

Informatique (Algorithmique et Langage C)

TABLE DES MATIÈRES

1. Algorithme et organigramme	2
1.1. Généralités	2
1.2. Principales structures algorithmiques	3
2. Le passage au langage de programmation	6
2.1. Quel langage ?	6
2.2. Le Langage C : un peu d'histoire	6
2.3. Mon premier programme C	7
3. Les briques de bases du langage C	7
3.1. généralités	7
3.2. Les variables	8
3.3. Les entrées/sorties	9
3.4. Résumé des principales instructions	10
3.5. Quelques exemples	11
3.6. La bibliothèque standard du C	12
3.7. Ce qu'il faut retenir des briques de bases en C	13
4. Principe de base de la compilation	14
4.1. Compiler un programme	14
4.2. debugger un programme	14
5. Opérateurs logiques	14
6. Les tableaux	15
6.1. Définition et manipulation	15
6.2. Gestion dynamique des tableaux	16
6.3. Exercice	17
7. Notion de sous-programme	18
7.1. Généralités	18
7.2. Retour sur les variables	18
7.3. Exemples	19
8. Programmation récursive	20
8.1. Principe	20
8.2. En C	20
9. Les pointeurs	21
9.1. Sémantique et manipulation	21
9.2. Occupation mémoire	22
9.3. Arithmétique des pointeurs	22
10. Les chaînes de caractères	24
10.1. Utilisation	24
10.2. Manipulation	24
11. Les fichiers	25
11.1. Manipulation	25

11.2. Lecture/écriture	25
12. Types énumérés et types structurés	26
12.1. Besoin de nouveaux types	26
12.2. Type énuméré	26
12.3. Type Structuré	26
13. Quelques types complexes	27
13.1. Les listes chaînées	27

1. ALGORITHME ET ORGANIGRAMME

1.1. Généralités.

Algorithme.

Définition

Un algorithme est un ensemble de règles opératoires rigoureuses, ordonnant à un processeur d'exécuter dans un ordre déterminé une succession d'opérations élémentaires, pour résoudre un problème donné. C'est un outil méthodologique général qui ne doit pas être confondu avec le programme proprement dit.

Un algorithme peut être :

- représenté graphiquement par un **organigramme** (ou **ordinogramme**),
- écrit sous forme littérale avec un langage algorithmique.

Mon premier algorithme.

la recette du brownie

- (1) Mélanger les sucres semoule et vanillé, les oeufs et la farine tamisée
- (2) Faire fondre le beurre avec le chocolat
- (3) Mélanger le beurre et le chocolat à la pâte
- (4) Mélanger les noix de Pécan et la poudre d'amande à la pâte
- (5) Versez la pâte dans un moule à gâteau beurré
- (6) Mettre à cuire 35 minutes dans le four préchauffé à 170°

Mon deuxième algorithme.

L'addition en colonnes

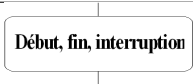
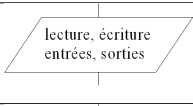

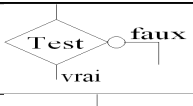

- (1) Ecrire chaque nombre sur une ligne en alignant les chiffres de même poids
- (2) Se positionner sur la colonne la plus à droite (chiffre de plus faible poids) et initialiser la retenue à 0
- (3) Additionner la retenue et tous les chiffres de la colonne courante
- (4) Mettre à jour la retenue qui devient égale à la somme précédemment obtenue à laquelle le chiffre des unités a été retiré.
- (5) Reporter dans la ligne résultat le chiffre des unités de la somme, et dans la ligne retenue, la nouvelle retenue
- (6) Se positionner sur la colonne suivante (la plus proche à gauche de la colonne courante)
- (7) Recommencer toutes les étapes depuis le point 3 si la colonne contient encore au moins un chiffre ou si la retenue est différente de 0

Généralités sur les algorithmes.

On peut remarquer que

- les algorithmes comportent une ou plusieurs entrées
- peuvent renvoyer un résultat en sortie
- On peut les comparer à “fonctions” mathématiques (on associe une sortie à des entrées)
- Les instructions sont séquentielles, c-à-d. elles se suivent et doivent être réalisées l’une après l’autre : on ne peut pas faire cuire le brownie avant d’avoir mis la pâte dans le moule.

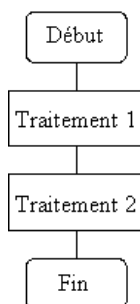
Organigramme : symboles.

Le début, la fin ou l’interruption d’un programme	
Mise à disposition d’une information à traiter ou enregistrement d’une information traitée.	
Les opérations ou groupes d’opérations à effectuer sur les données, les instructions, . . . , ou opération pour laquelle il n’existe aucun symbole	
Les tests ou branchements conditionnels	
Appel de sous-programmes	

1.2. Principales structures algorithmiques.

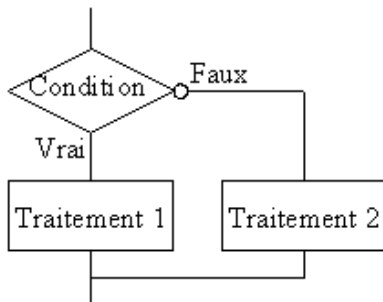
Structure linéaire ou séquence.

La structure linéaire se caractérise par une suite d’actions à exécuter successivement dans l’ordre de leur énoncé.

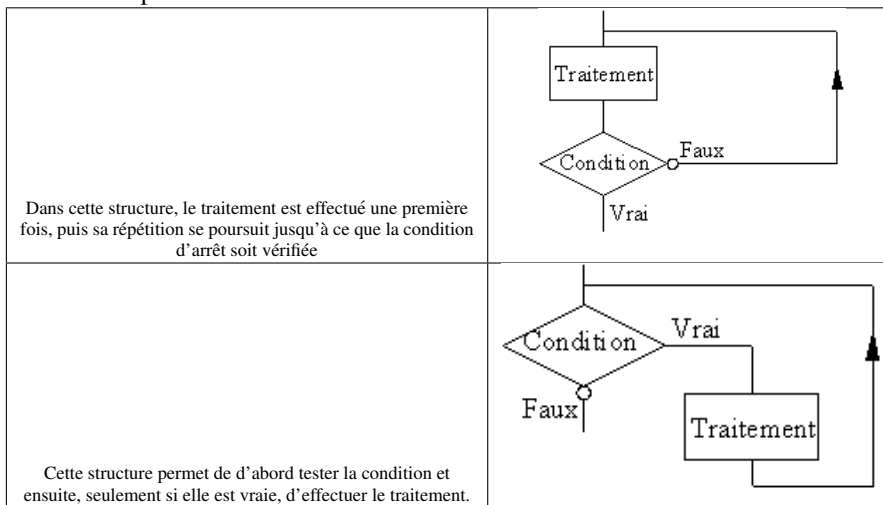


Structure alternative.

Une structure alternative n’offre que deux issues possible s’excluant mutuellement. Les structures alternatives définissent une **fonction de choix** ou de **sélection** entre l’exécution de l’un ou de l’autre des deux traitements. Également nommées **structures conditionnelles**, elles traduisent un **saut** ou une rupture de séquence dans un algorithme.



Structures répétitives.

