# Mini-compilateur C [1]

première partie : analyse syntaxique
deuxième partie : typage

Le langage considéré dans ce TER est un sous-ensemble de C. Vos fichiers pourront être compilés aussi bien par votre compilateur que par tout autre compilateur C.

**Notation** Dans la suite, les symboles « $\star$ », « $+$ » et « ? » indiquent la répétition du symbole les précédant (zéro fois ou plus pour « $\star$ », zéro ou une fois pour « ? » et une fois ou plus pour « $+$ »). Attention de ne pas confondre « $\star$ » et « $+$ » avec « `*` » et « `+` » qui sont des symboles du langage C.

# 1 Analyse syntaxique

## 1.1 Analyse lexicale

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires peuvent prendre deux formes :

— débutant par `/*` et s'étendant jusqu'à `*/`, et ne pouvant être imbriqués ;
— débutant par `//` et s'étendant jusqu'à la fin de la ligne (ou du fichier s'il s'agit de la dernière ligne).

De plus on ignorera les lignes suivantes :

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

quelle que soit leur position dans le fichier.

Les identificateurs obéissent à l'expression régulière ⟨*ident*⟩ suivante :

⟨*chiffre*⟩ ::= `0`–`9`
⟨*alpha*⟩ ::= `a`–`z` | `A`–`Z`
⟨*ident*⟩ ::= (⟨*alpha*⟩ | `_`) (⟨*alpha*⟩ | `_` | ⟨*chiffre*⟩)$\star$

Les identificateurs suivants sont des mots clés :

```
char   double else   extern   for      if   int   long
return short  sizeof struct   unsigned void while
```

Les constantes entières obéissent à l'expression régulière ⟨*entier*⟩ suivante :

⟨*entier*⟩ ::= ⟨*chiffre*⟩+(`u`|`U`) ?(`l`|`L`) ?

1. Ce projet reprend certains aspects du projet de compilation donné il y a trois ans, ce dernier lui-même inspiré d'un projet similaire donné par Jean-Christophe Filliâtre dans le cadre de son cours de compilation à l'École Normale Supérieure `http://www.lri.fr/~filliatr`.

Les constantes flottantes obéissent à l'expression régulière ⟨*double*⟩ suivante :

$$\langle double \rangle \quad ::= \quad (\langle chiffre \rangle + \texttt{.} \langle chiffre \rangle \star \mid \quad \texttt{.} \langle chiffre \rangle +)((\texttt{e}|\texttt{E})\texttt{-}\,?\langle chiffre \rangle +)\,?$$

les littéraux caractères sont délimités par des *quotes* simples (`'`). Les caractères dont le code ASCII est celui du blanc (code ASCII décimal 32) ou supérieur sont représentés tels quels, à l'exception des caractères « `'` », « `"` » et « `\` ». Ces derniers peuvent être représentés en utilisant les séquences d'échappement « `\'` », « `\"` » et « `\\` » respectivement. Il est aussi possible d'utiliser la séquence d'échappement « `\x`$xx$ » où $x$ est un chiffre hexadécimal (« `0-9` », « `a-f` » ou « `A-F` »). Dans ce cas, la séquence représente le caractère de code ASCII hexadécimal $xx$. Enfin, la séquence « `\n` » représente un retour chariot.

Enfin les chaînes de caractères littérales sont des séquences de caractères (tels que définis ci-dessus) délimitées par des *quotes* doubles : « `"` ». Les chaînes ne peuvent pas s'étendre sur plusieurs lignes. Dans la suite, les caractères et les chaînes de caractères sont représentés par les non-terminaux ⟨*car*⟩ et ⟨*chaîne*⟩ respectivement.

## 1.2 Analyse syntaxique

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

Les associativités et précédences des divers opérateurs sont données par la table suivante, de la plus faible à la plus forte précédence :

| opérateur | associativité | précédence |
|---|---|---|
| `=` | à droite | faible |
| `\|\|` | à gauche | |
| `&&` | à gauche | |
| `== !=` | à gauche | |
| `<  <=  >  >=` | à gauche | ↓ |
| `+  -` | à gauche | |
| `*  /  %` | à gauche | |
| `!  ++  --  &  *` (unaire)  `+` (unaire)  `-` (unaire) | à droite | |
| `)  [  ->  .` | à gauche | forte |

