Langage C

E. Boucharé

Table des Matières

Introduction 1

Chapitre 1 Les types de données de C 3
1.1. Notion de type de données 3
1.2. Les types de base 3
1.2.1. Le type entier 3
1.2.2. Le type réel 4
1.2.3. Le type caractère 4
1.2.4. Les opérateurs arithmétiques raccourcis 5
1.3. Le type tableau 5
1.4. Les chaînes de caractères 5
1.5. Construction de types abstraits 6
1.5.1. la structure 6
1.5.2. L'union 7
1.5.3. L'énumération 7
1.6. Le type pointeur 7
1.7. L'instruction typedef 8
1.8. L'opérateur sizeof 9
1.9. La conversion de type 9

Chapitre 2. Les structures de contrôle en C 10
2.1. Expression booléenne 10
2.2. Le test simple, if ... else 10
2.3. La sélection, swith ... case 11
2.4. La boucle while 11
2.5. La boucle do ... while 12
2.6. La boucle for 12
2.7. Les instructions break, continue et goto 13

Chapitre 3. Les fonctions en C 14
3.1. Concept 14
3.2. Modes de passage des arguments 14
3.3. Statut et portée des identificateurs 15
3.4. La récursivité 15
3.5. Les pointeurs de fonction 16

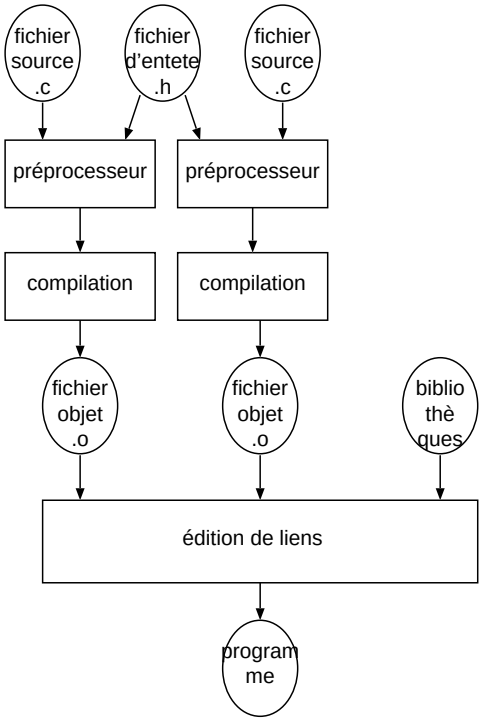
Chapitre 4. Les directives de compilation 17

Index 19

Introduction

Le langage C a été développé au début des années 70, en même temps que le système d'exploitation UNIX. Le langage C que nous utilisons est le C ANSI, issu de la norme établie en 1983 par L'American National Standard Institute (ANSI). Il s'agit d'un langage à usage général, relativement proche de la machine, mais néanmoins portable (indépendant de la mahine). Tous les systèmes d'exploitation sont aujourd'hui codés en C (UNIX, Linux, Windows) ou en un de ses dérivés C++ (BeOS) ou Objective C (OpenStep, MaOS X).

Le langage C offre un petit nombre de fonctions en interne, mais il permet d'étendre ses possibilités à l'infini par l'ajou de bibliothèques (pour écrire ou lire des caractères, dessiner à l'écran, gérer les périphériques …). Les sources d'un programme peuvent être séparés dans plusieurs fichiers et compilés séparément offrant ainsi une grande souplesse de développement. Avant la phase de compilation il y a une phase de prétraitement des sources par le préprocesseur (voir le chapitre 4). Une fois tous les sources compilés, l'éditeur de lien permet de rassembler tous les fichiers objets issus de la compilation, ainsi que les bibliothèques pour former le fichier éxécutable final.



Les mots clés du langage sont réservés : ils ne peuvent pas servir comme identificateur de variable ou de fonction.

```
auto    break   case      char    const     continue  default  do
double  else    enum      extern  float     for       goto     if
int     long    register  return  short     signed    sizeof   static
struct  switch  typedef   union   unsigned  void      while
```

Un programme permet de manipuler des données plus ou moins complexes (un ou des nombres, du texte, une liste de noms …). Ces données sont associées à des *variables*. Ces variables sont manipulées à l'aide d'*opérateurs* (+, -, …) et de *fonctions* (trouver un mot dans un texte, …). Tout le problème consiste souvent à analyser correctement la situation afin de définir quelles sont les données manipulées et quelles fonctions définir pour les manipuler.

