Langage C

Samba Ndojh NDIAYE
IUT Lyon 1
Laboratoire d'InfoRmatique en Image et Systèmes d'information
LIRIS UMR 5205 CNRS/Université Claude Bernard Lyon 1
samba-ndojh.ndiaye@univ-lyon1.fr

Évaluation

TP

- Bonus / Malus

Contrôle Continu

- écrit : 3 évaluations

DS promo

- écrit : 1 évaluation

Note finale

- $1/3 * CC + 2/3 * DS + Bonus/Malus$

Supports

- `liris.cnrs.fr/~snndiaye/fichiers/Cours_C/`

Sommaire

Algorithmique et programmation

- Programmation informatique : ensemble des activités permettant l'écriture des programmes informatiques
- Résolution de problèmes : proposer des solutions (algorithmes)
- Un algorithme : une séquence d'actions "simples" qui décrit une démarche pour résoudre un problème donné
- L'écriture de cette démarche dans un langage de programmation est la base de la programmation informatique

Algorithmique et programmation

- Milliers de langages de programmation informatique : Pascal, Java, Fortran, C, C++...
- Une syntaxe stricte : sans ambiguïtés
- Le langage machine : suite de 0 et 1 associée aux instructions élémentaires exécutables par le microprocesseur
- Langages plus évolués : compréhension élevée grâce à la maîtrise de leur syntaxe
- Une traduction en langage machine est nécessaire

Langage C

- Langage de programmation impérative : les instructions sont exécutées pour transformer l'état actuel du programme
- Les opérations possibles sont limitées :
  - les assignations,
  - les branchements conditionnels,
  - les branchements inconditionnels,
  - les boucles.

Algorithmique et programmation

```
1 #include<stdio.h>
2 int main( )
3   {
4       printf("Bonjour !");
5       return 0;
6   }
```

Structure d'un programme

Algorithme 1 : Structure d'un programme C

```
1 #directives adressées au préprocesseur
2 int main( )
3   {
4       déclarations (les variables, les objets qui seront manipulés)
5       instructions (les opérations à exécuter)
6   }
```

Structure

- Un ou plusieurs fichiers sources (avec l'extension ".c" ou ".h")
- Chaque fichier contient des déclarations et/ou des définitions de fonctions et/ou de variables
- Le programme doit contenir une et une seule fonction main
- Espaces, tabulations, fins de lignes sont ignorés
- Les commentaires sont également ignorés :
  /* Ceci est un commentaire */
  // Tout ce qui suit, jusqu'à la fin de la ligne, est un commentaire

Sommaire

Sommaire

La mémoire centrale

Mémoire

- L'information est codée en binaire (suite de 0 et 1)
- L'unité de codage est le bit
- le plus petit espace adressable est l'octet (8 bits)
- Organisation et manipulation de l'information par paquets d'octets

Information en mémoire

- Une information en mémoire centrale est "repérée" par :
  - son adresse de stockage ET son type
  - identificateur (lien symbolique)

Types d'information

- Il est possible de manipuler différents types de données : entiers, flottants, caractères, chaînes de caractères...

Les variables

Une variable

Un emplacement mémoire dans lequel est codé une information modifiable et utilisable grâce à un identificateur.

Déclaration

Spécification du nom et du type de la variable

- Syntaxe : < type de base > < identificateur <= initialisation -opt > >,... ;

Identificateur

Une suite contigüe composée de lettres et de chiffres et du caractère " - " et commençant obligatoirement par une lettre.

Initialisation

Attribution d'une valeur à la variable dès sa création.

- Les mots réserves du langage ne peuvent être des identificateurs.
- Sensibilité à la casse : pas d'équivalence entre les majuscules et les minuscules

Sommaire

Affichage à l'écran

printf

Fonction de la bibliothèque stdio (Standard Input / Output)

- Syntaxe 1 : printf("texte ") ;
- Syntaxe 2 : printf("texte1 %x1 texte2 %x2... " , <expression1>, <expression2>,...) ;

Spécificateurs : %x

- int (décimale) : %d
- char (caractères) : %c
- float (décimale) : %f
- double (décimale) : %lf
- chaîne de caractères : %s

Saisie au clavier

scanf

Fonction de la bibliothèque stdio (Standard Input / Output)

- Syntaxe : scanf("%x1 %x2...", &<variable1>, &<variable2>,...) ;