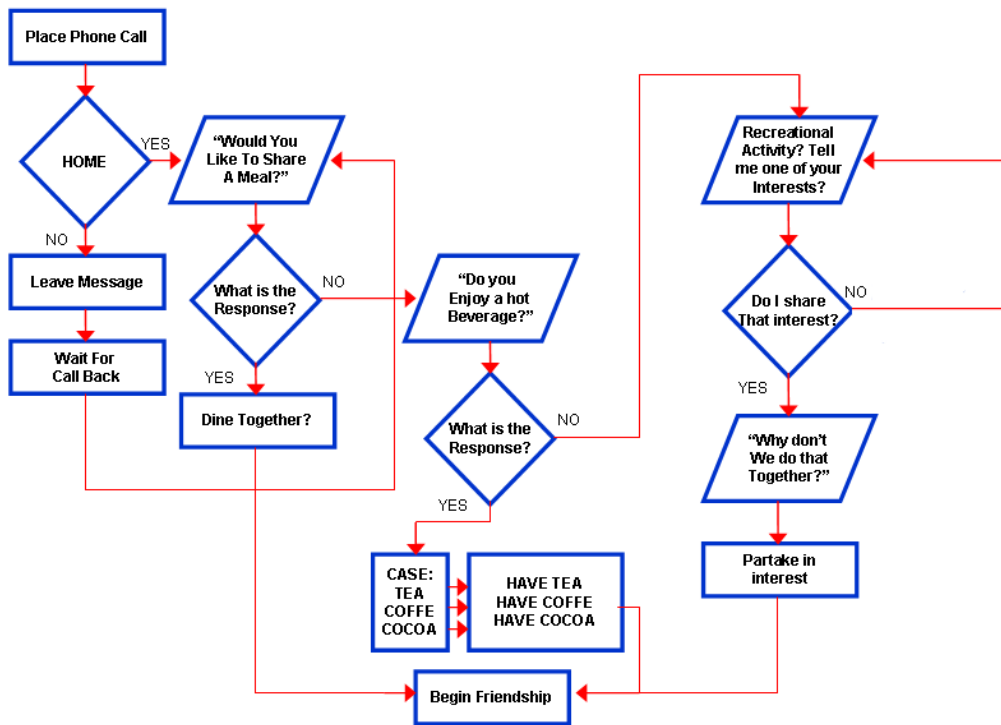


Algorithme → organigramme.

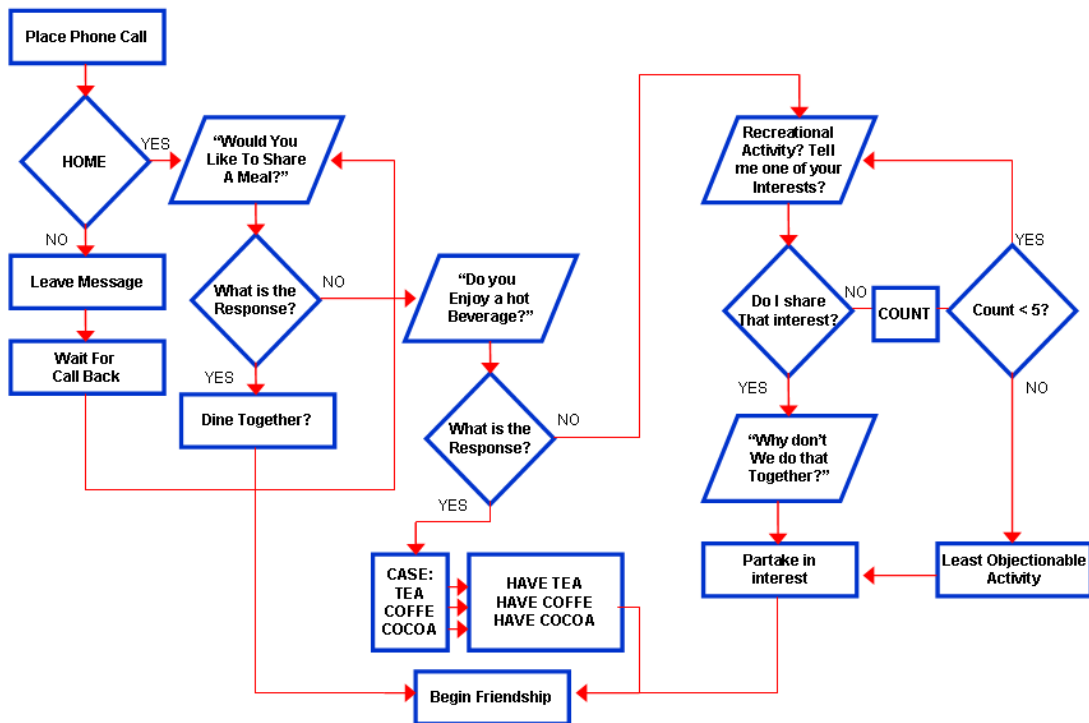
Déterminer si un nombre est pair

- Lire un nombre
- Vérifier s'il est divisible par 2
- Si oui, afficher "Le nombre est pair", sinon afficher "Le nombre est impair"

The friendship algorithm.



The friendship algorithm.



2. LE PASSAGE AU LANGAGE DE PROGRAMMATION

2.1. Quel langage ?

L'univers des langages de programmation.

Comment choisir son langage ?

Il existe des centaines de langages de programmation...

- Pour choisir un langage, il convient de définir certaines propriétés du programme qui doit être développé :
 - durée de vie du programme
 - programme commercial ?
 - programme opensource ?
 - portabilité du programme ?
 - temps de réponse attendu
 - ...
- Il faut également prendre en considération les outils de développement disponibles (éditeurs avancés, débogueur, compilateurs efficaces, ...)
- Il faut étudier l'existant et essayer de réutiliser le plus d'éléments déjà conçus, développés et testés
- ...

2.2. Le Langage C : un peu d'histoire.

Le langage C.

Les origines

- Créé en 1972 par B. Kernighan et D. Ritchie en s'inspirant des langages B (1969) et BCPL (1966)
- Langage de bas niveau conçu pour manipuler directement des "mots machine" (bits, octets)
- A été massivement utilisé pour développer des systèmes d'exploitation entre 1975 et 1993
- Fortement orienté programmation système
- Extrêmement utilisé dans la programmation embarquée sur microcontrôleurs, les calculs intensifs, les systèmes d'exploitation et tous les domaines où la rapidité de traitement est prioritaire

Le langage C.

La popularité

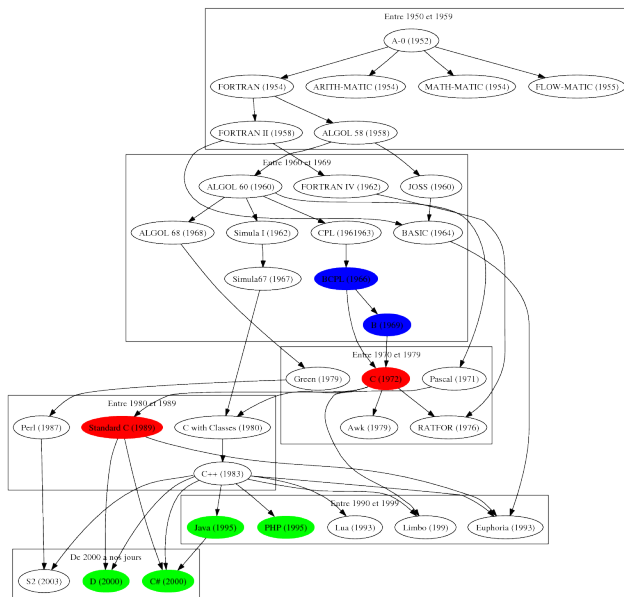
- C a été conçu pour être très facilement portable : Kernighan et Ritchie estimaient qu'un nouveau compilateur pour le C pouvait être écrit en deux mois
- Présent sur de très nombreuses architectures de processeurs

La descendance

De nombreux langages plus récents sont directement liés au langage C ou s'en sont fortement inspirés

- 1983 : C++
- 1995 : Java, JavaScript, PHP
- 2003 : C#

Un bref historique du langage C.



