » pour défini (Translated)
- « U » pour non défini (Untranslated)

La compilation séparée

- Le fichier objet **principal.o** :

➢ **Table des symboles de principal** :

```
Prompt>nm principal
00004000 d __DYNAMIC
000040c0 B ___CTOR_LIST__
000040c8 B ___DTOR_LIST__
000023a8 T ___do_global_ctors
00002330 T ___do_global_dtors
00002290 t ___gnu_compiled_c
000022d8 t ___gnu_compiled_c
00002420 t ___gnu_compiled_c
00002330 t ___gnu_compiled_c
00002308 t ___gnu_compiled_c
000023f4 T ___main
000040b8 D __exit_dummy_decl
000040b0 D __exit_dummy_ref
00002330 t __main.o
000040c0 D _edata
000040d0 B _end
000040a8 D _environ
00002768 T _etext
00002420 t _exit.o
000022e4 T _fct1
00002314 T _fct2
000040b4 d _initialized.6
000022a8 T _main
00002290 t cca006231.o
00002020 t crt0.o
000022d8 t fct1.o
00002308 t fct2.o
00002308 t gcc2_compiled.
00002330 t gcc2_compiled.
00002290 t gcc2_compiled.
000022d8 t gcc2_compiled.
00002420 t gcc2_compiled.
00002020 T start
```

- « t » pour du texte (code)
- « T » pour défini (Translated)
- « U » pour non défini (Untranslated)

IUT GRAND OUEST NORMANDIE iNFO R 2.04

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

JFA 71

L'utilitaire « Make »

R 2.04

- L'utilitaire « make » permet de réaliser la compilation séparée de façon automatique (en s'appuyant sur les dates d'accès aux fichiers), en utilisant un fichier de configuration appelé « Makefile ».
- La cible définit les dépendances des commandes utiles à la compilation du fichier correspondant, et liste la commande à exécuter pour la réaliser :

```
Cible : dépendances
    Commande
```

  - La cible indique le but désiré, par exemple le nom du binaire exécutable.
  - Les dépendances mentionnent tous les fichiers dont la règle a besoin pour s'exécuter.
  - Les commandes précisent comment obtenir la cible à partir des dépendances.

```
prompt> make cible
```

- Par défaut, sans cible, « make » commence au début du fichier « Makefile ».
- Le but est de décomposer de façon la plus détaillée possible les différentes étapes pour passer des fichiers sources/entêtes vers le binaire exécutable (en partant du binaire exécutable).
  - ***Cela permet de ne recompiler que les fichiers qui ont été modifiés depuis la dernière compilation.***

JFA 72

Les programmes exemples

R 2.04

- ***prompt>cat main.c***

```
#include "a.h"
#include <unistd.h>
#include <stdlib.h>

void fonction_deux();
void fonction_trois();
int main()
{
   fonction_deux();
   fonction_trois();
   exit(0);
}
```

- ***prompt>cat 2.c***

```
#include "a.h"
#include "b.h"
#include <stdio.h>
void fonction_deux()
{
   printf("f2\n");
}
```

- ***prompt>cat 3.c***

```
#include "b.h"
#include "c.h"
#include <stdio.h>
 void fonction_trois()
{
   printf("f3\n");
}
```

JFA 73

Exemple de fichier « Makefile »

- Exemple de fichier « Makefile ».

```
# création de l'application monapp
all : monapp
 monapp : main.o 2.o 3.o          //pour monapp, il faut main.o, 2.o et 3.0
   gcc -Wall -o monapp main.o 2.o 3.o // commande pour créer monapp.o
 main.o : main.c a.h              //pour main.o, il faut main.c et a.h
   gcc -Wall -c main.c -o main.o  // commande pour créer monapp.o
 2.o : 2.c a.h b.h                //pour 2.o, il faut 2.c, a.h et b.h
   gcc -Wall -c 2.c -o 2.o        // commande pour créer 2.o
 3.o : 3.c b.h c.h                //pour 3.o, il faut 3.c, b.h et c.h
   gcc -Wall -c 3.c -o 3.o        // commande pour créer 3.o
 clean :
   rm *.o
   rm *.~c
   rm monapp
```

- Compilation totale :

  Make all

- Nettoyage :

  Make clean

JFA 74

Autres outils

gprof :

- **gprof** est un outil de profilage qui permet de savoir dans quelles parties du code le programme passe la plus grande partie de son temps d'exécution.

```
prompt>gcc -Wall -pg programme.c -o programme
prompt>programme
prompt>gprof programme gmon.out | less
```

Strace :

- **strace** est un outil de traçage de tous les appels systèmes effectués par un programme.

```
prompt>gcc -Wall programme.c -o programme
prompt>strace programme
```

Lint ou Splint (sous Linux) :

- **splint** recherche d'éventuelles erreurs sémantiques dans le programme qui se compile correctement. Il détecte les instructions qui risquent de nuire à la portabilité du code.

```
prompt>splint programme.c
```

En plaçant plusieurs fichiers sources sur la ligne de commande, **splint** réalise un contrôle de cohérence (types) entre tous les fichiers, considérant qu'ils seront liés par l'éditeur de lien ld.

JFA 75

Autres outils

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

Strip :

- **strip** permet de retirer les informations symboliques de débogage d'un fichier.

```
prompt>strip programme
```

Indent :

- **indent** met en forme un fichier source suivant un style donné (Gnu, Kernighan et Ritchie ou encore Berkeley original, …)...

```
// Avec la recommandation officielle du GNU
   prompt>indent -gnu programme.c -o programme2.c
// Avec le style employé par Brian Kernighan et Dennis Ritchie
   prompt>indent -kr programme.c -o programme3.c
//Avec le style Berkeley
   prompt>indent -orig programme.c -o programme4.c
```

https://www.gnu.org/software/indent/manual/indent.pdf

JFA 76

Autres outils

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

Time :

- **time** fournit trois temps machine à la fin de l'exécution de la commande passée en paramètre :
  - Le temps total d'horloge écoulé entre le début et la fin du processus (real),
  - Le temps passé par le processus en mode utilisateur (user),
  - Le temps passé par le système en mode noyau (sys).

```
prompt> time mon_programme
   real 0m0.089s
   user 0m0.088s
   sys 0m0.000s
```

JFA - 77

Les bibliothèques et librairies

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

21/07/2024

JFA 78

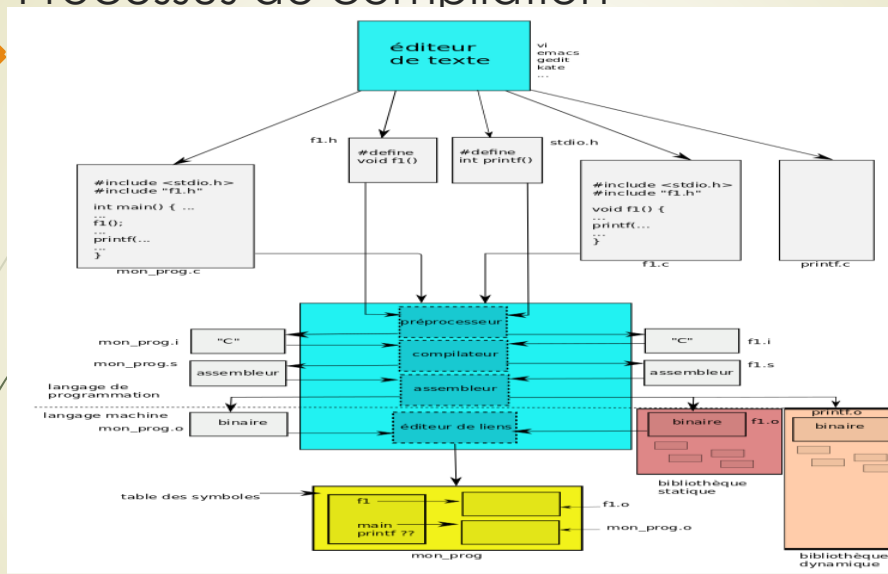
La construction de bibliothèques/librairies