Spécificateurs : %x

- int (décimale) : %d
- char (caractères) : %c
- float (décimale) : %f
- double (décimale) : %lf
- chaîne de caractères : %s

Les Entrées-Sorties

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i;
5        char j;
6        double k;
7        scanf("%c", &j);
8        scanf("%d%lf", &i, &k);
9        printf("i=%d, j=%c et k=%lf \n",i, j, k);
10       return 0;
11   }
```

Sommaire

Les entiers

Très petite taille : (unsigned) char

- entiers représentés sur au moins 1 octet (8 bits)
  - de -128 à 127
  - de 0 à 255 (pour les unsigned)

Petite taille : (unsigned) short

- entiers représentés sur au moins 2 octets (16 bits)
  - de -32.768 à 32.767
  - de 0 à 65.535 (pour les unsigned)

Petite taille : (unsigned) int

- entiers représentés sur au moins 2 octets (16 bits)
  - de -32.768 à 32.767
  - de 0 à 65.535 (pour les unsigned)

Les entiers

Grande taille : (unsigned) long

- entiers représentés sur au moins 4 octets (32 bits)
  - de -2.147.483.648 à 2.147.483.647
  - de 0 à 4.294.967.296 (pour les unsigned)

Très grande taille (Iso C99) : (unsigned) long long

- Les entiers représentés sur au moins 8 octets (64 bits)
  - de -9.223.372.036.854.775 808 à 9.223.372.036.854.775.807
  - de 0 à 18.446.744.073.709.551.615 (pour les unsigned)

Le type int

- le plus adapté parmi sur la machine utilisée

Les entiers

```
1 #include<stdio.h>
2 int main( )
3   {
4       int somme = 0;
5       printf("somme=%d, taille somme=%d, taille int=%d \n",somme, sizeof(somme),
  sizeof(int));
6       printf("taille char=%d, taille short=%d, taille long=%d, taille long long= %d \n",
  sizeof(char), sizeof(short), sizeof(long),sizeof(long long));
7       return 0;
8   }
```

Les flottants

Nombre flottant

Un nombre flottant (123.456E-78) est composé de :

- une partie entière (123)
- un point qui fait office de virgule (.)
- une partie fractionnaire (456)
- une des deux lettres E ou e qui précède la valeur de l'exposant (E)
- un exposant (-78)

On peut omettre :

- la partie entière ou la partie fractionnaire, mais pas les deux
- le point ou l'exposant, mais pas les deux

Les flottants

Simple précision : float

- flottants représentés sur au moins 4 octets (32 bits)
  - de -1.70E38 à -0.29E-38 et de 0.29E-38 à 1.70E38

Grande précision : double

- flottants représentés sur au moins 8 octets (64 bits)
  - de -0.90E308 à -0.56E-308 et de 0.56E-308 à 0.90E308

Très grande précision : long double

- flottants représentés sur au moins 8 octets (64 bits)
  - de -0.90E308 à -0.56E-308 et de 0.56E-308 à 0.90E308

Les flottants

```
1 #include<stdio.h>
2 int main( )
3   {
4       float produit = 1.0;
5       printf("produit=%f, taille produit=%d, taille float=%d \n",produit, sizeof(produit),
  sizeof(float));
6       printf("taille double=%d, taille long double=%d \n", sizeof(double), sizeof(long
  double));
7       return 0;
8   }
```

Les caractères

Type char

- Pas de type dédié pour la représentation des caractères
- Codage des caractères par des entiers : code ASCII
- Utilisation du type **char** pour représenter tous les caractères
- Les constantes caractères sont données entre apostrophes : 'A'
- Possibilité d'utiliser directement le code ASCII
  - 'A' : 65 (décimal) 101 (octal) 41 (hexadécimal) 01000001 (binaire)
  - Equivalence entre 'A' et '\101'

Les caractères

```
1 #include<stdio.h>
2 int main( )
3   {
4       char var = 'A';
5       printf("var=%d, var=%c, taille var=%d, taille char=%d \n",var, var, sizeof(var),
  sizeof(char));
6       return 0;
7   }
```

Les caractères : scanf

```
1  #include<stdio.h>
2  int main( )
3    {
4        char var;
5        printf("Saisir un caractere : ");
6        scanf("%c", &var);
7        printf("var=%d et var=%c \n",var, var);
8        printf("Saisir un autre caractere : ");
9        scanf("%c", &var);
10       printf("var=%d et var=%c \n",var, var);
11       return 0;
12   }
```

Les entiers et les flottants : scanf

