Module TC2
-
Master 1 EEA-AEETS

Nicolas Rivière

Université Toulouse III

Septembre 2011

Langage C : Rappels

- Généralités
- Variables
- Les tableaux
- Opérateurs
- Instructions
- Les fonctions
- Entrées/Sorties

Langage C : Préliminaires aux structures de données

- Les pointeurs
- Les structures
- Les unions

- Du programme à l'exécution.
- Structure d'un programme C.

Du programme à l'exécution

- Fichier source C (extension .c)
  ⇓ Préprocesseur
- Fichier C
  ⇓ Compilateur
- Ficher objet (extension .o)
  ⇓ Editeur de liens
- Programme exécutable

Structure d'un programme C (1)

```
/* 1 - directives du preprocesseur */

/* 2 - définition des constantes */

/* 3 - définition des types créés */

/* 4 - définition des variables globales */

/* 5 - déclaration des fonctions */

/* 6 - définition de la fonction principale */

/* 7 - définition des autres fonctions */
```

Structure d'un programme C (2)

```
/* Le cours commence ici */
/* ceci est un commentaire */

#include <stdio.h>                          /* directive du preprocesseur */

long int resultat[10];                      /* definition de variable */

long int puissance(long int X, int Y); /* declaration de fonction */
```

Structure d'un programme C (3)

```
int main()      /* fonction principale */
{
  int i, N=10; /* definition de variables locales */

  for (i=0; i<=N; i++)
    {
      resultat[i]=puissance(2,i);
      printf("2 puissance %d = %ld \n",i,resultat[i]);
    }
  return 0;
}
```

Structure d'un programme C (4)

```
long int puissance(long int X, int Y) /* definition de fonction */
{
  int i;                              /* definition de variables */
  long int P=1;                       /* locales à la fonction */

  for (i=0; i<Y; i++)
    {
      P=X*P;
    }
  return P;
}
```

- Identificateurs.
- Variables.
- Constantes.
- Type énuméré.

Identificateurs

- Nom donné aux composantes : variables, fonctions, ...
- Taille : 31 caractères maximum
- Caractères permis :
  1. a-z
  2. A-Z
  3. 0-9
  4. caractère soulignement (underscore) _ (à éviter en début de nom)
- Toujours commencer par une lettre $\Rightarrow$ Ne pas commencer par un chiffre !
- Le nom 2_resultat est interdit
- Le nom resultat_2 est bon

Types (1)

- Trois familles de variables simples :
  1. caractère : `char` (1 octet)
  2. entier : `short` (2 octets), `int` (2 ou 4 octets), `long int` (4 octets) et `long long int` (8 octets) dans le C99
  3. flottant (ou réel en virgule flottante) : `float` (4 octets), `double` (8 octets)
- Par défaut, ces types sont signés
- Possibilité d'utiliser les types NON signés

Types (2)

- Existence de quatre qualificatifs : `unsigned` pour les entiers et les caractères, `short` et `long` pour les entiers, `long` pour les réels de type `double`.
- Exemples de types en C :
  - `unsigned char` : entiers naturels sur 1 octet (0 à 255),
  - `short` ou `short int` : Entiers relatifs sur 2 octets (-32768 à 32767),
  - `unsigned long int` ou `unsigned long` : Entiers naturels sur 4 octets,
  - `long double` : Réels sur 10 octets ($3,4e^{-4932}$ à $3,4e^{4932}$)

Types (3)

- Différentes possibilités d'affectation pour le type caractère :
  1. alphabétique (`'8'`)
  2. hexadécimal (`0x38`)
  3. octal (`'\070'`)
  4. caractères spéciaux (`'\n'`)
- Absence de type booléen : la valeur 0 signifie Faux et toute autre valeur signifie Vrai
- Type `bool` introduit dans le C99 avec `true` (1) et `false` (0)