Une bonne habitude à prendre lorsqu'on programme est de commenter le code au fur et à mesure. Cela rend le source d'un programme exploitable sur le long terme de commenter le code (modifiation ultérieures) et permet aux autres de comprendre ce qu'on a voulu faire. Les commentaires sont placés entre les symbôles suivants :

```
/*  commentaires */.
```

Ce document est probablement assez difficile à lire d'un bout à l'autre quand on n'a pas déjà une expérience en programmation. Il est davantage conçu comme un document qui passe en revue de manière systématique les bases et la syntaxe du langage C et que l'on consulte pour avoir une précision.

Chapitre 1. Les types de données de C

1.1. Notion de type de données

Une variable est caractérisée par

- l'endroit où se trouve stockée la variable en mémoire : c'est son *adresse*,
- la nature de l'information (nombre, caractère, …) : c'est son *type*,
- le contenu de la variable : c'est sa *valeur*.

Le C est un langage typé. Toutes les variables utilisées doivent être définies explicitement avant leur première utilisation. La déclaration d'une variable se fait par :

`Type identficateur [= valeur_initiale];`

`identificateur` est le nom donné à la variable (n'importe quel nom ne commençant pas par un chiffre). La pratique veut que l'on utilise des noms significatifs pour les variables. La valeur d'une variable peut être modifiée. Il est également possible de déclarer des constantes :

`Type const identifiateur = valeur;` ou

`const Type identificateur = valeur;`

1.2. Les types de base

1.2.1. Le type entier

`int`	:	entier signé sur 4 octets	;	`int i = -2;`
`short int`	:	entier signé sur 2 octets	;	`short int i = 2;`
`long int`	:	entier signé sur 4 octets	;	`long int i = 12l;`
`unsigned int`	:	entier non signé sur 4 octets	;	`unsigned int i = 12u;`
`unsigned short int`	:	entier non signé sur 2 octets	;	`unsigned short int i = 12u;`
`unsigned long int`	:	entier non signé sur 4 octets	;	`unsigned long int i = 12lu;`

- *les bases de représentation des entiers*

  base 10 (décimale) : `i = 12;`
  base 8 (octale) : `i = 075;`
  base 16 (hexadécimale) : `i = 0x45;`

- *les opérations sur les entiers*

`+`	:	l'addition
`-`	:	la soustraction
`*`	:	la multiplication
`/`	:	la division entière (le quotient)
`%`	:	le reste de la division entière
`a = b;`	:	l'affectation (a reçoit la valeur de b)

- *les opérations bit à bit*

`~`	:	NON logique bit à bit : (-8) $(11111000)_2$ donne 7 $(0000\ 0111)_2$
`&`	:	ET logique bit à bit
`|`	:	OU logique bit à bit
`^`	:	OU exclusif bit à bit
`<< n`	:	décalage à gauche de n bits
`>> n`	:	décalage à droite de n bits

1.2.2. Le type réel

`float`	:	réel en virgule flottante sur 4 octets	;	`float a = 7.4f, b = 1.2e2f;`
`double`	:	réel en double précision sur 8 otets	;	`double a = 7.4, b = 1.2e2;`
`long double`	:	réel sur 10 otets	;	`long double a = 7.4l, b = 1.2e2l;`

- *les opérations sur les réels*

`+`	:	l'addition
`-`	:	la soustraction
`*`	:	la multiplication
`/`	:	la division
`a = b;`	:	l'affectation (a reçoit la valeur de b)

Les fonctions mathématiques de base (sin, cos, tan, log, …) ne font pas partie intégrante du langage mais sont fournies avec la bibliothèque standard (voir le fichier math.h).

1.2.3. Le type caractère

Les caractères sont considérés comme des *entiers* codés sur 1 octet, et sont représentés par leur code ASCII.

