Eléments de Langage C

Gilles Renversez[a]

Faculté des Sciences de St Jérôme
Université Paul Cézanne Aix-Marseille III
année universitaire 2009-2010

[a]e-mail : gilles.renversez@fresnel.fr et internet : www.fresnel.fr/perso/renversez

Plan du cours

1 Introduction

1.1 Objectifs de l'enseignement

apprentissage et pratique :

- d'un système d'exploitation : Linux (système de type UNIX)
- d'un minimum d'algorithmique
- d'un langage de programmation structurée : le C
- de la résolution de problèmes numériques de base avec comme discipline d'illustration la Physique : recherche de zéros, calcul matriciel, équations différentiels, marches au hasard...

Cela nécessite un investissement personnel lors de séances (VOUS tapez et corrigez vos programmes) et en dehors des séances (VOUS apprenez les syntaxes et les algorithmes vus en cours).

2 Préliminaires

2.1 Les systèmes d'exploitation

définition : Un système d'exploitation est le logiciel qui gère l'ordinateur (la machine plus généralement) : la lecture/écriture sur le disques, le processeur, les périphériques, ...

utilisation : Il permet notamment à un utilisateur de contrôler l'ordinateur

type : Il y a actuellement de deux grandes familles de systèmes d'exploitation : UNIX (ex : linux, True64Unix) et Windows (ex : XP)

2.2 Le système de fichiers

– Un fichier est tout simplement une suite d'octets (une suite donnée de cases mémoire ayant certaines valeurs).
– Le système d'exploitation ne fait aucune interprétation du contenu d'un fichier, seul le programme qui produit ou exploite le fichier l'interpréte.

On distingue 3 sortes de fichiers :

1. **Un répertoire** ou catalogue (ou en anglais directory) est un fichier dont le contenu est une liste de noms de fichiers et leurs caractéristiques. Dans cette liste, il peut y avoir des répertoires...
2. Un fichier spécial est fichier servant pour la gestion des en-

trées/sorties (peu ou pas utiliser lors de cet enseignement).

3. **Un fichier ordinaire** est un fichier qui n'est ni un répertoire ni un fichier spécial.

2.3 Les noms de fichiers

Les régles de constructions :

Les règles de constructions des noms de fichiers sont les suivantes pour les s systèmes de type UNIX :

1. La manière de nommer les fichiers dans tous les systèmes de type UNIX utilise **la structure arborescente des répertoires**.
2. Tout nom de fichier figure dans un répertoire qui figure à son tour dans un autre répertoire, et ainsi de suite jusqu'au répertoire **racine**.
3. Ce répertoire est le premier répertoire du système de fichier, on le note **/**.
4. On fait référence à un fichier particulier en écrivant tous les noms de répertoires entre lui et le répertoire racine, séparé par des '/'.

Exemple :

`/var/home/l3p58/TD1/exo1-TD1.c`

On dit alors que `/var/home/l3p58/TD1/` est le **chemin absolu** qui mène au fichier `exo1-TD1.c`

Le répertoire courant :

Pour alléger les références aux fichiers, un répertoire particulier est distingué comme **répertoire de travail** ou **répertoire courant**.

On parle alors de **chemin relatif**.

Exemple :
Si le répertoire courant est `/var/home/`
alors `l3p58/TD1/exo1-TD1.c` est une référence relative au fichier `/var/home/l3p58/TD1/exo1-TD1.c`,
le chemin relatif étant `l3p58/TD1/`.

remarques :
-le caractère '**.**' constitue une référence relative au répertoire de travail (un synonyme, un lien en quelque sorte).
-les caractères '**..**' constituent une référence relative au répertoire ancêtre ou père du répertoire courant.

2.4 La ligne de commande ou 'shell'

C'est par cette interface que l'on peut lancer des commandes

sur les fichiers de l'ordinateur.

Voici les principales commandes :

`pwd` : indique où on se trouve dans le système de fichiers, *i.e.* le répertoire courant.

`ls` : liste l'ensemble des fichiers (y compris ceux de type répertoire) présents dans le répertoire courant.

`mkdir nom-du-répertoire` : crée le répertoire `nom-du-répertoire`.

`cd nom-du-répertoire` : se place dans le répertoire `nom-du-répertoire`.

`rmdir nom-du-répertoire` : détruit le répertoire `nom-du-répertoire`.

`touch nom-du-fichier` : crée le fichier `nom-du-fichier` s'il n'existe pas.

`rm nom-du-fichier` : détruit le fichier

`cp nom-du-fichier-1 nom-du-fichier-2` : copie le fichier `nom-du-fichier-1` sur le fichier `nom-du-fichier-2`.

`mv nom-du-fichier-1 nom-du-fichier-2` : change le nom le fichier `nom-du-fic` en `nom-du-fichier-2`.

Exemple :

```
> pwd
/var/home/l3p58
> ls
TD1  a-lire.txt  bonjour.c
>mkdir TD2
> ls
TD1  TD2  a-lire.txt  bonjour.c
> touch exo1.c
> ls
TD1  TD2  a-lire.txt  bonjour.c exo1.c
```

Il y a beaucoup d'autres commandes accessibles que l'on verra

ultérieurement.

3 Les bases de la programmation en C

3.1 Historique

Le langage C a été crée en 1972 par Dennis Richie et Ken Thompson. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C. En 1989, le langage C est normalisé par 'American National Standards Institute' devant le C ANSI.

3.2 La compilation

Le C est un langage **compilé** (par opposition aux langages **interprétés**).

Cela signifie qu'un programme C est décrit par un fichier texte, appelé **fichier source**.

Pour pouvoir le faire exécuter par le processeur, il faut le traduire dans le jeux des instructions de ce dernier. Cette traduction est nommée **compilation**, elle est faite par le **compilateur**.

La compilation se déroule en 4 phases :

1. **Le traitement par le préprocesseur :** Le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles.
2. **Le compilation :** Le résultat de l'étape ci-dessus est tra-

duit en assembleur.

3. **L'assemblage :** Cette opération transforme le résultat de l'étape précédante en fichier binaire. Par défaut, la compilation et l'assemblage se font dans la foulée. On obtient le **fichier objet**
4. **L'édition de lien :** On obtient le **fichier exécutable**

Schématiquement, on a successivement :
mon-programme.c → mon-programme.o → mon-programme.out

3.3 Le compilateur `gcc`

Nous allons utiliser le compilateur **`gcc`** qui est le compilateur GNU. Il est disponible gratuitement avec ses sources et sa documentation, c'est un logiciel libre de la Free Software Foundation[a].

La syntaxe d'appel est la suivante :

`gcc` [`-o nom-du-fichier-resultat-compilateur`] [*autres-options*] `nom-du-fichier-source` [`-l`*librairies*]

Dans une syntaxe de ce type que l'on rencontre souvent dans les très nombreuses documentations disponibles, les crochets encadrent des arguments optionels.

[a] http///www.fsf.org

Voici la signification de quelques unes de ces options, nous en verrons d'autres ultérieurement.

`-E` : n'active que le préprocesseur (étape 1).

`-S` : n'active que le préprocesseur et la compilation. Le compilateur produit seulement le fichier assembleur (étapes 1 et 2).

`-c` : n'effectue pas l'édition de lien. Le compilateur produit seulement le fichier objet (étapes 1 à 3).

`-lm` : charge la librairie mathématique `libm.a`

Exemple :

```
> gcc -o bonjour.i -E bonjour.c
> gcc -o fonction.o -c fonction.c
> gcc -o bonjour.out bonjour.c
> gcc -o calcul.out calcul.c -lm
```

3.4 Connexion sur un compte et premières manipulations avec la ligne de commande

3.5 Un éditeur de fichiers surpuissant : emacs

```
#include <stdio.h> /* ligne s'adressant au préprocesseur */
/* car commençant par # */
/* Elle commande le chargement de la librairie 'stdio.h'*/
/* qui gère les entrées/sorties */

int main()  /* déclaration de la fonction principale 'main' */
            /* elle est de type 'int' c'est-à-dire entier   */
{
  printf("bonjour\n"); /* écrit ``bonjour'' avec un saut  */
/* de ligne représenté par la séquence d'échappement '\n'  */
  return(0); /* la fonction 'main' prendra la valeur 0 */
             /* en fin de programme */
}
```

Dans la ligne de commande (sur vos machines il s'agit en fait d'un **terminal**), tapez successivement les commandes suivantes :

```
> gcc -o bonjour.o -c bonjour.c
> ls
> gcc -o bonjour.out bonjour.c
> ls
> ./bonjour.out
```

3.6 Les composants élémentaires du langage C

1. **Les identificateurs :** Le rôle d'un identificateur est de donner un nom à une entité du programme que cela soit une variable ou une fonction.

   Un identificateur est une suite de caractères dans les ensemble suivants :
   - les lettres majuscules ou minuscules mais non accentuées
   - les chiffres
   - le "blanc souligné" ou "underscore" en anglais

   Un identificateur ne peut pas commencer par un chiffre.
2. **Les mots-clefs :** Les mots suivants, appelés **mots-clefs** sont réservés pour le langage C et ne peuvent pas être pris comme identificateurs.

   Les spécificateurs de type : `char double enum float int long short signed struct union unsig void`

   Les spécificateurs de stockage : `auto register static extern typedef`

   Les qualificateurs de type : `const volatile`

   Les instructions de contrôle : `break case continue default do else for goto if switch while`

autres catégories : `return sizeof`

3. **Les commentaires :** Ils commencent par `/*` et s'achève par `*/`. Il n'est pas permis de les imbriquer.

3.7 Structure d'un programme C

La structure d'un programme C est la suivante :

[directives aux préprocesseurs]
[déclarations des variables externes]
[fonctions]

```
int main()
{
```

déclarations des variables internes
instructions

```
}
```

4 Les types prédéfinis

Le C est un langage *typé* :
Toute variable, constante, ou fonction est d'un type précis.
Le type d'un objet définit la façon dont il est représenté en mémoire.