Les constantes

- Il existe 4 types de valeurs constantes :
  1. entière en décimal ou hexadécimal (`12,0xC`)
  2. réelle en base 10 : `[.]Chiffres[.][Chiffres][Exposant]`
  3. caractère (`'a'`, `'T'`)
  4. chaîne de caractères (`"une chaine"`)
- Les expressions constantes :
  `#define identificateur_expression valeur_expression`
- Les constantes énumérées ou type énuméré (valeurs entières) :
  `enum identificateur_type_énuméré ...`
  `...{identificateurs_des_valeurs};`
- Pour les constantes énumérées, affectation automatique ou manuelle des valeurs possible, démarre à 0 sauf si explicité

Les déclarations

- Déclaration d'une variable : `type identificateur;`
- Initialisation à la déclaration : `type identificateur = valeur;`
- Une déclaration réserve un espace mémoire égal au type de la variable déclarée.
- Utilisation du mot réservé `const` pour indiquer que la variable n'est pas modifiable : `const type identificateur = valeur;`

- Définition.
- De caractères.
- Multidimensionnels.

Définition

- Collection de variables de même type rangées consécutivement en mémoire, sans trou.
- Premier indice du tableau est toujours 0.
- Déclaration comme une variable mais taille en plus : `type_elements_contenus identificateur[nombre_elements];`
- Initialisation possible lors de la déclaration avec des accolades.
- Chaque élément est indicé par une valeur entière.
- Remarque : pas de détection de débordement de tableau par le compilateur !

Chaîne de caractères

- Une *chaîne de caractères* n'existe pas en C : utilisation d'un tableau de caractères
- Dernier élément est le caractère *null* (`'\0'`)
- Initialisation caractère par caractère ou globalement
- `char chaine[10] = "UPS";` n'est possible qu'à la définition
- `char chaine[10] = {'U','P','S'};` idem ci-dessus
- Il y aura ici 4 caractères dans la chaîne.
- `char chaine[10]; chaine = "UPS";` est impossible ! !

Tableaux multi-dimensionnels

- Tableaux à plusieurs dimensions ≡ Tableaux de tableaux de ...
- Tableau à 2 dimensions (matrice) :
  `type identificateur[nb_ligne][nb_colonne] ;`
- Accession aux éléments en donnant un indice par dimension.
- `int matrice[2][3]={{1,2,3},{4,5,6}}` avec 1 dans la case (0,0)

Opérateurs

Deux types d'opérateurs :

- Opérateur unaire : s'applique à une seule opérande
- Opérateur binaire : s'applique à deux opérandes
- On appelle *expression* une combinaison d'opérateurs et d'opérandes qui conduit à un résultat unique.
- Type du résultat dépend des opérandes.
- Ordre de conversion : char $\longrightarrow$ int $\longrightarrow$ float $\longrightarrow$ double
- Conversion *explicite* (casting) et *implicite*

Opérateurs arithmétiques

- 5 opérateurs : `+`, `-`, `*`, `/`, `%`
- Opération effectuée avec deux opérandes du même type (casting si nécessaire)
- `(float) 2/3` vaut 0.677777
- `(float) (2/3)` vaut 0
- Conversion possible au moment de l'affectation : `int toto = 4.8;` donne un entier (4)

Opérateurs relationnels

- Il y a six opérateurs relationnels
- Ils renvoient une valeur entière (type int) égale à : 0 $\rightarrow$ faux, 1 $\rightarrow$ vrai
- Egalité : ==
- Différence : !=
- Comparaison : > , >= , <= , <

Opérateurs logiques

- Il y a trois opérateurs logiques
- Ils s'appliquent aux opérandes qui ont la valeur logique Faux (0) ou Vrai
- Le résultat renvoyé est 0 ou 1
- ET logique : &&
- OU logique : ||
- Négation logique : !

