Programmation en langage C

Anne CANTEAUT

INRIA - projet CODES
B.P. 105
78153 Le Chesnay Cedex
`Anne.Canteaut@inria.fr`
`http://www-rocq.inria.fr/codes/Anne.Canteaut/COURS_C`

Table des matières

Liste des tableaux

Chapitre 1

Les bases de la programmation en C

1.1 Historique

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language* [6]. Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

1.2 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*. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé *compilateur*. La compilation se décompose en fait en 4 phases successives :

1. **Le traitement par le préprocesseur :** le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).

2. **La compilation :** la compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.

3. **L'assemblage :** cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé *fichier objet*.

4. **L'édition de liens :** un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit *exécutable*.

Les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe. Les fichiers source sont suffixés par `.c`, les fichiers prétraités par le préprocesseur par `.i`, les fichiers assembleur par `.s`, et les fichiers objet par `.o`. Les fichiers objets correspondant aux librairies pré-compilées ont pour suffixe `.a`.

Le compilateur C sous UNIX s'appelle `cc`. On utilisera de préférence le compilateur `gcc` du projet GNU. Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, `gcc` active toutes les étapes de la compilation. On le lance par la commande

`gcc [`*`options`*`] fichier.c [-l`*`librairies`*`]`

Par défaut, le fichier exécutable s'appelle `a.out`. Le nom de l'exécutable peut être modifié à l'aide de l'option `-o`.

Les éventuelles librairies sont déclarées par la chaîne `-l`*`librairie`*. Dans ce cas, le système recherche le fichier `lib`*`librairie`*`.a` dans le répertoire contenant les librairies pré-compilées (généralement `/usr/lib/`). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie `-lm`. Le fichier objet correspondant est `libm.a`. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option `-L`.

Les options les plus importantes du compilateur `gcc` sont les suivantes :

`-c` : supprime l'édition de liens ; produit un fichier objet.

`-E` : n'active que le préprocesseur (le résultat est envoyé sur la sortie standard).

`-g` : produit des informations symboliques nécessaires au débogueur.

`-I`*`nom-de-répertoire`* : spécifie le répertoire dans lequel doivent être recherchés les fichiers en-têtes à inclure (en plus du répertoire courant).

`-L`*`nom-de-répertoire`* : spécifie le répertoire dans lequel doivent être recherchées les librairies précompilées (en plus du répertoire usuel).

`-o` *`nom-de-fichier`* : spécifie le nom du fichier produit. Par défaut, le exécutable fichier s'appelle `a.out`.

`-O`, `-O1`, `-O2`, `-O3` : options d'optimisations. Sans ces options, le but du compilateur est de minimiser le coût de la compilation. En rajoutant l'une de ces options, le compilateur tente de réduire la taille du code exécutable et le temps d'exécution. Les options correspondent à différents niveaux d'optimisation : `-O1` (similaire à `-O`) correspond à une faible optimisation, `-O3` à l'optimisation maximale.

`-S` : n'active que le préprocesseur et le compilateur ; produit un fichier assembleur.

`-v` : imprime la liste des commandes exécutées par les différentes étapes de la compilation.

`-W` : imprime des messages d'avertissement (warning) supplémentaires.

`-Wall` : imprime tous les messages d'avertissement.

Pour plus de détails sur `gcc`, on peut consulter le chapitre 4 de [8].

1.3 Les composants élémentaires du C

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

- les identificateurs,
- les mots-clefs,
- les constantes,
- les chaînes de caractères,
- les opérateurs,
- les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

1.3.1 Les identificateurs

Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

- un nom de variable ou de fonction,
- un type défini par `typedef`, `struct`, `union` ou `enum`,
- une étiquette.

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le "blanc souligné" (_).

Le premier caractère d'un identificateur ne peut pas être un chiffre. Par exemple, `var1`, `tab_23` ou `_deb` sont des identificateurs valides ; par contre, `1i` et `i:j` ne le sont pas. Il est cependant déconseillé d'utiliser _ comme premier caractère d'un identificateur car il est souvent employé pour définir les variables globales de l'environnement C.

Les majuscules et minuscules sont differenciées.

Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Cette limite dépend des implémentations, mais elle est toujours supérieure à 31 caractères. (Le standard dit que les identificateurs externes, c'est-à-dire ceux qui sont exportés à l'édition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères).

1.3.2 Les mots-clefs

Un certain nombre de mots, appelés *mots-clefs*, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs :

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

que l'on peut ranger en catégories

- les spécificateurs de stockage

  `auto   register   static   extern   typedef`

- les spécificateurs de type

  `char   double   enum   float   int   long   short   signed   struct`
  `union   unsigned   void`

- les qualificateurs de type

  `const   volatile`

- les instructions de contrôle

  `break   case   continue   default   do   else   for   goto   if`
  `switch   while`

- divers

  `return   sizeof`

1.3.3 Les commentaires

Un commentaire débute par `/*` et se termine par `*/`. Par exemple,

```
/*  Ceci est un commentaire */
```

On ne peut pas imbriquer des commentaires. Quand on met en commentaire un morceau de programme, il faut donc veiller à ce que celui-ci ne contienne pas de commentaire.

1.4 Structure d'un programme C

Une *expression* est une suite de composants élémentaires syntaxiquement correcte, par exemple
`x = 0`
ou bien

```
(i >= 0) && (i < 10) && (p[i] != 0)
```

Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression". Plusieurs instructions peuvent être rassemblées par des accolades { et } pour former une *instruction composée* ou *bloc* qui est syntaxiquement équivalent à une instruction. Par exemple,

```
if (x != 0)
{
  z = y / x;
  t = y % x;
}
```

Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés par une virgule est une *déclaration*. Par exemple,

```
int a;
int b = 1, c;
double x = 2.38e4;
char message[80];
```

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Un programme C se présente de la façon suivante :

```
[ directives au préprocesseur]
[ déclarations de variables externes]
[ fonctions secondaires]

main()
{
   déclarations de variables internes
   instructions
}
```

La fonction principale `main` peut avoir des paramètres formels. On supposera dans un premier temps que la fonction `main` n'a pas de valeur de retour. Ceci est toléré par le compilateur mais produit un message d'avertissement quand on utilise l'option `-Wall` de `gcc` (*cf.* page 67).

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale. Une fonction secondaire peut se décrire de la manière suivante :

```
 type ma_fonction (  arguments )
{
   déclarations de variables internes
   instructions
}
```

Cette fonction retournera un objet dont le type sera ***type*** (à l'aide d'une instruction comme `return` ***objet***;). Les ***arguments*** de la fonction obéissent à une syntaxe voisine de celle des déclarations : on met en argument de la fonction une suite d'expressions ***type*** `objet` séparées par des virgules. Par exemple, la fonction secondaire suivante calcule le produit de deux entiers :

```
int produit(int a, int b)
{
  int resultat;

  resultat = a * b;
  return(resultat);
}
```

1.5 Les types prédéfinis

Le C est un langage *typé*. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.

La mémoire de l'ordinateur se décompose en une suite continue d'octets. Chaque octet de la mémoire est caractérisé par son *adresse*, qui est un entier. Deux octets contigus en mémoire ont des adresses qui diffèrent d'une unité. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type.

La taille mémoire correspondant aux différents types dépend des compilateurs ; toutefois, la norme ANSI spécifie un certain nombre de contraintes.

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :

```
char
int
float  double
short  long    unsigned
```

1.5.1 Le type caractère

Le mot-clef `char` désigne un objet de type caractère. Un `char` peut contenir n'importe quel élément du jeu de caractères de la machine utilisée. La plupart du temps, un objet de type `char` est codé sur un octet ; c'est l'objet le plus élémentaire en C. Le jeu de caractères utilisé correspond généralement au codage ASCII (sur 7 bits). La plupart des machines utilisent désormais le jeu de caractères ISO-8859 (sur 8 bits), dont les 128 premiers caractères correspondent aux caractères ASCII. Les 128 derniers caractères (codés sur 8 bits) sont utilisés pour les caractères propres aux différentes langues. La version ISO-8859-1 (aussi appelée ISO-LATIN-1) est utilisée pour les langues d'Europe occidentale. Ainsi, le caractère de code 232 est le è, le caractère 233 correspond au é... Pour plus de détails sur l'historique du codage des caractères pour les différentes langues ainsi que sur la norme UNICODE (sur 16 bits, qui permet de coder les caractères pour toutes les langues) et sur la norme ISO/IEC-10646 (sur 32 bits, ce qui permet d'ajouter les caractères anciens), consulter l'article de J. André et M. Goossens [1].

<table>
<tr><th></th><th>déc.</th><th>oct.</th><th>hex.</th><th></th><th>déc.</th><th>oct.</th><th>hex.</th><th></th><th>déc.</th><th>oct.</th><th>hex.</th></tr>
<tr><td></td><td>32</td><td>40</td><td>20</td><td>@</td><td>64</td><td>100</td><td>40</td><td>‘</td><td>96</td><td>140</td><td>60</td></tr>
<tr><td>!</td><td>33</td><td>41</td><td>21</td><td>A</td><td>65</td><td>101</td><td>41</td><td>a</td><td>97</td><td>141</td><td>61</td></tr>
<tr><td>”</td><td>34</td><td>42</td><td>22</td><td>B</td><td>66</td><td>102</td><td>42</td><td>b</td><td>98</td><td>142</td><td>62</td></tr>
<tr><td>#</td><td>35</td><td>43</td><td>23</td><td>C</td><td>67</td><td>103</td><td>43</td><td>c</td><td>99</td><td>143</td><td>63</td></tr>
<tr><td>$</td><td>36</td><td>44</td><td>24</td><td>D</td><td>68</td><td>104</td><td>44</td><td>d</td><td>100</td><td>144</td><td>64</td></tr>
<tr><td>%</td><td>37</td><td>45</td><td>25</td><td>E</td><td>69</td><td>105</td><td>45</td><td>e</td><td>101</td><td>145</td><td>65</td></tr>
<tr><td>&</td><td>38</td><td>46</td><td>26</td><td>F</td><td>70</td><td>106</td><td>46</td><td>f</td><td>102</td><td>146</td><td>66</td></tr>
<tr><td>’</td><td>39</td><td>47</td><td>27</td><td>G</td><td>71</td><td>107</td><td>47</td><td>g</td><td>103</td><td>147</td><td>67</td></tr>
<tr><td>(</td><td>40</td><td>50</td><td>28</td><td>H</td><td>72</td><td>110</td><td>48</td><td>h</td><td>104</td><td>150</td><td>68</td></tr>
<tr><td>)</td><td>41</td><td>51</td><td>29</td><td>I</td><td>73</td><td>111</td><td>49</td><td>i</td><td>105</td><td>151</td><td>69</td></tr>
<tr><td>*</td><td>42</td><td>52</td><td>2a</td><td>J</td><td>74</td><td>112</td><td>4a</td><td>j</td><td>106</td><td>152</td><td>6a</td></tr>
<tr><td>+</td><td>43</td><td>53</td><td>2b</td><td>K</td><td>75</td><td>113</td><td>4b</td><td>k</td><td>107</td><td>153</td><td>6b</td></tr>
<tr><td>,</td><td>44</td><td>54</td><td>2c</td><td>L</td><td>76</td><td>114</td><td>4c</td><td>l</td><td>108</td><td>154</td><td>6c</td></tr>
<tr><td>-</td><td>45</td><td>55</td><td>2d</td><td>M</td><td>77</td><td>115</td><td>4d</td><td>m</td><td>109</td><td>155</td><td>6d</td></tr>
<tr><td>.</td><td>46</td><td>56</td><td>2e</td><td>N</td><td>78</td><td>116</td><td>4e</td><td>n</td><td>110</td><td>156</td><td>6e</td></tr>
<tr><td>/</td><td>47</td><td>57</td><td>2f</td><td>O</td><td>79</td><td>117</td><td>4f</td><td>o</td><td>111</td><td>157</td><td>6f</td></tr>
<tr><td>0</td><td>48</td><td>60</td><td>30</td><td>P</td><td>80</td><td>120</td><td>50</td><td>p</td><td>112</td><td>160</td><td>70</td></tr>
<tr><td>1</td><td>49</td><td>61</td><td>31</td><td>Q</td><td>81</td><td>121</td><td>51</td><td>q</td><td>113</td><td>161</td><td>71</td></tr>
<tr><td>2</td><td>50</td><td>62</td><td>32</td><td>R</td><td>82</td><td>122</td><td>52</td><td>r</td><td>114</td><td>162</td><td>72</td></tr>
<tr><td>3</td><td>51</td><td>63</td><td>33</td><td>S</td><td>83</td><td>123</td><td>53</td><td>s</td><td>115</td><td>163</td><td>73</td></tr>
<tr><td>4</td><td>52</td><td>64</td><td>34</td><td>T</td><td>84</td><td>124</td><td>54</td><td>t</td><td>116</td><td>164</td><td>74</td></tr>
<tr><td>5</td><td>53</td><td>65</td><td>35</td><td>U</td><td>85</td><td>125</td><td>55</td><td>u</td><td>117</td><td>165</td><td>75</td></tr>
<tr><td>6</td><td>54</td><td>66</td><td>36</td><td>V</td><td>86</td><td>126</td><td>56</td><td>v</td><td>118</td><td>166</td><td>76</td></tr>
<tr><td>7</td><td>55</td><td>67</td><td>37</td><td>W</td><td>87</td><td>127</td><td>57</td><td>w</td><td>119</td><td>167</td><td>77</td></tr>
<tr><td>8</td><td>56</td><td>70</td><td>38</td><td>X</td><td>88</td><td>130</td><td>58</td><td>x</td><td>120</td><td>170</td><td>78</td></tr>
<tr><td>9</td><td>57</td><td>71</td><td>39</td><td>Y</td><td>89</td><td>131</td><td>59</td><td>y</td><td>121</td><td>171</td><td>79</td></tr>
<tr><td>:</td><td>58</td><td>72</td><td>3a</td><td>Z</td><td>90</td><td>132</td><td>5a</td><td>z</td><td>122</td><td>172</td><td>7a</td></tr>
<tr><td>;</td><td>59</td><td>73</td><td>3b</td><td>[</td><td>91</td><td>133</td><td>5b</td><td>{</td><td>123</td><td>173</td><td>7b</td></tr>
<tr><td><</td><td>60</td><td>74</td><td>3c</td><td>\</td><td>92</td><td>134</td><td>5c</td><td>|</td><td>124</td><td>174</td><td>7c</td></tr>
<tr><td>=</td><td>61</td><td>75</td><td>3d</td><td>]</td><td>93</td><td>135</td><td>5d</td><td>}</td><td>125</td><td>175</td><td>7d</td></tr>
<tr><td>></td><td>62</td><td>76</td><td>3e</td><td>^</td><td>94</td><td>136</td><td>5e</td><td>~</td><td>126</td><td>176</td><td>7e</td></tr>
<tr><td>?</td><td>63</td><td>77</td><td>3f</td><td>_</td><td>95</td><td>137</td><td>5f</td><td>DEL</td><td>127</td><td>177</td><td>7f</td></tr>
</table>

TAB. 1.1 – *Codes ASCII des caractères imprimables*

Une des particularités du type `char` en C est qu'il peut être assimilé à un entier : tout objet de type `char` peut être utilisé dans une expression qui utilise des objets de type entier. Par exemple, si `c` est de type `char`, l'expression `c + 1` est valide. Elle désigne le caractère suivant dans le code ASCII. La table de la page 15 donne le code ASCII (en décimal, en octal et en hexadécimal) des caractères imprimables. Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
  char c = 'A';
```

```
  printf("%c", c + 1);
}
```

Suivant les implémentations, le type `char` est signé ou non. En cas de doute, il vaut mieux préciser `unsigned char` ou `signed char`. Notons que tous les caractères imprimables sont positifs.

1.5.2 Les types entiers

Le mot-clef désignant le type entier est `int`. Un objet de type `int` est représenté par un mot "naturel" de la machine utilisée, 32 bits pour un DEC alpha ou un PC Intel.

Le type `int` peut être précédé d'un attribut de précision (`short` ou `long`) et/ou d'un attribut de représentation (`unsigned`). Un objet de type `short int` a au moins la taille d'un `char` et au plus la taille d'un `int`. En général, un `short int` est codé sur 16 bits. Un objet de type `long int` a au moins la taille d'un `int` (64 bits sur un DEC alpha, 32 bits sur un PC Intel).

	DEC Alpha	PC Intel (Linux)	
`char`	8 bits	8 bits	caractère
`short`	16 bits	16 bits	entier court
`int`	32 bits	32 bits	entier
`long`	64 bits	32 bits	entier long
`long long`	n.i.	64 bits	entier long (non ANSI)

TAB. 1.2 – *Les types entiers*

Le bit de poids fort d'un entier est son signe. Un entier positif est donc représenté en mémoire par la suite de 32 bits dont le bit de poids fort vaut 0 et les 31 autres bits correspondent à la décomposition de l'entier en base 2. Par exemple, pour des objets de type `char` (8 bits), l'entier positif 12 sera représenté en mémoire par 00001100. Un entier négatif est, lui, représenté par une suite de 32 bits dont le bit de poids fort vaut 1 et les 31 autres bits correspondent à la valeur absolue de l'entier représentée suivant la technique dite du "complément à 2". Cela signifie que l'on exprime la valeur absolue de l'entier sous forme binaire, que l'on prend le complémentaire bit-à-bit de cette valeur et que l'on ajoute 1 au résultat. Ainsi, pour des objets de type `signed char` (8 bits), -1 sera représenté par 11111111, -2 par 11111110, -12 par par 11110100. Un `int` peut donc représenter un entier entre -2^{31} et $(2^{31} - 1)$. L'attribut `unsigned` spécifie que l'entier n'a pas de signe. Un `unsigned int` peut donc représenter un entier entre 0 et $(2^{32} - 1)$. Sur un DEC alpha, on utilisera donc un des types suivants en fonction de la taille des données à stocker :

`signed char`	$[-2^7; 2^7[$
`unsigned char`	$[0; 2^8[$
`short int`	$[-2^{15}; 2^{15}[$
`unsigned short int`	$[0; 2^{16}[$
`int`	$[-2^{31}; 2^{31}[$
`unsigned int`	$[0; 2^{32}[$
`long int` (DEC alpha)	$[-2^{63}; 2^{63}[$
`unsigned long int` (DEC alpha)	$[0; 2^{64}[$

Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard `limits.h`.

Le mot-clef `sizeof` a pour syntaxe

`sizeof(`*`expression`*`)`

où *`expression`* est un type ou un objet. Le résultat est un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet. Par exemple