Quand une variable est définie, il lui est attribué une adresse, l'étendue de la zone mémoire occupée par la variable est fixée par son type.

le type caractère : le mot-clé `char` désigne un objet de type charactère. Une variable de type `char` peut contenir n'importe quel élément du jeu de caractères de la machine utilisé.
La plupart du temps un `char` est codé sur un seul octet.
Une des particularités du type `char` en C est qu'il peut être assimilé à un entier

le type entier : le mot clé désignant le type entier est `int`.
Un objet de type `int` est représenté par un emplacement de taille "naturel" par rapport à la machine utilisée, 32 bits pour un PC.
Le type `int` peut-être précédé d'un attribut de précision, `short` ou `long`, et ou d'un attribut de représentation `signed` ou `unsigned`.
Les valeurs maximales et minimales des différents types d'en-

tiers sont définies dans la librairie `limits.h`.

mots-clés	taille sur PC 32 bits	taille sur PC 64 bits	description
char	8 bits	8 bits	caractère
short int	16 bits	16 bits	entier court
int	32 bits	32 bits	entier
long int	32 bits	64 bits	entier long

TAB. 1: Les types entiers

les types flottants : Les types `float`, `double` et `long double` servent à représenter des nombre en virgule flottantes. Ils correspondent aux différentes précisions possibles.

mots-clés	taille sur PC 32 bits	taille sur PC 64 bits	des
float	32 bits	32 bits	flottant si
double	64 bits	64 bits	flottant do
long double	96 bits	128 bits	flottant qua

TAB. 2: Les types flottants (réels)

```
#include <stdio.h>
int main()
{
```

```
  /*  sizeof() est une fonction qui retourne le nombre d'octet
*/
  /* %u est le format d'affichage d'un entier  non-signé, unsi
  char carac,carac1;


  printf("la taille mémoire occupée par un caractère:\n");
  printf("\t char : %du\n",sizeof(char));
  printf("\t pour la variable carac : %u\n",sizeof(carac));

  printf("\t short int : %u\n",sizeof(short int));
  printf("\t int : %u\n",sizeof(int));
  printf("la taille mémoire occupée par les entiers de type:\n
  printf("\t long int : %u\n",sizeof(long int));
  printf("la taille mémoire occupée par les réels de type:\n")
  printf("\t float : %u\n",sizeof(float));
  printf("\t double : %u\n",sizeof(double));
  printf("\t long double : %u\n",sizeof(long double));
  return (0);
}
```

remarque : Dans le C ANSI, le type de `sizeof` peut changer d'un système à un autre. Pour cette raison, il est définie sous l'appellation `size_t` quand cela est requis.

les types énumérés : Un type énuméré, ou énumération, est constitué par une famille finie de nombres entiers, chacun associé à un identificateur qui en est le nom.

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

```
int main()
{
/*    illustration du type enuméré  */
  enum {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimar
   printf(" on représente \'mercredi\' par %d\n",mercredi);
   printf(" ce qui est équivalent à \'mardi+1\': %d\n",mardi+1
   return(0);
}
```

5 Les objets structurés

5.1 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses successives.

La déclaration d'un tableau unidimensionnel s'effectue ainsi :

```
type nom_du_tableau[nbre_elements];
```

où `nbre_elements` est une expression constante entière positive.

On accède aux éléments du tableau en lui appliquant l'opérateur []. Les éléments sont numérotés de 0 à `nbre_elements` -1.

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Cela implique qu'aucune affectation globale n'est possible sur un tableau : `tab1 = tab2` n'est pas autorisé.

Il faut effectuer l'affectation élément par élément.

Exemple de tableau 1d :

```
#include <stdio.h>
#include <math.h>
int main()
```

```
{
  int tab[] ={1,2,3,4,5,6,7};
  /* déclaration avec initialisation de tab */
  /* et équivalent aux lignes ci-dessous    */
  /*    int tab[7]; */
  /*    tab[0]=1;   */
  /*    ....        */
  /*    tab[6]=7;   */
  int nb_elts;
  printf("taille globale du tableau:%u\n", sizeof(tab));
  printf("taille d\'un élement du tableau: %u\n", sizeof(tab[0]));
  nb_elts=sizeof(tab)/sizeof(tab[3]);
  printf("nombre total d\'éléments du tableau:%d \n",nb_elts);
  return (0);
}
/* %u est le format d'affichage des entiers non-signés
```

Le langace C ne prévoit que les tableaux à un seul indice mais les éléments de tableau peuvent être à leur tour des tableaux. La syntaxe pour un tableau multidimensionnel de `n` dimension est :

```
type nom_du_tableau[dimension_1][dimension_2]... [dimension_n];
```

où `nbre_elements` est une expression constante entière positive. Exemple de tableau 2d :

```
#include <stdio.h>
#define NB_LIGNES  2 /* instruction au pré-processeur */
#define NB_COLONNES 4
int main()
```

```
{
  int matrice[NB_LIGNES][NB_COLONNES] ={{11,21,31,41},{55,56,57,
  int taille,i,j;
  printf("taille mémoire globale de la matrice: %u\n", sizeof(ma
  printf("taille mémoire d\'un élement de la matrice: %u\n", siz
  printf("taille mémoire d\'une ligne de la matrice: %u\n", size
  taille=sizeof(matrice)/sizeof(matrice[0][0]);
  printf("nombre total d\'éléments de la matrice: %d\n",taille);
  for (i=0; i< NB_LIGNES; i++)
    {
      for (j=0; j< NB_COLONNES ; j++)
        {
          printf("matrice[%d][%d]=%d\t",i,j,matrice[i][j]);
        }
      printf("\n");
    }
   for (i=0; i< NB_LIGNES; i++)
    {
      for (j=0; j< NB_COLONNES; j++)
        {
          matrice[i][j]=0;
        }
    }
  matrice[0][0]=1;
  matrice[1][0]=1;
  return (0);
}
```

5.2 Les structures

les structures sont des variables composées de champ (ou membre) de types égaux ou différents. On distingue la déclaration d'un *modèle de structure* de celle d'un objet utilisant un tel modèle. Dans le premier cas on utilise la syntaxe suivante :

```
struct nom_du_type_de_structure
{
type_1 champ_1;
type_2 champ_2;
...
type_n champ_n;
};
```

La syntaxe pour déclarer une variable de type `nom_du_type_de_` est :

```
struct nom_du_type_de_structure nom_de_variable;
```

On accède aux différents champs d'une structure en utilisant l'opérateur *membre de structure* noté ".".
Le `j`-ème champ de `nom_de_variable` est accessible par : `nom_de_variable.champ_j`.

```
/* Exemple d'une structure */
# include <stdio.h>
int main(void)
{
  struct date_chiffre
  {
    int jours;
    int mois;
```

```
    int annee;
  };
  struct date_mixte
  {
    int jours;
    char mois[11];
    int annee;
  };
  struct date_chiffre aujourdhui;
  struct date_mixte demain;
  struct date_mixte hier={17,"octobre",2005};
  aujourdhui.jours=18;
  aujourdhui.mois=10;
  aujourdhui.annee=2005;
  demain.jours=19;
  demain.mois[0]='o';
  demain.mois[1]='c';
  demain.mois[2]='t';
  demain.mois[3]='o';
  demain.mois[4]='b';
  demain.mois[5]='r';
  demain.mois[6]='e';
  demain.mois[7]='\0';
  demain.annee=2005;

  printf("La date d\'aujourd\'hui est: %d/%d/%d\n",aujourdhui);
  printf("La date de demain est: %d %s %d\n",demain.jours,demain
  printf("La date d\'hier est: %d %s %d\n",hier.jours,hier.mois,
  return(0);
}
```

On peut définir des tableaux de structures :

```
/* Exemple d'un tableau de structures */
# include <stdio.h>

int main(void)
{
  struct date_mixte
  {
    int jours;
    char mois[12];
    int annee;
  };

  struct date_mixte vacances[100]={{1,"avril",1998},
{1,"septembre",1998},{2,"septembre",1998}};

  printf("La début des vacances fût le : %d %s %d\n",
vacances[0].jours,vacances[0].mois,vacances[0].annee);
  printf("La fin des vacances fût le : %d %s %d\n",
vacances[2].jours,vacances[2].mois,vacances[2].annee);
  return(0);
}
```

On peut transmettre la valeur d'une structure à une fonction :

```
# include <stdio.h>
struct enreg
{
  int a;
```

```
  float b;
};
void fct(struct enreg);
int main(void)
{
  struct enreg x;
  x.a=2;
  x.b=12.5;
  printf("\navant l\'appel de fct: %d %e",x.a,x.b);
  fct(x);
  printf("\naprès l\'appel de fct: %d %e\n",x.a,x.b);
  return(0);
}
  void fct(struct enreg s)
{
  s.a =0;
  s.b=2.0;
  printf("\n dans fct: %d %e",s.a,s.b);
  return;
}
```

On peut transmettre l'adresse d'une structure à une fonction. Si on veut faire cela il faudra utiliser dans l'appel l'opérateur d'adressage &. L'entête de la fonction `fct` précédente sera modifiée en :

```
void fct(struct enreg *adresse_enreg);
```

Le problème qui se pose est celui d'accéder, au sein de la dé-

finition de `fct` à chacun des champs de la structure d'adresse `adresse_enreg`. L'opérateur . ne convient plus car il suppose comme premier opérande un nom de structure et non une adresse. On a alors deux solutions :

1. adopter une notation telle que (*`adresse_enreg`).`champ` pour désigner le champ de la structure d'adresse `adresse_en`
2. faire appel à un nouvel opérateur noté "->", lequel permet d'accéder aux différents champs d'une structure à partir de son adresse de début. Dans la fonction `fct`, on écrira `adresse_enreg->champ` pour accéder au champ.

```
# include <stdio.h>
struct enreg
{
  int a;
  float b;
};
void fct(struct enreg *);
int main(void)
{
  struct enreg x;

  x.a=2;
  x.b=12.5;
  printf("\navant l\'appel de fct: %d %e",x.a,x.b);
  fct(&x);
  printf("\naprès l\'appel de fct: %d %e\n",x.a,x.b);
```

```
  return(0);
}
  void fct(struct enreg *adresse_enreg)
{
  adresse_enreg->a =0;
  adresse_enreg->b =2.0;
  printf("\n dans fct: %d %e",adresse_enreg->a,
  adresse_enreg->b);
/*    autre syntaxe possible en utilisant l'operateur d'indirect
*/
  printf("\n dans fct (autre syntaxe): %d %e",
  (*adresse_enreg).a,(*adresse_enreg).b);
  return;
}
```

5.3 les unions

Une `union` désigne un ensemble de variables de types différents susceptibles d'occuper *alternativement* une même zone mémoire.

Une `union` permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types.

Les déclarations et les opérations sur les variables de type `union` sont les mêmes que celles sur les variables de type `struct`.

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

```
{
  char lettre;
  int numero;
};

int main(void)
{
  union jour hier,demain;
  hier.lettre='J';
  printf("hier = %c\n",hier.lettre);
  hier.numero=4;
  printf("hier = %c\n",hier.lettre);
  demain.numero=hier.numero+2;
  printf("demain = %d\n",demain.numero);
  return(0);
}
```

Remarques : les unions permettent de considérer un même objet comme possédant plusieurs types : elles semblent faire double emploi avec l'opérateur de changement de type. Ce n'est pas le cas car :

- L'opérateur de changement de type ne supporte pas les structures.
- lorsqu'on change de type d'un objet au moyen de l'opérateur de changement de type, le langage C convertit la donnée, afin que l'expression ait sous le nouveau type, une valeur utile. Au contraire, lorsqu'on fait référence à un champ

d'une union, la donnée ne subit aucune transformation, même si elle a été affectée en faisant référence à un autre champ.