Opérateurs de manipulation de bits

- Il y a six opérateurs manipulant les bits
- ET bit à bit : `&` ; utile pour les masques
- OU inclusif : `|`
- OU exclusif : `^`
- Complément à 1 : `~`
- Décalage à gauche : << avec introduction de zéros par la droite
- Décalage à droite : >> avec extension du bit de signe si signé sinon 0

Affectation

- Réalisation : `variable = expression`
- Possibilité d'affectation entre des types différents mais dangereux !
- Variantes :
  - `val = val <opérateur> opérande`
  - `val <opérateur>= opérande` (affectation combinée)

Opérateurs d'incrémentation et décrémentation

- La position de l'opérateur (avant ou après) indique si l'utilisation de la valeur se fait avant ou après l'opération
- `val++` équivaut à `val = val + 1` (ou `++val`)
- `val--` équivaut à `val = val -1` (ou `--val`)
- Utilisation exclusive avec des variables considérées isolément
- Post-<opération> : utilisation de la variable avant incrémentation
- Pré-<opération> : utilisation de la variable après incrémentation

Opérateurs d'incrémentation et décrémentation (exemples)

```
Val = 5;

/* Incrémentation de Val avant la multiplication par 3 */
Page = ++Val * 3;   /* Val = 6 et Page=18 */

/* Incrémentation de Val après la multiplication par 3 */
Delta = (Val++) * 3;   /* Val=7 et Delta=18 */
```

- De contrôle
- Itératives

Instructions de contrôle : if-else

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

- Si `condition` est évaluée à vrai, alors `corps du alors` exécuté, sinon `corps du sinon` est exécuté
- Plusieurs instructions de type if-else imbriquées possibles, attention aux accolades
- Pas obligation de `corps du sinon`
- `corps` peut être une seule instruction $\Rightarrow$ pas d'accolade

Instructions de contrôle : opérateur conditionnel ?

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

- Si `condition` est évaluée à vrai, alors la valeur de l'opération sera `expression-si-vrai`, sinon elle sera `expression-si-faux`.

```
Max = (4 > 8) ? 4 : 8;
Min = (4 < 8) ? 4 : 8;
```

Instructions de contrôle : switch

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

- Une expression est de type entier (ou caractère).
- Une étiquette est de type entier (ou caractère) qui peut être une constante explicite ou construite à partir d'opérateurs (logiques, logiques bit à bit, arithmétiques, relationnels, conditionnel)

Instructions de contrôle : switch (suite)

- L'expression est évaluée et comparée aux étiquettes
- S'il y a égalité avec une étiquette `i`, alors les instructions `instructions_i ... instructions_n` et `instructions_d` sont exécutées.
- Si pas d'égalité d'étiquette alors on exécute `instructions_d`.
- Possibilité de grouper plusieurs entrées :

```
case <etiquette_1> :
case <etiquette_2> : <instructions_12>;
```

Instructions de déroutement : break

- Permet de sortir d'une boucle avant terme
- Très utilisé avec `switch`
- Exemples :

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

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

Instructions de déroutement : continue

- Permet de passer au début de l'itération suivante dans une boucle.
- Jamais dans un `switch` !
- La condition de boucle sera immédiatement testée après `continue` .
- Exemples :

```
for (i=0; i<5; i++)
{
  if (i==3) continue;
  printf("%d\n",i);
}
```

Instructions itératives : for

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

- `pre-actions` : effectuées une seule fois au début de la boucle. En général, initialisation de variable.
- `conditions` : conditions de poursuite de boucle.
- `post-actions` : effectuées à la fin de chaque itération.
- `corp de boucle` : une seule instruction ou un ensemble d'instuctions `{...}`.
- Les pré-actions, les conditions et les post-actions sont facultatives.
- Si plusieurs pré-actions, conditions ou post-actions : séparation par une virgule.