```
unsigned short x;

taille = sizeof(unsigned short);
taille = sizeof(x);
```

Dans les deux cas, `taille` vaudra 4.

Pour obtenir des programmes portables, on s'efforcera de ne jamais présumer de la taille d'un objet de type entier. On utilisera toujours une des constantes de `limits.h` ou le résultat obtenu en appliquant l'opérateur `sizeof`.

1.5.3 Les types flottants

Les types `float`, `double` et `long double` servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

	DEC Alpha	PC Intel	
`float`	32 bits	32 bits	flottant
`double`	64 bits	64 bits	flottant double précision
`long double`	64 bits	128 bits	flottant quadruple précision

TAB. 1.3 – *Les types flottants*

Les flottants sont généralement stockés en mémoire sous la représentation de la virgule flottante normalisée. On écrit le nombre sous la forme "signe $0,\text{mantisse}\ B^{\text{exposant}}$". En général, $B = 2$. Le digit de poids fort de la mantisse n'est jamais nul.

Un flottant est donc représenté par une suite de bits dont le bit de poids fort correspond au signe du nombre. Le champ du milieu correspond à la représentation binaire de l'exposant alors que les bits de poids faible servent à représenter la mantisse.

1.6 Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, énumération. Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

1.6.1 Les constantes entières

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

- **décimale :** par exemple, `0` et `2437348` sont des constantes entières décimales.
- **octale :** la représentation octale d'un entier correspond à sa décomposition en base 8. Les constantes octales doivent commencer par un zéro. Par exemple, les représentations octales des entiers 0 et 255 sont respectivement `00` et `0377`.
- **hexadécimale :** la représentation hexadécimale d'un entier correspond à sa décomposition en base 16. Les lettres de `a` à `f` sont utilisées pour représenter les nombres de 10 à 15. Les constantes hexadécimales doivent commencer par `0x` ou `0X`. Par exemple, les représentations hexadécimales de 14 et 255 sont respectivement `0xe` et `0xff`.

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` et `unsigned long int` tandis qu'une constante octale ou hexadécimale est représentée avec le format interne le plus court permettant encore de la représenter parmi les formats des types `int`, `unsigned int`, `long int` et `unsigned long int`.

On peut cependant spécifier explicitement le format d'une 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`. Par exemple :

constante	type
`1234`	`int`
`02322`	`int` /* octal */
`0x4D2`	`int` /* hexadécimal */
`123456789L`	`long`
`1234U`	`unsigned int`
`123456789UL`	`unsigned long int`

1.6.2 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 décimal éventuellement signé.

Par défaut, une constante réelle est représentée avec le format du type `double`. On peut cependant influer sur la représentation interne de la constante en lui ajoutant un des suffixes `f` (indifféremment `F`) ou `l` (indifféremment `L`). Les suffixes `f` et `F` forcent la représentation de la constante sous forme d'un `float`, les suffixes `l` et `L` forcent la représentation sous forme d'un `long double`. Par exemple :

constante	type
`12.34`	`double`
`12.3e-4`	`double`
`12.34F`	`float`
`12.34L`	`long double`

1.6.3 Les constantes caractères

Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. `'A'` ou `'$'`). Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par `\\` et `\'`. Le point d'interrogation et les guillemets peuvent aussi être désignés par les notations `\?` et `\"`. Les caractères non imprimables peuvent être désignés par `'\`*`code-octal`*`'` où *`code-octal`* est le code en octal du caractère. On peut aussi écrire `'\x`*`code-hexa`*`'` où *`code-hexa`* est le code en hexadécimal du caractère (*cf.* page 15). Par exemple, `'\33'` et `'\x1b'` désignent le caractère *escape*. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

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

1.6.4 Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple,

```
"Ceci est une chaîne de caractères"
```

Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple,

```
"ligne 1 \n ligne 2"
```

A l'intérieur d'une chaîne de caractères, le caractère `"` doit être désigné par `\"`. Enfin, le caractère `\` suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple,

```
"ceci est une longue longue  longue longue longue longue longue longue \
chaîne de caractères"
```

1.7 Les opérateurs

1.7.1 L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe `=`. Sa syntaxe est la suivante :

`variable = expression`

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer *`expression`* et d'affecter la valeur obtenue à *`variable`*. De plus, cette expression possède une valeur, qui est celle *`expression`*. Ainsi, l'expression `i = 5` vaut 5.

L'affectation effectue une *conversion de type implicite* : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant

```
main()
{
  int i, j = 2;
  float x = 2.5;
  i = j + x;
  x = x + i;
  printf("\n %f \n",x);
}
```

imprime pour `x` la valeur 6.5 (et non 7), car dans l'instruction `i = j + x;`, l'expression `j + x` a été convertie en entier.

1.7.2 Les opérateurs arithmétiques

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

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- Contrairement à d'autres langages, le C ne dispose que de la notation `/` pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur `/` produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple,

  ```
  float x;
  x = 3 / 2;
  ```

  affecte à `x` la valeur 1. Par contre

  ```
  x = 3 / 2.;
  ```

  affecte à `x` la valeur 1.5.

- L'opérateur `%` ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction `pow(x,y)` de la librairie `math.h` pour calculer x^y.

1.7.3 Les opérateurs relationnels

`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

Leur syntaxe est

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type `int` (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.

Attention à ne pas confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affection `=`. Ainsi, le programme

```
main()
{
  int a = 0;
  int b = 1;
  if (a = b)
    printf("\n a et b sont egaux \n");
  else
    printf("\n a et b sont differents \n");
}
```

imprime à l'écran `a et b sont egaux`!

1.7.4 Les opérateurs logiques booléens

`&&`	et logique
`||`	ou logique
`!`	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un `int` qui vaut 1 si la condition est vraie et 0 sinon.

Dans une expression de type

expression-1 op-1 expression-2 op-2 ...expression-n

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans

```
int i;
int p[10];

if ((i >= 0) && (i <= 9) && !(p[i] == 0))
...
```

la dernière clause ne sera pas évaluée si `i` n'est pas entre 0 et 9.

1.7.5 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (`short`, `int` ou `long`), signés ou non.

`&`	et	`|`	ou inclusif
`^`	ou exclusif	`~`	complément à 1
`<<`	décalage à gauche	`>>`	décalage à droite

En pratique, les opérateurs `&`, `|` et `~` consistent à appliquer bit à bit les opérations suivantes

`&`	0	1
0	0	0
1	0	1

`|`	0	1
0	0	1
1	1	1

`^`	0	1
0	0	1
1	1	0

L'opérateur unaire `~` change la valeur de chaque bit d'un entier. Le décalage à droite et à gauche effectuent respectivement une multiplication et une division par une puissance de 2. Notons que ces décalages ne sont pas des décalages circulaires (ce qui dépasse disparaît).

Considérons par exemple les entiers `a=77` et `b=23` de type `unsigned char` (*i.e.* 8 bits). En base 2 il s'écrivent respectivement 01001101 et 00010111.

	valeur		
expression	binaire	décimale	
`a`	01001101	77	
`b`	00010111	23	
`a & b`	00000101	5	
`a | b`	01011111	95	
`a ^ b`	01011010	90	
`~a`	10110010	178	
`b << 2`	01011100	92	multiplication par 4
`b << 5`	11100000	112	ce qui dépasse disparaît
`b >> 1`	00001011	11	division entière par 2

1.7.6 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

`+=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=`

Pour tout opérateur *`op`*, l'expression

`expression-1 op= expression-2`

est équivalente à

`expression-1 = expression-1 op expression-2`

Toutefois, avec l'affection composée, *`expression-1`* n'est évaluée qu'une seule fois.

1.7.7 Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation `++` et de décrémentation `--` s'utilisent aussi bien en suffixe (`i++`) qu'en préfixe (`++i`). Dans les deux cas la variable `i` sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de `i` alors que dans la notation préfixe se sera la nouvelle. Par exemple,

```
int a = 3, b, c;
b = ++a;       /* a et b valent 4 */
c = b++;       /* c vaut 4 et b vaut 5 */
```

1.7.8 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

expression-1, expression-2, ... , expression-n

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple, le programme

```
main()
{
  int a, b;
  b = ((a = 3), (a + 2));
  printf("\n b = %d \n",b);
}
```

imprime `b = 5`.

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie. Par exemple l'instruction composée

```
{
int a=1;
printf("\%d \%d",++a,a);
}
```

(compilée avec gcc) produira la sortie `2 1` sur un PC Intel/Linux et la sortie `2 2` sur un DEC Alpha/OSF1.

1.7.9 L'opérateur conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :

condition ? *expression-1* : *expression-2*

Cette expression est égale à *expression-1* si *condition* est satisfaite, et à *expression-2* sinon. Par exemple, l'expression

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre. De même l'instruction

```
m = ((a > b) ? a : b);
```

affecte à `m` le maximum de `a` et de `b`.

1.7.10 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'un objet. On écrit

`(`*`type`*`)` *`objet`*

Par exemple,

```
main()
{
  int i = 3, j = 2;
  printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

1.7.11 L'opérateur adresse

L'opérateur d'adresse `&` appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est

`&`*`objet`*

1.7.12 Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est définie par la flèche de la seconde colonne du tableau. On préferera toutefois mettre des parenthèses en cas de doute...

<table>
<tr><th>opérateurs</th><th></th></tr>
<tr><td>() [] -> .</td><td>→</td></tr>
<tr><td>! ~ ++ -- -(unaire) (type) *(indirection) &(adresse) sizeof</td><td>←</td></tr>
<tr><td>* / %</td><td>→</td></tr>
<tr><td>+ -(binaire)</td><td>→</td></tr>
<tr><td><< >></td><td>→</td></tr>
<tr><td>< <= > >=</td><td>→</td></tr>
<tr><td>== !=</td><td>→</td></tr>
<tr><td>&(et bit-à-bit)</td><td>→</td></tr>
<tr><td>^</td><td>→</td></tr>
<tr><td>|</td><td>→</td></tr>
<tr><td>&&</td><td>→</td></tr>
<tr><td>||</td><td>→</td></tr>
<tr><td>? :</td><td>←</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>←</td></tr>
<tr><td>,</td><td>→</td></tr>
</table>

TAB. 1.4 – *Règles de priorité des opérateurs*

Par exemple, les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut parenthéser les expressions. Par exemple, il faut écrire `if ((x ^ y) != 0)`

1.8 Les instructions de branchement conditionnel

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les *instructions de branchement* et les *boucles*. Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

1.8.1 Branchement conditionnel `if---else`

La forme la plus générale est celle-ci :

```
if ( expression-1 )
   instruction-1
else if ( expression-2 )
   instruction-2
    ...
else if ( expression-n )
   instruction-n
else
   instruction-∞
```

avec un nombre quelconque de `else if ( ... )`. Le dernier `else` est toujours facultatif. La forme la plus simple est

```
if ( expression )
   instruction
```

Chaque *instruction* peut être un bloc d'instructions.

1.8.2 Branchement multiple `switch`

Sa forme la plus générale est celle-ci :

```
switch ( expression )
  {
  case  constante-1:
     liste d'instructions 1
    break;
  case  constante-2:
     liste d'instructions 2
    break;
    ...
  case  constante-n:
     liste d'instructions n
```

```
    break;
  default:
     liste d'instructions ∞
    break;
  }
```

Si la valeur de *expression* est égale à l'une des *constantes*, la *liste d'instructions* correspondant est exécutée. Sinon la *liste d'instructions* ∞ correspondant à `default` est exécutée. L'instruction `default` est facultative.

1.9 Les boucles

Les *boucles* permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

1.9.1 Boucle `while`

La syntaxe de `while` est la suivante :

```
while ( expression )
  instruction
```

Tant que *expression* est vérifiée (*i.e.*, non nulle), *instruction* est exécutée. Si *expression* est nulle au départ, *instruction* ne sera jamais exécutée. *instruction* peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
i = 1;
while (i < 10)
  {
    printf("\n i = %d",i);
    i++;
  }
```

1.9.2 Boucle `do---while`

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle `do---while`. Sa syntaxe est

```
do
   instruction
while ( expression );
```

Ici, *instruction* sera exécutée tant que *expression* est non nulle. Cela signifie donc que *instruction* est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;

do
```

```
  {
    printf("\n Entrez un entier entre 1 et 10 : ");
    scanf("%d",&a);
  }
while ((a <= 0) || (a > 10));
```

1.9.3 Boucle `for`

La syntaxe de `for` est :

```
for ( expr 1 ; expr 2 ; expr 3)
    instruction
```

Une version équivalente plus intuitive est :

```
 expr 1;
 while ( expr 2 )
   {
     instruction
     expr 3;
   }
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

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

A la fin de cette boucle, `i` vaudra 10. Les trois expressions utilisées dans une boucle `for` peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; i++)
  fact *= i;
printf("%d ! = %d \n",n,fact);
```

On peut également insérer l'instruction `fact *= i;` dans la boucle `for` ce qui donne :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; fact *= i, i++);
printf("%d ! = %d \n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

1.10 Les instructions de branchement non conditionnel

1.10.1 Branchement non conditionnel `break`

On a vu le rôle de l'instruction `break;` au sein d'une instruction de branchement multiple `switch`. L'instruction `break` peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
main()
{
  int i;
  for (i = 0; i < 5; i++)
    {
      printf("i = %d\n",i);
      if (i == 3)
        break;
    }
  printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

1.10.2 Branchement non conditionnel `continue`

L'instruction `continue` permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
  int i;
  for (i = 0; i < 5; i++)
    {
      if (i == 3)
        continue;
      printf("i = %d\n",i);
    }
  printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime

```
i = 0
```

```
i = 1
i = 2
i = 4
valeur de i a la sortie de la boucle = 5
```

1.10.3 Branchement non conditionnel `goto`

L'instruction `goto` permet d'effectuer un saut jusqu'à l'instruction ***etiquette*** correspondant. Elle est à proscrire de tout programme C digne de ce nom.

1.11 Les fonctions d'entrées-sorties classiques

Il s'agit des fonctions de la librairie standard `stdio.h` utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie `stdio.h` par la directive au préprocesseur

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

n'est pas nécessaire pour utiliser `printf` et `scanf`.

1.11.1 La fonction d'écriture `printf`

La fonction `printf` est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est

`printf("`*chaîne de contrôle* `",`*expression-1*`, ..., `*expression-n*`);`

La ***chaîne de contrôle*** contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère `%`, suivi d'un caractère désignant le format d'impression. Les formats d'impression en C sont donnés à la table 1.5.

En plus du caractère donnant le type des données, on peut éventuellemnt préciser certains paramètres du format d'impression, qui sont spécifiés entre le `%` et le caractère de conversion dans l'ordre suivant :

- largeur minimale du champ d'impression : `%10d` spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier. Par défaut, la donnée sera cadrée à droite du champ. Le signe `-` avant le format signifie que la donnée sera cadrée à gauche du champ (`%-10d`).

- précision : `%.12f` signifie qu'un flottant sera imprimé avec 12 chiffres après la virgule. De même `%10.2f` signifie que l'on réserve 12 caractères (incluant le caractère `.`) pour imprimer le flottant et que 2 d'entre eux sont destinés aux chiffres après la virgule. Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule. Pour une chaîne de caractères, la précision correspond au nombre de caractères imprimés : `%30.4s` signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

format	conversion en	écriture
`%d`	`int`	décimale signée
`%ld`	`long int`	décimale signée
`%u`	`unsigned int`	décimale non signée
`%lu`	`unsigned long int`	décimale non signée
`%o`	`unsigned int`	octale non signée
`%lo`	`unsigned long int`	octale non signée
`%x`	`unsigned int`	hexadécimale non signée
`%lx`	`unsigned long int`	hexadécimale non signée
`%f`	`double`	décimale virgule fixe
`%lf`	`long double`	décimale virgule fixe
`%e`	`double`	décimale notation exponentielle
`%le`	`long double`	décimale notation exponentielle
`%g`	`double`	décimale, représentation la plus courte parmi `%f` et `%e`
`%lg`	`long double`	décimale, représentation la plus courte parmi `%lf` et `%le`
`%c`	`unsigned char`	caractère
`%s`	`char*`	chaîne de caractères

TAB. 1.5 – *Formats d'impression pour la fonction* `printf`

Exemple :

```
#include <stdio.h>
main()
{
  int i = 23674;
  int j = -23674;
  long int k = (1l << 32);
  double x = 1e-8 + 1000;
  char c = 'A';
  char *chaine = "chaine de caracteres";

  printf("impression de i: \n");
  printf("%d \t %u \t %o \t %x",i,i,i,i);
  printf("\nimpression de j: \n");
  printf("%d \t %u \t %o \t %x",j,j,j,j);
  printf("\nimpression de k: \n");
  printf("%d \t  %o \t %x",k,k,k);
  printf("\n%ld \t %lu \t %lo \t %lx",k,k,k,k);
  printf("\nimpression de x: \n");
  printf("%f \t %e \t %g",x,x,x);
  printf("\n%.2f \t %.2e",x,x);
  printf("\n%.20f \t %.20e",x,x);
  printf("\nimpression de c: \n");
  printf("%c \t %d",c,c);
  printf("\nimpression de chaine: \n");
  printf("%s \t %.10s",chaine,chaine);
```

```
  printf("\n");
}
```

Ce programme imprime à l'écran :

```
impression de i:
23674   23674           56172   5c7a
impression de j:
-23674  4294943622      37777721606     ffffa386
impression de k:
0       0       0
4294967296      4294967296      40000000000     100000000
impression de x:
1000.000000     1.000000e+03    1000
1000.00         1.00e+03
1000.00000001000000000000       1.00000000001000000000e+03
impression de c:
A       65
impression de chaine:
chaine de caracteres    chaine de
```

1.11.2 La fonction de saisie `scanf`

La fonction `scanf` permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonctions.

`scanf("`*`chaîne de contrôle`*`",`*`argument-1`*`,...,`*`argument-n`*`)`

La *`chaîne de contrôle`* indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de `\n`). Comme pour `printf`, les conversions de format sont spécifiées par un caractère précédé du signe `%`. Les formats valides pour la fonction `scanf` diffèrent légèrement de ceux de la fonction `printf`.

Les données à entrer au clavier doivent être séparées par des blancs ou des `<RETURN>` sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple `%3s` pour une chaîne de 3 caractères, `%10d` pour un entier qui s'étend sur 10 chiffres, signe inclus.

Exemple :

```
#include <stdio.h>
main()
{
  int i;
  printf("entrez un entier sous forme hexadecimale i = ");
  scanf("%x",&i);
  printf("i = %d\n",i);
}
```

Si on entre au clavier la valeur `1a`, le programme affiche `i = 26`.

format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

TAB. 1.6 – *Formats de saisie pour la fonction* `scanf`

1.11.3 Impression et lecture de caractères

Les fonctions `getchar` et `putchar` permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

La fonction `getchar` retourne un `int` correspondant au caractère lu. Pour mettre le caractère lu dans une variable `caractere`, on écrit

```
caractere = getchar();
```

Lorsqu'elle détecte la fin de fichier, elle retourne l'entier `EOF` (End Of File), valeur définie dans la librairie `stdio.h`. En général, la constante `EOF` vaut -1.

La fonction `putchar` écrit *`caractere`* sur la sortie standard :

```
putchar(caractere);
```

Elle retourne un `int` correspondant à l'entier lu ou à la constante `EOF` en cas d'erreur.

Par exemple, le programme suivant lit un fichier et le recopie caractère par caractère à l'écran.

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

```
  char c;

  while ((c = getchar()) != EOF)
    putchar(c);
}
```

Pour l'exécuter, il suffit d'utiliser l'opérateur de redirection d'Unix :
programme-executable < *nom-fichier*

Notons que l'expression `(c = getchar())` dans le programme précédent a pour valeur la valeur de l'expression `getchar()` qui est de type `int`. Le test `(c = getchar()) != EOF` compare donc bien deux objets de type `int` (signés).

Ce n'est par contre pas le cas dans le programme suivant :

```
#include <stdio.h>
main()
{
  char c;
  do
    {
      c = getchar();
      if (c != EOF)
        putchar(c);
    }
  while (c != EOF);
}
```

Ici, le test `c != EOF` compare un objet de type `char` et la constante `EOF` qui vaut -1. Si le type `char` est non signé par défaut, cette condition est donc toujours vérifiée. Si le type `char` est signé, alors le caractère de code 255, ÿ, sera converti en l'entier -1. La rencontre du caractère ÿ sera donc interprétée comme une fin de fichier. Il est donc recommandé de déclarer de type `int` (et non `char`) une variable destinée à recevoir un caractère lu par `getchar` afin de permettre la détection de fin de fichier.

1.12 Les conventions d'écriture d'un programme C

Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles. Aussi existe-t-il un certain nombre de conventions.

- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.
- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.
- On laisse un blanc
  - entre les mots-clefs `if`, `while`, `do`, `switch` et la parenthèse ouvrante qui suit,

  - après une virgule,
  - de part et d'autre d'un opérateur binaire.
- On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.
- Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées. Le mieux est d'utiliser le mode C d'`Emacs`.

Chapitre 2

Les types composés

A partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés *types composés*, qui permettent de représenter des ensembles de données organisées.

2.1 Les tableaux

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

La déclaration d'un tableau à une dimension se fait de la façon suivante :

`type nom-du-tableau[nombre-éléments];`

où `nombre-éléments` est une expression constante entière positive. Par exemple, la déclaration `int tab[10];` indique que `tab` est un tableau de 10 éléments de type `int`. Cette déclaration alloue donc en mémoire pour l'objet `tab` un espace de 10×4 octets consécutifs.