5.4 le déclarateur de synonyme

Le mot clef `typedef` permet de donner un nom à un type dans le but d'alléger les futures expressions où il figure.

```
typedef type synomine;
```

```
#include <stdio.h>

int main(void)
{
  struct nombre_complexe
  {
    double reelle;
    double imaginaire;
  };
  typedef struct nombre_complexe complexe;
  complexe z;
  z.reelle=1.0;
  z.imaginaire=2.0;

  return(0);
}
```

6 Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source, le type de la constante étant déterminé par la manière dont elle est écrite.

– Les constantes entières

  Une constante peut être écrite de 3 manières différentes suivant la base dans laquelle est représentée :

  – décimale
  – octale
  – héxadécimale

  Par défaut une constante décimale est représentée avec le format interne le plus court permettant de la représenter parmi les formats des types `int`,`long int`, `unsigned long int`. On peut spécifier explicitement le format d'un constante entière en la suffixant par `u` ou `U` pour indiquer qu'elle est non signée, ou en la suffixant par `l` ou `L` pour indiquer qu'elle est de type `long`.

– Les constantes réelles

  Les constantes réelles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre `e` ou `E`. Il s'agit d'un nombre décimale éventuellement signé.

Par défaut, une constante réelle est représentée avec le format du type double. On peut influer sur la représentation avec les suffixes `f` ou `F` pour forcer la représentation en `float`, les suffixes `l` ou `L` forcent la représentation en `long double`.

– Les constantes caractères Une constante de type caractère se note en écrivant le caractère entre apostrophes. On représente l'antislash par \\ et l'apostrophe par \', le point d'interrogation par \ ? et les guillemets par \".
Les caractères non-imprimables les plus utilisés sont accessibles par les *séquences d'échappement* suivantes :

\n saut de ligne	\r retour chariot
\t tabulation horizontale	\f saut de page
\v tabulation verticale	\a signal d'alerte
\b retour arrière	

TAB. 3: Les séquences d'échappement

– Les constantes chaînes de caractères
Une chaîne de caractères est une suite de caractères entourés par des guillemets

```
chaine_caract="Ceci devrait être une chaîne de caractères."
```

7 Les opérateurs

7.1 Les opérateurs à effet de bord

lCes opérateurs modifient les valeurs prises par les variables

1. Les opérateurs d'affectation
   - l'affectation simple :

     Sa syntaxe est `variable = expression`

     Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante.

     On nomme **g-valeur**, **l-value** en anglais, tout élément pouvant figurer à gauche d'une affectation.

     `expression` est évaluée avant que sa valeur ne soit affectée à `variable`.

     L'affectation effectue une conversion *implicite* la valeur de `expression` est convertie dans le type du terme de gauche.
   - les affectations combinées :

+=	-=	*=	/=	%=

TAB. 4: opérateurs d'affectation combinée

L'expession de la forme `exp_1 operateur= exp_2`

est équivalente à :

```
exp_1 = exp_1 operateur exp_2
```

– Les opérateurs d'incrémentation et de décrémentation
Ces opérateurs d'incrémentation ++ et de décrémentation −− sont des opérateurs unaires permttant respectivement d'ajouter ou de retrancher 1 au contenu de leur opérande.
Ces opérateurs ne s'appliquent qu'à des g-valeurs.
Remarque :
  – Ces opérations sont effectuées avant l'évaluation de l'expression si l'opérateur la précède, après l'évaluation s'il la suit.

2. les opérateurs arithmétiques :
Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires :

+	-	*	/	%

TAB. 5: opérateurs arithmétiques classiques

Remarques :
– Le langage C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flot-

tants. Si les deux opérandes sont de type entier, l'opérateur / produira une division entière. Si une des opérandes est un flottant, la valeur résultante sera de type flottant.

– l'opérateur n'agit que sur des opérandes de type entier

7.2 les opérateurs d'adressage et d'indirection

8 Les structures de contrôle

8.1 Les blocs d'instructions

Toute instruction simple est terminée par un point virgule **;**.

Un bloc d'instructions est une suite de déclarations et d'instructions encadrées par les deux accolades **{** et **}**. Du point de vue de la syntaxe un bloc se comporte comme un instruction unique.

Un bloc peut se réduire à une seule instruction.

```
{i=3;}
```

8.2 Les structures de contrôle conditionnel

1. Instruction de test `if ... else`
   (a) `if (expression) instruction_1`
   (b) `if (expression) instruction_1 else instruction_2`

   Un `else` se rapporte toujours au dernier `if` rencontré auquel un `else` n'a pas encore été attribué.
2. Branchement multiple `switch`
   Sa syntaxe est la suivante :
   `switch (expression)`

```
{
  case cte_1:
    liste_instructions_1
    break;
  case cte_2:
    liste_instructions_2
    break;
  ...
  case cte_n:
    liste_instructions_n
    break;
  default:
    liste_instructions_defaut
    break;
}
```

Si la valeur de `expression` est égale à l'une des constantes `cte_m` la `liste_instructions` correspondante est exécutée. Sinon, `liste_instructions_defaut` est exécutée.

8.3 Les boucles

1. boucle `for`

   (a) `for(expr_1; expr_2; expr_3)`

(b) expr_1 est une expression d'initialisation nécessaire avant l'entrée dans la boucle

(c) expr_2 est le test de continuation de la boucle, il est évalué avant le corps de boucle

(d) expr_3 est souvent une incrémentation évaluée à la fin du corps de la boucle

Les expressions expr_1 et expr_3 peuvent être absentes.

2. boucle `while`

```
while (expression)
    bloc_instructions
```

Tant que `expression` est non nulle, `bloc_instructions` est exécuté. Si `expression` est nulle dès le début, `bloc_instructions` n'est jamais exécuté.

3. boucle `do ... while`

Cette boucle permet d'effectuer le test de poursuite d'exécution après une exécution du bloc d'instructions.

```
do
    bloc_instructions
while (expression);
```

8.4 Les structures de contrôle non conditionnel

– Branchement non conditionnel `break`
  L'instruction `break` permet d'interrompre le déroulement d'une boucle (ou d'une instruction de branchement d'un `switch` et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne.

```
#include <stdio.h> /* inclusion de la librairie stdio.h , inst
int main() /* Entête de la fonction principale main */
{
  int i; /* déclaration de l'entier i */
  for (i=0; i < 6; i++ )
    {
      if (i == 3) /* operateur de test == et pas celui d affec
        {
          break; /* le break interrompt le déroulement de la b
        }
      printf("i= %d\n",i);
    }
  printf("La valeur de i en sortie de boucle = %d\n",i);
  return(0);/*  Pour que la fonction main prenne la valeur 0 *
}
```

– Branchement non conditionnel `continue`
  L'instruction `continue` permet de passer directement au tour de boucle suivant sans exécuter les autres instructions de la boucle.

```
#include <stdio.h> /* inclusion de la librairie stdio.h , inst
```

```
int main() /* Entête de la fonction principale main */
{
  int i; /* déclaration de l'entier i */
  for (i=0; i < 6; i++ )
    {
      if (i == 3)
        {
          continue; /* le continue permet de passer directemen
*/
        }
      printf("i= %d\n",i);

    }
  printf("La valeur de i en sortie de boucle = %d\n",i);
  return(0);/*  Pour que la fonction main prenne la valeur 0 *
}
```

9 Les pointeurs

9.1 Déclaration des pointeurs, et les opérateurs d'adressage et d'indirection

Un pointeur est une variable dont le contenu va être l'adresse d'une autre variable. Le type du pointeur dépend de celui de la variable pointé.
La syntaxe de déclaration d'un pointeur `pt` est :

```
type * pt
```

On rappelle que les opérateurs d'adressage & et d'indirection * ont été définis page 34.

```
#include <stdio.h>
int main()
{
  int * adresse1,* adresse2,* adresse3;
  double * adresse4;
  int n=10, p=20, q;
  double x;
  x=3.151;
  adresse1=&n; /* utilisation de l'opérateur d'adressage */
  adresse2=&p;
  printf("le pointeur adresse1 pointe initialement sur la valeur
  printf("le pointeur adresse2 pointe initialement sur la valeur
/*  Il est nécessaire d'effectuer l'affectation ci-dessous sinon
  adresse3=&q;
  adresse4=&x;
```

```
  *adresse1 = (*adresse2) +2; /* utilisation de l'opérateur d'in
  (*adresse2)++;
  *adresse3 = (*adresse1)*2;
  printf("le pointeur adresse1 pointe sur la valeur %d\n",*adres
  printf("le pointeur adresse2 pointe sur la valeur %d\n",*adres
  printf("le pointeur adresse3 pointe sur la valeur %d\n",*adres
  printf("le pointeur adresse4 pointe sur la valeur %f\n",*adres
  printf("p= %d\n",p);
  return(0);
}

/* exemple issu du cours de Langage C de l'INRIA
   d'Anne Canteaut et modifié par Gilles Renversez */
/* ------------------------------------------------------------ */
/* Initiation aux pointeurs:
   déclaration et premières manipulations                       */
/* ------------------------------------------------------------ */
#include <stdio.h>
int main()
{
  int i= 3, j=5;
  int *p1, *p2, *p3, *p4;
  p1=&i;
  p2=&j;
  p3=&i;
  p4=&j;
  printf("Initialisation\n");
  printf(" objet || adresse     | valeur \n");
  printf(" i     || %lu | %ld \n",&i,i);
  printf(" p1    || %lu | %lu \n",&p1,p1);
```

```
  printf(" *p1   || %lu | %ld \n",&(*p1),*p1); /*   &(*p1) est b
*/
  printf("-----------------------------------\n");
  printf(" j     || %lu | %ld \n",&j,j);
  printf(" p2    || %lu | %lu \n",&p2,p2);
  printf(" *p2   || %lu | %ld \n",p2,*p2); /*   p2 est bien évid
*/
  printf("-----------------------------------\n");
  printf(" p3    || %lu | %lu \n",&p3,p3);
  printf(" p4    || %lu | %lu \n",&p4,p4);
  *p1=*p2; /* On manipule le contenu des pointeurs, l'objet poin
  printf("-----------------------------------\n");
  printf("Manipulation 1:  *p1=*p2  \n");
  printf(" objet || adresse     | valeur \n");
  printf(" i     || %lu | %ld \n",&i,i);
  printf(" j     || %lu | %ld \n",&j,j);
  printf(" p1    || %lu | %lu \n",&p1,p1);
  printf(" p2    || %lu | %lu \n",&p2,p2);
  i= 3, j=5; /* on réinitialise i et j à leurs valeurs initiales
  p3=p4; /* On manipule directement les pointeurs */
  printf("-----------------------------------\n");
  printf("Manipulation 2:  p3=p4 (précédée de la réinitialiation
  printf(" objet || adresse     | valeur \n");
  printf(" i     || %lu | %ld \n",&i,i);
  printf(" j     || %lu | %ld \n",&j,j);
  printf(" p3    || %lu | %lu \n",&p3,p3);
  printf(" p4    || %lu | %lu \n",&p4,p4);

  return (0);
}
```

9.2 L'allocation dynamique de mémoire

On a déjà vu comment initialiser un pointeur `p` par une affection d'adresse :

```
int i=3;
int * p;
p=&i;
```

Il est aussi possible d'affecter directement une valeur à `*p`. Mais, il faut pour cela avoir déjà réservé au pointeur un espace mémoire suffisant. L'adresse de cette espace mémoire sera la valeur de `p`.

