UNIVERSITÉ DE NANTES

POLYTECH NANTES

Langage procédural (C)

Partie 2

Support de Cours

Dpt Informatique Polytech'Nantes

Fabien Picarougne

UNIVERSITÉ DE NANTES

POLYTECH NANTES

Partie II

Langage C avancé

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

Préprocesseur

- Utilité du pré-processing
  - Le préprocesseur est un programme qui analyse un fichier texte et qui lui fait subir certaines transformations
    - Inclusion d'un fichier
    - Suppression/remplacement d'une zone de texte
  - Appelé automatiquement par le compilateur
    - Avant la compilation
      - Traitement sur les fichiers sources
    - Pour chaque fichier à compiler
  - Toutes les commandes (directives) du préprocesseur commencent
    - En début de ligne
    - Par le caractère `#`
      - Ex : `#define`

Préprocesseur

- Macros et constantes symboliques
  - Le préprocesseur parcourt le texte de chaque fichier source
  - Il n'interprète pas les données qu'il lit
    - Considère le fichier uniquement comme du texte

  - Constantes symboliques
    - Utilisation du préprocesseur pour définir une constante symbolique
      - Définition de chaînes de remplacement

        - `#define texte_a_remplacer texte_de_remplacement`
        - Pas de vérification de types comme avec le `const` ou le `enum`
          - Le compilateur agit après le préprocesseur
        - Ex :

```
#define TVA 20.0

printf("Prix final : %.2f\n", prix*TVA/100);
// équivalent à
printf("Prix final : %.2f\n", prix*20.0/100);
```

Préprocesseur

- Remplace le texte sans interprétation
  - Utilisation pour n'importe quel type
  - Ex :

```
#define PI 3.141529
#define YES 'y'
#define BEGIN {
```

- Texte de remplacement sur plusieurs lignes
  - Nécessité d'utiliser \
  - Ex :

```
#define MESSAGE "mon message est beaucoup trop long pour tenir\
     sur une seule ligne"
```

- Par convention
  - Texte à remplacer en majuscule
  - Mais rien ne l'y oblige
- Non remplacement lorsque
  - La constante symbolique est incluse dans une chaîne de caractère
  - La constante symbolique est incluse dans un autre libellé
    - Ex :

```
#define VALEUR 1
int nombreVALEUR;
```

- Constante symbolique sans texte de remplacement
  - Sert principalement à établir un marquage pour la suite
  - `#define UNIX` est valide
    - Supprime toute occurrence de `UNIX`

Préprocesseur

- Suppression d'une constante symbolique

```
#undef texte_a_remplacer
```

- Le préprocesseur lit le fichier source de haut en bas
  - Possibilité de définir en cours de fichier une constante symbolique
  - Possibilité de supprimer en cours de fichier une constante symbolique

- Ex :

```
#define PRINT printf
int main()
{
    PRINT("Cette instruction utilise la constante symbolique PRINT");
#undef PRINT
    PRINT("Cette instruction engendre une erreur");
}
```

Préprocesseur

- Macros
  - Une macro représente une suite de commandes élémentaires
  - Un nom = abréviation de cette suite de commande
  - Défini via des constantes symboliques
    - Directive `#define`

  - Macros sans paramètres
    - Ex :

```
#define CLS printf("\033[2J");   // Efface l'écran
int main()
{
   CLS
}

// Est compilé comme
int main()
{
   printf("\033[2J");
}
```

Préprocesseur

- Macros composées de plusieurs instructions
  - Seule limite à la constante symbolique : fin de ligne
  - Ex :

```
#define ERRORHANDLER {printf("\nDivision par 0 interdite.\
<Entrée> pour continuer.");getch();continue;}
```

  - Encadrer les instructions par un bloc permet de considérer la suite d'instruction comme une seule instruction

- Macros imbriquées
  - À chaque remplacement de texte, le préprocesseur ré-analyse la nouvelle chaîne
  - Ex :

```
#define INTRO {printf("\033[2J");printf("\ERREUR !\n");}
#define ERRORHANDLER {INTRO printf("\nDivision par 0 interdite.\
<Entrée> pour continuer.");getch();continue;}
```

Préprocesseur

- Macro avec paramètres
  - Les macros, comme les fonctions, peuvent avoir des paramètres
  - Syntaxe : `#define nom(param1,param2,…,paramN) Texte_de_remplacement`
    - Nom de macro suivi directement des paramètres
  - Ex :

```
#define SUB(x,y) x-y
```

    - Remplacement des paramètres dans le texte de remplacement
  - Attention à bien encadrer les paramètres de parenthèses !
  - Ex :

```
#define MUL(x,y) x*y
int main()
{
   MUL(3-2,4);          // Équivaut à 3-2*4 et non (3-2)*4
}
```

  - Toujours encadrer les macros par des parenthèses pour ne pas avoir d'erreur de priorité
  - Ex :

```
#define SUB(x,y) ((x)-(y))
```

Préprocesseur

- Macro avec paramètres et chaînes de caractères
  - Un paramètre de macro peut apparaitre dans une chaîne de caractère
  - Utilisation d'un opérateur spécial `#` (*stringizing operator*)
  - Ex :

```
#define SHOW(p) printf("p");
SHOW(testvalue) // Affiche « p »
```

  - `testvalue` n'a pas été transformée dans la chaîne de caractère
    - Utilisation de l'opérateur `#`
  - Ex :

```
#define SHOW(p) printf(#p);
SHOW(testvalue) // Affiche « testvalue »
```

  - `#p` transforme le paramètre p en une chaîne de caractère
- Opérateur de concaténation de mots `##`
  - Permet de concaténer des chaînes de caractère dans une macro
  - Ex :

```
#define TOK(t1,t2) t1##t2
int TOK(number,6) // Équivaut à int number6;
```

Préprocesseur

- Inclusion de fichiers
  - Le préprocesseur permet de réutiliser du code via les directives d'inclusion
  - `#include` permet de recopier le contenu d'un fichier à la place de la directive
  - Deux syntaxes sont disponibles
    - `#include <fichier>`
      - Le préprocesseur recherche le fichier dans les répertoires prédéfinis
    - `#include "fichier"`
      - Le préprocesseur recherche le fichier à partir du répertoire courant du fichier analysé
  - Usage de l'inclusion
    - Il est possible d'inclure tout type de fichier
    - Mais il est EXTREMEMENT recommandé de n'inclure que des définitions
      - Regroupées au sein d'un fichier *header* (extension *.h*)
      - Ex : définition du prototype des fonctions
    - Attention à l'ordre d'inclusion
      - Ne pas redéfinir 2 fois le même prototype -> erreur !
      - Utilisation de la compilation conditionnelle

Préprocesseur

- Programme typique
  - Regroupement des déclarations de fonctions dans un fichier header (extension .h)
  - Inclusion du fichier dans le ficher source (extension .c) correspondant

monfichier.c

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

int main()
{
    int i;

    for (i=0;i<10;i++)
        Affiche(i);
}

void Affiche(int nb)
{
    printf("Le nombre est
%d\n",nb);
}
```

monfichier.h

```
void Affiche(int nb);
```

Préprocesseur

- Compilation conditionnelle
  - De même que l'exécution des instructions C, la compilation de portions de programme peut être dépendante de certaines conditions
  - Utilisation de primitives de précompilation

```
#if
#ifdef    // Exécuté uniquement si l'élément suivant est défini via #define
#ifndef   // Exécuté uniquement si l'élément suivant n'est pas défini via
             #define
#elif
#else
#endif
```

  - Permet de bâtir des constructions semblables aux structures normales if-else
  - Ex :

```
#if EXPRESSION
    // Portion de programme
#elif EXPRESSION
    // Portion de programme
#else
    // Portion de programme
#endif
```

Préprocesseur

- Exemple d'utilisation : contrôle des *include*
  - Permet de simplifier le développement
  - Inclusion d'un fichier dès que l'on doit utiliser une fonction

monfichier.c

```
#include "header1.h"
#include "header2.h"

...
```

header1.h

```
#ifndef _header1_h
    #define _header1_h

#include "header2.h"

void Affiche(int nb);
...