Pour plus de clarté, il est recommandé de donner un nom à la constante `nombre-éléments` par une directive au préprocesseur, par exemple
`#define nombre-éléments 10`

On accède à un élément du tableau en lui appliquant l'opérateur `[]`. Les éléments d'un tableau sont toujours numérotés de 0 à `nombre-éléments -1`. Le programme suivant imprime les éléments du tableau `tab` :

```
#define N 10
main()
{
  int tab[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation.

Par exemple, on ne peut pas écrire "`tab1 = tab2;`". Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10
main()
{
  int tab1[N], tab2[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    tab1[i] = tab2[i];
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

`type nom-du-tableau``[N] = {`*`constante-1,constante-2,...,constante-N`*`};`

Par exemple, on peut écrire

```
#define N 4
int tab[N] = {1, 2, 3, 4};
main()
{
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro si le tableau est une variable globale (extérieure à toute fonction) ou une variable locale de classe de mémorisation `static` (*cf.* page 64).

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec un caractère nul `'\0'`. Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

```
#define N 8
char tab[N] = "exemple";
main()
{
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %c\n",i,tab[i]);
}
```

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation. Ainsi le programme suivant imprime le nombre de caractères du tableau `tab`, ici 8.

```
char tab[] = "exemple";
```

```
main()
{
  int i;
  printf("Nombre de caracteres du tableau = %d\n",sizeof(tab)/sizeof(char));
}
```

De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions :

type nom-du-tableau[*nombre-lignes*][*nombre-colonnes*]

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression "`tableau[i][j]`". Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main()
{
  int i, j;
  for (i = 0 ; i < M; i++)
    {
      for (j = 0; j < N; j++)
        printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
}
```

2.2 Les structures

Une *structure* est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.

On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est *modele* suit la syntaxe suivante :

```
struct  modele
{
    type-1 membre-1;
    type-2 membre-2;
     ...
    type-n membre-n;
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

struct *modele objet*;

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct  modele
{
   type-1 membre-1;
   type-2 membre-2;
    ...
   type-n membre-n;
}  objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur *membre de structure*, noté "`.`". Le `i`-ème membre de ***objet*** est désigné par l'expression

objet.membre-i

On peut effectuer sur le `i`-ème membre de la structure toutes les opérations valides sur des données de type ***type-i***. Par exemple, le programme suivant définit la structure `complexe`, composée de deux champs de type `double` ; il calcule la norme d'un nombre complexe.

```
#include <math.h>
struct complexe
{
  double reelle;
  double imaginaire;
};

main()
{
  struct complexe z;
  double norme;
   ...
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
}
```

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :
`struct complexe z = {2. , 2.};`
En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
...
main()
{
  struct complexe z1, z2;
```

```
  ...
  z2 = z1;
}
```

2.3 Les champs de bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (`int` ou `unsigned int`). Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration. Par exemple, la structure suivante

```
struct registre
{
  unsigned int actif : 1;
  unsigned int valeur : 31;
};
```

possède deux membres, `actif` qui est codé sur un seul bit, et `valeur` qui est codé sur 31 bits. Tout objet de type `struct registre` est donc codé sur 32 bits. Toutefois, l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation.

Le champ `actif` de la structure ne peut prendre que les valeurs 0 et 1. Aussi, si `r` est un objet de type `struct registre`, l'opération `r.actif += 2;` ne modifie pas la valeur du champ.

La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier. Notons enfin qu'un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur `&`.

2.4 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. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type `struct`. Dans l'exemple suivant, la variable `hier` de type `union jour` peut être soit un entier, soit un caractère.

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

main()
{
  union jour hier, demain;
  hier.lettre = 'J';
```

```
  printf("hier = %c\n",hier.lettre);
  hier.numero = 4;
  demain.numero = (hier.numero + 2) % 7;
  printf("demain = %d\n",demain.numero);
}
```

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type `unsigned int` d'une structure en les identifiant à un objet de type `unsigned long` (en supposant que la taille d'un entier long est deux fois celle d'un `int`).

```
struct coordonnees
{
  unsigned int x;
  unsigned int y;
};
union point
{
  struct coordonnees coord;
  unsigned long mot;
};

main()
{
  union point p1, p2, p3;
  p1.coord.x = 0xf;
  p1.coord.y = 0x1;
  p2.coord.x = 0x8;
  p2.coord.y = 0x8;
  p3.mot = p1.mot ^ p2.mot;
  printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

2.5 Les énumérations

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

`enum` *`modele`* {*`constante-1`*, *`constante-2`*,...,*`constante-n`*};

En réalité, les objets de type `enum` sont représentés comme des `int`. Les valeurs possibles *`constante-1`*, *`constante-2`*,...,*`constante-n`* sont codées par des entiers de 0 à *`n-1`*. Par exemple, le type `enum booleen` défini dans le programme suivant associe l'entier 0 à la valeur `faux` et l'entier 1 à la valeur `vrai`.

```
main()
```

```
{
  enum booleen {faux, vrai};
  enum booleen b;
  b = vrai;
  printf("b = %d\n",b);
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :
`enum booleen {faux = 12, vrai = 23};`

2.6 Définition de types composés avec `typedef`

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de `typedef` :

`typedef` *`type synonyme`*`;`

Par exemple,

```
struct complexe
{
  double reelle;
  double imaginaire;
};
typedef struct complexe complexe;

main()
{
  complexe z;
   ...
}
```

Chapitre 3

Les pointeurs

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

3.1 Adresse et valeur d'un objet

On appelle *Lvalue* (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *Lvalue* est caractérisée par :

– son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;

– sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

Dans l'exemple,

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

Si le compilateur a placé la variable `i` à l'adresse 4831836000 en mémoire, et la variable `j` à l'adresse 4831836004, on a

objet	adresse	valeur
`i`	4831836000	3
`j`	4831836004	3

Deux variables différentes ont des adresses différentes. L'affectation `i = j;` n'opère que sur les valeurs des variables. Les variables `i` et `j` étant de type `int`, elles sont stockées sur 4 octets. Ainsi la valeur de `i` est stockée sur les octets d'adresse 4831836000 à 4831836003.

L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures. Sur un DEC alpha, par exemple, une adresse a toujours le format d'un entier long (64 bits).

L'opérateur `&` permet d'accéder à l'adresse d'une variable. Toutefois `&i` n'est pas une Lvalue mais une constante : on ne peut pas faire figurer `&i` à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs.

3.2 Notion de pointeur

Un *pointeur* est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

*type *nom-du-pointeur*;

où ***type*** est le type de l'objet pointé. Cette déclaration déclare un identificateur, ***nom-du-pointeur***, associé à un objet dont la valeur est l'adresse d'un autre objet de type ***type***. L'identificateur ***nom-du-pointeur*** est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.

Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet, pour un pointeur sur un objet de type `char`, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type `int`, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké. Dans l'exemple suivant, on définit un pointeur `p` qui pointe vers un entier `i` :

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

p = &i;
```

On se trouve dans la configuration

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

L'*opérateur unaire d'indirection* `*` permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si `p` est un pointeur vers un entier `i`, `*p` désigne la valeur de `i`. Par exemple, le programme

```
main()
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
}
```

imprime `*p = 3`.

Dans ce programme, les objets `i` et `*p` sont identiques : ils ont mêmes adresse et valeur. Nous sommes dans la configuration :

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	4831836000
`*p`	4831836000	3

Cela signifie en particulier que toute modification de `*p` modifie `i`. Ainsi, si l'on ajoute l'instruction `*p = 0;` à la fin du programme précédent, la valeur de `i` devient nulle.

On peut donc dans un programme manipuler à la fois les objets `p` et `*p`. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
}
```

et

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

objet	adresse	valeur
`i`	4831836000	3
`j`	4831836004	6
`p1`	4831835984	4831836000
`p2`	4831835992	4831836004

Après l'affectation `*p1 = *p2;` du premier programme, on a

objet	adresse	valeur
`i`	4831836000	6
`j`	4831836004	6
`p1`	4831835984	4831836000
`p2`	4831835992	4831836004

Par contre, l'affectation `p1 = p2` du second programme, conduit à la situation :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

3.3 Arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

– l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;

– la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;

– la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée.

Si `i` est un entier et `p` est un pointeur sur un objet de type ***type***, l'expression `p + i` désigne un pointeur sur un objet de type ***type*** dont la valeur est égale à la valeur de `p` incrémentée de `i * sizeof(`***type***`)`. Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation `++` et `--`. Par exemple, le programme

```
main()
{
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche `p1 = 4831835984        p2 = 4831835988`.

Par contre, le même programme avec des pointeurs sur des objets de type `double` :

```
main()
{
  double i = 3;
  double *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche `p1 = 4831835984          p2 = 4831835992`.

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux. Ainsi, le programme suivant imprime les éléments du tableau `tab` dans l'ordre croissant puis décroissant des indices.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int *p;
  printf("\n ordre croissant:\n");
  for (p = &tab[0]; p <= &tab[N-1]; p++)
    printf(" %d \n",*p);
  printf("\n ordre decroissant:\n");
  for (p = &tab[N-1]; p >= &tab[0]; p--)
    printf(" %d \n",*p);
}
```

Si `p` et `q` sont deux pointeurs sur des objets de type ***type***, l'expression `p - q` désigne un entier dont la valeur est égale à `(p - q)/sizeof(`***type***`)` .

3.4 Allocation dynamique

Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection `*`, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée `NULL` définie dans `stdio.h`. En général, cette constante vaut 0. Le test `p == NULL` permet de savoir si le pointeur `p` pointe vers un objet.

On peut initialiser un pointeur `p` par une affectation sur `p`. Par exemple, on peut affecter à `p` l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à `*p`. Mais pour cela, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate. L'adresse de cet espace-mémoire sera la valeur de `p`. Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle *allocation dynamique*. Elle se fait en C par la fonction `malloc` de la librairie standard `stdlib.h`. Sa syntaxe est

`malloc(`*nombre-octets*`)`

Cette fonction retourne un pointeur de type `char *` pointant vers un objet de taille ***nombre-octets*** octets. Pour initialiser des pointeurs vers des objets qui ne sont pas de type `char`, il faut convertir le type de la sortie de la fonction `malloc` à l'aide d'un cast. L'argument ***nombre-octets*** est souvent donné à l'aide de la fonction `sizeof()` qui renvoie le nombre d'octets utilisés pour stocker un objet.

Ainsi, pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
int *p;
p = (int*)malloc(sizeof(int));
```

On aurait pu écrire également
`p = (int*)malloc(4);`
puisqu'un objet de type `int` est stocké sur 4 octets. Mais on préférera la première écriture qui a l'avantage d'être portable.

Le programme suivant

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i = 3;
  int *p;
  printf("valeur de p avant initialisation = %ld\n",p);
  p = (int*)malloc(sizeof(int));
  printf("valeur de p apres initialisation = %ld\n",p);
  *p = i;
  printf("valeur de *p = %d\n",*p);
}
```

définit un pointeur `p` sur un objet `*p` de type `int`, et affecte à `*p` la valeur de la variable `i`. Il imprime à l'écran :

```
valeur de p avant initialisation = 0
valeur de p apres initialisation = 5368711424
valeur de *p = 3
```

Avant l'allocation dynamique, on se trouve dans la configuration

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	0

A ce stade, `*p` n'a aucun sens. En particulier, toute manipulation de la variable `*p` générerait une violation mémoire, détectable à l'exécution par le message d'erreur `Segmentation fault`.

L'allocation dynamique a pour résultat d'attribuer une valeur à `p` et de réserver à cette adresse un espace-mémoire composé de 4 octets pour stocker la valeur de `*p`. On a alors

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	5368711424
`*p`	5368711424	? (int)

`*p` est maintenant définie mais sa valeur n'est pas initialisée. Cela signifie que `*p` peut valoir n'importe quel entier (celui qui se trouvait précédemment à cette adresse). L'affectation `*p = i;` a enfin pour résultat d'affecter à `*p` la valeur de `i`. A la fin du programme, on a donc

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	5368711424
`*p`	5368711424	3

Il est important de comparer le programme précédent avec

```
main()
{
  int i = 3;
  int *p;

  p = &i;
}
```

qui correspond à la situation

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Dans ce dernier cas, les variables `i` et `*p` sont identiques (elles ont la même adresse) ce qui implique que toute modification de l'une modifie l'autre. Ceci n'était pas vrai dans l'exemple précédent où `*p` et `i` avaient la même valeur mais des adresses différentes.

On remarquera que le dernier programme ne nécessite pas d'allocation dynamique puisque l'espace-mémoire à l'adresse `&i` est déjà réservé pour un entier.

La fonction `malloc` permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i = 3;
  int j = 6;
  int *p;
  p = (int*)malloc(2 * sizeof(int));
  *p = i;
  *(p + 1) = j;
  printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
```

On a ainsi réservé, à l'adresse donnée par la valeur de `p`, 8 octets en mémoire, qui permettent de stocker 2 objets de type `int`. Le programme affiche
`p = 5368711424    *p = 3    p+1 = 5368711428    *(p+1) = 6` .

La fonction `calloc` de la librairie `stdlib.h` a le même rôle que la fonction `malloc` mais elle initialise en plus l'objet pointé `*p` à zéro. Sa syntaxe est

`calloc(`*`nb-objets`*`,`*`taille-objets`*`)`

Ainsi, si `p` est de type `int*`, l'instruction

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

est strictement équivalente à

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
  *(p + i) = 0;
```

L'emploi de `calloc` est simplement plus rapide.

Enfin, lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement (c'est-à-dire quand on n'utilise plus le pointeur `p`), il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction `free` qui a pour syntaxe

`free(`*nom-du-pointeur*`);`

A toute instruction de type `malloc` ou `calloc` doit être associée une instruction de type `free`.

3.5 Pointeurs et tableaux

L'usage des pointeurs en C est, en grande partie, orienté vers la manipulation des tableaux.

3.5.1 Pointeurs et tableaux à une dimension

Tout tableau en C est en fait un pointeur constant. Dans la déclaration

```
int tab[10];
```

`tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, `tab` a pour valeur `&tab[0]`. On peut donc utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    {
      printf(" %d \n",*p);
      p++;
    }
}
```

On accède à l'élément d'indice `i` du tableau `tab` grâce à l'opérateur d'indexation `[]`, par l'expression `tab[i]`. Cet opérateur d'indexation peut en fait s'appliquer à tout objet `p` de type pointeur. Il est lié à l'opérateur d'indirection `*` par la formule

`p[i] = *(p + i)`

Pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précédent peut aussi s'écrire

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    printf(" %d \n", p[i]);
}
```

Toutefois, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant. Ainsi

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

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à `n` éléments où `n` est une variable du programme, on écrit

```
#include <stdlib.h>
main()
{
  int n;
  int *tab;

   ...
  tab = (int*)malloc(n * sizeof(int));
   ...
  free(tab);
}
```

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on remplace l'allocation dynamique avec `malloc` par

```
  tab = (int*)calloc(n, sizeof(int));
```

Les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i`;
- un tableau n'est pas une Lvalue; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++;`).

3.5.2 Pointeurs et 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. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N];
```

`tab` est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier. `tab` a une valeur constante égale à l'adresse du premier élément du tableau, `&tab[0][0]`. De même `tab[i]`, pour `i` entre 0 et `M-1`, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice `i`. `tab[i]` a donc une valeur constante qui est égale à `&tab[i][0]`.

Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés.

On déclare un pointeur qui pointe sur un objet de type *type* * (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire

type ***nom-du-pointeur*;

De même un pointeur qui pointe sur un objet de type *type* ** (équivalent à un tableau à 3 dimensions) se déclare par

type ****nom-du-pointeur*;

Par exemple, pour créer avec un pointeur de pointeur une matrice à `k` lignes et `n` colonnes à coefficients entiers, on écrit :

```
main()
{
  int k, n;
  int **tab;

  tab = (int**)malloc(k * sizeof(int*));
  for (i = 0; i < k; i++)
    tab[i] = (int*)malloc(n * sizeof(int));
      ....

  for (i = 0; i < k; i++)
    free(tab[i]);
  free(tab);
}
```

La première allocation dynamique réserve pour l'objet pointé par `tab` l'espace-mémoire correspondant à `k` pointeurs sur des entiers. Ces `k` pointeurs correspondent aux lignes de la matrice. Les allocations dynamiques suivantes réservent pour chaque pointeur `tab[i]` l'espace-mémoire nécessaire pour stocker `n` entiers.

Si on désire en plus que tous les éléments du tableau soient initialisés à zéro, il suffit de remplacer l'allocation dynamique dans la boucle `for` par

```
    tab[i] = (int*)calloc(n, sizeof(int));
```

Contrairement aux tableaux à deux dimensions, on peut choisir des tailles différentes pour chacune des lignes `tab[i]`. Par exemple, si l'on veut que `tab[i]` contienne exactement `i+1` éléments, on écrit

```
for (i = 0; i < k; i++)
  tab[i] = (int*)malloc((i + 1) * sizeof(int));
```

3.5.3 Pointeurs et chaînes de caractères

On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type `char`, se terminant par le caractère nul `'\0'`. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`. On peut faire subir à une chaîne définie par

```
char *chaine;
```

des affectations comme

```
chaine = "ceci est une chaine";
```

et toute opération valide sur les pointeurs, comme l'instruction `chaine++;`. Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

```
#include <stdio.h>
main()
{
  int i;
  char *chaine;

  chaine = "chaine de caracteres";
  for (i = 0; *chaine != '\0'; i++)
    chaine++;
  printf("nombre de caracteres = %d\n",i);
}
```

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard `string.h`, procède de manière identique. Il s'agit de la fonction `strlen` dont la syntaxe est

`strlen(`*`chaine`*`);`

où *chaine* est un pointeur sur un objet de type `char`. Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaîne passée en argument (moins le caractère `'\0'`).
L'utilisation de pointeurs de caractère et non de tableaux permet par exemple de créer une chaîne correspondant à la concaténation de deux chaînes de caractères :

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

```
  int i;
  char *chaine1, *chaine2, *res, *p;

  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  p = res;
  for (i = 0; i < strlen(chaine1); i++)
    *p++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *p++ = chaine2[i];
  printf("%s\n",res);
}
```

On remarquera l'utilisation d'un pointeur intermédiaire `p` qui est indispensable dès que l'on fait des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de `res`, on aurait évidemment "perdu" la référence sur le premier caractère de la chaîne. Par exemple,

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  int i;
  char *chaine1, *chaine2, *res;

  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  for (i = 0; i < strlen(chaine1); i++)
    *res++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *res++ = chaine2[i];
  printf("\nnombre de caracteres de res = %d\n",strlen(res));
}
```

imprime la valeur `0`, puisque `res` a été modifié au cours du programme et pointe maintenant sur le caractère nul.

3.6 Pointeurs et structures

3.6.1 Pointeur sur une structure

Contrairement aux tableaux, les objets de type structure en C sont des Lvalues. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures. Ainsi, le programme suivant crée, à l'aide d'un pointeur, un tableau d'objets de type structure.

