Programmation Impérative 2
Introduction au langage C

Amaury Habrard

Faculté des Sciences et Techniques
Université Jean Monnet de Saint-Etienne

Licence Informatique 2 - Semestre 4
2021 - 2022

Qui suis-je ?

- Professeur des Universités

  à l'Université Jean Monnet, St-Etienne, France

  Responsable du Master Informatique, co-responsable parcours MLDM

  au Laboratoire de recherche Hubert Curien
  - Responsable de l'équipe de recherche *Data Intelligence*

- Domaine de recherche :
  - Domaine principal : Apprentissage Automatique *(Machine Learning)*
  - aspects fondamentaux (statistiques, algorithmes, garanties en généralisation)
  - Applications à des problèmes de détection de fraudes et anomalies, de traitement de données textuelles, de sécurité, de vision par ordinateur.

- Comment me joindre :

  Mail : `amaury.habrard@univ-st-etienne.fr`

Objectifs du cours

- Concepts "avancés" de programmation impérative en langage C : récursivité, Makefile, pré-processeur, debuggeur, pointeurs, allocation dynamique, tableaux, structures, unions, listes chainées, TAD Piles et Files, arbres, pointeurs sur fonction, projet
- Objectifs : consolider les acquis du premier semestre, maîtriser la gestion de la mémoire, création de types, et quelques concepts avancés

Références

- Les ≪ slides ≫ - la première partie de rappel est issue des transparents d'Emilie Morvant merci à elle ! - Ils sont susceptibles d'évoluer durant le semestre
- Poly de Anne Canteaut
- Claroline Connect (énoncés, corrections, exemples, polys, ...)
- Le Web ...

Elements d'organisation

- Organisation : CM/TD - lundi après-midis 13h30 et vendredi après-midi 13h30 (pourra commencer à 14h).
  - Importance de chercher les solutions par soi-même !
  - Les solutions des TD sont postées sur claroline une fois que la correction a été vue en cours.
- TP : 2 groupes avec
  Emilie Morvant - `emilie.morvant@univ-st-etienne.fr` Paul Viallard - `paul.viallard@univ-st-etienne.fr`)
- 2 Devoirs de contrôle
  - mi/fin-mars
  - mi/fin-avril
- 1 TP à rendre (consigne à venir)
- Un projet à partir de (fin) avril.

Questions ?

Quelques grands principes

- **Lisibilité des programmes (espace/tabulations) + commentaires réguliers et longs**
  → On programme rarement pour soit tout seul et sans réutilisation ultérieure
  Comment maintenir plusieurs millions de lignes de code faites par plusieurs programmeurs qui partent ou arrivent régulièrement sur un projet ?
  Durée de vie du code !
  → s'imposer des règles/conventions d'écriture.
- Ne pas forcément chercher à gagner une ligne ou minimiser le nombre de variables, sans pour autant manquer de concision : si le code peut être écrit simplement, faites simple
- **Modularité du code et fonctions**
  → Ne pas copier/coller plusieurs fois une même partie de code : faire une fonction
- **Tester les programmes le plus complètement possible** (plusieurs entrées, plusieurs cas possibles), compiler NE SUFFIT PAS
  → Jusqu'à créer des fonctions/programmes de test spécifique
- Quelqu'un qui maitrise un langage doit être capable d'écrire des (petits) programmes corrects sans avoir recours à un compilateur !

Le langage C
Rappels

Exemple de programme simple

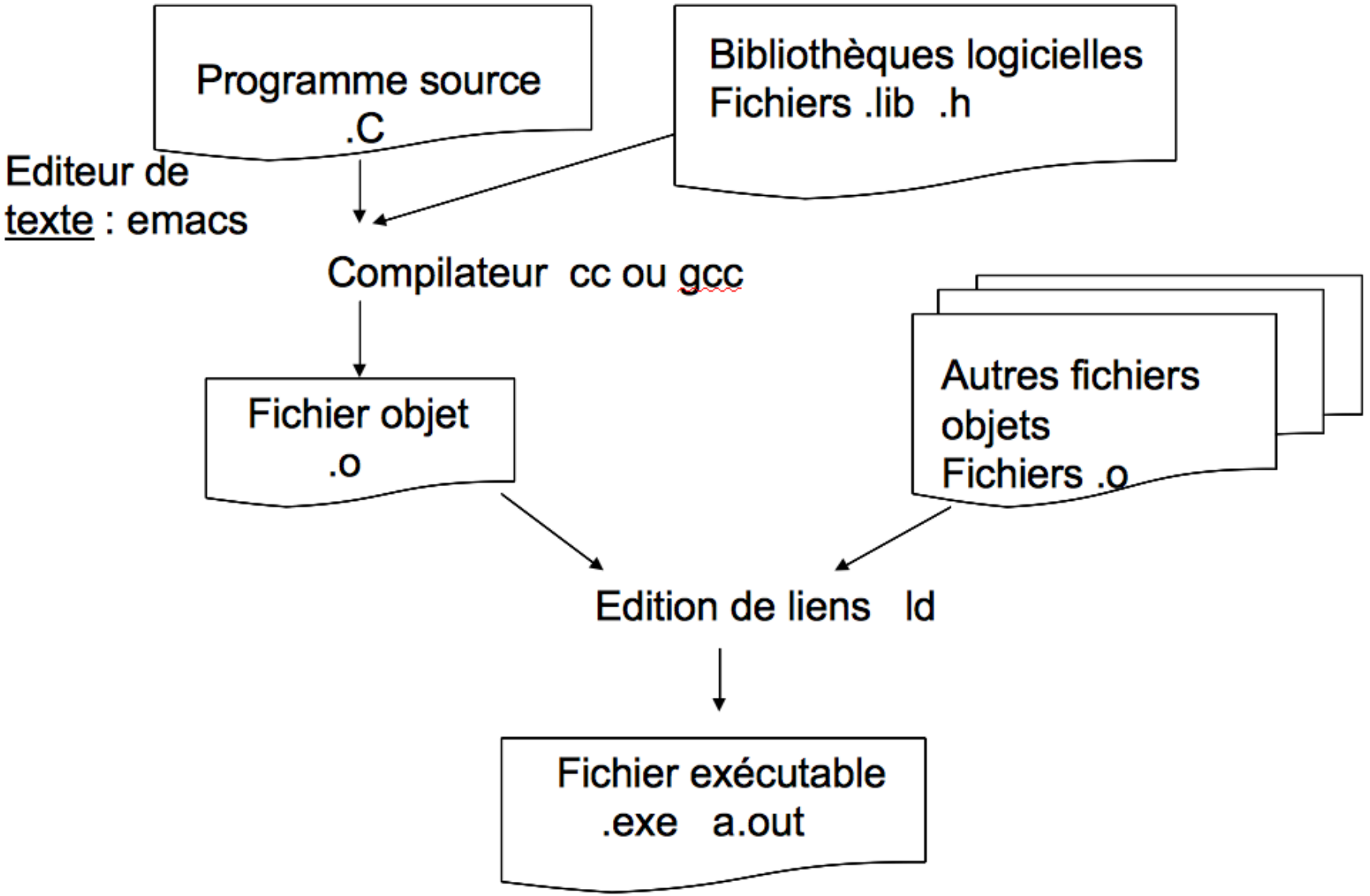
Fichier texte “bonjour.c”

```
#include <stdio.h>      /* bibliothèque utilisée = liste de fonctions*/
#include <stdlib.h>     /* "" */

int main ()             /* mot clé obligatoire pour le début du prog*/
{                       /* début d'un ensemble d'instructions = bloc*/
  printf ("Bonjour!\n");  /* instruction d'affichage */

  return EXIT_SUCCESS;
}
```

Lignes de commandes

```
                                          Lancement de la compilation
> gcc -W -Wall -pedantic -std=c89 bonjour.c -o bonjour
                              note : gcc fait aussi l'édition de liens
                                                Lancement du programme
> ./bonjour
Bonjour!                                        Résultat de l'exécution
>
```

Rappel : Schéma de production de logiciel en C



Les différentes phases de compilation

Les différentes phases de compilation sont :

- Pré-traitement - le traitement par le préprocesseur : le fichier source est analysé par un programme appelé préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source, etc.) - cf plus loin.
- La compilation : le fichier engendré par le préprocesseur est traduit en assembleur i.e. en une suite d'instructions associées aux fonctionnalités du microprocesseur (faire une addition, etc.)
- L'assemblage : transforme le code assembleur en un fichier objet i.e. en instructions compréhensibles par le processeur
- L'édition de liens : assemblage des différents fichiers objets (nous en reparlerons)

Rappel : Programme simple

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>                        /* Inclusion de fichiers      */
#include <stdlib.h>
void usage (char *s) {                    /* Déclaration d'une fonction */
  printf("Usage : %s <entier>",s);
  printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {  /* Déclaration d'une fonction  */
  int n, i;                          /* Déclaration de variables (glob.)*/
  if (argc < 2) {                    /* Suite d'instructions dans   */
     usage(argv[0]);                 /* des blocs { ... }           */
     return EXIT_FAILURE;                      /* chaque instruction termi
;*/
  }
  n= atoi(argv[1]);                  /* convertion chaîne vers entier */
  for (i=1 ; i <=n; i++)             /* boucle de traitement */
      if (n%i == 0)
           printf(" %d ", i);
  printf("\n");
  return EXIT_SUCCESS;
}
```

Vous avez déjà vu :

- Utilisation et inclusion de bibliothèque systèmes (`stdio`, `stdlib`, `ctype`, `string`, `time`, `math`, `...`
- Types simples de variables (`char`, `short`, `int`, `long`, `float`, `double`)
- Jeu d'instructions du C : `if`, `else`, `while`, `for`, `do-while`, `...` et notion de bloc $\cdots$
- Déclaration de fonctions
- La fonction `main`
- Portée des variables
- Lecture d'entrée/affichage de sortie
- Utilisation de fonctions pré-définies
- Tableaux, chaînes de caractères
- Lecture de fichiers

Rappel : descriptif

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>                          /* Inclusion de fichiers      */
#include <stdlib.h>
void usage (char *s) {                      /* Déclaration d'une fonction */
        printf("Usage : %s <entier>",s);
        printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {         /* Déclaration dumain         */
  int n, i;                                 /* Déclaration de variables   */
  if (argc < 2) {                           /* verif nombre arguments     */
     usage(argv[0]);                        /* si probleme                */
     return EXIT_FAILURE;                                 /* on sort du programme
  }                                         /*fin bloc if                 */
  n= atoi(argv[1]);                         /* recuperation de  n         */
  for (i=1 ; i <=n; i++)                    /* boucle de recherche        */
      if (n%i == 0)                         /* test diviseur
          */
           printf(" %d ", i);               /* diviseur trouvé et affich  */
  printf("\n");                             /* affichage saut de ligne    */
  return EXIT_SUCCESS;                      /* tout s'est bien passé      */
}
```

Rappel : Note sur les caractères

Le jeu de caractères normalisé par le langage C est le suivant :