Cette opération consistant à réserver un espace mémoire pour stocker l'objet pointé s'appelle l'*allocation dynamique*. La fonction C réalisant cela est définie dans la librairie standard, sa syntaxe est :

```
malloc(nb_octets_objet)
```

Cette fonction retourne un pointeur de type `char *` pointant vers un objet de taille `nb_octets_objet`.

Pour allouer des pointeurs vers des objets qui ne sont pas de type `char`, il faut effectuer une conversion de type du résultat de la fonction `malloc`. Ainsi pour allouer dynamiquement un

pointeur vers un entier on écrira :

```
int *p;
p=(int *)malloc(sizeof(int));
```

La fonction `calloc` de la librairie `stdlib.h` permet d'allouer dynamiquement la mémoire pour `nbre` pointeurs sur `nbre` objets du même type et en même temps elle initialise à 0 ces objets.

Sa syntaxe est :

```
calloc(nbre,nb_octets_objet)
```

`p` étant de type `int *` : l'instruction :

```
p=(int *)calloc(nbre,sizeof(int));
```

est équivalente à :

```
p=(int *) malloc(nbre*sizeof(int));
for (i=1; i<=nbre;i++)
  {
    *(p+i)=0;
  }
```

Après utilisation, il faut libérer la mémoire allouée, on utilise la fonction `free` :

free(p)

```
/* Exemple issu du cours de Langage C de l'INRIA
   d'A. C. et modifié par G. R. */
/* ------------------------------------------------------------ */
/* exemple simple d'allocation dynamique
   avec la fonction  malloc */
#include <stdio.h> /* chargement des fonctions de gestion
                      des entrées/sorties */
#include <stdlib.h> /* chargement des fonctions de gestion
                       dynamique de la mémoire */
int main ()
{
  int i =3;
  int *p;
  double r=1.50;
  double *pr;
  printf("avec entiers\n");
  printf("valeur avant initialisation de p\n");
  printf("objet ||  adresse   | valeur \n");
  printf("i     || %10lu | %ld\n",&i,i);
  printf("p     || %10lu | %10lu\n",&p,p);
  p=(int*) malloc(sizeof(int)); /* attribution  d'une valeur
                                   à p via l'allocation dynamiqu
  printf("valeur après initialisation de p\n");
  printf("objet ||  adresse   | valeur \n");
  printf("i     || %10lu | %ld\n",&i,i);
  printf("p     || %10lu | %10lu\n",&p,p);
  printf("*p    || %10lu | %ld\n",p,*p);
  *p=i; /* attribution  d'une valeur à *p via l'affectation */
```

```
printf("valeur après initialisation de p*\n");
printf("objet ||  adresse   | valeur \n");
printf("i     || %10lu | %ld\n",&i,i);
printf("p     || %10lu | %10lu\n",&p,p);
printf("*p    || %10lu | %ld\n",p,*p);
free(p); /* libération de l'espace mémoire alloué à p */
/* même chose mais avec des réels double precision */
printf("avec reéls double précision\n");
printf("valeur avant initialisation de pr\n");
printf("objet ||  adresse   | valeur \n");
printf("r     || %10lu | %lf\n",&r,r);
printf("pr     || %10lu | %10lu\n",&pr,pr);
pr=(double *) malloc(sizeof(double)); /* attribution d'une
                                         valeur à pr via
                                         l'allocation dynamique */
printf("valeur après initialisation de pr\n");
printf("objet ||  adresse   | valeur \n");
printf("r     || %10lu | %lf\n",&r,r);
printf("pr     || %10lu | %10lu\n",&pr,pr);
printf("*pr    || %10lu | %lf\n",pr,*pr);
* pr=r; /* attribution  d'une valeur à *p
          via l'affectation */
printf("valeur après initialisation de pr\n");
printf("objet ||  adresse   | valeur \n");
printf("r     || %10lu | %lf\n",&r,r);
printf("pr     || %10lu | %10lu\n",&p,p);
printf("*pr    || %10lu | %lf\n",pr,*pr);
free(pr); /* liberation de l'espace mémoire alloué à pr */
return (0);
```

```
}

/* Exemple issu du cours de Langage C de l'INRIA
   d'A. C. et modifie par G. R. */
/* ------------------------------------------------------ */
/* exemple d'allocation dynamique multiple */
#include <stdio.h> /* chargement des fonctions de gestion
                      des entrees/sorties */
#include <stdlib.h> /* chargement des fonctions de gestion
                       dynamique de la memoire */
int main ()
{
  int n;
  int i = 3;
  int j = 5;
  int *p;
  printf("allocation de deux objets par");
  printf(" un seul appel à la fonction malloc\n");
  p=(int*) malloc(2*sizeof(int)); /* allocation memoire pour
                          deux objets pointant sur
                          des entiers via l'allocation dynamique
  *p=i;
   *(p+1)=j;
  printf("p= %10lu \t *p =%10d \t p+1 = %10lu \t *(p+1) =%10d\n"
         p,*p,p+1,*(p+1));
  free(p);
  printf("allocation de N objets par un seul appel à la");
  printf(" fonction malloc, puis mise à zero des objets pointes\
  printf("donner le nbre d\'elements à stocker:\n");
  scanf("%d",&n);
```

```
  p=(int*) malloc(n*sizeof(int)); /* la fonction malloc a un
                                     seul argument */
  for (i=0;i<n;i++)
    {
      *(p+i)=0;
      printf("p[%d]=*(p+i)=  %10d\n",i,*(p+i));
    }
  free(p);
  printf("allocation de N objets par un seul appel à la");
  printf(" fonction calloc, avec mise à zero automatique de l\'o
  p=(int*) calloc(n,sizeof(int)); /* la fonction calloc a deux
                                     arguments */
  for (i=0;i<n;i++)
    {
      printf("p[%d]=*(p+i)=  %10d\n",i,*(p+i));
    }
  free(p);
  return(0);
}
```

9.3 Les pointeurs et les tableaux à plusieurs dimensions

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur.

```
/* Exemple issu du cours de Langage C de l'INRIA
   d'A. C. et modifie par G. R. */
/* ---------------------------------------------------------- */
/* exemple d'allocation dynamique de tableaux
```

```
   multidimensionnels:
   cas d'une matrice l lignes  m colonnes */
#include <stdio.h> /* chargement des fonctions de gestion des en
#include <stdlib.h> /* chargement des fonctions de gestion dynam
int main ()
{
  int i,l,m;
  int **tab;
  printf("donner le nbre de lignes:\n");
  scanf("%d",&l);
  printf("donner le nbre de colonnes:\n");
  scanf("%d",&m);
  printf("la matrice comprendra %d lignes x %d colonnes\n",l,m);
  printf("debut de l'allocation dynamique\n");
  tab = (int**)malloc(l*sizeof(int*)); /* reservation de la
 memoire pour l'objet pointe tab correspondant à l pointeurs
 sur des pointeurs vers des entiers, ces l pointeurs
 correspondent aux lignes de la matrice*/
  for (i=0; i<l; i++)
    {
      tab[i]=(int*)malloc(m*sizeof(int)); /* pour chaque
 pointeur tab[i], on reserve la memoire requise pour stocker
 m entiers, rappel : tab[i] est un pointeur constant vers un
 objet de type entier qui est le premier element de la ligne
 d'indice i, tab[i] a donc une valeur constante egale
 à &tab[i][0] */
    }
  /* liberation de la memoire allouee */
  printf("liberation de de la memoire allouee dynamiquement\n");
  for (i=0; i<l; i++)
```

```
    {
      free(tab[i]);
    }
  free(tab); /* instruction nécessaire pour libérer
                l'intégralité de la mémoire allouée */
  /* illustration de la possibilite de creer des tableaux
bidimensionnels dont les lignes n'ont pas toutes
le même nombre d'elements */
  printf("allocation dynamique de pointeurs bidimensionels");
  printf(" ayant un nombre de colonnes variables\n");
  tab = (int**)malloc(l*sizeof(int*));  /* reservation de
 la memoire pour l'objet pointe tab correspondant à
l pointeurs sur des pointeurs vers des entiers,
ces l pointeurs correspondent aux lignes de la matrice*/
  for (i=0; i<l; i++)
    {
      tab[i]=(int*)malloc((i+1)*sizeof(int)); /* tab[i] contient
 exactement i+1 elements et non plus m elements */
    }
    /* liberation de la memoire allouee */
  printf("liberation de la memoire allouee dynamiquement\n");
  for (i=0; i<l; i++)
    {
      free(tab[i]);
    }
  free(tab);
  return 0;
}
```

9.4 Les pointeurs et leurs relations avec les tableaux

```
/* Exemple issu du cours de Langage C de l'INRIA
   d'A. C. et modifié par G. R. */
/* ------------------------------------------------------ */
/* Utilisation de l'arithmétique des pointeurs
   pour la gestion des tableaux:

   illustration de la propriété: p[i]=*(p+i),
   opérateur de comparaison, d'incrémentation,
   et de décrémentation */
#include <stdio.h>
#define N 5
int main()
{
  int indi;
  int tab[] = {1,2,4,8,16};
  int *p;
  /* On boucle sur l'indice:  indi */
  for (p=tab,indi=0;indi<N;indi++,p++)  /* revoir si
                          besoin la syntaxe du for */
    {
      printf("tab[%2d] || %lu | %d\n",indi,p,*p);
    }
  /* On boucle sur le pointeur: p */
  printf("\n ordre croissant: \n");
  printf("objet   ||  adresse   | valeur \n");
  printf("-----------------------------\n");
  for (p=&tab[0],indi=0; p <= &tab[N-1]; p++,indi++)
    {
      printf("tab[%2d] || %lu | %d\n",indi,p,*p);
    }
```

```
  printf("\n ordre décroissant: \n");
  printf("objet   ||  adresse   | valeur \n");
  printf("-----------------------------\n");
  for (p=&tab[N-1],indi=4; p >= &tab[0]; p--,indi--)
    {
      printf("tab[%2d] || %lu | %d\n",indi,p,*p);
    }
  return (0);
}
```

La manipulation des tableaux, et non des pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant :

1. on ne peut pas créer de tableau dont la taille est une variable du programme
2. on ne pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Les deux différences entre un tableau et un pointeur sont :

1. un pointeur doit toujours être initialisé soit par une allocation dynamique soit par une affectation d'une adresse.
2. un tableau n'est pas une G-valeur ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. Un tableau ne supporte par d'arithmétique contrairement au pointeur.