2.3. Mon premier programme C.

Hello world !

Le code .c

```

1  #include <stdio.h>
2
3  int main () {
4      printf("Hello_world_!\n");
5
6      return 0;
7  }
```

Le code .c commenté

```

1  #include <stdio.h> /* Header inclusion */
2
3  /* main function: must return an integer */
4  int main () {
5      printf("Hello_world_!\n"); /* print on the screen */
6
7      return 0; /* return value */
8  }
```

3. LES BRIQUES DE BASES DU LANGAGE C

3.1. généralités.

Fichiers Headers.

- ils sont inclus avec la commande `#include` (ex : `#include <stdio.h>`)
- Les fichiers `.h` inclus au débuts sont les *headers*
- Ils permettent d'utiliser des fonctions déjà programmées (ex : `printf`)

La fonction `main`.

- C'est LE point d'entrée de tout programme C
- Quand un fichier est exécuté, le point de départ est la fonction `main`
- À partir de cette fonction, le programme se déroule selon les choix du programmeur
- Il peut y avoir d'autres fonctions appelées dans le `main`
- Tout programme C **doit avoir une et une seule** fonction `main`

Les commentaires.

- Les commentaires sont ignorés par le compilateur
- Les commentaires s'écrivent entre : `/* */`
`/* commentaire */`
- On peut aussi utiliser les commentaires C++ : `//`
`// commentaire`

définition de macros : `#define`.

- se déclare juste après les `include`
- permet de substituer du code
- on se limite ici aux macros "simples".
- par convention, les macros ont un nom majuscule.

Syntaxe

```
#define <NOM> <CODE>
```

Exemples

- `#define TAILLE 100`
- `#define PI 3.14`

3.2. Les variables.

Variables et leur nom.

Variables

Une variable désigne une information qui peut être modifiée au cours du programme. Elle permet :

- de mémoriser une information provenant de l'extérieur (fournie par l'utilisateur par exemple)
- de stocker le résultat d'une opération.
- ce n'est pas une variable mathématique (ou algébrique) !

C'est donc un *nom* qui permet de repérer un *emplacement mémoire*.

Nom des variables

- Uniquement les 26 caractères de l'alphabet (majuscule et minuscule), les chiffres, et l'underscore "`_`"
- Pas de caractères spéciaux (accents, `&`, espaces etc.)
- Le premier caractère ne doit pas être un chiffre
- Le nom doit faire au plus 32 caractères
- Les majuscules sont distingués des minuscules : `a` et `A` sont deux noms différents.

Variables : Types et déclaration.

Types

Les variables sont dites typées : un entier n'est pas codé de la même manière qu'un réel. Il existe plusieurs types de base :

- `char` (caractères) Par exemple : `'a'...'z'`, `'A'...'Z'`, ...
- `int` (entiers) Par exemple : 129, -45, 0, ...
- `float` (réels) Par exemple : 3.14, -0.005, 67.0, ...
- Voir la Tableau 1 du *C dans la poche*

Déclaration

- de façon générale :
 - <type> <nom> ;
 - <type> <nom1>, <nom2>, <nom3> ;
- exemples :
 - int a;
 - float valeur, res;
- Déclarer une variable revient à lui réserver un emplacement mémoire. On ne connaît pas sa valeur initiale !

Variables : affectation.

But

- Stocker une information dans un variable, pour la réutiliser.
- L'opérateur d'affectation est le signe =. Ne pas confondre avec le test d'égalité !
- <variable> = <expression> ;
- Se lit de **droite à gauche**, en *évaluant* toutes les expressions à droite du signe =.

Exemple

```

1  int n;
2  int p;
3
4  n = 10;
5
6  p = 2*n-3;

```

Variables : Opérations élémentaires – 1.

généralités

- Toutes les opérations élémentaires de bases sont permises : + , - , * , /
- Ces opérations dépendent du type des variables !

cas du type int

- la division / est la division **euclidienne** (ou entière). Ex : 11 / 4 = 2 et non pas 2.75 !)
- il existe l'opérateur **modulo** %, qui donne le *reste* de la division euclidienne. Ex : 11%4 = 3

Variables : Opérations élémentaires – 2.

cas du type float

- la division / donne un résultat décimal
- L'opérateur **modulo** % n'existe pas

mélange de type

Attention aux *transtypages* implicites !

- le résultat d'une opération entre un int et un float est un float.
- Si on affecte un float, dans un int, on convertit la partie entière du float

Quelques opérateurs particuliers.

Quelques opérateurs bien pratiques

- $i++$: opérateur permettant d'ajouter une unité à la variable i (de type int ou char)
- $i--$: idem mais pour retirer une unité
- $x* = y$, $x/ = y$, $x- = y$, $x+ = y$: opérateurs permettant de multiplier (diviser, soustraire ou ajouter) x par y (pas de restriction de type)

3.3. Les entrées/sorties.

Affichage avec `printf`.

- Permet d'écrire sur la sortie standard (la console)
- `printf('<chaine de caractères> ', arg1, arg2, ..., argn);`

Exemples

- `printf('hello \n ');`
- `int x=10; printf('valeur de x = %d\n ',x);`
- `int x=10; float y = 3.4; printf('valeur de x = %d et de y+x = %f\n ',x,y+x);`

Principaux codes de format

- `%d` pour le type `int`
- `%c` pour le type `char`
- `%f` pour le type `float`

Caractères de mise en forme

- `\n` : retour à la ligne
- `\r` : retour en début de ligne courante
- `\t` : tabulation

Saisie avec `scanf`.

- Permet de lire sur l'entrée standard (le clavier)
- `scanf('<code format>', &<variable>);`

avec `<code format> = %d, %f ou %c` pour lire un entier, un flottant ou un caractère.

Exemples

- `int n; scanf('%d', &n);`
- `float x; scanf('%f', &x);`

3.4. Résumé des principales instructions.

Instruction conditionnelle.

L'instruction *if*

permet de programmer

- une structure de choix :

```
1  if (condition_logique) {
2    ...
3  }
4  else{
5    ...
6  }
```

- une exécution conditionnelle :

```
1  if (condition_logique) {
2    ...
3  }
```

Le choix multiple : l'instruction `switch`.

`switch`

```
1  switch( variable ){
2    case valeur1:
3    instructions;
4    break;
5
6    case valeur2:
7    instructions;
8    break;
9  }
```

```

10  case valeur3:
11      instructions;
12      break;
13
14  default:
15      instructions;
16  }

```

Structure de répétition.

boucle conditionnelle

– de type $N + 1$:

```

1  do{
2  ...
3  }while(condition_logique)

```

– de type N

```

1  while(condition_logique){
2  ...
3  }

```

répétition inconditionnelle : le *for*

```

1  for (i=0; i<n; i++){
2  ...
3  }

```

3.5. Quelques exemples.

Quelques exemples.

Deux programmes

- Ecrire un programme qui affiche les 10 premières puissances de 2
- Ecrire un programme qui affiche les n premières puissances de x

Les 10 premières puissances de 2.

