Le langage C

V. Nicomette, T. Monteil, S. Hernando, F. Pompignac

TLS-SEC

Généralités

- Langage créé par Dennis Ritchie et implémenté sur le système UNIX : le système lui-même et beaucoup d'applications qui s'exécutent sur UNIX sont écrits en C
- Première édition du manuel de référence du C en 83
- Définition du C ANSI en 1988
  - nouvelle syntaxe de déclaration et définition de fonctions
  - standardisation de la librairie C (appels système)
  - la majorité des compilateurs supportent le C ANSI
- Quelques modifications fin des années 90 avec le standard C99
- Langage typé MAIS *non fortement typé* à la différence d'ADA
- Langage qui permet la création de programmes structurés (blocs, instructions de contrôle, instructions itératives, ...)
- Langage qui reste un langage de "bas-niveau" (très adapté pour les pilotes de périphériques par exemple)

Aperçu d'un programme C

```
/* Ca commence ici ... */     <-- commentaire

#include <stdio.h>            <-- directive du preprocesseur

char chaine[]="Hello";        <-- definition de variable
int affiche(char ch[]);       <-- declaration de fonction

int main()                    <-- fonction principale
{
  affiche(chaine);
  return(0);
}

int affiche(char ch[])        <-- definition de fonction
{
  int i=3;                    <-- definition de variable
  printf("%d %s\n",i,ch);
}
```

Variables et opérateurs (chap. I)

- Trois familles de variables simples (I.1.1)
  - caractère : `char` (1 octet)
  - entier : `short` (2 octets), `int` (4 octets) `long int` (4 octets sur un système 32 bits, 8 octets sur un système 64 bits), `long long int` (8 octets) défini dans le standard C99
  - flottant : `float` (4 octets), `double` (8 octets)
- tailles non normalisées : utilisez `sizeof`
- ces types sont signés (un bit pour coder le signe) par défaut
- on peut utiliser des types non signés : `unsigned short mot;` (de 0 à 65535)
- Type caractère (I.1.2)
  - différentes possibilités d'affectation : alphabétique (`'8'`), hexadécimal (0x38), octal (`'\070'`), caractères spéciaux (`'\n'`)

Variables et opérateurs (chap. I)

- Absence de type booléen (I.1.3)
  - la valeur 0 signifie faux
  - tout autre valeur signifie vrai
  - type `bool` introduit dans le standard C99 (valeurs `true (1)` et `false(0)`)
- Les tableaux (I.2.1-I.2.3)
  - collection de variables d'un même type
  - premier indice est toujours 0
  - Pas de détection de débordement de tableau par le compilateur ! ! !
  - Exemples :
    ```
    char Mot[10] (dernier élément : Mot[9] ! !)
    int Codes[5]
    Mot[0]='a'; Codes[1]=10;
    ```
  - tableaux à plusieurs dimensions
    ```
    char Codage[12][5];
    ```

Variables et opérateurs (chap. I)

- Les tableaux (I.2.1-I.2.3 suite)
  - pas de tableaux non-contraints
  - tableaux dont la dimension est une variable introduits dans le standard C99
- Les chaînes de caractères (I.2.4)
  - le type chaîne de caractères n'existe pas : utiliser un tableau de `char` dont le caractère `null('\0')` représente la fin de la chaîne
  - `char logo[10]` **contient 9 caractères maximum ! !**
  - initialisation caractère par caractère ou globalement
  - Exemples :
    ```
    char logo[5]="INSA";
    char logo[5]={'I','N','S','A','\0'};
    ```
  - **Initialisations uniquement possibles lors de la définition ! :**
    `char jamais[10]; jamais="INSA"` : **IMPOSSIBLE**

Variables et opérateurs (chap. I)

- Les opérateurs arithmétiques(I.3.1)
  - 5 opérateurs : `+, -, *, /, %`
  - les deux opérandes doivent être du même type : conversion implicite ou explicite
  - conversion explicite : `(double)f, (int)'A'`
    `(float)2/3` vaut 0.6777 MAIS `(float)(2/3)` vaut 0
  - conversion implicite : l'opérande de type le plus faible doit être converti dans le type de l'autre opérande avant de commencer les calculs :
    `char < short < int < long < long long`
    `float < double`
    Exemple : `'A' + 2` est de type `int`
  - conversion au moment de l'affectation :
    `int Entier = 4.8;` (`Entier` vaut 4)

Variables et opérateurs (chap. I)

- Les opérateurs logiques (I.3.2)
  - 3 opérateurs : `&&`, `||`, `!` (et, ou, négation)
- Les opérateurs relationnels (I.3.3)
  - 6 opérateurs : `>`, `>=`, `==`, `!=`, `<=`, `<`
- Les opérateurs bit à bit (I.4)
  - 4 opérateurs : `&`, `|`, `^`, `~` (et, ou inclusif, ou exclusif, complément à 1)
  - Exemple : `Masque = Masque & 0xF0`
    Les 4 bits de poids faible de `Masque` à 0, 4 bits de forts conservés

Variables et opérateurs (chap. I)

- Les opérateurs de décalage (I.5)
  - 2 opérateurs : `>>`, `<<` (décalage à droite, décalage à gauche)
  - extension du signe dans les décalages à droite (`-2 >> 1` vaut `-1`)
  - introduction de 0 lors de décalage à gauche
- Les opérateurs d'incrémentation et décrémentation (I.6)
  - `val++` équivaut à `val=val+1` (ou `++val`)
  - `val--` équivaut à `val=val-1` (ou `--val`)
  - ne peut être utilisé que pour des variables considérées isolément
  - la position de l'opérateur (avant ou après `val`) indique si l'utilisation de la valeur de `val` se fait avant ou après l'opération d'incrémentation ou de décrémentation : on parle de pré-incrémentation ou post-incrémentation (idem pour décrémentation)

Variables et opérateurs (chap. I)

- Les opérateurs d'incrémentation et décrémentation (I.6 suite)

```
Val = 5
Page = ++Val * 3; /*Val=6 Page=18 */
Delta = (Val++)* 3; /*Val=7 Delta=18 */
```

- L'affectation (I.7)
  - se réalise ainsi : `variable = expression`
  - affectation entre types différents est possible mais dangereuse ! !
  - variante d'écriture :
    `val = val <operateur> <operande>` est équivalent à :
    `val <operateur>= <operande>`
    Exemple : `val=val-6 <=> val-=6`

Instructions itératives (chap. II)

- La boucle `for`(II.1)

```
for ( <pre-actions>; <conditions>; <post-actions>)
  <corps-de-la-boucle>
```

  - `pre-actions` : effectuées une seule fois au début de la boucle ; en général, initialisation de variable (séparation de pré-actions par virgule)
  - `conditions` : en général, la condition de poursuite de la boucle
  - `post-actions` : effectuées à la fin de chaque itération (séparation des post-actions par virgule)
  - Exemple : `for (Ind=0;Ind<6;Ind++)`
    6 itérations successives pour `Ind` allant de 0 à 5
  - `corps-de-la-boucle` : une seule instruction ou un ensemble d'instructions commençant par { et finissant par }

Instructions itératives (chap. II)

- La boucle `for`(II.1 suite)
  - Exemples :

```
for (Ind=0;Ind<6;Ind++) a+=3;
for (Ind=0;Ind<6;Ind++) {
  a+=3;
  b-=5;
}
```

  - imbrication de boucles possibles
  - **Remarque :**
    `for (int Ind=0;Ind<5;Ind++)` **n'est possible qu'en C99 !**

Instructions itératives (chap. II)

- La boucle `while` (II.2)

```
while (<condition>)
   <corps-de-la-boucle>
```

  - même remarque que la boucle `for` pour le `corps-de-la-boucle`
  - il est possible que le corps de la boucle ne soit jamais exécuté
- La boucle `do-while` (II.3)

```
do
  <corps-de-la-boucle>
while (<condition>);
```

  - la boucle est effectuée au moins une fois

Instructions de contrôle (chap. II)

- L'instruction `if-else` (II.4)

```
if (<condition>)
  <corps-alors>
[ else
  <corps-sinon> ]
```

  - si `condition` est évaluée à vrai, alors `corps-alors` est exécuté, sinon `corps-sinon` est exécuté
  - attention aux ambiguités sur les conditions imbriquées => utilisation des accolades
  - **l'erreur classique du C : utiliser = à la place de == dans les conditions**
    `if (Longueur=1)` est toujours vrai ! ! et le compilateur ne génère aucune erreur par défaut puisque l'expression est correcte ! !