- A toute bibliothèque est associé un fichier d'inclusion (*header file*) d'extension *.h*, destiné à être inclus dans un fichier source (.c) via la directive *#include* du préprocesseur (*cpp*). Il permet l'interface des applications utilisant la bibliothèque correspondante avec cette bibliothèque. On y trouve :
  - La définition de constantes symboliques à utiliser,
  - La définition de types,
  - La déclaration des différentes fonctions de la bibliothèque.
- Une bibliothèque est un *fichier* constitué de modules objets (*binaires*).

  Pour une même bibliothèque, il peut en exister :
  - Une version ***statique*** (Ex. libc.a dans /usr/lib/i386-linux-gnu)
    que l'on appelle aussi bibliothèque/librairie d'archive,
  - Et une version ***dynamique*** (Ex. libc.so ou libc.s1 dans /usr/lib/i386-linux-gnu)
    que l'on appelle bibliothèque/librairie partagée.
- Lors de la compilation, l'option *-l bibliotheque* indique au compilateur de rechercher les symboles dans la bibliothèque nommée :
- Lib bibliotheque.so, ou lib bibliotheque.a si la première n'existe pas.

Processus de compilation

JFA 79



Le binaire de la fonction « printf » étant déjà en mémoire, c'est le lanceur (éditeur de lien dynamique) du binaire qui donnera l'adresse en mémoire de cette fonction à ce dernier (mon_prog).

Bibliothèque statique (archive)

JFA 80

- La commande **`ar`** permet de réunir un ensemble de modules objets (.o) en un seul fichier (bibliothèque statique) en vue d'une liaison statique avec ***`ld`***. L'archive ainsi constituée contient des informations pour reconstituer chacun des fichiers qu'elle contient.

```
prompt> ar clé ref_archive ref_objet1 ref_objet2 …
```

Les valeurs de *clé* sont les suivantes :

```
c : l'écriture de l'archive se fait au début,
d : les références de l'archive,
r : remplace les modules et ajoute les nouveaux en fin,
q : ajoute de nouveaux fichiers en fin sans test d'existence,
t : affiche le nom des modules de l'archive,
x : extrait de l'archive les modules donnés.
V : affiche une ligne de commentaire à chaque module ajouté.
```

La librairie statique commence par un index (table des symboles), consultable par la commande *nm*.

En compilant un binaire exécutable avec une librairie statique, **on inclut le code binaire des modules objets** (.o) de la librairie dans l'exécutable. L'exécution du code peut alors se faire sans avoir la librairie sur la machine. **L'espace disque utilisé par l'exécutable sera d'autant plus important**.

Les programmes exemples

JFA 81

Exemple d'une bibliothèque statique manipulant les nombres complexes ; elle contiendra les deux fonctions : add_complexe et mult_complexe.

```
prompt> cat complexe.h
typedef struct
{
   float reel;
   float imaginaire;
}
complexe;
complexe add_complexe(complexe,
complexe);
complexe mult_complexe(complexe,
complexe);
```

```
prompt> cat add_complexe.c
#include "complexe.h"
complexe add_complexe(complexe
a, complexe b)
{
   complexe c;
   c.reel = a.reel + b.reel;
   c.imaginaire = a.imaginaire
+ b.imaginaire;
   return c;
}
```

```
prompt> cat mult_complexe.c
#include "complexe.h"
 complexe mult_complexe(complexe a, complexe b)
{
   complexe c;
   c.reel = a.reel *b.reel - a.imaginaire *b.imaginaire;
   c.imaginaire = a.reel *b.imaginaire + a.imaginaire *b.reel;
   return c;
}
```

Construire la bibliothèque

JFA 82

Pour construire la bibliothèque, il faut faire les modules objets, puis lancer la commande ar (appelée l'archiveur) :

```
prompt> gcc -c add_complexe.c
mult_complexe.c
prompt> ls
add_complexe.c
add_complexe.o
mult_complexe.c
mult_complexe.o
complexe.h

prompt> ar rv
$HOME/lib/libcomplexe.a
add_complexe.o mult_complexe.o
ar: creating
/home/francois/lib/libcomplexe.a
a - add_complexe.o
a - mult_complexe.o
prompt> rm *.o
```

La commande ***ranlib*** génère l'index pour la bibliothèque ***libcomplexe.a*** et stocke cet index dedans :

```
prompt> ranlib libcomplexe.a
prompt> nm -s  libcomplexe.a
Archive index:
add_complexe in add_complexe.o
mult_complexe in mult_complexe.o
add_complexe.o:     00000000     T
add_complexe
mult_complexe.o:    00000000     T
mult_complexe

       The GNU ranlib program is
another form of GNU ar; running
ranlib is completely equivalent
to executing ar -s.
```

Intégrer la bibliothèque

JFA 83

Il faut ensuite compiler un programme qui utilisera la liaison statique au moment de lancer la chaîne de compilation :

```
prompt> cat complexe.c
#include <stdio.h>
#include "complexe.h"

int main()
{
   static complexe cplx1 = { -98.13, 12.3 };
   …
}

prompt> gcc complexe.c -L$HOME/lib  -lcomplexe -o complexe
```

- L'option **-L** indique au compilateur le chemin où chercher les bibliothèques ;
- L'option **-I** indique le répertoire pour les fichiers entête (.h) et
- L'option **-l** donne le nom (en ajoutant **lib**) de la librairie à aller chercher.

Bibliothèque dynamique / Partagée

JFA 84

➨ Une bibliothèque partagée, **.so** (**s**hared **o**bject) suivi du numéro de version, contient des modules objets ; elle est chargée par l'éditeur de liens dynamique (chargeur) *ld.so*, au moment de l'exécution, en respectant l'ordre suivant :

- Les répertoires mentionnés par la variable d'environnement LD_LIBRARY_PATH,
- Le fichier cache */etc/ld.so.cache*,
- Puis les répertoires */lib* et */usr/lib*.

➨ La bibliothèque partagée est chargée une seule fois et toute entière, quel que soit le nombre de binaires exécutable qui y font référence, d'où un gain de place évident en mémoire. L'exécution d'un tel binaire nécessite donc la présence de cette librairie sur la machine.

JFA 85

Bibliothèque dynamique / Partagée

R 2.04

- Par défaut gcc utilise les bibliothèques système (libc ...) partagées/dynamiques. L'option -static permet l'utilisation statique de ces bibliothèques.