#endif
```

header2.h

```
#ifndef _header2_h
    #define _header2_h

void Draw(void);
...

#endif
```

Préprocesseur

- Messages et *pragma*
  - Il est possible de signaler au compilateur une erreur via la directive `#error`
  - Ex :

```
#if TRUE==0
    #error TRUE doit être défini avec une valeur non nulle
#endif
```

  - Le message est renvoyé lors de la compilation et non lors de l'exécution !
    - Il existe des constantes prédéfinies accessibles au précompilateur

```
_LINE_// Numéro de ligne du fichier source traité
_FILE_// Nom du fichier source traité
_DATE_// Date de la compilation
_TIME_// Heure de la compilation
_STDC_// Défini si le compilateur respecte le standard ANSI-C
```

  - Il est possible de donner des instructions au compilateur depuis le fichier source via la directive `#pragma`
    - Dépendant du compilateur
    - Utile pour inclure des librairies à la volée sans manipuler une configuration de projet
    - Ex :

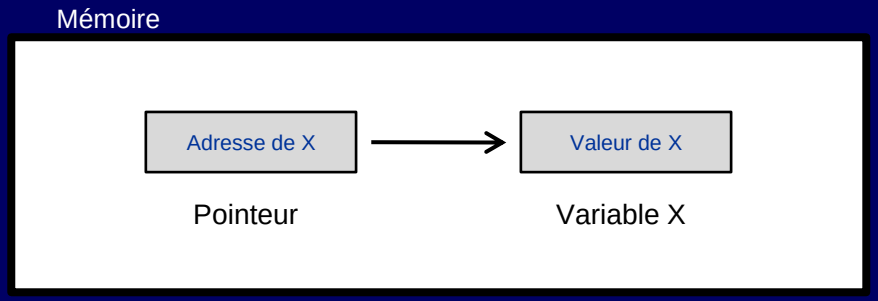
```
#pragma once
#pragma (lib,"lib.a")
```

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

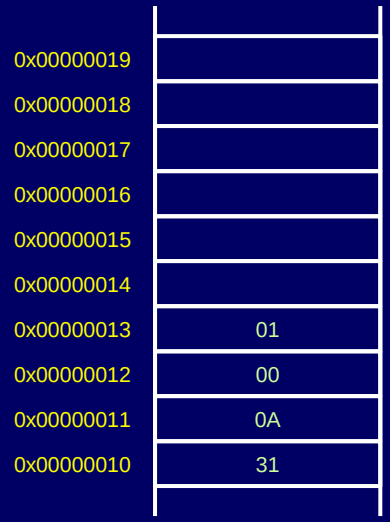
Pointeurs

- Qu'est-ce qu'un pointeur ?
  - Donnée constante ou variable
  - Mémorise l'adresse d'une variable
    - Renvoi ou pointe vers la variable concernée



- Variable et adresse
  - L'opérateur unaire `&` permet d'accéder à l'adresse mémoire d'une variable
  - Ex :

```
int x=0x01000A31;
&x;        // accède à l'adresse de la variable x
           // valeur : 0x00000010
```

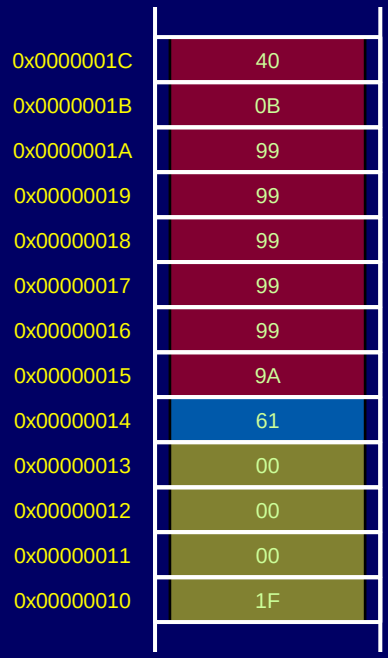


Pointeurs

- Homogénéité des adresses
  - Quelque soit le type de la variable pointée la taille de l'adresse est identique
  - Ex :

```
int i=31;          // 4 octets
char c='a';        // 1 octet
double d=3.45      // 8 octets

&i                 // 4 octets (0x00000010)
&c                 // 4 octets (0x00000014)
&d                 // 4 octets (0x00000015)
```

  - Le type du pointeur permet au compilateur d'interpréter la taille de l'élément pointé



Pointeurs

- Définition de variables pointeur
  - Il existe un type pointeur pour chaque type de variable
  - Variable permettant de stocker une adresse
    - Syntaxe `type *nom;`

- Ex :

```
char *pc;                  // pointeur vers une variable char
short *ps;                 // pointeur vers une variable short
int *pi;                   // pointeur vers une variable int
long *pl;                  // pointeur vers une variable long
float *pf;                 // pointeur vers une variable float
double *pd;                // pointeur vers une variable double
struct personne *psp;      // pointeur vers une variable struct personne

// Attention !!!
int* i1,i2;          // i1 : pointeur vers un entier
                     // i2 : entier
                     // Préférer int *i1,i2;
```

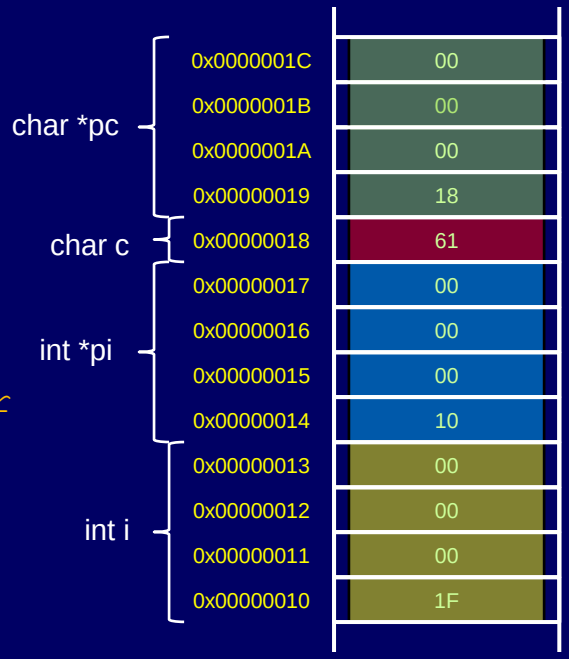
Pointeurs

- Initialisation d'un pointeur
  - Opérateur d'affectation avec une adresse de variable
  - Ex :

```
int i=31;
int *pi=&i;

char c='a';
char *pc=&c;
```

- Accès indirect aux variables
  - Opérateur unaire de déréférencement *
  - Ex :

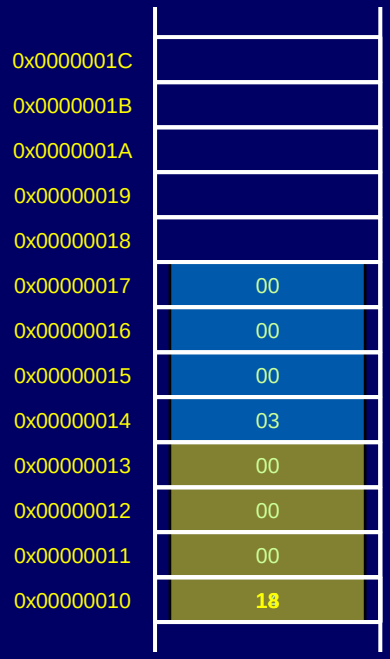
```
int i=31;
int *pi=&i;       // ici * n'est pas l'opérateur
                  // de déréférencement !