Instructions de contrôle (chap. II)

- L'opérateur conditionnel ? (II.4.5)

```
<condition> ? <expr-si-vrai> : <expr-si-faux>
```

  - si `condition` est évaluée à vrai, la valeur de cette opération est `expr-si-vrai`, sinon la valeur est `expr-si-faux`
  - Exemples :
    `Max = (X > Y)? X : Y; Min = (X < Y)? X : Y;`

Instructions de contrôle (chap. II)

- L'instruction `switch` (II.5)

```
switch (<expression>) {
  case <etiquette_1> : <instructions_1>
  case <etiquette_2> : <instructions_2>
                       .
                       .
  case <etiquette_n> : <instructions_n>
  default : <instructions_d>
}
```

  - `expression` est évaluée et comparée aux étiquettes ; s'il y a égalité avec l'étiquette `i`, les instructions `instructions_i ... instructions_n` et `instructions_d` sont exécutées (!) ; si aucune étiquette ne correspond, les instructions `instructions_d` sont exécutées
  - `expression` est de type *entier* ou *caractère* ; peut être une constante ou construite à partir d'opérateurs

Instructions de contrôle (chap. II)

- L'instruction `break` (II.6)
  - instruction de déroutement pour sortir avant terme d'une boucle (`for`, `while` ou `do-while`) ou d'un `switch`

```
for (i=0;i<5;i++) {
   ...
   if (i==butoir) break;
}
```

```
switch (i) {
   case 0:
      printf("coucou");
      break;
   case 1:
      ...
}
```


Instructions de contrôle (chap. II)

- L'instruction `continue` (II.7)
  - instruction de déroutement uniquement dans le corps des boucles ; elle permet de passer au début de l'itération suivante
  - Exemple :

```
for (i=0; i<10; i++) {
   // n'affiche pas l'entier 6
   if (i==6) continue;
   printf("%d\n",i);
}
```

Les fonctions (chap. III)

- Les fonctions permettent de modulariser un programme en le découpant en actions simples (III.1)
  - programmation structurée
  - compilation séparée, création de bibliothèques (boîtes à outils)
- Tout est fonction : pas de distinction entre procédure et fonction comme en ADA par exemple.
- Définition de fonction :

```
<type_resultat> Nom ( <declaration_arguments>)
{
  <corps de la fonction>
}
```

- Le résultat peut être un type (`int`, `char`, ...) ou `void` (équivalent d'une procédure ADA)

Les fonctions (chap. III)

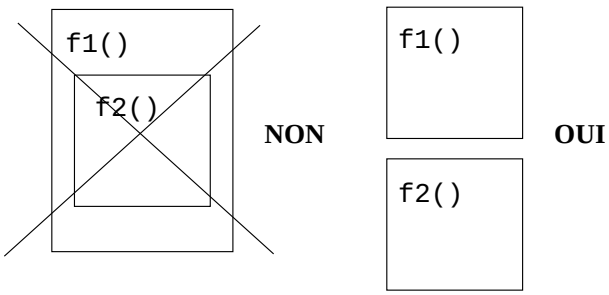
- Exemples :

```
double Pi ()          int Calcul(int val1)     void Rien()
{                     {                        { }
   ...                   ...
}                     }
```

- L'instruction `return` permet de quitter une fonction (et éventuellement de renvoyer le résultat) :
  `return;` (cas d'une fonction de type `void`)
  `return <expression>;` (autres cas)
- Plusieurs `return` sont possibles à l'intérieur d'une fonction mais un seul `return` est meilleur (lisibilité du programme, source d'erreur)
- La déclaration des arguments est une suite de déclarations de variables (nom et type de la variable) ; les déclarations sont séparées par des virgules
- Le stockage des arguments se fait dans la *pile* (sur un système 32 bits, ça peut être différent sur un système 64 bits)

Les fonctions (chap. III)

- Pas d'imbrications de fonctions (III.2)



- L'appel d'une fonction peut constituer une instruction simple ou être utilisée dans une expression
  `a = Racine(b);`
  `aire = Pi()* Rayon * Rayon;`
  Attention ! les parenthèses sont obligatoires en absence d'arguments
- `f1(f2())` est correct

Les fonctions (chap. III)

- Deux types de passage de paramètres : par *valeur* et par *adresse* (III.3)
  - par *valeur* : à l'appel de la fonction, une copie de la valeur de l'argument est passée à la fonction, elle ne peut donc **PAS** modifier l'argument original
  - par *adresse* : la fonction utilise directement l'argument (son *adresse en mémoire* est passée à la fonction appelée) : utilisation de **pointeurs**
- Les tableaux sont toujours passés par adresse
- Les autres variables peuvent être passées par valeur ou pas adresse : dans le cas du passage par adresse, **c'est au programmeur à passer l'adresse de l'argument en paramètre** (*les puristes parlent de passage par valeur de l'adresse et non de passage par adresse qui est réservé au passage implicite d'adresse*)

Les fonctions (chap. III)

- Déclaration - définition de fonctions (III.4)
  - si, dans une fonction f2, la fonction f1 est appelée, il faut que f1 ait été *définie* auparavant ou au moins *déclarée*
  - Déclaration :

```
<type> <nom_fonction> ( <type>, <type>, ...)
Ex : int f(int) ;
```

  - Définition :

```
int f(int x)
{
  return x*2;
}
```

  - une fonction appelée doit être déclarée avant la définition appelante mais le compilateur ne génère pas d'erreur par défaut !

Les fonctions (chap. III)

- Exemples :

```
int f1(char b)
{

  ...
}


int main(void)
{
  int i;

  i=f1('c');
}
```

f1 définie avant `main`

```
int f1(char);
int f2(double b);
{
  int i;
  i=f1('c');
}

int f1(char b)
{
  ...
}

int main(void)
{
  ...
}
```

f1 définie après f2 mais déclarée avant f2

Les fonctions (chap. III)

- Prototype valide pour la fonction `main` :

```
int main(void)
```

- Prototype **NON VALIDE** pour la fonction `main` :

```
void main(void) <- Beurk !
```

Ecriture et évaluation des expressions (chap. IV)

1	`() [] -> .`	expressions, structures
2	`! - ++ -- - * & (type) sizeof`	unaires, forçage
3	`* / %`	mult., div. modulo
4	`+ -`	addition et soustraction
5	`<< >>`	décalage gauche/droite
6	`< <= > >=`	comparaisons
7	`== !=`	égalité, différence
8	`&`	ET bit à bit
9	`^`	OU exclusif bit à bit
10	`|`	OU bit à bit
11	`&&`	ET logique
12	`||`	OU logique
13	`?:`	opérateur conditionnel
14	`= += -= *= /= %= <<= >>= &= ^=`	affectations
15	`,`	opérateur virgule

Ecriture et évaluation des expressions (chap. IV)

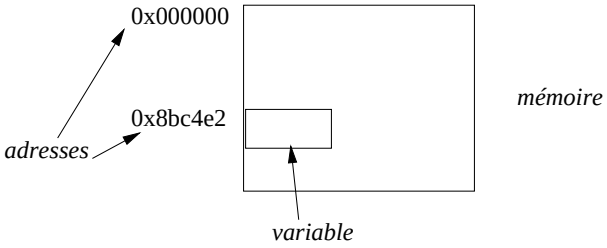
- Exemples (IV.1.2)

```
p = m << (d - n + 2) -1
equivalent a :
p = m << (d - n + 2 - 1)
equivalent a :
p = m << d - n + 2 - 1
```