Compilation dynamique par défaut :

```
prompt> gcc complexe.c add_complexe.o mult_complexe.o -o complexe_dyn
```

Compilation statique forcée :

```
prompt> gcc -static complexe.c add_complexe.o mult_complexe.o -o complexe_sta
```

Comparaisons des fichiers exécutables :

```
prompt> ls -lh complexe*
-rwxrwxr-x 1 francois francois 7,2K 2012-01-16 15:44 complexe _dyn
-rwxrwxr-x 1 francois francois 630K 2012-01-16 15:42 complexe_sta
```

On remarque la différence de taille importante antre les deux types de compilation !

JFA 86

Bibliothèque dynamique / Partagée

R 2.04

- La commande **nm** permet de connaître les symboles contenus dans l'exécutable passé en paramètre.
- La commande **objdump** liste les entêtes d'un fichier binaire (exécutable ou bibliothèque partagée) ; on peut ainsi connaître parmi les symboles appelés dans le binaire, ceux qui sont non définis et contenus dans des librairies dynamiques externes.

```
prompt> objdump -x /bin/bash | grep NEEDED
  NEEDED               libtinfo.so.5
  NEEDED               libdl.so.2
  NEEDED               libc.so.6
prompt> objdump -x complexe_dyn | grep NEEDED
  NEEDED               libc.so.6
```

- Enfin, la commande ldd permet de visualiser les bibliothèques dynamiques (.so) dans lesquelles sont contenus les symboles non définis du binaire ; cela permet de définir ce que l'on appelle les dépendances dynamiques.

```
prompt> ldd complexe_dyn
  linux-gate.so.1 => (0xf7748000)
  libc.so.6 => /lib32/libc.so.6 (0xf757c000)
  /lib/ld-linux.so.2 (0xf7749000)
```

JFA 87

Bibliothèque dynamique / Partagée

R 2.04

- Pour utiliser une librairie dynamique (bibliothèque partagée), les fichiers source (.c) de cette librairie doivent être compilés avec l'option **-fPIC** pour que le code soit relogeable (Position Independent Code) ; la génération du code est donc indépendante de son adresse de chargement, qui peut être variable lors de chargements dynamiques. Cela évite toute limite de taille de la table globale des symboles (offset mémoire). Cette option **-fPIC** nécessite un support matériel spécial, qui n'existe pas sur certaines machines, donc ne marche pas dans ces contextes.
- La bibliothèque partagée est créée lors de la compilation en utilisant l'option **-shared**.

```
prompt> cat main1.c
int main()
{
int a = 3,
    b = 4,
    c;
c = add(a, b);
printf("%d\n", c);
}
```

```
prompt> cat add.c
int add(int x, int y)
{
return x + y;
}
```

```
prompt> cat mult.c
int mult(int x, int y)
{
return x * y;
}
```

JFA 88

Bibliothèque dynamique / Partagée

R 2.04

```
prompt> gcc -c -fPIC main1.c add.c mult.c
prompt> gcc -shared -o libOperation.so add.o mult.o
prompt> gcc main1.o libOperation.so -o main1
prompt> main1
7
prompt> ldd main1
linux-gate.so.1 => (0xb785b000) libOperation.so (0xb7856000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb76c8000)
/lib/ld-linux.so.2 (0xb785c000)
```

- La commande *ldd* permet de connaître les dépendances dynamiques (bibliothèques dynamiques) de l'exécutable. ***ld-linux.so.2*** est l'éditeur de lien dynamique.
- Une bibliothèque partagée se trouve dans **/usr/lib**. Sans les droits (*root*) pour le faire, on peut choisir un répertoire dans son espace (*$HOME/lib*). Il faudra alors compiler avec l'option **-Wl,-rpath,$HOME/lib**.

```
prompt> gcc main1.o -I $HOME/include -L $HOME/lib -
Wl,rpath,$HOME/lib -o main1
```

JFA 89

Bibliothèque dynamique / Partagée

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Sous Linux les bibliothèques ont des extensions ***.so*** ou ***.so.nb1*** ou ***.so.nb1.nb2***.
- Lorsque ***nb1*** change, la mise à jour de la bibliothèque est majeure. Dans ce cas il ne faut pas remplacer la bibliothèque courante par cette nouvelle bibliothèque.
- Lorsque c'est ***nb2*** qui change, la mise à jour étant mineure, on peut remplacer la bibliothèque courante par la nouvelle.
- Une bibliothèque a deux noms :
  - Le nom réel qui correspond au fichier qui contient le code (***libm-2.2.4.so***),
  - Et un nom partagé ou soname, qui est un lien symbolique vers le nom réel ; ce nom est composé du nom de la bibliothèque et de son extension majeure (*libm.so.6***)**.

JFA - 90

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

DUT Informatique – Semestre 1

Ressource R2.04

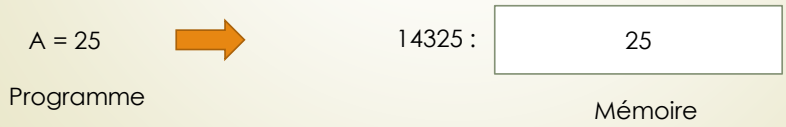
Responsable : Jean-François ANNE

JFA 91

Les pointeurs

R 2.04

- Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale.
- Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle adresse.
- Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets).
- Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.



JFA 92

Adresse et valeur d'un objet

R 2.04

- On appelle Lvalue (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :
  - son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
  - sa valeur, c'est-à-dire ce qui est stocké à cette adresse.
- Dans l'exemple,

```
int i, j;
i = 3;
j = i;
```

- Si le compilateur a placé la variable i à l'adresse 4831836000 en mémoire, et la variable j à l'adresse 4831836004, on a :

Objet	Adresse	Valeur
i	4831836000	3
j	4831836004	3

- Deux variables différentes ont des adresses différentes. L'affectation i = j; n'opère que sur **les valeurs des variables**. Les variables i et j étant de type int, elles sont stockées sur 4 octets. Ainsi la valeur de i est stockée sur les octets d'adresse 4831836000 à 4831836003.

JFA 93

Adresse d'un objet

- L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'**un entier** quel que soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures. Sur un processeur 32 bits, par exemple, une adresse a toujours le format d'un entier (32 bits).

	0	*1*	*2*	*3*
0x8048400	Donnée / 4 octets			
0x8048404	Donnée / 2 octets		Donnée / 2 octets	
0x8048408	Donnée / 1 octet	Donnée / 1 octet	Donnée / 1 octet	Donnée / 1 octet
0x804840C	Donnée / 2 octets		Donnée / 1 octet	vide

https://ilay.org/yann/articles/mem/

- L'opérateur **&** permet d'accéder à l'adresse d'une variable.
- Toutefois **&i** n'est pas une **Lvalue** mais une constante : on ne peut pas faire figurer **&i** à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs.