```
1  #include<stdio.h>
2  int main( )
3    {
4        int somme;
5        int produit;
6        printf("Saisir un entier : ");
7        scanf("%d", &somme);
8        printf("somme=%d \n", somme);
9        printf("Saisir un flottant : ");
10       scanf("%f", &produit);
11       printf("produit=%f \n", produit);
12       return 0;
13   }
```

Sommaire

Expression

Définition

Une **expression** est une entité syntaxiquement correcte et qui a une valeur dont le type est l'un des types de base.

Exemples

- 123 + 456.78
- 'A'
- -35
- somme
- somme * 35

Affectation

Opération

Une affectation permet de modifier la valeur d'une variable

Syntaxe

<variable> = <expression>

- La valeur de l'expression est convertie au type de la variable si cela a un sens.

Affectation

```
1  #include<stdio.h>
2  int main( )
3    {
4        int somme = 0;
5        printf("somme=%d \n", somme);
6        somme = somme + 10 * (1 + 2 + 3 + 4 + 5);
7        printf("somme=%d \n", somme);
8        somme = 123.45;
9        printf("somme=%d \n", somme);
10       return 0;
11   }
```

Opérateurs arithmétiques

Opérations

- Addition : +
- Soustraction : -
- Multiplication : *
- Division : /
- Modulo (reste de la division entière) : %

Syntaxe

<expression1> OP <expression2>

Types du résultat

Le type le plus "général" parmi ceux des deux expressions opérandes

- **int** OP **int** —> **int**
- **char** OP **long** —> **long**
- **entier** OP **flottant** —> **flottant**

Opérateurs arithmétiques

```
1 #include<stdio.h>
2 int main( )
3   {
4       printf("%d et %d et %d \n", 3/2, 3/2.0, 3.0/2.0);
5       printf("%lf et %lf et %lf \n", 3/2, 3/2.0, 3.0/2.0);
6       printf("%d et %lf et %lf \n", 3/2, 3/2.0, 3.0/2.0);
7       return 0;
8   }
```

Affectations généralisées

Opérations

Combinaisons affectation et opérateurs arithmétiques : +=, -=, *=, /= et %=

Syntaxe

<variable> OP= <expression>

- Équivalent à : <variable> = <variable> OP <expression>

Incrémentation/Décrémentation

Post-incrémentation : ++

- Syntaxe : <variable>++
- Pré-requis : <variable> doit être de type entier, flottant, pointeur
- Opération : idem <variable> = <variable> + 1

Pré-incrémentation : ++

- Syntaxe : ++<variable>
- Pré-requis : <variable> doit être de type entier, flottant, pointeur
- Opération : idem <variable> = <variable> + 1

Décrémentation : - -

- Idem Incrémentation

Incrémentation/Décrémentation

Quelle différence

- <variable>++ a la même valeur que <variable> avant l'évaluation de l'expression.
- ++<variable> a la même valeur que <variable> après l'évaluation de l'expression.

Incrémentation/Décrémentation

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i, j, k;
5        i = 1;
6        j = i++ ; /* équivaut à j = i ; i = i + 1 ; */
7        i = 1;
8        k = ++i ; /* équivaut à i = i + 1 ; k = i ;*/
9        return 0;
10   }
```

sizeof

Opération

sizeof renvoie un entier indiquant la taille en octets d'une variable ou d'un type de données

Syntaxe

- sizeof(<variable>)
- sizeof(<type>)

Priorité des opérateurs

15	()	[]	.	->			
14	!	++	–	-(unaire)	*(unaire)	&	sizeof()
13	*	/	%				
12	+	-					
10	<	<=	>	>=			
9	==	!=					
5	&&						
4	||						
2	=	*=	/=	%=	+=	-=	
1	,						

Sommaire

Définitions des expressions

Expression

- Une constante littérale est une expression
  - 1, 2.34e-56, 'A', "bonjour"
- Une variable est une expression
- Une expression correcte formée par l'application d'un opérateur à une (unaire) ou deux (binaire) expressions respectant la syntaxe de l'opérateur est une expression
  - 1+(-1.56), 'A'*3, "bonjour", x=y+360*.56

Les instructions simples

Instruction vide

Syntaxe : **;**

Instruction expression

Syntaxe : **<expression>;**

Appel d'une fonction

Syntaxe : **fonction(...);**

Instruction bloc

Syntaxe :
{
déclarations
instructions
}

Opérateurs de comparaison

Comparaison

- Égalité : ==
- Supérioté stricte : >
- Supérioté non stricte : >=
- Inférioté stricte : <
- Inférioté non stricte : <=
- Différence : !=

Syntaxe

- <expression1> OP <expression2>
- <expression1> et <expression2> doivent être de type simple (entiers, flottants ou pointeurs).