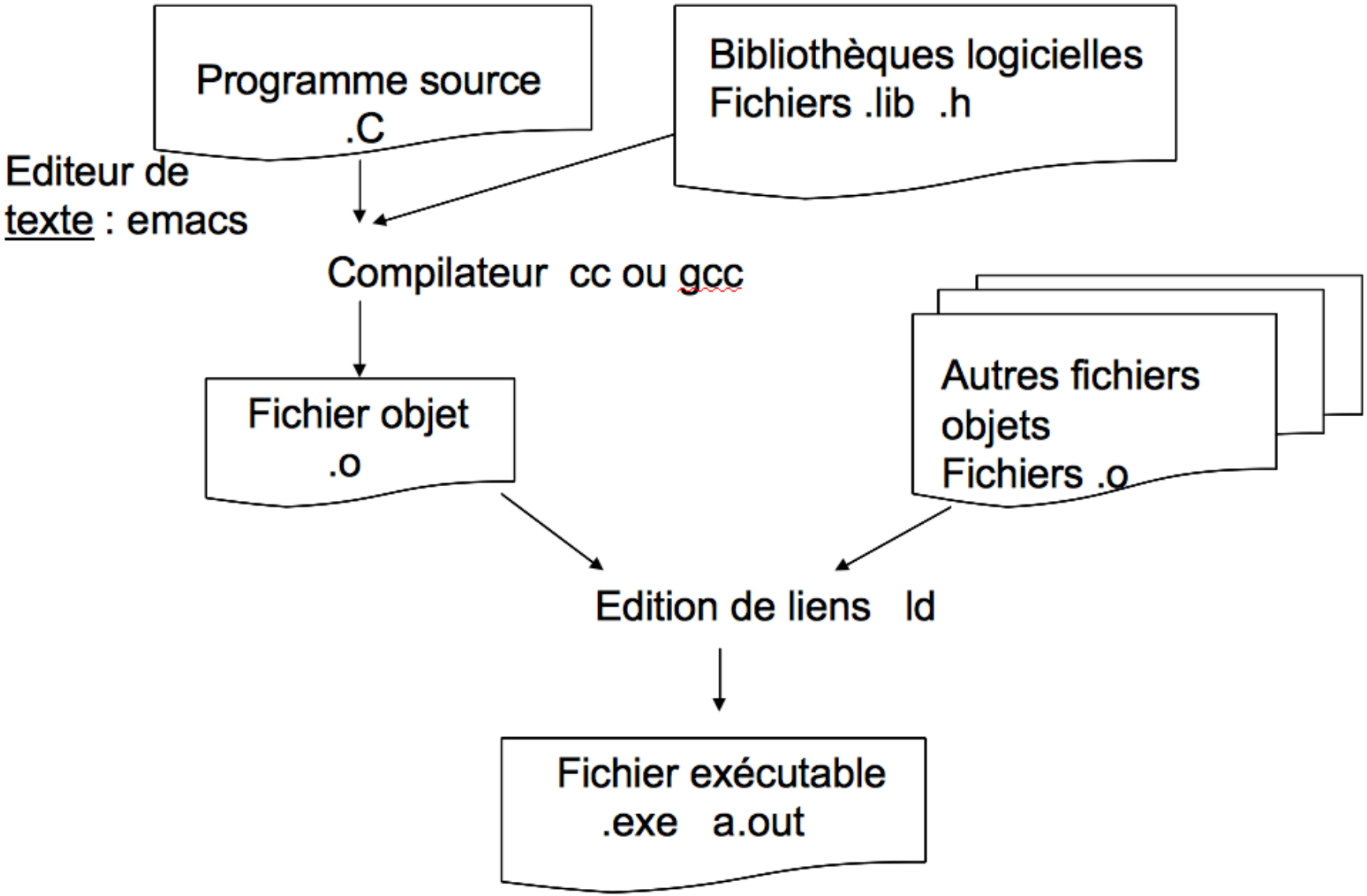
```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

A ceci s'ajoute l'espace, et les caractères de commande représentant :

- le saut de ligne
- le retour chariot
- la tabulation horizontale
- la tabulation verticale
- le saut de page.

L'utilisation d'autres caractères (comme les caractères accentués, par exemple) invoque un comportement défini par l'implémentation.
Dans la pratique, la plupart des compilateurs acceptent les extensions courantes comme IBM PC8 et ISO 8859-1 (aussi appelés respectivement OEM et ANSI dans le monde MS-DOS/Windows). Mais ils peuvent cependant être sujets à des problèmes de portabilité.

Rappel : Schéma de production de logiciel C

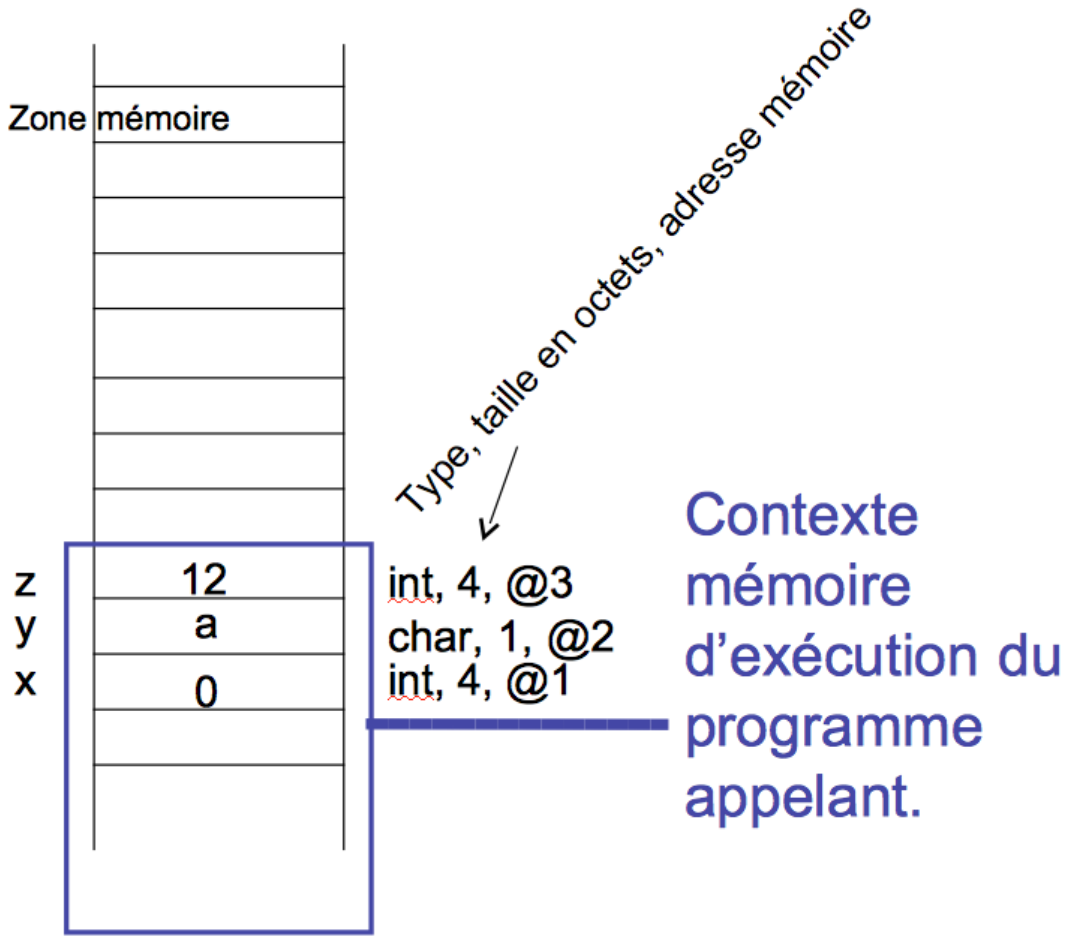


Rappel Important : Paramètres de fonction et zone mémoire

Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

int main (){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);

    return EXIT_SUCCESS;
}
```

Avant l'appel de `maFonction ()`



Rappel Important : Paramètres de fonction et zone mémoire

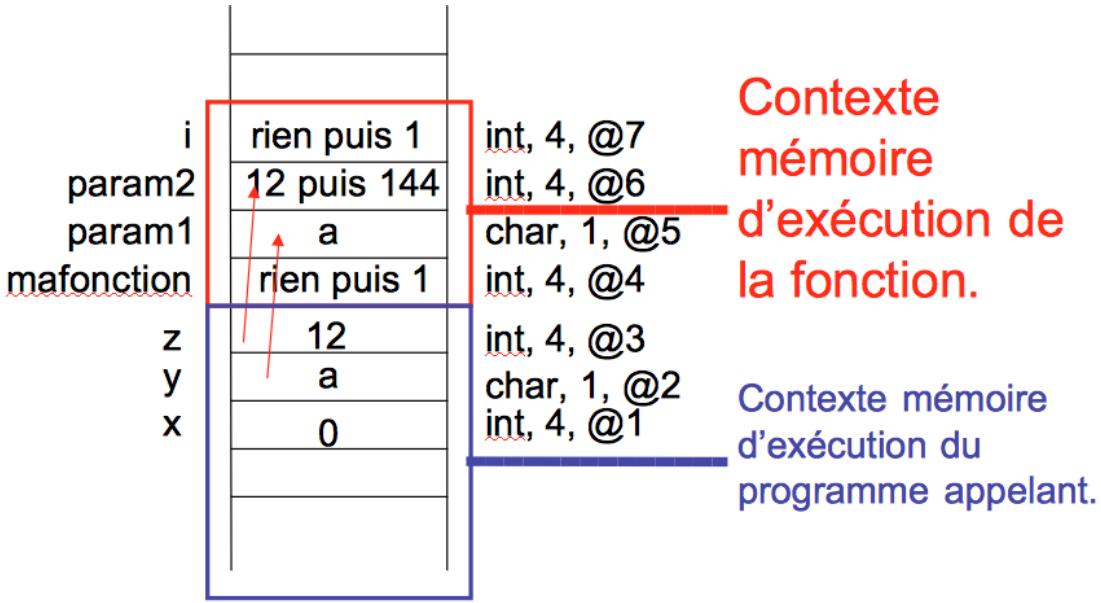
Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

int main (){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);

    return EXIT_SUCCESS;
}
```

Exécution de `x = maFonction (y, z);`

L'appel de la fonction $\Rightarrow$ changement de contexte d'exécution

1. Copie des valeurs données aux paramètres `param1` $\leftarrow$ `y` et `param1`$\leftarrow$ `z`
2. Exécution des instructions



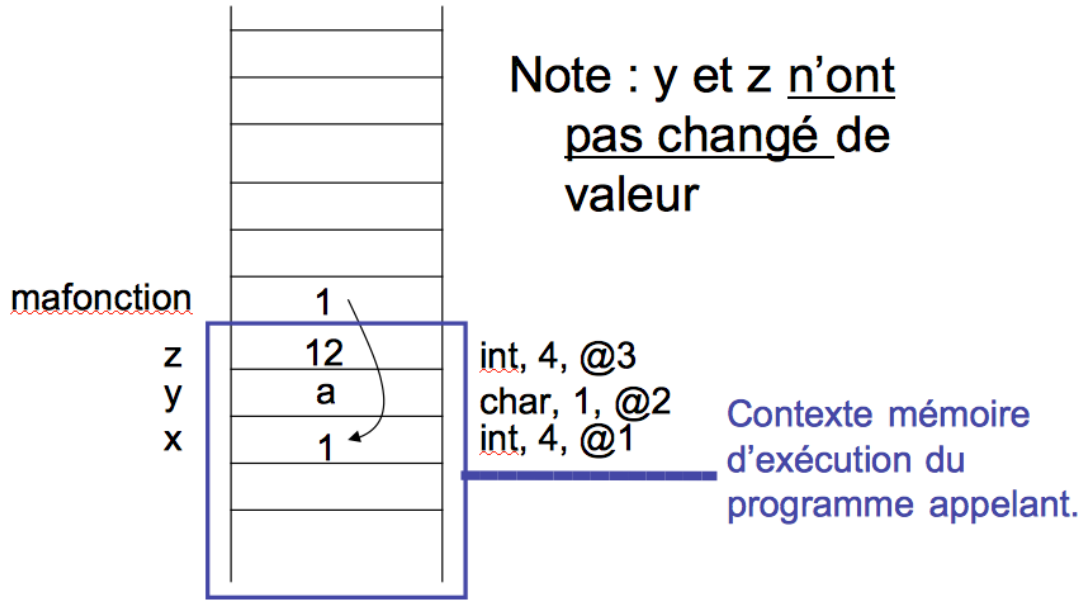
Rappel Important : Paramètres de fonction et zone mémoire

Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

int main (){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);

    return EXIT_SUCCESS;
}
```

après l'exécution de `x = maFonction (y, z);`

Retour au programme appelant
$\Rightarrow$ retour au précédent contexte d'exécution
avec copie du résultat de `x=maFonction(y,z)`

