Syntaxe et outils de base du langage C

Gérard Dethier

Février 2011

Chapitre 1

Introduction

Le but de ce document est de permettre la consultation rapide de la syntaxe C associée à des notions de programmation courantes telles que les types, opérateurs, déclarations de fonctions, pointeurs, etc.

Il ne constitue pas nécessairement une base d'apprentissage pour la programmation et pour le langage C en particulier. Le lecteur souhaitant consulter un document plus complet à ce sujet devrait plutôt se tourner vers les transparents [1] du cours INFO2009–1 donné par le Professeur B. Boigelot.

La version C99 du langage C est considérée dans ce document. Celle-ci est décrite dans la norme ISO 9899 de 1999.

La syntaxe du langage C est décrite en utilisant le meta-langage EBNF. Ce choix s'oppose à une description basée sur des exemples. En effet, les exemples ne permettent généralement pas de présenter toutes les possibilités syntaxiques d'un langage. Cependant, ils participent souvent à la bonne compréhension d'une description théorique plus complète.

Le document est composé de deux parties. La première décrit la syntaxe du langage. La seconde décrit certains outils de base du C, notamment des fonctions utiles définies par la librairie standard du C. Le contenu de ces deux parties n'est cependant pas exhaustif. Concernant la syntaxe, les constructions les plus rares ont été omises. Enfin, les outils décrits dans la seconde partie ne constituent qu'une partie de la librairie standard du C.

1.1 Description de la syntaxe

Un langage de programmation peut être décrit par une grammaire qui spécifie les constructions autorisées. Une grammaire est constituée d'un ensemble de symboles terminaux, qui représentent les jetons de base du langage, ainsi qu'un ensemble de symboles non terminaux qui représentent les sous-constructions autorisées du langage. Le langage est décrit par un symbole non terminal « racine ».

Le meta-langage EBNF (Extended Backus-Naur Form) décrit dans la norme ISO 14977 de 1996 permet de décrire la grammaire d'un langage de programmation. Dans ce document, nous allons utiliser une version légèrement modifiée du langage EBNF pour décrire la syntaxe du langage C.

Une règle de production décrit une construction autorisée associée à un symbole non terminal. Celle-ci est décrite par la notation $\boxed{X ::= a}$ où X est un identificateur de symbole non terminal et a la description EBNF d'une construction autorisée faisant intervenir des symboles terminaux et non terminaux ainsi que les opérateurs décrits par le langage EBNF.

1. http://www.montefiore.ulg.ac.be/~boigelot/cours/info/slides/info-2009.pdf

Les opérateurs du langage EBNF sont les suivants (a, b sont des descriptions EBNF de constructions possibles) :

- (a) : permet de délimiter une expression EBNF à laquelle un opérateur peut s'appliquer.
- a^* : représente la répétition d'une construction représentée par a un nombre n de fois avec $n \geq 0$.
- a^+ : représente la répétition d'une construction représentée par a un nombre n de fois avec $n \geq 1$.
- $a \mid b$: représente soit une construction décrite par a, soit une construction décrite par b.
- [a] : la construction décrite par a est optionnelle (elle peut apparaître 0 ou 1 fois).

Les symboles terminaux sont représentés en utilisant une typographie particulière : par exemple, a est un symbole terminal.

Voici un exemple simple de grammaire EBNF décrivant une somme d'entiers contenant au moins un terme :

somme ::= entier (+ entier)*
entier ::= [-]chiffre+
chiffre ::= 0 | ... | 9

La construction suivante est valide : `42 + 666 + -1`.

1.2 Utilisation du document

La lecture du document dans son entièreté permettra au lecteur d'avoir une vue d'ensemble des possibilités de base du langage C.

Pour consulter la syntaxe liée à un concept de programmation, la consultation de la table des matières devrait permettre de trouver la section appropriée.

Un index reprenant les mots-clé du langage C, les symboles non terminaux importants utilisés dans la description syntaxique du langage C ainsi que les fonctions et directives décrites dans la deuxième partie est fourni. Cet index permet de facilement retrouver la syntaxe et/ou la définition liée à un mot-clé, un symbole non terminal, une fonction prédéfinie ou une directive.