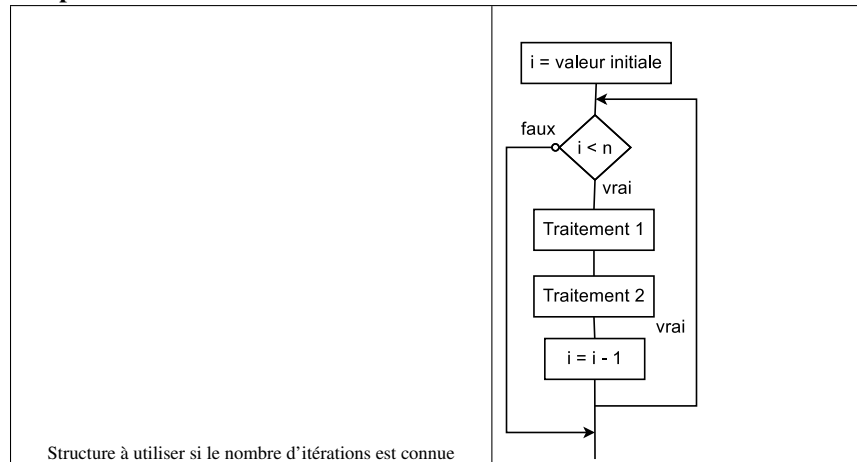
```

1  #include <stdio.h>
2
3  #define NB_PUISS 10
4
5  int main (void) {
6      int i, puiss_2;
7
8      puiss_2 = 1;
9      printf ("Calcul_des_%d_premieres_puissances_de_2\n", NB_PUISS);
10     i = 0;
11     while (i < NB_PUISS) {
12         printf ("2^%d=%d\n", i, puiss_2);
13         puiss_2 = puiss_2 * 2;
14         i = i + 1;
15     }
16
17     return 0;
18 }

```

Nouvelle forme de boucle.

Structure de répétition contrôlée



Les n premières puissances de x .

```

1  #include <stdio.h>
2
3  int main () {
4      int i, puiss_x, n, x;
5
6      puiss_x = 1;
7      printf ("Calcul_des_n_premieres_puissances_de_x\n");
8
9      printf ("Indiquer_la_valeur_de_n:");
10     scanf ("%d", &n);
11
12     printf ("Indiquer_la_valeur_de_x:");
13     scanf ("%d", &x);
14     for (i=0; i < n; i++) {
15         printf ("%d^%d=%d\n", x, i, puiss_x);
16         puiss_x = puiss_x * x;
17     }
18
19     return 0;
20 }
  
```

3.6. La bibliothèque standard du C.

Aperçu des bibliothèques existantes.

- <stdio.h> : fonctions utiles pour les entrées/sortie de base
- <stdlib.h> : fonctions générales (manipulation mémoire, tirage aléatoire etc.)
- <string.h> : manipulation des chaînes des caractères
- <math.h> : fonctions mathématiques de base (sin, cos, sqrt, ...)

Quelques fonctions de <stdio.h>.

stdio : standard input/output

Bibliothèque d'entrée/sortie standard

- printf : Permet d'afficher sur la sortie standard (en général, la console), du texte formaté
- scanf : Permet de lire et de stocker des informations sur l'entrée standard (en général, le clavier)

Quelques fonctions de <stdlib.h>.

stdlib : standard library

- calloc : Permet d'allouer dynamiquement de la mémoire
- free : Permet de libérer de la mémoire qui a été dynamiquement allouée

- rand : Permet d'obtenir des valeurs aléatoires

Quelques fonctions de <string.h>.

string : chaîne de caractères

Bibliothèque de manipulation des chaînes de caractères (copie, longueur, recherche...)

- strcmp : Permet de comparer deux chaînes de caractères
- strcpy : Permet de copier une chaîne de caractères dans une autre
- strlen : Permet de connaître la taille d'une chaîne de caractères

Quelques fonctions de <math.h>.

math

Bibliothèque contenant des fonctions mathématiques de base.

- sqrt : Permet d'obtenir la racine carrée d'un flottant
- cos, sin : Permet d'obtenir le cosinus (resp. sinus) d'une valeur
- exp, log, log2, log10 : ...
- Définition de la constante π : M_PI

3.7. Ce qu'il faut retenir des briques de bases en C.

Aspect général d'un programme C.

```

1  /* declaration des entetes*/
2  #include <stdio.h>
3  #include <stdlib.h>
4  /* etc. */
5
6
7  /* definition des macros */
8  #define MACRO_1 100
9  #define MACRO_2 42.10
10
11 /* fonction principale */
12 int main(){
13     /* declaration des variables */
14     int ma_variable_1;
15     float ma_variable_2;
16     float ma_variable_3;
17
18     /* instructions executables */
19     ma_variable_3 = ma_variable_2/ma_variable_3;
20
21     printf("ma_variable_3_vaut:_%f\n",ma_variable_3);
22
23     /* valeur de retour */
24     return 0;
25 }
```

En résumé.

Bibliothèque standard du C

#include + <stdio.h>, <stdlib.h>, <math.h>, <string.h>

Les différents types du C

char, int, float

Les opérateurs spécifiques aux entiers

- division entière : /
- reste de la division entière : %

Les variables

- Toujours choisir le bon type
- Toujours initialiser ces variables

4. PRINCIPE DE BASE DE LA COMPILATION

4.1. Compiler un programme.

Les bases de la compilation.

Les trois étapes

- La précompilation : `gcc -E test.c`
 - Les `#include` et les `#define` sont résolus.
- La compilation : `gcc -c test.c`
 - Le fichier `test.o` est créé.
 - La vérification des fonctions et de leur paramètres est réalisée lors de cette étape.
- L'édition de liens et la création de l'exécutable : `gcc -o test test.o tris.o`
 - Vérification que les fonctions définies dans les `.h` sont bien définies dans les `.o` correspondants et création du programme exécutable.

Les options de compilation.

Les options de gcc

- `-E` : précompilation
- `-c` : compilation
- `-g` : mode débogage
- `-ansi` : impose la vérification du format ANSI
- `-Wall` : active la détection de tous les warnings

4.2. debugger un programme.

Aide sur le langage C.

Utilisation des pages de man de C

En cas de doute sur les paramètres d'une fonction, ...

- Utiliser les pages de **man** du C
- Dans une console : `man printf` (ou toute autre fonction)
- Sur google : `man scanf`

Trouver les erreurs simples de vos programmes.

```
#include <stdio.h>
int main()
{
    printf("Hello world\n");
    return (0)
}
```

Console + gcc

```
helloWorld.c : In function 'main' :
helloWorld.c :5 : erreur : expected ';' before '}' token
```

5. OPÉRATEURS LOGIQUES

Valeurs logiques.

- Pas de type booléen en C
- La valeur **Faux** est représentée par la valeur 0
- La valeur **Vrai** est représentée par la valeur 1

Opérateurs de comparaison.

- test égalité : $x == y$ (renvoie 1 si x est égal à y et 0 sinon)
- test d'inégalité : $x != y$ (renvoie 1 si x est différent de y et 0 sinon)
- comparaison de valeurs ordonnées : $x < y$ (renvoie 1 si x est inférieur à y et 0 sinon) idem avec $<=$, $>$ et $>=$

Opérations logiques.

- Et logique : $A \&\& B$ (renvoie 1 si **A et B** sont vrais et 0 sinon)
- Ou logique : $A \|\| B$ (renvoie 1 si **A ou B** sont vrais et 0 sinon)
- Négation : $!A$ (renvoie 1 si A est faux et 0 sinon)

6. LES TABLEAUX

6.1. Définition et manipulation.

Qu'est-ce qu'un tableau ?