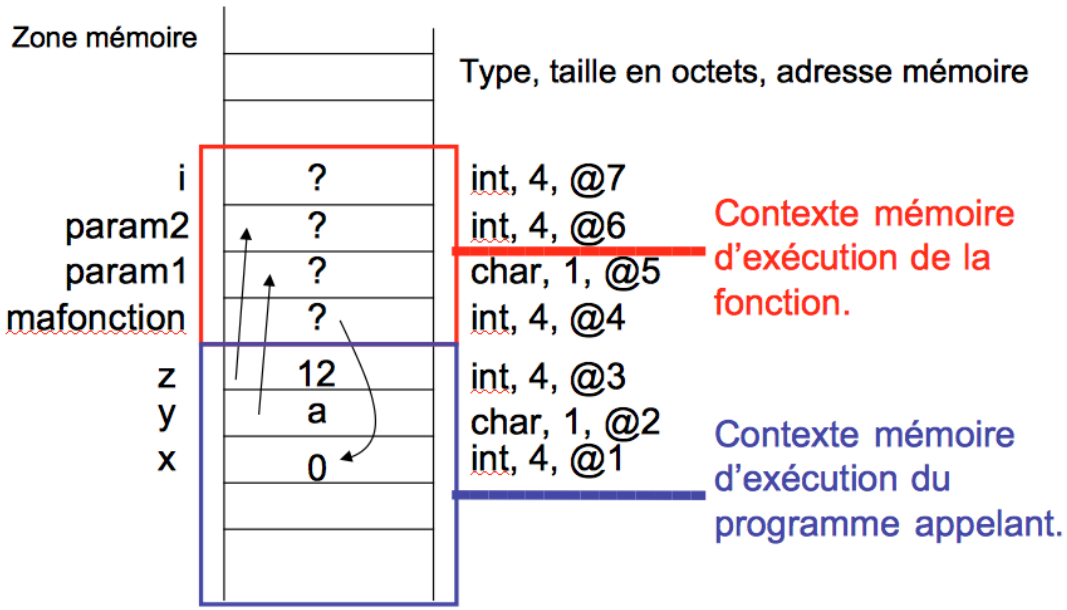


Rappel Important : Paramètres de fonction et zone mémoire

Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

int main (){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);

    return EXIT_SUCCESS;
}
```

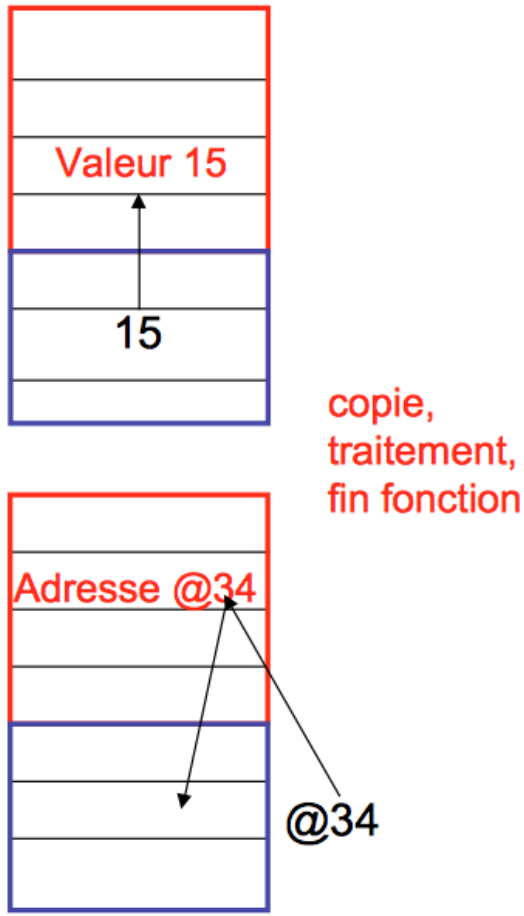
Synthèse



Un petit bilan

Passage de paramètres

- soit par valeur
  - Copie de la valeur
  - Pas d'accès à la variable initiale copiée
  - Modification possible de la valeur locale
- soit par adresse (avec le signe &)
  - Copie l'adresse
  - Accès (modification possible) à la variable initiale via l'adresse



Les fonctions récursives

Il est tout à fait possible d'appeler depuis une fonction cette même fonction. On appelle cela un appel récursif.

Exemple

```
int fact(int n)
{
  if (n==0)
    return 1;
  else
    return n*fact(n-1);
}
```

- **Définir la condition d'arrêt**
- Lorsque la récursivité ne peut pas être faire directement, on utilise une fonction auxiliaire qui se charge de la récursivité (cf TD).

Rappel : tableaux

- Tableau : regroupement de données de **même nature**

- `<type> <nom>[<nbElem>];`

↪ Déclaration d'un tableau de taille `<nbElem>`

- `<type> <nom>[<nbElem1>][<nbElem2>];`

↪ Déclaration un tableau à 2 entrées *ie* une matrice

- `<type> <nom>[];` ou `<type> *<nom>;`

↪ Réserve uniquement le pointeur vers le tableau (pas de réservation de zone mémoire)

- Les indices d'un tableau varient de 0 à `<nbElem>`-1

  `<nom>[0]` est la première case du tableau — `<nom>[<nbElem>-1]` est la dernière

- Les chaînes de caractères sont des tableaux de caractères
  `char chaine[<taille>];` ou `char *chaine;` ou `char chaine[];`

- Les chaînes se terminent par le caractère '\0'

- de nombreuses fonctions pour manipuler les chaînes sont pré-définies dans la bibliothèque `string.h`

Lecture dans un fichier — Un exemple

```
#include<stdio.h>
#include<stdlib.h>
#define TAILLE_MAX 1000

int main(){
   FILE* fichier = NULL;
   char chaine[TAILLE_MAX];
   fichier = fopen("test.txt", "r");
   if(fichier==NULL){
     printf("Vous tentez d'ouvrir un fichier inexistant\n");
     return EXIT_FAILURE;
   }
     /* On lit le fichier tant qu'on ne reçoit pas d'erreur (NULL) */
   while (fgets(chaine, TAILLE_MAX, fichier) != NULL){
     printf("%s",chaine); /* On affiche la chaîne qu'on vient de lire */
   }

   fclose(fichier);
   return EXIT_SUCCESS;
}
```

Lecture dans un fichier — Un autre exemple

On suppose que le fichier `test.txt` contient 3 nombres séparés par un espace (ex : 15 20 30)

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

int main(){
    FILE* fichier = NULL;
    int score[3];
    fichier = fopen("test.txt", "r");
    if(fichier==NULL){
       printf("Vous tentez d'ouvrir un fichier inexistant\n");
       return EXIT_FAILURE;
    }

    fscanf(fichier, "%d %d %d", &score[0], &score[1], &score[2]);
    printf("Les meilleurs scores sont : %d, %d et %d",
                        score[0], score[1], score[2]);

    fclose(fichier);
    return EXIT_SUCCESS;
}
```

Écriture dans un fichier — Un exemple

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

int main(){
    FILE* fichier = NULL;
    int age ;
    fichier = fopen("test.txt", "w");
    if(fichier==NULL){
       printf("Impossible d'ouvrir le fichier\n");
       return EXIT_FAILURE;
    }

    fputs("Salut !\nComment allez-vous ?", fichier);
    fclose(fichier);
    return EXIT_SUCCESS;
}
```

Écriture dans un fichier — Un autre exemple

```
#include<stdio.h>
#include<stdlib.h>
#define TAILLE_MAX 1000

int main(){
    FILE* fichier = NULL;
    int age ;
    fichier = fopen("test.txt", "w");
    if(fichier==NULL){
       printf("Impossible d'ouvrir le fichier\n");
       return EXIT_FAILURE;
    }
    /* on demande votre age */
    printf("Quel age avez-vous ?\n");
    scanf("%d", &age);

    /* On l'écrit dans le fichier */
    fprintf(fichier, "Le Monsieur qui utilise le programme, il a %d ans", age);

    fclose(fichier);
    return EXIT_SUCCESS;
}
```

Quelques mots sur `printf/fprintf` et `scanf/fscanf`

- `scanf` et `printf` utilisent le flot standard (`stdin`, `stdout`)
- `fscanf` et `fprintf` permettent préciser le flot
  (c'est pourquoi on peut les utiliser sur un `FILE *`)

$\Rightarrow$ `scanf("%d",&x);` $\Longleftrightarrow$ `fscanf(stdin,"%d",&x);`

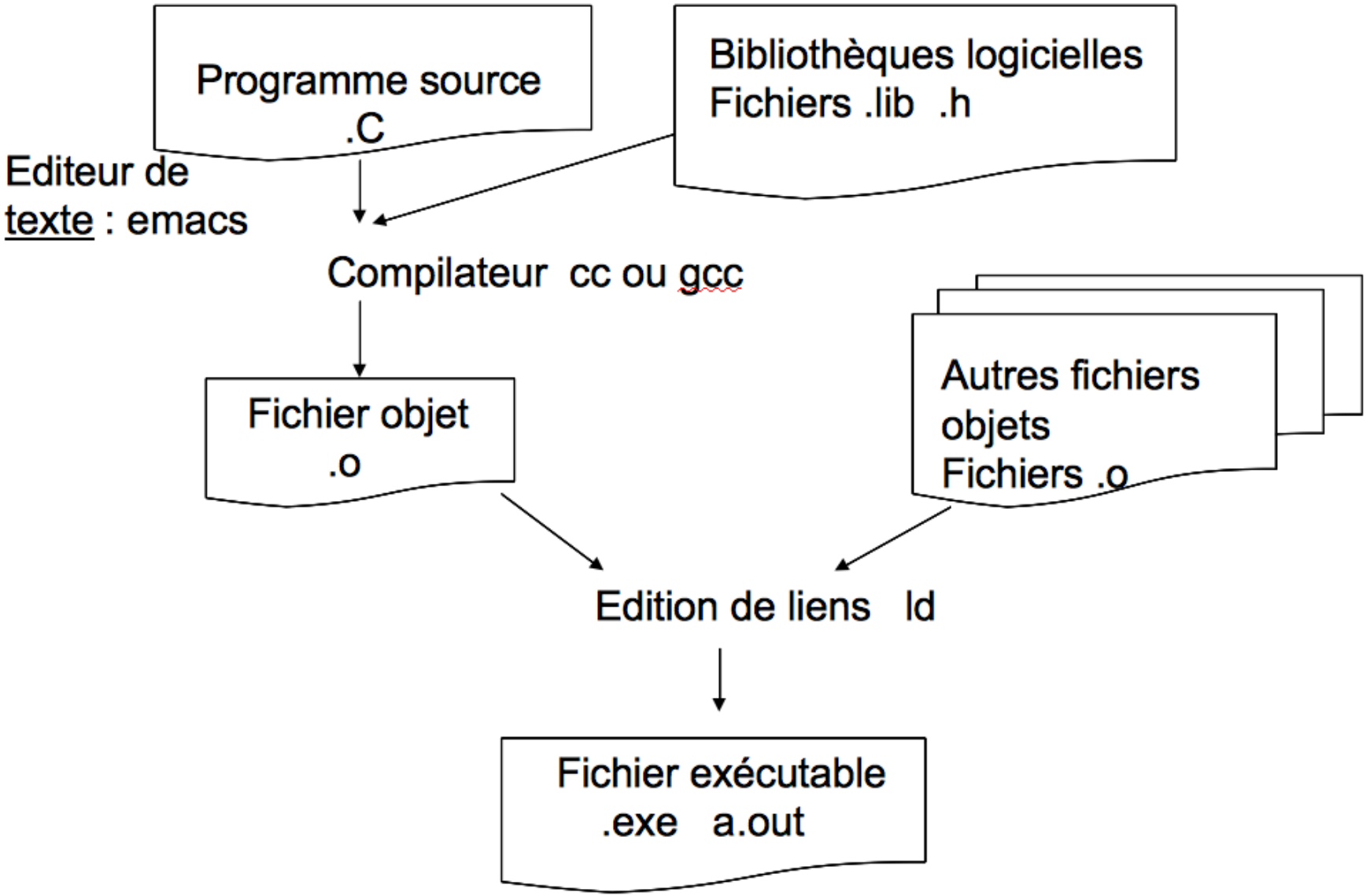
$\Rightarrow$ `printf("salut !");` $\Longleftrightarrow$ `fprintf(stdout,"salut!",&x);`

- Utile pour différencier les affichages d'erreurs des affichages "normaux"

$\Rightarrow$ On peut afficher nos messages d'erreurs sur `stderr` :
`fprintf(stderr,"ceci est un message d'erreur\n");`

Programmation modulaire

Rappel : Schéma de production de logiciel en C



Retour sur la production de programmes

1. Pré-processeur : `.c` $\rightarrow$ un `.c` pour chaque fichier C
2. Compilateur : `.c` $\rightarrow$ un `.o` (langage machine) pour chaque fichier `.c`
3. Edition de liens : $\sum$ `.o` $\rightarrow$ executable (langage machine)

Abstraction de constantes littérales

Les constantes doivent être définies par des constantes à l'aide de la directive `#define`, les utilisations de ce type sont à éviter :