`char`	:	caractère considéré comme un entier signé (-128 à 127)	;	`char a = 'a', b = 0x61;`
`unsigned char`	:	caractère considéré comme un entier non signé (0 à 255)	;	`unsigned char a = 'a', b = 0x61;`

- *les opérations sur les caractères* : toutes les opérations s'appliquant aux entiers peuvent également

s'appliquer aux caractères.

1.2.4. Les opérateurs arithmétiques raccourcis

`a = i++` : post incrémentation : `a` reçoit la valeur de `i`, puis `i` est incrémenté de 1

`a = ++i` : pré incrémentation[1] : `i` est incrémenté de 1 et `a` reçoit la valeur de `i`.

`a += b` : notation contractée pour écrire `a = a + b`. Le même type de notations existe avec les opérateurs `-=`, `*=`, `/=` et `%=`.

1.3. Le type tableau

Il permet d'associer sous un même nom un ensemble fini d'éléments de même type.

```
int mon_tableau[5] = {10, 20, 45, -15, 5};
```

`mon_tableau` est déclaré comme étant un tableau de 5 entiers (`int`).

mon_tableau	10	20	45	-15	5
indice	0	1	2	3	4

Pour accéder à un élément du tableau on utilise

```
a = mon_tableau[1];
```

`1` est l'indice de l'élément auquel on veut accéder. L'indice du premier élément est `0`. l'indice du dernier élément est `(taille du tableau - 1)`. Le compilateur ne vérifie pas les débordements d'indice …

On peut déterminer des tableaux à plusieurs dimensions. Par exemple

```
int mon_tableau[2][3] = {{1,2,3},{4,5,6}}
```

définit un tableau à deux dimensions (deux lignes et trois colonnes). On accède aux éléments en spécifiant les indices de ligne et de colonne

```
a = mon_tableau[i][j];
```

		indice colone j		
mon_tableau		0	1	2
indice ligne i	0	1	2	3
	1	4	5	6

Un tableau est obligatoirement de *taille fixe* définie lors de la déclaration de celui-ci.

- *Les opérations sur les tableaux* :

  aucune !!! En particulier, on ne peut pas affecter de manière globale les valeurs d'un tableau à un autre tableau. Il faut affecter élément par élément (voir la boucle *for* dans le chapitre suivant).

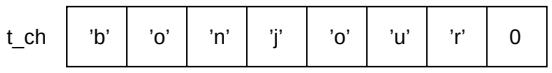
1.4. Les chaînes de caractères

Ce sont des tableaux de caractères. Le caractère nul (0) marque la fin de la chaîne.

---

[1] `a = i--;` ou `a = --i;` : même principe mais avec la décrémentation.

```
char t_ch[] = "bonjour";
```



le compilateur réserve un tableau de 8 octets (7 octets pour bonjour + 1 pour le caractère de fin de chaîne 0).

```
char t_ch[20] = "bonjour";
```

les 8 premiers caractères de `t_ch` sont initialisés, les 12 autres non.

- *Les opérations sur les chaînes de caractères* :

  aucune opération prédéfinie dans le langage (la chaîne de caractère est traitée comme un tableau). Cependant, la bibliothèque standard livrée avec les compilateurs répare ce problème (voir le fichier string.h).

1.5. Construction de types abstraits

1.5.1. la structure

C'est un type de donnée construit à partir d'une association de types différents.

```
struct Date {int jour, mois, annee;};
struct Fiche {
        char    nom[32];
        struct Date     naissance;
        float   taille;
};

/* Declaration de variables */
struct Fiche Mr_X = {"Toto",{1,2,1980},1.70};
struct Fiche Dupond;
```

L'accès à un des membres de la structure se fait par

```
float a = Mr_X.taille;
Mr_X.naissance.annee = 1982;
```

- *Les opérations sur les structures* : l'affectation de manière globale est possible.

  ```
  Dupond = Mr_X;
  ```

- *Les champs de bits* : Le plus petit élement spécifiable est de la taille d'un octet (type char). Il est néanmoins possible de nommer explicitement un bit ou un groupe de bits. On définit ainsi des champ de bits.

  ```
  struct bits {
          short   unbit           : 1,    /* le bit de poids le plus faible */
                  demioctet       : 4,
                                  : 5,    /* 5 bits non utilises */
                  triplet         : 3;
                                          /* les 3 derniers bits sont inutilises
  */
  };

  struct bits ex = {0,11,7};      /* unbit = 0, demioctet = 11, triplet = 7 */

  ex.unbit = 1;
  ```