JFA 94

Notions de pointeurs

- Un pointeur est un objet (Lvalue) dont **la valeur** est égale à **l'adresse** d'un autre objet. On déclare un pointeur par l'instruction :

```
type *nom-du-pointeur;
```

- Où **type** est le type de l'objet pointé. Cette déclaration, déclare un identificateur **nom-du-pointeur**, associé à un objet dont la valeur est l'adresse d'un autre objet de type **type**. L'identificateur **nom-du-pointeur** est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.
- Pour déclarer un pointeur, on met une "*" après son type :

```
int *ptr1; // ptr1 est un pointeur sur un entier
float *ptr2; // ptr2 est un pointeur sur un réel
char *ptr3; // ptr3 est un pointeur sur un caractère
```

- Pour obtenir l'adresse d'une variable, on met le symbole "**&**" devant la variable :

```
scanf("%d", &nb1);
```

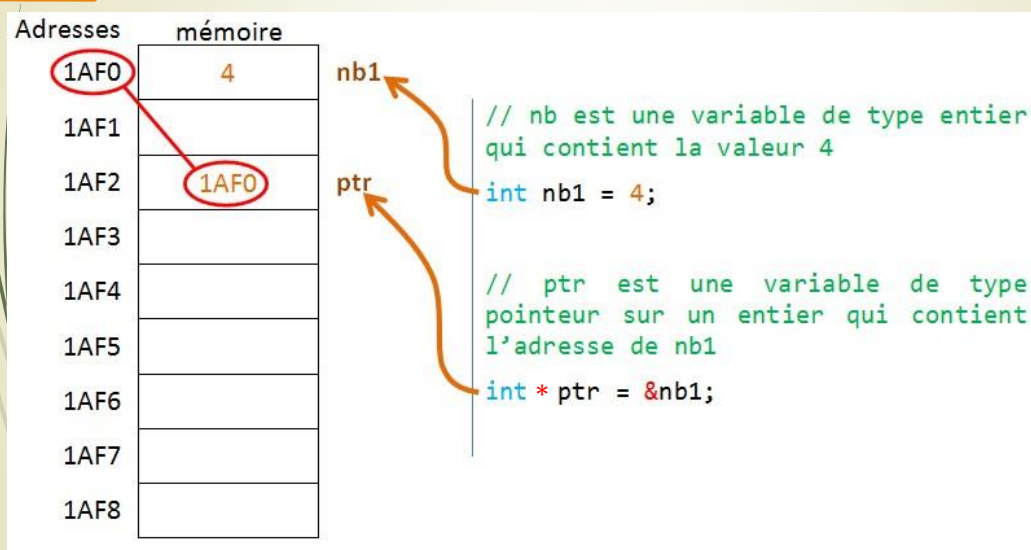
- Pour écrire dans une variable à partir de son pointeur, on met une "*" avant le nom du pointeur :

```
Int *ptr1; // ptr1 est un pointeur sur un entier
*ptr1 = 3; // On écrit la valeur 3 à l'adresse pointée par ptr1
```

JFA 95

Notions de pointeurs

R 2.04



https://www.electro-info.ovh/cours-langage-c-les-fonctions

JFA 96

Notions de pointeurs

R 2.04

- Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet, pour un pointeur sur un objet de type char, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type int, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké. Dans l'exemple suivant, on définit un pointeur p qui pointe vers un entier i :

```
int i = 3;
int *p;
p = &i;
```

- On se trouve dans la configuration :

Objet	Adresse	Valeur
i	4831836000	3
p	4831836004	4831836000

JFA 97

Notions de pointeurs

R 2.04

- L'opérateur unaire d'indirection * permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si **p** est un pointeur vers un entier **i**, ***p** désigne la valeur de **i**. Par exemple, le programme

```
Int main(void)
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
}
```

Objet	Adresse	Valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Imprime :

```
*p = 3.
```

- Dans ce programme, les objets **i** et ***p** sont identiques : ils ont les mêmes adresse et valeur. Nous sommes dans la configuration ci-dessus.
- Cela signifie en particulier que toute modification de ***p** modifie **i**. Ainsi, si l'on ajoute l'instruction ***p = 0**; à la fin du programme précédent, la valeur de **i** devient nulle.

JFA 98

Notions de pointeurs

R 2.04

- On peut donc dans un programme manipuler à la fois les objets **p** et ***p**. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :
- ***Programme 1 :***

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
  printf("p1 = %p \n",p1);
  printf("p2 = %p \n",p2);
  printf("*p1 = %i \n",*p1);
  printf("*p2 = %i \n",*p2);
}
```

- ***Affiche :***

```
prompt >./pointers2
p1 = 0x7ffe39644310
p2 = 0x7ffe39644314
*p1 = 6
*p2 = 6
prompt >
```

Après l'affectation ***p1 = *p2;*** du premier programme, on a :

Objet	Adresse	Valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

JFA 100

Notions de pointeurs

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- ***Programme 2 :***

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
  printf("p1 = %p \n",p1);
  printf("p2 = %p \n",p2);
  printf("*p1 = %i \n",*p1);
  printf("*p2 = %i \n",*p2);
}
```

- ***Affiche :***

```
prompt > ./pointers2b
p1 = 0x7ffd4cdc9354
p2 = 0x7ffd4cdc9354
*p1 = 6
*p2 = 6
prompt >
```

Après l'affectation ***p1 = p2;*** du premier programme, on a :

Objet	Adresse	Valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

JFA 101

Arithmétique des pointeurs

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :
  - L'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
  - La soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
  - La différence de deux pointeurs pointant tous **deux vers des objets de même type**. Le résultat est un entier.
- Notons que la somme de deux pointeurs n'est pas autorisée.
- Si **i** est un entier et **p** est un pointeur sur un objet de type type, l'expression **p + i** désigne un pointeur sur un objet de type **type** dont la valeur est égale à la valeur de **p** incrémentée de **i * sizeof(type)**. Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation **++** et **--**.

JFA 102

Arithmétique des pointeurs

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Par exemple, le programme :

```
main()
{
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %p \t p2 = %p\n",p1,p2);
}
```

Affiche :

p1 = 4831835984 p2 = 4831835988. p2-p1 = 4

On a donc bien 4 octets pour un int

Par contre, le même programme avec des pointeurs sur des objets de type double affiche :