1.3 Description générale d'un programme C

Un programme C est essentiellement constitué d'un ensemble de définitions et déclarations de types (voir Section 2.3), variables globales et constantes (voir Section 2.4), et fonctions (voir Chapitre 3). Les variables globales peuvent être modifiées par les fonctions du programme.

Le comportement d'une fonction (donné lors de sa définition) est décrit par une séquence d'instructions (voir Chapitre 5) et de définitions de types, constantes et variables locales.

Un identificateur (de variable, constante, fonction, type ; voir Section 2.1) doit toujours avoir été déclaré avant de pouvoir être utilisé. On peut noter que dans le cas des fonctions et des types, déclaration ne signifie pas définition : on peut déclarer une fonction sans pour autant donner son comportement ; de même, on peut déclarer un type sans en définir la structure.

Cette possibilité est particulièrement intéressante en cas de références croisées : une fonction `f` fait appel, dans son comportement, à la fonction `g` qui, dans son comportement, fait aussi appel à la fonction

`f`. Une solution est de déclarer les 2 fonctions puis de les définir à la suite de ces déclarations. Le même problème peut se poser lors de la déclaration d'un type et la même solution peut être apportée à ce problème.

Lors de l'invocation du programme, une fonction particulière est appelée : c'est la fonction `main` (voir Section 3.2). Le comportement de cette fonction donne donc les instructions qui seront exécutées lors de l'appel du programme. Le comportement de la fonction `main` contient généralement des appels vers les fonctions auxiliaires du programme.

Le langage C propose une librairie (c'est-à-dire une ensemble de fonctions) standard qui implémente des opérations courantes ou contient des déclarations de constantes et de types d'utilité générale (voir chapitres 6, 7, 8 et 9).

Première partie

Syntaxe

Chapitre 2

Éléments de base

2.1 identificateurs et constantes

Un identificateur (id) peut être utilisé pour nommer une variable (voir sections 2.4 et 3.1), une fonction (voir Chapitre 3) ou un type (voir Section 2.3).

id ::= letter(letter|digit)*
letter ::= a | ... | z | A | ... | Z | _
digit ::= 0 | ... | 9

Une constante (cst) peut être un entier, un nombre réel, un caractère, une chaîne de caractère, un tableau ou une structure. Une constante peut être utilisée pour initialiser une variable (voir sections 2.4 et 3.1) ou intervenir dans une expression (voir Chapitre 4).

cst ::= cst-entier | cst-reel | cst-car | cst-chaine | cst-array | cst-struct | cst-expr
cst-entier ::= [-](digit)+[l | L | ll | LL | u | U]
cst-reel ::= [-](digit)*.(digit)* [Ecst-entier][f | F | l | L]
cst-car ::= 'car'
cst-chaine ::= "(car)*"
cst-array ::= { [cst (, cst)*] }
cst-struct ::= { [id : cst (, id : cst)*] }
car ::= *Un symbole ASCII*
cst-expr ::= cst-entier | cst-reel | cst-car | (cst-expr) | cst-expr bin-op cst-expr | un-op cst-expr
bin-op ::= bin-arith-op | bin-logic-op | bin-comp-op
un-op ::= un-arith-op | un-logic-op

Pour faciliter leur description, les symboles id, cst-entier, cst-reel, cst-car et cst-chaine sont représentés ici comme des symboles non terminaux. En réalité, ils constituent des jetons du langage C et devraient

donc être considérés comme des symboles terminaux. En pratique, cela signifie que les idenfitificateurs et constantes entières, réelles et de type caractère ne peuvent pas contenir de séparateurs c'est-à-dire des espaces, des tabulations ou des retours à la ligne.

Pour les constantes entières et réelles, des modificateurs optionnels peuvent être spécifiés (voir Section 2.3.1). Les constantes reprises dans un tableau doivent toutes être du même type.

Des constantes peuvent être calculées à l'aide d'une expression (cst-expr). Cette expression ne peut faire intervenir que d'autres constantes et implique des opérateurs décrits dans les sections 4.2 et 4.3.