Instructions itératives : for (suite)

- Les pré-actions peuvent être :
  - une valeure numérique
  - une expression
  - une valeur retournée par une fonction
- Si pas de condition, boucle infinie (test=1).
- Si plusieurs conditions $\Rightarrow$ "ET logique" des conditions.
- Imbrication de boucles `for` possible

```
for ( ; ; )
  for ( ; ; )
    for ( ; ; )
    {
      ...
}
```

```
for ( ; ; );
  for ( ; ; );
    for ( ; ; )
    {
      ...
}
```

```
for ( ; ; )
  for ( ; ; )
    for ( ; ; );
    {
      ...
}
```

Instructions itératives : while

```
while (<conditions>)
  <corps de boucle>
```

- `corps de boucle` : une seule instruction ou un ensemble d'instuctions `{...}`.
- Si condition vraie ($\neq 0$) alors on exécute le corps de boucle.
- Le corps de la boucle peut ne jamais être exécuté !
- La condition peut être une expression d'affectation : `while (reste=dividende%diviseur)`
- Imbrication de boucles `while` possible

Instructions itératives : do-while

```
do
  <corps de boucle>
while (<conditions>);
```

- `corps de boucle` : une seule instruction ou un ensemble d'instuctions `{...}`.
- Le corps de la boucle est exécuté au moins une fois.
- `while (condition) ;` peut être la fin d'un do-while ou un while sans corps $\Rightarrow$ Préférer un `while (condition) {};`

- Définition.
- Paramètres.

Présentation

- Une fonction permet de "modulariser" un programme en le découpant en actions simples :
  $\Rightarrow$ programmation structurée
  $\Rightarrow$ compilation séparée, création de bibliothèques
- Traitement de données :
  - locales,
  - globales,
  - passées par des paramètres formels.
- En C, que des fonctions !
- La fonction `main` est la fonction principale

Définition

```
<type du resultat> nom_fonction (<declaration arguments>)
{
  <corps de la fonction>
}
```

- La première ligne terminée par un " ;" est une *déclaration*.
- Résultat peut être un type (int, char, ...) ou void ("vide")
- Exemples :

```
double Pi()
{
   ...
}
```

```
int calcul(int valeur)
{
   ...
}
```

```
void Rien()
{}
```

Paramètres

- Les paramètres ou arguments sont déclarés à la suite, séparés par une virgule, avec le type de chacun précédant le nom (pas obligatoire lors de la déclaration).
- Le stockage des arguments se fait dans la *pile*.
- L'instruction *return* permet de quitter une fonction et de renvoyer un résultat si souhaité :
  `return;` $\Rightarrow$ pour une fonction de type void (pas indispensable)
  `return <expression>;` $\Rightarrow$ dans les autres cas
- Pas d'imbrications de fonction possible $\Rightarrow$ toutes sœurs.
- Pas de déclaration de fonction dans une autre fonction.

Passage de paramètres et appels de fonction

- Deux types de passage de paramètres :
  - par *valeur* : à l'appel de la fonction, une copie de la valeur de l'argument est passée à la fonction ⇒ pas de modification de l'argument initial
  - par *adresse* : la fonction utilise l'argument car c'est l'adresse qui est passée à la fonction appelée ⇒ utilisation de *pointeurs*.
- Les tableaux sont toujours passés par adresse.
- Choix libre du passage pour les autres variables.
- L'appel de fonction peut constituer une instruction simple ou être utilisée dans une expression.
- Parenthèses obligatoires lors de l'appel même si pas d'argument.
- Une fonction appelée doit être déclarée ou définie avant son appel.

Exemples

```
int f1(char b)
{
  ...
}

int main(void)
{
  int i;
  i = f1('c');
}
```

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

int f1(char b)
{
  ...
}

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

- Accès par caractère.
- Formatées.
- Accès par ligne.
- Fichier.