int j=*pi;        // *pi valeur de la variable
                  // pointée par pi
```



Pointeurs

- Arithmétique des pointeurs
  - Il est possible d'utiliser des opérateurs pour manipuler une variable
    - Opérateur `+`, `-`, `*`, `/`, `%`, …
  - Il est possible d'utiliser certains opérateurs sur les pointeurs
    - Comportement différent : arithmétique des pointeurs
    - La valeur d'un pointeur peut être augmentée d'un entier
      - La valeur de la variable pointeur (une adresse) est augmentée de la taille de l'élément pointé multiplié par l'entier
      - Ex :

```
int *pi;
int i=3;
pi=&i;
pi++;    // incrément du pointeur de 1*4
```



Pointeurs

- Utilisable uniquement avec des nombres entiers
- Utile pour le parcours des tableaux
  - Indépendant du type
  - Tableau = zone de donnée continue en mémoire
  - Utilisable en addition et soustraction seulement
  - Il est possible de soustraire deux pointeurs
    - Résultat : nombre d'éléments d'un type donné se trouvant entre les deux pointeurs

- Comparaison de pointeurs
  - Compare la valeur d'un pointeur (l'adresse qu'il contient)

```
pi1==pi2        // compare l'adresse contenue dans pi1 et l'adresse
                // contenue dans pi2
```

  - Comparaison des valeurs des variables pointés

```
*pi1==*pi2
```

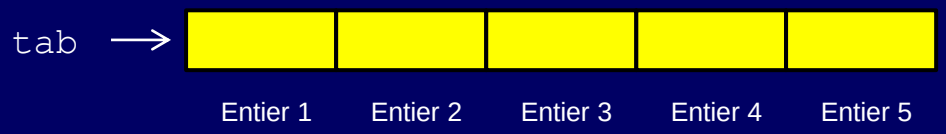
- Pointeur particulier
  - Pointeur NULL a pour valeur 0
    - Adresse particulière de la mémoire ne contenant aucune variable

```
pi=NULL;
```

Pointeurs

- Pointeurs et tableaux
  - Un tableau est une suite continue en mémoire d'éléments d'un même type
  - L'arithmétique des pointeurs permet de naviguer dans une suite continue en mémoire d'éléments d'un même type
    - Il y a une équivalence entre pointeurs et tableaux
    - La variable nom d'un tableau est un pointeur sur la première case du tableau
    - Ex :

```
int tab[5];   // déclare un tableau de 5 entiers
              // tab est un pointeur constant vers un entier
              // (le 1er du tableau)
```



Pointeurs

- Ainsi, `&tab[0]` est équivalent à `tab`
  - L'adresse de la première case du tableau est la valeur contenue dans la constante tab
  - `*(tab+n)` équivaut à `tab[n]`
  - Parcours d'un tableau
    - Ex :

```
int tab[10];
int *pt=tab;

for (int i=0;i<10;i++)
{
   printf("%d\n", *pt);
   pt++;
}
```

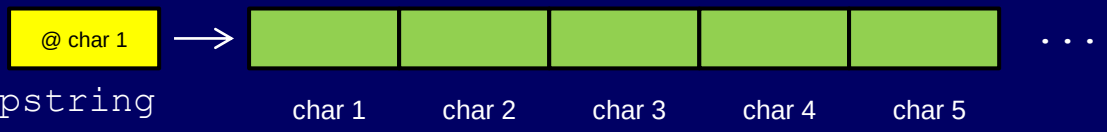
  - Plus efficace que l'opérateur `[]`
    - L'opérateur nécessite un calcul d'adressage

Pointeurs

- Chaînes de caractères constantes et pointeurs
  - L'adresse d'une donnée est affectée à un pointeur via l'opérateur d'adressage `&`
  - Il existe une autre possibilité d'affectation
    - Définition d'une chaîne constante
    - Ex :

      `"Ma chaine de caractère"`
    - Le compilateur crée en mémoire un tableau (sans nom) et y range dedans les caractères `"Ma chaine de caractère"` ainsi que le caractère `'\0'`
    - Une opération sur une chaîne ne peut s'effectuer que par l'intermédiaire d'un pointeur
      - Il est possible de créer une variable pointeur pointant vers la première case du tableau sans nom
      - Ex :

```
char *pstring="Ma chaine de caractère";
```



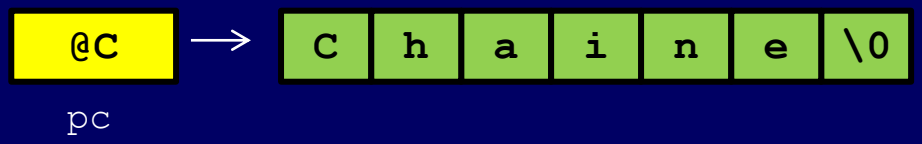
Pointeurs

- Différence entre pointeur et tableau sur une chaîne de caractère constante
  - Ex :

```
char tab[]="Chaine"
char *pc="Chaine"
```

  - Dans les 2 cas, création d'un tableau de 7 caractères
    - Dans le deuxième cas, création d'une variable pointeur vers caractère
      - `tab` est une r-value (ne possède pas de place en mémoire)

      - `pc` est une l-value (possède une place en mémoire)



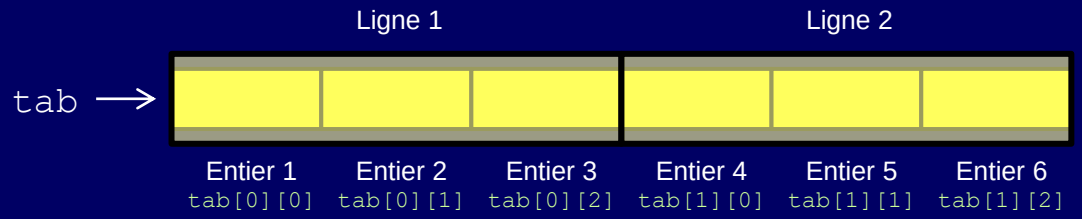
  - Par la suite

```
tab="Nouvelle chaine";     // est impossible
pc="Nouvelle chaine";      // est possible
```

Pointeurs

- Tableaux multidimensionnels
  - Rappel :
    - Il est possible de créer autant de dimensions que nécessaire
    - La mémoire est un tableau unidimensionnel
      - Stockage d'un tableau multidimensionnel en *collant* les dimensions
      - Ex :

```
int tab[2][3];  // déclare une matrice de 2 lignes de 3 colonnes

tab[0][1];        // accède à la 2ème case de la 1ère ligne
*(tab+0*3+1);

tab[1][0];        // accède à la 1ère case de la 2ème ligne
*(tab+1*3+0);
```



Pointeurs

- Passage de paramètres dans les fonctions
  - Le calcul d'adresse nécessité de connaitre toutes les dimensions à l'exception de la dernière
    - Ex :

```
int tab[A][B][C];

tab[x][y][z];      // expressions équivalentes
*(tab + x*(B*C) + y*C + z);
```

  - Nécessité de donner ces informations dans une affectation (et donc un passage de paramètre)
    - Ex :