## 1.3 Localisation des erreurs

Lorsqu'une erreur lexicale ou syntaxique est détectée par votre compilateur, elle doit être signalée le plus précisémment possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.c", line 4, characters 5-6:
syntax error
```

La première ligne du message d'erreur (« `File "test.c"` ...») doit respecter *strictement* ce format. La seconde ligne est libre, vous pouvez vous en servir pour donner un message d'explication (éventuellement en français). Les localisations peuvent être obtenues grâce aux fonctions `Lexing.lexeme_start_p`, `Lexing.lexeme_end_p` pour l'analyseur lexical et `Parsing.rhs_start_pos`, `Parsing.rhs_end_pos` pour l'analyseur syntaxique.

⟨*fichier*⟩ ::= ⟨*decl*⟩⋆ `EOF`
⟨*decl*⟩ ::= ⟨*decl_vars*⟩ | ⟨*decl_typ*⟩ | ⟨*decl_fct*⟩ | ⟨*decl_ext*⟩
⟨*decl_vars*⟩ ::= ⟨*type*⟩ ⟨*var*⟩ `;`
⟨*decl_typ*⟩ ::= `struct` ⟨*ident*⟩ `{` ⟨*decl_vars*⟩⋆ `};`
⟨*decl_fct*⟩ ::= ⟨*type*⟩ `*`⋆ ⟨*ident*⟩ `(` ⟨*arguments*⟩? `)` ⟨*bloc*⟩
⟨*decl_ext*⟩ ::= `extern` ⟨*type*⟩ `*`⋆ ⟨*ident*⟩ `(` ⟨*arguments*⟩? `);`
⟨*type*⟩ ::= `void` | ⟨*int_type*⟩ | `double` | `struct` ⟨*ident*⟩
⟨*int_type*⟩ ::= `unsigned`? (`char` | `short` | `int` | `long`)
⟨*arguments*⟩ ::= ⟨*type*⟩ ⟨*var*⟩ | ⟨*type*⟩ ⟨*var*⟩ `,` ⟨*arguments*⟩
⟨*var*⟩ ::= ⟨*ident*⟩ | `*` ⟨*var*⟩
⟨*expr*⟩ ::= ⟨*entier*⟩ | ⟨*car*⟩ | ⟨*chaîne*⟩ | ⟨*double*⟩
| ⟨*ident*⟩
| `*` ⟨*expr*⟩
| ⟨*expr*⟩ `[` ⟨*expr*⟩ `]`
| ⟨*expr*⟩ `.` ⟨*ident*⟩
| ⟨*expr*⟩ `->` ⟨*ident*⟩
| ⟨*expr*⟩ `=` ⟨*expr*⟩
| ⟨*ident*⟩ `(` ⟨*l_expr*⟩? `)`
| `++` ⟨*expr*⟩ | `--` ⟨*expr*⟩ | ⟨*expr*⟩ `++` | ⟨*expr*⟩ `--`
| `&` ⟨*expr*⟩ | `!` ⟨*expr*⟩ | `-` ⟨*expr*⟩ | `+` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*opérateur*⟩ ⟨*expr*⟩
| `sizeof` `(` ⟨*cplx_type*⟩ `)`
| `(` ⟨*cplx_type*⟩ `)` ⟨*expr*⟩
| `(` ⟨*expr*⟩ `)`
⟨*opérateur*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=` | `+` | `-` | `*` | `/` | `%` | `&&` | `||`
⟨*cplx_type*⟩ ::= ⟨*type*⟩ `*`⋆
⟨*l_expr*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ `,` ⟨*l_expr*⟩

⟨*instruction*⟩ ::= `;`
| ⟨*expr*⟩ `;`
| `if` `(` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `if` `(` ⟨*expr*⟩ `)` ⟨*instruction*⟩ `else` ⟨*instruction*⟩
| `while` `(` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `for` `(` ⟨*l_expr*⟩? `;` ⟨*expr*⟩? `;` ⟨*l_expr*⟩? `)` ⟨*instruction*⟩
| ⟨*bloc*⟩
| `return` ⟨*expr*⟩? `;`
⟨*bloc*⟩ ::= `{` ⟨*decl_vars*⟩⋆ ⟨*instruction*⟩⋆ `}`

FIGURE 1 – Grammaire des fichiers C

# 2 Typage

## 2.1 Types et environnements de typage

Dans tout ce qui suit, les expressions de types sont de la forme suivante :

$$\tau ::= \texttt{void} \mid \iota \mid \texttt{double} \mid \texttt{struct}\ id \mid \tau\texttt{*} \mid \texttt{typenull}$$

$$\iota ::= \texttt{unsgigned}\ \nu \mid \nu$$
$$\nu ::= \texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long}$$

où *id* désigne un identificateur de structure. Le type `typenull` est un type particulier introduit pour pouvoir typer la constante `0` qui en C représente soit l'entier 0 soit le pointeur nul (la macro `NULL`, définie dans le fichier `stdlib.h` n'est en général qu'un alias pour `((void *)0)` ). Il s'agit là d'une notation pour la syntaxe *abstraite* des expressions de types. On introduit la relation $\approx$ sur les types comme la plus petite relation réflexive et symétrique telle que

$$\frac{\tau_1, \tau_2 \in \{\iota, \texttt{typenull}, \texttt{double}\}}{\tau_1 \approx \tau_2} \qquad \frac{}{\texttt{typenull} \approx \tau\texttt{*}} \qquad \frac{}{\texttt{void*} \approx \tau\texttt{*}}$$