2.2 Le programme

Un programme C a la structure suivante :

progC : := (decl-fonct | def-fonct | decl-var | ext-var | def-struct | def-type)*

progC est le symbole racine de notre description du langage C. Un programme C est donc décrit par un entrelacement de déclarations/définitions de fonction (voir Chapitre 3), de déclarations et définitions de variables globales ou de constantes (voir Section 2.4), de définitions de types structurés (voir Section 2.3.5) et de définitions de types personnalisés (voir Section 2.3.6).

Cette suite d'éléments peut être répartie en plusieurs fichiers (voir Section 6.1).

2.3 Types

id-type ::= types-prim | struct-type | custom-id | `void`

Un identificateur de type peut représenter :
- un type primitif (types-prim, voir Section 2.3.1),
- le type spécial `void` (voir Section 2.3.3),
- un type structuré (struct-type, voir Section 2.3.5),
- un type défini par le programmeur (custom-id, voir Section 2.3.6).

2.3.1 Types primitifs

types-prim ::=
[`unsigned`|`signed`] `char` |
[`unsigned`|`signed`] [`short` | (`long` [`long`])] `int` |
`float` |
[`long`] `double`

Les types primitifs sont :
– l'entier : `int`,
– le caractère ou entier sur 8 bits : `char`,
– le nombre réel (simple précision) : `float`,
– le nombre réel (double précision) : `double`.
Des modificateurs peuvent être associés à ces types de base :
– `signed` / `unsigned` : le type associé est signé / non-signé,
– `short` : diminue la taille de l'entier
– `long` : augmente la taille de l'entier ou la précision d'un nombre réel (double précision vers précision étendue),
– `long long` : augmente encore la taille de l'entier.

2.3.2 Pointeurs

Pour indiquer qu'une variable de type T est un pointeur, son identificateur est préfixé lors de sa déclaration (voir Section 2.4) :

ptr-pref ::= (*)*

Le nombre d'étoiles réprésente le nombre de niveaux de « redirection » :
– aucune étoile : la variable n'est pas un pointeur,
– 1 étoile : la variable est un pointeur de T,
– 2 étoiles : la variables est un pointeur de pointeur de T,
– ...

2.3.3 Type `void`

Le type `void` ne peut être utilisé que dans certaines circonstances particulières :
– avec un pointeur : on ne fait pas de supposition sur le type de la zone mémoire pointée.
– dans le contexte de la déclaration/définition d'une fonction (voir Chapitre 3).

2.3.4 Tableaux

Pour indiquer qu'une variable est un tableau, un suffixe est associé à son identificateur lors de la déclaration (voir Section 2.4) :

arr-suff ::= ([[cst-entier]])*

Le nombre de paires d'accolades représente le nombre de dimensions du tableau et la constante comprise entre les accolades représente la taille de la dimension associée :
– aucune paire d'accolades : la variable n'est pas un tableau,

– $[n_1]$: la variable est un vecteur de taille n_1,
– $[n_1][n_2]$: la variable est un tableau à 2 dimensions de taille (n_1, n_2),
– ...

2.3.5 Type structuré

Un type structuré est défini comme suit :

struct-type ::= `struct` [nom-struct] [`{` liste-champs `}`]
liste-champs ::= (id-type ptr-pref id arr-suff (`,` ptr-pref id arr-suff) `;`)*
nom-struct ::= id

Une structure peut être définie sans pour autant devoir déclarer de variable :

def-struct ::= struct-type `;`

2.3.6 Définition de type

Le mot-clé `typedef` peut être utilisé pour représenter une définition de type potentiellement complexe par un identificateur :

custom-id ::= id
def-type ::= `typedef` id-type id `;`

Cette construction est particulièrement intéressante pour remplacer la définition d'un type structuré par un identificateur de type simple.

2.3.7 Chaînes de caractères

Il n'y a pas de type « chaîne de caractères » explicite en C. Celles-ci sont simplement représentées par un tableau de caractères. La fin de la chaîne est indiquée par le caractère spécial `'\0'`. En pratique, cela implique qu'un tableau de N caractères peut contenir une chaîne d'au plus $(N-1)$ caractères, le $N^{\text{ème}}$ caractère étant le caractère de terminaison de chaîne.