9.5 Les pointeurs et les chaînes de caractères

Une chaîne de caractères est un tableau unidimensionnel d'objets de type `char` se terminant par le caractère nul codé par la séquence d'échappement '\0'.

On va donc pouvoir manipuler une chaîne de caractères via un pointeur sur un objet de type `char`.

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

int main(void)
{
  int i;
  char *chaine;

  chaine= "voici une chaîne de caractères";
  printf("le nombre de caracteres est:%d\n",strlen(chaine));
  for (i=0; *chaine != '\0'; i++)
    {
      chaine++; /* arithmetique sur le pointeur  "chaine" */
    }
  printf("le nombre de caracteres est:%d\n",i);
  printf("le nombre de caracteres est:%d\n",strlen(chaine));
  return(0);
}
```

9.6 Les pointeurs et les structures

Contrairement aux tableaux, les structures sont des *G-valeurs*. Ces structures possédent une adresse que l'on va pouvoir manipuler via les pointeurs. Ainsi, dans l'exemple suivant on crée à l'aide d'un pointeur un tableau de structures.

```
/* Exemple d'un tableau de structures géré par pointeur */
# include <stdio.h>
# include <stdlib.h>

struct personne
{
  char nom[20];
  int identifiant;
};
typedef struct personne *liste_personne; /* le synonyme
"liste_personne" définit donc un pointeur sur une structure
de type "personne" */

int main(void)
{
  int n,i;
  liste_personne liste_entrant;
  liste_personne liste_sortant;
  do{
    printf("\nDonner le nombre d\'entrants (>=2)\n");
    scanf("%d",&n);
  }  while (n<2);
  liste_entrant=(liste_personne)calloc(n,
                sizeof(struct personne));
```

```
  liste_sortant=(liste_personne)calloc(n,
                  sizeof(struct personne));

  for (i=0 ; i <n ; i++)
    {
      printf("\n saisie de la personne %d\n",i);
      printf("donner son nom\n");
      scanf("%s",&liste_entrant[i].nom); /* essayez la
                              compilation sans le '&' ! */
      printf("donner son identifiant\n");
      scanf("%d",&liste_entrant[i].identifiant);
    }
  liste_sortant=liste_entrant; /* affectation directe possible
                                  car G-valeurs */
  i=0;
  printf("personne sortante %d, nom %s , identifiant %d\n",
   i,liste_sortant[i].nom,liste_sortant[i].identifiant);
  i=1;
  printf("personne sortante %d, nom %s , identifiant %d\n",
   i,(liste_sortant+i)->nom,(*(liste_sortant+i)).identifiant);
/* 2 écritures équivalentes mais la seconde est plus lourde */

  free(liste_entrant);
  /*   free(liste_sortant); *//* il est en fait déjà libéré */

  return(0);
}
```

10 Les fonctions

10.1 Introduction

On peut séparer un programme C en plusieurs fonctions.

Une seule de ces fonctions est obligatoire : la fonction principale `main`.

Chaque fonction secondaire peut appeler d'autres fonctions secondaires ou même s'appeler elle-même (récursivité).

10.2 Définition

La définition d'une fonction correspond au texte de son algorithme, qu'on appelle corps de la fonction précédé de l'*entête* de la fonction. La syntaxe est la suivante pour une fonction ayant n arguments.

```
type nom_fonction(type_1 argu_1, type_2 argu_2, ...
                  , type_n argu_n)
{
[déclarations des variables locales]
liste d'instructions
return(expression);
}
```

La valeur de `expression` est la valeur que retourne la fonction : le type de `expression` doit être le même que celui de la fonction déclarée dans l'entête.

10.3 Appel d'une fonction

La syntaxe d'appel est :

```
nom_fonction (argu_1, argu_2, ...,  argu_n)
```

10.4 Déclaration d'une fonction

Il est nécessaire que la fonction ait été définie ou déclarée avant tout appel à cette fonction.

La syntaxe pour la déclaration est : type nom_fonction (type_1,type_2, ..., typ

On parle de *prototype* de la fonction.

Dans le cas où la définition de la fonction est faite avant le premier appel, il est fortement recommandé de déclarer la fonction car :

- seule cette déclaration permet au compilateur de vérifier que le nombre et le type des arguments d'appel sont en accord avec la définition.
- elle permet au compilateur d'effectuer d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types donnés dans le prototype.

10.5 Durée de vie et portée des variables

Durée de vie :(ou classe d'allocation)
Définition : **La durée de vie** d'une variable est le temps pendant lequel cette variable a un existence en mémoire.

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière.

En particulier, elles n'ont pas toutes la même *durée de vie*.

1. les variables *permamentes* (ou *statiques*)

   Une variable permamente occupe un emplacement en mémoire qui reste le même durant toute l'éxécution du programme. Par conséquent, elle existe pendant toute la durée du programme. Son emplacement est alloué une fois pour toutes lors de la compilation. Elles sont caractérisées par l'attribut `static`. Elles sont initialisées à zéro par le compilateur.

2. les variables *temporaires*

   les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

La durée de vie des variables est lié à leur *portée*, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

Portée des variables :(ou espace de validité ou visibilité)
La **portée** d'une variable est l'endroit du programme où elle existe et est accessible.

1. les variables *globales*

   On appelle variable globale une variable déclarée en dehors de toute fonction.

   Une telle variable est connue du compilateur dans toute la portion de code qui suit sa déclaration.

   Par construction, une variable globale est *permanente*.

```
/* exemple de déclaration et d'utilisation d'une variable gl
#include <stdio.h>
int n; /* n est une variable globale */
void ma_fonction();  /* déclaration de la fonction */

void ma_fonction() /* définition de la fonction */
{
  n++;
  printf("appel numéro %d\n",n);
  return;
}
int main()
{
  int i;
  for (i=0 ; i< 5; i++)
    {
      ma_fonction(); /* appel de la fonction */
    }
  return(0);
}
```

2. les variables *locales*

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instruction) du programme.
Par défaut, ce type de variable est *temporaire*.
Quand une fonction est appelée, elle place ses variables locales dans une pile, à la sortie de la fonction, les variables locales sont perdues.
Si une variable est déclarée dans une fonction, sa durée de vie est limitée à une seule exécution de cette fonction.
On peut créer une variable locale de classe statique en utilisant l'attribut `static`. La variable reste locale mais sa valeur est conservée d'un appel au suivant.

```
/* exemple de déclaration et d'utilisation d'une variable gl
#include <stdio.h>
int n=10; /* n est une variable GLOBALE  */
void ma_fonction(); /* déclaration de la fonction */

void ma_fonction()
{
  static int n; /* n est une variable LOCALE, cette définiti
  n++;
  printf("appel numéro %d\n",n);
```

```
  return;
}

int main()
{
  int i;
  printf("dans le main, avant les appels à la fonction, n= %
  for (i=0 ; i<5; i ++)
  {
    ma_fonction();
  printf(" on est sorti de ma_fonction \n");
  }
  printf("dans le main, après les appels à la fonction, n= %
  return(0);
}
```

10.6 Le passage d'arguments d'une fonction lors d'un appel : passage par valeurs

Les *paramètres d'appel* d'une fonction (aussi nommés *arguments d'appel* ou *arguments effectifs*) sont traités comme des variables locales : lors de l'appel de la fonction, les paramètres d'appel sont recopiés.

La fonction travaille uniquement sur cette copie. Cette copie disparaît lors du retour à la fonction appelante. Ceci implique que si la fonction modifie la valeur d'un de ses paramètres, seule la copie est modifiée.

```
#include <stdio.h>
void echange( int , int );
void echange (int aa, int bb) /* fonction de type void i.e. "san
     /* rien ne suit le return finissant la fonction */
{
  int tempo;
  printf("\tdébut échange: aa=%d et bb=%d\n",aa,bb);
  tempo=aa;
  aa=bb;
  bb=tempo;
  printf("\tfin échange: aa=%d et bb=%d\n",aa,bb);
  return;
}
int main()
{
  int n=10, p=20;
  printf("avant l'appel: n=%d et p=%d\n",n,p);
  echange(n,p);
  printf("apres l'appel: n=%d et p=%d\n",n,p);
  return(0);
}
```

10.7 Le passage d'arguments par adresse

En langage C, le passage des arguments entre fonctions doit s'effectuer par adresse.

```
#include <stdio.h>
void echange_adresse(int *, int *);/* prototype de la fonction "
int main()
```

```
{
  int n=10, p=20;
  printf("avant l'appel: n=%d et p=%d\n",n,p);
  echange_adresse(&n,&p); /* utilisation de l'opérateur d'adress
dans l appel de la fonction "echange_adresse" */
  printf("apres l'appel: n=%d et p=%d\n",n,p);
  return(0);
}
void echange_adresse(int * pa, int * pb)
{
  int tempo;
  printf("\tdébut échange: *pa=%d et *pb=%d\n",*pa,*pb);
  tempo= *pa; /* utilisation de l'opérateur d'indirection * */
  *pa= *pb;
  *pb= tempo;
  printf("\tfin échange: *pa=%d et *pb=%d\n",*pa,*pb);
  return;
}
```

10.8 La fonction `main`

La fonction `main` est de type `int` (même si le type `void` est toléré).

La valeur de retour 0 correspond à une terminaison correcte, toute autre valeur correspond à une terminaison sur une erreur.

Deux constantes symboliques sont définies dans `stdlib.h` :

1. EXIT_SUCCESS qui vaut 0
2. EXIT_FAILURE qui vaut 1

La fonction `main` peut posséder des paramètres formels. Par convention :

1. le premier se nomme `argc` : il est de type `int`, sa valeur est égale au nombre de mots composants la ligne de commande (*i.e.* au nombre de paramètres effectifs plus un).
2. le second se nomme `argv` : c'est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande.
   `argv[0]` contient donc le nom de l'exécutable et `argv[1]` contient le premier paramètre.

Le prototype le plus complet de la fonction `main` est :