```
perimetre = 2 * 3.14 * rayon;
```

Factorisation du code

- Eviter la duplication du code
- Ne pas hésiter à définir une multitude de de fonctions de petite taille

Fragmentation du code

- Découper le programme en plusieurs fichiers
- Réutilisation facile d'une partie du code pour d'autres applications

Pré-processeur

- `#include` : inclusion textuelle de fichiers
  - Bibliothèque C (ex : `stdio.h`, `math.h`,...)
  - Fichier perso de déclarations de fonctions

- `#define` : remplacement textuel
  - ex : `#define PI 3.1415926`

- Option de `gcc` : `-E` pour visualiser le texte C modifié après passage du pré-processeur

Compilateur

- Traduction en langage machine
- Contrôle des arguments et des types des valeurs de retour pour toutes les fonctions
- Il ne connaît pas le code des fonctions seulement déclarées (option `-c`)

Édition des liens

- Crée l'exécutable à partir des fichiers objets
- "Résolution" des noms de fonctions et de variables indéfinis
- Arrêt dès qu'il y a un défaut

Compilation des modules

- Compilation séparée de chaque module indépendamment
- Permet
  - lisibilité
  - partage du travail
  - maintenance
  - réutilisabilité

Compilation des modules — Un exemple

1. Fichier `pluriel.c`

```
char pluriel_simple (int n) {
   if (n ==1)
      return ' ';
   else
      return 's';
}
```

2. Compilation : `gcc -W -Wall -pedantic -std=c89 -c pluriel.c`
   → `pluriel.o`

3. Utilisation dans un programme `prog.c`
   - Déclarer la fonction : `char pluriel_simple(int n);`
   - Compiler `prog.c` :
     `gcc -W -Wall -pedantic -std=c89 -c prog.c`
   - Lier sa définition : `gcc prog.o pluriel.o -o prog`

Découpage d'un programme en modules

Un module

- Propose un service via une interface de programmation
- Masque l'implantation (la réalisation)
- Un module = 3 fichiers : `.h`, `.c`, `.o`

- Le fichier **interface de programmation** : `.h`

↪ Déclarations des fonctions du module

- Le fichier **réalisation** : `.c`

↪ Définitions des fonctions et fonctions cachées (non présentes dans le `.h`)

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

↪ Code machine des fonctions du module

Exemple : main.c

```
/*****************************************************************/
/***  fichier: main.c                                          ***/
/***  saisit 2 entiers et affiche leur produit                 ***/
/*****************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include "produit.h"
int main(void)
{
  int a, b, c;
  scanf("%d",&a);
  scanf("%d",&b);
  c = produit(a,b);
  printf("\nle produit vaut %d\n",c);
  return EXIT_SUCCESS;
}
```

Exemple : produit.h

```
/*****************************************************************/
/***  fichier: produit.h                                       ***/
/***  produit de 2 entiers                                     ***/
/*****************************************************************/
int produit(int a, int b);
```

Exemple : produit.c

```
/*****************************************************************/
/***  fichier: produit.c                                       ***/
/***  produit de 2 entiers                                     ***/
/*****************************************************************/
#include <stdio.h>
#include "produit.h"

int produit(int a, int b)
{
  return(a * b);
}
```

Compilation séparée

Compilation

```
>gcc -W -Wall -pedantic -std=c89 -c produit.c
>gcc -W -Wall -pedantic -std=c89 -c main.c
>gcc main.o produit.o -o mon_prog
```

Execution

```
>./mon_prog
```

Eviter les double inclusions

```
/*****************************************************************/
/***  fichier: produit.h                                       ***/
/***  en-tete de produit.c                                     ***/
/*****************************************************************/
#ifndef PRODUIT_H_
#define PRODUIT_H_
int produit(int a, int b);
#endif /* PRODUIT_H_ */
```

Note : parfois on trouve le mot clé `extern` indiquant que la fonction/variable est définie dans un autre fichier
Ex : `extern int x;`
variable globale partagée déclarée dans un autre fichier - **Mais les variables globales sont à éviter !**

Règles

A tout fichier `.c` on associe un fichier `.h` qui définit son interface

Fichier `.h`

- directives du pré-processeur (inclusion, compilation conditionnelle)
- déclaration des constantes symboliques et macros
- déclarations des fonctions (utilisées dans d'autres fichiers)

Fichier `.c`

- variables permanentes utilisées que dans le fichier `.c`
- définition des fonctions d'interface présentes dans le fichier `.h`
- définition de fonctions locales au fichier `.c`
- le fichier `.h` doit être inclus dans le fichier `.c` et dans tous les `.c` qui ont besoin des fonctions du fichier

$\Rightarrow$ Fastidieux de gérer la compilation $\ll$ à la main $\gg$

Structure d'un fichier `Makefile`

Un `Makefile` se compose différentes sections

- **déclarations de variables** sous la forme : `<nom> = <valeur>`
- **cible** : un nom d'exécutable ou de fichier objet
- **dépendances** : les éléments ou le code source nécessaires pour créer une cible
- **règles** : les commandes nécessaires pour créer la cible

```
<déclarations de variables>

<cible> : <dépendances>
      <TABULATION><regle1>
      ...
      <TABULATION><regle n>
```

L'utilitaire make

Définition de la liste des dépendances

```
cible: liste de dependances
<TAB> commandes UNIX
```

Exemple

```
prod: produit.o main.o
        gcc -o prod produit.o main.o
main.o: main.c produit.h
        gcc -W -Wall -pedantic -std=c89 -O2 -c main.c
produit.o: produit.c produit.h
        gcc -W -Wall -pedantic -std=c89 -O2 -c produit.c
```

Utilisation

```
>make
gcc -W -Wall -pedantic -std=c89 -O2 -c produit.c
gcc -W -Wall -pedantic -std=c89 -O2 -c main.c
gcc -o prod produit.o main.o
>make
make: 'prod' is up to date.
```

Si on modifie `produit.c`, `main.o` est à jour et n'est pas recompilé

```
>make
gcc -W -Wall -pedantic -std=c89 -O2 -c produit.c
gcc -o prod produit.o main.o
```

Si on recommence :

```
>make
'prod' is up to date
```

Exemple avec debug

```
# Fichier executable prod
prod: produit.o main.o
        gcc -o prod produit.o main.o
main.o: main.c produit.h
        gcc -W -Wall -pedantic -std=c89 -O2 -c main.c
produit.o: produit.c produit.h
        gcc -W -Wall -pedantic -std=c89 -O2 -c produit.c


# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        gcc -o prod.db produit.do main.do
main.do: main.c produit.h
        gcc -o main.do -W -Wall -pedantic -std=c89 -g -Og -c main.c
produit.do: produit.c produit.h
        gcc -o produit.do -W -Wall -pedantic -std=c89 -g -Og -c produit.c

#cible permettant de nettoyer
clean:
        rm -f *.o *.do
```

```
>make prod.db
gcc -o main.do -W -Wall -pedantic -std=c89 -g -O2 -c main.c
gcc -o produit.do -W -Wall -pedantic -std=c89 -g -Og -c produit.c
gcc -o prod.db produit.do main.do
```

$\Rightarrow$ cf TP 3 (`make clean` efface les `.o` et les `.do`)

Note sur les options d'optimisation

Les différentes options -O

- `-O` : Premier niveau d'optimisation. Le compilateur tente de réduire la taille du code et le temps d'exécution tout en limitant le temps de compilation.
- `-O2` : Deuxième niveau d'optimisation, le compilateur essaie d'effectuer une tâche d'optimisation plus importante. Par comparaison avec `-O`, cette option augmente à la fois le temps de compilation et les performances du code généré en termes de temps d'exécution. La taille du code peut être plus importante.
- `-O3` : Troisième niveau d'optimisation, le compilateur essaie d'effectuer une tâche d'optimisation encore plus importante. `-O3` active toutes les options d'optimisation spécifiées par `-O2` plus d'autres. La taille du code produit peut être encore plus importante et le temps mis pour compiler est supérieur, cette option implique une utilisation mémoire bien plus importante.
- `-Os` : Optimisation concernant la taille du code produit. `-Os` active toutes les optimisations de `-O2` qui n'ont pas d'impact négatif sur la taille du code, ainsi que d'autres optimisations conçues pour réduire la taille du code.
- `-O0` : Aucune optimisation effectuée. C'est le comportement par défaut de GCC.
- `-Og` : autorise les options de compilation qui n'interfèrent pas avec le débogage.

Il n'y a pas de garantie que l'option `-O3` produise un code plus efficace que `-O2`. La ≪ science de l'optimisation ≪ de programmes n'est pas toujours exacte et dépend de l'environnement d'exécution.

Note : en cas de code ≪ pas forcément très propre ≫, les options d'optimisation peuvent induire des bugs inexistants sans leur activation. **Ces options sont donc à utiliser avec prudence.**

Remarques sur le respect des normes

Pour spécifier une norme : l'option `-std`/`-ansi`

```
>gcc -o hello1 -W -Wall -pedantic -ansi hello.c
>gcc -o hello2 -W -Wall -pedantic -std=c89 hello.c
```

`-ansi` est équivalent à `-std=c89`.
`-std=c11` est une version récente de la norme

L'option `-pedantic` affiche tous les warnings requis par le standard ISO C (ou `-pedantic-errors` si vous préférez avoir des erreurs plutôt que des warnings).

Plus d'information sur les standards :
`https://gcc.gnu.org/onlinedocs/gcc/Standards.html`

Importance de compiler en respectant des normes !

Macros et abréviation : une meilleure généralisation

```
# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -W -Wall -c -pedantic -std=c89 -O2
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -W -Wall -c -pedantic -std=c89 -g -Og

# Fichier executable prod
prod: produit.o main.o
        $(CC) -o prod produit.o main.o
main.o: main.c produit.h
        $(CC) $(PRODUCTFLAGS) main.c
produit.o: produit.c  produit.h
        $(CC) $(PRODUCTFLAGS)  produit.c

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        $(CC) -o prod.db produit.do main.do
main.do: main.c produit.h
        $(CC) -o main.do $(DEBUGFLAGS) main.c
produit.do: produit.c produit.h

$(CC) -o produit.do $(DEBUGFLAGS) produit.c
```

Macros prédéfinies

Un certain nombre de macros sont prédéfinies. En particulier,

- `$@` désigne le fichier cible courant
- `$*` désigne le fichier cible courant privé de son suffixe
- `$<` désigne le fichier qui a provoqué l'action.

Dans le `Makefile` précédent, la partie concernant la production de `main.do` peut s'écrire par exemple :

```
main.do: main.c produit.h
	$(CC) -o $@ $(DEBUGFLAGS) $<
```

Exemple avec macros

```
# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -W -Wall -pedantic -std=c89 -c -O2
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -W -Wall -pedantic -std=c89 -c -g -Og

# suffixes correspondant a des regles generales
.SUFFIXES: .c .o .do
# regle de production d'un fichier .o
.c.o:; $(CC) -o $@ $(PRODUCTFLAGS) $<
# regle de production d'un fichier .do
.c.do:; $(CC) -o $@ $(DEBUGFLAGS)  $<

# Fichier executable prod
prod: produit.o main.o
        $(CC) -o prod produit.o main.o
produit.o: produit.c produit.h
main.o: main.c produit.h
# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        $(CC) -o prod.db produit.do main.do
produit.do: produit.c produit.h
main.do: main.c produit.h

clean:
        rm -f prod prod.db *.o *.do
```

Cf exemples vus en cours et présents sur claroline

Créer une librairie (statique)

Imaginons que l'on souhaite créer une librairie `libmesmath` contenant la fonction `produit` du fichier `produit.c` et la fonction `somme` du fichier `somme.c`

Procédures

1. Compiler chaque source :
   `gcc -W -Wall -pedantic -std=c89 -c produit.c somme.c`
2. Créer la bibliothèque `libmesmath.a` (option r pour concatener avec l'existant) :
   `ar -r libmesmath.a produit.o somme.o`
3. Générer l'index de la librairie :
   `ranlib libmesmat.a`
4. Créer l'en-tête `libmesmath.h`
   (peut inclure `somme.h` et `produit.h` ou on indique les en-têtes directement)
5. On place la bibliothèque dans un répertoire (ex. `mylib/`) et l'en-tête dans un autre répertoire (ex. `myinclude/`)
6. Pour compiler un fichier source `essaifichier.c` avec cette bibliothèque, on utilise :
   `gcc -I myinclude -L mylib essaifichier.c -o essaifichier -lmesmath`

On peut bien sûr automatiser tout ça avec un Makefile.

Portée des variables et des fonctions

Portée des déclarations dans une fonction

À l'intérieur d'une fonction, on a accès à

- les **fonctions** déclarées **avant**
- les **types** et les **variables** déclarés
  - **avant** et **dans la fonction**
  - **avant dans le fichier source** et **hors d'un bloc** (ex : hors d'une fonction)

À l'intérieur d'une fonction, on **ne voit pas**

- les **variables masquées** (par une déclaration locale de même nom)
- les **variables** déclarées **dans un bloc** (ex : dans une autre fonction)

Portée d'une fonction

Une fonction est **utilisable**

- dans **son module** de définition : après la définition ou après la déclaration
- dans un **autre module** : après la déclaration

Rq : Une fonction définie `static` **n'est pas accessible** depuis un **autre module**