2.4 Déclaration de variables globales et de constantes

decl-var ::= [`static`] [`const`] id-type ptr-pref id arr-suff [= cst] (, ptr-pref id arr-suff [= cst])* ;
ext-var ::= `extern` type-id prt-pref id arr-suff ;

Lors de la déclaration d'une variable, son type et son identificateur son définis. Un préfixe (ptr-pref) peut être ajouté à l'identificateur de variable afin d'indiquer que la variable est un pointeur (voir Section 2.3.2). Un suffixe (arr-suff) permet d'indiquer si la variable est un tableau (voir Section 2.3.4).

Pour faciliter l'écriture, un type peut être associé à une liste d'identificateurs et leur préfixe et suffixe.

Dans l'exemple qui suit, on déclare 4 variables : `w` un nombre réel représenté en double précision, `x` un entier, `y` un pointeur de pointeur d'entier et `z` un tableau de 5 entiers.

Le mot-clé `static` signifie qu'une variable globale est statique c'est-à-dire qu'elle a une portée limitée au fichier source dans lequel elle est définie.

Le mot-clé `const` signifie que l'on définit une constante et non une variable.

Dans le cas où une variable globale n'est pas statique, elle peut être utilisée dans un autre fichier source que celui où elle est définie. La variable doit alors être déclarée (ext-var) dans les fichiers où elle sera utilisée.

2.5 Exemples de déclarations de types et variables

```
typedef struct A {
    int x;
    double z;
} B;

B c = {x : 0, z : 1E-2};
const double w = 3.14;
int x = 0, **y = 0, z[5] = {1, 2, 3, 4, 5};
```

L'exemple ci-dessus reprend, dans l'ordre :
- la définition d'un type structuré `struct A` représenté par un identificateur `B`,
- la déclaration d'une variable globale `c` de type `B`,
- la déclaration d'une constante `w` qui a pour valeur $3,14$,
- la déclaration de 3 variables :
  - `x` une variable entière,
  - `y` un pointeur de pointeur d'entier,
  - `z` un tableau de 5 entiers.

Chapitre 3

Fonctions

Une fonction peut être déclarée (decl-fonct) plusieurs fois dans un programme (à condition que ce soit toujours de la même manière) mais ne peut être définie (def-fonct) qu'une seule fois.

decl-fonct ::= [`static`] [`const`] id-type ptr-pref id `(` proto-args-list `)` `;`
proto-args-list ::= [`const`] id-type ptr-pref [id] arr-suff (`,` [`const`] id-type ptr-pref [id] arr-suff)*
def-fonct ::= [`static`] [`const`] id-type ptr-pref id `(` args-list `)` bloc-instr
args-list ::= [`const`] id-type ptr-pref id arr-suff (`,` [`const`] id-type ptr-pref id arr-suff)*
bloc-instr ::= `{` instr-seq `}`

Le type fictif `void` (voir Section 2.3.3]) est utilisé afin d'indiquer l'absence de type de retour (la fonction est en fait une procédure). Il peut également être utilisé pour indiquer l'absence de paramètres d'une fonction (la liste de types utilisée lors de la déclaration d'une fonction contient un élément : `void`).

Le mot-clé `static` signifie qu'une fonction est statique c'est-à-dire qu'elle a une portée limitée au fichier source dans lequel elle est définie.

Le mot-clé `const` signifie que la valeur de l'argument ou du type de retour associé est une constante.

La définition d'une fonction inclu le corps de la fonction qui représenté par un bloc d'instructions. Un bloc d'instructions contient une séquence d'instructions.

instr-seq ::= (instr `;` | decl-loc-var | def-struct | def-type)*
instr ::= base-instr | expr | `return` [expr]

L'instruction `return` a pour effet d'interrompre la fonction en cours d'exécution et implique le renvoi de la valeur de l'expression spécifiée.

3.1 Variables locales

Lorsqu'une variable est déclarée dans le corps d'une fonction, elle est dite locale.