```
int main(int argc, char *argv[]);
```

```c
#include <stdio.h>
#include <stdlib.h> /* pour avoir avoir accès aux définitions de
int main(int argc, char *argv[])
{
  int a,b,prod;
  if (argc != 3)
    {
      printf("\nErreur: le nombre d\'arguments est invalide\n");
      printf("Usage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a= atoi(argv[1]); /* cette fonction définie dans stdlib.h pren
  b= atoi(argv[2]);
  prod=a*b;
  printf("\nLe produit de %d par %d est égal à: %d\n",a,b,prod);
  return(EXIT_SUCCESS);
}
```

10.9 Fonctions avec un nombre variable de paramètres

On peut en C définir des fonctions ayant un nombre variable d'arguments.

Une telle fonction doit posséder au moins un paramètre formel fixe. La notation ... spécifie que la fonction posséde un nombre quelconque de paramètres (potentiellement de types différents) en plus des paramètres formels fixés (au moins un).

Pour accéder à cette fonctionnalité, on utilise des macros définies dans le fichier `stdarg.h`. On procéde comme suit :

- dans le corps de la fonction, on définit une variable `ptr` pointant sur la liste des paramètres d'appel. Son type (défini dans `stdarg.h`) est `va_list`. Elle sera uniquement utilisée comme argument des deux macros ci-dessous :
- cette variable est initialisée à l'aide de la macro `va_start(ptr, nom_dernier_argu_explicit)`. Le nombre total d'argument est requis.
- On accède aux différents arguments de la liste par la macro `va_arg(ptr,type_argu_vari)`
- En fin de traitement des paramètres, on libère la liste des paramètres via la macro `va_end(ptr)`

- la définition : `type_fct nom_fct(type_argu_fixe_1 argu_`
  `argu_fixe_2,`
  `,type_argu_fixe_n argu_fixe_n,...)`
- la déclaration : `type_fct nom_fct(type_argu_fixe_1,type`
  `,type_argu_fixe_n,...)`
- appel avec 2 arguments optionnels : `nom_fct(argu_fixe_effecti`
  `,argu_fixe_effectif_n,argu_vari_1, _argu_vari_2)`

```
#include <stdio.h>
#include <stdlib.h> /* pour avoir avoir accès aux définitions
                       de EXIT_SUCCESS and EXIT_FAILURE */
#include <stdarg.h> /* pour avoir avoir accès aux macros va_xxxx
int addition_param(int , ...);

int addition_param(int  nbre_parametres, ...)
{
  int res= 0;
  int i;
  va_list liste_parametres; /* déclaration de liste_parametres
                               via la macro définie dans stdarg.

  va_start(liste_parametres, nbre_parametres);
  for (i=0; i< nbre_parametres; i++)
    {
      res +=va_arg(liste_parametres, int);
    }
  va_end(liste_parametres);
  return(res);
}
```

```
int main()
{
   int somme;
   printf("\n Appel de addition_param avec 1+3 paramètres\n");
   somme=addition_param(3,2,4,6);
   printf("la somme vaut: %d\n",somme);
   printf("\n Appel de addition_param avec 1+5 paramètres\n");
   somme=addition_param(5,1,2,3,4,5);
   printf("la somme vaut: %d\n",somme);
   return(EXIT_SUCCESS);
}
```

10.10 Pointeurs sur des fonctions

Le type d'une fonction n'est défini que par le type de l'objet rendu et par le nombre et les types des arguments. Ce type ne donne aucune indication sur l'encombrement de la fonction (*i.e.* la taille de son code).

Il faut donc bien comprendre que le type *"fonction rendant un ..."* n'a pas de taille prédéfinie.

L'objet variable le plus simple permettant de désigner une fonction doit avoir le type *"adresse d'une fonction rendant un ..."*

Un **pointeur sur une fonction** correspond à l'adresse du début du code de la fonction. Un pointeur sur une fonction de prototype :

```
type ma_fonction(type_1, type_2, ...,
                 type_n)
```

est de type :

```
type (*)(type_1, type_2, ..., type_n)
```

-Un exemple de fonction en argument d'une autre fonction (on parle de fonction formelle) :

```
#include <stdio.h>
#define EPSILON 1.0e-6
double polynome_2(double);
double polynome_3(double);
double dichotomie(double,double,double,double (*)(double));
double appel_generique(double,double (*)(double));

double dichotomie(double a,double b,double epsilon,double (*f)(d
{
  double x;
  while (b-a > epsilon)
    {
      x=(a+b)/2.0;
      if ( (*f)(x) <0 )
        {
          a=x;
        }
      else
        {
          b=x;
        }
    }
  return(x);
}
double appel_generique(double x,double (*f)(double))
{
  double y;
  y=(*f)(x);
  return(y);
}
```

```
int main()
{
  double a,b,racine,epsilon;
  epsilon=EPSILON;
  do
    {
      printf("Entrez a et b tels que a<b et f(a)<0<f(b)\n");
      printf("a=\n");
      scanf("%lf",&a);
      printf("b=\n");
      scanf("%lf",&b);
    } while ( (a<b) && (appel_generique(a,polynome_3)<0) &&
              (appel_generique(b,polynome_3)>0) ) ;
  racine=dichotomie(a,b,epsilon,polynome_3);
  printf("La racine est en %lf a %g pres\n",racine,epsilon);
  return(0);
}
double polynome_2(double x)
{
  double y;
  y=x*x-3.0*x+2.0;
  return(y);
}
double polynome_3(double x)
{
  double y;
  y=x*x*x-2.0*x+4.0;
  return(y);
}
```

-Un exemple de tableaux de fonctions :
chaque élément du tableau est formé de deux champs :

1. le nom d'une fonction standard, sous forme de chaîne de caractères
2. l'adresse de la fonction correspondante

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#define NB_FCT (sizeof tableau_fct/ sizeof tableau_fct[0])
struct
{
  char *nom;
  double (*fonct)(double);
} tableau_fct[] ={"sin", sin,
                  "cos", cos,
                  "exp", exp,
                  "log", log};


int main()
{
  int i;
  char nom[80];
  double x;

  for(;;)
    {
```

```
      printf("donner le nom de la fonction a evaluer et ");
      printf("une valeur a calculer (taper \'fin\' pour arreter)
      scanf("%s", nom);
      if (strcmp(nom,"fin") == 0)
        {
          exit(0);
        }
      scanf("%lf", &x);
      for (i=0; i < NB_FCT &&
             strcmp(tableau_fct[i].nom,nom) != 0; i++ ) {}
      if (i< NB_FCT)
        {
          printf("%s(%.5f) = %f\n",nom,x,(*tableau_fct[i].fonct)
        }
      else
        {
          printf("%s pas definie\n",nom);
        }
    }
  return(0);
}
```

11 Les entrées/sorties

On peut bien évidemment lire et écrire des données dans un fichier.
Les accès à un fichier se font par l'intermédiaire d'une mémoire tampon *buffer*.
Les informations dont a besoin le programme pour manipuler le fichier sont rassemblées dans une structure dont le type, `FILE *`, est défini dans `stdio.h`.
Un objet de type `FILE *` est appelé *flot de données*.

3 flots spécifiques sont définis en langage C : ils n'ont pas besoin d'avoir été ouverts ou d'être fermés.

1. `stdin` : unitée d'entrée, par défaut le clavier
2. `stdout` : unitée de sortie, par défaut l'écran
3. `stderr` : unitée d'affichage des messages d'erreur, par défaut l'écran

11.1 Déclaration

FILE *nom_du_flot

11.2 Ouverture et fermeture d'un fichier

ouverture :

nom_du_flot=fopen("nom_du_fichier","type_d_action")

r	ouverture d'un fichier texte en lecture
w	ouverture d'un fichier texte en écriture
a	ouverture d'un fichier texte en écriture à la fin
r+,w+	ouverture d'un fichier texte en lecture/écriture
a+	ouverture d'un fichier texte en lecture/écriture à la fin

TAB. 6: Principaux types d'action possible pour la fonction `fopen` pour des fichiers textes

- Si le type d'action contient la lettre `r` : le fichier doit exister.
- Si le type d'action contient la lettre `w` : le fichier peut ne pas exister. Dans ce cas, il sera créé, dans le cas contraire, son contenu précédent est écrasé.
- Si le type d'action contient la lettre `a` : le fichier peut ne pas exister. Dans ce cas, il sera créé, dans le cas contraire, les écritures sont faites à la suite contenu précédent.

fermeture :(utilisation d'une fonction de type `void`)

fclose (nom_du_flot)

11.3 Ecriture et lecture formatées d'un fichier

écriture :

fprintf (nom_du_flot , "`chaine_de_controle`",expr_1,expr_2,...,expr_n)

lecture :

fscanf(nom_du_flot,"chaine_de_controle",argu_1,argu_2,...,argu_n)

11.4 Test de fin de fichier

test de fin :

int feof(nom_du_flot)

Cette fonction prend la valeur 1 (vrai) si la fin du fichier est rencontrée,

11.5 Ecriture et lecture de caractères

écriture :

int fputc(**int** caractere ,FILE *nom_du_flot)

lecture :

int fgetc(FILE *nom_du_flot)

Elle retourne l'entier correspondant au caractère lu (ou la constante EOF en cas d'erreur).

Un exemple simple réalisant la recopie caractère par caractère d'un fichier existant dans un autre existant ou pas :

```
#include <stdio.h>
#include <stdlib.h>  /* contient les déclarations
                        de EXIT_SUCCESS et EXIT_FAILURE */
#define FICHIER_ENTREE "entree.txt"
#define FICHIER_SORTIE "sortie.txt"
```

```
int main()
{
  FILE *f_in,*f_out;
  int c;

  if ((f_in = fopen(FICHIER_ENTREE,"r")) == NULL )
    {
      fprintf(stderr,
              "\nErreur: Impossible d'ouvrir le fichier %s\n"
              ,FICHIER_ENTREE);
      return(EXIT_FAILURE);
    }
  if ((f_out = fopen(FICHIER_SORTIE,"w")) == NULL )
    {
      fprintf(stderr,
              "\nErreur: Impossible d'ouvrir le fichier %s\n"
              ,FICHIER_SORTIE);
      return(EXIT_FAILURE);
    }
  fprintf(stdout,"\n");
  while ( (c=fgetc(f_in)) != EOF )
    /* enlever les secondes paranthèses, recompiler et tester!*/
    {
      fputc(c,f_out);
      fprintf(stdout,"%c",c);
    }
  fprintf(stdout,"\n");
  fclose(f_in);
  fclose(f_out);
  return(EXIT_SUCCESS);
```

}

12 La gestion des chaînes de caractères

12.1 La représentation des chaînes de caractères
12.2 Lecture et écriture des chaînes de caractères
12.3 Autres fonctions de gestion des chaînes de caractères

On doit fournir en argument de ces fonctions les adresses des chaînes de caractères. Le fichier `string.h` définie plusieurs fonctions utiles à la gestion des chaînes de caractères :

- `strcat( chaine_1, chaine_2)` : recopie `chaine_2` à la suite de `chaine_1` en effaçant la séquence d'échappement terminale `\0` de `chaine_1`.
- `strcat(chaine_1, chaine_2, taille_max)` : *idem* mais fixe la taille maximale possible pour `chaine_1` à `taille_`
- `strcmp(chaine_1,chaine_2)` : compare les deux chaînes et fournit une valeur entière définie comme :
  - positive si `chaine_1` >`chaine_2` (`chaine_1` arrive après `chaine_2` au sens lexicographique)
  - nulle si `chaine_1` =`chaine_2` (`chaine_1` est équivalent à `chaine_2` au sens lexicographique)
  - négative si `chaine_1` <`chaine_2` (`chaine_1` arrive avant `chaine_2` au sens lexicographique)

– `strlen(chaine_1)` retourne l'entier associé à la taille de `chaine_1`.