1.5.2. L'union

L'union est un moyen pour associer plusieurs types à une même zone mémoire, et de choisir le type de la variable stockée dans cette zone au cas par cas. La taille de la zone mémoire réservée pour une variable de ce type correspond à l'espace mémoire nécessaire pour stoker le type le plus grand.

```
struct Date {jour, mois, annee}; /* 3*4 octets */
union ex_union {
        float   f;      /* 4 octets */
        char    c;      /* 1 octet */
        struct Date d;  /* 12 octets */
};

/* declaration des variables */
union ex_union a;

/* utilisation */

a.f = 7.2;
a.c = 'a';
a.d.annee = 1975;
```

La mémoire allouée est de 12 octets, ce qui permet de stocker, au chois, un réel, un caratère ou une date.

1.5.3. L'énumération

C'est un type permettant de définir un ensemble de constantes, parmi lesquelles les variables de ce type prendront leur valeur.

```
enum Jours {
        Lundi,
        Mardi,
        Mercredi,
        Jeudi
        Vendredi,
        Samedi,
        Dimanche
};

enum Jours jr = Mardi;
```

Chaque constante est associée à un entier (son numéro d'ordre en partant de 0 pour le premier). Mais on peut aussi spécifier la valeur associée

```
enum Jours {Lundi=1,Mardi,Mercredi=5,Jeudi,Vendredi,Samedi,Dimanche};
```

`Mardi` prend la valeur `2`, `Jeudi` la valeur `6`, …

Les valeurs associées aux constantes ne sont pas modifiables dans le programme.

1.6. Le type pointeur

un pointeur est une variable qui stocke l'adresse d'une autre variable (de type entier, réel, tableau, d'entier, structure, pointeur, …). Un pointeur sur une variable de type `"type"` est déclaré par

```
type * nom_pointeur [= valeur initiale];
```

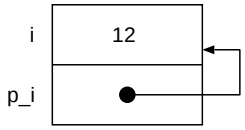
exemple

```
int * p_int;    /* pointeur sur un entier */
struct Date {int jour, mois, annee};
```

```
struct Date * p_date;   /* pointeur sur une date */
int ** p_p_int; /* pointeur sur (un pointeur sur (un entier)) */
```

Pour initialiser le pointeur, on lui affecte l'adresse d'une variable avec l'opérateur `&`

```
int i = 12;
int * p = &i;   /* p contient l'adresse de i */
```



Lorsqu'un pointeur est défini, mais qu'il ne pointe sur aucune zone mémoire en particulier, on lui affecte la valeur `NULL` c'est à dire 0.

```
int * p = NULL;
```

Pour déclarer un pointeur sur un type non défini, on utilise le type `void`

```
void * p = NULL:
```

- *Les opérations sur les pointeurs* : l'incrémentation, la décrémentation et l'affectation sont possibles. Lorsque l'on incrémente un pointeur sur un entier (codé sur 4 octets) de 1, on modifie en réalité l'adresse pointée de 1x (taille du type pointé), c'est à dire 4 dans le cas d'un pointeur sur un entier.

- *L'accès à la valeur de la variable pointée* peut être obtenu par l'opérateur `*`.

  ```
  int i = 12, a;
  int * p_i = &i;
  ...
  a = (*p_i) + 1; /* a va prendre la valeur 13 */
  ```

  Il faut bien sûr pour utiliser l'opérateur `*` que le pointeur contienne une adresse valide …

- *Les pointeurs et les tableaux* : le nom d'un tableau est considéré en C comme une constante pointeur sur le premier élément du tableau. Sa valeur n'est pas modifiable, néanmoins elle peut être utilisée pour initialiser un pointeur sur le premier élément du tableau.

  ```
  int table[5]={1,2,3,4,5}, * p_int = NULL;
  p_int = table;  /* p_int pointe sur le premier element de table */
  ```

  De la même manière, un pointeur sur le type `char` peut être initialisé grâce au nom d'une chaîne de caractères.