Tableau : Définition

- Structure de données permettant d'effectuer un même traitement sur des données de même nature.

tableau à une dimension

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

tableau à deux dimensions

Traitements Possibles.

On veut pouvoir :

- **créer** des tableaux
- **ranger** des valeurs dans un tableau
- **récupérer et consulter** des valeurs rangées dans un tableau
- **rechercher** si une valeur est dans un tableau
- **mettre à jour** des valeurs dans un tableau
- **modifier** la façon dont les valeurs sont rangées dans un tableau
- **effectuer des opérations** entre tableaux : comparaison, *etc.*

Intérêt des tableaux.

Stockage de valeurs

- Un tableau permet de stocker un nombre de valeurs prédéfinies.
- Evite de calculer plusieurs fois les mêmes informations.
- Par exemple : calcul des 10 premières puissances de 2 et utilisation du résultat

Manipulation des tableaux – 1.

Déclaration

La déclaration d'un tableau implique de connaître **lors de l'écriture du programme** :

- Le type des valeurs stockées
- La quantité de valeurs stockées

La syntaxe est la suivante :

type-des-valeurs-stockées nom_du_tableau [nombre-de-valeurs-stockées] ;

Exemple

Le tableau permettant de stocker les 10 premières puissances de 2 se déclare de la manière suivante : `int puiss10_2[10];`

Manipulation des tableaux – 2.

Accès au tableau

- Les tableaux sont indicés à partir de la case 0.
- La syntaxe suivante permet l'accès aux valeurs stockées d'un tableau contenant n valeurs : **nom_du_tableau** **[0] ... nom_du_tableau** **[$n - 1$]**

Les 10 premières puissances de 2.

```

1  #include <stdio.h>
2
3  #define TAILLE 10
4
5
6  int main () {
7      int puiss10_2[TAILLE];
8      int i, puiss_2 = 1;
9
10     printf ("calcul_des_%d_premieres_puissances_de_2\n", TAILLE);
11     for (i=0; i < TAILLE; i++) {
12         puiss10_2 [i] = puiss_2;
13         puiss_2 = puiss_2 * 2;
14     }
15
16     for(i=0; i < TAILLE; i++)
17         printf ("2^%d=%d\n", i, puiss10_2[i]);
18
19
20     return 0;
21 }
```

Principale limitation des tableaux.

Nombre de valeurs à stocker inconnu...

- Le nombre de valeurs à stocker n'est pas toujours connu lorsque le programme est écrit (par exemple, calcul des n premières puissances de x)
- Il s'agit du cas général de l'utilisation des tableaux
- Besoin de pouvoir déclarer et manipuler un tableau pouvant accepter un nombre "quelconque" de valeurs

6.2. Gestion dynamique des tableaux.

Gestion dynamique des tableaux.

Déclaration et manipulation

- Il est possible de gérer un tableau de manière dynamique
- Le tableau est déclaré de la manière suivante : `type-des-valeurs-stockées * nom_du_tableau;`
- Le nombre de valeurs stockées dans le tableau (donc la taille du tableau) est alors défini par l'instruction suivante : `nom_du_tableau = calloc(nombre-de-valeurs-à-stocker, occupation-mémoire);`
- L'occupation mémoire d'un élément se compte en **octet**. La fonction prédéfinie `sizeof(type)` permet de connaître le nombre d'octets associé à n'importe quel **type**.
- possibilité d'utiliser la **macro** `sizeof *<nom_du_tableau>`

Exemple

```

int * puissN_X; puissN_X = calloc (n, sizeof(int));
ou
int * puissN_X; puissN_X = calloc (n, sizeof *puissN_X);
```


Gestion dynamique des tableaux.

Libération d'un tableau dynamique

- Tous les tableaux alloués dynamiquement doivent être libérés.
- L'instruction `free` permet de libérer un tableau.
- Exemple : `free (puissN_X);`

Tableaux à deux dimensions – 1.

Déclaration

- Gestion dynamique : `int ** tab2d;`
- Gestion statique : `int tab[10][8]` (pour déclarer un tableau contenant 10 lignes et 8 colonnes)

Manipulation identique à celle des tableaux à une dimension : par exemple `tab2d[5][2] = 4;`

Tableaux à deux dimensions – 2.

Allocation et libération de la mémoire

- Commencez par allouer une dimension
- Puis allouer la deuxième dimension pour chaque ligne :

```
1 tab2d = calloc(nb_lignes, sizeof *tab2d);
2 for(i=0; i< nb_lignes; i++){
3     tab2d[i] = calloc(nb_colonnes, sizeof **tab2d);
4 }
```

- Idem pour la libération : libérer les lignes une par une, puis l'ensemble des lignes

```
1 for(i=0; i< nb_lignes; i++){
2     tab2d[i] = free(tab2d[i]);
3 }
4 free(tab2d);
```

Ce qu'il faut retenir des tableaux.

Déclaration et manipulation

- Déclaration statique : `int tab[100];`
- Déclaration dynamique : `int * tab;`
- Allocation : `int * tab = calloc (100, sizeof(int));`
- Accès : `tab[i]` ($0 \leq i \leq \text{taille} - 1$)
- Libération : `free (tab);`

Les fonctions utiles

- `calloc` et `free` définies dans `<stdlib.h>`
- `sizeof` définie dans `<stdio.h>`

6.3. Exercice.

Exercice de gestion dynamique de tableaux.

Enoncé

Écrire un programme permettant d'afficher la représentation binaire d'un nombre (sous notation décimale).

Exemples

- $10 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 \rightarrow 1010$
- $27 = 2^4 + 2^3 + 2^1 + 2^0 \rightarrow 11011$

Travail à réaliser

- (1) Trouver comment décomposer un nombre sous forme décimale en sa représentation binaire
- (2) Traduire l'algorithme en langage C

7. NOTION DE SOUS-PROGRAMME

7.1. Généralités.

Notions abordées.

Points clés

- Définition de briques élémentaires pouvant être réutilisées
- Règles pour définir un sous-programme
 - passage de paramètres typés
 - renvoi d'un unique résultat typé
- **Portée des variables**

Intérêt des sous-programmes.

- Permet de concevoir, d'écrire et de tester une fonction particulière
- Assure la réutilisabilité simple de fonctions plus ou moins complexes
- Permet la conception et la construction de programmes complexes à partir de briques élémentaires

Principe.

Tous les sous-programmes sont des fonctions :

- Accepte zéro ou plusieurs paramètres
- Renvoie au plus une valeur de sortie
- Peuvent effectuer tous type de traitement, y compris appeler d'autres sous-programmes

Syntaxe.

type-de-retour NomDeLaFonction (**liste-des-parametres**)

- Si la fonction ne renvoie aucune valeur, le type de retour est **void**
- Si la fonction renvoie un paramètre, celui-ci est retourné par l'instruction **return valeur** ; (le premier **return** exécuté termine la fonction)
- Si la fonction ne prend pas de paramètre, alors le type **void** est l'unique paramètre
- Sinon, les paramètres sont séparés par des **,** et déclarés de la manière suivante : **type1 var1, type2 var2, ...**

Appel d'une fonction (sous-programme).

- Si la fonction ne prend pas de paramètre, alors l'appel est le suivant : **NomDeLaFonction ()**
- Sinon, les paramètres doivent être fournis dans l'ordre et la quantité attendu

Une fonction particulière : **main**.