Type du résultat

- Le résultat est égal à 0 (pour Faux) ou 1 (pour Vrai)

Opérateurs logiques

Conjonction : &&

- Syntaxe : <expression1> && <expression2>
- Opération : l'expression a pour valeur 0 si la valeur d'une des deux expressions est nulle et 1 sinon.

Disjonction : ||

- Syntaxe : <expression1> || <expression2>
- Opération : l'expression a pour valeur 0 si les valeurs des deux expressions sont nulles et 1 sinon.

Négation : !

- Syntaxe : !<expression>
- Opération : l'expression a pour valeur 1 si la valeur de l'expression est nulle et 0 sinon.

L'instruction if

Instruction if

- Syntaxe1 : if (< expression >) < instruction1 > else < instruction2 >
- Syntaxe2 : if (< expression >) < instruction >

L'instruction if

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i,j;
5        printf("Saisir deux entiers : ");
6        scanf("%d %d", &i, &j);
7        if(i < j) printf("%d est plus petit que %d \n", i, j);
8        else {
9                if(i > j) printf("%d est plus grand que %d \n", i, j);
10               else printf("%d est egal a %d \n", i, j);
11           }
12       return 0;
13   }
```

L'instruction if

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i,j;
5        printf("Saisir deux entiers : ");
6        scanf("%d %d", &i, &j);
7        if(i < j) printf("%d est plus petit que %d \n", i, j);
8        else if(i > j) printf("%d est plus grand que %d \n", i, j);
9        else printf("%d est egal a %d \n", i, j);
10       return 0;
11   }
```

L'instruction switch

Instruction switch

- Syntaxe :

```
switch (< expression >) {
    case <expression-constante1> : <instruction1>
    case <expression-constante2> : <instruction2>
    ...
    case <expression-constantek> : <instructionk>
    default : <instruction>
    }
```

L'instruction switch

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i;
5        printf("Saisir un entier : ");
6        scanf("%d", &i);
7        switch(i) {
8            case 1 : printf("je suis dans le cas 1 \n");
9            case 2 : printf("je suis dans le cas 2 \n");
10           case 3 : printf("je suis dans le cas 3 \n");
11           default : printf("je ne suis ni dans le cas 1, ni dans le cas 2, ni dans le cas 3 \n");
12           }
13       return 0;
14   }
```

L'instruction switch

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i;
5        printf("Saisir un entier : ");
6        scanf("%d", &i);
7        switch(i) {
8            case 1 : printf("je suis dans le cas 1 \n");
9                     break;
10           case 2 : printf("je suis dans le cas 2 \n");
11                    break;
12           case 3 : printf("je suis dans le cas 3 \n");
13                    break;
14           default : printf("je ne suis ni dans le cas 1, ni dans le cas 2, ni dans le cas 3 \n");
15           }
16       return 0;
17   }
```

Les instructions itératives

Instruction while

- Syntaxe : while(< expression >) < instruction >

Instruction dowhile

- Syntaxe : do < instruction > while(< expression >) ;

Instruction for

- Syntaxe :
  for(< expression1-opt > ;< expression2-opt > ;< expression3-opt >)
  < instruction >

Les instructions itératives

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i = 10;
5        while(i <= 20) {
6             printf("i = %d \n",i);
7             i++;
8             }
9        return 0;
10   }
```

Les instructions itératives

```
1  #include<stdio.h>
2  int main( )
3    {
4        int i = 10;
5        do {
6             printf("i = %d \n",i);
7             i++;
8             }
9        while(i <= 20);
10       return 0;
11   }
```

Les instructions itératives