Le fichier `stdlib.h` définie aussi plusieurs fonctions utiles à la gestion des chaînes de caractères :

– `atoi(chaine_1)` convertie `chaine_1` en un résultat de type `int`.

12.4 La librairie `string.h`

La librairie `string.h` contient plusieurs fonctions utiles pour le traitement de chaînes de caractères. On peut citer notamment `strlen` qui retourne l'entier associé à la taille de la chaîne de caractères donnée en argument (sans compter le caractère nul). Sa syntaxe est :

```
strlen(nom_chaine_caractere)
```

où `chaine` est un pointeur sur une chaîne de caractère.

13 Bases de programmation modulaire

13.1 Introduction

En langage C, l'unité de compilation est le fichier. Mais il est possible de compiler séparément plusieurs fichiers sources et de rassembler les fichiers objets correspondants (et donc les moddules qu'ils contiennent) au moment de l'édition de lien.

Les possibilités de la compilation séparée ont une incidence importante au niveau de la portée des variables globales.

13.2 La portée d'une variable globale-déclaration `extern`

```
/* fichier source 1   */  /* fichier source 2 */
int x;
int main()                  double fct2(....)
{                           {
 ....                        ....
 ....                        ....
}                           }
double fct1(....)           double fct3(...)
{                           {
  ....                       ....
}                           }
```

On ne peut pas faire référence à la variable globale `x` déclarée dans le premier fichier source dans les fonctions `fct2` et `fct3` du second fichier source.

```
/* fichier source 1   */  /* fichier source 2 */
```

```
extern int x; /* <--- */
int main()                    double fct2(....)
{                             {
 ....                          ....
 ....                          ....
}                             }
double fct1(....)             double fct3(...)
{                             {
  ....                         ....
}                             }
```

L'éditeur de lien retrouvera dans le premier module objet (issu du premier fichier source) l'adresse effective de la variable globale ayant l'attribut **`extern`** et la rapportera dans le second module objet (issu du second fichier source).

Lorsqu'on alloue l'emplacement mémoire pour une variable on parle de *définition*.
Lorsqu'on référence une variable on parle de *déclaration*.
Les règles générales sont les suivantes :

1. Une variable initialisée sans l'attribut `extern` fait l'objet d'une *définition*.
2. Une variable avec l'attribut `extern` sans *initialisation* fait l'objet d'une *déclaration*.

13.3 Portées et durées de vie des variables : tableau

récapitulatif

Type de variable	Déclaration	Portée	Durée de vie
globale	en dehors de toute fonction	la partie du fichier suivant sa déclaration / n'importe quel fichier source avec l'attribut `extern`	permanente
globale cachée	en dehors de toute fonction avec l'attribut `static`	uniquement la partie du fichier source suivant sa déclaration	permanente
locale "persistante"	au début d'une fonction avec l'attribut `static`	la fonction	permanente
locale à une fonction	au début d'une fonction	la fonction	temporaire
locale à un bloc	au début d'un bloc	le bloc	temporaire

TAB. 7: Tableau récapitulatif des durées de vie et portées des variables selon leurs déclarations et attributs

13.4 La compilation séparée

13.5 L'utilitaire make

Afin de faciliter la compilation séparée des fichiers sources, on dispose d'un utilitaire nommé `make`.

1. **3 principes de base :**
   (a) `make` effectue uniquement les étapes de compilation requises à la création d'un exécutable ou d'une librairie.
   (b) `make` recherche par défaut dans le répertoire courant un fichier de nom `makefile` ou, si ce dernier n'existe pas, `Makefile`.
   (c) Ce fichier (appelons le `makefile`) contient les dépendances entre les différents fichiers sources, objets, et exécutables (et librairies éventuellement).
2. **Le contenu et le fonctionnement d'un fichier `makefile` :** Ce fichier est constitué d'une liste de règles de dépendance (*i.e.* une liste de fichiers dont dépend la cible) dont la syntaxe est :

```
une cible: liste de dépendances
<TAB> suite de commandes UNIX sur des lignes successives
```

Exemple avec deux règles :

```
## Exemple 1 de makefile
# (les commentaires commencent par un #)
```

```
# règle 1:
# la cible est l'éxecutable intitulé "operation.out"
# dépend des 3 fichiers sources .c
# et des deux fichiers d'en-tête .h
executable: main_operations.c produit.c somme.c produit.h somme.
        gcc -o operations.out main_operations.c produit.c somme.
# la cible "clean" usuellement définie sert à nettoyer
# le répertoire courant des fichiers intermédiaires créés
# lors de la compilation
# règle 2:
clean:
        rm -f *.o
## exemple d'utilisation: make -f makefile_1 executable
```

Exemple plus performant avec cinq règles :

```
## Exemple 2 de makefile
# cet exemple est plus performant que le 1

# la cible finale est mise en premier
# elle est lancée par un simple make
# elle a pour dépendance les cibles suivantes
# règle 1:
operations.out: main_operations.o produit.o somme.o
        gcc -o operations.out main_operations.o produit.o somme.
# des cibles intermédiaires sont définies
# règle 2:
main_operations.o: main_operations.c
        gcc -c main_operations.c
# règle 3:
```

```
produit.o: produit.c produit.h
        gcc -c produit.c
# règle 4:
somme.o: somme.c somme.h
        gcc -c somme.c
# règle 5:
clean:
        rm -f *.o
```

Pour faciliter et optimiser l'écriture des fichiers `makefile`, on dispose des *macros* et *abbréviations*, la syntaxe de définition d'une macro est la suivante :

```
nom_de_la_macro = contenu de la macro
```

Sa syntaxe d'utilisation est

```
$(nom_de_la_macro)
```

Exemple avec une macro :

```
## Exemple 3 de makefile
# avec une macro pour définir le compilateur
CC      =       gcc
# et une autre pour une option de compilation
DEBUG_FLAGS =   -g
# la cible finale est mise en premier
# elle est lancée par un simple make
# elle a pour dépendance les cibles suivantes
main_operations.out: main_operations.o produit.o somme.o
```

```
	$(CC) -o main_operations.out main_operations.o produit.o
# des cibles intermédiaires sont définies
main_operations.o: main_operations.c
	$(CC) -c main_operations.c
produit.o: produit.c produit.h
	$(CC) -c produit.c
somme.o: somme.c somme.h
	$(CC) -c somme.c
## On rajoute des cibles pour la compilation permettant le débog
#  avec l'option $(DEBUG_FLAGS) (fixée à -g)
main_operations.debug: main_operations.db produit.db somme.db
	$(CC) $(DEBUG_FLAGS) -o main_operations.debug main_opera
# des cibles intermédiaires sont définies
main_operations.db: main_operations.c
	$(CC) $(DEBUG_FLAGS) -o main_operations.db -c main_opera
produit.db: produit.c produit.h
	$(CC) $(DEBUG_FLAGS) -o produit.db -c produit.c
somme.db: somme.c somme.h
	$(CC) $(DEBUG_FLAGS) -o somme.db -c somme.c
clean:
	rm -f *.o
```

Certaines macros sont prédéfinies de manière à faciliter l'écriture du `makefile` :

- $@ pour le fichier cible courant
- $* pour le fichier cible courant sans son suffixe
- $< pour le fichier qui a provoqué l'action

On peut ainsi réécrire la commande de la règle de dépendance

de `somme.db` sous la forme :

```
$(CC) $(DEBUG_FLAGS) -o $@ -c $*.c
```

13.6 La création de bibliothèques

Une *bibliothèque* est une succession de *fichiers objets* contenant des fonctions, l'ensemble ainsi formé est en fait un *fichier d'archive*. Pour créer ce fichier d'archive (et donc la bibliothèque) on dispose de la commande `ar`. La syntaxe d'appel est la suivante :

```
ar [options] nom_bibliotheque liste_fichiers_à_inclure
```

Trois options sont très utiles :

1. -r : ajoute les fichiers listés dans une bibliothèque (et remplace ce qui aurait le même nom) ou crée la bibliothèque si elle n'existe pas.
2. -s : indexe le contenu des fichiers présents dans l'archive afin d'en faciliter l'extraction
3. -u (combinée avec -r) : effectue une mise à jour des éléments spécifiés de la bibliothèque

Pour créer la bibliothèque `libdeux_operations.a` à partir de l'exemple précédent on lancera :

```
ar -rs libdeux_operations.a produit.o somme.o
```

Pour compiler on lancera la commande (édition de liens) :

```
gcc -L. main_operations.o -ldeux_operations
```

En effet, il faut préciser le répertoire où est stocké cette librairie personnelle (non standard) ce qui est fait avec l'option -L,

et l'option -l recontruit le nom complet de la bibliothèque en rajoutant le préfixe `lib` ainsi que le suffixe `.a` au nom donné.

14 Quelques algorithmes de base

14.1 Le crible d'Eratosthène

Objectif : déterminer tous les nombres premiers compris entre 1 et une borne supérieure N

Méthode : on va éliminer successivement tous les nombres non premiers de l'intervalle [1,N] par tests successifs en éliminant les multiples des nombres premiers déjà trouvés.