```
void f1(int tab[A][B][C]);
void f2(int *tab[B][C]);   // omission de la 1ère dimension
void f3(int **tab[C]);     // omission des 2 première dimensions

int tab[A][B][C];
f1(tab)                    // OK
f2(tab)                    // OK
f3(tab)                    // ne compile pas
```

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

Fonctionnement de la mémoire

- L'exécution d'un programme nécessite le stockage de données
  - Il existe 3 zones différentes où il est possible de stocker des données
    - Allocation statique
      - Au moment de la compilation, le compilateur prévoit un espace mémoire
      - Cet espace ne peut varier au cours de l'exécution du programme
      - Utilisé pour stocker les constantes (r-values)
      - Très peu couteuse en termes de performances
    - Allocation sur la pile
      - L'exécution d'un programme est structurée autour d'une pile contenant les cadres d'appel à des fonctions
      - Permet le passage de paramètres dans les fonctions
      - Utilisé par le compilateur pour allouer des variables (toute variable déclarée)
      - Mémoire gérée par le compilateur, portée définie par les blocs
    - Allocation sur le tas
      - Il est nécessaire de pouvoir, à des moments arbitraires de l'exécution, demander au système l'allocation de nouvelles zones de mémoire
      - Dépends et de la responsabilité du programmeur
      - Indépendant de la notion de bloc
      - Utilisation de fonctions spéciales d'allocation de cette mémoire

Fonctionnement de la mémoire

- Constantes
  - Les constantes doivent être initialisées à la déclaration
  - Leur valeur ne peut changer au cours de l'exécution du programme
  - Elles sont gérées par le compilateur
  - Généralement placées dans le segment de données du programme
  - Mot clé `const`
  - Ex :

```
const int x=3;
```

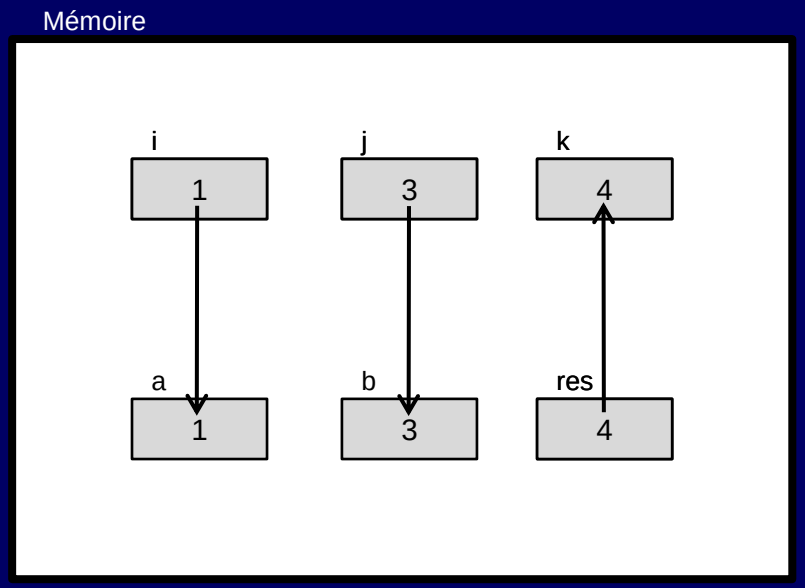
- Pointeurs constants
  - 2 possibilités de constantes pour un pointeur
    - Constance de la valeur pointée (`const Type *Variable` ou `Type const *Variable`)
    - Constance de l'adresse contenue dans le pointeur (`Type *const Variable`)
    - Constance des 2 éléments (`const Type *const Variable`)
  - Le mot clé `const` s'applique à ce qui le suit directement
  - Ex :

```
int i1=0,i2=0;
const int *pi1;
int *const pi2=&i2;      // Initialisation OBLIGATOIRE

pi1=&i2;                 // VALIDE
pi2=&i1;                 // NON VALIDE
*pi1=3;                  // NON VALIDE
*pi2=3;                  // VALIDE
```

Fonctionnement de la mémoire

- Allocation mémoire par le compilateur
  - Lors de la déclaration d'une variable, le compilateur alloue l'espace mémoire sur la pile
    - Désalloue l'espace mémoire automatiquement à la fin d'un bloc (portée d'une variable)
    - Ex :

```
int add(int a,int b)
{
    int res;

    res=a+b;
    return(res);
}

int main()
{
    int i,j,k;

    i=1;
    j=3;
    k=add(i,j);
}
```



Fonctionnement de la mémoire

- Variables statiques
  - Une variable de classe statique n'est créée qu'une seule fois en mémoire
    - Même si le bloc qui la contient est exécuté plusieurs fois
  - Elle peut être initialisée à sa déclaration
    - Dans le cas contraire, sa valeur sera de `0`
    - Une variable statique n'est jamais indéfinie
  - Mot clé `static`
  - Ex :

```
void MyFunction()
{
    static int i=0;
    int j=0;

    i++;
    j++;
}

void main()
{
    MyFunction();      // 1ère itération ; à la fin de la fonction i=1;j=1;
    MyFunction();      // 2ème itération ; à la fin de la fonction i=2;j=1;
}
```

Fonctionnement de la mémoire

- Allocation mémoire dynamique par l'utilisateur
  - Il peut être utile d'avoir une portée supérieure au bloc pour certaines données
  - De contrôler la quantité de mémoire allouée
    - Tableau dynamique
    - Modification de la taille d'un tableau
  - Allocation de données sur le tas
  - L'API standard du C expose plusieurs fonctions de manipulation de la mémoire
    - Bibliothèque `<stdlib.h>`

```
void* malloc(size_t size);
    // Allocation d'une zone continue en mémoire

void* calloc(size_t num, size_t size);
    // Allocation d'une zone continue de num éléments de taille size
    // en mémoire en initialisation à 0 de cette zone

void* realloc(void * ptr, size_t size);
    // Réallocation d'une zone continue en mémoire

void free(void *ptr);
    // Libération de la mémoire allouée
```

Fonctionnement de la mémoire

- Chacune des fonctions d'allocation retourne un `void*`
- L'API d'allocation de mémoire du C ne peut que
  - Réserver un certain nombre d'octets contigus en mémoire
  - Renvoyer l'adresse de début de cette zone
- On accède à la mémoire qui a été alloué via l'adresse retournée par `malloc`, `calloc` ou `realloc`
  - Si on perd cette adresse, on ne pourra plus libérer la mémoire !

- La fonction `malloc` prend en paramètre le nombre d'octets du tableau qu'il faut allouer
  - Indépendant du type du tableau !
  - Penser à réserver suffisamment de place mémoire pour stocker les données
    - Utilisation de l'opérateur `sizeof`
- Il est nécessaire de caster le résultat dans le type (pointeur) voulu
  - Ex :

```
char *str=(char*) malloc(nb*sizeof(char));          // nb*1 octets alloués
double *tabD=(double*) malloc(nb*sizeof(double));   // nb*8 octets alloués
```

Fonctionnement de la mémoire

- La fonction `calloc` permet de réserver et d'initialiser un bloc mémoire
  - Initialise les éléments à la valeur 0
  - 2 paramètres
    - Nombre d'éléments à réserver
    - Taille de chaque élément
  - Ex :