```
1 #include<stdio.h>
2 int main( )
3   {
4       int i;
5       for(i=10 ;i<=20 ;i++) printf("i = %d \n",i);
6       return 0;
7   }
```

Les instructions itératives

Instruction break

- Syntaxe : break ;
- Opération : arrêt de la boucle.

Instruction continue

- Syntaxe : continue ;
- Opération : retour à l'évaluation de la condition de continuation.

Sommaire

Les fonctions

Programmation modulaire

- Décomposition en sous-problèmes "simples"
- Une fonction traite un sous-problème
- Une fonction peut éventuellement renvoyer un résultat

Définition : Syntaxes

- <type> <identificateur> (<déclaration-param>,...,<déclaration-param>)
  instruction-bloc
- <type> <identificateur> (void)
  instruction-bloc
- void <identificateur> (<déclaration-param>,...,<déclaration-param>)
  instruction-bloc
- void <identificateur> (void)
  instruction-bloc

Fonction de type non vide : return < expression > ;

Appel de fonctions

Syntaxe : appel d'une fonction

- <identificateur> (<argument>,...,<argument>)

Appel d'une fonction

- A chaque appel, le type de chaque argument est converti au type du paramètre formel correspondant
- Les types des arguments doivent donc être compatibles avec ceux des paramètres formels
- Par défaut, toute fonction peut en appeler une autre

Déclaration de fonctions

Déclaration par défaut

Si le compilateur rencontre une référence à une fonction dont il ne connaît pas encore la définition : il lui attribue par défaut le type int.

Déclaration d'une fonction

- définition de la fonction
- prototype de la fonction : <type> <identificateur> (<déclaration-ident>,...,<déclaration-ident>)

Passage de paramètres

Copie des arguments

- La transmission des paramètres se fait par valeur
- Pas de modifications des arguments
- Exception : les tableaux

Visibilités des variables

Types de variables

- Les variables de fichier (globales) : déclarées à l'extérieur de toute fonction
- Les variables de bloc : déclarées à l'intérieur d'un bloc
- Les paramètres d'une fonction

Variables globales

Visibles de leur déclaration à la fin du fichier

Variables de bloc

Visibles de leur déclaration à la fin du bloc contenant cette déclaration

Paramètres d'une fonction

Visibles du début à la fin du bloc associé à la fonction

Visibilités des variables

Variables de même nom

- Pas d'ambiguïtés possibles : redéclaration interdite
- La déclaration d'un paramètre / d'une variable de bloc masque localement la déclaration de toute variable / tout paramètre de même nom faite à l'extérieur du bloc.

Sommaire

Les tableaux

Collection de variables

Un tableau est une collection de variables de type identique.

Déclaration

- <type> <identificateur> [taille] ;
- <type> <identificateur> [taille] = {<expression$_1$>,...,<expression$_n$>} ;
- <type> <identificateur> [ ] = {<expression$_1$>,...,<expression$_n$>} ;

Éléments d'un tableau

- Les éléments du tableau sont rangés dans des cases mémoires contigües indicées à partir de 0
- Chaque élément d'un tableau a un comportement similaire à celui d'une variable
- Syntaxe d'accès aux éléments : <identificateur> [<indice>]

Les tableaux

Passage de paramètres

<type> identificateur_fonction(<type_element> identificateur_tableau[ ],...)

Les tableaux multidimentsionnels

Déclaration

- <type> <identificateur> [taille$_1$]...[taille$_k$] ;
- <type> <identificateur> [taille$_1$]...[taille$_k$] ;

Passage de paramètres

<type> identificateur_fonction(int taille$_1$,..., int taille$_k$, <type_element> identificateur_tableau[taille$_1$]...[taille$_k$],...)

Sommaire

Les pointeurs

Variable de type pointeur

Un pointeur est une variable dont la valeur est l'adresse d'une cellule de la mémoire

Adresse

Une adresse est un nombre entier qui donne l'indice d'un élément dans la mémoire de l'ordinateur

Valeur pointée

Un pointeur donne la possibilité de manipuler/modifier la valeur pointée (valeur se situant à l'adresse stockée dans la variable pointeur).

Déclaration et Initialisation de pointeurs

Déclaration pointeur typé

<type> * <identificateur> <= initialisation-opt >;

Déclaration pointeur atypique

void * <identificateur> <= initialisation-opt >;

```
1 #include<stdio.h>
2 int main( )
3   {
4       int var = 1;
5       int * pt = NULL;
6       int ** pt_pt = NULL;
7       int **** pt4 = NULL;
8       return 0;
9   }
```

L'opérateur &

L'opérateur &

L'opérateur & permet de connaître l'adresse d'une variable

Syntaxe

& <variable>