```
#include <stdlib.h>
```

```
#include <stdio.h>

struct eleve
{
  char nom[20];
  int date;
};

typedef struct eleve *classe;

main()
{
  int n, i;
  classe tab;

  printf("nombre d'eleves de la classe = ");
  scanf("%d",&n);
  tab = (classe)malloc(n * sizeof(struct eleve));

  for (i =0 ; i < n; i++)
    {
      printf("\n saisie de l'eleve numero %d\n",i);
      printf("nom de l'eleve = ");
      scanf("%s",&tab[i].nom);
      printf("\n date de naissance JJMMAA = ");
      scanf("%d",&tab[i].date);
    }
  printf("\n Entrez un numero  ");
  scanf("%d",&i);
  printf("\n Eleve numero %d:",i);
  printf("\n nom = %s",tab[i].nom);
  printf("\n date de naissance = %d\n",tab[i].date);
  free(tab);
}
```

Si `p` est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression

`(*p).`*`membre`*

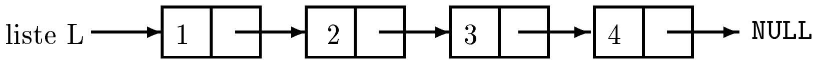
L'usage de parenthèses est ici indispensable car l'opérateur d'indirection `*` à une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté `->`. L'expression précédente est strictement équivalente à

`p->`*`membre`*

Ainsi, dans le programme précédent, on peut remplacer `tab[i].nom` et `tab[i].date` respectivement par `(tab + i)->nom` et `(tab + i)->date`.

3.6.2 Structures auto-référencées

On a souvent besoin en C de modèles de structure dont un des membres est un pointeur vers une structure de même modèle. Cette représentation permet en particulier de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutefois, cette représentation, dite *contiguë*, impose que la taille maximale de la liste soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique). Pour résoudre ce problème, on utilise une représentation *chaînée* : l'élément de base de la chaîne est une structure appelée *cellule* qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`. La liste est alors définie comme un pointeur sur le premier élément de la chaîne.



Pour représenter une liste d'entiers sous forme chaînée, on crée le modèle de structure `cellule` qui a deux champs : un champ `valeur` de type `int`, et un champ `suivant` de type pointeur sur une `struct cellule`. Une liste sera alors un objet de type pointeur sur une `struct cellule`. Grâce au mot-clef `typedef`, on peut définir le type `liste`, synonyme du type pointeur sur une `struct cellule`.

```
struct cellule
{
  int valeur;
  struct cellule *suivant;
};

typedef struct cellule *liste;
```

Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un élément à un endroit quelconque de la liste. Ainsi, pour insérer un élément en tête de liste, on utilise la fonction suivante :

```
liste insere(int element, liste Q)
{
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}
```

Le programme suivant crée une liste d'entiers et l'imprime à l'écran :

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

struct cellule
{
  int valeur;
```

```
  struct cellule *suivant;
};

typedef struct cellule *liste;

liste insere(int element, liste Q)
{
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}

main()
{
  liste L, P;

  L = insere(1,insere(2,insere(3,insere(4,NULL))));
  printf("\n impression de la liste:\n");
  P = L;
  while (P != NULL)
    {
      printf("%d \t",P->valeur);
      P = P->suivant;
    }
}
```

On utilisera également une structure auto-référencée pour créer un arbre binaire :

```
struct noeud
{
  int valeur;
  struct noeud *fils_gauche;
  struct noeud *fils_droit;
};

typedef struct noeud *arbre;
```

Chapitre 4

Les fonctions

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée `main`. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est *récursive*).

4.1 Définition d'une fonction

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme

```
 type  nom-fonction ( type-1 arg-1,..., type-n arg-n)
{
  [ déclarations de variables locales ]
   liste d'instructions
}
```

La première ligne de cette définition est l'*en-tête* de la fonction. Dans cet en-tête, ***type*** désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef `void`. Les arguments de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef `void`.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, `return`, dont la syntaxe est

```
return(expression);
```

La valeur de ***expression*** est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type `void`), sa définition s'achève par

```
return;
```

Plusieurs instructions `return` peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier `return` rencontré lors de l'exécution. Voici quelques exemples de définitions de fonctions :

```
int produit (int a, int b)
{
  return(a*b);
}

int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}

void imprime_tab (int *tab, int nb_elements)
{
  int i;
  for (i = 0; i < nb_elements; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}
```

4.2 Appel d'une fonction

L'appel d'une fonction se fait par l'expression

`nom-fonction(para-1,para-2,...,para-n)`

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur *virgule*. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur. Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (`++` ou `--`) dans les expressions définissant les paramètres effectifs (*cf.* Chapitre 1, page 23).

4.3 Déclaration d'une fonction

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale `main`. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction `main`), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son *prototype*, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

`type nom-fonction(type-1,...,type-n);`

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction `main`. Par exemple, on écrira

```
int puissance (int, int );

int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}

main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction `main` et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le protype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype. Ainsi les fichiers d'extension `.h` de la librairie standard (fichiers headers) contiennent notamment les prototypes des fonctions de la librairie standard. Par exemple, on trouve dans le fichier `math.h` le prototype de la fonction `pow` (élévation à la puissance) :

```
extern  double pow(double , double );
```

La directive au préprocesseur

```
#include <math.h>
```

permet au préprocesseur d'inclure la déclaration de la fonction `pow` dans le fichier source. Ainsi, si cette fonction est appelée avec des paramètres de type `int`, ces paramètres seront convertis en `double` lors de la compilation.

Par contre, en l'absence de directive au préprocesseur, le compilateur ne peut effectuer la conversion de type. Dans ce cas, l'appel à la fonction `pow` avec des paramètres de type `int` peut produire un résultat faux !

4.4 Durée de vie des variables

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*. On distingue deux catégories de variables.

Les variables permanentes (ou statiques) Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée *segment de données*. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef `static`.

Les variables temporaires Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée *segment de pile*. Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, `auto`, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

Une variable temporaire peut également être placée dans un registre de la machine. Un registre est une zone mémoire sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire. On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type `register`. Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt. Grâce aux performances des optimiseurs de code intégrés au compilateur (*cf.* options `-O` de `gcc`, page 10), il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.

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

4.4.1 Variables globales

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, `n` est une variable globale :

```
int n;
void fonction();

void fonction()
{
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
```

```
  for (i = 0; i < 5; i++)
    fonction();
}
```

La variable `n` est initialisée à zéro par le compilateur et il s'agit d'une variable permanente. En effet, le programme affiche

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

4.4.2 Variables locales

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant

```
int n = 10;
void fonction();

void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

affiche

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef `static` :

`static` *`type nom-de-variable`*`;`

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, `n` est une variable locale à la fonction secondaire `fonction`, mais de classe statique.

```
int n = 10;
void fonction();

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Ce programme affiche

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale `n` est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

4.5 Transmission des paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont *transmis par valeurs*. Par exemple, le programme suivant

```
void echange (int, int );

void echange (int a, int b)
{
  int t;
  printf("debut fonction :\n a = %d \t b = %d\n",a,b);
  t = a;
  a = b;
  b = t;
  printf("fin fonction :\n a = %d \t b = %d\n",a,b);
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(a,b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

imprime

```
debut programme principal :
 a = 2   b = 5
debut fonction :
 a = 2   b = 5
fin fonction :
 a = 5   b = 2
fin programme principal :
 a = 2   b = 5
```

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *, int *);

void echange (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
```

```
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(&a,&b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction. Par exemple, le programme

```
#include <stdlib.h>

void init (int *, int );

void init (int *tab, int n)
{
  int i;
  for (i = 0; i < n; i++)
    tab[i] = i;
  return;
}

main()
{
  int i, n = 5;
  int *tab;
  tab = (int*)malloc(n * sizeof(int));
  init(tab,n);
}
```

initialise les éléments du tableau `tab`.

4.6 Les qualificateurs de type `const` et `volatile`

Les qualificateurs de type `const` et `volatile` permettent de réduire les possibilités de modifier une variable.

`const` Une variable dont le type est qualifié par `const` ne peut pas être modifiée. Ce qualificateur est utilisé pour se protéger d'une erreur de programmation. On l'emploie principalement pour qualifier le type des paramètres d'une fonction afin d'éviter de les modifier involontairement.

`volatile` Une variable dont le type est qualifié par `volatile` ne peut pas être impliquée dans les optimisations effectuées par le compilateur. On utilise ce qualificateur pour les variables susceptibles d'être modifiées par une action extérieure au programme.

Les qualificateurs de type se placent juste avant le type de la variable, par exemple

```
const char c;
```

désigne un caractère non modifiable. Ils doivent toutefois être utilisés avec précaution avec les pointeurs. En effet,

```
const char *p;
```

définit un pointeur sur un caractère constant, tandis que

```
char * const p;
```

définit un pointeur constant sur un caractère.

4.7 La fonction `main`

La fonction principale `main` est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type `void`, ce qui est toléré par le compilateur. Toutefois l'écriture

```
main()
```

provoque un message d'avertissement lorsqu'on utilise l'option `-Wall` de `gcc` :

```
% gcc -Wall prog.c
prog.c:5: warning: return-type defaults to `int'
prog.c: In function `main':
prog.c:11: warning: control reaches end of non-void function
```

En fait, la fonction `main` est de type `int`. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur. On peut utiliser comme valeur de retour les deux constantes symboliques `EXIT_SUCCESS` (égale à 0) et `EXIT_FAILURE` (égale à 1) définies dans `stdlib.h`. L'instruction `return(`*`statut`*`);` dans la fonction `main`, où *`statut`* est un entier spécifiant le type de terminaison du programme, peut être remplacée par un appel à la fonction `exit` de la librairie standard (`stdlib.h`). La fonction `exit`, de prototype

```
void exit(int statut);
```

provoque une terminaison normale du programme en notifiant un succès ou un échec selon la valeur de l'entier `statut`.

Lorsqu'elle est utilisée sans arguments, la fonction `main` a donc pour prototype

```
int main(void);
```

On s'attachera désormais dans les programmes à respecter ce prototype et à spécifier les valeurs de retour de `main`.

La fonction `main` peut également posséder des paramètres formels. En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction `main` reçoit tous ces éléments de la part de l'interpréteur de commandes. En fait, la fonction `main` possède deux paramètres formels, appelés par convention `argc` (argument count) et `argv` (argument vector). `argc` est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1. `argv` est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément `argv[0]` contient donc le nom de la commande (du fichier exécutable), le second `argv[1]` contient le premier paramètre....

Le second prototype valide de la fonction `main` est donc

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

Ainsi, le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

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

int main(int argc, char *argv[])
{
  int a, b;

  if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
  return(EXIT_SUCCESS);
}
```

On lance donc l'exécutable avec deux paramètres entiers, par exemple,

```
a.out 12 8
```

Ici, `argv` sera un tableau de 3 chaînes de caractères `argv[0]`, `argv[1]` et `argv[2]` qui, dans notre exemple, valent respectivement `"a.out"`, `"12"` et `"8"`. Enfin, la fonction de la librairie standard `atoi()`, déclarée dans `stdlib.h`, prend en argument une chaîne de caractères et retourne l'entier dont elle est l'écriture décimale.

4.8 Pointeur sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages. Pour cela, on utilise un mécanisme de pointeur. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Un pointeur sur une fonction ayant pour prototype
type `fonction(`*type_1*`,...,`*type_n*`);`
est de type
type `(*)(`*type_1*`,...,`*type_n*`);`

Ainsi, une fonction `operateur_binaire` prenant pour paramètres deux entiers et une fonction de type `int`, qui prend elle-même deux entiers en paramètres, sera définie par :

```
int operateur_binaire(int a, int b, int (*f)(int, int))
```

Sa déclaration est donnée par

```
int operateur_binaire(int, int, int(*)(int, int));
```

Pour appeler la fonction `operateur_binaire`, on utilisera comme troisième paramètre effectif l'identificateur de la fonction utilisée, par exemple, si `somme` est une fonction de prototype

```
int somme(int, int);
```

on appelle la fonction `operateur_binaire` pour la fonction `somme` par l'expression

```
operateur_binaire(a,b,somme)
```

Notons qu'on n'utilise pas la notation `&somme` comme paramètre effectif de `operateur_binaire`.

Pour appeler la fonction passée en paramètre dans le corps de la fonction `operateur_binaire`, on écrit `(*f)(a, b)`. Par exemple

```
int operateur_binaire(int a, int b, int (*f)(int, int))
{
  return((*f)(a,b));
}
```

Ainsi, le programme suivant prend comme arguments deux entiers séparés par la chaîne de caractères `plus` ou `fois`, et retourne la somme ou le produit des deux entiers.

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

void usage(char *);
int somme(int, int);
int produit(int, int);
int operateur_binaire(int, int, int(*)(int, int));

void usage(char *cmd)
```

```
{
  printf("\nUsage: %s int [plus|fois] int\n",cmd);
  return;
}

int somme(int a, int b)
{
  return(a + b);
}

int produit(int a, int b)
{
  return(a * b);
}

int operateur_binaire(int a, int b, int (*f)(int, int))
{
  return((*f)(a,b));
}

int main(int argc, char *argv[])
{
  int a, b;


  if (argc != 4)
    {
      printf("\nErreur : nombre invalide d'arguments");
      usage(argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[3]);
  if (!strcmp(argv[2], "plus"))
    {
      printf("%d\n",operateur_binaire(a,b,somme));
      return(EXIT_SUCCESS);
    }
  if (!strcmp(argv[2], "fois"))
    {
      printf("%d\n",operateur_binaire(a,b,produit));
      return(EXIT_SUCCESS);
    }
  else
    {
      printf("\nErreur : argument(s) invalide(s)");
      usage(argv[0]);
```

```
      return(EXIT_FAILURE);
    }
}
```

Les pointeurs sur les fonctions sont notamment utilisés dans la fonction de tri des éléments d'un tableau `qsort` et dans la recherche d'un élément dans un tableau `bsearch`. Ces deux fonctions sont définies dans la libriarie standard (`stdlib.h`).

Le prototype de la fonction de tri (algorithme quicksort) est

```
void    qsort(void *tableau, size_t nb_elements, size_t taille_elements,
int(*comp)(const void *, const void *));
```

Elle permet de trier les `nb_elements` premiers éléments du tableau `tableau`. Le paramètre `taille_elements` donne la taille des éléments du tableau. Le type `size_t` utilisé ici est un type prédéfini dans `stddef.h`. Il correspond au type du résultat de l'évaluation de `sizeof`. Il s'agit du plus grand type entier non signé. La fonction `qsort` est paramétrée par la fonction de comparaison utilisée de prototype :

```
int comp(void *a, void *b);
```

Les deux paramètres `a` et `b` de la fonction `comp` sont des pointeurs génériques de type `void *`. Ils correspondent à des adresses d'objets dont le type n'est pas déterminé. Cette fonction de comparaison retourne un entier qui vaut 0 si les deux objets pointés par `a` et `b` sont égaux et qui prend une valeur strictement négative (resp. positive) si l'objet pointé par `a` est strictement inférieur (resp. supérieur) à celui pointé par `b`.

Par exemple, la fonction suivante comparant deux chaînes de caractères peut être utilisée comme paramètre de `qsort` :

```
int comp_str(char **, char **);

int comp_str(char **s1, char **s2)
{
  return(strcmp(*s1,*s2));
}
```

Le programme suivant donne un exemple de l'utilisation de la fonction de tri `qsort` pour trier les éléments d'un tableau d'entiers, et d'un tableau de chaînes de caractères.

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

#define NB_ELEMENTS 10

void imprime_tab1(int*, int);
void imprime_tab2(char**, int);
int comp_int(int *, int *);
int comp_str(char **, char **);

void imprime_tab1(int *tab, int nb)
```

```
{
  int i;
  printf("\n");
  for (i = 0; i < nb; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}

void imprime_tab2(char **tab, int nb)
{
  int i;
  printf("\n");
  for (i = 0; i < nb; i++)
    printf("%s \t",tab[i]);
  printf("\n");
  return;
}

int comp_int(int *a, int *b)
{
  return(*a - *b);
}
int comp_str(char **s1, char **s2)
{
  return(strcmp(*s1,*s2));
}

int main()
{
  int *tab1;
  char *tab2[NB_ELEMENTS] = {"toto", "Auto", "auto", "titi", "a", "b",\
"z", "i , "o","d"};
  int i;

  tab1 = (int*)malloc(NB_ELEMENTS * sizeof(int));
  for (i = 0 ; i < NB_ELEMENTS; i++)
    tab1[i] = random() % 1000;
  imprime_tab1(tab1, NB_ELEMENTS);
  qsort(tab1, NB_ELEMENTS, sizeof(int), comp_int);
  imprime_tab1(tab1, NB_ELEMENTS);
  /************************/
  imprime_tab2(tab2, NB_ELEMENTS);
  qsort(tab2, NB_ELEMENTS, sizeof(tab2[0]), comp_str);
  imprime_tab2(tab2, NB_ELEMENTS);
  return(EXIT_SUCCESS);
}
```

La librairie standard dispose également d'une fonction de recherche d'un élément dans un tableau *trié*, ayant le prototype suivant :

```
void    *bsearch((const void *clef, const void *tab, size_t nb_elements,
size_t taille_elements, int(*comp)(const void *, const void *)));
```

Cette fonction recherche dans le tableau trié `tab` un élément qui soit égal à l'élément d'adresse `clef`. Les autres paramètres sont identiques à ceux de la fonction `qsort`. S'il existe dans le tableau `tab` un élément égal à celui pointé par `clef`, la fonction `bsearch` retourne son adresse (de type `void *`). Sinon, elle retourne le pointeur `NULL`.

Ainsi, le programme suivant prend en argument une chaîne de caractères et détermine si elle figure dans un tableau de chaînes de caractères prédéfini, sans différencier minuscules et majuscules. Rappelons que `bsearch` ne s'applique qu'aux tableaux triés ; il faut donc appliquer au préalable la fonction de tri `qsort`.

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

#define NB_ELEMENTS 4

int comp_str_maj(char **, char **);

int comp_str_maj(char **s1, char **s2)
{
  int i;
  char *chaine1, *chaine2;

  chaine1 = (char*)malloc(strlen(*s1) * sizeof(char));
  chaine2 = (char*)malloc(strlen(*s2) * sizeof(char));
  for (i = 0; i < strlen(*s1); i++)
    chaine1[i] = tolower((*s1)[i]);
  for (i = 0; i < strlen(*s2); i++)
    chaine2[i] = tolower((*s2)[i]);
  return(strcmp(chaine1,chaine2));
}

int main(int argc, char *argv[])
{
  char *tab[NB_ELEMENTS] = {"TOTO", "Auto", "auto", "titi"};
  char **res;

  qsort(tab, NB_ELEMENTS, sizeof(tab[0]), comp_str_maj);
  if ((res = bsearch(&argv[1],tab,NB_ELEMENTS,sizeof(tab[0]),comp_str_maj)) ==\
NULL)
    printf("\nLe tableau ne contient pas l'element %s\n",argv[1]);
  else
```

```
    printf("\nLe tableau contient l'element %s sous la forme %s\n",argv[1], \
*res);
  return(EXIT_SUCCESS);
}
```

4.9 Fonctions avec un nombre variable de paramètres

Il est possible en C de définir des fonctions qui ont un nombre variable de paramètres. En pratique, il existe souvent des méthodes plus simples pour gérer ce type de problème : toutefois, cette fonctionnalité est indispensable dans certains cas, notamment pour les fonctions `printf` et `scanf`.

Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. La notation ... (obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes. Ainsi, une fonction ayant pour prototype

```
int f(int a, char c, ...);
```

prend comme paramètre un entier, un caractère et un nombre quelconque d'autres paramètres. De même le prototype de la fonction `printf` est

```
int printf(char *format, ...);
```

puisque `printf` a pour argument une chaîne de caractères spécifiant le format des données à imprimer, et un nombre quelconque d'autres arguments qui peuvent être de types différents.

Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction.

Pour accéder à la liste des paramètres de l'appel, on utilise les macros définies dans le fichier en-tête `stdarg.h` de la librairie standard. Il faut tout d'abord déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel; cette variable a pour type `va_list`. Par exemple,

```
va_list liste_parametres;
```

Cette variable est tout d'abord initialisée à l'aide de la macro `va_start`, dont la syntaxe est

```
va_start(liste_parametres, dernier_parametre);
```

où `dernier_parametre` désigne l'identificateur du dernier paramètre formel fixe de la fonction. Après traitement des paramètres, on libère la liste à l'aide de la `va_end`:

```
va_end(liste_parametres);
```

On accède aux différents paramètres de liste par la macro `va_arg` qui retourne le paramètre suivant de la liste:

```
va_arg(liste_parametres, type)
```

où `type` est le type supposé du paramètre auquel on accède.

Notons que l'utilisateur doit lui-même gérer le nombre de paramètres de la liste. Pour cela, on utilise généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Cette méthode est utilisée dans le programme suivant, où la fonction `add` effectue la somme de ses paramètres en nombre quelconque.