La fonction principale d'un programme

- La fonction principale, **main**, est unique et doit toujours être présente dans un programme
- La norme ANSI a normalisée cette fonction
 - type de retour : **int** (code d'erreur interprété par le programme appelant, **return 0** ; si aucune erreur a signaler)
 - paramètres :
 - **int argc** : permet de connaître le nombre de paramètres de la fonction
 - **char * argv[]** : liste des paramètres de la fonction
- **int main (int argc, char * argv[]);**

7.2. Retour sur les variables.

La portée des variables.

- Un programme est constitué de plusieurs fonctions
- Le texte de chaque fonction est indépendant des autres fonctions
- La communication entre les fonctions ne peut se faire que par les paramètres des fonctions et les résultats renvoyés
- Les variables d'une fonction sont indépendantes des variables d'une autre fonction (même si elles sont nommées de la même manière)

Exemple illustrant la portée des variables.

```

1  #include <stdio.h>
2
3  void Compte10 (int debut);
4
5  int main (int argc, char * argv[]) {
6      int i;
7      for (i=0; i < 100; i+=10) { Compte10 (i); }
8      return 0;
9  }
10
11 void Compte10 (int debut) {
12     int i;
13     for (i=debut+10; i > debut; i--) { printf ("%d_", i); }
14     printf ("\n");
15 }

```

7.3. Exemples.

Notion de sous-programme.

Exemple 1

Ecrire un programme qui affiche les 10 premières puissances de 2

- Simple à écrire depuis le programme précédemment vu
- Aucun paramètre à définir : mot clé **void** en guise de paramètre
- Aucune valeur de sortie : type de retour = **void**

Les 10 premières puissances de 2.

```

1  #include <stdio.h>
2
3  void Puiss2 (void) {
4      int i, puiss_2;
5
6      puiss_2 = 1;
7      printf ("calcul_des_10_premieres_puissances_de_2\n");
8      for (i=0; i < 10; i++) {
9          printf ("2^%d=%d\n", i, puiss_2);
10         puiss_2 = puiss_2 * 2;
11     }
12 }
13
14 int main (int argc, char * argv[]) {
15     Puiss2 ();
16     return 0;
17 }

```

Notion de sous-programme.

Exemple 2

Ecrire un programme qui affiche les n premières puissances de x

- Simple à écrire depuis le programme précédemment vu
- Deux paramètres à définir : n et x
- Aucune valeur de sortie : type de retour = **void**

Les n premières puissances de x .

```

1  #include <stdio.h>
2
3  void PuissNX (int n, int x) {
4      int i, puiss_x = 1;
5
6      printf ("calcul_des_%d_premieres_puissances_de_%d\n", n, x);
7      for (i=0; i < n; i++) {
8          printf ("%d^%d=%d\n", x, i, puiss_x);
9          puiss_x = puiss_x * x;
10     }
11 }

```

```
12
13 int main (int argc, char * argv[]) {
14     int n, x;
15
16     printf ("calcul_des_n_premieres_puissances_de_x\n");
17     printf ("Indiquer_la_valeur_de_n:_"); scanf ("%d", &n);
18     printf ("Indiquer_la_valeur_de_x:_"); scanf ("%d", &x);
19     PuissNX (n, x);
20     return 0;
21 }
```

Ce qu'il faut retenir des sous-programmes.

Les fonctions

- Au plus une information retournée
- Zéro ou plusieurs paramètres
- Syntaxe : type-de-retour NomDeLaFonction (Liste-des-paramètres)
- L'instruction **return** arrête la fonction et retourne un **unique** résultat

Le programme principal

- Type de retour : **int**. Par défaut, la valeur 0 est renvoyée
- Paramètres :
 - **int** argc : permet de connaître le nombre de paramètres de la fonction
 - **char * argv[]** : liste des paramètres de la fonction
- **int main (int argc, char * argv[]);**

8. PROGRAMMATION RÉCURSIVE

8.1. Principe.

Programmation récursive.

Définition : fonction récursive

Une fonction récursive est une fonction qui s'appelle elle-même

Avantages

- similarité avec les suites récursives en mathématiques ;
- "souvent" simple ou naturelle à programmer

Inconvénient

- En pratique "moins efficace" qu'une version itérative

Principe de programmation.

- Trouver le critère d'arrêt : en général, le rang 1 de la récursion. Revient souvent à gérer les cas particuliers "simples"
- Gérer l'appel récursif

8.2. En C.

Un exemple classique : $n!$.

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \dots \times n - 1 \times n \\ &= n * (n - 1)! \end{aligned}$$

Version itérative

```

1  int factoriel(int n){
2      int i;
3      int res=1;
4
5      for(i=1; i<=n; i++){
6          res *= i;
7      }
8
9      return res;
10 }
```

Version récursive

```

1  int factoriel(int n){
2      int res=1;
3
4      if(n==0)
5          return 1;
6      else
7          return n*factoriel(n-1);
8
9  }
```

9. LES POINTEURS

9.1. Sémantique et manipulation.

Qu'est ce qu'un pointeur ?

- Un pointeur est une **référence** sur une variable
- Chaque variable occupe un emplacement mémoire repérable par son **adresse**
- Un pointeur sur une variable correspond à l'adresse de cet emplacement

Exemple

Par exemple, si on suppose la mémoire initialement vide, alors les instructions suivantes mettent la mémoire dans l'état indiqué ci-après :

```

int x, y;
x = 10;
y = x * 2;
```

adresses :	@1	@2	@3	...
variables :	x	y		
mémoire :	10	20		

Intérêt des pointeurs.

- Les pointeurs permettent de récupérer l'adresse (dans la mémoire) d'une variable donnée
- Il devient alors possible d'accéder aux variables **x** et **y** directement via leurs adresses
- L'un des avantages majeur des pointeurs réside dans le fait de pouvoir manipuler directement des zones mémoires (tableaux, variables passées en paramètres, ...)

Quelques notations.

- Prendre l'adresse d'une variable : `&variable;`
- Accéder au contenu d'une zone pointée (dont on connaît l'adresse) : `*adresse;`
- Déclarer une variable comme un pointeur sur une zone mémoire d'un type donné : `int * variable;`

Une valeur particulière.

- Le pointeur `NULL` correspond à une adresse mémoire non valide
- Cette valeur permet, entre autre, d’initialiser les pointeurs
- Par exemple : `int * tab = NULL;`

Un type “pointeur générique”.

- Le type `void *` correspond à un type d’adresse générique
- Il est compatible avec tous les types de pointeurs

Un exemple complet.

```

1  #include <stdio.h>
2
3  int main (int argc, char * argv[]) {
4      int x;
5      int * y;
6      printf ("Saisir_un_entier:_"); scanf ("%d", &x);
7      printf ("L'entier_saisi_de_valeur_%d\n", x);
8      printf ("x_se_trouve_a_l'adresse_memoire_%d\n", (int)&x);
9      y = &x; /* y contient l'adresse memoire de x */
10     printf ("y_se_trouve_a_l'adresse_memoire_%x\n", (int)&y);
11     printf ("y_pointe_sur_l'adresse_memoire_%x\n", (int)y);
12
13     /* Le contenu de x peut maintenant etre manipule par x ou (*y) */
14     (*y) *= 10;
15     printf ("L'entier_saisi_a_ete_multiplie_par_10:_%d\n", x);
16     return 0;
17 }

```

9.2. Occupation mémoire.

Aspect “physique” des variables.

- Les différents types (`char`, `int` et `float`) ne permettent pas de stocker les mêmes informations
- Ceci est, entre autre, dû à l’espace mémoire occupé par les variables de ces types
- L’unité de base est l’**octet : 8 bits**
- Une variable de type `char` occupe 1 octet
- Une variable de type `int` ou `float` occupe **en général** 4 octets

Illustration.