```
char *str=(char*) calloc(nb,sizeof(char));          // nb*1 octets alloués
double *tabD=(double*) calloc(nb,sizeof(double));   // nb*8 octets alloués
```

- La fonction `realloc` permet de modifier la taille d'un bloc mémoire
  - Le bloc doit avoir été précédemment alloué par `malloc` ou `calloc`
  - 2 paramètres
    - Pointeur vers le bloc de mémoire à étendre
    - Nombre d'octets du tableau qu'il faut allouer
      - Préserve la valeur des octets communs
      - Ne peut initialiser les octets supplémentaires
  - Ex :

```
char *str=(char*) realloc(ptr,nb*sizeof(char));          // nb*1 octets alloués
double *tabD=(double*) realloc(tabD,nb*sizeof(double)); // nb*8 octets alloués
```

Fonctionnement de la mémoire

- Particularités du `realloc`
  - Si le pointeur passé à `realloc` a la valeur NULL, alors `realloc` se comporte comme un `malloc`
  - La fonction `realloc` peut déplacer les éléments précédents pour préserver une zone contigu en mémoire
    - Modifie la valeur du pointeur
    - Attention si la réallocation se fait dans une sous-fonction

- Libération de la mémoire
  - Quel que soit la fonction d'allocation utilisée, la mémoire se libère avec la fonction `free`
  - Prends en paramètre un pointeur désignant l'adresse de début de la zone mémoire à libérer
  - Ex :

```
free(str);              // libère la zone mémoire pointée par str
```

- Gestion des erreur
  - En cas de problème d'allocation mémoire
    - Plus de mémoire disponible
  - `malloc`, `calloc`, `realloc` retournent `NULL`
  - Penser à vérifier la valeur de retour
    - Stocker le pointeur de retour du realloc dans une variable différente du bloc de départ
    - Source de beaucoup de problèmes !

Fonctionnement de la mémoire

- Alignement
  - L'alignement de données est la façon dont les données sont organisées en mémoire
  - Les compilateurs sont souvent obligés d'aligner les données en mémoire
    - Pour des raisons de performance
    - Pour faciliter l'exécution du code
  - Ex :
    - La taille d'une structure est multiple de la taille du registre du processeur

```
struct _noalign
{
    char c;
    double d;
    int i;
    char c2[3];
};
```

```
struct _align
{
    double d;
    int i;
    char c2[3];
    char c;
};
```

    - Les deux structures contiennent les mêmes champs, on pourrait croire qu'elles ont la même taille
      - les champs de type `char` sont codés sur 1 octet, `int` sur 4 octets, `double` sur 8 octets
      - La taille totale devrait être 1+8+4+3*1=16 octets
    - Mais

```
printf("noalign %d\n",sizeof(struct _noalign));        // Affiche 24
printf("align %d\n",sizeof(struct _align));   // Affiche 16
```

Fonctionnement de la mémoire

- La structure `struct _align` est bien alignée mais pas `struct _noalign`
- Le compilateur rajoute des bits de « padding » pour respecter l'alignement
- `struct _noalign` est transformé par le compilateur en

```
struct _noalign_corrige
{
    char c;
    char _pad1[7];
    double d;
    int i;
    char c2[3];
    char _pad2;
};
```

- `_pad1` permet à `d` de débuter à une adresse multiple de 8
- `_pad2` complète la structure pour atteindre 24 (multiple de la taille des registres du processeur)

- Moralité : TOUJOURS utiliser l'opérateur `sizeof` pour calculer la taille des différents éléments en mémoire

Fonctionnement de la mémoire

- Allocations mémoires au niveau du système
  - L'API du C expose des fonction d'allocation mémoire standard
  - Il existe des mécanismes d'allocation mémoire spécifique à chaque SE
  - Mais
    - Non portable
    - Construction et destruction manuelle

  - Besoins spécifiques
    - Optimisation : pool de mémoire
      - Fragmentation
      - Temps d'allocation
      - Quantum d'allocation
    - Utilisation d'une zone mémoire spéciale
      - Pile
      - Thread Local Storage
      - Mémoire partagée
      - …

Fonctionnement de la mémoire

- Unix/Linux
  - Contrôle taille segment de données
    - Section « data » + tas
    - Fonctions `brk`, `sbrk`
  - `mmap`
    - Usages divers
      - File Mapping
      - Allocation segments privés
      - Allocation mémoire partagée

```
int brk(void* fin_segment_donnée)
void* sbrk(ptrdiff_t incrément)
void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset)
int munmap(void* start, size_t length)
```

Fonctionnement de la mémoire

- Allocation variable sur la pile
  - `alloca`
  - Attention ! Implémentation parfois buggée
    - Mauvaise portabilité

- Allocation mémoire alignée
  - `posix_memalign`
  - Fonctions équivalentes obsolètes
    - `valloc`, `memalign`

```
void* alloca(size_t size)
int posix_memalign(void** memptr, size_t alignment, size_t size)
void* memalign(size_t boundary, size_t size)
void* valloc(size_t size)
```

Fonctionnement de la mémoire

- Windows
  - Allocation sur le tas
    - `GetProcessHeap`
    - `HeapAlloc`, `HeapFree`, `HeapReAlloc`
    - Fonctions obsolètes
      - `LocalAlloc`, `LocalFree`, `LocalReAlloc`
      - `GlobalAlloc`
        - Systèmes 32 bits : tas local et global identiques
  - Création de tas supplémentaires
    - `HeapCreate`, `HeapDestroy`
  - Allocation mémoire virtuelle
    - `VirtualAlloc`, `VirtualFree`
  - File mapping, mémoire partagée
    - `CreateFileMapping`, `CloseHandle`
    - `MapViewOfFile`, `UnmapViewOfFile`

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

Les fonctions (2)

- Fonctions avec un nombre variables de paramètres
  - Il est possible de créer en C des fonctions avec un nombre variable de paramètres
    - Ex : `printf`, `scanf`
  - Définition à l'aide d'une syntaxe particulière

```
type nom_fonction(type1 param1, ...)
```

  - Ex :

```
int f(char a, int b, ...);     // déclare une fonction à au moins 2 paramètres
```

  - Évaluation des paramètres
    - L'évaluation des paramètres variables se fait à l'aide de macros disponibles dans la librairie standard via l'entête `<stdarg.h>`
    - Macros `va_start`, `va_arg`, `va_end`
    - Type `va_list`
    - On accède aux différents paramètres optionnels dans leur ordre d'apparition
    - A chaque accès à une variable optionnelle, on pointe sur la variable optionnelle suivante

Les fonctions (2)

- Macro `va_start`
  - Permet d'initialiser la liste des arguments optionnels
  - Admet 2 paramètres
    - `va_list` qui sauvegarde le contexte d'accès aux variables optionnelles
    - Le dernier argument fixe de la fonction
      - Toute fonction à paramètres variables doit admettre au moins un argument non variable

- Macro `va_arg`
  - Permet de récupérer l'argument optionnel courant dans le contexte d'accès aux variables optionnelles
  - Donne toujours la valeur du paramètre optionnel pointé dans le contexte
  - Déplace le pointeur du contexte sur la variable optionnelle suivante
  - Admet 2 paramètres
    - `va_list` qui sauvegarde le contexte d'accès aux variables optionnelles
    - Le type de la variable à récupérer
      - Attention ! Un type `char` se récupère via un type `int` (conversion interne)

- Macro `va_end`
  - Permet de libérer la mémoire réservée pour sauvegarder le contexte d'accès aux variables optionnelles
  - Admet 1 paramètre
    - `va_list` qui sauvegarde le contexte d'accès aux variables optionnelles

Les fonctions (2)

- Ex :