```
static int maFonctionLocale(){ ... }
```

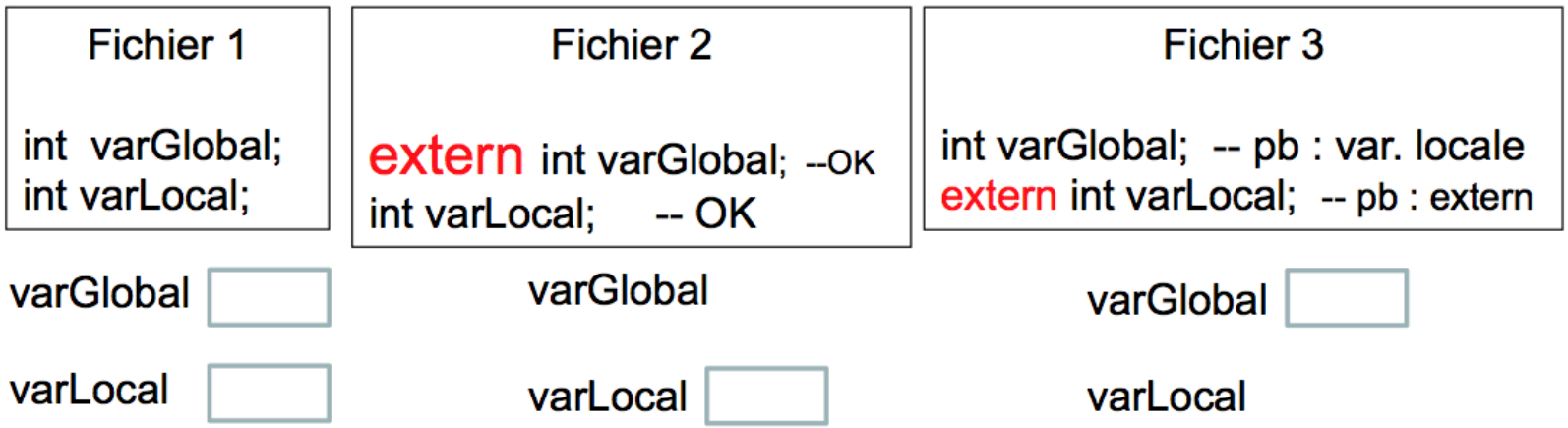
Cette fonction ne sera utilisable QUE dans son module

Portée des variables

Une variable **définie en dehors d'un bloc (ou fonction)** est **accessible**

- dans **son module** de définition : après la déclaration
- dans un **autre module** : après une déclaration `extern`

(déclaration sans réserve de zone mémoire)



Portée des types

Un nouveau type (défini par `typedef`) a une portée **limitée à son module**

Pour pouvoir l'utiliser ailleurs

↪ Il faut créer et inclure un fichier de définition de types

Quelques mots sur `static` et `extern`

Attributs `static` et `extern` dans une programmation par modules

- `static` : pour une variable et une fonction

↪ Implique que la définition reste locale au module

- `extern` : pour une variable

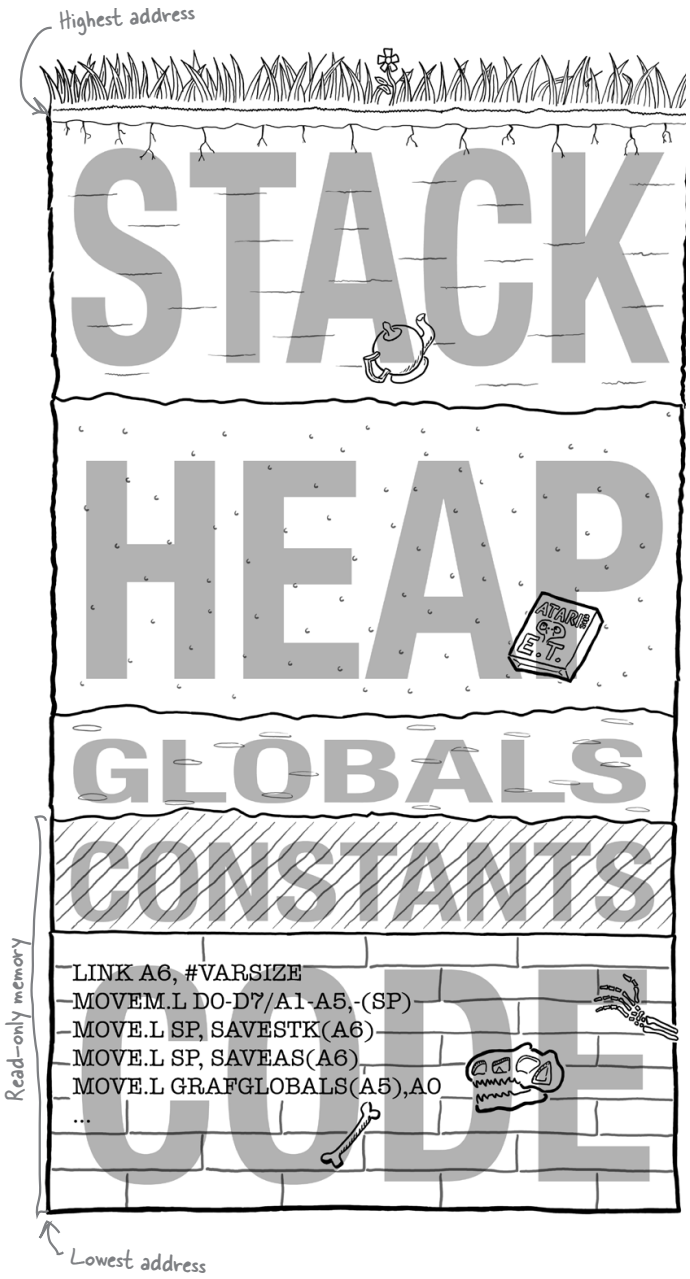
↪ déclaration sans réserve de zone mémoire

⇒ variable dite “globale” qui doit être déclarée sans `static` dans un autre module

- `extern` : pour une fonction (implicite devant la déclaration)

Les différentes parties de la Mémoire

Les différentes partie en mémoire



La pile - stack

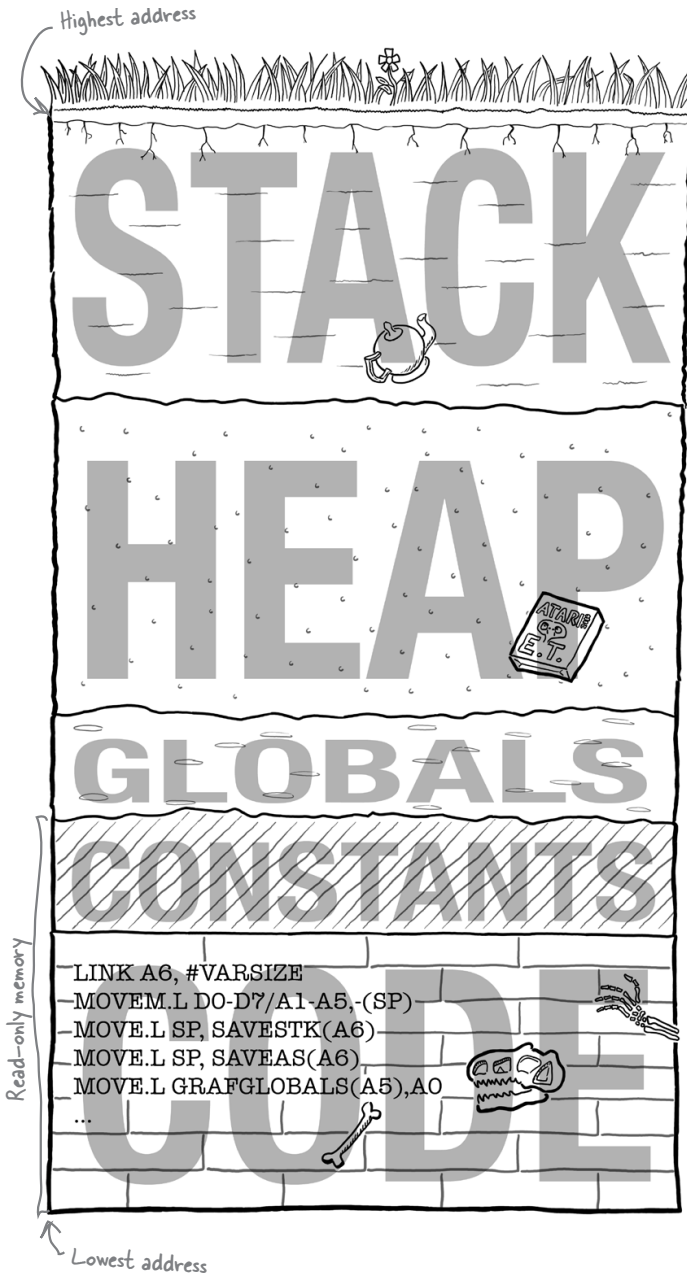
Section utilisée pour le stockage des **variables locales**. A chaque appel de fonction, toutes ses variables locales seront créées sur la pile. pile est en haut !

Les différentes partie en mémoire

Le tas - Heap

Le tas sert à l'allocation de mémoire **dynamique** que nous allons voir. Les variables allouées dynamiquement sont supposées rester longtemps en mémoire.

Les différentes partie en mémoire



Variables Globales

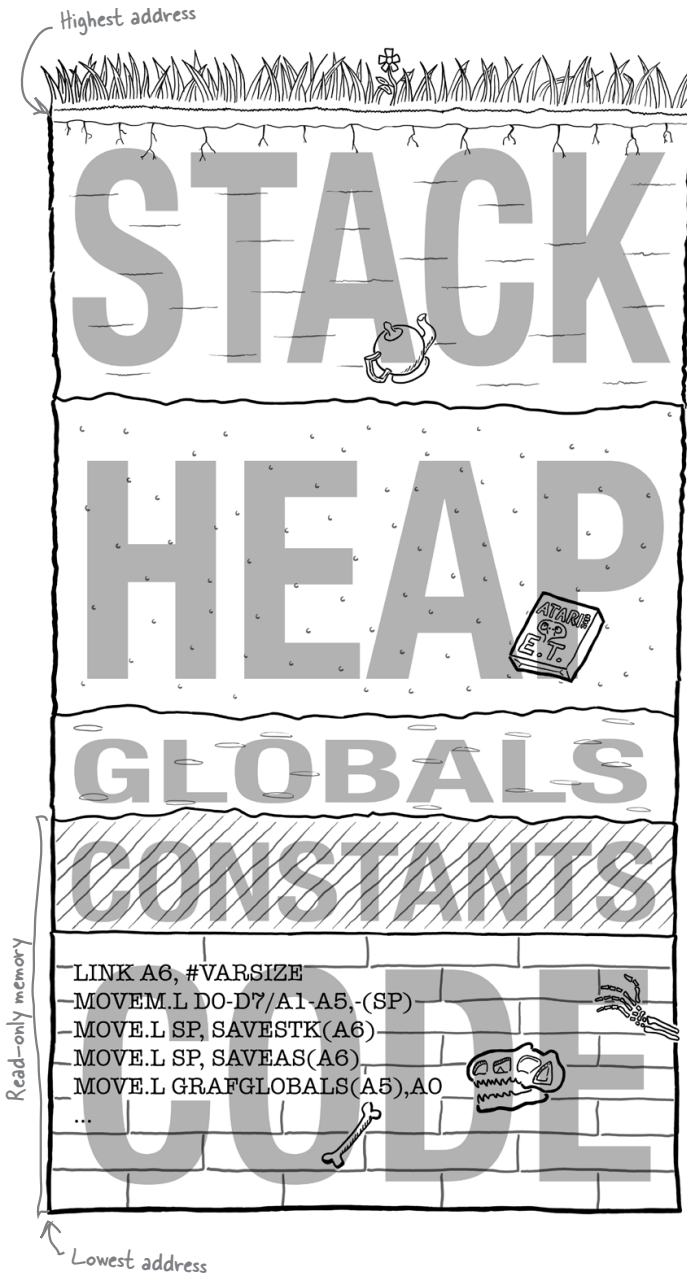
Une variable globale est créée et vit en dehors de toute fonction et est visible pour toutes les fonctions. Elles sont créées au début du programme et peuvent être modifiées quand vous le voulez. Mais ce n'est pas conseillé.

Les différentes partie en mémoire

Constantes

Les constantes sont aussi créées au début du programme mais elles sont stockées dans une zone de la mémoire en **lecture seule**. Ces constantes ne peuvent pas être modifiées.

Les différentes partie en mémoire



Le code

Pour terminer, le segment de code. Beaucoup de systèmes d'exploitation placent le code à la plus petite adresse mémoire possible. Cette partie de code est en lecture seule. C'est la partie de la mémoire où le code assembleur est chargé.

Pointeurs

Quelques définitions

Lvalue

tout objet pouvant être placé à gauche d'un opérateur d'affectation, il est caractérisé par une adresse mémoire et une valeur.

Pointeur

Un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. Déclaration :

```
type * nom_pointeur
```

Exemple

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

p=&i;
```

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

Pointeurs et affectation

L'opérateur * permet d'accéder directement à la valeur de l'objet pointé.