p1= 4831835984 p2 = 4831835992. p2-p1 = 8

Et 8 octets pour un double

JFA 103

Arithmétique des pointeurs

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.
- L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux.
- Si **p** et **q** sont deux pointeurs sur des objets de type **type**, l'expression **p - q** désigne un entier dont la valeur est égale à **(p - q)/sizeof(type)**.
- Ainsi, le programme suivant imprime les éléments du tableau **tab** dans l'ordre croissant puis décroissant des indices.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int *p;
  printf("\n ordre croissant:\n");
  for (p = &tab[0]; p <= &tab[N-1]; p++)
    printf(" %d \n",*p);
  printf("\n ordre décroissant:\n");
  for (p = &tab[N-1]; p >= &tab[0]; p--)
    printf(" %d \n",*p);
}
```

JFA - 104

Gestion de la mémoire

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

21/07/2024

JFA 105

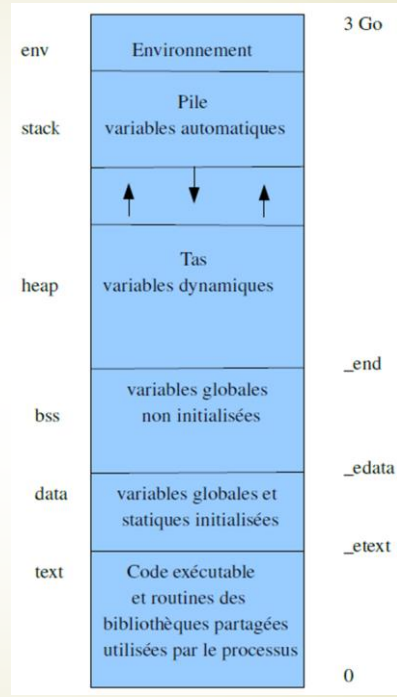
Allocation dynamique de la mémoire en «C»

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Les variables d'un programme « C » peuvent être allouées de différentes manières :
  - **Les variables globales** ou les variables statiques au sein des fonctions, sont allouées une fois pour toutes lors du chargement du programme dans les zones « data » et « bss ».
  - **Les variables locales et les arguments des fonctions** sont réservés sur la pile (zone « stack ») lors de l'invocation des blocs ou fonctions.
  - **Les variables dynamiques** sont allouées dynamiquement/explicitement via des fonctions spécifiques (malloc, …) à travers des pointeurs sur les zones réservées sur le tas (zone « heap »).
- Les variables globales sont automatiquement initialisées à 0 (y compris les pointeurs, avec tous les bits à 0). Les variables locales (sur la pile) ne sont pas initialisées ; leur valeur dépend du contenu de la pile et donc de l'exécution précédente du programme.

JFA 106

- Sur un ordinateur 32 bits un processus dispose d'un adressage de 232 octets soit 4 Go, dont 1 Go pour le noyau, d'où les 3 Go.
- Sur une machine 64 bits l'espace d'adressage va jusqu'à 128 To pour les processus et autant pour le noyau.



IUT GRAND OUEST NORMANDIE iNFO

R 2.04

JFA 107

Allocation dynamique de la mémoire en «C»

- Pourquoi utiliser les variables dynamiques ?
  - Si l'on ne connaît pas la taille de la variable lors de la compilation (par exemple un nombre variable d'éléments d'une liste chaînée) ;
  - Si l'on veut allouer une zone mémoire de très grande taille ; qui plus est, si cette allocation se fait dans une fonction (donc sur la pile) récursive ; cela peut de faire déborder la pile !
  - Si l'on veut optimiser la gestion de la mémoire en réallouant les zones de mémoire au fur et à mesure des besoins, ou en utilisant des organisations de données telles que la liste chaînée, l'arbre binaire ou encore la table de hachage.
- ATTENTION aux problèmes de fuite mémoire, surtout avec des processus fonctionnant sur de très longues durées.
- Lorsque le noyau augmente la taille du segment de données d'un processus, il n'a pas pour autant réservé physiquement de la place dans la mémoire du système. Le principe de la mémoire virtuelle est de ne réserver effectivement une place demandée, que lorsque le processus tente d'y accéder. Il est donc possible de demander plus de mémoire que le système ne peut en fournir.
- Le risque est donc qu'une allocation réussisse, alors qu'elle conduira à la mort du processus dès que l'on tentera d'accéder à cette « zone allouée ».

JFA 108

Allocation dynamique de la mémoire en «C»

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Pour allouer dynamiquement une zone mémoire on utilise « **malloc** » :

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

  - « **taille** » correspond à la taille, en octets, de la zone contigüe désirée;
  - « **size_t** » est redéfini en un type entier « int »,
  - « **malloc** » renvoie un pointeur (void *) sur cette zone, ou le pointeur NULL si la mémoire disponible ne permet pas de faire cette allocation.
- ***Exemple d'utilisation :***

```
ma_struct_t  * ma_struct;
 if ((ma_struct  =  malloc(sizeof(ma_struct_t))) == NULL) {
      fprintf(stderr, "Pas assez de mémoire pour la structure\n");
exit(EXIT_FAILURE);
}
```

JFA 109

Allocation dynamique de la mémoire en «C»

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- En fait la fonction « malloc() » utilise les fonctions bas niveaux :
  - Soit la fonction « sbrk() » pour l'allocation de petits blocs mémoire,
  - Soit la fonction « mmap() » pour l'allocation de gros blocs mémoire.
- La fonction « sbrk() » modifie la fin de la zone d'allocation dans le processus et donc augmente ou diminue la mémoire allouée ;
- Alors que la fonction « mmap() » alloue des blocs mémoires indépendants donc externes au processus appelant (recours aux segments de mémoire partagés IPC).
- La fonction « malloc() » encapsule l'appel à ces fonctions et assure les fonctionnalités suivantes :
  - Alignement mémoire : pratiqué tous les 16 octets sur une machine 64 bits, l'alignement des données permet au processeur de manipuler directement des entiers ou des réels ;
  - Configuration de l'algorithme d'allocation (sbrk/mmap) ;
  - Vérifications propres aux blocs alloués ;
  - Optimisation de l'allocation.
- Elle doit aussi gérer d'éventuels conflits d'accès simultanés à la limite de la zone de données, lors de l'utilisation d'un processus avec plusieurs threads.

JFA 110

Allocation mémoire dynamique initialisée

- Pour allouer dynamiquement des tableaux, initialisés à 0, on utilise « **calloc** » :

```
#include <stdlib.h>
void *  calloc  (size_t  nb_elements, size_t  taille_element);
```

  - « **nb_elements** » correspond au nombre d'éléments à allouer,
  - « **taille_element** » correspond à la taille de chaque élément,.
  - « **calloc** » renvoie un pointeur (void *) sur cette zone, ou le pointeur NULL si la mémoire disponible ne permet pas de faire cette allocation.
- La fonction « **malloc()** » ne garantit aucune initialisation de la zone mémoire allouée, contrairement à la fonction « **calloc()** ».

JFA 111

Allocation mémoire dynamique initialisée

IUT GRAND OUEST NORMANDIE iNFO
R 2.04

- Attention, lorsque la fonction « calloc() » est utilisée pour allouer de grandes zones mémoire, cette dernière appelle la fonction « mmap() » qui ne réservera effectivement cette zone que lors des écritures dedans, donc sans initialisation avec des 0 :

```
int *fonction(int nb_valeurs)
{
    int *table = NULL;
    int cpt;
    if ((table = calloc(nb_valeurs, sizeof(int))) == NULL)
    {
        fprintf(stderr, "Pas assez de mémoire\n");
        exit(EXIT_FAILURE);
    }
    for (cpt = 0; cpt < nb_valeurs; cpt++) table[cpt] = cpt;
    return table;
}
```

```
int main()
{
    int *table;
    int nb_valeurs;
    scanf("%i", &nb_valeurs);
    table = fonction(nb_valeurs);
    free(table);
    return EXIT_SUCCESS;
}
```

JFA 112

Allocation mémoire en C

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Sur un système 32 bits de 3 Go de mémoire centrale on peut lancer un processus qui alloue 4 Go sans avoir de code d'erreur à l'allocation :

```
for (i = 0; i < 4000; i++)
 {
   if ((bloc[i] = calloc(1, 1024 *1024)) == NULL)
   {
        fprintf(stderr, " Echec pour i =%d\n", i);
        break;
   }   // fin if
 } //  fin for
 fprintf(stderr, "Alocation : %d blocs de 1Mo\n", i);
 return EXIT_SUCCESS;
```

- Par contre en utilisant des blocs de 1 Ko, la fonction « calloc() » utilise les fonctions « sbrk() » puis « memset() », provoquant une écriture sur les pages allouées, donc un débordement de l'espace disponible ...

JFA 113

Allocation mémoire dynamique initialisée

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Pour modifier dynamiquement une table déjà allouée par « **malloc()** » ou « **calloc()** », on utilise « **realloc()** » :