- Les expressions logiques (IV.1.3)
  - Les opérateurs logiques sont des opérateurs progressifs : si le premier opérande donne déjà la valeur de l'expression, le second opérande n'est pas évalué
  - `x || y ++ : y n'est incremente que si x est faux`

Les pointeurs (chap. V)

- Un pointeur est une variable qui contient l'adresse d'une autre variable (V.1) (toujours 4 octets sur un système 32 bits, toujours 8 octets sur un système 64 bits)



- Déclaration d'un pointeur :

```
<type> * pointeur ;
Ex : int * pEntier;
```

- L'adresse d'une variable est obtenue à l'aide de l'opérateur unaire &

```
int * pEntier; int Pointee;
pEntier = &Pointee;
```


Les pointeurs (chap. V)

- Obtenir l'objet sur lequel pointe un pointeur : `* pointeur ;`

```
int * pEntier; int Pointee; int Largeur;
pEntier = &Pointee;
Largeur = *pEntier;
```

- Tableau et pointeur (V.2)

```
int Salaires[12]; int * pActif;
```

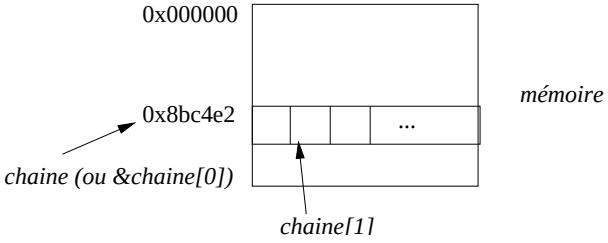
  - `Salaires` est l'adresse du 1er élément du tableau : équivalent à `& Salaires[0]`
  - `pActif = Salaires` est correct !
  - `*(pActif + 4)` est le 4ème élément du tableau et équivaut donc à `Salaires[4]` (`pActif + 4` provoque un déplacement en mémoire de 12 octets !)
  - **Attention :** : `Salaires` est une constante alors que `pActif` est une variable !

Les pointeurs (chap. V)

- Tableau et pointeur (V.2 suite)
  - Exemples d'opérations valides :

```
int Mois; Mois = Salaires[0];
pActif = Salaires + 4; pActif = &Salaires[4];
```

- Une chaîne de caractères est un tableau donc son nom représente l'adresse du premier caractère



- Opérations et pointeurs (V.3)
  - affectation, 6 opérateurs relationnels, incrémentation, décrémentation
  - addition et soustraction avec entier

Les pointeurs (chap. V)

- Passage des arguments de fonctions "par adresse" (V.4)
  - un tableau est toujours passé par adresse

```
// f1 et f2 sont equivalentes
void f1 (int Table []);
void f2 (int * Table);
```

[] et * sont équivalents en tant que paramètres formels => représentent une adresse

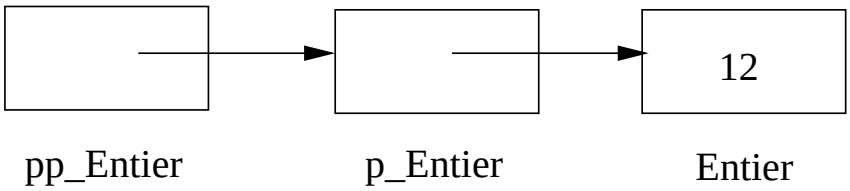
Les pointeurs (chap. V)

- Passage des arguments de fonctions "par adresse" (V.4 suite)
  - pour les variables scalaires

```
void f1 (double * ref_double) {
  // ref_double : adresse
  // * ref_double est le nombre reel
  // que l'on peut modifier
  * ref_double += 2.0;
}
int main()
{
  double Nb_Reel=0.0;
  //explicitement passer l'adresse de Nb_Reel
  f1(&Nb_Reel);
}
```


Les pointeurs (chap. V)

- Les indirections multiples (V.6)




```
int Entier; int * pEntier; int ** ppEntier;
ppEntier = & Entier incorrect !
```

ppEntier est un pointeur sur un pointeur et non sur un entier : notion de profondeur

Les pointeurs (chap. V)

- Les tableaux de pointeurs (V.7)

```
int * Mois[12];
```

  - `Mois` est un tableau de 12 pointeurs sur un entier
  - `Mois` est l'adresse du premier élément du tableau
  - `Mois` est l'adresse d'un pointeur
- Les tableaux de chaînes de caractères

```
char * gaz[] = {"Vide", "Pression", "Vapeur",
                "Bars"};
```

  est un tableau de 4 chaînes de caractères de longueur automatiquement ajustée

Les pointeurs (chap. V)

- Allocation de mémoire (V.7)
  - la déclaration d'un pointeur provoque de la réservation mémoire pour le pointeur, pas pour la variable sur laquelle il pointe (ou pointera) ! !
    `int * ptr; char * ch;`
    `* ptr = 4; strcpy(ch, "on est les champions");`
    => provoque une erreur à l'exécution ! !
  - possibilité d'utiliser l'allocation de mémoire grâce aux fonctions `malloc` et `calloc`
    `void * malloc(int taille)`
    => provoque la réservation en mémoire d'un tableau de `taille` octets contigues
    `void * calloc (int nb, int taille)`
    => provoque la réservation en mémoire d'un tableau de `nb` éléments de `taille` octets
    => la mémoire utilisée pour l'allocation dynamique est le *tas*
  - libération de la mémoire à l'aide de la fonction `free`

Fonctions et pointeurs (chap. VII)

- Fonction renvoyant un pointeur (VII.1)
  - déclaration/définition (VII.1.1)
    `<type> * fonct()`
    => `fonct` est une fonction qui renvoie l'adresse d'un objet du type indiqué
  - utilisation (VII.1.2)

```
int * Appelee()
{
  ...
}
void Appelante()
{
  int * retour;
  ...
  retour = Appelee();
  ...
}
```

Fonctions et pointeurs (chap. VII)

- Fonction renvoyant un pointeur (VII.1 suite)
  - problème des adresses périmées (VII.1.4)
    => lié à la duré de vie des variables : **ATTENTION à ne pas renvoyer une adresse périmée**

```
int * Suite(void)
{
  int ephemere[10];
  ...
  return ephemere;
}
```

```
int Appelante(void)
{
  int * retour;

  retour=Suite();
}
```

NON ! ! La fonction `Suite` **renvoie l'adresse d'un tableau qui est local à cette fonction. Dès la sortie de la fonction** `Suite`**, la mémoire associée à cette adresse est libérée**.

Fonctions et pointeurs (chap. VII)

- Appel indirect de fonctions (VII.2)
  - le nom d'une fonction représente l'adresse de la fonction
    => cette adresse peut être attribuée à un pointeur, passée en paramètre de fonction, ...
  - déclaration d'un pointeur de fonction
    `type ( * ptrfct )(type, type, ...);`
    => **parenthèses autour de** `ptrfct` **obligatoires ! !**
    Ex : `double (* Math)(int , int, double);`
- Tableaux de pointeurs de fonctions (VII.3)
  - déclaration d'un tableau de pointeurs de fonctions
    `type ( * tabptrfct[5] )(type, type, ...);`
    Ex : `double ( * Tab[4])(double, double);`

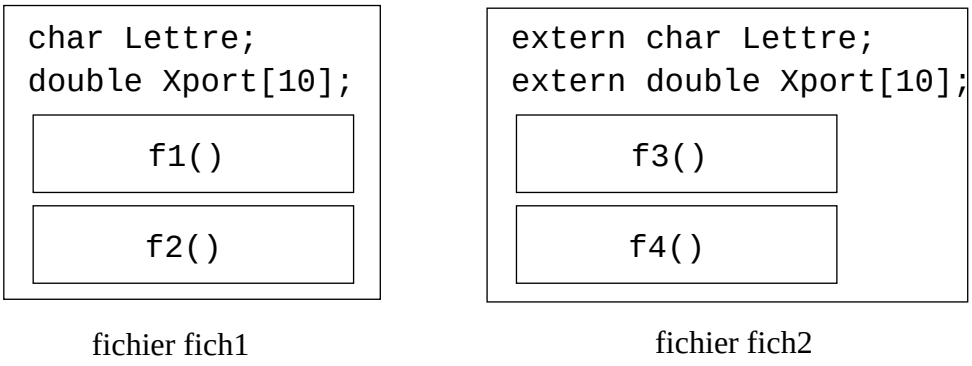
Les classes de mémorisation (chap. VI)