Attention, la relation *n'est pas transitive*. En effet, on a : `int` $\approx$ `typenull` (par la première règle) et `typenull` $\approx$ `char*` (par la deuxième règle) mais on n'a surtout pas `int` $\approx$ `char*`. Un environnement de typage $\Gamma$ est une suite de déclarations de variables de la forme $\tau\ x$, de déclarations de structures de la forme `struct` $S$ `{`$\tau_1\ x_1 \cdots \tau_n\ x_n$`}`, et de déclarations de profils de fonctions de la forme $\tau\ f(\tau_1, \ldots, \tau_n)$. On notera `struct` $S$ `{`$\tau\ x$`}` pour indiquer que la structure $S$ contient un champ $x$ de type $\tau$.

Dans la suite, on dira qu'un type est *numérique*, et on notera $num(\tau)$ si $\tau$ n'est compatible ni avec `void` ni avec un type de structure (autrement dit, $\tau$ doit être compatible avec `typenull`, un type d'entier $\iota$ `double` ou un type pointeur $\tau'$`*`) et on dira qu'un type est *arithmétique*, et on notera $arith(\tau)$ si $\tau$ est numérique et n'est pas un type pointeur.

On définit ensuite une fonction `rank` sur les types arithmétiques. Cette dernière sert à exprimer de manière compacte les conversions entre les différents types d'entiers (et elle représente, intuitivement le nombre de bits utilisés par le type, hors bit de signe).

$$\begin{array}{lcl} rank(\texttt{char}) & = & 7 \\ rank(\texttt{short}) & = & 15 \\ rank(\texttt{int}) & = & 31 \\ rank(\texttt{long}) & = & 63 \end{array} \qquad \begin{array}{lcl} rank(\texttt{unsigned}\ \nu) & = & 1 + \texttt{rank}(\nu) \\ rank(\texttt{double}) & = & 100 \\ rank(\texttt{typenull}) & = & 0 \end{array}$$

On définit à partir de *rank*, la relation d'ordre partielle sur les types :

$$\tau_1 < \tau_2 \Leftrightarrow rank(\tau_1) < rank(\tau_2)$$

et on note $\tau_1 \leq \tau_2$ le fait que $\tau_1 < \tau_2$ ou $\tau_1 = \tau_2$ et on définit :

$$\begin{array}{lcll} max(\tau_1, \tau_2) & = & \tau_2 & \text{si } \tau_1 < \tau_2 \\ max(\tau_1, \tau_2) & = & \tau_1 & \text{si } \tau_2 < \tau_1 \end{array}$$

## 2.2 Typage

**Bonne formation des types** On dit qu'un type $\tau$ est *bien formé* dans un environnement $\Gamma$, et on note $\Gamma \vdash \tau$ `bf`, si tous les identificateurs de structures apparaissant dans $\tau$ correspondent à des structures déclarées dans $\Gamma$.

**Typage des expressions** En C, il existe une classe d'expressions particulières nommées « *valeurs gauches* ». Intuitivement, les valeurs gauches sont les expressions que l'on a le droit de placer *à gauche* d'une affectation (d'où leur nom). À l'inverse d'autres langages, le fait d'être une valeur gauche en C dépend du type de l'expression. C'est pourquoi

on introduit simultanément le jugement $\Gamma \vdash e : \tau$ signifiant « dans l'environnement $\Gamma$, l'expression $e$ est bien typée de type $\tau$ » et le jugement $\Gamma \vdash_l e : \tau$ signifiant « dans l'environnement $\Gamma$, l'expression $e$ est une valeur gauche bien typée de type $\tau$ ». Ces jugements sont définis par les règles d'inférence suivantes :

**(constantes)**

$$\frac{}{\Gamma \vdash \texttt{0} : \texttt{typenull}} \qquad \frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma \vdash \tau \ \texttt{bf} \quad \tau \not\equiv \texttt{void}}{\Gamma \vdash \texttt{sizeof(}\tau\texttt{)} : \texttt{unsigned long}}$$

**(affectations)**

$$\frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \approx \tau_2}{\Gamma \vdash e_1 \texttt{ = } e_2 : \tau_1}$$