- Les pointeurs et les structures : pour accéder aux membres d'une structure, deux solutions sont envisageables : l'utilisation de l'opérateur `*` associé à l'opérateur `.`

  ```
  struct Date {int jout, mois, annee;};
  struct Date date = {1,2,1980}, * p_date = &date;
  (*p_date).jour = 5;
  ```

  ou l'utilisation de l'opérateur `->`

  ```
  struct Date {int jout, mois, annee;};
  struct Date date = {1,2,1980}, * p_date = &date;
  p_date->jour = 5;
  ```

1.7. L'instruction typedef

L'instruction `typedef` permet de définir des nouveaux types en renommant un type existant, une structure, un type pointé, …

```
typedef int Carotte;
```

```
typedef int ChouFleur;

Carotte   i = 2;
ChouFleur j = 3;

/* interdit d'ajouter des variables de type Carotte avec des variables de type
ChouFleur ! */

typedef struct _Date{int jour, mois, annee;} Date;
/*
 * Maintenant plus besoin de specifier struct pour les dates
 */
Date date = {1,2,1980};

date.jour = 5;
```

1.8. L'opérateur sizeof

L'opérateur `sizeof` prend en argument un type ou un nom de variable et renvoie la taille (en octets) de l'espace mémoire nécessaire pour stocker ce type ou cette variable.

```
int i = sizeof(int); /* i vaut 4 */
```

1.9. La conversion de type

La conversion de type ou le transtypage consiste à changer le type d'une variable (ce qui peut aboutir à une modifiation de la représentation interne).

- Elle est automatique vers un type occupant plus de place en mémoire pour les expressions numériques. Le transtypage est effectué implicitement par le compilateur : on peut ainsi effectuer la multiplication d'un entier par un réel sans se soucier du fait que les représentations machine sont incompatibles. Le transtypage implicite a lieu en suivant l'ordre :

  $$\texttt{(char)} \rightarrow \texttt{(short)} \rightarrow \texttt{(int)}, \texttt{(long)}, \texttt{(float)} \rightarrow \texttt{(double)} \rightarrow \texttt{(long double)}$$

  Par contre la conversion de `double` vers `int` n'est pas implicite.

- Elle peut être forcée

  ```
  double a = 2.2;
  int b = (int)a; /* b vaudra 2 */
  ```

Chapitre 2. Les structures de contrôle en C

2.1. Expression booléenne

Une variable booléenne est une variable qui peut prendre uniquement les valeurs vrai ou faux. Il n'y a pas de type booléen en C. Cependant, le langage suit la convention :

- faux = 0
- vrai = valeur différente de 0 (1 par défaut pour le compilateur).

Une expression booléenne est une relation dont l'évaluation donne un résultat booléen (ex : l'égalité entre deux variables peut être vérifiée ou pas). Les différents opérateurs de relation sont :

`<`	:	test inférieur à
`>`	:	test supérieur à
`<=`	:	test inférieur ou égal à
`>=`	:	test supérieur ou égal à
`==`	:	test égal à
`!=`	:	test différent de
`&&`	:	ET logique
`||`	:	OU logique
`!`	:	NON logique

2.2. Le test simple, if … else …

```
if (condition)
        instruction si la condition est vraie;
else
        instruction si la condition est fausse;
```

Exemple : trouver le maximum entre deux nombres

```
int i = 2, j = 3, max;

if (i>j)
        max = i;
else
        max = j;
```

Il est à remarquer qu'après le `if`, une seule instruction peut être mise. Si on veut exécuter plusieurs instructions dans le cas où la condition est vraie (ou fausse) il faut les inclure dans un bloc d'instructions délimité par { et }. Un bloc d'instruction est considéré par le C comme une seule instruction.

Il existe également une forme très condensée pour réaliser des tests qui fait usage de l'opérateur `?`.

```
condition ? instruction si vrai : instruction sinon
```

Ainsi, l'exemple précédent peut être codé par

```
int i = 2, int j = 3, max;
```

```
max = i>j ? i : j;
```

Cette forme est à utiliser avec modération. Elle ne contribue pas à la clarté des programmes.