```
void bprint(int nb, ...)          // Fonction affichant un nombre variable d'entiers
{
    unsigned int i;
    int temp;
    va_list pp;

    va_start(pp, nb);
    for (i=0;i<nb;i++)
    {
            temp=va_arg(pp, int);
            printf("%u ", temp);
    }
    printf("\n");

    va_end(pp);
}
```

- Attention
  - Nécessite de connaitre le nombre d'arguments optionnels (via les paramètres fixes par exemple)
  - Aucun contrôle de type automatique
    - Explique les erreurs mémoires occasionnées par l'utilisation de `scanf`, `printf`, …

Les fonctions (2)

- Pointeurs de fonction
  - Une fonction n'est pas une variable en mémoire
  - Mais elle possède une adresse
    - L'adresse de la première instruction de la fonction
  - Cette adresse est manipulable
    - Stockable dans une variable
    - Il est possible de créer une variable pointeur vers fonction

  - L'adresse d'une fonction est matérialisée par son nom
    - De manière similaire à un nom de tableau
    - Ex : `printf` est un pointeur (constant) vers la fonction concernée

  - Définition d'un pointeur de fonction

    `<type> (*<pointeur>)(paramètres)`
    - Si les paramètres ne sont pas spécifiés, le pointeur de fonction est générique concernant les paramètres
    - Ex :

```
int (*fp)();    // Désigne une variable fp qui est un pointeur vers une
                // fonction retournant un int
```

Les fonctions (2)

- Permet d'appeler plusieurs fonction de signature identiques à travers une même variable
  - Utile pour les fonction de tri par exemple
  - Ex :

```
int func1(int a, double b)
{
    /* ... */
    return(0);
}

int func2(int nombre, double val)
{
    /* ... */
    return(1);
}

int (*fp)(int,double);     // Définition d'un pointeur de fonction
int (*fp2)();              // Définition d'un pointeur de fonction générique

fp=func1;                  // Valide
fp=func2;                  // Valide
fp2=func1;                 // Valide
```

Les fonctions (2)

- Appel d'une fonction via un pointeur de fonction
  - Syntaxe

    `(*<pointeur>)(paramètres)`

  - Ex :

```
int func1(int a, double b)
{
    /* ... */
    return(0);
}

int func2(int nombre, double val)
{
    /* ... */
    return(1);
}

int (*fp)(int,double);

fp=func1;
(*fp)(1,2.3)      // Appel de la fonction func1;

fp=func2;
(*fp)(1,2.3)      // Appel de la fonction func2;
```

  - Ex : fonction de tri

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

Entrée sortie (2)

- Un fichier permet un stockage de données hors mémoire vive
  - Survit à une coupure de l'alimentation de la mémoire (extinction d'un ordinateur)

- En C, il existe 2 façons d'accéder aux fichiers
  - Niveau bas : fonctions du système d'exploitation (appels systèmes)
    - Ne fait pas partie du standard ANSI
  - Niveau haut : fonctions de la bibliothèque standard
    - Fait partie du standard ANSI, portable d'un système à l'autre

- Bibliothèque standard
  - Les données lues dans un fichier passent par un buffer
    - Zone mémoire dans la RAM
    - Accélère la lecture des fichiers : accès en mode bloc
  - Nécessité d'utiliser une structure en C pour accéder à cette zone mémoire
    - `struct FILE;`

Entrée sortie (2)

- Structure FILE
  - Définition

```
typedef struct
{
    char *buffer;     // pointeur vers le buffer de données
    char *ptr;        // pointeur vers le caractère suivant dans le buffer
    int cnt;          // nombre de caractères dans le buffer
    int flags;        // bits donnant l'état du fichier
    int fd;           // descripteur de fichier (identifiant pour le SE)
} FILE;
```

  - Cette structure ne doit pas être manipulée directement, mais via des fonctions fournies dans la bibliothèque standard
    - Opérations d'ouverture et de fermeture d'un fichier
    - Opérations de lecture et d'écriture d'un fichier
    - Opérations d'accès aléatoire sur le fichier

Entrée sortie (2)

- Ouverture d'un fichier
  - Avant de pouvoir manipuler un fichier, il faut l'ouvrir
  - Fonction `fopen`

```
FILE* fopen(char *file, char *mode);
```

  - Permet de créer une instance de la structure file pour le fichier concerné
  - Demande au SE de vérifier les droits sur le fichier
  - Paramètres
    - Nom du fichier à ouvrir (URL d'accès)
      - Ex : `"/etc/fstab"`
    - Mode d'accès sur le fichier
      - `"r"` : lecture seule
      - `"w"` : écriture, si le fichier n'existe pas, il sera créé, s'il existe son contenu sera écrasé et perdu
      - `"a"` : ajout de donnée (écriture à la fin du fichier), si le fichier n'existe pas, il sera créé
      - `"r+"` : lecture et écriture, si le fichier n'existe pas, `fopen` retourne `NULL`
      - `"w+"` : lecture et écriture, si le fichier n'existe pas, il sera créé, s'il existe son contenu sera écrasé et perdu
      - `"a+"` : lecture et ajout, le fichier est créé s'il n'existe pas

Entrée sortie (2)

- Fichier texte et binaire
  - Le concept de fichier texte correspond à la représentation d'un texte sous forme de lignes
    - Chaque ligne se termine par un caractère spécial
  - Le concept de fichier binaire correspond à une suite d'octets représentant toutes sortes de données
  - La bibliothèque C peut prendre en compte cette différenciation
    - Spécification de fin de ligne différente suivant le Système d'Exploitation
    - `"\n"` pour UNIX, `"\r\n"` pour Windows, `"\r"` pour MAC (précédant OSX)
    - Ouverture en mode texte
      - rajouter `"t"` au mode d'accès
    - Ouverture en mode binaire (par défaut)
      - Rajouter `"b"` au mode d'accès

- Fermeture d'un fichier
  - Fonction `fclose`

```
void fclose(FILE *fp);
```

Entrée sortie (2)

- Ex :

```
FILE *fp;

if (fp=fopen("/etc/fstab", "r"))
{
    // traitements ...

    fclose(fp);
}
```

Entrée sortie (2)

- Opérations de lecture/écriture
  - Fonctions analogues à celles permettant d'afficher des données à l'écran ou de lire depuis le clavier
  - Lecture/écriture en mode caractère
    - Fonctions `fgetc`/`fputc`

```
int fgetc(FILE *<pointeur_fichier>);
int fputc(int <caractère>, FILE *<pointeur_fichier>);
```

    - Ex :

```
fputc('a',fp);
c=fgetc(fp);
```

  - Lecture/écriture en mode chaîne
    - Fonctions `fgets`/`fputs`

```
char* fgets(char *<pointeur_tampon>, int <nombre>, FILE *<pointeur_fichier>);
int fputs(char *<pointeur_tampon>, FILE *<pointeur_fichier>);
```

    - Ex :

```
char buffer[200]="Texte exemple";
fputs(buffer,fp);
fgets(buffer,200,fp);
```

Entrée sortie (2)

- Lecture/écriture formatés
  - Fonctions `fprintf`/`fscanf`

```
int fprintf(FILE *< pointeur_fichier >, const char * <format>, ...);
int fscanf(FILE *< pointeur_fichier >, const char * <format>, ...);
```

  - Ex :