$$\frac{\Gamma \vdash_l e : \tau \quad num(\tau) \quad op \in \{\texttt{++}, \texttt{--}\}}{\Gamma \vdash op\ e : \tau} \qquad \frac{\Gamma \vdash_l e : \tau \quad num(\tau) \quad op \in \{\texttt{++}, \texttt{--}\}}{\Gamma \vdash e\ op : \tau}$$

**(comparaisons)**

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad num(\tau_1)\ \tau_1 \approx \tau_2 \quad op \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

**(arithmétique et logique)**

$$\frac{\Gamma \vdash e : \tau \quad arith(\tau) \quad op \in \{\texttt{+}, \texttt{-}\}}{\Gamma \vdash op\ e : \tau} \qquad \frac{\Gamma \vdash e : \tau \quad num(\tau)}{\Gamma \vdash \texttt{!}e : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad max(\tau_1, \tau_2) \leq \texttt{unsigned long}}{\Gamma \vdash e_1 \texttt{ \% } e_2 : max(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \approx \tau_2 \quad \tau_1 \approx \texttt{double} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1\ op\ e_2 : max(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \approx \tau_2 \quad \tau_1 \approx \texttt{double} \quad op \in \{\texttt{||}, \texttt{\&\&}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

**(arithmétique de pointeurs)**

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_1' \texttt{*} \quad max(\tau_2, \texttt{int}) \leq \texttt{unsigned long} \quad op \in \{\texttt{+}, \texttt{ -}\}}{\Gamma \vdash e_1\ op\ e_2 : \tau_1' \texttt{*}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2' \texttt{*} \quad \tau_2 \equiv \tau_2' \texttt{*}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{long}} \qquad \frac{\Gamma \vdash e_2 \texttt{ + } e_1 : \tau}{\Gamma \vdash e_1 \texttt{ + } e_2 : \tau}$$

**(appel de fonction)**

$$\frac{\Gamma \vdash e_i : \tau_i \quad \tau\ f(\tau_1', \ldots, \tau_n') \in \Gamma \quad \tau_i \approx \tau_i'}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

**(accès)**

$$\frac{\Gamma \vdash e : \texttt{struct}\ S \quad \texttt{struct}\ S\ \{\tau\ x\} \in \Gamma}{\Gamma \vdash e.x : \tau}$$

**(adresse)**

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \texttt{\&}e : \tau\texttt{*}}$$

**(valeurs gauches)**

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash e : \tau\texttt{*}}{\Gamma \vdash_l \texttt{*}e : \tau}$$

$$\frac{\Gamma \vdash_l e : \texttt{struct}\ S \quad \texttt{struct}\ S\ \{\tau\ x\} \in \Gamma}{\Gamma \vdash_l e.x : \tau}$$

**Typage des instructions** On introduit le jugement $\Gamma,\ \tau_0 \vdash_i i$ signifiant « dans l'environnement $\Gamma$, l'instruction $i$ est bien typée, pour un type de retour $\tau_0$ ». Intuitivement, $\tau_0$ représente le type de retour de la fonction dans la quelle se trouve l'instruction $i$. Ce jugement est établi par les règles d'inférence suivantes :

$$\frac{}{\Gamma, \tau_0 \vdash_i \texttt{;}} \qquad \frac{\Gamma, \tau_0 \vdash e : \tau}{\Gamma, \tau_0 \vdash_i e\texttt{;}} \qquad \frac{}{\Gamma, \texttt{void} \vdash_i \texttt{return;}} \qquad \frac{\Gamma \vdash e : \tau_0}{\Gamma, \tau_0 \vdash_i \texttt{return}\ e\texttt{;}}$$

$$\frac{\Gamma \vdash e : \tau \quad \mathit{num}(\tau) \quad \Gamma, \tau_0 \vdash_i i_1 \quad \Gamma, \tau_0 \vdash_i i_2}{\Gamma, \tau_0 \vdash_i \texttt{if (}e\texttt{)}\ i_1\ \texttt{else}\ i_2}$$

$$\frac{\Gamma, \tau_0 \vdash_i i_1 \quad \Gamma \vdash e : \tau \quad \mathit{num}(\tau) \quad \Gamma, \tau_0 \vdash_i i_2 \quad \Gamma, \tau_0 \vdash_i i_3}{\Gamma, \tau_0 \vdash_i \texttt{for(}i_1\texttt{;}\ e\texttt{;}\ i_2\texttt{)}\ i_3}$$