- Deux types de variables : *internes et externes* (VI.1)
  - une variable *interne* est définie à l'intérieur d'un bloc (en général d'une fonction)
  - une variable *externe* est définie hors de tout bloc
- Portée d'une variable
  - une variable *interne* porte sur tout le bloc dans lequel elle est définie
  - une variable *externe* porte au moins sur tout le fichier dans lequel elle est définie
- Visibilité d'une variable
  - la redéclaration locale (à l'intérieur d'une fonction) d'une variable masque cette variable sans en altérer la portée
    Exemple :

```
int Redefinie; /* variable de niveau 0 */
int main(void)
{
  char Redefinie; //variable de niveau 1
  //  redefinition de Redefinie
  // => Redefinie definie en 0 n'est plus visible
  ...
```

Les classes de mémorisation (chap. VI)

- Augmentation de la portée des variables externes
  - pour rendre une variable de niveau 0 (niveau 0 seulement) visible dans un second fichier, utilisation de la clause `extern`



fichier fich1 fichier fich2

=> portée de `Lettre` et `Xport` : fich1 et fich2

Les classes de mémorisation (chap. VI)

- Durée de vie des variables (VI.2)
  - la durée de vie d'une variable *interne* est réduite au temps d'exécution du bloc dans lequel elle est déclarée => doit être initialisée à chaque appel de fonction
  - la durée de vie d'une variable *externe* est la durée de vie du programme
- Durée de vie des variables *internes statiques*
  - le mot-clé `static` rend la mémorisation d'une variable interne permanente

```
void Compter()
{
   // valeur de i conservee d'invocation
   // en invocation
   static int i=0;

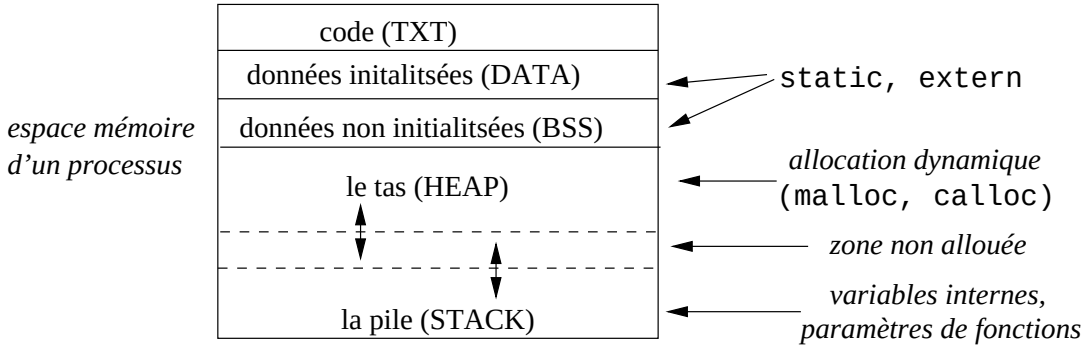
   printf("%d\n",i);
   i++;
}
```

Les classes de mémorisation (chap. VI)

- Durée de vie des variables *externes statiques*
  - le mot-clé `static` pour une variable *externe* de niveau 0 limite la portée de cette variable au fichier dans lequel elle est déclarée
- Fonctions statiques
  - le mot-clé `static` pour une fonction rend cette fonction uniquement visible dans le fichier dans lequel elle est déclarée
- Initialisation des variables (VI.4)
  - peut se faire au moment de la déclaration :
    `int var=3; char Lettre = 'A'; char Chaine[]="ca roule?";`
  - en absence d'initialisation, les variables externes et statiques sont mises à zéro
  - en absence d'initialisation, les variables internes peuvent avoir n'importe quelle valeur
    $=>$ **Initialiser les variables ! !**

Les classes de mémorisation (chap. VI)

- Stockage en mémoire des variables - le cas d'Unix (VI.2.6)
  - les variables internes sont stockées dans la pile (de même que les paramètres de fonctions)
  - les variables statiques et externes sont stockées dans un segment de données réservé dans l'espace d'adressage du processus (DATA pour les données initialisées, BSS pour les données non initialisées)



Les structures (chap. VIII)

- A la façon du RECORD d'ADA, il est possible de créer des structures de données comprenant plusieurs champs de types différents
- Déclaration d'une structure (VIII.1.1)

```
struct <etiquette>
{
   /* corps */
};
```

- Déclaration de variables dont le type est une structure :
  `struct <etiquette> a,b ; /*2 variables */`
  => pour a et b, le compilateur réserve en mémoire la place nécessaire

Les structures (chap. VIII)

- Une structure est composée de champs (VIII.1.2) :

```
struct Entreprise
{
  char Sigle[10];
  char Adresse[40];
  int Telephone;
  // Sigle , Adresse et Telephone sont les 3 champs
};
```


Les structures (chap. VIII)

- Accéder à un champ d'une structure (VIII.1.2 suite)
  - utilisation de l'opérateur . (point)

```
struct Entreprise Ingenieurs;
strcpy(Ingenieurs.Signe,"INSA");
Ingenieurs.Telephone = 61559513;
```

  - possibilité d'initialiser lors de la définition

```
struct Entreprise Ingenieurs = {"INSA", "RANGUEIL",
61559513};
```

- Deux structures du même type peuvent être affectées (VIII.1.3)

```
struct Entreprises Ingenieurs, Facultes;
Ingenieurs = Facultes;
```

Impossible pour les tableaux !

Les structures (chap. VIII)

- Les structures peuvent être imbriquées (VIII.1.4)

```
struct Personnel {
  unsigned int Age;
  struct Entreprise Ecole;
};
```

- Structures et pointeurs (VIII.2)

```
struct Modele
{
  int a;
  char c;
  double f;
};
struct Modele VarStruct;
struct Modele * pStruct;
```


Les structures (chap. VIII)

- Structures et pointeurs (VIII.2 suite)
  - l'accès aux champs se fait par l'opérateur ->

```
pStruct = &VarStruct;
char d = pStruct->c;
```

=> équivalent à `(*pStruct).c` mais moins utilisé

Les structures (chap. VIII)

- Les tableaux de structure (VIII.3)

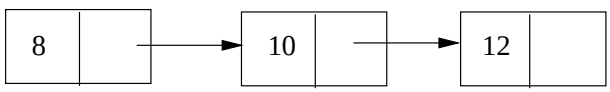
```
struct Mois {
  char Nom[20];
  int Jours;
};
```

`struct Mois TabMois[12]; //tableau de 12 structures`
`TabMois` est l'adresse de la première structure du tableau
=> `struct Mois * pMoisCourant = TabMois` est donc correct !
`pMoisCourant ++` pointe sur la structure suivante dans le tableau (déplacement en mémoire de la taille de la structure `Mois`)

Les structures (chap. VIII)

- Les structures se référant à elle-même (VIII.4)
  - il est interdit de déclarer un modèle de structure se contenant elle-même, mais un de ses champs peut être un pointeur sur une structure de même type

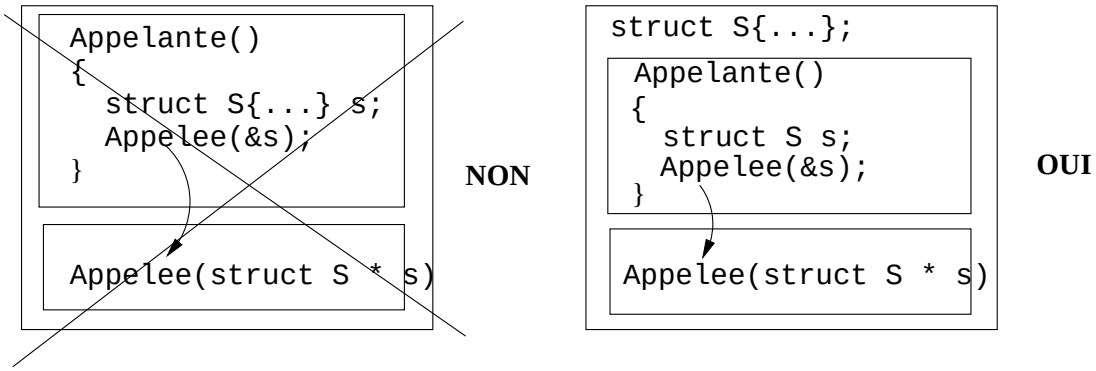
```
struct Liste {
  double valeur;
  struct Liste * suivant;
};
```