```
1 #include<stdio.h>
2 int main( )
3   {
4       int var = 1;
5       int * pt_var = &var;
6       int ** pt_pt = &pt_var;
7       return 0;
8   }
```

L'opérateur *

L'opérateur *

L'opérateur d'indirection * permet de manipuler/modifier la valeur pointée

Syntaxe

* <expression>

```
1 #include<stdio.h>
2 int main( )
3   {
4       int var = 1;
5       int * pt_var = NULL;
6       pt_var = &var;
7       *pt_var = 2 ;       return 0;
8   }
```

Arithmétique des adresses

Arithmétique des adresses

- Si <expression> est interprétée comme l'adresse d'une valeur O1 de type Type (grâce à un pointeur typé)
- + 1 : ajoute sizeof(Type) à <expression>
- + n correspond à (Type *)(<expression> + n*sizeof(Type))

Notations équivalentes

- *(<expression> + 1) équivaut à <expression>[1]
- ...
- *(<expression> + n) équivaut à <expression>[n]

Tableaux et Pointeurs

Identificateur d'un tableau

L'identificateur d'un tableau en C est un pointeur constant vers le premier élément du tableau

Notations équivalentes

Considérons la déclaration : int tab[10] ;

- tab est équivalent à &tab[0]
- *tab équivaut à tab[0]
- ...
- *(tab + n) équivaut à tab[n]

Passages de paramètres et Pointeurs

Exemple

Fonction d'échange des valeurs de deux arguments

```
1  #include<stdio.h>
2  void echangern(int * p, int *q)
3    {
4        int x = *p;
5        *p = *q;
6        *q = x;
7    }
8  int main( )
9    {
10       int a = 2 , b = 3;
11       int * p = &a, * q = &b;
12       echange(p, q);
13       return 0;
14   }
```

Conversion de type : opérateur de "cast"

Conversion de type

La conversion de type (cast) permet de modifier localement l'interprétation d'une valeur

Syntaxe

(nouveau_type) <expression>

Exemples

- (float) 1 / 2 : permet d'interpréter pour cette expression l'entier 1 comme un flottant et donc d'évaluer la valeur de l'expression dans les flottants.
- Un pointeur générique (atypique) ne permet pas l'utilisation et la modification de la valeur pointée : il est nécessaire de repasser à un pointeur typé pour accéder à la valeur pointée.

Conversion de type : opérateur de "cast"

Les conversions légitimes

- entier vers entier plus long : le codage est étendu de sorte que la valeur soit inchangée
- entier vers entier plus court : si la valeur est assez petite la valeur est inchangée, sinon elle est tronquée
- entier signé vers non signé et vice versa : l'interprétation du compilateur change alors que le codage reste inchangé
- flottant vers entier : la partie fractionnaire est supprimée
- entier vers flottant : le flottant obtenu est celui qui approche le mieux l'entier
- adresse d'une valeur de type Type1 vers adresse d'une valeur de type Type2 : l'interprétation change alors que le codage demeure inchangé

Opération rarement nécessaire

L'opération de "cast" est rarement nécessaire car l'objectif principal est de faire taire le compilateur

Sommaire

Les chaînes de caractères

Une chaîne de caractères

Une chaîne de caractères est un tableau de caractères qui se termine par le caractère NUL ('\0')

Constante chaîne de caractères

Une constante chaîne de caractères est délimitée par "..."

Variable contenant une chaîne de caractères

Une constante chaîne de caractères doit être stockée dans un tableau (suite contigüe de variables)

Pointeurs et chaînes de caractères

Un pointeur peut pointer vers le premier caractère d'une chaîne de caractères (stockée dans un tableau)

Les chaînes de caractères

```
1 #include<stdio.h>
2 int main( )
3   {
4       char * message = "Coucou le monde";
5       char tab1[ ] = "Coucou le monde";
6       char tab2[ ] = {'C','o','u','c','o','u',' ','l','e',' ','m','o','n','d','e','\0'};
7       return 0;
8   }
```

Manipulation de chaînes de caractères

scanf

La fonction scanf permet la saisie de chaînes de caractères grâce au spécificateur de type %s