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

int add(int,...);

int add(int nb,...)
{
  int res = 0;
  int i;
  va_list liste_parametres;

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

int main(void)
{
  printf("\n %d", add(4,10,2,8,5));
  printf("\n %d\n", add(6,10,15,5,2,8,10));
  return(EXIT_SUCCESS);
}
```

Chapitre 5

Les directives au préprocesseur

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de *directives*. Les différentes directives au préprocesseur, introduites par le caractère `#`, ont pour but :

– l'incorporation de fichiers source (`#include`),

– la définition de constantes symboliques et de macros (`#define`),

– la compilation conditionnelle (`#if`, `#ifdef`,...).

5.1 La directive `#include`

Elle permet d'incorporer dans le fichier source le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (`stdio.h`, `math.h`,...) ou n'importe quel autre fichier. La directive `#include` possède deux syntaxes voisines :

```
#include <nom-de-fichier>
```

recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, `/usr/include/`) ;

```
#include "nom-de-fichier"
```

recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option `-I` du compilateur (*cf.* page 10).

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

5.2 La directive `#define`

La directive `#define` permet de définir :

– des constantes symboliques,

– des macros avec paramètres.

5.2.1 Définition de constantes symboliques

La directive

`#define` *nom reste-de-la-ligne*

demande au préprocesseur de substituer toute occurence de *nom* par la chaîne de caractères *reste-de-la-ligne* dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée. Par exemple :

```
#define NB_LIGNES 10
#define NB_COLONNES 33
#define TAILLE_MATRICE NB_LIGNES * NB_COLONNES
```

Il n'y a toutefois aucune contrainte sur la chaîne de caractères *reste-de-la-ligne*. On peut écrire

```
#define BEGIN {
#define END }
```

5.2.2 Définition de macros

Une macro avec paramètres se définit de la manière suivante :

`#define` *nom(liste-de-paramètres) corps-de-la-macro*

où *liste-de-paramètres* est une liste d'identificateurs séparés par des virgules. Par exemple, avec la directive

```
#define MAX(a,b) (a > b ? a : b)
```

le processeur remplacera dans la suite du code toutes les occurences du type

```
MAX(x,y)
```

où `x` et `y` sont des symboles quelconques par

```
(x > y ? x : y)
```

Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.

La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante. Ainsi, si l'on écrit par erreur

```
#define CARRE (a) a * a
```

la chaîne de caractères `CARRE(2)` sera remplacée par

```
(a) a * a (2)
```

Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères. En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses :

```
#define CARRE(a) a * a
```

le préprocesseur remplacera `CARRE(a + b)` par `a + b * a + b` et non par `(a + b) * (a + b)`. De même, `!CARRE(x)` sera remplacé par `! x * x` et non par `!(x * x)`.

Enfin, il faut être attentif aux éventuels effets de bord que peut entraîner l'usage de macros. Par exemple, `CARRE(x++)` aura pour expansion `(x++) * (x++)`. L'opérateur d'incrémentation sera donc appliqué deux fois au lieu d'une.

5.3 La compilation conditionnelle

La *compilation conditionnelle* a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- la valeur d'une expression
- l'existence ou l'inexistence de symboles.

5.3.1 Condition liée à la valeur d'une expression

Sa syntaxe la plus générale est :

```
#if  condition-1
   partie-du-programme-1
#elif  condition-2
   partie-du-programme-2
     ...
#elif  condition-n
   partie-du-programme-n
#else
   partie-du-programme-∞
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque *`condition-i`* doit être une expression constante.

Une seule *`partie-du-programme`* sera compilée : celle qui correspond à la première *`condition-i`* non nulle, ou bien la *`partie-du-programme`*-∞ si toutes les conditions sont nulles.

Par exemple, on peut écrire

```
#define PROCESSEUR ALPHA
```

```
#if PROCESSEUR == ALPHA
  taille_long = 64;
#elif PROCESSEUR == PC
  taille_long = 32;
#endif
```

5.3.2 Condition liée à l'existence d'un symbole

Sa syntaxe est

```
#ifdef  symbole
   partie-du-programme-1
#else  condition-2
   partie-du-programme-2
#endif
```

Si *symbole* est défini au moment où l'on rencontre la directive `#ifdef`, alors *partie-du-programme-1* sera compilée et *partie-du-programme-2* sera ignorée. Dans le cas contraire, c'est *partie-du-programme-2* qui sera compilée. La directive `#else` est évidemment facultative.

Da façon similaire, on peut tester la non-existence d'un symbole par :

```
#ifndef  symbole
   partie-du-programme-1
#else  condition-2
   partie-du-programme-2
#endif
```

Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme :

```
#define DEBUG
  ....
#ifdef DEBUG
  for (i = 0; i < N; i++)
    printf("%d\n",i);
#endif /* DEBUG */
```

Il suffit alors de supprimer la directive `#define DEBUG` pour que les instructions liées au débogage ne soient pas compilées. Cette dernière directive peut être remplacée par l'option de compilation `-D`*symbole*, qui permet de définir un symbole. On peut remplacer

```
#define DEBUG
```

en compilant le programme par

```
gcc -DDEBUG fichier.c
```

Chapitre 6

La gestion des fichiers

Le C offre la possibilité de lire et d'écrire des données dans un fichier.

Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (*buffer*), ce qui permet de réduire le nombre d'accès aux périphériques (disque...).

Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture) ... Ces informations sont rassemblées dans une structure dont le type, `FILE *`, est défini dans `stdio.h`. Un objet de type `FILE *` est appelé *flot de données* (en anglais, stream).

Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande `fopen`. Cette fonction prend comme argument le nom du fichier, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction `fclose`.

6.1 Ouverture et fermeture d'un fichier

6.1.1 La fonction `fopen`

Cette fonction, de type `FILE*` ouvre un fichier et lui associe un flot de données. Sa syntaxe est :

```
fopen("nom-de-fichier","mode")
```

La valeur retournée par `fopen` est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur `NULL`. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction `fopen` est égale à `NULL` afin de détecter les erreurs (lecture d'un fichier inexistant...).

Le premier argument de `fopen` est le nom du fichier concerné, fourni sous forme d'une chaîne de caractères. On préférera définir le nom du fichier par une constante symbolique au moyen de la directive `#define` plutôt que d'expliciter le nom de fichier dans le corps du programme.

Le second argument, *`mode`*, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré. On distingue

- les *fichiers textes*, pour lesquels les caractères de contrôle (retour à la ligne ...) seront interprétés en tant que tels lors de la lecture et de l'écriture ;
- les *fichiers binaires*, pour lesquels les caractères de contrôle se sont pas interprétés.

Les différents modes d'accès sont les suivants :

`"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
`"rb"`	ouverture d'un fichier binaire en lecture
`"wb"`	ouverture d'un fichier binaire en écriture
`"ab"`	ouverture d'un fichier binaire en écriture à la fin
`"r+"`	ouverture d'un fichier texte en lecture/écriture
`"w+"`	ouverture d'un fichier texte en lecture/écriture
`"a+"`	ouverture d'un fichier texte en lecture/écriture à la fin
`"r+b"`	ouverture d'un fichier binaire en lecture/écriture
`"w+b"`	ouverture d'un fichier binaire en lecture/écriture
`"a+b"`	ouverture d'un fichier binaire en lecture/écriture à la fin

Ces modes d'accès ont pour particularités :

- Si le mode contient la lettre `r`, le fichier doit exister.
- Si le mode contient la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu.
- Si le mode contient la lettre `a`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standard peuvent être utilisés en C sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- `stdin` (standard input) : unité d'entrée (par défaut, le clavier) ;
- `stdout` (standard output) : unité de sortie (par défaut, l'écran) ;
- `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur `stderr` afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

6.1.2 La fonction `fclose`

Elle permet de fermer le flot qui a été associé à un fichier par la fonction `fopen`. Sa syntaxe est :

`fclose(`*`flot`*`)`

où *`flot`* est le flot de type `FILE*` retourné par la fonction `fopen` correspondant.

La fonction `fclose` retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

6.2 Les entrées-sorties formatées

6.2.1 La fonction d'écriture `fprintf`

La fonction `fprintf`, analogue à `printf`, permet d'écrire des données dans un fichier. Sa syntaxe est

`fprintf(`*`flot`*`,"`*`chaîne de contrôle`*`",`*`expression-1, ..., expression-n`*`)`

où *`flot`* est le flot de données retourné par la fonction `fopen`. Les spécifications de format utilisées pour la fonction `fprintf` sont les mêmes que pour `printf` (*cf.* page 30).

6.2.2 La fonction de saisie `fscanf`

La fonction `fscanf`, analogue à `scanf`, permet de lire des données dans un fichier. Sa syntaxe est semblable à celle de `scanf` :

`fscanf(`*`flot`*`,"`*`chaîne de contrôle`*`",`*`argument-1,...,argument-n`*`)`

où *`flot`* est le flot de données retourné par `fopen`. Les spécifications de format sont ici les mêmes que celles de la fonction `scanf` (*cf.* page 32).

6.3 Impression et lecture de caractères

Similaires aux fonctions `getchar` et `putchar`, les fonctions `fgetc` et `fputc` permettent respectivement de lire et d'écrire un caractère dans un fichier. La fonction `fgetc`, de type `int`, retourne le caractère lu dans le fichier. Elle retourne la constante `EOF` lorsqu'elle détecte la fin du fichier. Son prototype est

`int fgetc(FILE* `*`flot`*`);`

où *`flot`* est le flot de type `FILE*` retourné par la fonction `fopen`. Comme pour la fonction `getchar`, il est conseillé de déclarer de type `int` la variable destinée à recevoir la valeur de retour de `fgetc` pour pouvoir détecter correctement la fin de fichier (*cf.* page 33).

La fonction `fputc` écrit *`caractere`* dans le flot de données :

`int fputc(int `*`caractere`*`, FILE *`*`flot`*`)`

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

Il existe également deux versions optimisées des fonctions `fgetc` et `fputc` qui sont implémentées par des macros. Il s'agit respectivement de `getc` et `putc`. Leur syntaxe est similaire à celle de `fgetc` et `fputc` :

`int getc(FILE* `*`flot`*`);`
`int putc(int `*`caractere`*`, FILE *`*`flot`*`)`

Ainsi, le programme suivant lit le contenu du fichier texte `entree`, et le recopie caractère par caractère dans le fichier `sortie` :

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

```
#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"

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

  if ((f_in = fopen(ENTREE,"r")) == NULL)
    {
      fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
      return(EXIT_FAILURE);
    }
  if ((f_out = fopen(SORTIE,"w")) == NULL)
    {
      fprintf(stderr, "\nErreur: Impossible d'ecrire dans le fichier %s\n", \
SORTIE);
      return(EXIT_FAILURE);
    }
  while ((c = fgetc(f_in)) != EOF)
    fputc(c, f_out);
  fclose(f_in);
  fclose(f_out);
  return(EXIT_SUCCESS);
}
```

6.4 Relecture d'un caractère

Il est possible de replacer un caractère dans un flot au moyen de la fonction `ungetc` :

`int ungetc(int` *`caractere`*`, FILE *`*`flot`*`);`

Cette fonction place le caractère *`caractere`* (converti en `unsigned char`) dans le flot *`flot`*. En particulier, si *`caractere`* est égal au dernier caractère lu dans le flot, elle annule le déplacement provoqué par la lecture précédente. Toutefois, `ungetc` peut être utilisée avec n'importe quel caractère (sauf `EOF`). Par exemple, l'exécution du programme suivant

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"

int main(void)
{
  FILE *f_in;
  int c;

  if ((f_in = fopen(ENTREE,"r")) == NULL)
    {
```

```
      fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
      return(EXIT_FAILURE);
    }

  while ((c = fgetc(f_in)) != EOF)
    {
      if (c == '0')
        ungetc('.',f_in);
      putchar(c);
    }
  fclose(f_in);
  return(EXIT_SUCCESS);
}
```

sur le fichier `entree.txt` dont le contenu est `097023` affiche à l'écran `0.970.23`

6.5 Les entrées-sorties binaires

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entrée-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines.

Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé. Leurs prototypes sont :

```
size_t fread(void *pointeur, size_t taille, size_t nombre, FILE *flot);
size_t fwrite(void *pointeur, size_t taille, size_t nombre, FILE *flot);
```

où *pointeur* est l'adresse du début des données à transférer, *taille* la taille des objets à transférer, *nombre* leur nombre. Rappelons que le type `size_t`, défini dans `stddef.h`, correspond au type du résultat de l'évaluation de `sizeof`. Il s'agit du plus grand type entier non signé.

La fonction `fread` lit les données sur le flot *flot* et la fonction `fwrite` les écrit. Elles retournent toutes deux le nombre de données transférées.

Par exemple, le programme suivant écrit un tableau d'entiers (contenant les 50 premiers entiers) avec `fwrite` dans le fichier `sortie`, puis lit ce fichier avec `fread` et imprime les éléments du tableau.

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

#define NB 50
#define F_SORTIE "sortie"

int main(void)
{
  FILE *f_in, *f_out;
  int *tab1, *tab2;
  int i;
```

```
  tab1 = (int*)malloc(NB * sizeof(int));
  tab2 = (int*)malloc(NB * sizeof(int));
  for (i = 0 ; i < NB; i++)
    tab1[i] = i;

  /* ecriture du tableau dans F_SORTIE */
  if ((f_out = fopen(F_SORTIE, "w")) == NULL)
    {
      fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  fwrite(tab1, NB * sizeof(int), 1, f_out);
  fclose(f_out);

  /* lecture dans F_SORTIE */
  if ((f_in = fopen(F_SORTIE, "r")) == NULL)
    {
      fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  fread(tab2, NB * sizeof(int), 1, f_in);
  fclose(f_in);
  for (i = 0 ; i < NB; i++)
    printf("%d\t",tab2[i]);
  printf("\n");
  return(EXIT_SUCCESS);
}
```

Les éléments du tableau sont bien affichés à l'écran. Par contre, on constate que le contenu du fichier `sortie` n'est pas encodé.

6.6 Positionnement dans un fichier

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en *mode séquentiel*: les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en *mode direct*, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction `fseek` permet de se positionner à un endroit précis ; elle a pour prototype :

```
int fseek(FILE *flot, long deplacement, int origine);
```

La variable ***deplacement*** détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'octets. La variable ***origine*** peut prendre trois valeurs :

- `SEEK_SET` (égale à 0) : début du fichier ;
- `SEEK_CUR` (égale à 1) : position courante ;

– `SEEK_END` (égale à 2) : fin du fichier.

La fonction

```
int rewind(FILE *flot);
```

permet de se positionner au début du fichier. Elle est équivalente à `fseek(flot, 0, SEEK_SET);`

La fonction

```
long ftell(FILE *flot);
```

retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

Par exemple

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

#define NB 50
#define F_SORTIE "sortie"

int main(void)
{
  FILE *f_in, *f_out;
  int *tab;
  int i;

  tab = (int*)malloc(NB * sizeof(int));
  for (i = 0 ; i < NB; i++)
    tab[i] = i;

  /* ecriture du tableau dans F_SORTIE */
  if ((f_out = fopen(F_SORTIE, "w")) == NULL)
    {
      fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  fwrite(tab, NB * sizeof(int), 1, f_out);
  fclose(f_out);

  /* lecture dans F_SORTIE */
  if ((f_in = fopen(F_SORTIE, "r")) == NULL)
    {
      fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }

  /* on se positionne a la fin du fichier */
  fseek(f_in, 0, SEEK_END);
```

```
  printf("\n position %ld", ftell(f_in));
  /* deplacement de 10 int en arriere */
  fseek(f_in, -10 * sizeof(int), SEEK_END);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);
  /* retour au debut du fichier */
  rewind(f_in);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);
  /* deplacement de 5 int en avant */
  fseek(f_in, 5 * sizeof(int), SEEK_CUR);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d\n", i);
  fclose(f_in);
  return(EXIT_SUCCESS);
}
```

L'exécution de ce programme affiche à l'écran :

```
 position 200
 position 160    i = 40
 position 0      i = 0
 position 24     i = 6
```

On constate en particulier que l'emploi de la fonction `fread` provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante.

Chapitre 7

La programmation modulaire

Dès que l'on écrit un programme de taille importante ou destiné à être utilisé et maintenu par d'autres personnes, il est indispensable de se fixer un certain nombre de règles d'écriture. En particulier, il est nécessaire de fractionner le programme en plusieurs fichiers sources, que l'on compile séparemment.

Ces règles d'écriture ont pour objectifs de rendre un programme lisible, portable, réutilisable, facile à maintenir et à modifier.

7.1 Principes élémentaires

Trois principes essentiels doivent guider l'écriture d'un programme C.

L'abstraction des constantes littérales L'utilisation explicite de constantes littérales dans le corps d'une fonction rend les modifications et la maintenance difficiles. Des instructions comme :

```
fopen("mon_fichier", "r");
perimetre = 2 * 3.14 * rayon;
```

sont à proscrire.

Sauf cas très particuliers, les constantes doivent être définies comme des constantes symboliques au moyen de la directive `#define`.

La factorisation du code Son but est d'éviter les duplications de code. La présence d'une même portion de code à plusieurs endroits du programme est un obstacle à d'éventuelles modifications. Les fonctions doivent donc être systématiquement utilisées pour éviter la duplication de code. Il ne faut pas craindre de définir une multitude de fonctions de petite taille.

La fragmentation du code Pour des raisons de lisibilité, il est nécessaire de découper un programme en plusieurs fichiers. De plus, cette règle permet de réutiliser facilement une partie du code pour d'autres applications. Une possibilité est de placer une partie du code dans un fichier en-tête (ayant l'extension `.h`) que l'on inclut dans le fichier contenant le programme principal à l'aide de la directive `#include`. Par exemple, pour écrire un programme qui saisit deux entiers au clavier et affiche leur produit, on peut placer la fonction `produit` dans un fichier `produit.h`, et l'inclure dans le fichier `main.c` au moment du traitement par le préprocesseur.

```
/*****************************************************************/
/***  fichier: main.c                                          ***/
/***  saisit 2 entiers et affiche leur produit                 ***/
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "produit.h"

int main(void)
{
  int a, b, c;
  scanf("%d",&a);
  scanf("%d",&b);
  c = produit(a,b);
  printf("\nle produit vaut %d\n",c);
  return EXIT_SUCCESS;
}

/*****************************************************************/
/***  fichier: produit.h                                       ***/
/***  produit de 2 entiers                                     ***/
/*****************************************************************/

int produit(int, int);

int produit(int a, int b)
{
  return(a * b);
}
```

Cette technique permet juste de rendre le code plus lisible, puisque le fichier effectivement compilé (celui produit par le préprocesseur) est unique et contient la totalité du code.

Une méthode beaucoup plus pratique consiste à découper le code en plusieurs fichiers sources que l'on compile séparemment. Cette technique, appelée *compilation séparée*, facilite également le débogage.

7.2 La compilation séparée

Si l'on reprend l'exemple précédent, le programme sera divisé en deux fichiers : `main.c` et `produit.c`. Cette fois-ci, le fichier `produit.c` n'est plus inclus dans le fichier principal. Les deux fichiers seront compilés séparément ; les deux fichiers objets produits par la compilation seront liés lors l'édition de liens. Le détail de la compilation est donc :

```
gcc -c produit.c
```

```
gcc -c main.c
gcc main.o produit.o
```

La succession de ces trois commandes peut également s'écrire

```
gcc produit.c main.c
```

Toutefois, nous avons vu au chapitre 4, page 61, qu'il était risqué d'utiliser une fonction sans l'avoir déclarée. C'est ici le cas, puisque quand il compile le programme `main.c`, le compilateur ne dispose pas de la déclaration de la fonction `produit`. L'option `-Wall` de `gcc` signale

```
main.c:15: warning: implicit declaration of function `produit'
```

Il faut donc rajouter cette déclaration dans le corps du programme `main.c`.

7.2.1 Fichier en-tête d'un fichier source

Pour que le programme reste modulaire, on place en fait la déclaration de la fonction `produit` dans un fichier en-tête `produit.h` que l'on inclut dans `main.c` à l'aide de `#include`.

Une règle d'écriture est donc d'associer à chaque fichier source *nom*`.c` un fichier en-tête *nom*`.h` comportant les déclarations des fonctions non locales au fichier *nom*`.c`, (ces fonctions sont appelées *fonctions d'interface*) ainsi que les définitions des constantes symboliques et des macros qui sont partagées par les deux fichiers. Le fichier en-tête *nom*`.h` doit être inclus par la directive `#include` dans tous les fichiers sources qui utilisent une des fonctions définies dans *nom*`.c`, ainsi que dans le fichier *nom*`.c`. Cette dernière inclusion permet au compilateur de vérifier que la définition de la fonction donnée dans *nom*`.c` est compatible avec sa déclaration placée dans *nom*`.h`. C'est exactement la procédure que l'on utilise pour les fonctions de la librairie standard : les fichiers `.h` de la librairie standard sont constitués de déclarations de fonctions et de définitions de constantes symboliques.

Par ailleurs, il faut faire précéder la déclaration de la fonction du mot-clef `extern`, qui signifie que cette fonction est définie dans un autre fichier. Le programme effectuant le produit se décompose donc en trois fichiers de la manière suivante.