Les variables locales sont déclarées d'une manière similaire aux variables globales. Elles peuvent cependant être initialisées avec la valeur d'une expression et non plus seulement avec une constante.

decl-loc-var ::= [`static`] [`const`] id-type ptr-pref id arr-suff [`=` expr] (`,` ptr-pref id arr-suff [`=` expr])* `;`

Le mot-clé `static` permet de déclarer une variable locale statique : la valeur d'une telle variable est préservée d'un appel à l'autre de cette fonction.

3.2 La fonction `main`

Dans un programme C, exactement une fonction doit s'appeler `main`. Les instructions constituant le corps de cette fonction seront exécutées à l'appel du programme.

La fonction `main` peut être déclarée de 2 manières différentes :

```
int main();
```

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

Le premier prototype implique qu'on ignore les paramètres fournis au programme. Le deuxième prototype donne accès à ces paramètres :

- `argc` est une variable entière qui indique le nombre d'arguments,
- `argv` est un tableau de chaînes de caractères représentant ces arguments.

La valeur de retour de la fonction `main` est utilisé comme code de retour du programme.

La fonction `void exit(int code)` permet d'interrompre l'exécution du programme à partir de n'importe où dans le code. Le code de retour du programme est celui donné en argument.

3.3 Exemples de déclarations de fonctions

```
void f(int *x, const char c);
int *g(double);
int h(void);
```

L'exemple ci-dessus comporte la déclaration de 3 fonctions :

- `f` est en fait une procédure et prend 2 arguments : un pointeur d'entier `x` et une constante de type caractère `c`,

– `g` est une fonction qui retourne un pointeur d'entier et qui prend en argument un nombre réel représenté en double précision,
– `h` est une fonction qui ne prend aucun argument et qui retourne un entier.

Chapitre 4

Expressions

Une expression est une instruction exécutable qui représente une valeur.

expr ::= cst | id | (expr) | affect-expr | arith-expr | logic-expr | arith-affect-expr | cond-expr | array-access-expr | field-access-expr | func-call-expr | ptr-deref-expr | ptr-get-expr | type-conv-expr | pair-expr | sizeof-expr

4.1 Affectation

L'affectation permet de stocker une valeur dans un variable via un opérateur prenant deux opérandes.

affect-expr ::= expr = expr

L'opérande de gauche doit avoir une valeur à gauche : elle doit représenter un emplacement de stockage valide (une variable, un champ de structure, la case d'un tableau).

Par exemple, l'affectation suivante est invalide :

```
(4*3) = 2;
```

Par contre, l'affectation suivante est valide :

```
int *x;
...
(*x) = 2;
```

4.2 Expressions arithmétiques

Une expression arithmétique fait intervenir des opérandes numériques et a également une valeur numérique.

arith-expr ::= expr bin-arith-op expr | un-arith-op expr
bin-arith-op ::= + | - | * | / | %
un-arith-op ::= -

Les opérateurs arithmétiques sont les suivants :
- + : addition,
- - : soustraction (opérateur binaire) ou changement de signe (opérateur unaire),
- / : division,
- * : mutliplication,
- % : reste de la division (modulo).

Le type de division dépend des opérandes : si les deux opérandes sont entières, alors la division entière est employée. Sinon, la division réelle est utilisée.

4.3 Expressions logiques

Une expression logique fait intervenir des opérandes booléennes ou numériques et a une valeur booléenne.

Le C n'a pas de type booléen explicite. À la place, une valeur entière est interprétée comme une valeur de vérité : si une expression a comme valeur 0, elle aura comme valeur booléenne *false* ; si une expression a une valeur différente de 0, elle aura comme valeur booléenne *true*.

logic-expr ::= expr bin-logic-op expr | un-logic-op expr | expr bin-comp-op expr
bin-logic-op ::= && | ||
un-logic-op ::= !
bin-comp-op ::= > | < | >= | <= | == | !=

Les opérateurs logiques sont les suivants :
- ! : négation,
- && : « et » logique,

– || : « ou » logique.