  - permet de gérer un ensemble de valeurs dont on ne connaît pas le nombre a priori => nécessité d'allouer de la mémoire de façon dynamique : opérateurs `malloc` et `calloc`

Les structures (chap. VIII)

- Structures et fonctions (VIII.5)
  - utiliser de préférence le passage par adresse d'une structure lorsqu'elle est passée en paramètre d'une fonction, de façon à ne pas encombrer la mémoire
  - la fonction appelée doit connaître la définition de la structure



  - on peut importer un modèle de structure (de la forme `extern struct S s;`) => il faut connaître la définition de la structure !

Les structures (chap. VIII)

- Structures et fonctions (VIII.5 suite)
  - pour partager une structure entre plusieurs fichiers, déclarer cette structure dans un fichier d'en-tête (fichier dont le suffixe est `.h`), et inclure ce fichier (par un mécanisme `#include` dans chacun des fichiers qui utilisent la structure

```
struct S {
  ...
};
```

fich.h

```
#include "fich.h"
```

```
#include "fich.h"
```

  - une fonction peut renvoyer une structure ou un pointeur de structure

Les structures (chap. VIII)

- Le constructeur typedef (VIII.6)
  - permet d'attribuer un nom symbolique à un type simple ou composé
    => ne provoque aucune réservation mémoire
    => ne crée pas de nouveau type tel qu'en ADA

```
typedef unsigned char UCHAR;
typedef char PHRASE[21];
UCHAR lettre = 'A';
PHRASE message = "Salut les poteaux";
```

- Le constructeur typedef et les structures
  - permet d'alléger l'utilisation des structures

```
struct Modele {
  ...
};
typedef struct Modele NEW;
NEW var;
```

ou

```
typedef struct {
...;
} NEW;

NEW var;
```


Les structures (chap. VIII)

- Les champs de bits (VIII.7.1)
  - un champ de bits est une structure où chaque champ contient un nombre quelconque de bits (très utilisé dans les *drivers* par exemple pour décrire des registres de composants)

```
struct Registre
{
  unsigned char PG : 1;        /* 1 bit */
  unsigned char I : 1;         /* 1 bit */
  unsigned char R : 1;         /* 1 bit */
  unsigned char C : 1;         /* 1 bit */
  unsigned char NB : 2;        /* 2 bits */
  unsigned char LRU : 2;       /* 2 bits */
};
```

  - accès aux champs de façon classique pour une structure
- Les unions (VIII.7.2)
  - une variable de type union peut contenir à différents moments des objets de types différents ; la taille de ces objets peut être différente

Les structures (chap. VIII)

- Les unions (VIII.7.2 suite)

```
union Nombre {
  int Entier;
  double Reel;
};
```

  - toute variable de type `union Nombre` aura la taille d'un double !
  - **Attention : une seule valeur à la fois ! !**

```
union Nombre nb;
nb.Entier = 3;
/* est interprete comme un entier */
nb.Reel = 5.0;
/* est interprete comme un double */
```


Les structures (chap. VIII)

- Les unions (VIII.7.2 suite)
  - les unions peuvent être employés pour créer des structures avec une partie variante (à la manière des RECORD à discriminants ADA)

```
struct Modele {
  char * Nom;
  int Entier;
  union {
    int Val_Ent;
    double Val_Reel;
  } Champ_Union;
}
```

Les structures (chap. VIII)

- Les unions (VIII.7.2 suite)
  - les unions permettent d'accéder à une même valeur de diverses manières

```
union Ref_Donnees {
  char B[4];
  short W[2];
  long L;
};
```

  => simulation d'un registre d'un processeur qui peut être considéré comme 1 mot long (32 bits), deux mots (16 bits) ou quatre octets (valable sur architecture 32 bits)

Les structures (chap. VIII)

- Les énumérations (VIII.7.3)
  - permet de construire un type énuméré

```
enum Jours {lun, mar, mer, jeu, ven, sam, dim};
enum Jours un_jour;
un_jour=lun;
```

  - taille de type `int` utilisé par défaut
  - il est possible cependant de choisir les valeurs de chaque élément de type ou d'écrire explicitement une rupture de séquence :

```
enum Operateur {plus = '+', moins = '-'};
enum Jour {lun, mar, mer, jeu=4, ven, sam};
            0    1    2    4     5    6
```

Les entrées/sorties (chap. IX)

- La bibliothèque *stdio* permet d'accéder aux fichiers et de les manipuler à l'aide d'un ensemble de fonctions ; tout programme qui utilise cette biblioothèque doit inclure le fichier `stdio.h` avec la directive `#include <stdio.h>`
- Ouverture et fermeture de fichier (IX.1-IX.2)

```
FILE * fopen (char * nomfic, char * mode)
int fclose (FILE * pFic)
```

  - cette fonction renvoie l'adresse d'une structure de type `FILE`
  - `mode` représente la manière dont est ouvert le fichier (`"r"` : en lecture, `"w"` : en écriture, etc)
  - si le fichier n'existe pas et le mode est écriture, le fichier est créé
  - si le fichier n'existe pas et que le mode est lecture, la constante `NULL` est renvoyée

=> à chaque fichier ouvert est associé un tampon d'entrée/sortie ; cette association est appelée un *flux (stream)* : on parle d'entrées/sorties *bufferisées*

Les entrées/sorties (chap. IX)

- Accès par caractère (IX.3)

```
getc (FILE * ) et putc (char, FILE *)
```

  - macro-instructions qui permettent de lire/écrire un caractère dans un fichier (`getc`/`putc`)
  - `getc` lit un caractère dans le flux référencé par le pointeur de type `FILE *`, ce pointeur est automatiquement post-incrémenté
  - quand on lit le dernier caractère du fichier, la constante `EOF` est retournée
- Flux d'entrée, de sortie, d'erreur (IX.4)
  - l'entrée, la sortie et la sortie d'erreur standards sont aussi 3 flux de type `FILE *` respectivement dénommés : `stdin, stdout, stderr`
  - les instructions d'entrée au clavier et d'affichage à l'écran sont donc : `getc(stdin)` et `putc(caractere, stdout)` => macro-instructions équivalentes : `getchar()` et `putchar (caractere)`

Les entrées/sorties (chap. IX)

- Accès par lignes (IX.5)

```
char * fgets (char * Ligne, int MAX, FILE * pFic)
```

  - lit la ligne suivante du fichier retourné par `pFic`, la range en mémoire à partir de l'adresse `Ligne` (**qui doit être de taille suffisante**) et ajoute le caractère `\0`
  - la longueur de la ligne doit être au plus de `MAX-1` caractères
  - la fin de ligne est marquée par le caractère `'\n'`
  - la constante `NULL` est renvoyée en cas de problème

```
int fputs (char * Ligne, FILE * pFic)
```

  - écrit la chaîne `Ligne` dans le fichier référencé par `pFic`
  - pas de retour à la ligne automatique
  - deux macro-instructions `gets` et `puts` travaillent avec l'entrée standard et la sortie standard

Les entrées/sorties (chap. IX)

- Entrées/sorties par blocs (IX.6)

```
int fread (void * Tampon, int Taille_Bloc,
           int Nombre, FILE * pFic)
int fwrite (void * Tampon, int Taille_Bloc,
            int Nombre, FILE * pFic)
```