```
/**********************************************************************/
/***  fichier: produit.h                                            ***/
/***  en-tete de produit.c                                          ***/
/**********************************************************************/

extern int produit(int, int);

/**********************************************************************/
/***  fichier: produit.c                                            ***/
/***  produit de 2 entiers                                          ***/
/**********************************************************************/
#include "produit.h"

int produit(int a, int b)
{
  return(a * b);
}
```

```
/************************************************************************/
/***  fichier: main.c                                                 ***/
/***  saisit 2 entiers et affiche leur produit                        ***/
/************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "produit.h"

int main(void)
{
  int a, b, c;
  scanf("%d",&a);
  scanf("%d",&b);
  c = produit(a,b);
  printf("\nle produit vaut %d\n",c);
  return EXIT_SUCCESS;
}
```

Une dernière règle consiste à éviter les possibilités de double inclusion de fichiers en-tête. Pour cela, il est recommandé de définir une constante symbolique, habituellement appelée *NOM*_H, au début du fichier *nom*.h dont l'existence est précédemment testée. Si cette constante est définie, c'est que le fichier *nom*.h a déjà été inclus. Dans ce cas, le préprocesseur ne le prend pas en compte. Sinon, on définit la constante et on prend en compte le contenu de *nom*.h. En appliquant cette règle, le fichier `produit.h` de l'exemple précédent devient :

```
/************************************************************************/
/***  fichier: produit.h                                              ***/
/***  en-tete de produit.c                                            ***/
/************************************************************************/

#ifndef PRODUIT_H
#define PRODUIT_H

extern int produit(int, int);
#endif /* PRODUIT_H */
```

En résumé, les règles d'écriture sont les suivantes :

- A tout fichier source *nom*.c d'un programme on associe un fichier en-tête *nom*.h qui définit son interface.
- Le fichier *nom*.h se compose :
  - des déclarations des fonctions d'interface (celles qui sont utilisées dans d'autres fichiers sources) ;
  - d'éventuelles définitions de constantes symboliques et de macros ;
  - d'éventuelles directives au préprocesseur (inclusion d'autres fichiers, compilation conditionnelle).

- Le fichier `nom.c` se compose :
  - de variables permanentes, qui ne sont utilisées que dans le fichier `nom.c` ;
  - des fonctions d'interface dont la déclaration se trouve dans `nom.h` ;
  - d'éventuelles fonctions locales à `nom.c`.
- Le fichier `nom.h` est inclus dans le fichier `nom.c` et dans tous les autres fichiers qui font appel à une fonction d'interface définie dans `nom.c`.

Enfin, pour plus de lisibilité, il est recommandé de choisir pour toutes les fonctions d'interface définies dans `nom.c` un identificateur préfixé par le nom du fichier source, du type `nom_fonction`.

7.2.2 Variables partagées

Même si cela doit être évité, il est parfois nécessaire d'utiliser une variable commune à plusieurs fichiers sources. Dans ce cas, il est indispensable que le compilateur comprenne que deux variables portant le même nom mais déclarées dans deux fichiers différents correspondent en fait à un seul objet. Pour cela, la variable doit être déclarée une seule fois de manière classique. Cette déclaration correspond à une définition dans la mesure où le compilateur réserve un espace-mémoire pour cette variable. Dans les autres fichiers qui l'utilisent, il faut faire une référence à cette variable, sous forme d'une déclaration précédée du mot-clef `extern`. Contrairement aux déclarations classiques, une déclaration précédée de `extern` ne donne pas lieu à une réservation d'espace mémoire.

Ainsi, pour que les deux fichiers sources `main.c` et `produit.c` partagent une variable entière `x`, on peut définir `x` dans `produit.c` sous la forme

```
int x;
```

et y faire référence dans `main.c` par

```
extern int x;
```

7.3 L'utilitaire `make`

Losrqu'un programme est fragmenté en plusieurs fichiers sources compilés séparemment, la procédure de compilation peut devenir longue et fastidieuse. Il est alors extrèmement pratique de l'automatiser à l'aide de l'utilitaire `make` d'Unix. Une bonne utilisation de `make` permet de réduire le temps de compilation et également de garantir que celle-ci est effectuée correctement.

7.3.1 Principe de base

L'idée principale de `make` est d'effectuer uniquement les étapes de compilation nécessaires à la création d'un exécutable. Par exemple, si un seul fichier source a été modifié dans un programme composé de plusieurs fichiers, il suffit de recompiler ce fichier et d'effectuer l'édition de liens. Les autres fichiers sources n'ont pas besoin d'être recompilés.

La commande `make` recherche par défaut dans le répertoire courant un fichier de nom `makefile`, ou `Makefile` si elle ne le trouve pas. Ce fichier spécifie les dépendances entre les différents fichiers sources, objets et exécutables. Il est également possible de donner un autre nom au fichier `Makefile`. Dans ce cas, il faut lancer la commande `make` avec l'option `-f` *nom_de_fichier*.

7.3.2 Création d'un `Makefile`

Un fichier `Makefile` est composé d'une liste de règles de dépendance de la forme :

```
cible:  liste de dépendances
<TAB>   commandes UNIX
```

La première ligne spécifie un fichier cible, puis la liste des fichiers dont il dépend (séparés par des espaces). Les lignes suivantes, qui commencent par le caractère `TAB`, indiquent les commandes Unix à exécuter dans le cas où l'un des fichiers de dépendance est plus récent que le fichier cible.

Ainsi, un fichier `Makefile` pour le programme effectuant le produit de deux entiers peut être

```
## Premier exemple de Makefile

prod: produit.c main.c produit.h
	gcc -o prod -O3 produit.c main.c

prod.db: produit.c main.c produit.h
	gcc -o prod.db -g -O3 produit.c main.c
```

L'exécutable `prod` dépend des deux fichiers sources `produit.c` et `main.c`, ainsi que du fichier en-tête `produit.h`. Il résulte de la compilation de ces deux fichiers avec l'option d'optimisation `-O3`. L'exécutable `prod.db` utilisé par le débogueur est, lui, obtenu en compilant ces deux fichiers avec l'option `-g` nécessaire au débogage. Les commentaires sont précédés du caractère `#`.

Pour effectuer la compilation et obtenir un fichier cible, on lance la commande `make` suivie du nom du fichier cible souhaité, ici

```
make prod
```

ou

```
make prod.db
```

Par défaut, si aucun fichier cible n'est spécifié au lancement de `make`, c'est la première cible du fichier `Makefile` qui est prise en compte. Par exemple, si on lance pour la première fois `make`, la commande de compilation est effectuée puisque le fichier exécutable `prod` n'existe pas :

```
% make
gcc -o prod -O3 produit.c main.c
```

Si on lance cette commande une seconde fois sans avoir modifié les fichiers sources, la compilation n'est pas effectuée puisque le fichier `prod` est plus récent que les deux fichiers dont il dépend. On obtient dans ce cas :

```
% make
make: `prod' is up to date.
```

Le `Makefile` précédent n'utilise pas pleinement les fonctionnalités de `make`. En effet, la commande utilisée pour la compilation correspond en fait à trois opérations distinctes : la compilation des fichiers sources `produit.c` et `main.c`, qui produit respectivement les fichiers objets `produit.o` et `main.o`, puis l'édition de liens entre ces deux fichiers objet, qui produit l'exécutable `prod`. Pour utiliser pleinement `make`, il faut distinguer ces trois étapes. Le nouveau fichier `Makefile` devient alors :

```
## Deuxieme exemple de Makefile

prod: produit.o main.o
	gcc -o prod produit.o main.o
main.o: main.c produit.h
	gcc -c -O3 main.c
produit.o: produit.c produit.h
	gcc -c -O3 produit.c
```

Les fichiers objet `main.o` et `produit.o` dépendent respectivement des fichiers sources `main.c` et `produit.c`, et du fichier en-tête `produit.h`. Ils sont obtenus en effectuant la compilation de ces fichiers sources sans édition de liens (option `-c` de `gcc`), et avec l'option d'optimisation `-O3`. Le fichier exécutable `prod` est obtenu en effectuant l'édition de liens des fichiers `produit.o` et `main.o`. Lorsqu'on invoque la commande `make` pour la première fois, les trois étapes de compilation sont effectuées :

```
% make
gcc -c -O3 produit.c
gcc -c -O3 main.c
gcc -o prod produit.o main.o
```

Si l'on modifie le fichier `produit.c`, le fichier `main.o` est encore à jour. Seules deux des trois étapes de compilation sont exécutées :

```
% make
gcc -c -O3 produit.c
gcc -o prod produit.o main.o
```

De la même façon, il convient de détailler les étapes de compilation pour obtenir le fichier exécutable `prod.db` utilisé pour le débogage. Le fichier `Makefile` devient alors :

```
## Deuxieme exemple de Makefile

# Fichier executable prod
prod: produit.o main.o
	gcc -o prod produit.o main.o
main.o: main.c produit.h
	gcc -c -O3 main.c
produit.o: produit.c produit.h
	gcc -c -O3 produit.c

# Fichier executable pour le debuggage prod.db
```

```
prod.db: produit.do main.do
        gcc -o prod.db produit.do main.do
main.do: main.c produit.h
        gcc -o main.do -c -g -O3 main.c
produit.do: produit.c produit.h
        gcc -o produit.do -c -g -O3 produit.c
```

Pour déterminer facilement les dépendances entre les différents fichiers, on peut utiliser l'option `-MM` de `gcc`. Par exemple,

```
% gcc -MM produit.c main.c
produit.o: produit.c produit.h
main.o: main.c produit.h
```

On rajoute habituellement dans un fichier `Makefile` une cible appelée `clean` permettant de détruire tous les fichiers objets et exécutables créés lors de la compilation.

```
clean:
        rm -f prod prod.db *.o *.do
```

La commande `make clean` permet donc de "nettoyer" le répertoire courant. Notons que l'on utilise ici la commande `rm` avec l'option `-f` qui évite l'apparition d'un message d'erreur si le fichier à détruire n'existe pas.

7.3.3 Macros et abbréviations

Pour simplifier l'écriture d'un fichier `Makefile`, on peut utiliser un certain nombre de macros sous la forme

nom_de_macro = corps de la macro

Quand la commande `make` est exécutée, toutes les instances du type `$(`*nom_de_macro*`)` dans le `Makefile` sont remplacées par le corps de la macro. Par exemple, on peut définir une macro `CC` pour spécifier le compilateur utilisé (`cc` ou `gcc`), une macro `PRODUCTFLAGS` pour définir les options de compilation utilisées pour générer un fichier produit, une macro `DEBUGFLAGS` pour les options de compilation utilisées pour générer un fichier produit pour le débogage... Le fichier `Makefile` suivant donne un exemple :

```
## Exemple de Makefile avec macros

# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -c -O3
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -c -g -O3

# Fichier executable prod
prod: produit.o main.o
        $(CC) -o prod produit.o main.o
```

```
main.o: main.c produit.h
        $(CC) $(PRODUCTFLAGS) main.c
produit.o: produit.c  produit.h
        $(CC) $(PRODUCTFLAGS)  produit.c

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        $(CC) -o prod.db produit.do main.do
main.do: main.c produit.h
        $(CC) -o main.do $(DEBUGFLAGS) main.c
produit.do: produit.c produit.h
        $(CC) -o produit.do $(DEBUGFLAGS) produit.c
```

La commande `make` produit alors

```
% make
gcc -c -O3  produit.c
gcc -c -O3 main.c
gcc -o prod produit.o main.o
```

Cette écriture permet de faciliter les modifications du fichier `Makefile` : on peut maintenant aisément changer les options de compilation, le type de compilateur...

Un certain nombre de macros sont prédéfinies. En particulier,

- `$@` désigne le fichier cible courant :
- `$*` désigne le fichier cible courant privé de son suffixe :
- `$<` désigne le fichier qui a provoqué l'action.

Dans le `Makefile` précédent, la partie concernant la production de `main.do` peut s'écrire par exemple

```
main.do: main.c produit.h
        $(CC) -o $@ $(DEBUGFLAGS) $<
```

7.3.4 Règles générales de compilation

Il est également possible de définir dans un `Makefile` des règles générales de compilation correspondant à certains suffixes. On peut spécifier par exemple que tout fichier `.o` est obtenu en compilant le fichier `.c` correspondant avec les options définies par la macro `PRODUCTFLAGS`. Pour cela, il faut tout d'abord définir une liste de suffixes qui spécifient les fichiers cibles construits à partir d'une règle générale. Par exemple, avant de définir des règles de compilation pour obtenir les fichiers `.o` et `.do`, on écrit :

```
.SUFFIXES: .o .do
```

Une règle de compilation est ensuite définie de la façon suivante : on donne le suffixe du fichier que `make` doit chercher, suivi par le suffixe du fichier que `make` doit produire. Ces deux suffixes sont suivis par `:;` puis par une commande Unix (définie de la façon la plus générale possible). Les règles de production des fichiers `.o` et `.do` sont par exemple :

```
# regle de production d'un fichier .o
.c.o:; $(CC) -o $@ $(PRODUCTFLAGS) $<
# regle de production d'un fichier .do
.c.do:; $(CC) -o $@ $(DEBUGFLAGS)  $<
```

Si les fichiers `.o` ou `.do` dépendent également d'autres fichiers, il faut aussi spécifier ces dépendances. Ici, il faut préciser par exemple que ces fichiers dépendent aussi de `produit.h`. Le fichier `Makefile` a donc la forme suivante :

```
## Exemple de Makefile

# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -c -O3
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -c -g -O3

# suffixes correspondant a des regles generales
.SUFFIXES: .c .o .do
# regle de production d'un fichier .o
.c.o:; $(CC) -o $@ $(PRODUCTFLAGS) $<
# regle de production d'un fichier .do
.c.do:; $(CC) -o $@ $(DEBUGFLAGS)  $<

# Fichier executable prod
prod: produit.o main.o
        $(CC) -o prod produit.o main.o
produit.o: produit.c produit.h
main.o: main.c produit.h

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        $(CC) -o prod.db produit.do main.do
produit.do: produit.c produit.h
main.do: main.c produit.h

clean:
        rm -f prod prod.db *.o *.do
```

Annexe A

La librairie standard

Cette annexe donne la syntaxe des principales fonctions de la librairie standard. Une liste exhaustive de toutes les fonctions disponibles figure à l'annexe B de l'ouvrage de Kernighan et Richie [6]. Pour obtenir plus d'informations sur ces fonctions, il suffit de consulter les pages de `man` correspondant.

A.1 Entrées-sorties <stdio.h>

A.1.1 Manipulation de fichiers

L'usage des fonctions de manipulation de fichiers suivantes est détaillé au chapitre 6, page 81.

fonction	action
`fopen`	ouverture d'un fichier
`fclose`	fermeture d'un fichier
`fflush`	écriture des buffers en mémoire dans le fichier

A.1.2 Entrées et sorties formatées

La syntaxe de ces fonctions et leur action sont décrites aux paragraphes 1.11 et 6.2-6.3.

fonction	prototype	action
`fprintf`	`int fprintf(FILE *stream, char *format, ...)`	écriture sur un fichier
`fscanf`	`int fscanf(FILE *stream, char *format, ...)`	lecture depuis un fichier
`printf`	`int printf(char *format, ...)`	écriture sur la sortie standard
`scanf`	`int scanf(char *format, ...)`	lecture depuis l'entrée standard
`sprintf`	`int sprintf(char *s, char *format, ...)`	écriture dans la chaîne de caractères `s`
`sscanf`	`int sscanf(char *s, char *format, ...)`	lecture depuis la chaîne de caractères `s`

A.1.3 Impression et lecture de caractères

fonction	prototype	action
`fgetc`	`int fgetc(FILE *stream)`	lecture d'un caractère depuis un fichier
`fputc`	`int fputc(int c, FILE *stream)`	écriture d'un caractère sur un fichier
`getc`	`int getc(FILE *stream)`	équivalent de `fgetc` mais implémenté par une macro
`putc`	`int putc(int c, FILE *stream)`	équivalent de `fputc` mais implémenté par une macro
`getchar`	`int getchar(void)`	lecture d'un caractère depuis l'entrée standard
`putchar`	`int putchar(int c)`	écriture d'un caractère sur la sortie standard
`fgets`	`char *fgets(char *s, FILE *stream)`	lecture d'une chaîne de caractères depuis un fichier
`fputs`	`int *fputs(char *s, FILE *stream)`	écriture d'une chaîne de caractères sur un fichier
`gets`	`char *gets(char *s)`	lecture d'une chaîne de caractères sur l'entrée standard
`puts`	`int *puts(char *s)`	écriture d'une chaîne de caractères sur la sortie standard

A.2 Manipulation de caractères <ctype.h>

Toutes les fonctions ci-dessous permettent de tester une propriété du caractère passé en paramètre. Elles renvoient la valeur 1 si le caractère vérifie la propriété et 0 sinon. Leur prototype est :