```
#include <stdlib.h>
void * realloc (void * ancien, size_t taille);
```

  - « **ancien** » correspond à l'adresse de l'ancienne zone à agrandir,
  - « **taille** » correspond à la taille de la nouvelle zone,
  - « **realloc()** » renvoie un pointeur (void *) sur la nouvelle zone, ou le pointeur NULL si l'allocation échoue. En cas de réussite l'ancien pointeur n'est plus utilisable et le contenu de l'ancienne zone se retrouve au début de la nouvelle zone. En cas d'échec (NULL) l'ancienne zone n'est pas touchée.
- ***Exemple d'utilisation :***

```
void * nouveau ;
nouveau = realloc(bloc_de_donnees, nouvelle_taille)
 if ( nouveau!= NULL)
        bloc_de_donnees = nouveau;
else
        fprintf( stderr, "Pas assez de mémoire \n");
```

- Une réduction de la zone libère de la mémoire et tronque les données.

JFA 114

Allocation dynamique de la mémoire en «C»

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Quelques règles pour éviter les fuites mémoire :
  - Initialisez avec NULL tout pointeur déclaré pour une allocation dynamique, y compris dans les membres de structures ;
  - Avant toute allocation dynamique vérifiez que le pointeur utilisé est bien NULL ;
  - Après tout appel de « malloc() » on vérifie qu'il n'y a pas eu d'erreur, sinon on la gère ;
  - Avant de libérer un pointeur vérifiez qu'il n'est pas NULL ;
  - Dès qu'on a libéré un pointeur avec « free() » on lui redonne immédiatement la valeur NULL.

JFA 115

Allocation mémoire dynamique initialisée

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Il ne faut pas oublier de désallouer la mémoire allouée dynamiquement par une fonction utilitaire. Attention certaines fonctions systèmes allouent dynamiquement de la mémoire sur un pointeur en retour, en utilisant une variable statique. Dans ce cas il ne faut surtout pas désallouer ce pointeur.
- Pour libérer une zone déjà allouée par « malloc() », un « calloc() » ou un « realloc() », on utilise « free() » :

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

  - «**pointeur**» correspond à l'adresse de la zone à libérer et qui a été allouée dynamiquement.
- Une fois la zone libérée, ne jamais réutiliser le pointeur d'accès à cette zone, ni relibérer cette zone.
- ***Bon Exemple d'utilisation :***

```
// préférer
for (ptr = debut ; ptr!= NULL ; ptr = suite) {
suite = ptr->suivant ;
free (ptr) ;
}
```

- ***Mauvais Exemple d'utilisation :***

```
for (ptr = debut ; ptr!= NULL ; ptr = ptr→suivant)
free (ptr) ;
// ERREUR : on vient de libérer « ptr » et on s'en sert pour chercher le pointeur suivant !
```

JFA - 116

DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

21/07/2024

JFA 117

Gestion des fichiers

- Sous UNIX toutes les Entrées/Sorties sont traitées de la même façon à travers la notion de fichier. C'est l'une des caractéristiques majeures d'UNIX. La notion d'enregistrement au sens classique n'existe pas, sinon elle se limite au caractère, et qu'un fichier est donc vu comme un **flot** d'octets (stream) sans structure. Cette dernière revenant aux applications utilisateurs, dont en particulier, les fonctions bibliothèques. C'est ainsi, qu'un fichier peut être un terminal, un fichier classique sur disque ou même une ressource particulière gérée en mode First-In/First-Out qu'est un tube.

- UNIX fait la distinction, entre les divers types de fichiers, par la sémantique des opérations qui leur sont applicables. On parle alors de fichiers réguliers pour les fichiers disques classiques avec un accès direct possible, ils ont une taille de définie et les fichiers spéciaux qui n'ont pas de taille par exemple. : pour les terminaux et les autres devices d'entrées sorties ou de fichiers tube pour les pipes etc...

- Se référer à un manuel UNIX pour plus de détails. Notons seulement que les fichiers spéciaux sont lus en deux modes :

  1. **mode bloc** : pour les disques et
  2. **mode caractère** : pour les terminaux par exemple.

- La distinction réside dans les transferts au niveau système qui, dans le mode bloc, se font par blocs de 512 ou 1024 caractères et transitent donc par les caches système. Dans ce qui suit, on ne parlera que les fichiers réguliers sauf indication contraire.

https://www.emi.ac.ma/~ntounsi/COURS/C/CC/C-IO.html

JFA 118

Gestion des fichiers

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- En C, il y a deux niveaux de traitement de fichiers :
  - Le niveau des primitives de bases (**creat()**, **open()**, **close()**, **read()**, **write()**, **lseek()** ...) qui sont des appels systèmes qui permettent de lire ou écrire des fichiers par blocs de 1 ou plusieurs caractères.
  - Des fonctions bibliothèques plus abstraites comme **fopen()**, **fread()**, **fwrite()**, **printf()**, **fgetc()**, **fprintf()** ...qui se trouvent dans la bibliothèques Entrées/Sorties Standard **stdio.h** (écrite par D. RITCHIE) et qui sont de plus haut niveau et contiennent des fonctionnalités plus nombreuses. Elles offrent par ailleurs des performances meilleures, les appels systèmes y sont gardés au strict minimum. Certaines sont explicitement programmées et d'autres sont macro-définies. Elles présentent une interface uniforme indépendante du système d'exploitation sous-jacent et donc favorables à l'écriture de programmes portables. (Ces fonctions sont réécrites une fois pour toute pour une machine hôte, en fonction des facilités offertes par cette machine).
- Un fichier C est donc un flot d'octets. On peut y écrire tout ce qu'on veut. Cela par opposition à d'autres langages où les fichiers sont typés (on n'y écrit que des objets d'un même type : un record en général). En Pascal par exemple on a : `file of <typeObjet>`. Ainsi, en C, on peut conserver/retrouver tout objet dans un fichier (objet ne contenant pas de pointeurs toutefois). Il suffit pour cela de fournir l'adresse et la taille de la zone mémoire contenant l'objet. Le transfert se fera de cette zone vers le fichier pour l'opération d'écriture (**write**) ou, inversement, du fichier vers cette zone pour l'opération de lecture (**read**). On retrouve ce mécanisme aux deux niveaux de traitement des fichiers.

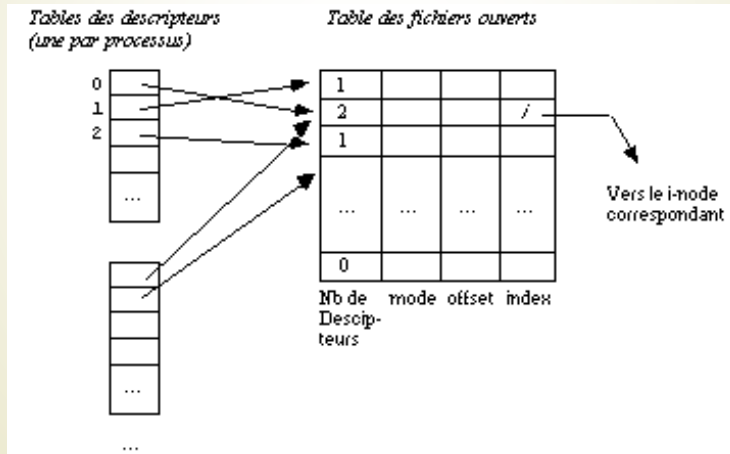
https://www.emi.ac.ma/~ntounsi/COURS/C/CC/C-IO.html

JFA 119

Accès à un fichier

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Un fichier est accessible à travers un descripteur local à un processus qui permet de le référencer indépendamment de son nom physique. Ce descripteur permet d'accéder à une table qui indique l'ensemble des fichiers ouverts. Plus exactement, chaque processus possède une table de descripteurs, un pour chaque fichier ouvert, selon le schéma :



https://www.emi.ac.ma/~ntounsi/COURS/C/CC/C-IO.html

Gestion des fichiers

JFA 120

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

Points à noter :

1. L'indice dans la table des descripteurs est en fait le descripteur lui-même (e.g. 0, 1, 2) qui sont les descripteurs associés à l'entrée standard stdin, la sortie standard stdout et la sortie erreur standard stderr.
2. Chaque entrée dans cette table contient un pointeur vers la table de fichiers ouverts du système UNIX qui, quant à elle, est composées d'une entrée par fichier disque ouvert. Chacune de ces entrées a la structure FILE (prédéfinie dans <sys/file.h>) dont les principales informations sont :
   - Le nombre total de descripteurs correspondants au fichier disque ouvert, (e.g. cas de plusieurs processus partageant un même fichier hérité)
   - Le mode d'ouverture (lire "r", écrire "w", lire/écrire "rw")
   - La Position (dite offset) courante en lecture/écriture, i.e. prochain caractère en E/S
   - Pointeur vers un I_Node (Index_Node) qui est un élément de la table d'allocation des fichiers d'un disque. Cette dernière, est la table (cf. FAT, VTOC, ...) qui se trouve dans les premiers blocs d'un disque logique, et qui décrit l'ensemble des fichiers de ce disque. Elle est en partie chargée en mémoire au démarrage.

Gestion des fichiers

JFA 121

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- Un fichier est désigné, dans un programme, par une variable, qu'habituellement on considère comme le nom logique du fichier, par opposition à son nom physique sur disque. En cas de manipulation directe par les primitives de base, cette variable est du type entier et contient le descripteur du fichier et en cas d'usage de la bibliothèque <stdio.h> cette variable est du type FILE*, un pointeur vers toute une structure qui contient beaucoup plus d'informations, notamment le vrai descripteur entier et un tampon cache (buffer) de taille BUFSIZE. En effet, comme ces fonctions sont de haut niveau, elles utilisent une structure plus appropriée. A ce propos, elles ont par exemple leur propre tampon cache d'E/S qui se trouvera donc, comme une donnée, dans l'espace d'adressage du processus (les primitives de base, travaillant elles sur le cache système). D'où la présence d'une fonction fflush() et la nécessité de vider le tampon utilisateur dans le vrai fichier (ou le cache system du moins). Dans la suite, on va en faire abstraction, car cela est transparent.
- Remarque : On peut vider le cache système, par la primitive sync(), ou la commande UNIX homologue sync.

JFA 122

Gestion des fichiers : Les Primitives de Base

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Nous allons donc commencer par voir les primitives fondamentales que UNIX offre pour manipuler des fichiers en programmation, et qui sont listées dans la liste suivante :
- Les Primitives Fondamentales d'Accès à un Fichier sont :
  - ***Nom*** : Description
    - ***open()*** : Ouvre un fichier
    - ***creat()*** : Crée un fichier vide
    - ***close()*** : Ferme un fichier
    - ***read()*** : Lit dans un fichier
    - ***write()*** : Ecrit dans un fichier
    - ***lseek()*** : Se positionne sur un octet
    - ***unlink()*** : Supprime un fichier
    - ***rename()*** : *Renomme un fichier*

❖ *Pour utiliser ces primitives, il faut déclarer les includes suivants :*