Exemple

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

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

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Le programme affiche 3 et toute modification de `*p` modifie `i`, `i` et `*p` sont identiques dans ce programme.

Autre exemple

Exemple

```
int i = 3, j = 6;
int *p1, *p2;
p1=&i;
p2=&j;
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

2 cas

`*p1=*p2;`

objet	adresse	valeur
i	4831836000	**6**
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

`p1=p2;`

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	**4831836004**
p2	4831835992	4831836004

Arithmétique des pointeurs

Opérations

- Addition d'un entier à un pointeur : le résultat est un pointeur de même type que le pointeur de départ ;
- Soustraction d'un entier à un pointeur : le résultat est un pointeur de même type que le pointeur de départ ;
- Différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

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 `--`.

```
int i = 3;
int *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld.\n",p1,p2);
```

Affiche :
`p1 = 4831835984 p2 = 4831835988.`

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

Exemple 2

Avec des doubles

```
double i = 3;
double *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n",p1,p2);
```

Affiche :
`p1 = 4831835984 p2 = 4831835992.`

Tableaux

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

Imprime les éléments du tableau `tab` dans l'ordre croissant puis décroissant des indices.

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)`.

Allocation dynamique

Initialisation

- Avant de manipuler un pointeur (*c-à-d* avant d'appliquer l'opérateur `*`) il faut l'initialiser.
- On peut utiliser la constante `NULL` définie dans `<stdio.h>` (elle vaut en général 0). Le test `p == NULL` permet de savoir si un pointeur `p` pointe vers un objet.
- Pour initialiser un pointeur `p`, on peut lui affecter l'adresse d'une variable `&i`, l'autre solution est d'affecter une valeur directement par *allocation dynamique* de manière à réserver un espace mémoire de taille adéquate :
  on réserve à `*p` un espace mémoire de taille suffisante, cet espace sera la valeur `p`.
  $\Rightarrow$ la fonction `malloc` de `stdlib.h`

La fonction `malloc`

- `void* malloc (size_t size)` où `size_t size` est un nombre d'octets - on utilisera en général la fonction `sizeof()` pour obtenir ce nombre de manière portable
- Elle retourne un pointeur de type non spécifié, elle permet les conversions implicites mais on prendra l'habitude de convertir explicitement les types - attention dans l'indication dans le poly correspond à d'anciens compilateurs

Malloc - exemple 1

On définit un pointeur p de type int, et affecte à *p la valeur de la variable i.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i = 3;
  int *p=NULL;

  printf("valeur de p avant initialisation = %ld\n",p);
  p = (int *) malloc(sizeof(int));
  printf("valeur de p apres initialisation = %ld\n",p);
  *p = i;
  printf("valeur de *p = %d\n",*p);

  return EXIT_SUCCESS;
}
```

Résultat à l'écran :

```
valeur de p avant initialisation = 0
valeur de p apres initialisation = 5368711424
valeur de *p = 3
```

Analyse - exemple 1

Avant l'allocation

objet	adresse	valeur
i	4831836000	3
p	4831836004	0

A ce stade `*p` n'a aucun sens et toute manipulation provoquerait une violation d'accès mémoire `Segmentation fault`.

L'allocation dynamique

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

A partir de `*p`, on a réservé 4 octets pour représenter un entier, non initialisé.

A la fin du programme

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

L'instruction `*p = i` permet d'affecter la valeur de `i` à `*p`.

Comparaison avec le programme précédent

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

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

Il correspond à la situation

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

- Ici `i` et `*p` sont identiques : elles ont la même adresse $\rightarrow$ toute modification de l'une modifie l'autre.
- Ce n'est pas vrai dans le programme précédent car `*p` et `i` ont la même valeur mais **des adresses différentes**
- Le programme ci-dessus ne nécessite pas d'allocation dynamique puisque l'espace-mémoire à l'adresse `&i` est déjà réservé pour un entier.

Allocation d'espace pour plusieurs objets contigus (tableaux)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i = 3;
  int j = 6;
  int *p;
  p = (int *) malloc(2 * sizeof(int));
  *p = i;
  *(p + 1) = j;
  printf("p = %ld,  *p = %d, p+1 = %ld, *(p+1) = %d, p[1] = %d.\n",
p,*p,p+1,*(p+1), p[1]);
  return EXIT_SUCCESS;
}
```

- On a ainsi réservé, à l'adresse donnée par la valeur de `p`, 8 octets en mémoire, qui permettent de stocker 2 objets de type `int`, soit un tableau de 2 entiers.
- Le programme affiche donc
  `p = 5368711424, *p = 3, p+1 = 5368711428, *(p+1) = 6, p[1] = 6.`
- $\rightarrow$ on vient de voir comment allouer des tableaux dynamiquement !

Autres fonctions d'allocations de `stdlib.h`

- `void * calloc(int nb_objets, size_t size)`
  Même rôle que `malloc` mais elle initialise en plus l'objet pointé (comprendre l'ensemble de l'espace mémoire) `*p` à zéro (`NULL`)
  `p = (int*)calloc(N,sizeof(int));` est strictement équivalent à

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
  *(p + i) = 0;
```

  L'emploi de `calloc` est simplement plus rapide.

- `void *realloc( void *pcourrant, size_t size );`
  Réalloue la mémoire à `*pcourant` de taille `size`, s'il n'y a pas assez de place, `realloc` essaie une nouvelle allocation de taille `size` et renvoie le nouveau pointeur, copie le contenu initialement pointé par `p` et libère cette zone mémoire, si l'allocation n'est toujours pas possible il renvoie `NULL` comme `malloc`.

```
p = realloc(p, 8*sizeof(int));
```

Libération mémoire et test

- `void free(void * pointeur)`
  Permet de libérer l'espace mémoire pointé par `*p`
  A toute instruction de type `malloc` ou `calloc`, on doit associer une instruction de type `free`.

- **Important :** l'allocation peut ne pas avoir marché (plus de mémoire disponible), il est donc capital de tester le pointeur alloué dynamique et de générer une erreur si l'allocation n'a pas fonctionné.

```
p = (int *) malloc (4 * sizeof(int));
/* on verifie si l'allocation a marche*/
if (p==NULL)
  {
    fprintf(stderr,"Impossible d'allouer la memoire %d, %d\n",4,
sizeof(int));
    exit(EXIT_FAILURE);
  }
```

⇒ Module `allocation.c`

Module allocation.c (1/2)

allocation_mem

```
void * allocation_mem(size_t nobjets,size_t taille)
{
    void * pt;
    pt = malloc (nobjets * taille);/*allocation*/
    /* on verifie si l'allocation a marche*/
    if (pt==NULL)
      mon_erreur("Impossible d'allouer la memoire %d %d\n",nobjets,taille);
    return(pt);
}
```

allocation_mem_init0 - version avec calloc

```
void * allocation_mem_init0(size_t nobjets,size_t taille)
{
    void * pt;
    pt = calloc (nobjets,taille);/*allocation avec calloc*/
    /* on verifie si l'allocation a marche*/
    if (pt==NULL)
      mon_erreur("Impossible d'allouer la memoire %d * %d\n",nobjets
,taille);
    return(pt);
}
```

Module allocation.c (2/2)

reallocation_mem - version avec realloc

```
void* reallocation_mem(void **pt, size_t nobjets,size_t taille)
{
     void *pt_realloc = realloc(*pt, nobjets*taille);
     if (pt_realloc != NULL)
         *pt = pt_realloc;
     else
       mon_erreur("Impossible de reallouer la memoire %d * %d a
l'adresse %p\n",nobjets,taille,*pt);

     return pt_realloc;
}
```

libere_mem - fonction de libération (version pédagogique)

```
void libere_mem_peda(void * *pt)
{
  if((*pt)!=NULL)
    free(*pt); /*liberation de *pt */

  *pt=NULL; /* *pt pointe maintenant sur NULL, cad rien*/
}
```

Le fichier allocation.h

```
#ifndef _ALLOCATION_H_
#define _ALLOCATION_H_

/*- ... */
void * allocation_mem(size_t nobjets,size_t taille);

/*- ... */
void * allocation_mem_init0(size_t nobjets,size_t taille);

/*- ... */
void* reallocation_mem(void **pt, size_t nobjets,size_t taille);

/*- ...  */
void libere_mem(void *pt);

/*- ...  */
void libere_mem_peda(void * *pt);

#endif
```

Une note sur `void *`

- Le type `void *` fait référence à tout type de pointeur, nous l'avons vu avec la fonctions `malloc` : `int **`, `char *`, `float ***`, ...
- Il faut donc généralement utiliser une conversion de type (cast) pour pouvoir utiliser un élément de type `void *` de sorte que le bon type soit associé à la variable.
- La fonction `libere_mem_peda` est un peu bizarre dans sa formulation en demandant un type `void **` (générateur de warning). Une version plus générique peut être définie de la manière suivante :

```
void libere_mem(void *pt)
{
  void ** adr_pt=(void **) pt;
  if((*adr_pt)!=NULL)
    free(*adr_pt); /*liberation de *pt */

  *adr_pt=NULL; /* *pt pointe maintenant sur NULL, cad rien*/
}
```

1. Il faut par contre bien appeler la fonction avec l'adresse du pointeur sur l'adresse mémoire que l'on souhaite libérer (en utilisant l'opérateur `&`.
2. **On utilisera plutôt cette version de `libere_mem`.**
   La fonction précédente peut être vue comme un outil pédagogique.

Pointeurs et tableaux - 1 dimension

Note : `tab` a pour valeur `&tab[0]`

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    {
      printf(" %d \n",*p);
      p++;
    }
  return EXIT_SUCCESS;
}
```

Tableaux à une dimension

Autre version - *(p+i)=p[i]

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

  return EXIT_SUCCESS;
}
```

La manipulation de tableaux a des inconvénients (un tableau est un pointeur constant)

- On ne peut pas créer de tableaux dont la taille est donnée par une variable
- on ne peut pas créer de tableaux bi(multi)-dimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

⇒ Mais c'est possible par allocation dynamique.