2.3. La sélection, swith … case …

on utilise l'instruction switch lorsque l'on souhaite sélectionner une valeur parmi un ensemble défini de valeurs.

```
switch (variable)
{
        case valeur_1:
                instructions;
                break;  /* sortie du switch */
        case valeur_2:
                instructions;
                break;

        ...

        default:        /* autres cas */
                instructions;
}
```

Exemple : une petite calculatrice

```
char op = '+';
error = 0;
double a = 2.0, b = 3.0, resultat;

swith (op)
{
        case '+':
                resultat = a + b;
                break;  /* pour sortir du swith */
        case '-':
                resultat = a - b;
                break;
        case '*':
                resultat = a * b;
                break;
        case '/':
                resultat = a / b;
        default:
                error = 1;
}

/* on affiche le resultat */
printf("%g %c %g = %g",a,op,b,resultat);
```

2.4. La boucle while …

```
while (condition)
        instruction;
```

Tant que la condition est vraie (résultat non nul), l'instruction ou le bloc d'instructions entre { } est exécuté.

Exemple : chercher un nombre dans un tableau

```
int i = 0, t[5]={1,5,4,-1,2}, found = 0;
```

```
int n = 4;

while (i<5)
{
        if (t[i]==n)
        {
                found = 1; /* on a trouve … */
                break;  /* on sort du while */
        }
        i++;    /* incrementer l'indice */
}
if (found)
        printf("%d est le %d ieme element du tableau\n",n,i);
else
        printf("%d n'est pas dans le tableau\n");
```

2.5. La boucle do … while

```
do
        instruction
while (condition)
```

La condition n'est testée qu'en fin de boucle. Le bloc d'instructions situé dans la boucle sera donc éxécuté au moins une fois.

2.6. La boucle for

```
for (init;test;post)
        instruction;
```

L'expression init est exécuté à l'entrée dans la boucle `for`, puis le test est évalué. Si la condition est fausse, (résultat nul), on sort de la boucle. Si la condition est vraie, le bloc d'instructions de booucle est exécuté, puis l'expression post est évaluée. On teste à nouveau la condition, etc …

Exemple : afficher la table de multiplication de 7,

```
int i, n = 7;

for (i = 0 ; i <= 10 ; i++)
        printf("%d x %d = %d\n",n,i,n*i);
```

Exemple : chercher un nombre dans un tableau

```
int i, t[5]={1,5,4,-1,2}, found = 0;
int n = 4;

for (i=0;i<5;i++)
{
        if (t[i]==n)
        {
                found = 1; /* on a trouve … */
                break;  /* on sort du for */
        }
}
if (found)
        printf("%d est le %d ieme element du tableau\n",n,i);
else
        printf("%d n'est pas dans le tableau\n");
```

La boucle `for` est particulièrement bien adaptée pour parcourir les tableaux.

2.7. Les instructions break, continue et goto

L'instruction `break` utilisée dans une boucle provoque la sortie forcée de cette boucle. L'instruction `continue` provoque un saut en début de boucle. L'instruction `goto` provoque un saut à l'étiquette spécifiée.

```
etiquette:
...
goto etiquette;
```

Chapitre 3. Les fonctions en C

3.1. Concept

Une fonction permet de donner un nom et des paramètres à un bloc d'instructions délimité par { et }. Dans un bloc d'instructions, on peut trouver des déclarations de variables (au début) et des instructions.

```
type_retour nom_fonction(liste arguments)
{
        /* declaration de variables */

        ...

        /* instructions */
}
```

Les fonctions C renvoient généralement un paramètre dont le type est `type_retour` grâce à l'instruction `return`. Lorsqu'une fontion ne renvoie aucun paramètre, le type est `void`.

```
int add(int a, int b)
{
        return a+b;
}

...

int i = add(2,5);
```

Le point d'entrée d'un programme en C est la fonction `main`.

Une fonction ne peut pas être définie dans le corps (blo d'instructions) d'une autre fonction.