$\Rightarrow$ Pour avoir accès aux entrées/sorties, il faut utiliser la bibliothèque *stdio.h* à l'aide de la directive : `#include <stdio.h>`

Accès par caractère ou standard

- Trois flux standards en C : *stdin* (entrée), *stdout* (sortie) et *stderr* (sortie erreur).
- `int getchar(void)` : lecture d'un seul caractère sur l'entrée standard.
- `int getc(<flux d'entrée>)` : idem getchar() dans le flux spécifié.
- La saisie est validée par validation avec la touche "entrée".
- `int putchar(int)` : écriture d'un seul caractère sur la sortie standard.
- `int putc(int, <flux de sortie>)` : idem putchar() dans le flux spécifié.

Formatées ou mises en formes (1)

```
int printf(char *format, <arguments>)
int * sprintf(char *chaine, char *format, <arguments>)
```

- Met en forme des arguments (optionnels) et les envoie vers la sortie standard (printf) ou vers une chaîne de caractère (sprintf).
- La chaîne de caractères "char *format" contrôle l'affichage
- Conversion spécifiée par % suivi d'un caractère : c (caractère), d ou i (int décimal), s (chaîne de caractères), f (double), x (int hexadécimal).

Formatées ou mises en formes (2)

```
int scanf(char *format, <arguments>)
int * sscanf(char *chaine, char *format, <arguments>)
```

- Lit des données dont le nombre et le type sont spécifiés dans la chaîne *format*.
- Pour sscanf(), les données sont lues à partir d'une chaîne.
- Stockage des données lues dans les arguments.
- Conversion spécifiée par % suivi d'un caractère : c (caractère), d ou i (int décimal), s (chaîne de caractères), f (double), x (int hexadécimal).
- La conversion s'arrête dès qu'il y a un séparateur implicite (espace, tabulation, interligne) ou explicite (précisé dans le format), un caractère incompatible avec un type du format.

Les arguments sont des paramètres de sortie $\Rightarrow$ Passage par adresse !

Accès par ligne

- `char * gets(char * Ligne)` : lit une chaîne de caractères sur l'entrée standard et la range en mémoire dans le tableau de caractères à partir de l'adresse Ligne en ajoutant `\0` et supprimant `\n`.
- `int puts(char * Ligne)` : affiche la chaîne de caractères Ligne sur la sortie standard (retour chariot ajouté automatiquement ici).

Fichier (1)

- Pour traiter les données contenues dans un fichier, fonctions de manipulation.
- `FILE * fopen(char * nomfic, char * mode)` : ouverture d'un fichier
- Renvoi l'adresse d'une structure de type FILE qui sera accessible suivant le mode spécifié en paramètre ("r", "w")
- Association du fichier à un tampon (bloc mémoire) par un flux
- `int fclose(FILE * pfic)` : fermeture d'un fichier
- Transfert du contenu du tampon sur le disque

Fichier (2)

- Les entrées-sorties par caractère sont effectuées avec les mêmes primitives que précédemment avec comme flux d'entrée `FILE *` :
  `int getc(FILE *)`
  `int putc(char, FILE *)`
- Les entrées-sorties formatées sont effectuées avec les primitives `fprintf` et `fscanf` qui sont dérivées respectivement de sprintf et sscanf :
  `int fprintf(FILE *pfic, char * format, <arguments>)`
  `int fscanf(FILE * pfic, char * format, <arguments>)`
- Les entrées-sorties par ligne sont effectuées comme `gets` et `puts` avec les fonctions `fgets` et `fputs` :
  `char * fgets(char * Ligne, int Max, FILE * pfic)`
  `int fputs(char * Ligne, FILE * pfic)`

- Principes.
- Pointeurs et tableaux.
- Tableaux de pointeurs.
- Pointeurs et fonctions.
- Indirections multiples.
- Allocation dynamique.