```
int fonction(char c)
```

fonction	renvoie 1 si le caractère est
`isalnum`	une lettre ou un chiffre
`isalpha`	une lettre
`iscntrl`	un caractère de commande
`isdigit`	un chiffre décimal
`isgraph`	un caractère imprimable ou le blanc
`islower`	une lettre minuscule
`isprint`	un caractère imprimable (pas le blanc)
`ispunct`	un caractère imprimable qui n'est ni une lettre ni un chiffre
`isspace`	un blanc
`isupper`	une lettre majuscule
`isxdigit`	un chiffre hexadécimal

On dispose également de deux fonctions permettant la conversion entre lettres minuscules et lettres majuscules :

fonction	prototype	action
`tolower`	`int tolower(int c)`	convertit `c` en minuscule si c'est une lettre majuscule, retourne `c` sinon.
`toupper`	`int toupper(int c)`	convertit `c` en majuscule si c'est une lettre minuscule, retourne `c` sinon.

A.3 Manipulation de chaînes de caractères <string.h>

fonction	prototype	action
`strcpy`	`char *strcpy(char *ch1, char *ch2)`	copie la chaîne `ch2` dans la chaîne `ch1` ; retourne `ch1`.
`strncpy`	`char *strncpy(char *ch1, char *ch2, int n)`	copie `n` caractères de la chaîne `ch2` dans la chaîne `ch1` ; retourne `ch1`.
`strcat`	`char *strcat(char *ch1, char *ch2)`	copie la chaîne `ch2` à la fin de la chaîne `ch1` ; retourne `ch1`.
`strncat`	`char *strncat(char *ch1, char *ch2, int n)`	copie `n` caractères de la chaîne `ch2` à la fin de la chaîne `ch1` ; retourne `ch1`.
`strcmp`	`int strcmp(char *ch1, char *ch2)`	compare `ch1` et `ch2` pour l'ordre lexicographique ; retourne une valeur négative si `ch1` est inférieure à `ch2`, une valeur positive si `ch1` est supérieure à `ch2`, 0 si elles sont identiques.
`strncmp`	`int strncmp(char *ch1, char *ch2, int n)`	compare les `n` premiers caractères de `ch1` et `ch2`.
`strchr`	`char *strchr(char *chaine, char c)`	retourne un pointeur sur la première occurence de `c` dans `chaine`, et `NULL` si `c` n'y figure pas.
`strrchr`	`char *strrchr(char *chaine, char c)`	retourne un pointeur sur la dernière occurence de `c` dans `chaine`, et `NULL` si `c` n'y figure pas.
`strstr`	`char *strshr(char *ch1, char *ch2)`	retourne un pointeur sur la première occurence de `ch2` dans `ch1`, et `NULL` si `ch2` n'y figure pas.
`strlen`	`int strlen(char *chaine)`	retourne la longueur de `chaine`.

A.4 Fonctions mathématiques <math.h>

Le résultat et les paramètres de toutes ces fonctions sont de type `double`. Si les paramètres effectifs sont de type `float`, ils seront convertis en `double` par le compilateur.

fonction	action
`acos`	arc cosinus
`asin`	arc sinus
`atan`	arc tangente
`cos`	cosinus
`sin`	sinus
`tan`	tangente
`cosh`	cosinus hyperbolique
`sinh`	cosinus hyperbolique
`tanh`	tangente hyperbolique
`exp`	exponentielle
`log`	logarithme népérien
`log10`	logarithme en base 10
`pow`	puissance
`sqrt`	racine carrée
`fabs`	valeur absolue
`fmod`	reste de la division
`ceil`	partie entière supérieure
`floor`	partie entière inférieure

A.5 Utilitaires divers <stdlib.h>

A.5.1 Allocation dynamique

Ces fonctions sont décrites au chapitre 3, paragraphe 3.4.

fonction	action
`calloc`	allocation dynamique et initialisation à zéro.
`malloc`	allocation dynamique
`realloc`	modifie la taille d'une zone préalablement allouée par `calloc` ou `malloc`.
`free`	libère une zone mémoire

A.5.2 Conversion de chaînes de caractères en nombres

Les fonctions suivantes permettent de convertir une chaîne de caractères en un nombre.

fonction	prototype	action
`atof`	`double atof(char *chaine)`	convertit `chaine` en un `double`
`atoi`	`int atoi(char *chaine)`	convertit `chaine` en un `int`
`atol`	`long atol(char *chaine)`	convertit `chaine` en un `long int`

A.5.3 Génération de nombres pseudo-aléatoires

La fonction `rand` fournit un nombre entier pseudo-aléatoire dans l'intervalle [0,`RAND_MAX`], où `RAND_MAX` est une constante prédéfinie au moins égale à $2^{15} - 1$. L'aléa fournit par la fonction `rand` n'est toutefois pas de très bonne qualité.

La valeur retournée par `rand` dépend de l'initialisation (germe) du générateur. Cette dernière est égale à 1 par défaut mais elle peut être modifiée à l'aide de la fonction `srand`.

fonction	prototype	action
`rand`	`int rand(void)`	fournit un nombre entier pseudo-aléatoire
`srand`	`void srand(unsigned int germe)`	modifie la valeur de l'initialisation du générateur pseudo-aléatoire utilisé par `rand`.

A.5.4 Arithmétique sur les entiers

fonction	prototype	action
`abs`	`int abs(int n)`	valeur absolue d'un entier
`labs`	`long labs(long n)`	valeur absolue d'un `long int`
`div`	`div_t div(int a, int b)`	quotient et reste de la division euclidienne de `a` par `b`. Les résultats sont stockés dans les champs `quot` et `rem` (de type `int`) d'une structure de type `div_t`.
`ldiv`	`ldiv_t ldiv(long a, long b)`	quotient et reste de la division euclidienne de `a` par `b`. Les résultats sont stockés dans les champs `quot` et `rem` (de type `long int`) d'une structure de type `ldiv_t`.

A.5.5 Recherche et tri

Les fonctions `qsort` et `bsearch` permettent respectivement de trier un tableau, et de rechercher un élément dans un tableau déjà trié. Leur syntaxe est détaillée au chapitre 4, page 71.

A.5.6 Communication avec l'environnement

fonction	prototype	action
`abort`	`void abort(void)`	terminaison anormale du programme
`exit`	`void exit(int etat)`	terminaison du programme ; rend le contrôle au système en lui fournissant la valeur `etat` (la valeur 0 est considérée comme une fin normale).
`system`	`int system(char *s)`	exécution de la commande système définie par la chaîne de caractères `s`.

A.6 Date et heure <time.h>

Plusieurs fonctions permettent d'obtenir la date et l'heure. Le temps est représenté par des objets de type `time_t` ou `clock_t`, lesquels correspondent généralement à des `int` ou à des `long int`.

fonction	prototype	action
`time`	`time_t time(time_t *tp)`	retourne le nombre de secondes écoulées depuis le 1er janvier 1970, 0 heures G.M.T. La valeur retournée est assignée à `*tp`.
`difftime`	`double difftime(time_t t1, time_t t2)`	retourne la différence `t1 - t2` en secondes.
`ctime`	`char *ctime(time_t *tp)`	convertit le temps système `*tp` en une chaîne de caractères explicitant la date et l'heure sous un format prédéterminé.
`clock`	`clock_t clock(void)`	retourne le temps CPU en microsecondes utilisé depuis le dernier appel à `clock`.

Annexe B

Le débogueur GDB

Le logiciel `gdb` est un logiciel GNU permettant de déboguer les programmes C (et C++). Il permet de répondre aux questions suivantes :

- à quel endroit s'arrête le programme en cas de terminaison incorrecte, notamment en cas d'erreur de segmentation ?
- quelles sont les valeurs des variables du programme à un moment donné de l'exécution ?
- quelle est la valeur d'une expression donnée à un moment précis de l'exécution ?

`Gdb` permet donc de lancer le programme, d'arrêter l'exécution à un endroit précis, d'examiner et de modifier les variables au cours de l'exécution et aussi d'exécuter le programme pas-à-pas.

B.1 Démarrer `gdb`

Pour pouvoir utiliser le débogueur, il faut avoir compilé le programme avec l'option `-g` de `gcc`. Cette option génère des informations symboliques nécessaires au débogueur. Par exemple:

```
gcc -g -Wall -ansi -o exemple exemple.c
```

On peut ensuite lancer `gdb` sous le shell par la commande

`gdb` *`nom de l'exécutable`*

Toutefois, il est encore plus pratique d'utiliser `gdb` avec l'interface offerte par `Emacs`. Pour lancer `gdb` sous `Emacs`, il faut utiliser la commande

```
M-x gdb
```

où `M-x` signifie qu'il faut appuyer simultanément sur la touche `Méta` (`Alt` sur la plupart des claviers) et sur `x`. `Emacs` demande alors le nom du fichier exécutable à déboguer : il affiche dans le mini-buffer

```
Run gdb (like this): gdb
```

Quand on entre le nom d'exécutable, `gdb` se lance : le lancement fournit plusieurs informations sur la version utilisée et la licence GNU. Puis, le prompt de gdb s'affiche :

```
(gdb)
```

On peut alors commencer à déboguer le programme.

On est souvent amené au cours du débogage à corriger une erreur dans le fichier source et à recompiler. Pour pouvoir travailler avec le nouvel exécutable sans avoir à quitter `gdb`, il faut le redéfinir à l'aide de la commande `file` :

```
(gdb) file nom_executable
```

B.2 Quitter gdb

Une fois le débogage terminé, on quitte `gdb` par la commande

```
(gdb) quit
```

Parfois, `gdb` demande une confirmation :

```
The program is running.  Exit anyway? (y or n)
```

Il faut évidemment taper `y` pour quitter le débogueur.

B.3 Exécuter un programme sous gdb

Pour exécuter un programme sous `gdb`, on utilise la commande `run` :

```
(gdb) run [arguments du programme]
```

où *arguments du programme* sont, s'il y en a, les arguments de votre programme. On peut également utiliser comme arguments les opérateurs de redirection, par exemple :

```
(gdb) run 3 5 > sortie
```

`gdb` lance alors le programme exactement comme s'il avait été lancé avec les mêmes arguments :

```
./a.out 3 5 > sortie
```

Comme la plupart des commandes de base de `gdb`, `run` peut être remplacé par la première lettre du nom de la commande, `r`. On peut donc écrire également

```
(gdb) r 3 5 > sortie
```

On est souvent amené à exécuter plusieurs fois un programme pour le déboguer. Par défaut, `gdb` réutilise donc les arguments du précédent appel de `run` si on utilise `run` sans arguments.

À tout moment, la commande `show args` affiche la liste des arguments passés lors du dernier appel de `run` :

```
(gdb) show args
Argument list to give program being debugged when it is started is "3
5 > sortie".
(gdb)
```

Si rien ne s'y oppose et que le programme s'exécute normalement, on atteint alors la fin du programme. `gdb` affiche alors à la fin de l'exécution

```
Program exited normally.
(gdb)
```

B.4 Terminaison anormale du programme

Dans toute la suite, on prendra pour exemple le programme de la page 110, dont le but est de lire deux matrices entières dont les tailles et les coefficients sont fournis dans un fichier `entree.txt`, puis de calculer et d'afficher leur produit.

On exécutera ce programme sur l'exemple suivant (contenu du fichier `entree.txt`)

```
3 2
1 0
0 1
1 1

2 4
2 3 4 5
1 2 3 4
```

Pour déboguer, on exécute donc la commande

```
(gdb) run < entree.txt
```

Ici le programme s'arrête de façon anormale (erreur de segmentation). Dans ce cas, `gdb` permet d'indentifier l'endroit exact où le programme s'est arrêté. Il affiche par exemple

```
(gdb) run < entree.txt
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

 Affichage de A:

Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)
```

On en déduit que l'erreur de segmentation s'est produite à l'exécution de la ligne 38 du programme source, lors d'un appel à la fonction `affiche` avec les arguments `M = 0x8049af8`, `nb_lignes = 1073928121`, `nb_col = 134513804`. Par ailleurs, la fenêtre `Emacs` utilisée pour déboguer se coupe en 2, et affiche dans sa moitié inférieure le programme source, en pointant par une flèche la ligne qui a provoqué l'arrêt du programme :

```
      for (j=0; j < nb_col; j++)
=>      printf("%2d\t",M[i][j]);
      printf("\n");
```

Dans un tel cas, on utilise alors la commande `backtrace` (raccourci `bt`), qui affiche l'état de la pile des appels lors de l'arrêt du programme. Une commande strictement équivalente à `backtrace` est la commande `where`.

```
(gdb) backtrace
#0  0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
#1  0x8048881 in main () at exemple.c:78
(gdb)
```

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

/* declaration des fonctions secondaires */
int **lecture_matrice(unsigned int, unsigned int);
void affiche(int **, unsigned int, unsigned int);
int **produit(int **, int **, unsigned int, unsigned int, unsigned
 int, unsigned int);

int **lecture_matrice(unsigned int nb_lignes, unsigned int nb_col)
{
  int **M;
  int i, j;

  scanf("%d", &nb_lignes);
  scanf("%d", &nb_col);
  M = (int**)malloc(nb_lignes * sizeof(int*));
  for (i=0; i<nb_lignes; i++)
    M[i] = (int*)malloc(nb_col * sizeof(int));
  for (i=0; i<nb_lignes; i++)
    {
      for (j=0; j<nb_col; j++)
        scanf("%d",&M[i][j]);
    }
  return(M);
}

void affiche(int **M, unsigned int nb_lignes, unsigned int nb_col)
{
  int i, j;
  if (M == NULL)
    {
      printf("\n Erreur: la matrice a afficher est egale a NULL\n");
      return;
    }
  for (i=0; i < nb_lignes; i++)
    {
      for (j=0; j < nb_col; j++)
        printf("%2d\t",M[i][j]);
      printf("\n");
    }
  return;
}


int **produit(int **A, int **B, unsigned int nb_lignes1, unsigned
int nb_col1, unsigned int nb_lignes2, unsigned int nb_col2)
{
  int **P;
  int i, j, k;

  if (nb_col1 != nb_lignes2)
    {
      printf("\n Impossible d'effectuer le produit : dimensions incompatibles\n");
      return(NULL);
    }
  P = (int**)malloc(nb_lignes1 * sizeof(int*));
  for (i=0; i<nb_lignes1; i++)
    P[i] = (int*)calloc(nb_col2, sizeof(int));
  /* calcul de P[i][j] */
  for (i=0; i < nb_lignes1; i++)
    {
      for (j=0; j < nb_col2; j++)
        {
          for (k=0; k < nb_lignes2; k++)
            P[i][j] += A[i][k] * B[k][j];
        }
    }
  return(P);
}


int main()
{
  int **A, **B, **P;
  unsigned int nb_lignesA, nb_lignesB, nb_colA, nb_colB;

  A = lecture_matrice(nb_lignesA, nb_colA);
  printf("\n Affichage de A:\n");
  affiche(A, nb_lignesA, nb_colA);
  B = lecture_matrice(nb_lignesB, nb_colB);
  printf("\n Affichage de B:\n");
  affiche(B, nb_lignesB, nb_colB);
  P = produit(A, B, nb_lignesA, nb_colA, nb_lignesB, nb_colB);
  printf("\n Affichage du produit de A par B:\n");
  affiche(P, nb_lignesA, nb_colB);
  return(EXIT_SUCCESS);
}
```

On apprend ici que l'erreur a été provoqué par la ligne 38 du programme, à l'intérieur d'un appel à la fonction `affiche` qui, elle, avait été appelée à la ligne 78 par la fonction `main`. L'erreur survient donc à l'affichage de la première matrice lue. `Gdb` fournit déjà une idée de la source du bogue en constatant que les valeurs des arguments de la fonction `affiche` ont l'air anormales.

B.5 Afficher les données

Pour en savoir plus, on peut faire afficher les valeurs de certaines variables. On utilise pour cela la commande `print` (raccourci `p`) qui permet d'afficher la valeur d'une variable, d'une expression... Par exemple ici, on peut faire

```
(gdb) print i
$1 = 0
(gdb) print j
$2 = 318
(gdb) print M[i][j]
Cannot access memory at address 0x804a000.
```

L'erreur provient clairement du fait que l'on tente de lire l'élément d'indice [0][318] de la matrice qui n'est pas défini (puisque le fichier `entree.txt` contenait une matrice à 3 lignes et 2 colonnes).

Par défaut, `print` affiche l'objet dans un format "naturel" (un entier est affiché sous forme décimale, un pointeur sous forme hexadécimale...). On peut toutefois préciser le format d'affichage à l'aide d'un spécificateur de format sous la forme

`(gdb) print /`*`f expression`*

où la lettre *f* précise le format d'affichage. Les principaux formats correspondent aux lettres suivantes : `d` pour la représentation décimale signée, `x` pour l'hexadécimale, `o` pour l'octale, `c` pour un caractère, `f` pour un flottant. Un format d'affichage spécifique au débogueur pour les entiers est `/t` qui affiche la représentation binaire d'un entier.

```
(gdb) print j
$3 = 328
(gdb) p /t j
$4 = 101001000
```

Les identificateurs `$1` ... `$4` qui apparaissent en résultat des appels à `print` donnent un nom aux valeurs retournées et peuvent être utilisés par la suite (cela évite de retaper des constantes et minimise les risques d'erreur). Par exemple

```
(gdb) print nb_col
$5 = 134513804
(gdb) print M[i][$5-1]
Cannot access memory at address 0x804a000.
```

L'identificateur `$` correspond à la dernière valeur ajoutée et `$$` à l'avant-dernière. On peut visualiser les 10 dernières valeurs affichées par `print` avec la commande `show values`.

Une fonctionnalité très utile de `print` est de pouvoir afficher des zones-mémoire contiguës (on parle de tableaux dynamiques). Pour une variable `x` donnée, la commande

`print x@`*`longueur`*

affiche la valeur de `x` ainsi que le contenu des *`longueur`*`-1` zones-mémoires suivantes. Par exemple

```
(gdb) print M[0][0]@10
$4 = {1, 0, 0, 17, 0, 1, 0, 17, 1, 1}
```

affiche la valeur de `M[0][0]` et des 9 entiers suivants en mémoire. De même,

```
(gdb) print M[0]@8
$5 = {0x8049b08, 0x8049b18, 0x8049b28, 0x11, 0x1, 0x0, 0x0, 0x11}
```

affiche la valeur de `M[0]` (de type `int*`) et des 7 objets de type `int*` qui suivent en mémoire.

Quand il y a une ambiguïté sur le nom d'une variable (dans le cas où plusieurs variables locales ont le même nom, ou que le programme est divisé en plusieurs fichiers source qui contiennent des variables portant le même nom), on peut préciser le nom de la fonction ou du fichier source dans lequel la variable est définie au moyen de la syntaxe

`nom_de_fonction::variable`

`'nom_de_fichier'::variable`

Pour notre programme, on peut préciser par exemple

```
(gdb) print affiche::nb_col
$6 = 134513804
```

La commande `whatis` permet, elle, d'afficher le type d'une variable. Elle possède la même syntaxe que `print`. Par exemple,

```
(gdb) whatis M
type = int **
```

Dans le cas de types structures, unions ou énumérations, la commande `ptype` détaille le type en fournissant le nom et le type des différents champs (alors `whatis` n'affiche que le nom du type).

Enfin, on peut également afficher le prototype d'une fonction du programme à l'aide de la commande `info func` :

```
(gdb) info func affiche
All functions matching regular expression "affiche":

File exemple.c:
void affiche(int **, unsigned int, unsigned int);
(gdb)
```

B.6 Appeler des fonctions

À l'aide de la commande `print`, on peut également appeler des fonctions du programme en choisissant les arguments. Ainsi pour notre programme, on peut détecter que le bogue vient du fait que la fonction `affiche` a été appelée avec des arguments étranges. En effet, si on appelle `affiche` avec les arguments corrects, on voit qu'elle affiche bien la matrice souhaitée :

```
(gdb) print affiche(M, 3, 2)
 1       0
 0       1
 1       1
$8 = void
```

On remarque que cette commande affiche la valeur retournée par la fonction (ici `void`).

Une commande équivalente est la commande `call` :

```
(gdb) call fonction(arguments)
```

B.7 Modifier des variables

On peut aussi modifier les valeurs de certaines variables du programme à un moment donné de l'exécution grâce à la commande

```
(gdb) set variable nom_variable = expression
```

Cette commande affecte à *`nom_variable`* la valeur de *`expression`*.

Cette affectation peut également se faire de manière équivalente à l'aide de la commande `print` :

```
(gdb) print nom_variable = expression
```

qui affiche la valeur de *`expression`* et l'affecte à *`variable`*.

B.8 Se déplacer dans la pile des appels

À un moment donné de l'exécution, `gdb` a uniquement accès aux variables définies dans ce contexte, c'est-à-dire aux variables globales et aux variables locales à la fonction en cours d'exécution. Si l'on souhaite accéder à des variables locales à une des fonctions situées plus haut dans la pile d'appels (par exemple des variables locales à `main` ou locales à la fonction appelant la fonction courante), il faut au préalable se déplacer dans la pile des appels.

La commande `where` affiche la pile des appels. Par exemple, dans le cas de notre programme, on obtient

```
(gdb) where
#0  0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
#1  0x8048881 in main () at exemple.c:78
```

On constate ici que l'on se situe dans la fonction `affiche`, qui a été appelée par `main`. Pour l'instant, on ne peut donc accéder qu'aux variables locales à la fonction `affiche`. Si l'on tente d'afficher une variable locale à `main`, `gdb` produit le message suivant :

```
(gdb) print nb_lignesA
No symbol "nb_lignesA" in current context.
```

La commande `up` permet alors de se déplacer dans la pile des appels. Ici, on a

```
(gdb) up
#1  0x8048881 in main () at exemple.c:78
```

Plus généralement, la commande
`(gdb) up [nb_positions]`

permet de se déplacer de n positions dans la pile. La commande
`(gdb) down [nb_positions]`

permet de se déplacer de n positions dans le sens inverse.