  - accès le plus rapide dès lors que les blocs sont assez gros
  - `Tampon` est l'adresse en mémoire d'un tableau de caractères (origine ou destination) ; dans le cas d'un tableau destination, **c'est au programmeur à réserver l'espace mémoire suffisant !**
  - `Taille_Bloc` est la taille d'un bloc en octets
  - `Nombre` est le nombre de blocs à lire
  - ces 2 fonctions retournent le nombre de blocs transférés

Les entrées/sorties (chap. IX)

- Gestion du tampon du flux (IX.7)

```
int fseek (FILE * pFic , int Deplac, int Origine)
```

  - déplace le pointeur dans le tampon de `Deplac` octets par rapport à `Origine` suivant la convention suivante :
    - si `Origine` vaut 0, le déplacement est compté par rapport au début du fichier
    - si `Origine` vaut 1, le déplacement est compté par rapport à la position courante du pointeur
    - si `Origine` vaut 2, le déplacement est compté en partant de la fin du fichier

```
int ftell (FILE * pFic)
```

  - donne la valeur en octets de la position du pointeur dans le tampon, par rapport au début du fichier

```
FILE * f;
f=fopen("/etc/hosts","r");
fseek(f,0,2);                    Que fait ce programme ?
printf("%d\n",ftell(f));
fclose(f);
```

Les entrées/sorties (chap. IX)

- Gestion du tampon du flux (IX.7 suite)

```
int fflush(FILE * pFic)
```

  - vide la tampon associé au flux `pFic`, le fichier reste ouvert
  - permet notamment de synchroniser les lectures et écritures sur les péripériques standards :
    `fflush (stdout)` : vide le tampon de sortie standard
  - très utile lorsqu'un programme plante et qu'on arrive pas à afficher les messages d'erreurs

Les entrées/sorties (chap. IX)

- Entrées/sorties formatées (IX.8.1)

```
int printf (char * format [, <arguments>]);
```

  - cette fonction met en forme les arguments et les envoie à la sortie standard
  - la chaîne `format` contrôle l'affichage ; cette chaîne peut contenir des caractères ordinaires et des spécifications de conversion :
    - `%c` : un caractère
    - `%s` : une chaîne de caractères
    - `%d` : un entier codé en décimal
    - `%x` : un entier codé en hexadécimal
    - `%f` : un réel ...

```
printf("Bonjour !");
// affichage entier puis hexadecimal
printf("%d %x",val,val);
// affiche bonjour et un retour chariot
printf("Bonjour \n");
```

Les entrées/sorties (chap. IX)

- Entrées/sorties formatées (IX.8.2)

```
int scanf(char * format [, <arguments>]);
```

  - la conversion s'arrête dès lors que l'on rencontre un séparateur implicite (espace, tabulation ou interligne), un séparateur explicite (précisé dans le format) ou un caractère incompatible avec le type spécifié dans le format
  - les arguments étant des paramètres de sorties, il faut obligatoirement les passer par adresse !

```
int Entier; char Article[20];
scanf("%d",&Entier);
/* un tableau est deja une adresse ! */
scanf("%s",Article);
/* si on entre au clavier "La ville Rose", Article
     vaudra "La" */
```


Les entrées/sorties (chap. IX)

- Entrées/sorties formatées (IX.8.3)

```
int fscanf ( FILE * pFic, char * format
             [, <arguments>]);
```

  - lit dans le flux pointé par `pFic` les données dont le nombre et le type sont précisés dans la chaîne de format
  - `scanf` est la fonction `fscanf` avec `stdin` comme premier paramètre

Les entrées/sorties (chap. IX)

- Entrées/sorties formatées (IX.8.4)

```
int fprintf (FILE * pFic, char * format
             [, <arguments>]);
```

  - écrit dans le flux pointé par `pFic` les données dont le nombre et le type sont précisés dans la chaîne de format
  - `printf` est la fonction `fprintf` avec `stdout` comme premier paramètre

Les entrées/sorties (chap. IX)

- Fonctions de conversion en mémoire (IX.8.5)

```
int sprintf (char * chaine, char * format
             [, <arguments>]);
```

  - exécute les mêmes conversions que `printf` mais range le résultat dans une chaîne de caractères en mémoire (ici `chaine`)

```
int sscanf (char * chaine, char * format
            [, <arguments>]);
```

  - extrait d'une chaîne de caractères en mémoire (`chaine`) selon le format indiqué par `format`

Le préprocesseur (Annexe A)

- Lors de chaque compilation, le compilateur C fait appel lors du premier passage au préprocesseur
- Le préprocesseur est un traitement de texte spécialisé qui permet
  - la macro-substitution
  - l'inclusion de fichiers
  - la compilation conditionnelle
- Le préprocesseur n'obéit qu'aux lignes commençant par le caractère #, qui doit être placé en première colonne

Le préprocesseur (Annexe A)

- La macro-substitution (A.1)
  - permet de remplacer dans le texte du fichier, un identificateur (mot, opérateur, symbole, ...) par un texte de substitution

```
#define identificateur texte_de_substitution
```

  - Ex : `#define TAILLE 100`
    ⇒ après le passage du préprocesseur, TAILLE est remplacé par 100 dans le fichier
  - il est d'usage d'utiliser les majuscules pour les constantes
  - permet d'alléger le programme
  - si le texte de substitution s'écrit sur plusieurs lignes, on doit utiliser en fin de ligne le caractère \ (sauf pour la dernière)

```
#define RC putchar('\n');
#define BEGIN {
#define LONG "Ca c'est un texte qui tient meme \
     pas sur une seule ligne"
```


Le préprocesseur (Annexe A)

- La macro-substitution (A.1 suite)
  - possibilité de définir des macro-instructions

```
#define ident (arg1, ..., argn)
    expression_de_substitution
```

Ex : `#define CUBE(X) X * X * X`
`int Puiss3, val = 4;`
`Puiss3 = CUBE(val)`
⇒ la substitution effectuée est : `Puiss3 = val * val * val`
⇒ `CUBE(val + 1)` donne
`val + 1 * val + 1 * val + 1 = 13!!!`

  - dans le cas d'expressions numériques, utiliser les parenthèses autour de l'expression et autour de chaque argument !
    `#define CUBE(X) ((X) * (X) * (X))`

Le préprocesseur (Annexe A)

- La macro-substitution (A.1 suite)
  - dans le cas d'instructions, utiliser des accolades autour de l'expression

```
#define FATAL(num) \
  { fprintf(stderr, "\nErreur %d\n",(Num));\
    exit(1);\
  }
```


Le préprocesseur (Annexe A)

- L'inclusion de fichiers (A.2)
  - le contenu d'un fichier texte peut être inséré dans le texte d'un autre fichier, grâce à la directive `include`
  - les fichiers inclus contiennent en général des déclarations de fonctions, de types, de constantes ; leur suffixe est `.h`
  - un grand nombre de fichiers standards à inclure se trouvent dans le répertoire `/usr/include` ; leur inclusion s'effectue ainsi : `#include <stdio.h>`
  - les chevrons <,> indiquent qu'il faut chercher le fichier dans un ou des répertoires standards des fichiers à inclure
  - `#include <sys/tty.h>` provoque l'inclusion du fichier `/usr/include/sys/tty.h`
  - l'inclusion de fichiers créés par l'utilisateur se fait en utilisant les guillemets : `#include "perso.h"` => le fichier est alors recherché dans le répertoire courant en supplément des répertoires standards

Le préprocesseur (Annexe A)

- L'inclusion de fichiers (A.2 suite)
  - possibilité à la compilation de spécifier un répertoire où se trouvent des fichiers à inclure : `gcc -I ~/include prog.c`
  - le compilateur va utiliser le répertoire `~/include` comme répertoire de recherche pour les fichiers à inclure
  - dans `prog.c`, il suffit d'écrire : `#include <perso.h>` (le fichier `perso.h` se trouvant dans le répertoire `~/include`)
- La compilation conditionnelle (A.3)
  - il est possible de choisir des parties du programme source qui seront compilées ou pas dans le programme objet, selon les valeurs prises par des expressions constantes
  - la compilation conditionnelle emploie les directives `if else endif ifdef ifndef elif`