Principes

- Un pointeur est une variable qui contient l'adresse d'une autre variable.
- Déclaration d'un pointeur : `<type> * pointeur`
  Ex : `int * pointeur` est un pointeur sur entier.
- L'adresse d'une variable est obtenue à l'aide de l'opérateur unaire &.
- L'objet sur lequel pointe un pointeur est obtenu avec l'opérateur unaire * : `* pointeur;`

```
int * p_entier;
int pointee;
int largeur;
p_entier=&pointee;
largeur=*p_entier;
```

Pointeurs et tableaux

```
int tableau[7];
int * p_jour;
```

- Le nom tableau est l'adresse du 1er élément du tableau (`&tableau[0]`)
- L'opération suivante est correcte : `p_jour = tableau;`
- Le pointeur peut être utilisé pour se déplacer dans le tableau : `*(p_jour+4)` est le 4è élément du tableau soit `tableau[4]`
- Que fait l'opération suivante ? : `p_jour=tableau+4`
- Une chaîne de caractères est un tableau (de caractères) donc son nom est l'adresse du premier caractère.

Tableaux de pointeurs

```
type_pointe * Nom_tableau[taille_tableau];
```

- Un tableau de pointeur contient des pointeurs vers des variables du type spécifié.
- Nom_tableau est un tableau de taille_tableau pointeurs sur type_pointe.
- Nom_tableau est l'adresse du premier élément du tableau.
- Nom_tableau est l'adresse d'un pointeur.
- Un tableau de chaînes de caractères est un tableau de longueur automatiquement ajustée.

```
char * journee[] = {"matin","midi","apres-midi","soir"};
```

Pointeurs et fonctions

- Passage des arguments de fonctions *par adresse*
- Pour les tableaux, équivalence entre [] et * en tant que paramètres formels :

```
void fonction1(int vecteur[])
void fonction2(int * vecteur)
```

- Pour les scalaires :

```
void fonction3(double * valeur) {
  *valeur +=2.0;
}
int main() {
  double reel=0.0;
  fonction3(&reel);
  return 0;
}
```

Indirections multiples

- On parle d'indirections multiple lorsqu'on a des pointeurs de pointeurs de ...
- Indirection simple $\equiv$ pointeur
- Indirection double $\equiv$ pointeur de pointeur
- Exemple :

```
int un_entier;
int * p_sur_entier;
int ** p_sur_p_sur_entier;
```

- `p_sur_p_sur_entier` est un pointeur sur un pointeur
- `p_sur_p_sur_entier = &un_entier` $\Rightarrow$ c'est faux !

Allocation mémoire

- La déclaration d'un pointeur provoque la réservation mémoire pour le pointeur mais pas pour la variable sur laquelle il pointe !
- L'allocation mémoire pour la variable s'effectue au moment de l'affectation sauf sauf pour les tableaux !
- Possibilité d'allocation dynamique avec les fonctions `malloc` et `calloc`
- `void * malloc(int taille)` : provoque la réservation mémoire de *taille* octets contigus.
- `void * calloc(int nb, int taille)` : provoque la réservation d'un tableau de *nb* éléments de *taille* octets.
- Libération de la mémoire avec la fonction `free`.

Pointeurs génériques

- Le langage C ne permet pas d'affecter des pointeurs de types différents
- Possibilité de créer des fonctions indépendamment du types des paramètres
  $\Rightarrow$ pointeurs génériques de type `void *`
- Problème car on ne peut accéder au contenu avec l'opérateur *
- Utilisation de la fonction `memcpy()` pour copier le contenu d'un variable pointée vers une autre :
  `void *memcpy(void * pa, void * pb, unsigned int taille)`
- On copie "taille" octets de adresse "pb" vers adresse "pa"
- Pour obtenir la taille : fonction `sizeof()`

Pointeurs génériques : exemple

Une fonction qui permute n'importe quel type de valeurs

```
void permuter(void* pa, void* pb, unsigned int taille)
{
void * pc;

pc = malloc(taille);
memcpy(pc,pa, taille);
memcpy(pa,pb,taille);
memcpy(pb,pc,taille);
free(pc);
}
```

Pointeurs génériques : exemple

Elle pourra être utiliser ainsi :

```
int ia=4,ib=5;
float fa=2.1,fb=1.2;

permuter(&ia,&ib,sizeof(ia));
permuter(&fa,&fb,sizeof(fa));
```

- Principes.
- Utilisation.
- Pointeurs et structure.