Création dynamique d'un tableau de taille variable

tableau à *n* éléments où *n* est une variable du programme

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

int main()
{
  int n;
  int *tab=NULL;
   ...
  tab = (int*)malloc(n * sizeof(int)); //(int *)allocation_mem(n,sizeof(int))
   ...
  free(tab); // libere_mem(&tab);
  ...
  return EXIT_SUCCESS;
}
```

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on utilise : `tab = (int*)calloc(n, sizeof(int));`
Les éléments de `tab` peuvent être manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux, ou avec l'opérateur `*`.

Différences importantes

Deux différences principales entre tableau et pointeur

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i;`
- un tableau n'est pas une `Lvalue`, il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++;`), par contre les pointeurs oui - **mais attention à ce que vous faites !**.
- Implication importante pour l'appel de fonctions avec tableaux à plusieurs dimensions ! (cf plus tard)

On reviendra sur ce sujet plus tard.

Pointeurs et tableaux à plusieurs dimensions

- Tableau à deux dimensions : lorsqu'ils sont manipulés par des pointeurs ils correspondent à un tableau de tableaux et donc un pointeur vers un pointeur
- Exemple avec un tableau de taille fixée : `int tab[M][N];` :
  - `tab` peut être associé à un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier.
  - `tab` a une valeur constante égale à l'adresse du premier élément du tableau, `&tab[0][0]`.
  - `tab[i]`, pour `i` entre `0` et `M-1`, peut être vu comme un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice `i`. `tab[i]` a donc une valeur constante qui est égale à `&tab[i][0]`.

Pointeur de pointeurs

- Objet à 2 dimensions de type `type` : `type **nom-du-pointeur;`
- Objet à 3 dimensions de type `type` : `type ***nom-du-pointeur;` et ainsi de suite.

Exemple matrice d'entiers à k lignes et n colonnes à partir de pointeur de pointeur

Allocation en 2 temps : les lignes puis les colonnes de chaque ligne

```
int main() {
  int k, n;
  int **tab=NULL;

  tab = (int**)malloc(k * sizeof(int*));
  for (i = 0; i < k; i++)
    tab[i] = (int*)malloc(n * sizeof(int)); //(int *)allocation_mem(n,sizeof(int))
      ....

  for (i = 0; i < k; i++)
    free(tab[i]); // libere_mem(&tab[i])

  free(tab); // libere_mem(&tab)
  return EXIT_SUCCESS;
}
```

La première allocation de `tab` crée `k` pointeurs sur des entiers, ces pointeurs correspondent aux lignes de la matrice. On réserve ensuite pour chaque `tab[i]` les n entiers pour avoir les colonnes.

Exemple - Matrices - suite

Si on veut que les éléments du tableau soient initialisés à 0

```
tab[i] = (int *) calloc(n,sizeof(int));
```

ou avec le module `allocation.c` :

```
tab[i] = allocation_mem_init0(n,sizeof(int));
```

Des tailles de colonnes différentes pour chaque ligne

Si on veut que `tab[i]` contienne exactement `i+1` éléments, on peut écrire :

```
for(i = 0; i< k; i++)
  tab[i]=(int *) malloc((i+1)*sizeof(int));
```

ou avec le module `allocation.c` :

```
for(i = 0; i< k; i++)
  tab[i]=(int *) allocation_mem(i+1,sizeof(int));
```

Attention il ne faut pas oublier de vérifier que l'allocation a marché !

Pointeurs et chaînes de caractères

On sait qu'une chaîne de caractères est un tableau à une dimension d'objets de type `char`, se terminant par le caractère nul `'\0'`. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`.

- `char * chaine = NULL;`
- `chaine = "ceci est une chaine"; //l'allocation est "automatique"`

Exemple : affichage du nb de caractères d'une chaîne

```
#include <stdio.h>
int main()
{
  int i;
  char *chaine;
  chaine = "chaine de caracteres";
  for (i = 0; *chaine != '\0'; i++)
    chaine++;
  printf("nombre de caracteres = %d\n",i);
  return EXIT_SUCCESS;
}
```

Note : à cause de l'instruction `chaine++;` la chaîne est perdue ici.

Exemple : concaténation de 2 chaînes (avec une erreur)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
  int i;
  char *chaine1=NULL, *chaine2=NULL, *res=NULL, *p=NULL;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";

  res = (char*)allocation_mem((strlen(chaine1) + strlen(chaine2)),
sizeof(char));
  p = res;
  for (i = 0; i < strlen(chaine1); i++)
    { *p = chaine1[i]; p++; }
  for (i = 0; i < strlen(chaine2); i++)
    { *p = chaine2[i]; p++; }
  printf("%s\n",res);

  return EXIT_SUCCESS;
}
```

Exemple : concaténation de 2 chaînes - NE PAS OUBLIER LE '\0'

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
  int i;
  char *chaine1=NULL, *chaine2=NULL, *res=NULL, *p=NULL;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";

  res = (char*)allocation_mem((strlen(chaine1) + strlen(chaine2)+1),
sizeof(char));
  p = res;
  for (i = 0; i < strlen(chaine1); i++)
    { *p = chaine1[i]; p++; }
  for (i = 0; i < strlen(chaine2)+1; i++)
    { *p = chaine2[i]; p++; }
  printf("%s\n",res);

  return EXIT_SUCCESS;
}
```

Exemple suite - attention si l'on se passe d'un pointeur ...

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
  int i;
  char *chaine1=NULL, *chaine2=NULL, *res=NULL;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";

  res = (char*)allocation_mem((strlen(chaine1) + strlen(chaine2)+1),
sizeof(char));
  for (i = 0; i < strlen(chaine1); i++)
    { *res = chaine1[i]; res++;}
  for (i = 0; i < strlen(chaine2)+1; i++)
    *res++ = chaine2[i];
  printf("\nnombre de caracteres de res = %d\n",strlen(res));
  return EXIT_SUCCESS;
}
```

$\Rightarrow$ imprime la valeur 0, puisque `res` a été modifié au cours du programme et pointe maintenant sur le caractère nul.

Exemple d'affichage inversé de chaîne avec pointeur

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

void affiche_inv_recu(char * chaine)
{
   if(*chaine!='\0')
   {
      affiche_inv_recu(chaine+1);
      printf("%c",*chaine);
   }
}

int main(int argc,char * argv[])
{
   if(argc==2)
   {
      char * pt_chaine=argv[1];
      affiche_inv_recu(pt_chaine);
      printf("\n");
   }
   return EXIT_SUCCESS;
}
```

Note sur scanf et const

scanf

- Hormis pour écrire des petits programmes simples d'illustration, il est en général déconseillé d'utiliser `scanf`
- Si vous demandez un entier `%d` et que l'utilisateur commence par écrire, le `scanf` risque de se "bloquer", ceci peut provoquer des dépassements de *"tampon"* (buffer)
- On préférera utiliser `fgets` ;

```
char buffer[256];
...
fgets(buffer, 256*sizeof(char), stdin);
```

⇒ lecture au maximum des 256 premiers caractères sur l'entrée standard puis copie dans `buffer`. Grâce au maximum on évite tout dépassement.

const

- sert à indiquer une constante ne pouvant pas être modifiée

```
const int x = 10;
const double t[3] = {10,11,12};
...
int f(const int * t, const char c, int a);
```

⇒ Ecrire `t[1]=3;` ou `c=4;` provoquera une erreur (souvent bus error)

Note sur sizeof

- S'applique sur des types (`int`, `double`, ...), une expression, ou un tableau :

```
double t[10];
sizeof(t) // Taille de t en octets
10 * sizeof(double) // 10 fois taille double
sizeof(double[10]) // taille tableau de double
10 * sizeof(t[0]) // 10 fois taille 1er element
```

- **Attention :** dans le contexte d'une fonction, ce n'est plus vrai !

```
#include <stdio.h>
void ma_fonction(int * t) {
  int nb = sizeof(t) / sizeof(t[0]);
  printf("nb=%i\n", nb);
}
int main() {
  double t[10];
  int nb = sizeof(t) / sizeof(t[0]);
  printf("nb=%i\n", nb);
  ma_fonction(t);
  ...
}
```

$\Rightarrow$ affiche `nb=10` puis `nb=1` !

Tableaux à plusieurs dimensions et passage d'arguments

Tableaux ≪ constants ≫

`int tab[2][3];`
Accès à l'élement `i,j` : `tab+i*3*sizeof(int)+j`
Il faut connaître la dimension du dernier argument

Pointeurs

```
int **p;  //il faudra rajouter une allocation
```
Accès à l'élement `i,j` :* (*(p+i)+j)
Pas besoin de connaître la dimension du 2eme argument, on passe pour les adresses.

Il faut prendre en compte cet aspect dans les appels de fonction (cf correction TD3).
Pour un tableau ≪ constant ≫ à N dimensions, il faut indiquer les N-1 dernières dimensions lors d'un passage d'arguments.

Différence entre tableaux et pointeurs

Ce qui dit la norme c90

Sauf quand elle est l'opérande de l'opérateur `sizeof` ou de l'opérateur unaire `&`, ou est une `chaine de caractères littérale` utilisée pour initialiser un tableau, une expression de type « *tableau de type* » est convertie en une expression de type « *pointeur de type* » qui pointe sur l'élément initial de l'objet tableau et n'est pas une `lvalue`. Si le tableau est d'une classe de stockage registre (mot clé `register`, le comportement est indéterminé.

Conséquence 1

```
int tab[3];
int * p;
p=&tab;
```

L'affectation ci-dessus est correcte : `p` pointe sur un tableau de 3 entiers

Différence entre tableaux et pointeurs

Conséquence 2

```
char *p = "hello";
chat t1[] = "world";
chat t2[] = p;                  /* Faux */
chat t3[] = (char *) "titi"; /* Faux */
```

Les deux dernières lignes sont fausses car un tableau ne peut pas être initialisé avec une valeur scalaire.
`t1` par contre est un tableau de 6 caractères initialisé avec la chaîne *world*.
Le pointeur `p` récupère l'adresse de la chaîne constante, on ne peut donc plus modifier son contenu, pour être plus propre on pourrait écrire

```
const char *p= "hello";
```

Pour le problème de passage de tableau à plusieurs dimensions, cf le TD.
Les points liés au cas `register` ne seront pas traités dans ce cours - il signifie que la variable doit être stockées dans le registre du processeur et dans la mémoire vive - dans le contexte de tableau le comportement est indéterminé par rapport à la relation avec les pointeurs (ça n'a pas forcément de sens déterminé ici).

Structure de données

Définition

Assemblage de données de types éventuellement distincts

Motivation : Regrouper des informations de types différents

Exemple

Un étudiant est défini par :

- un nom
- un (ou des) prénom(s)
- une adresse
- une date de naissance
- l'année d'inscription
- ...

Définition

Suite finie d'objets de types différents. Déclaration (notez le ; à la fin) :

```
struct modele
{
  type_1 membre_1;
  type_2 membre_2;
  ...
  type_n membre_n;
};
```

On peut ensuite utiliser la structure comme nouveau type :
`struct modele nom_variable;`

Accès

On accède aux différents membres d'une structure grâce à l'opérateur `.` :
`nom_variable.membre_i;`

Exemple classique

```
#include <math.h>
struct complexe
{
  double reelle;
  double imaginaire;
};
int main() {
  struct complexe z;
  double norme;
   ...
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
  return EXIT_SUCCESS;
}
```

Option d'écritures

- Déclaration + initialisation : `struct complexe z = {2., 2.};`
- On peut appliquer l'opérateur d'affectation à des structures (implique une recopie des champs)

```
struct complexe z1, z2;
...
z2 = z1;
```

Les types composés avec typedef

Déclaration

```
typedef type nom;
```

Exemple avec 2 solutions possibles

```
struct complexe
{
  double reelle;
  double imaginaire;
};

typedef struct complexe complexe;

int main() {
  complexe z;
   ...
}
```

```
struct struct_complexe
{
  double reelle;
  double imaginaire;
};

typedef struct struct_complexe complexe;

int main() {
  complexe z;
   ...
}
```

Un autre exemple

```
struct date {
   int an;
   short mois, jour;
} ;
typedef struct date date;
struct personne {
   char nom[20], prenom[20];
   date naissance;
} ;
typedef struct personne etudiant;
```

Il existe une écriture simplifiée :

```
typedef struct  {
   int an;
   short mois, jour;
} date ;
typedef struct {
   char nom[20], prenom[20];
   date naissance;
}  etudiant;
```

Pour accéder aux différents champs d'une variable de type étudiant :

```
etudiant etu; /* declaration d'une variable de type etudiant */
etu.nom="Dupond";
etu.prenom="Marcel";
etu.naissance.annee=1995;
etu.naissance.mois=3;
etu.naissance.jour=21;
```

Autre type : les unions (1/2)

Définition

Une union désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Exemple/déclaration

```
union jour {
  char lettre;
  int numero;
};
main() {
  union jour hier, demain;
  hier.lettre = 'J';
  printf("hier = %c\n",hier.lettre);
  hier.numero = 4;
  demain.numero = (hier.numero + 2) % 7;
  printf("demain = %d\n",demain.numero);
}
```

Autre type : les unions (2/2)

Autre exemple

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type `unsigned int` d'une structure en les identifiant à un objet de type `unsigned long` (en supposant que la taille d'un entier long est deux fois celle d'un int).