Le préprocesseur (Annexe A)

- La compilation conditionnelle (A.3 suite)
  - Exemples :

```
#ifndef TAILLE
#   define TAILLE 100
#endif
```

=> permet de tester l'existence d'une constante et de la définir au cas où elle n'existe pas

```
#if 0
#   zone non compilee
#endif
```

=> pour la mise au point de programme, il suffit de remplacer 0 par 1 pour compiler la zone de code

Le préprocesseur (Annexe A)

- La compilation conditionnelle (A.3 suite)
  - le problème des inclusions multiples : lorsqu'un fichier est inclus, faire en sorte qu'il le soit bien une seule fois

```
#include "f.h"
#include "g.h"
```

g.c

```
#include "f.h"
```

g.h

```
struct S {
  int i;
  int j;
};
```

f.h

Compilation de g.c => error : redefinition of 'struct S'

Solution :

utilisation d'une constante F_H qui sert de verrou booleen

=> **tous les fichiers .h doivent ainsi être construits !!**

```
#ifndef F_H
#define F_H
struct S {
  int i;
  int j;
};
#endif
```


Le préprocesseur (Annexe A)

- La compilation conditionnelle (A.3 suite)

```
# ifdef SOLARIS
      code pour solaris
# endif
# ifdef LINUX
      code pour linux
# endif
```

- permet de réaliser des programmes qui sont destinés à s'exécuter sur différentes plateformes ou systèmes d'exploitation
- à la compilation, utiliser `gcc -DSOLARIS prog.c`

Les dangers du langage C

- Les débordements
  - Pas de contrôle lors de l'écriture dans un tableau
    ⇒ possibilité de débordement
    ⇒ écrasement d'autres variables ou défaillance du programme
  - Pas de contrôle lors d'utilisation d'affectation basique (`tab[i]=10`) mais aussi dans de nombreuses fonctions fournies (notamment pour les chaînes de caractères)
  - Les fonctions de la famille `strcpy` permettant la copie de chaînes de caractères ne font aucune vérification sur la taille de la chaîne source
    ⇒ Très facile d'écraser une donnée en mémoire
  - Idem pour les fonctions `strcat, sprintf, gets ...`

Les dangers du langage C

- Les chaînes de caractères (pointeur et tableau)
  - Manipulation délicate entre `char *` et `char []`
  - `char * ch1="toto"` et `char ch2[]="toto"` ne désignent pas la même chose
  - `ch1` est une variable qui contient l'adresse d'une zone de mémoire **read only**
    ⇒ `strcpy(ch1,"titi")` provoque une erreur de segmentation mais `ch1="titi"` est possible
  - `ch2` est une constante qui contient l'adresse d'une zone de mémoire modifiable
    ⇒ `strcpy(ch2,"titi")` est correcte mais `ch2="titi"` ne compile pas

Les dangers du langage C

- L'arithmétique des pointeurs
  - Il est possible d'ajouter un entier à un pointeur (donc une adresse mémoire) mais il est important de bien réaliser le résultat de cette opération
  - Tout dépend de la taille du type sur lequel le pointeur pointe

    `type * p ; p = p + 4`
    provoque un déplacement en mémoire de `4 * sizeof(type)`

Les dangers du langage C

- Les chaînes de format
  - Les chaînes de format sont utilisées dans les fonctions `printf`, `scanf` et leurs cousines
  - Pas de vérification que le nombre de variables utilisées correspond au nombre de caractères de formats indiqués dans la chaîne de format
  - C'est le nombre de caractères de format qui l'emporte !

    `printf("%p %p\n", i)` affiche bien 2 valeurs !

Les dangers du langage C

- Le code non portable
  - Attention à la manipulation de code non portable

```
int i = 0x30313233;
char * ch;
ch=(char *)(&i);
printf("%c %c %c %c \n",
        ch[0],ch[1],ch[2],ch[3]);
bash-intel$ ./a.out
3 2 1 0       <- Sur Intel
bash-sparc$ ./a.out
0 1 2 3       <- Sur Sparc
```

⇒ En fonction du processeur sur lequel on exécute ce code, le résultat est différent

Les dangers du langage C

- Débordement d'entiers non signés (1/2)
  - Les opérations sont effectuées modulo 2^{32} sur une architecture 32 bits d'où troncature

```
unsigned int a;
a=0xe0000020;
a=a+0x20000020; // a tronque

unsigned int a=0x40000020;
a=a*4; // a tronque
```


Les dangers du langage C

- Débordement d'entiers non signés (2/2)
  - Attention aux conditions sur les entiers lors de tests

```
unsigned int n;
n=get_from_user();
if (n>0) {  <- si n tres grand
                  n*sizeof(char *) est tronque
  r=malloc(n*sizeof(char *));
  for (i=0;i<n;i++)
    r[i]= ...     <- overflow
}
```

⇒ Test sur `n` insuffisant car il peut être très grand et `n * sizeof(char *)` peut être tronqué

Les dangers du langage C

- Conversion de types
  - Attention aux entiers interprétés signés ou non

```
int length;
length=get_from_user();
if (length> 1024) exit(1);
if (read(sock,buffer,length) ...
```

⇒ Si `length` est négatif, il passe le test `if (length > 1024)` mais il devient un nombre très grand pour la fonction `read`

Les dangers du langage C

- Evaluation des opérandes
  - L'ordre d'évaluation des opérandes pour les opérateurs `&& || ?: ,` est garanti mais pas pour les autres opérations
  - `x=f()+g()`

    On ne sait pas si `f` est appelée avant ou après `g`.
  - `t[i]=f()`

    Si f modifie `i`, l'instruction a un comportement indeterminé

Les dangers du langage C

- Précédence des opérateurs
  - Une évaluation d'opérations se fait en fonction de la précédence des opérateurs et de leur associativité (gauche ou droite)
  - Le choix de la priorité des opérateurs est mauvais en langage C

```
if ( len & 0x80000000 != 0)
  printf("mauvaise longueur !");
```

⇒ L'absence de parenthèse et la priorité des opérateurs fait que l'opération effectuée est `if (len & (0x8000000 !=0))`

Les dangers du langage C

- Undefined behaviors et optimisations des compilateurs
  - Un certain nombre d'opérations sont décrites dans la norme comme étant indéterminées ⇒ chaque compilateur est libre de générer le code qu'il veut
  - En fonction des compilateurs et du niveau d'optimisation choisi au moment de la compilation, le résultat de la compilation (et donc de l'exécution) peut être très différent

Les dangers du langage C

- Undefined behaviors et optimisations des compilateurs
  - Division par zero

```
// code dans noyau Liux
if (!msize)
  msize=1/msize; // provoke a signal
```

  - Le code est volontairement destiné à déclencher un signal mais la division par 0 est un comportement indéfini
    ⇒ La division par 0 ne lève pas d'exception sur un PowerPC
    ⇒ Lors d'optimisations du compilateur, le test peut être purement supprimé

Les dangers du langage C

- Undefined behaviors et optimisations des compilateurs
  - Division par zero

```
// code dans PostgreSQL
if (arg2==0)
  ereport(ERROR, (errcode(
     ERRCODE_DIVISON_BY_ZERO),
                  errmsg("division by zero")));
PG_RETURN_INT32((int32) arg1/arg2;
```

  - `ereport` est une fonction qui ne retourne pas (déclenche une exception) mais le compilateur ne peut le savoir
    ⇒ Le compilateur considère que la division a lieu systématiquement
    ⇒ La division par 0 étant considérée comme impossible sur certaines architectures, gcc déplace la division avant le test !