Exemple

`char c1 = 'a'`; → 97^{eme} caractère ASCII : code binaire “01100001” `int i = 10`; → code binaire “00000000 00000000 00000000 00001010” `char c2 = '0'`; → 48^{eme} caractère ASCII : code binaire “00110000”

@10				@11			
@12				@13			
0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0
@14				@15			
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1
@16				@17			
@18				@19			

9.3. Arithmétique des pointeurs.

Pointeurs et opérations.

- Les pointeurs sont de “simples” variables
- Les opérateurs de base s’appliquent donc sur les pointeurs
- Il faut cependant manipuler avec beaucoup de précautions les opérateurs associés aux pointeurs.

Principaux intérêts.

Les pointeurs permettent de

- (1) Manipuler dynamiquement des tableaux
- (2) D’offrir aux fonctions l’accès direct en écriture à des variables “hors portée”

Accès direct en écriture à des variables

- Indispensable dès qu’une fonction doit renvoyer plus d’un résultat
- Par exemple : calcul des valeurs min et max sur une suite de valeurs
- Par souci de gain de temps, il est inutile de parcourir deux fois les mêmes valeurs pour déterminer les extremums.

Un exemple complet.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int FindMinMax (int * tab, int nb_val, int * min, int * max) {
5      int i = 0;
6      if (nb_val > 0) {
7          *min = *max = tab[0];
8          for (i = 1; i < nb_val ; i++) {
9              if (tab[i] > *max) { *max = tab[i]; }
10             if (tab[i] < *min) { *min = tab[i]; }
11         }
12         return 1;
13     } else { return 0; }
14 }
15
16 void ShowTab (int * tab, int nb_val) {
17     int i; for (i = 0; i < nb_val; i++) { printf ("%d_", tab[i]); } printf ("\n");
18 }
19
20 int main (int argc, char * argv[]) {
21     int i, nb_val = 0, min, max; int * tab = NULL;
22     printf ("Combien_de_valeurs_souhaitez_vous_manipuler_?_"); scanf ("%d", &nb_val);
23     tab = (int *) calloc (nb_val, sizeof(int));
24     for(i = 0; i < nb_val; i++) { tab[i] = rand(); }
25
26     ShowTab (tab, nb_val);
27     if (FindMinMax (tab, nb_val, &min, &max) == 1) {
28         printf ("Min=_%d_,_Max=_%d\n", min, max);
29     }
30     free (tab);
31     return 0;
32 }

```

Retour sur scanf.

- La fonction **scanf** utilisait déjà l’opérateur **&**
- Ceci permettait d’offrir un **accès direct** à chaque variable de stockage
- Indispensable pour pouvoir écrire dans les variables

Pièges à éviter.

- Les pointeurs permettent de manipuler (lecture, écriture) directement la mémoire !
- Les modifications sont irréversibles et peuvent donc être fatales.
- L’utilisation des pointeurs doit donc s’accompagner d’une grande vigilance lors de l’écriture des programmes
- Dans le langage C, les pointeurs sont “visibles”, c’est-à-dire que :
 - il faut les allouer et les désallouer à la main,
 - accéder à un objet pointé se fait en *déréférençant* le pointeur.

Ce qu’il faut retenir des pointeurs.

Les opérateurs

- **&** : renvoie l’adresse mémoire d’une variable
- ***** : renvoie le contenu d’une zone mémoire pointée

Les pointeurs

- offrent un accès direct à la mémoire
- permettent de manipuler dynamiquement des tableaux
- permettent aux fonctions d’accéder à des variables “hors portée” et donc de renvoyer plusieurs résultats

La manipulation des pointeurs peut s’avérer désastreuse... il convient donc d’être **très rigoureux** !

10. LES CHAÎNES DE CARACTÈRES

10.1. Utilisation. Principe d'une interaction utilisateur

- Les chaînes de caractères n'existent pas en C.
- Une chaîne de caractères = un tableau de caractères.
- Caractère spécial pour terminer les chaînes de caractères : `'\0'`
- La librairie `<string.h>` offre une liste de fonctions utiles pour manipuler les chaînes.
- Interaction avec un utilisateur :
 - Création d'une chaîne "buffer" de taille suffisante pour contenir le texte à saisir : `char buffer[256];`
 - Calcul de la taille exacte : `int strlen (buffer);`
 - Définir une chaîne de la bonne taille
 - Recopier le buffer dans la nouvelle chaîne

Les chaînes de caractères.

Exemple d'interaction utilisateur

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main (int argc, char * argv[]) {
6      char buffer[256], * prenom;
7      int taille;
8
9      printf ("Entrez_votre_prenom_:"); scanf ("%255s", buffer);
10     taille = strlen (buffer);
11     prenom = (char *) calloc (taille+1, sizeof(char));
12     strcpy (prenom, buffer);
13
14     printf ("Bonjour_%s\n", prenom);
15     free (prenom);
16     return 0;
17 }
```

10.2. Manipulation.

Fonctions utiles de `<string.h>`.

- La fonction `int strlen (const char * string)` : renvoie la longueur de la chaîne pointée par *string* sans compter le caractère `'\0'`
- La fonction `char * strcpy (char * dest, const char * src)` : copie la chaîne pointée par *src* (y compris le caractère `'\0'` final) dans la chaîne pointée par *dest*.
- La fonction `int strcmp (const char * s1, const char * s2)` : compare les deux chaînes *s1* et *s2*. Elle renvoie un entier :
 - négatif si $s1 < s2$
 - nul si $s1 = s2$
 - ou positif si $s1 > s2$
- La fonction `char * strcat (char * dest, const char * src)` ajoute la chaîne *src* à la fin de la chaîne *dest* en écrasant le caractère `'\0'` à la fin de *dest*, puis en ajoutant un nouveau caractère `'\0'` final.
- pour plus de détails : **man string**

Ce qu'il faut retenir des chaînes de caractères en C.

Déclaration et manipulation

- Déclaration : `char * chaine;`
- Taille : nombre de caractères "utiles" + le caractère `'\0'`
- Allocation : `chaine = (char *) calloc (1+taille, sizeof(char));`
- Libération : `free(chaine);`

Fonctions utiles dans <string.h>

- **strlen**(chaîne) : permet de connaître le nombre de caractères “utiles” de chaîne
- **strcpy**(dest, src) : permet de copier la chaîne *src* dans *dest*
- **strcmp**(s1, s2) : permet de comparer les chaînes s1 et s2
- ...

11. LES FICHIERS

11.1. Manipulation.

La manipulation des fichiers.

Intérêt

- Persistance de l’information
- Possibilité d’initialisation complexe
- Reproductibilité d’une séquence d’actions
- ...

Principe

- Création d’un fichier sur le disque
- Association d’un **flux** à un nom de fichier physique
- Manipulation du flux en mémoire
- Lors de la fermeture du flux : écriture sur le disque

Fonctions de base.

- Utilisation de <stdio.h>
- **FILE *** : type des flux
- **FILE *fopen** (**const char *** path, **const char *** mode) ;
 - Permet l’association d’un flux à un nom de fichier
 - Le fichier associé est ouvert en fonction du mode (lecture, écriture, lecture+écriture, ...)
 - Le flux peut être manipulé
 - En cas d’erreur, **fopen** renvoie **NULL**
- **int fclose** (**FILE *** fp) ;
 - Fermeture du flux
 - Écriture sur le disque du fichier associé (le cas échéant)
 - Renvoie 0 si aucun problème, sinon retour de **EOF** (End Of File)

La fonction fopen.