3.2. Modes de passage des arguments

Le passage des arguments à une fonction lors de son appel est réalisé grâce à la pile. Il peu s'effectuer de deux manières :

- *Passage d'arguments par valeurs* : La valeur de la variable passée en argument est recopiée sur la pile pour qu'elle soit utilisée par la fonction. La fonction n'a donc pas accès à la variable elle-même et ne peut donc pas la modifier. C'est sécurisant pour les variables, mais cela peut devenir pénalisant si on recopie des variables qui occupent de grandes zones mémoire…

- *Passage d'arguments par référene ou par adresse* : Cela consiste à fournir à la fonction, non pas une copie de la valeur de la variable passée en argument, mais l'adresse de cette variable. La fonction a donc accès directement à l'emplacement mémoire où est stockée la variable, et peut donc modifier la valeur de la variable. Cela peut aboutir à des modifications non désirées si on utilise mal cette possibilité, mais cela peut permettre également de n'avoir à recopier qu'une adresse (4 octets) au lieu d'une quantité de mémoire plus importante.

  Le passage d'arguments par référence ne fait pas, à proprement parler, partie du langage C. On contourne cependant le problème en passant par valeur un pointeur sur la variable.

  ```
  int add(int *a, int *b) /* les parametres sont des pointeurs sur des entiers */
  {
          return *a + *b;
  }
  ```

```
int c = 2, d = 3;

/* on passe en arguments les adresses de c et de d */
int i = add(&c,&b);
```

Lorsque l'on passe un tableau en argument à une fonction, c'est toujours son adresse qui est recopiée et pas le tableau tout entier. De même, l'instruction `return` ne peut renvoyer qu'un pointeur sur un tableau.

3.3. Statut et portée des identificateurs

Les règles de portée des identificateurs (nom de variable, type, fonction) permettent de répondre à la question : si je déclare une variable (type, fonction) à cet endroit du programme, dans quelle mesure est-ce que je pourrai l'utiliser ? Ces règles de portée qui régissent la visibilité des variables sont :

- Les identificateurs ne sont reconnus qu'à partir de l'endroit où se trouve leur définition.
- Lorsqu'un identificateur n'est reconnu qu'à l'intérieur d'un bloc d'instructions entre accolades, il n'est reconnu qu'à l'intérieur de ce bloc. Une telle variable est dite *locale*. En particulier, le corps d'une fonction est constitué comme un bloc.
- Lorsqu'un identificateur est redéfini dans un sous bloc, la nouvelle définition remplace la précédente dans ce sous bloc.
- Lorsqu'un identificateur est défini en dehors de tout bloc, cet identificateur est visible pour l'ensemble des fichiers du projet (sauf s'il a le statut `static`). Une telle variable est dite *globale*. Les fonctions sont visibles dans l'ensemble des fichiers.
- Les identificateurs locaux sont toujours prépondérants sur les identificateurs globaux, en cas de conflit de nom.

En plus de ces règles, on peut spécifier des statuts particuliers :

- `register` : permet de demander au compilateur de stocker la variable précédée de ce statut dans un des registres du microprocesseur.
- `const` : permet de déclarer des variables qui ne pourront pas être modifiées dans le programme.
- `extern` : permet de faire référence à des variables, fontions, … définies dans un autre fichier.
- `static` : permet de conserver la valeur d'une variable, même si elle n'est reconnue qu'à l'intérieur du bloc où elle a été spécifiée. Dans le cas d'une fonction, cela limite la portée de la fonction au seul fichier dans lequel il a été défini.

3.4. La récursivité

Une fonction peut être récursive, c'est à dire qu'elle peut s'appeler elle-même. Cet aspect est à utiliser avec attention ; en particulier, il faut prévoir une condition pour laquelle la fonction arrête de s'appeler, sinon le programme reste bloqué dans le corps de cette fonction.

exemple : fonction factorielle n! = n*(n-1)*…*2*1= n*(n-1)! avec 0! = 1

```
int fact(int n)
{
        if (n!=0)
                return n*fact(n-1); /* appel recursif de fact */
        else
                return 1;
}
```

La fonction s'appelle jusqu'à ce que l'argument soit égal à 0.

3.5. Les pointeurs de fonction

Un pointeur de fonction est une variable qui permet de stocker l'adresse d'une fonction.

```
int add(int a, int b)
{
        return a+b;
}

int (*p_add)(int,int);

p_add = add;

int i = (*p_add)(2,3);
```

Il est aussi possible d'avoir des membres de structures qui sont des pointeurs sur des fonctions.