```
struct coordonnees
{
  unsigned int x;
  unsigned int y;
};
union point {
  struct coordonnees coord;
  unsigned long mot;
};
int main() {
  union point p1, p2, p3;
  p1.coord.x = 0xf;
  p1.coord.y = 0x1;
  p2.coord.x = 0x8;
  p2.coord.y = 0x8;
  p3.mot = p1.mot ^ p2.mot;
  printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

Autre type : les énumérations

Définition

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur, suivis de la liste des valeurs que peut prendre cet objet :

```
enum modele {constante-1, constante-2,...,constante-n};
```

Exemple

En pratique les objets de type `enum` sont souvent représentés comme des `int` (les valeurs possibles étant codées par des entiers). Par exemple, dans le programme suivant le type `booleen` associe l'entier 0 à `faux` et l'entier 1 à `vrai`.

```
int main(){
  enum booleen {faux, vrai};
  enum booleen b;
  b = vrai;
  printf("b = %d\n",b);
}
```

On peut aussi modifier le codage par défaut des valeurs lors de la déclaration :

```
enum booleen {faux = 12, vrai = 23};
```

Créations de types, structures et pointeurs

Pointeur et structure : on peut assigner des pointeurs sur des structures

```
#include <stdio.h>
#include <stdlib.h>
struct eleve{
  char nom[20];
  int date;
};
typedef struct eleve * classe;

int main() {
  int n, i;  classe tab;
  printf("nombre d'eleves de la classe = ");
  scanf("%d",&n);
  tab = (classe)allocation_mem(n, sizeof(struct eleve));
  for (i =0 ; i < n; i++){
      printf("\n saisie de l'eleve numero %d\n",i);
      printf("nom de l'eleve = ");
      scanf("%s",tab[i].nom);
      printf("\n date de naissance JJMMAA = ");
      scanf("%d",&tab[i].date);
    }
  printf("\n Entrez un numero  ");
  scanf("%d",&i);
  printf("\n Eleve numero %d:\n nom = %s",i,tab[i].nom);
  printf("\n date de naissance = %d\n",tab[i].date);
  free(tab);
  return EXIT_SUCCESS;
}
```

Pointeurs et structures : accès

Accès à un membre d'une structure pointée par un pointeur `p`

```
(*p).membre
```

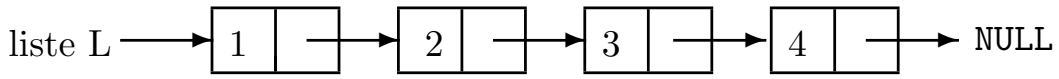
L'usage de parenthèses est indispensable. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté `->`, l'expression précédente est ainsi strictement équivalente à :

```
p-> membre
```

Dans le programme précédent, on peut ainsi remplacer `tab[i].nom` et `tab[i].date` respectivement par `(tab + i)->nom` et `(tab + i)->date`.

Le type liste : structure auto-référencée

- Besoin de modèles de structure dont un des membres est un pointeur vers une structure de même modèle.
- Cette représentation permet en particulier de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur).
- Pour éviter de fixer une taille a priori, on utilise une représentation chaînée : l'élément de base de la chaîne est une structure appelée cellule qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`.
- La liste est alors définie comme un pointeur sur le premier élément de la chaîne.



Définition d'une cellule

```
typedef struct cellule
{
  int valeur;
  struct cellule * suivant;
}struct_cellule;
typedef struct_cellule * liste;
```

Liste suite

Insertion d'un élément en tête de liste

```
liste insere(int element, liste l)
{
  liste newcell=liste_vide();
  newcell=(liste)allocation_mem(1,sizeof(struct_cellule));
  newcell->valeur = element;
  newcell->suivant = l;
  return newcell;
}
```

Fonction de création d'une liste vide

```
liste liste_vide()
{
  return NULL;
}
```

Exemple d'utilisation dans un main

```
int main() {
  liste l, p;

  l = insere(1,insere(2,insere(3,insere(4,liste_vide()))));

  printf("\n impression de la liste:\n");
  p = l;
  while (p != liste_vide())
  {
      printf("%d \t",p->valeur);
      p = p->suivant;
  }
  printf("\n");

  return EXIT_SUCCESS;
}
```

TAD (Type Abstrait de Données) - ex : listes

TAD

Définition d'un type (représentation) et de l'ensemble des fonctionnalités pour le manipuler

Les objets

- La notion de cellule
- La notion de liste

Les fonctionnalités

- liste vide : renvoie une liste
- savoir si une liste est vide : prend une liste et renvoie un booléen (int)
- ajouter un élément à une liste : prend une liste et un élément et renvoie la nouvelle liste
- extraire le premier élément de la liste : prend une liste et renvoie un élément
- supprimer une cellule d'une liste : prend une liste et renvoie la nouvelle liste

(on peut en ajouter d'autre : rechercher, parcourir, détruire, insérer au début/au milieu/à la fin, trier, ...)

TAD - Le fichier `liste.h` (rajouter les `#ifndef et #endif`)

```
#include <stdio.h>
#include "allocation.h"

typedef int element;

typedef struct cellule{
 element objet;
 struct cellule * suivant;
}struct_cellule;
typedef struct_cellule * liste;

/* renvoie la liste vide */
liste liste_vide();
/* teste si une liste est vide */
int est_liste_vide(liste l);
/*ajoute un element elem a la liste l */
liste inserer_element_liste(liste l, element elem);
/* renvoie le premier element de la liste l */
element renvoie_premier_liste(liste l);
/* supprime la premiere cellule de la liste l */
liste supprimer_premier_liste(liste l);
```

Les fonctions du TAD (`liste.c` 1/3)

liste vide

```
liste liste_vide()
{
  return NULL;
}
```

savoir si une liste est vide

```
int est_liste_vide(liste l)
{
  if(l==liste_vide())
    return 1;
  return 0;
}
```

Les fonctions du TAD (liste.c 2/3)

inserer un element (au début de la liste)

```
liste inserer_element_liste(liste l,element elem)
{
  liste lnew=(liste)allocation_mem(1,sizeof(struct_cellule));
  lnew->objet=elem;
  lnew->suivant=l;
  return lnew;
}
```

(cf schéma au tableau)

extraire le premier élément de la liste

```
element renvoie_premier_liste(liste l)
{
  if(est_liste_vide(l))
   mon_erreur("Erreur la liste est vide dans la fonction
renvoie_premier\n");
  return l->objet;
}
```

Les fonctions du TAD (liste.c 3/3)

supprimer une cellule (au début de la liste)

```
liste supprimer_premier_liste(liste l)
{
  liste lsuivant=l->suivant;
  libere_mem(&l);
  return lsuivant;
}
```

Exercices :

1. écrire une fonction de suppression qui peut supprimer un élément (donné en argument) au milieu de la liste
2. ré-écrire les fonctions d'insertion et suppression de manière à modifier la liste donnée en argument de manière adéquate.

Type Abstrait de Données (TAD) - pile

Objet de type LIFO : Last In First Out.

TAD Pile

- Créer une pile vide
- Empiler un nouvel élément
- Récupérer l'élément au sommet de la pile
- Dépiler le premier élément
- Connaître le nombre d'éléments de la pile

pile.h - On utilise le type Liste pour implémenter une pile

```
#ifndef _PILE_H_
#define _PILE_H_

#include "liste.h"

typedef liste pile;

pile pile_vide();

int est_pile_vide(pile p);

pile empiler(pile p,element e);

element sommet(pile p);

pile depiler(pile p);

int taille_pile(pile p);
#endif
```

pile.c (1/2)

```
#include <stdio.h>
#include "pile.h"

pile pile_vide()
{
  return liste_vide();
}

int est_pile_vide(pile p)
{
  return est_liste_vide(p);
}

pile empiler(pile p,element e)
{
  return inserer_element_liste(p,e);
}
```

pile.c (2/2)

```
element sommet_pile(pile p)
{
  return renvoie_premier_liste(p);
}

pile depiler(pile p)
{
  return supprimer_premier_liste(p);
}

int taille_pile(pile p)
{
  int nb=0;
  liste lcourant=p;
  while(lcourant)
  {
     nb++;
     lcourant=lcourant->suivant;
  }
  return nb;
}
```

Exercice

1. ré-écrire les fonctions d'insertion et suppression de manière à modifier la liste donnée en argument de manière adéquate

Type Abstrait de Données (TAD) - file

Objet de type FIFO : First In First Out.

TAD File

- Créer une file vide
- enfiler un nouvel élément
- Récupérer l'élément au début de la file
- défiler le premier élément
- Connaître le nombre d'éléments de la file

file.h - on utilise encore le type Liste pour implémenter une file

```
#ifndef _FILE_H_
#define _FILE_H_

#include "liste.h"

typedef liste file;

file file_vide();

int est_file_vide(file f);

file enfiler(file f,element e);

element debut_file(file f);

file defiler(file f);

int taille_file(file f);
#endif
```

file.c (1/3)

```
#include <stdio.h>
#include "file.h"

file file_vide()
{
  return liste_vide();
}

int est_file_vide(file f)
{
  return est_liste_vide(f);
}

file defiler(file f)
{
  return supprimer_premier_liste(f);
}
```

file.c (2/3)

```
element debut_file(file f)
{
  return renvoie_premier_liste(f);
}

int taille_file(file f)
{
  int nb=0;
  liste lcourant=f;

  while(lcourant)
  {
     nb++;
     lcourant=lcourant->suivant;
  }

  return nb;
}
```

file.c (3/3) - plus de travail : insérer à la fin

```
file enfiler(file f,element e)
{
  liste lnew=inserer_element_liste(liste_vide(),e);
  liste lcourant=f;

  if(est_file_vide(f))
    return lnew;

  while(lcourant->suivant!=liste_vide())
  {
    lcourant=lcourant->suivant;
  }

  lcourant->suivant=new;

  return f;
}
```

Exercices

1. Ré-écrire les fonctions d'insertion et suppression de manière à modifier la liste donnée en argument de manière adéquate (facile, c'est comme pour les piles).
2. Implémenter une file à l'aide de 2 pointeurs : un sur le début de la file et un sur le dernier élément de la file (permet de faciliter l'écriture de la fonction enfiler)

File avec 2 pointeurs de liste - Fichier h

```
#ifndef _FILE_H_
#define _FILE_H_
#include "liste.h"

typedef struct cell_file{
  liste debut;
  liste fin;
}cell_file;
typedef cell_file * file;

file file_vide();

int est_file_vide(file f);

file enfiler(file f,element e);

element debut_file(file f);

file defiler(file f);

int taille_file(file f);
#endif
```

file.c (1/4)

```
#include <stdio.h>
#include "file.h"
#include "allocation.h"

file file_vide()
{
  file f=(file)allocation_mem(1,sizeof(cell_file));
  f->debut=liste_vide();
  f->fin=liste_vide();
  return f;
}

int est_file_vide(file f)
{
  return (est_liste_vide(f->debut) && est_liste_vide(f->fin));
}

element debut_file(file f)
{
  return renvoie_premier_liste(f->debut);
}
```

file.c (2/4) - taille file

```
int taille_file(file f)
{
  int nb=0;
  liste lcourant=f->debut;

  while(lcourant)
  {
     nb++;
     lcourant=lcourant->suivant;
  }

  return nb;
}
```

file.c (3/4) - suppression

```
/*supprimer l'element en tete de file */
file defiler(file f)
{
  if(est_file_vide(f))
      return f;

  f->debut=supprimer_premier_liste(f->debut);

  if(est_liste_vide(f->debut))
    f->fin=liste_vide();

  return f;
}
```

file.c (4/4) - insertion

```
file enfiler(file f,element e)
{
  liste lnew=insere_element_liste(liste_vide(),e);

  if(est_file_vide(f))
    f->debut=lnew;
  else f->fin->suivant=lnew;

  f->fin=lnew;

  return f;
}
```

file.c (annexe) - autre prototype de suppression

Si on change le prototype de defiler de manière à renvoyer l'élément à supprimer, on doit ajouter le prototype suivant dans le fichier d'en-tête :
`element defiler(file f);`

La fonction peut alors être définie de la manière suivante, nous devons par contre gérer un cas particulier ne cas de file vide, on propose de renvoyer une constante appelée `ELEMENT_VIDE` à définir dans le fichier d'en-tête (par exemple `-1`).

```
element defiler_av_element(file f)
{
  element e;

  if(est_file_vide(f))
      return ELEMENT_VIDE;

  e=f->debut->objet;

  f->debut=supprimer_premier_liste(f->debut);

  if(est_liste_vide(f->debut))
      f->fin=liste_vide();

  return e;
}
```