La commande `frame` *`numero`* permet de se placer directement au numéro *`numero`* dans la pile des appels. Si le nume'ro n'est pas spécifié, elle affiche l'endroit où l'on se trouve dans la pile des appels. Par exemple, si on utilise la commande `up`, on voit grâce à `frame` que l'on se situe maintenant dans le contexte de la fonction `main` :

```
(gdb) up
#1  0x8048881 in main () at exemple.c:78
(gdb) frame
#1  0x8048881 in main () at exemple.c:78
```

On peut alors afficher les valeurs des variables locales définies dans le contexte de `main`. Par exemple

```
(gdb) print nb_lignesA
$9 = 1073928121
(gdb) print nb_colA
$10 = 134513804
```

B.9 Poser des points d'arrêt

Un point d'arrêt est un endroit où l'on interrompt temporairement l'exécution du programme enfin d'examiner (ou de modifier) les valeurs des variables à cet endroit. La commande permettant de mettre un point d'arrêt est `break` (raccourci en `b`). On peut demander au programme de s'arrêter avant l'exécution d'une fonction (le point d'arrêt est alors défini par le nom de la fonction) ou avant l'exécution d'une ligne donnée du fichier source (le point d'arrêt est alors défini par le numéro de la ligne correspondant). Dans le cas de notre programme, on peut poser par exemple deux points d'arrêt, l'un avant l'exécution de la fonction `affiche` et l'autre avant la ligne 24 du fichier, qui correspond à l'instruction de retour à la fonction appelante de `lecture_matrice` :

```
(gdb) break affiche
Breakpoint 1 at 0x80485ff: file exemple.c, line 30.
(gdb) break 24
Breakpoint 2 at 0x80485e8: file exemple.c, line 24.
```

En présence de plusieurs fichiers source, on peut spécifier le nom du fichier source dont on donne le numéro de ligne de la manière suivante

`(gdb) break nom_fichier:numero_ligne`

```
(gdb) break nom_fichier:nom_fonction
```

Sous `Emacs`, pour mettre un point d'arrêt à la ligne numéro n (ce qui signifie que le programme va s'arrêter juste avant d'exécuter cette ligne), il suffit de se placer à la ligne n du fichier source et de taper `C-x SPC` où `SPC` désigne la barre d'espace.

Quand on exécute le programme en présence de points d'arrêt, le programme s'arrête dès qu'il rencontre le premier point d'arrêt. Dans notre cas, on souhaite comprendre comment les variables `nb_lignesA` et `nb_colA`, qui correspondent au nombre de lignes et au nombre de colonnes de la matrice lue, évoluent au cours de l'exécution. On va donc exécuter le programme depuis le départ à l'aide de la commande `run` et examiner les valeurs de ces variables à chaque point d'arrêt.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

Breakpoint 2, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:24
(gdb)
```

Le premier message affiché par `gdb` demande si l'on veut reprendre l'exécution du programme depuis le début. Si l'on répond oui (en tapant `y`), le programme est relancé (avec par défaut les mêmes arguments que lors du dernier appel de `run`). Il s'arrête au premier point d'arrêt rencontré, qui est le point d'arrêt numéro 2 situé à la ligne 24 du fichier. On peut alors faire afficher les valeurs de certaines variables, les modifier... Par exemple, ici,

```
(gdb) print nb_lignes
$11 = 3
(gdb) print nb_col
$12 = 2
```

La commande `continue` (raccourci en `c`) permet de poursuivre l'exécution du programme jusqu'au point d'arrêt suivant (ou jusqu'à la fin). Ici, on obtient

```
(gdb) continue
Continuing.

 Affichage de A:

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121,
    nb_col=134513804) at exemple.c:30
(gdb)
```

On remarque ici que les variables correspondant aux nombres de lignes et de colonnes avaient la bonne valeur à l'intérieur de la fonction `lecture_matrice`, et qu'elles semblent prendre une valeur aléatoire dès que l'on sort de la fonction. L'erreur vient du fait que les arguments `nb_lignes` et `nb_col` de `lecture_matrice` doivent être passés par adresse et non par valeur, pour que leurs valeurs soient conservées à la sortie de la fonction.

B.10 Gérer les points d'arrêt

Pour connaître la liste des points d'arrêt existant à un instant donné, il faut utiliser la commande `info breakpoints` (qui peut s'abbréger en `info b` ou même en `i b`).

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080485ff in affiche at exemple.c:30
2   breakpoint     keep y   0x080485e8 in lecture_matrice at exemple.c:24
```

On peut enlever un point d'arrêt grâce à la commande `delete` (raccourci `d`) :

`(gdb) delete` *`numero_point_arrêt`*

En l'absence d'argument, `delete` détruit tous les points d'arrêt.

La commande `clear` permet également de détruire des points d'arrêt mais en spécifiant, non plus le numéro du point d'arrêt, mais la ligne du programme ou le nom de la fonction où ils figurent. Par exemple,

`(gdb) clear` *`nom_de_fonction`*

enlève tous les points d'arrêt qui existaient à l'intérieur de la fonction. De la même façon, si on donne un numéro de la ligne en argument de `clear`, on détruit tous les points d'arrêt concernant cette ligne.

Enfin, on peut aussi désactiver temporairement un point d'arrêt. La 4e colonne du tableau affiché par `info breakpoints` contient un `y` si le point d'arrêt est activé et un `n` sinon. La commande

`disable` *`numero_point_arrêt`*

désactive le point d'arrêt correspondant. On peut le réactiver par la suite avec la commande

`enable` *`numero_point_arrêt`*

Cette fonctionnalité permet d'éviter de détruire un point d'arrêt dont on aura peut-être besoin plus tard, lors d'une autre exécution par exemple.

B.11 Les points d'arrêt conditionnels

On peut également mettre un point d'arrêt avant une fonction ou une ligne donnée du programme, mais en demandant que ce point d'arrêt ne soit effectif que sous une certaine condition. La syntaxe est alors

`(gdb) break` *`ligne_ou_fonction`* `if` *`condition`*

Le programme ne s'arrêtera au point d'arrêt que si la condition est vraie. Dans notre cas, le point d'arrêt de la ligne 24 (juste avant de sortir de la fonction `lecture_matrice`) n'est vraiment utile que si les valeurs des variables `nb_lignes` et `nb_col` qui nous intéressent sont anormales. On peut donc utilement remplacer le point d'arrêt numéro 2 par un point d'arrêt conditionnel :

```
(gdb) break 24 if nb_lignes != 3 || nb_col != 2
Breakpoint 8 at 0x80485e8: file exemple.c, line 24.
(gdb) i b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080485ff in affiche at exemple.c:30
```

```
        breakpoint already hit 1 time
3   breakpoint     keep y   0x080485e8 in lecture_matrice at exemple.c:24
        stop only if nb_lignes != 3 || nb_col != 2
(gdb)
```

Si on relance l'exécution du programme avec ces deux points d'arrêt, on voit que le programme s'arrête au point d'arrêt numéro 1, ce qui implique que les variables `nb_lignes` et `nb_col` ont bien la bonne valeur à la fin de la fonction `lecture_matrice` :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

 Affichage de A:

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:30
(gdb)
```

On peut aussi transformer un point d'arrêt existant en point d'arrêt conditionnel avec la commande `cond`

`(gdb) cond` ***`numero_point_arret condition`***

Le point d'arrêt numéro ***`numero_point_arret`*** est devenu un point d'arrêt conditionnel, qui ne sera effectif que si ***`condition`*** est satisfaite.

De même pour transformer un point d'arrêt conditionnel en point d'arrêt non conditionnel (c'est-à-dire pour enlever la condition), il suffit d'utiliser la commande `cond` sans préciser de condition.

B.12 Exécuter un programme pas à pas

`Gdb` permet, à partir d'un point d'arrêt, d'exécuter le programme instruction par instruction. La commande `next` (raccourci `n`) exécute uniquement l'instruction suivante du programme. Lors que cette instruction comporte un appel de fonction, la fonction est entièrement exécutée. Par exemple, en partant d'un point d'arrêt situé à la ligne 77 du programme (il s'agit de la ligne

```
  printf("\n Affichage de A:\n");
```

dans la fonction `main`), 2 `next` successifs produisent l'effet suivant

```
(gdb) where
#0  main () at exemple.c:77
(gdb) next

 Affichage de A:
(gdb) next
```

```
Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)
```

La premier `next` exécute la ligne 77 ; le second exécute la ligne 78 qui est l'appel à la fonction `affiche`. Ce second `next` conduit à une erreur de segmentation.

La commande `step` (raccourci `s`) a la même action que `next`, mais elle rentre dans les fonctions : si une instruction contient un appel de fonction, la commande `step` effectue la première instruction du corps de cette fonction. Si dans l'exemple précédent, on exécute deux fois la commande `step` à partir de la ligne 78, on obtient

```
(gdb) where
#0  main () at exemple.c:78
(gdb) step
affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804) at exemple.c:30
(gdb) step
(gdb) where
#0  affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:35
#1  0x8048881 in main () at exemple.c:78
(gdb)
```

On se trouve alors à la deuxième instruction de la fonction `affiche`, à la ligne 35.

Enfin, lorsque le programme est arrêté à l'intérieur d'une fonction, la commande `finish` termine l'exécution de la fonction. Le programme s'arrête alors juste après le retour à la fonction appelante. Par exemple, si l'on a mis un point d'arrêt à la ligne 14 (première instruction `scanf` de la fonction `lecture_matrice`), la commande `finish` à cet endroit fait sortir de `lecture_matrice` :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

Breakpoint 2, lecture_matrice (nb_lignes=3, nb_col=134513804) at exemple.c:14
(gdb) where
#0  lecture_matrice (nb_lignes=3, nb_col=134513804) at exemple.c:14
#1  0x804885b in main () at exemple.c:76
(gdb) finish
Run till exit from #0  lecture_matrice (nb_lignes=3, nb_col=134513804)
    at exemple.c:14
0x804885b in main () at exemple.c:76
Value returned is $1 = (int **) 0x8049af8
(gdb)
```

B.13 Afficher la valeur d'une expression à chaque point d'arrêt

On a souvent besoin de suivre l'évolution d'une variable ou d'une expression au cours du programme. Plutôt que de répéter la commande `print` à chaque point d'arrêt ou après chaque `next` ou `step`, on peut utiliser la commande `display` (même syntaxe que `print`) qui permet d'afficher la valeur d'une expression à chaque fois que le programme s'arrête. Par exemple, si l'on veut faire afficher par `gdb` la valeur de `M[i][j]` à chaque exécution de la ligne 38 (ligne

```
printf("%2d\t",M[i][j]);
```

dans les deux boucles `for` de `affiche`), on y met un point d'arrêt et on fait

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

 Affichage de A:

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb) display i
1: i = 0
(gdb) display j
2: j = 0
(gdb) display M[i][j]
3: M[i][j] = 1
(gdb) continue
Continuing.

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
3: M[i][j] = 0
2: j = 1
1: i = 0
(gdb) c
Continuing.

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
3: M[i][j] = 0
2: j = 2
1: i = 0
(gdb) next
3: M[i][j] = 0
2: j = 2
1: i = 0
(gdb)
```

On remarque que la commande `display` affiche les valeurs des variables à chaque endroit où le programme s'arrête (que cet arrêt soit provoqué par un point d'arrêt ou par une exécution pas-à-pas avec `next` ou `step`). A chaque expression faisant l'objet d'un `display` est associée un numéro. La commande `info display` (raccourci `i display`) affiche la liste des expressions faisant l'objet d'un `display` et les numéros correspondants.

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
3:   y  M[i][j]
2:   y  j
1:   y  i
(gdb)
```

Pour annuler une commande `display`, on utilise la commande `undisplay` suivie du numéro correspondant (en l'absence de numéro, tous les `display` sont supprimés)

```
(gdb) undisplay 1
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
3:   y  M[i][j]
2:   y  j
(gdb)
```

Comme pour les points d'arrêt, les commandes

```
(gdb) disable disp numero_display
(gdb) enable disp numero_display
```

respectivement désactive et active l'affichage du `display` correspondant.

B.14 Exécuter automatiquement des commandes aux points d'arrêt

On peut parfois souhaiter exécuter la même liste de commandes à chaque fois que l'on rencontre un point d'arrêt donné. Pour cela, il suffit de définir une seule fois cette liste de commandes à l'aide de `commands` avec la syntaxe suivante :

```
(gdb) commands numero_point_arret
commande_1
...
commande_n
end
```

où *numero_point_arret* désigne le numéro du point d'arrêt concerné. Cette fonctionnalité est notamment utile car elle permet de placer la commande `continue` à la fin de la liste. On peut donc automatiquement passer de ce point d'arrêt au suivant sans avoir à entrer `continue`. Supposons par exemple que le programme ait un point d'arrêt à la ligne 22 (ligne

```
        scanf("%d",&M[i][j]);
```

de la fonction `lecture_matrice`. A chaque fois que l'on rencontre ce point d'arrêt, on désire afficher les valeurs de `i`, `j`, `M[i][j]` et reprendre l'exécution. On entre alors la liste de commandes suivantes associée au point d'arrêt 1 :

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>echo valeur de i \n
>print i
>echo valeur de j \n
>print j
>echo valeur du coefficient M[i][j] \n
>print M[i][j]
>continue
>end
(gdb)
```

Quand on lance le programme, ces commandes sont effectuées à chaque passage au point d'arrêt (et notamment la commande `continue` qui permet de passer automatiquement au point d'arrêt suivant). On obtient donc

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$38 = 0
valeur de j
$39 = 0
valeur du coefficient M[i][j]
$40 = 0

Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$41 = 0
valeur de j
$42 = 1
valeur du coefficient M[i][j]
$43 = 0

Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$44 = 1
valeur de j
$45 = 0
valeur du coefficient M[i][j]
```

```
$46 = 0
```

...

```
Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$53 = 2
valeur de j
$54 = 1
valeur du coefficient M[i][j]
$55 = 0

 Affichage de A:

Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)
```

Il est souvent utile d'ajouter la commande `silent` à la liste de commandes. Elle supprime l'affichage du message `Breakpoint ...` fourni par `gdb` quand il atteint un point d'arrêt. Par exemple, la liste de commande suivante

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>echo valeur de i \n
>print i
>echo valeur de j \n
>print j
>echo valeur du coefficient M[i][j] \n
>print M[i][j]
>continue
>end
```

produit l'effet suivant à l'exécution :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt
valeur de i
$56 = 0
valeur de j
$57 = 0
valeur du coefficient M[i][j]
$58 = 0
valeur de i
```

```
$59 = 0
valeur de j
$60 = 1
valeur du coefficient M[i][j]
$61 = 0
...
valeur de i
$71 = 2
valeur de j
$72 = 1
valeur du coefficient M[i][j]
$73 = 0

 Affichage de A:

Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)
```

Notons enfin que la liste de commandes associée à un point d'arrêt apparaît lorsque l'on affiche la liste des points d'arrêt avec `info breakpoints`.

B.15 Les raccourcis des noms de commande

Tous les noms de commande peuvent être remplacés par leur plus court préfixe non-ambiguë. Par exemple, la commande `clear` peut s'écrire `cl` car aucune autre commande de `gdb` ne commence par `cl`. La lettre `c` seule ne peut pas être utilisée pour `clear` car plusieurs commandes de `gdb` commencent par `c` (`continue`, `call`...).

`Emacs` permet la complétion automatique des noms de commande `gdb` : quand on tape le début d'une commande et que l'on appuie sur `TAB`, le nom de la commande est complété s'il n'y a pas d'ambiguïté. Sinon, `Emacs` fournit toutes les possibilités de complétion. Il suffit alors de cliquer (bouton du milieu) pour choisir la commande.

Les commandes les plus fréquentes de `gdb` sont abbrégées par leur première lettre, même s'il existe d'autres commandes commençant par cette lettre. Par exemple, la commande `continue` peut être abbrégée en `c`. C'est le cas pour les commandes `break`, `continue`, `delete`, `frame`, `help`, `info`, `next`, `print`, `quit`, `run` et `step`.

B.16 Utiliser l'historique des commandes

Il est possible d'activer sous `gdb` l'historique des commandes, afin de ne pas avoir à retaper sans cesse la même commande. Pour activer cette fonctionnalité, il suffit d'entrer la commande

```
(gdb) set history expansion
```

Dans ce cas, comme sous Unix, `!!` rappelle la dernière commande exécutée et `!`***`caractère`*** rappelle la dernière commande commençant par ce caractère. Par exemple,

```
(gdb) !p
print nb_lignes
$75 = 1073928121
(gdb)
```

Cette fonctionnalité n'est pas activée par défaut car il peut y avoir ambiguïté entre le signe ! permettant de rappeler une commande et le ! correspondant à la négation logique du langage C.

B.17 Interface avec le shell

On peut à tout moment sous `gdb` exécuter des commandes shell. Les commandes `cd`, `pwd` et `make` sont disponibles. Plus généralement, la commande `gdb` suivante

`(gdb) shell` ***`commande`***

exécute ***`commande`***.

B.18 Résumé des principales commandes

commande		action	page
`backtrace`	`bt`	indique où l'on se situe dans la pile des appels (synonyme de `where`)	109
`break` (`M-x SPC`)	`b`	pose un point d'arrêt à une ligne définie par son numéro ou au début d'une fonction.	114
`clear`	`cl`	détruit tous les points d'arrêt sur une ligne ou dans une fonction	116
`commands`		définit une liste de commandes à effectuer automatiquement à un point d'arrêt	120
`cond`		ajoute une condition à un point d'arrêt	117
`continue`	`c`	continue l'exécution (après un point d'arrêt)	115
`delete`	`d`	détruit le point d'arrêt dont le numéro est donné	116
`disable`		désactive un point d'arrêt	116
`disable disp`		désactive un `display`	120
`display`		affiche la valeur d'une expression à chaque arrêt du programme	119
`down`		descend dans la pile des appels	114
`enable`		réactive un point d'arrêt	116
`enable disp`		réactive un `display`	120
`file`		redéfinit l'exécutable	108
`finish`		termine l'exécution d'une fonction	118
`frame`		permet de se placer à un endroit donné dans la pile des appels et affiche le contexte	114
`help`	`h`	fournit de l'aide à propos d'une commande	
`info breakpoints`	`i b`	affiche les points d'arrêt	116
`info display`		donne la liste des expressions affichées par des `display`	120
`info func`		affiche le prototype d'une fonction	112
`next`	`n`	exécute l'instruction suivante (sans entrer dans les fonctions)	117
`run`	`r`	lance l'exécution du programme (par défaut avec les arguments utilisés précédemment)	108
`print`	`p`	affiche la valeur d'une expression	111
`ptype`		détaille un type structure	112
`quit`	`q`	quitte `gdb`	108
`set history expansion`		active l'historique des commandes	123
`set variable`		modifie la valeur d'une variable	113
`shell`		permet d'exécuter des commandes shell	124
`show args`		affiche les arguments du programme	108
`show values`		réaffiche les 10 dernières valeurs affichées	111
`step`	`s`	exécute l'instruction suivante (en entrant dans les fonctions)	118
`undisplay`		supprime un `display`	120
`up`		monte dans la pile des appels	114
`whatis`		donne le type d'une expression	112
`where`		indique où l'on se situe dans la pile des appels (synonyme de `backtrace`)	109

Bibliographie

[1] André (J.) et Goossens (M.). – Codage des caractères et multi-linguisme : de l'ASCII à UNICODE et ISO/IEC-10646. *Cahiers GUTenberg* n° 20, mai 1985. `http://www.gutenberg.eu.org/pub/GUTenberg/publications/cahiers.html`.

[2] Braquelaire (J.-P.). – *Méthodologie de la programmation en C.* – Dunod, 2000, troisième édition.

[3] Cassagne (B.). – *Introduction au langage C.* – `http://clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html`.

[4] Delannoy (C.). – *Programmer en langage C.* – Eyrolles, 1992.

[5] Faber (F.). – *Introduction à la programmation en ANSI-C.* – `http://www.ltam.lu/Tutoriel_Ansi_C/`.

[6] Kernighan (B.W.) et Richie (D.M.). – *The C programming language.* – Prentice Hall, 1988, seconde édition.

[7] Léon (L.) et Millet (F.). – *C si facile.* – Eyrolles, 1987.

[8] Loukides (M.) et Oram (A.). – *Programming with GNU software.* – O'Reilly, 1997.

[9] Moussel (P.). – *Le langage C.* – `http://www.mrc.lu/LangC/`.

[10] Sendrier (N.). – *Notes d' introduction au C.* – Cours de DEA, Université de Limoges, 1998.

Index