Chapitre 4. Les directives de compilation

Ce sont des directives prises en compte avant la phase de compilation par le préprocesseur C. Ces directives permettent d'indiquer les fichiers à inclure, de définir des portions de codes à compiler sous certaines conditions, d'écrire des macros, …

- `#include <nom de fichier>` : cette directive permet d'insérer le fichier spécifié dans le fichier courant avant la compilation. l'usage de `< >` autour du nom du fichier à inclure indique qu'il faut le chercher dans le répertoire standard du compilateur. Si le nom du fichier est entouré par `" "`, le chemin complet doit être indiqué – à défaut de chemin, le répertoire de travail est pris.

- `#define PI 3.14159` : cette directive permet de définir des constantes symboliques. Partout où `PI` est utilisé dans le code, le préprocesseur le remplacera par la valeur associée. Il est également possible d'annuler la définition d'un symbole grâce à la directive `#undef`.

- `#define CARRE(x) ((x)*(x))` : cette directive permet de définir des macros avec autant de paramètres que l'on veut. Si le préprocesseur rencontre dans le fichier `CARRE(2)`, il remplacera le code par `((2)*(2))`. Ces macros peuvent s'avérer utiles dans certaines situations. Attention toutefois, aucun contrôle quant à la compatibilité des types n'est réalisé ; c'est lors de la compilation qu'une éventuelle erreur sera détectée.

- ```
  #if defined(TOTO)
  ...
  #elif defined
  ...
  #else
  ...
  #endif
  ```

  les directives `#if`, `#elif`, `#else` et `#endif` permettent de faire de la compilation conditionnelle. On peut ainsi de compiler uniquement le code adéquat. La macro `defined` permet de savoir si un symbole a été défini par la directive `#define`. Associé à la directive `#if`, on peut utiliser les formes condensées `#ifdef(TOTO)` ou pour savoir si le symbôle n'est pas défini `#ifndef(TOTO)`.

  Chaque système d'exploitation définit des macros qui le caractérisent (WIN32 pour Windows 95, LINUX pour Linux, BEOS pour BeOS, …). Ces directives permettent donc d'inclure dans un même fichier du code spécifique pour chacun de ces systèmes

  ```
  #ifdef(WIN32)
          /* code pour Windows */
  #elif defined(LINUX)
          /* code pour LINUX */
  #else
          /* code pour les autres syste/mes */
  #endif
  ```

  Ces directives sont également employées lorsqu'un fichier est susceptible d'être inclu dans plusieurs autres fichiers – c'est le cas des fichiers d'entête (en .h). L'inclusion multiple d'un fichier aboutit à une erreur de compilation (fonctions avec le même nom définie plus d'une fois …). Pour éviter ce problème, on encapsule le contenu du fichier (on va considérer le fichier toto.h) dans la construction suivante :

  ```
  #ifndef TOTO_H
  #define TOTO_H
  /* contenu du fichier */
  #endif
  ```
  ```

La première fois que le fichier est inclu dans un autre, le symbole TOTO_H est défini, permettant ainsi de n'inclure qu'une seule fois ce fichier dans le projet.

Index

Adresse, 3
Affectation, 3
ASCII, 4

Booléen, 10
Boucle
  do … while, 12
  for, 12
  while, 11
break, 13

Caractère, 4
case, 11
Chaîne de caractères, 5
char, 4
continue, 13
Conversion de type, 9

Directives de compilation
  #define, 17
  defined, 17
  #elif, 17
  #else, 17
  #if, 17
  #ifdef, 17
  #ifndef, 17
  #include, 17
  #undef, 17
do, 12
double, 4

else, 10
Entier, 3
Enumération, 7

float, 4
Fonction
  Arguments, 14
  Concept, 14
  Passage d'arguments par référence, 14
  Passage d'arguments par valeur, 14
  Pointeur de, 16
  Récursivité, 15
  return, 14
  void, 14
for, 12

goto, 13

if, 10
int, 3

long, 3

main, 14

Opérateur ?, 10

Pointeur, 7
Portée des identificateurs, 15

Réel, 4
return, 14

Sélection multiple, 11
short, 3
sizeof, 9
Structure, 6
switch, 11

Tableau, 5
Type, 3
typedef, 8

unsigned, 3

Variable, 3
void, 14

while, 11