```
fprintf(fp,"Chaine formatée : %d\n",3);
```

- Lecture/écriture par bloc
  - Fonctions `fwrite`/`fread`

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

  - `fwrite` écrit `size*count` octets rangés à l'emplacement mémoire référencé par `ptr`
  - `fread` lit un bloc de `size*count` octets et le range à l'emplacement mémoire référencé par `ptr`

Entrée sortie (2)

- Détection d'une fin de fichier
  - Lecture du caractère `EOF`
    - Mais possibilité de trouver le caractère `EOF` dans un fichier binaire
  - Utilisation de la fonction `feof`

    ```
    int feof(FILE *< pointeur_fichier >);
    ```
    - Renvoie une valeur non nulle si la fin de fichier est atteinte lors de la dernière lecture de caractère, zéro sinon
    - Attention à effectuer une lecture de caractère avant de tester la fin de fichier !

- Accès direct
  - Fonctions de lecture : accès séquentiel
  - Permet de se déplacer à l'intérieur du fichier
  - Fonction `fseek`

    ```
    int fseek(FILE *stream, long int offset, int origin);
    ```
  - Le paramètre `origin` désigne le point de départ du déplacement relatif de offset `octets`
    - `SEEK_SET` : début du fichier
    - `SEEK_CUR` : position courante
    - `SEEK_END` : fin du fichier

  - Ex :

    ```
    fseek(fp,-3,SEEK_END);      // Déplacement de 3 octets vers le début du fichier
                                // depuis la fin du fichier
    ```

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

Utilisation avancée du compilateur

- Options de compilations de gcc
  - Quelques paramètres de gcc utiles
    - `-Wall -W`
      - Affiche des warnings supplémentaires. Les warnings sont une façon pour le compilateur d'indiquer qu'il y a peut-être une erreur (mais le code compile quand même)
      - Il faut toujours essayer d'éliminer les warnings
    - `-Wuninitialized, -O`
      - Option inclue dans -Wall mais qui a aussi besoin de -O
      - Permet de générer des avertissements quand vous utilisez une variable avant de lui avoir donné une valeur
    - `-pedantic, -ansi`
      - Activent la norme C89 (à na pas utiliser de préférence)
    - `-std=c11`
      - Active la norme C11 (à préférer)
    - `-On` (`-O -O0 -O1 -O2 -O3 -Os`)
      - gcc peut réaliser un certain nombre d'optimisations sur le code généré
      - Avec l'option `-O`, le compilateur essaye de réduire la taille du code et le temps d'exécution sans chercher à optimiser le code
      - La valeur de `n`, et son effet exact, dépend de la version de gcc, mais s'échelonne normalement entre 0 (aucune optimisation) et 2 (un certain nombre) ou 3 (toutes les optimisations possibles)

Utilisation avancée du compilateur

- `-g`
  - Active le débogage
    - Permet de rajouter des informations supplémentaires dans le fichier compilé afin d'aider le processus de débogage

- `-Idir`
  - Ajoute des répertoires par défaut pour la recherche de fichiers exprimés via la directive `#include`
  - Ex :

```
gcc -I/home/user/mylib/include -I/home/user/prgm/inc
```

- `-lfile`
  - Indique les librairies supplémentaires à utiliser pendant le processus d'édition des liens
    - Le nom du fichier cible doit être de la forme `libfile.a` (librairie statique) ou `libfile.so` (librairie dynamique)
  - Ex :

```
gcc -lsock -lm       // utilise les librairies libcsock.a et libm.a
```

- `-Ldir`
  - Ajoute des répertoires par défaut pour la recherche de librairies exprimés via la directive `-l`
  - Ex :

```
gcc -L/home/user/lib -lmalib
```

Utilisation avancée du compilateur

- Création et utilisation de bibliothèque avec gcc
  - Il existe 2 types de bibliothèques
  - Statiques
    - Avantages d'une bibliothèque statique
      - L'exécutable est complètement autonome
      - Si l'édition de lien réussie, fonctionnement immuable
    - Inconvénients d'une bibliothèque statique
      - Duplication des modules dans l'exécutable
        - Consommation d'espace disque (gros exécutables)
        - Consommation de mémoire (pas de partage inter-processus)
      - Exécutable insensible aux mises à jour des bibliothèques utilitaires

Utilisation avancée du compilateur

- Dynamiques
  - Avantages d'une bibliothèque dynamique
    - Pas de duplication du code
      - Fichiers exécutables de petite taille
      - Chargées une seule fois en mémoire
        - partagées entre tous les processus qui en ont besoin
    - Exécutables sensibles aux mises à jour
  - Inconvénients d'une bibliothèque dynamique
    - Etre capable de retrouver la bibliothèque au lancement
    - Possibilité de casser d'anciens programmes

Utilisation avancée du compilateur

- Utilisation d'une bibliothèque statique
  - Généralement de la forme `libXXX.a`
  - Une simple collection de fichiers objets sur le même thème
    - Inspectée à l'édition de liens
    - Modules objets requis incorporés à l'exécutable
    - Uniquement les modules nécessaires (découper au maximum)
  - Edition de liens
    - `gcc ... -lXXX` (pour `libXXX.a`)
    - Problème : recherche des bibliothèques dynamiques avant
      - Utiliser l'option `-static` pas de bibliothèques dynamiques
      - Indiquer la bibliothèque « en dur »
        - `gcc ... ./libXXX.a`
        - Considérée comme la liste de ses .o

Utilisation avancée du compilateur

- Réalisation d'une bibliothèque statique
  - Utiliser la commande `ar` (`man 1 ar`)
  - De nombreuses options
    - Commande usuelle `ar cr libXXX.a *.o`
      - `c` : créer à partir de rien
      - `r` : insertion inconditionnelle
    - Possibilités d'ajout, de retrait, de mise à jour
  - Création d'un index
    - `ranlib libXXX.a`
    - Accélère l'édition de liens
    - Généralement facultatif (sauf sous MacOSX)

Utilisation avancée du compilateur

- Utilisation d'une bibliothèque dynamique
  - Généralement de la forme `libXXX.so`
  - Une simple collection de fichiers objets sur le même thème
    - Inspectée à l'édition de liens
    - Références aux symboles incorporés à l'exécutable
  - Edition de liens :
    - `gcc ... -lXXX` (pour `libXXX.so`)
    - Bibliothèques dynamiques recherchées avant les statiques
    - Indiquer la bibliothèque « en dur »
      - `gcc ... ./libXXX.so`

Utilisation avancée du compilateur

- Réalisation d'une bibliothèque dynamique
  - Utiliser le compilateur habituel avec l'option `-shared`
    - Exactement comme la génération d'un exécutable
    - Génère uniquement une bibliothèque dynamique
    - Pas besoin de `main()`
  - Ex :
    - `gcc -shared -o libXXX.so *.c`
  - Réduire la taille de la bibliothèque (`man 1 strip`)
    - Retirer les symbole inutiles (débogages , …)
    - `strip -s libXXX.so`

Utilisation avancée du compilateur

- Retrouver les bibliothèques dynamiques au lancement
  - Préciser des répertoires non-standards
    - `export LD_LIBRARY_PATH=$HOME/lib:/usr/local/XXX /lib`
  - Utile pendant la phase de développement de la bibliothèque
  - Solution temporaire, installation finale dans un répertoire standard
  - Répertoires standards implicites (`/usr/lib`, `/usr/X11R6/lib`, …)

- Le pré-chargement de symboles
  - Forcer le pré-chargement des symboles de bibliothèques spécifiques
    - `LD_PRELOAD="./libXXX.so ./libYYY.so" ./prog`
      - `prog` ne cherche pas ces symboles dans les autres bibliothèques
  - Permet de choisir une implémentation particulière parmi plusieurs
    - Mesures de performances, débogage, …

Utilisation avancée du compilateur

- Recherche de symboles dans les modules
  - L'utilitaire nm (`man 1 nm`)
    - Disponible dans tout environnement de développement
      - Compilateur, assembleur, linker ,débogueur , …
    - Affiche des informations concernant les symboles
      - Des fichiers objets
      - Des bibliothèques statiques
      - Des bibliothèques dynamiques
      - Des fichiers exécutables
    - Utilisation classique
      - Il manque un symbole à l'édition de liens
      - Un symbole est défini à multiples reprises
      - On le recherche parmi dans les bibliothèques disponibles

Utilisation avancée du compilateur

- Lister les dépendances
  - Commande ldd (`man 1 ldd`)
  - Affiche les bibliothèques dynamiques associées au chargement
    - D'un programme dynamiquement lié
    - D'une bibliothèque dynamique

Utilisation avancée du compilateur

- Chargement dynamique de symboles
  - Principe
    - Charger des ressources (code et données) à la demande
      - Ressources initialement inconnues
      - Choisies lors de l'exécution du programme
      - Dans des bibliothèques dynamiques
      - Chargées et déchargées explicitement
    - Différent de l'édition de liens habituelle
      - Liste de bibliothèques à charger au démarrage
      - Chargées implicitement
      - Chargées pour toute la durée de l'exécution
    - Donner des possibilités d'extension à une application
      - Principe des plug-ins

Utilisation avancée du compilateur

- Démarche de mise en œuvre
  - `#include <dlfcn.h>` (`man 3 dlopen`)
  - Ouverture avec `dlopen()` (`dlerror()` en cas d'échec)
  - Accès aux symboles avec `dlsym()` (`dlerror()` en cas d'échec)
  - Fermeture avec `dlclose()`
    - Les symboles obtenus avec `dlsym()` deviennent inaccessibles

- Réalisation du programme initial
  - Edition de liens avec `-ldl`
  - Edition de liens avec l'option `-rdynamic` (sous Linux)
    - Permet au code du plugin d'accéder au code initial
    - C'est généralement nécessaire
    - Sans cette option il faudrait mettre le code initial dans une bibliothèque dynamique pour la lier au plugin

Utilisation avancée du compilateur

- Débogage
  - Nécessite de compiler et effectuer l'édition de liens avec l'option `-g`, et sans l'option `-fomit-frame-pointer`
    - Vous ne devez compiler que les modules que vous avez besoin de déboguer
  - Utiliser un logiciel permettant de manipuler l'exécution
    - Définition de points d'arrêts
    - Exécution pas à pas
    - Dump de mémoire
      - Visualisation de l'évolution des valeurs des variables
  - Sous Linux, beaucoup de gens utilisent gdb
  - Autre outil utile pour déboguer
    - *strace*, qui affiche les appels systèmes que le processus lance
      - Nécessite de donner les chemins d'accès où ont été compilés les binaires
      - Donne les temps passés dans chacun des appels systèmes
      - Permet également de connaître les résultats des appels

Utilisation avancée du compilateur

- Fichiers core
  - Un fichier core est créé lors d'une faute grave, comme une violation d'espace mémoire
  - Lorsque Linux se lance, il n'est généralement pas configuré pour produire des fichiers core
    - Pour les valider utiliser les commandes