```
#include <sys/types.h>   /* types POSIX mode_t off_t size_t... */
#include <sys/stat.h>    /* status des E/S */
#include <fcntl.h>       /* Constantes POSIX O_RDONLY... */
```

JFA 123

Création de Fichier : creat()

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Avant d'utiliser un fichier, il faut d'abord vérifier qu'il existe déjà sur le support concerné et/ou qu'une entrée lui soit allouée dans la table des fichiers ouverts.
- La fonction ***creat()*** :

```
char *nom;
mode_t pmode;
int creat (nom, pmode)
```

permet de créer un nouveau fichier (ou de l'écraser s'il existe) appelé nom et retourne son descripteur. Sinon la valeur retour est -1.

- ***nom*** *est une chaîne qui indique le nom physique du fichier.*
- ***pmod****, de type mode_t (macro-défini short), est une protection initiale associée au fichier créé. Si le fichier existe déjà, il garde son ancien mode. Ce mode (chmod) définit les droits d'accès et les privilèges.*

❖ ***Exemple :***

```
int fd;
fd = creat("monFichier",0644);
```

➢ Crée un nouveau fichier, de nom "monFichier" et lui associe la protection "-rw-r--r--" : lecture/écriture pour le propriétaire, lecture pour tous.

JFA 124

Ouverture de Fichier : open()

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- La primitive **open ()** réalise l'opération d'ouverture d'un fichier en allouant une nouvelle entrée dans la table des fichiers ouverts (si elle n'est pas déjà allouée) et retourne son descripteur,
- La fonction ***open()*** :

```
char *nom, int mode_ouverture, mode_t pmode;
int open(nom, mode_ouverture [,pmode])
```

Les paramètres sont :.

- ***nom*** *est le nom physique du fichier à ouvrir.*
- **mode_ouverture** *est un entier indiquant le mode d'ouverture. Il est égal à 0 pour la lecture seule (read), 1 pour l'écriture seule (write) et 2 pour la lecture et l'écriture, c'est à dire la mise à jour (read/write).*
- ***pmode*** *est la protection initiale, comme pour* **creat()**, *si l'ouverture est aussi création de fichier (constante O_CREAT dans mode d'ouverture).*

On peut utiliser les constantes symboliques (drapeau, flag) macro-définies dans **<fcntl.h>**

- ***O_RDONLY*** *pour 0 donc lecture seule (ReaDONLY),*
- ***O_WRONLY*** *pour 1 donc écriture seule (WRiteONLY) et*
- ***O_RDWR*** *pour 2 donc lecture et écriture (ReaDWRite).*

JFA 125

Ouverture de Fichier : open()

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- Par ailleurs, on peut demander une paramétrisation plus poussée pour l'ouverture en faisant une disjonction or bit à bit (opérateur |) d'une de ces trois constantes avec d'autres constants drapeaux de **<fcntl.h>**. Parmi ces dernières, il y a : ,

- **O_TRUNC** pour écraser (tronquer) et ouvrir le fichier s'il existe. Sa taille devient nulle.
- **O_CREAT** pour créer s'il n'existe pas et ouvrir le fichier. Dans ce dernier cas le troisième paramètre pmode est nécessaire.
- **O_APPEND** qui permet d'ouvrir pour rallonger le fichier. Le positionnement se fait alors en bout de fichier (offset = taille du fichier) au lieu du début (offset= 0).

En cas de succès, **open ()** retourne un entier descripteur alloué au fichier, et en cas d'échec la valeur -1 (fichier inexistant, droits existants incompatibles avec mode d'ouverture, trop de fichiers ouverts etc. ...). La variable globale **errno** indique l'erreur produite. Les valeurs possibles avec leur signification sont dans **<errno.h>** et on peut utiliser la fonction **perror()**, pour ce besoin.

Le descripteur retourné est celui qui va servir à lire ou écrire dans le fichier. Par ailleurs, le fichier est positionné au début, offset 0 en l'absence de **O_APPEND**.

JFA 126

Ouverture de Fichier : open()

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

❖ ***Exemple :***

1. Tester si un fichier existe et l'ouvrir