$$\frac{\forall j \leq k, \Gamma \vdash \tau_j\ \mathsf{bf} \quad \tau_j \not\equiv \texttt{void} \quad \forall j \leq n, \{\tau_1\ x_1, \ldots, \tau_k\ x_k\} \cup \Gamma \vdash_i i_j}{\Gamma, \tau_0 \vdash_i \texttt{\{}\tau_1\ x_1 \cdots \tau_k\ x_k\texttt{;}i_1 \cdots i_n\texttt{\}}}$$

Cette dernière règle signifie que pour typer un *bloc* constitué de $k$ déclarations de variables (locales au bloc) et de $n$ instructions, on vérifie d'abord la bonne formation des déclarations puis on type chacune des instructions dans l'environnement augmenté des nouvelles déclarations. On rappelle aussi que le *parsing* a supprimé des instructions superflues (`while`, `if` sans `else`, ...).

**Typage des fichiers** On rappelle qu'un fichier est une liste de déclarations. On introduit le jugement « $\Gamma \vdash d \to \Gamma'$ » qui signifie « dans l'environnement $\Gamma$, la déclaration $d$ est bien formée et produit un environnement $\Gamma'$ ». Ce jugement est dérivable grâce aux règles suivantes :

**Déclarations de variables (globales)**

$$\frac{\Gamma \vdash \tau\ \mathsf{bf} \quad \tau \not\equiv \texttt{void}}{\Gamma \vdash \tau\ x \to \{\tau\ x\} \cup \Gamma}$$

**Déclarations de structures**

$$\frac{\Gamma, \texttt{struct}\ id\ \{\tau_1\ x_1 \cdots \tau_n\ x_n\} \vdash \tau_i\ \mathsf{bf}}{\Gamma \vdash \texttt{struct}\ id\ \{\tau_1\ x_1; \cdots \tau_n\ x_n;\} \to \{\texttt{struct}\ id\ \{\tau_1\ x_1 \cdots \tau_n\ x_n\}\} \cup \Gamma}$$

On vérifiera d'autre part que les types de champs $\tau_i$ ne font référence à la structure $id$ elle-même que sous un pointeur.

**Déclarations de fonctions et de fonction externes**

$$\frac{\Gamma \vdash \tau_i\ \mathsf{bf} \quad \forall i > 0, \tau_i \not\equiv \texttt{void} \quad \{\tau_0\ f(\tau_1, \ldots, \tau_n), \tau_1\ x_1, \ldots, \tau_n\ x_n\} \cup \Gamma, \tau_0 \vdash_i b}{\Gamma \vdash \tau_0\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ b \to \{\tau_0\ f(\tau_1, \ldots, \tau_n)\} \cup \Gamma}$$

$$\frac{\Gamma \vdash \tau_i\ \mathsf{bf} \quad \forall i > 0, \tau_i \not\equiv \texttt{void}}{\Gamma \vdash \tau_0\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ b \to \{\tau_0\ f(\tau_1, \ldots, \tau_n)\} \cup \Gamma}$$

On remarque que le prototype d'une fonction est ajouté à l'environnement pour le typage de cette dernière, cela dans le but de typer les fonctions récursives.

**Fichiers** On introduit finalement le jugement $\Gamma \vdash_f d_1 \cdots d_n$ signifiant « dans l'environnement $\Gamma$ le fichier constitué par la suite de déclarations $d_1$, …, $d_n$ est bien formé ». Le typage d'un fichier consiste à typer successivement les déclarations dans le contexte étendu par chaque nouvelle déclaration, d'où les règles :

$$\frac{}{\Gamma \vdash_f \emptyset} \qquad \frac{\Gamma \vdash d_1 \to \Gamma' \quad \Gamma' \vdash_f d_2 \cdots d_n}{\Gamma \vdash_f d_1\ d_2 \cdots d_n}$$

**Règles d'unicité** Enfin, on vérifiera l'unicité :

— des identificateurs de structures sur l'ensemble du fichier ;
— des champs de structure à l'intérieur d'une *même* structure ;
— des symboles (variables *globales* et fonctions) sur l'ensemble du fichier.

**Fonction principale** On vérifiera la présence d'une fonction **main** avec l'un des deux prototypes suivants :

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

dans le fichier.