Les dangers du langage C

- Undefined behaviors et optimisations des compilateurs
  - Overflow entiers signés (1/3)

```
#define MAX 500
int length;
length=net_from_user();
if (length<0 || length + 1 >= MAX)
  exit(1);
}
else {
 ...
}
```

  - Le débordement d'entiers signés est un undefined behavior en C mais très souvent les compilateurs le gèrent de façon identique (le complément à deux fait qu'on passe facilement d'un nombre positif à négatif et réciproquement)
  - `0x7FFFFFFF` passe les tests car `0x7FFFFFFF + 1` donne un nombre négatif

Les dangers du langage C

- Undefined behaviors et optimisations des compilateurs
  - Overflow entiers signés (2/3)

```
int main()
{
  int x=INT_MAX;

  if (x+100 > x)
    printf("oula\n");
}
```

  - En C, l'overflow d'entiers signés est un undefined behavior
    ⇒ le compilateur peut supposer que ca n'arrive donc jamais
    ⇒ le test peut disparaître du code lors d'optimisations du compilateur alors qu'il a été prévu volontairement pour le programmeur pour éviter les débordements

Les dangers du langage C

- Undefined behaviors et optimisations des compilateurs
  - Overflow entiers signés (3/3)

```
int do_fallocate(..., loff_t offset, loff_t len)
{ // code du noyau Linux
struct inode *inode = ...;
if (offset < 0 || len <= 0)
return -EINVAL;
/* Check for wrap through zero too */
if ((offset + len > inode->i_sb->s_maxbytes)
|| (offset + len < 0))
return -EFBIG;
...
}
```

  - Le programmeur ajoute un test pour prévoir le débordement de `offset + len`
    ⇒ gcc retire ce test considérant que cette somme ne peut être négative

Les dangers du langage C

- Undefined behaviors et optimisation des compilateurs
  - Décalages (1/2)

```
int main()
{
  unsigned int i=1 << 32;
  printf("%u\n",i);
}
bash$ clang decal.c ; ./a.out
4195632
bash$ gcc decal.c ; ./a.out
0
```

  - Un décalage de n bits ou plus sur un entier de n bits est un comportement indéterminé
    ⇒ En fonction de l'architecture, du compilateur et de l'optimisation utilisée, le résultat est différent

Les dangers du langage C

- Undefined behaviors et optimisation des compilateurs
  - Décalages (2/2)

```
// code du noyau Linux
groups_per_flex = 1 << sbi->
    s_log_groups_per_flex;
if (groups_per_flex == 0)
return 1;
flex_group_count = ... / groups_per_flex;
```

  - Extrait du noyau linux qui prévoit un test au cas ou le décalage donnerait une valeur nulle
    ⇒ `clang` assumant que ca ne peut jamais arriver, enlève le test

Les dangers du langage C

- Undefined behaviors et optimisation des compilateurs
  - Out-of-bound pointeurs

```
int vsnprintf(char * buf, size_t size, ...)
{ // code du noyau Linux
char * end;

if ((int )size<0) return 0;
end=buf+size;
if (end < buf)
   ...
}
```

  - Deux tests pour vérifier que `size` n'est pas trop grand et que la somme `buf + size` ne déborde pas
  - L'ajout d'un entier à un pointeur en C doit forcément donner l'adresse mémoire d'une variable du même type (tout autre manipulation est un undefined behavior) ⇒ le compilateur s'autorise donc à simplifier le test `end < buf` en `size < 0` et le supprime puisque redondant

Les dangers du langage C

- Undefined behaviors et optimisation des compilateurs
  - Déréférencement de pointeur NULL

```
unsigned int tun_chr_poll(struct file *file,
    poll_table * wait)
{ // code du noyau Linux
struct tun_file *tfile = file->private_data;
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
if (!tun)
return POLLERR;
...
}
```

  - Le déréférencement de pointeur NULL est un undefined behavior
    ⇒ Le compilateur peut assumer que le pointeur ne peut être NULL lors du déréférencement
    ⇒ Le test peut donc être supprimé par des compilateurs lors d'optimisations

Les dangers du langage C

- Autres problèmes liés aux optimisations des compilateurs
  - Terminaison de boucle (1/2)

```
int fermat (void) {
  const int MAX = 1000;
  int a=1,b=1,c=1;
  while (1) {
    if (((a*a*a) == ((b*b*b)+(c*c*c)))) return 1;
    a++;
    if (a>MAX) {
      a=1; b++;
    }
    if (b>MAX) {
      b=1; c++;
    }
    if (c>MAX) {
      c=1;
    }
  }
  return 0;
}

int main (void) {
  if (fermat()) printf ("Fermat's Last Theorem has been disproved.\n");
  else printf ("Fermat's Last Theorem has not been disproved.\n");
  return 0;
}
```

Les dangers du langage C

- Autres problèmes liés aux optimisations des compilateurs
  - Terminaison de boucle (2/2)
    ⇒ Cette boucle doit en principe être infinie, mais en fonction du niveau d'optimisation du compilateur, la boucle peut être simplement supprimée

  - ```
    bash:~/lang/C/undef$ clang fermat.c
    bash:~/lang/C/undef$ ./a.out
    ^C
    bash:~/lang/C/undef$ clang -O2 fermat.c
    bash:~/lang/C/undef$ ./a.out
    Fermat's Last Theorem has been disproved.
    ```


L'assembleur inline

- Il est possible de faire appel directement à du code assembleur dans du langage C
- L'appel à l'assembleur peut être utile lorsque le code équivalent en langage C n'est pas possible : récupération de contenu de registre par exemple
- La syntaxe :

```
asm("instructions assembleur"
      : operandes de sortie des instructions
      : operandes d'entree des instructions
      : liste des registres modifies
    );
```
```

L'assembleur inline

- Le bloc "instructions assembleur"
  - Une liste d'instructions assembleur
  - Les instructions assembleurs peuvent faire appel aux opérandes qui sont précisés dans les deux blocs suivants
  - La rérérence à ces opérandes est fait par la notation `%i` ou `i` est l'indice de l'opérande (indice commence à 0)
  - Si l'instruction assembleur utilise un registre, il est noté avec 2 caractères %.

```
asm("movq %%rax, %0" : "=r" (a) : : );
```

⇒ %0 fait référence au premier opérande `"=r" (a)`

L'assembleur inline

- Le bloc "opérandes de sortie"
  - Les opérandes sont écrits comme une expression C entre parenthèses précédée d'une chaîne de caractères indiquant une contrainte.
  - La chaîne est constituée d'un ou plusieurs modificateurs et lettres :
    - Le modificateur = indique que l'opérande est en écriture
    - Le modificateur + indique qu'il est lecture/écriture
    - La lettre `r` indique que l'opérande est stocké dans un registre ; on peut être plus précis pour x86 : `"a"`, `"b"`, `"c"`, `"d"` pour `eax`, `ebx`, `ecx`, `edx`, `"D"`, `"S"` pour `edi`, `esi` (les registres de r8 à r15 ne peuvent être spécifiés)
    - La lettre `m` spécifie un opérande mémoire
    - La lettre `i` spécifie un opérande immédiat

L'assembleur inline

- Le bloc "opérandes d'entrée"
  - Même format que opérande de sortie mais les modificateurs + ou = sont interdits.
  - Peut contenir une lettre faisant référence à l'indice d'un opérande de sortie pour expliciter que l'opérande d'entrée doit être stocké au même endroit
- Le bloc "liste des registres modifiés"
  - correspond à la liste des registres qui sont utilisés par les instructions assembleurs et qui ne doivent donc pas être utilisés pour stocker les opérandes

L'assembleur inline

- Quelques exemples :

```
int src=1;
int dst;
asm("mov %1, %0"
    : "=r" (dst)
    : "r" (src));
printf("%d\n",dst);
// copie de src dans dst
```

L'assembleur inline

- Quelques exemples :

```
int src;
asm("incl %0"
    : "=a" (src)
    : "0" (src)
    );
// le meme registre eax ("a") est utilise pour l'entree et
//la sortie
```


L'assembleur inline

- Quelques exemples :

```
int a,b,c;
asm("mov %1, %%eax\n\t"
    "mov %2, %%ecx\n\t"
    "add %%ecx, %%eax\n\t"
    "mov %%eax, %0"
     :"=r"(c)              /* ouput */
     :"r"(a), "r"(b)       /* input */
     :"%eax", "%ecx"       /* clobbered register */
   );
// somme a et b dans c
// eax et ecx etant utilises dans les instructions, ils ne
// peuvent etre utilises pour stocker les operandes
```