Algorithme :

initialisation du tableau `a_tester` d'état des nombres $\in$ [1,N] à 0 (type pseudo-booléen) (a priori ils sont tous premiers)
initialisation à 1 du premier élement du tableau `a_tester`
initialisation de la variable `premier` stockant le premier nombre du tableau pas encore éliminé (état $\neq 0$) à 1 (type entier)
tant que $(\texttt{premier})^2 \leq N$ **faire**
  `premier`= `premier` +1
  tant que `a_tester[premier]` = 1 ET `premier` < N **faire**
    `premier`= `premier` +1
  fin tant que
  pour tous tous les multiples de `premier` **faire**
    `a_tester[multiple]`= 1
  fin pour
fin tant que

Programme :

```
#include <stdio.h>
#define N 100
#define ELIMINE  1
#define CONSERVE 0
int main(void)
{
  int a_tester[N+1],
    premier,
    i,
    compteur=0  ;
  for (i=1; i<= N; i++)
    {
      a_tester[i]=CONSERVE;
    }
  a_tester[1]=ELIMINE;
  premier=1;
  while (premier*premier<=N)
    {
      premier++;
      while ( a_tester[premier] && premier < N)
        {
          premier++;
        }
      for(i=2*premier ; i<= N; i +=premier)
        {
          a_tester[i]=ELIMINE;
        }
    }
  for (i=1;i<=N; i++)
    {
```

```
        if (a_tester[i] == CONSERVE)
          {
            printf("%6d",i);
            compteur++;
            if (compteur%5 == 0 )
              {
                printf("\n");
              }
          }
      }
    printf("\n");
}
```

14.2 Le tri par extraction simple

Objectif : trier une liste de nombres par valeur croissante

Méthode : on va procéder par extraction simple et successive. On recherche l'élément le plus petit de la liste. On l'échange avec le premier élément de la liste. On répète l'opération pour les éléments restants de la liste.

Remarques : On considérera des `double` et on fera le tri dans une fonction de type `void`.

Algorithme :

allocation et affection du tableau `a_trier` des nombres de type `double`
pour tous les éléments du tableau `a_trier` **faire**
initialisation de l'indice du minimum partiel `i_mini`
pour les éléments d'indice plus grand que ceux déjà triés **faire**
si `a_trier[i_courant]<a_trier[i_mini]` **alors**
`i_mini=i_courant`
fin si
fin pour
on permute le minimum partiel trouvé avec le premier élément non trié de `a_trier`
fin pour

Programme :
Il contient aussi une routine d'affichage de l'ensemble de la liste afin de visualiser les permutations effectuées, on en profite aussi pour sauver dans un fichier la liste initiale des valeurs afin de l'utiliser dans les exemples suivants.

```
/* Programme de tri (croissant) d'une liste de
   nombres de type réel double précision .
   Un seul tableau unidimensionnel alloué
   dynamiquement est utilisé.
   Gilles Renversez , décembre 2004.
   révisions: novembre 2005
*/
#include <stdio.h>
#include <stdlib.h>
#define FICHIER_SORTIE "liste_non_triee.dat"
void tri(int, double []); /* déclaration de la fonction
                             tri de type void */
void affichage(int, double []);
int ecriture_fichier(int, double []);
int main()/* partie principale du programme */
{
  int i, n;
  double *a_trier;
  /* saisie de la valeur de n */
  printf("\n Quel est le nombre de valeurs à saisir?\n");
  scanf("%d",&n);
  printf("\n");
  /* allocation dynamique du tableau a_trier via
     la fonction calloc à deux arguments */
  a_trier=(double*) calloc(n,sizeof(double));
  /* lecture de la série de valeur à saisir */
  for (i=0;i<n;i++)
    {
      printf("i =%d, a_trier= ",i);
      scanf("%lf",&a_trier[i]);
```

```
    }
  printf("\n Valeurs initiales: \n");
  affichage(n,a_trier);
  ecriture_fichier(n,a_trier);
  /* appel de la fonction de tri */
  printf("appel du tri par extraction\n");
  tri(n,a_trier);
  /* affichage des valeurs triées */
  printf("\n Valeurs triées par ordre croissant: \n");
  affichage(n,a_trier);
  free(a_trier); /* libération de l'espace mémoire alloué
                    à a_trier */
  return(EXIT_SUCCESS);
}

/* fonction effectuant le tri dans un seul tableau */
void tri(int n, double a_trier[])
{
  int i, indice, i_mini;
  double tampon;
  for (indice=0; indice < n-1; ++indice)
    {
      printf("indice  =%d\n",indice);
      i_mini=indice;
      /* détermination du minimum des éléments restants */
      for ( i= indice +1 ; i  < n; ++i)
        {
          if ( a_trier[i] < a_trier[i_mini])
            {
              i_mini=i;
```

```
            }
          }
        /* permutation des deux éléments */
        tampon= a_trier[i_mini];
        a_trier[i_mini]=a_trier[indice];
        a_trier[indice]=tampon;
        affichage(n,a_trier);
      }
  return;
}
/* fonction effectuant l'affichage du tableau */
void affichage(int n, double a_trier[])
{
  int indice;
  printf("\n");
  for (indice=0; indice < n; ++indice)
    {
      printf(" %g",a_trier[indice]);
    }
  printf("\n");
}
/* fonction effectuant l'écriture dans un fichier:
   un nombre par ligne de fichier */
int ecriture_fichier(int n, double a_trier[])
{
  int indice;
  FILE * f_sortie;
  if ((f_sortie = fopen(FICHIER_SORTIE,"w")) == NULL )
    {
      fprintf(stderr,
```

```
                "\nErreur: Impossible d'ouvrir le fichier %s\n"
                ,FICHIER_SORTIE);
        return(EXIT_FAILURE);
      }
    for (indice=0; indice < n; ++indice)
      {
        fprintf(f_sortie,"%lf\n",a_trier[indice]);
      }
    fclose(f_sortie);
    return(EXIT_SUCCESS);
}

/* pour compiler:
gcc -Wall -o tri_un_seul_tableau_extraction.out
tri_un_seul_tableau_extraction.c  */
```

14.3 Le tri par permutation simple ou tri à bulles

Objectif : trier une liste de nombres par valeur croissante en permuttant successivement, et si cela est nécessaire, deux éléments contigus de la liste.

Méthode : On parcourt l'ensemble du tableau, depuis sa fin jusqu'à son début, en comparant deux éléments contigus, on les permute s'ils sont mal ordonnés. Le plus petit élément de la liste se trouve alors en début de tableau.

On renouvelle le parcours de la liste avec les éléments restants, on a donc en tête de liste les deux plus petits éléments. On réitère cette opération jusqu'à ce que :

-soit l'avant dernier élément ait été classé (le dernier élément est alors obligatoirement à sa place).

-soit qu'aucune permutation n'ait eu lieu lors du dernier parcours de la liste.

(On considérera des `double` et on fera le tri dans une fonction de type `void`, la liste aura été lue dans un fichier).

Algorithme :

initialisation de `indice` à -1
initialisation de la variable pseudo-logique `permutation` à VRAI (valeur 1)
lecture du fichier contenant le tableau `a_trier`, comptage du nb d'éléments
initialisation de la taille `n` du tableau `a_trier`
lecture/initialisation des éléments du tableau
tant que `indice` < `n-1` ET `permutation` **faire**
  `permutation` = FAUX
  pour `indice`<`i_actuel`< n-2 **faire**
    si `a_trier[i_actuel]` > `a_trier[i_actuel+1]`
    alors
      `permutation` = VRAI
      échange `a_trier[i_actuel]`
      avec `a_trier[i_actuel+1]`
    fin si
  fin pour
  incrémentation d'`indice` de 1
fin tant que

Programme :

Il contient aussi une routine d'affichage de l'ensemble de la liste afin de visualiser les permutations effectuées. Il lit la liste à trier dans le fichier crée précédemment (type des données : `double`).

```
/* Programme de tri (croissant)
   par permutation simple et successive de
   deux éléments
   d'une liste de nombres de type
   réel double précision
   Un seul tableau unidimensionnel alloué
   dynamiquement est utilisé.
   Gilles Renversez , décembre 2004.
   révisions: novembre 2005
*/
#include <stdio.h>
#include <stdlib.h>
#define FICHIER_ENTREE "liste_non_triee.dat"
#define VRAI 1
#define FAUX 0
void tri_permutation(int, double []);
void affichage(int, double []);
int lecture_taille_fichier(void);
int lecture_fichier(int, double []);
int main()/* partie principale du programme */
{
  int n;
  double *a_trier;
  n=lecture_taille_fichier();
  printf("n  =%d\n",n);
  a_trier=(double*) calloc(n,sizeof(double));
  lecture_fichier(n,a_trier);
  printf("\n Valeurs initiales: \n");
  affichage(n,a_trier);
  printf("\n Valeurs intermédiaires: \n");
```

```
  tri_permutation(n,a_trier);
  printf("\n Valeurs triées: \n");
  affichage(n,a_trier);
  free(a_trier);
  return(EXIT_SUCCESS);
}
/* fonction effectuant le tri dans un seul tableau */
void tri_permutation(int n, double a_trier[])
{
  int i, indice, permutation;
  double tampon;
  indice=-1;
  permutation= VRAI;
  while (indice < n-1 && permutation )
    {
      permutation = FAUX;
      for (i=n-2;i>indice;i--)
        {
          if ( a_trier[i] > a_trier[i+1] )
            {
              permutation=VRAI;
              tampon=a_trier[i];
              a_trier[i]=a_trier[i+1];
              a_trier[i+1]=tampon;
              affichage(n,a_trier);
            }
        }
      indice++;
    }
  return;
```

```
}
/* fonction effectuant l'affichage du tableau */
void affichage(int n, double a_trier[])
{
  int indice;
  printf("\n");
  for (indice=0; indice < n; ++indice)
    {
      printf(" %g",a_trier[indice]);
    }
  printf("\n");
}
/* fonction calculant le nbre d'éléments */
int lecture_taille_fichier(void)
{
  int compteur;
  FILE * f_entree;
  if ((f_entree = fopen(FICHIER_ENTREE,"r")) == NULL )
    {
      fprintf(stderr,
              "\nErreur: Impossible d'ouvrir le fichier %s\n"
              ,FICHIER_ENTREE);
      return(EXIT_FAILURE);
    }
  compteur=-1;
  while ( !feof(f_entree)  )
    {
      fscanf(f_entree,"%*s");
      compteur++;
    }
```

```
  fclose(f_entree);
  return (compteur);
}
/* fonction lisant les éléments du fichier */
int lecture_fichier(int n,  double a_trier[])
{
 int i;
  FILE * f_entree;
  if ((f_entree = fopen(FICHIER_ENTREE,"r")) == NULL )
    {
      fprintf(stderr,
              "\nErreur: Impossible d'ouvrir le fichier %s\n"
              ,FICHIER_ENTREE);
      return(EXIT_FAILURE);
    }
  for (i=0; i < n; i++)
    {
      fscanf(f_entree,"%lf",&a_trier[i]);
    }
  fclose(f_entree);
  return(EXIT_SUCCESS);
}

/* pour compiler:
gcc -Wall -o tri_un_seul_tableau_permutation.out
tri_un_seul_tableau_permuation.c  */
```

14.4 Un tri évolué : l'algorithme de tri rapide (Quick-sort)

Il a été inventé par C.A.R. Hoare en 1960.

Listes des programmes