```
char *argv;
int fd;
if ((fd = open(argv[1], 1)) == -1)
     printf("Je ne peux ouvrir %s", argv[1]);
```

2. Ouvrir « monFichier » en lecture seule

```
open("monFichier", O_RDONLY);
```

3. Créer ou écraser s'il existe « tonfichier » et l'ouvrir en écriture seule

```
open("tonFichier", O_WRONLY|O_CREAT|O_TRUNC, 0644 );
```

❖ *Cette dernière instruction est donc équivalente à*

```
creat("tonFichier",0644);
```

JFA 127

Fermeture de Fichier : close()

IUT GRAND OUEST NORMANDIE – iNFO – R 2.04

- La primitive **close ()** réalise l'opération de fermeture d'un fichier,
- La fonction ***close()*** :

```
int fd;
int close(int fildesc);
```

Le paramètre est :

- ***fildesc*** *est le nom du descriptor du fichier à fermer ; de manière à ce qu'il ne référence plus aucun fichier, et puisse être réutilisé. Tous les verrouillages d'enregistrement sur le fichier qui lui était associé, appartenant au processus, sont supprimés,*

S'il réussit, close() renvoie ***0****. S'il échoue, il renvoie* ***-1*** *et* ***errno*** *est renseignée en conséquence.*

❖ ***Exemple :***

```
int fd;
int close(fd);
```

➢ qui ferme le fichier de descripteur **fd**, en libérant ce descripteur. Auparavant, le tampon cache d'E/S est vidé, s'il est encore rempli en mode écriture. On peut forcer le vidage prématuré de ce tampon par la primitive **sync()**.

JFA 128

Lecture de Fichier : read()

R 2.04

- La primitive **read ()** réalise l'opération de lecture du fichier,
- La fonction **read ()** :

```
int fildesc; char *buf; int nb_octets; /* ou size_t nb_octets */
int read(fd, buf, nb_octets)
```

Les paramètres sont :

- ***fildesc*** *: est le nom du descripteur du fichier à lire ;*
- ***Buf :*** *buffer contenant les octets lus du fichier*
- ***Nb_octets :*** *le nombre d'octets à lire du fichier*

lit ***nb_octets*** *de données dans le fichier de descripteur* ***fd*** *et les place dans la zone pointée par* ***buf****. Cette zone doit donc être au moins de taille* ***nb_octets*** *.*

JFA 129

Lecture de Fichier : write()

R 2.04

- La primitive ***write ()*** réalise l'opération de lecture du fichier,
- La fonction ***write ()*** :

```
int fildesc; char *buf; int nb_octets; /* ou size_t nb_octets */
int write(fd, buf, nb_octets)
```

Les paramètres sont :

- ***fildesc*** *: est le nom du descripteur du fichier dans lequel écrire,*
- ***Buf :*** *buffer contenant les octets lus du fichier,*
- ***Nb_octets :*** *le nombre d'octets à lire du fichier.*

écrit ***nb_octets*** *de données à partir de la zone pointée par* ***buf,*** *dans le fichier de descripteur* ***fd.***

Lecture de Fichier : lseek()

JFA 130

IUT GRAND OUEST NORMANDIE – INFO – R 2.04

- La primitive ***lseek ()*** Positionner la tête de lecture/écriture dans un fichier,
- La fonction ***lseek ()*** :

```
int fildesc; off_t *offset; int whence;
int write(fd, buf, whence)
```

Les paramètres sont :

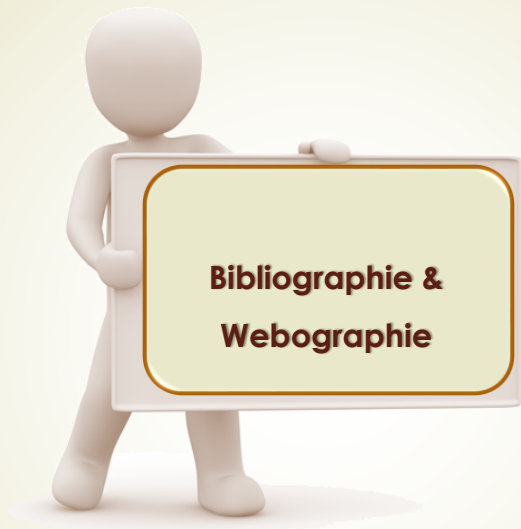
- ***fildesc*** *: est le nom du descripteur du fichier dans lequel écrire,*
- ***Buf :*** *buffer contenant les octets lus du fichier,*
- ***Directive whence :***
  - *SEEK_SET : La tête est placée à offset octets depuis le début du fichier.*
  - *SEEK_CUR : La tête de lecture/écriture est avancée de offset octets.*
  - *SEEK_END : La tête est placée à la fin du fichier plus offset octets*
  - *SEEK_DATA : Positionner la tête sur le prochain emplacement du fichier contenant des données.*
  - *SEEK_HOLE : Positionner la tête sur le prochain trou du fichier. Si offset pointe au milieu d'un trou, la tête est placée sur offset. S'il n'y a pas de trou après offset, la tête est positionnée à la fin du fichier,*

La fonction ***lseek()*** *place la tête de lecture/écriture à la position* ***offset*** *dans le fichier associé au descripteur* ***fd*** *en suivant la directive* ***whence.***

https://manpages.ubuntu.com/manpages/bionic/fr/man2/lseek.2.html

JFA - 131

IUT GRAND OUEST NORMANDIE – INFO – R 2.04



DUT Informatique – Semestre 1

Ressource R2.04

Responsable : Jean-François ANNE

WEBOGRAPHIE

- Liens Web :
  - http://perso.univ-lemans.fr/~cpiau/L2-Outils%20de%20Programmation/C1_L2$PI-Outils%20Avances%20de%20Prog-Compilation%20-etu.pdf
  - https://pagesperso.lip6.fr/Maria.Gradinariu/sites/Maria.Gradinariu/IMG/pdf/cours-7.pdf
  - https://koor.fr/C/Tutorial/gcc.wp
  - https://www.gnu.org/software/indent/manual/indent.pdf
  - https://koor.fr/C/Tutorial/Preprocesseur.wp#ifdef
  - https://www.emi.ac.ma/~ntounsi/COURS/C/CC/C-IO.html
  - https://magoe.fr/chapitre_2.php?idChapitre=522
  - Cours de M. BOURDON F.