Principes

- Structure de données avec plusieurs champs de types différents
- Déclaration d'une structure :

```
struct <etiquette>
{
  /* corps de la structure */
};
```

- Déclaration d'une variable dont le type est une structure :

```
struct <etiquette> a;
```

- Une structure est composée de champs :

```
struct Personne
{ char nom[15];
  char prenom[15];
  int age;
};
```

Manipulations

- Accession à un champ d'une structure par le . (point) :
```
struct Personne etudiant1;
etudiant1.age = 19;
```
- Pour affecter une chaîne de caractères, utilisation de `strcpy` :
```
strcpy(etudiant1.nom,"Ekins");
strcpy(etudiant1.prenom,"Bud");
```
- Possibilité d'initialiser lors de la définition :
```
struct Personne etudiant1 = {"Ekins","Bud",19};
```
- Affectation directe de deux structures de même type :
```
struct Personne etudiant1,etudiant2;
etudiant2=etudiant1;
```

Imbrication

- Les structures peuvent être imbriquées.

```
struct Identification
{
  int numero;
  struct Personne Etudiants;
}
```

- Accession aux champs à l'aide du point ou globalement :

```
struct Identification un_etudiant;
un_etudiant.numero = 24056012;
un_etudiant.Etudiants = etudiant1;
strcpy(un_etudiant.Etudiants.nom, "Durand");
un_etudiant.Etudiants.age = 20;
```

Structures et pointeurs

- Lors de l'appel de fonction avec une structure en paramètre, il est préférable de passer l'adresse plutôt que les valeurs pour un souci d'économie de mémoire.
- Déclaration : `struct Identification * p_structure;`
- Accès aux champs par l'opérateur ->

```
p_structure = &un_etudiant;
p_structure->numero=150;
(*p_structure).numero=150;
strcpy(p_structure->Etudiants.nom,"Dupont");
```

Tableaux de structure

- Tableaux dont les éléments sont des structures identiques.
- `struct Identification classe[21];`
- `classe` est l'adresse de la première structure du tableau.
- `classe[1].numero = 121;`
- `strcpy(classe[1].Etudiants.nom, "toto");`

Principes

- Structure de données avec plusieurs champs de types différents
- Un seul espace mémoire pour l'union
- Un seul membre d'une variable de type `union` contient une valeur à un instant donné
- Le programmeur doit savoir dans quel membre a été rangé une valeur :
  $\Rightarrow$ Association d'une variable à l'union
- Utilisation conjointe avec une structure afin de pouvoir indiquer le champ de l'union utilisé

Manipulations

- Déclaration d'une union :

```
union <etiquette>
{
  /* corps de l'union */
  /* composé de champs */
  /* comme une structure */
};
```

- Déclaration d'une variable dont le type est une union :

```
union <etiquette> u;
```

- Affectation par le biais du champ :

```
u.champ=valeur;
```

Manipulations avec une structure

- Déclaration d'un type pour indiquer le champ manipulé :
  `enum quel_champ {champ_1,champ_2,...,champ_n};`
- Déclaration de la structure contenant une union :

```
struct <etiquette_struct>
{
  enum quel_champ le_champ;
  union <etiquette> u;
};
```

- Déclaration d'une variable dont le type est une structure :
  `struct <etiquette_struct> <identificateur_var>;`
- Affectation par le biais des champs de la structure :

```
<identificateur_var>.u.champ=valeur;
<identificateur_var>.le_champ=champ_x;
```

Exemple

```
union nombre
{ int i;
  float f;
};
enum type {Entier,Flottant};
struct calcul
{
  enum type type_val;
  union nombre u;
};
struct calcul n;

n.typ_val=Entier;
n.u.i=10;
```