FILE *fopen (**const char *** path, **const char *** mode) ;

- mode = “r” : Ouvre le fichier en lecture. Le pointeur de flux est placé au début du fichier.
- mode = “r+” : Ouvre le fichier en lecture et écriture. Le pointeur de flux est placé au début du fichier.
- mode = “w” : Ouvre le fichier en écriture. Le fichier est créé s’il n’existait pas. S’il existait déjà, sa longueur est ramenée à 0. Le pointeur de flux est placé au début du fichier.
- mode = “w+” : Ouvre le fichier en lecture et écriture. Le fichier est créé s’il n’existait pas. S’il existait déjà, sa longueur est ramenée à 0. Le pointeur de flux est placé au début du fichier.
- mode = “a” : Ouvre le fichier en ajout (écriture à la fin du fichier). Le fichier est créé s’il n’existait pas. Le pointeur de flux est placé à la fin du fichier.
- mode = “a+” : Ouvre le fichier en lecture et ajout (écriture en fin de fichier). Le fichier est créé s’il n’existait pas. La tête de lecture initiale du fichier est placée au début du fichier mais la sortie est toujours ajoutée à la fin du fichier.

11.2. Lecture/écriture.

Lecture et écriture.

Fonctions définies dans <stdio.h>

Lecture : int fscanf (**FILE *** stream, **const char *** format, ...) ;

- Cette fonction est comparable à **scanf**
- Elle renvoie le nombre d’éléments d’entrées correctement mis en correspondance et assignés

- exemple : `fscanf` (fp, “%d %d”, &i, &j);

Écriture : `int fprintf (FILE * stream, const char * format, ...);`

- Cette fonction est comparable à `printf`
- Elle renvoie le nombre de caractères imprimés, sans compter l’octet nul ‘\0’ final dans les chaînes.
- exemple : `fprintf` (fp, “%d %d\n”, i, j);

12. TYPES ÉNUMÉRÉS ET TYPES STRUCTURÉS

12.1. Besoin de nouveaux types.

Limites des types de base.

Limites des types de base

- Les types de base ne permettent pas de tout représenter
- Exemple : un étudiant
 - Nom
 - Prénom
 - Homme/Femme
 - Age
 - Filière
 - Année
 - Notes
 - ...
- Chaque information isolée est représentable avec un type de base
- L’information complète n’est pas représentable

12.2. Type énuméré.

Types énumérés.

- Besoins : Manipuler un ensemble fini de valeurs
- Définition d’un ensemble de constantes de type `int`
- Les valeurs sont ordonnées : possibilité de les comparer
- Par exemple :
 - `plume` ↔ 0
 - `papier` ↔ 1
 - `plomb` ↔ 2

```
typedef enum { plume, papier, plomb } t_poids ;
```

Nous avons, par construction `plume < papier < plomb` ce qui nous permet de représenter un ordre sur les poids des matériaux listés.

12.3. Type Structuré.

Définition.

- Besoins : Manipuler des variables structurées (un étudiant par exemple)
- Définition d’un nouveau type regroupant différentes informations

```
typedef struct s_etudiant { int age; char * nom; char * prenom; int
hommeFemme; } t_etudiant;
```

Manipulation.

- La notation **pointée** permet d’accéder aux différents champs d’une variable de type structuré
- Par exemple,


```
t_etudiant un_etudiant; struct s_etudiant un_autre; un_etudiant.age
= 20; un_autre.hommeFemme = 0;
```

Initialisation.

Il existe trois manières d'initialiser une variable structurée

- (1) En initialisation les champs un par un
- (2) En initialisation, **à la déclaration de la variable uniquement, toute la structure en une opération** :
`un_etudiant = {20, NULL, NULL, 0}`. Ceci implique de connaître l'ordre exact des champs.
- (3) En initialisant, **à la déclaration de la variable uniquement et en une seule opération, seulement les champs qui nous intéressent** : `un_etudiant = {.age=20, .hommeFemme=0}`

Types structurés : un exemple.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5      int age;
6      char * nom;
7      char * prenom;
8      int hommeFemme;
9  } t_etudiant;
10
11 int main (int argc, char * argv[]) {
12     t_etudiant e1;
13     t_etudiant e2 = {25, NULL, NULL, 0};
14     t_etudiant e3 = {.age = 30};
15
16     e1.age = 20;
17     printf ("age_(e1) = %d\n", e1.age);
18     printf ("age_(e2) = %d\n", e2.age);
19     printf ("age_(e3) = %d\n", e3.age);
20     return 0;
21 }
```

Ce qu'il faut retenir des types construits.

Convention de nommage

Il est préférable de clairement différencier les types construits des types existants. Par exemple, en préfixant les types construits par `t_`.

Types énumérés

- Syntaxe : `typedef enum {val1, val2, ..., valN} t_type;`
- Ordre total défini entre les valeurs lors de la conception du type
- Utilisation : `t_type var = val2;`

Types structurés

- Syntaxe : `typedef struct s_type {type1 champ1; ...} t_type;`
- Initialisation : en bloc à la déclaration ou champ par champ ensuite.
- Manipulation : notation pointée (`var.champ1`)

13. QUELQUES TYPES COMPLEXES

13.1. Les listes chaînées.

Les listes chaînées.

Introduction

- Les listes chaînées sont des structures de données permettant de stocker un nombre quelconque d'éléments (potentiellement infini)
- Elles peuvent être représentées dans des tableaux ou à l'aide de structures récursives (avec des pointeurs).

- Nous développerons dans ce cours les structures avec pointeurs. Cette représentation est fortement recommandée lorsque le nombre d'éléments à stocker n'est pas connu à l'avance.

Les listes chaînées.

Intérêt

- Gestion d'une file d'attente (imprimante, guichets, ...)
- Gestion d'un ensemble d'évènements avec priorités différentes (processeurs, urgences, ordres bancaires, ...)
- Gestion d'objets dans un jeu vidéo (monstres, armes, joueur, ...)
- ...

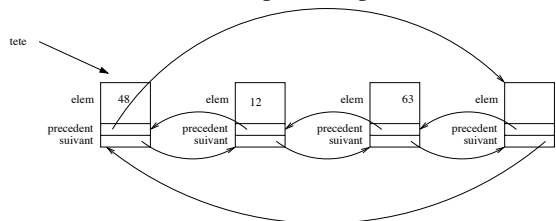
Les listes chaînées.

Principe

Il existe plusieurs formes de listes chaînées :

- Chaînage simple : avant ou arrière
- Chaînage double : avant et arrière
- Gestion circulaire de la liste

Ces trois formes peuvent être combinées et fournir par exemple, des listes circulaires doublement chaînées.



Les listes chaînées.

Structures de données

La structure de données associée à cette représentation est la suivante :

```
typedef struct _t_maillon {
    int valeur ;
    struct _t_maillon * suivant, * precedent ;
} t_maillon ;
```

La liste est alors représentée par un unique élément appelé **la tête**.

Les listes chaînées.

Opérations

Toutes les opérations suivantes doivent pouvoir être effectuées sur des listes chaînées. Cet ensemble d'opérations n'est pas exhaustif.

- liste_vide : $\emptyset \rightarrow$ liste
- est_vide : liste \rightarrow boolean
- est_present : liste \times element \rightarrow boolean
- supprimer : liste \times element \rightarrow liste (supprime le premier élément égal à element de la liste)
- ajouter_tete : liste \times element \rightarrow liste (ajoute element en tête de liste)