```
limit core unlimited
ulimit -c unlimited
```

  - Utilisation
    - Pour utiliser ce fichier core, il faut avoir compilé le programme avec l'option de déboguage, puis lancer gdb en indiquant le nom du programme suivit du nom du fichier core

```
gdb bogue core
```

      - Permet de récupérer la valeur des variables et des données à l'instant précis de l'erreur

    - La commande `backtrace` affiche la pile d'appels au moment où le problème est survenu
      - La première fonction affichée (`#0`) est aussi la dernière appelée au moment de l'erreur
        - Celles qui suivent ont été appelées dans l'ordre chronologique

      - La commande frame permet d'accéder à l'appel correspondant
        - Ex :

```
frame 2          // Accède à la fonction correspondant au numéro 2 (#2)

                 // Si le problème se situe en amont (dans une fonction
                 // précédente), utiliser la commande up pour remonter
                 // la pile d'appels
```

Programmation avancée en C : Plan

- Préprocesseur
  - Macros, inclusion de fichiers
  - Compilation conditionnelle
  - Messages et pragma
- Pointeurs
  - Qu'est-ce qu'un pointeur
  - Arithmétique des pointeurs
- Fonctionnement de la mémoire
  - Différents types de mémoire
  - Variables statiques et constantes
  - Allocation par le compilateur, dynamique par l'utilisateur, dépendantes du système
  - Alignement
- Les fonctions 2
  - Paramètres (nombre variable)
  - Pointeurs de fonction
- Entrée sortie 2 : gestion de fichiers
- Utilisation avancée du compilateur
  - Options de compilation
  - Création et utilisation de librairies
  - Débogage
- Bibliothèque standard

Bibliothèque standard

- Librairie math

```
#include <math.h>
```

  - Valeur absolue d'un réel
    `fabs`
  - Valeur absolue d'un entier
    `abs, labs`
  - Fonctions trigonométriques
    `acos, asin, atan, atan2, cos, sin`
  - Fonctions hyperboliques
    `cosh, sinh, tanh`
  - Fonctions exponentielle et puissance
    `exp, log, log10, pow, sqrt, ldexp, frexp`
  - Arrondi
    `ceil, floor`
  - Modulo et décomposition
    `fmod, modf`
  - Constantes mathématiques
    `M_E, M_LOG2E, M_LOG10E, M_LN2, M_LN10, M_PI, M_PI_2, M_PI_4, M_1_PI, M_2_PI, M_2_SQRTPI, M_SQRT2, M_SQRT1_2, MAXFLOAT`
  - Gestion des erreurs mathématiques
    `matherr`

Bibliothèque standard

- Librairie de traitement de chaîne de caractère

```
#include <string.h>
```

  - Copie de chaîne
    `strcpy, strncpy`
  - Longueur d'une chaîne
    `strlen`
  - Concaténation de chaîne
    `strcat, strncat`
  - Comparaison de chaînes
    `strcmp, strncmp, strcasecmp, strncasecmp`
  - Recherche de caractère
    `strchr, index, strrchr, rindex, strpbrk, strstr, strrstr, strspn, strcspn, strtok`
  - Duplication de chaînes
    `strdup`

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

  - Création d'une chaîne formatée
    `sprintf`

Bibliothèque standard

- Opérations sur la mémoire

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

  - Copie d'un bloc mémoire

    `memcpy, memccpy, memmove`
  - Recherche d'un octet dans un bloc

    `memchr`
  - Comparaison de zones mémoire

    `memcmp`
  - Initialisation d'une zone mémoire

    `memset`
  - Portabilité des applications BSD

    `bcopy, bcmp, bzero`

Bibliothèque standard

- Date et heure

```
#include <time.h>
```

  - Récupération de l'heure système
    `time`
  - Conversion de la date et l'heure en une chaîne
    `ctime`
  - Conversion d'une date et heure en une structure
    `localtime`
  - Conversion d'une date et heure au standard GMT
    `gmtime`
  - Conversion de la date et l'heure en une chaîne
    `asctime`
  - Calcul du délai séparant deux instants
    `Difftime`
  - Conversion du temps au format du calendrier
    `mktime`

Bibliothèque standard

- Utilitaires généraux

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

  - Conversions de chaînes
    `atof, atoi, atol`
  - Conversions d'un entier en chaîne
    `ltoa, ltostr`
  - Gestion de processus
    `exit, abort, system`
  - Nombres aléatoires
    `rand, srand`
  - Recherche binaire d'un objet dans un tableau
    `bsearch`
  - Tri par algorithme rapide « quicksort »
    `qsort`