```
1 #include<stdio.h>
2 int main( )
3   {
4       char chaine[100];
5       scanf("%s",chaine);
6       return 0;
7   }
```

Manipulation de chaînes de caractères

Bibliothèque string : gets vs fgets

- gets contient une faille
- Préférer fgets : fgets(machaine, expression-entiere, stdin) ;

Bibliothèque string : d'autres fonctions

- strlen (calcule la longueur d'une chaîne)
- strcpy (copie une chaîne dans une autre)
- strcat (concatène deux chaînes)
- strcmp (compare deux chaînes)
- strchr (recherche d'un caractère dans une chaîne)
- ...

Sommaire

La gestion dynamique de la mémoire

Opérations

- Réservation d'un emplacement mémoire par demande d'allocation dynamique
- Libération d'un emplacement mémoire réservé de manière dynamique

Fonctions

Bibliothèque stdlib

- void * malloc (size_t nb_octets)
- void * calloc (size_t nb, size_t taille)
- void * realloc (void * pointeur, size_t nb_octets)
- void free (void * pointeur)

Malloc

Malloc

- void * malloc (size_t nb_octets)
- malloc (nb_octets) renvoie un pointeur sur une zone de la mémoire de taille nb_octets ou NULL en cas d'échec

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int main( )
4   {
5       int * pt = NULL ;
6       pt = malloc(sizeof(int)) ;
7       if(pt == NULL) printf("Echec allocation dans malloc") ;
8       return 0;
9   }
```

Calloc

Calloc

- void * calloc (size_t nb, size_t taille)
- calloc(nb, taille) renvoie un pointeur sur une zone de la mémoire permettant de ranger nb objets de grandeur taille ou NULL en cas d'échec

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int main( )
4   {
5       int * pt = NULL ;
6       pt = calloc(5, sizeof(int)) ;
7       if(pt == NULL) printf("Echec allocation dans calloc") ;
8       return 0;
9   }
```

Realloc

Realloc

- void * realloc (void * pointeur, size_t nb_octets)
- realloc(pt, nb_octets) renvoie un pointeur vers une zone mémoire de taille nb_octets contenant au début les éléments stockés dans une zone précédemment allouée et repérée par pt, ou, en cas d'échec, retourne le pointeur NULL et le bloc pointé par pt peut être détruit.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main( )
4    {
5       int * pt1 = NULL, * pt2 = NULL ;
6       pt1 = calloc(5, sizeof(int)) ;
7       if(pt1 == NULL) printf("Echec allocation dans calloc") ;
8       else {
9                    pt2 = realloc(pt1, 10*sizeof(int)) ;
10                   if(pt2 == NULL) printf("Echec allocation dans realloc") ;
11           }
12      return 0;
13   }
```

Free

Calloc

- void free (void * pointeur)
- free(pointeur) libère l'emplacement mémoire précédemment alloué et repéré par pointeur

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main( )
4    {
5        int * pt1 = NULL ;
6        pt1 = calloc(5, sizeof(int)) ;
7        if(pt1 == NULL) printf("Echec allocation dans calloc") ;
8        else free(pt1) ;
9        return 0;
10   }
```

Les tableaux dynamiques

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main( )
4    {
5        int n, *tab = NULL ;
6        printf("Donner la taille du tableau") ;
7        scanf("%d", &n) ;
8        tab = calloc(n,sizeof(int)) ;
9        remplir(tab,n) ;
10       afficher(tab,n) ;
11       return 0;
12   }
```

Les tableaux dynamiques multidimensionnels

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main( )
4    {
5        int i, n, m, **matrice = NULL ;
6        printf("Donner le nombre de lignes") ;
7        scanf("%d", &n) ;
8        printf("Donner le nombre de colonnes") ;
9        scanf("%d", &m) ;
10       matrice = calloc(n,sizeof(int*)) ;
11       for(i = 0 ; i < n ; i++) matrice[i] =calloc(m, sizeof(int)) ;
12       remplir(matrice,n,m) ;
13       afficher(matrice,n,m) ;
14       return 0;
15   }
```

Organisation d'un programme en mémoire

Organisation de la mémoire

En mémoire, le stockage des informations associées à un programme est fait dans l'ordre suivant :

- Le code du programme
  (les instructions)
- La zone des données statiques
  (variables globales, les variables locales statiques, les constantes)
- Le tas qui contient les données gérées de manière dynamique
  (allocation/libération de mémoire)
- La pile qui contient les variables automatiques
  (variables locales et paramètres)