Les opérateurs de comparaison sont les suivants :

– > et < : inégalités strictes,
– >= et <= : inégalités non-strictes,
– == : égalité,
– != : différence.

4.4 Affectations combinées

Les opérateurs arithmétiques peuvent être combinés avec l'affectation pour réduire l'écriture. L'incrémentation et la décrémentation possèdent également une écriture raccourcie.

arith-affect-expr ::= expr com-affect-op expr | ++expr | --expr | expr++ | expr--
com-affect-op ::= += | -= | *= | /= | %=

La première opérande des opérateurs décrits par le symbole com-affect-op doit être une valeur à gauche valide. De même pour l'opérande des opérateurs d'incrémentation et de décrémentation.

Les opérateurs ++ et -- permettent respectivement d'incrémenter et de décrémenter la valeur stockée dans une variable.

Ainsi, soit `x` une variable entière, `++x` et `x++` (resp. `--x` et `x--`) incrémentent (resp. décrémentent) la valeur de `x`. La différence entre la version préfixée et la version suffixée est la valeur de l'expression : `++x` a pour valeur le résultat de l'incrémentation et `x++` a pour valeur la valeur de `x` avant incrémentation.

4.5 Condition

L'expression conditionnelle se base sur un opérateur ternaire : la valeur de la première opérande permet de sélectionner une des valeurs des deux opérandes suivantes.

cond-expr ::= expr ? expr : expr

L'expression qui précède ? est une expression logique. Si sa valeur est *true*, la valeur de l'expression conditionnelle est la valeur de l'expression comprise entre ? et :. Sinon, la valeur de l'expression conditionnelle est la valeur de l'expression suivant :.

Par exemple, l'expression `(b ? 1 : 0)` aura comme valeur `1` si la variable entière `b` a une valeur différente de 0. Sinon, l'expression vaut `0`.

4.6 Accès aux éléments d'un tableau

L'accès aux éléments d'un tableau se fait via un opérateur prenant deux opérandes : un tableau et un indice.

array-access-expr ::= expr [expr]

Soit un tableau *t* la valeur de la première opérande et un entier *i* la valeur de la seconde opérande. La valeur de l'expression est la valeur du $(i+1)^{\text{ème}}$ élément de *t*.

La valeur de la première opérande peut aussi être un pointeur vers un tableau. Ce tableau peut avoir été alloué dynamiquement (voir Chapitre 7).

Par exemple, l'expression `t[2]` retourne le 3ème élément du tableau `t` qui peut avoir la déclaration `int *t;` ou `int t[N]` où `N` est la taille du tableau.

4.7 Appel de fonction

L'appel de fonction se fait via un opérateur prenant au moins un opérande (la fonction à appeler). Les opérandes suivants représentent les valeurs qui seront passées en argument à la fonction.

func-call-expr ::= expr ([expr-list])
expr-list ::= expr (, expr)*

4.8 Accès au pointeur

La valeur du pointeur d'une variable est obtenue via un opérateur unaire.

ptr-get-expr ::= & expr

La valeur de l'opérande doit être une valeur à gauche valide (voir Section 4.1).

Par exemple, soit `x` une variable entière, alors l'expression `&x` a pour valeur le pointeur de `x`.

4.9 Déréférencement de pointeur

La valeur d'une variable pointée est obtenue via un opérateur unaire.

ptr-deref-expr ::= ptr-pref expr

La valeur de l'opérande doit être une valeur à gauche valide (voir Section 4.1).

Par exemple, soit `x` un pointeur d'entier, alors l'expression `*x` a pour valeur l'entier contenu dans la variable pointée.

4.10 Accès au champ d'une structure

L'accès au champ d'une structure peut se faire via deux opérateurs prenant deux opérandes :
- `.` : une structure et un identificateur de champ,
- `->` : un pointeur de structure et un identificateur de champ,

field-access-expr ::= expr `.` id | expr `->` id

La valeur de la première opérande doit être une structure (`.`) ou un pointeur de structure (`->`). La valeur de cette expression est la valeur du champ demandé dans la structure donnée.

L'opérateur `->` est un raccourci d'écriture. Soit `s` un pointeur de structure et `x` l'identificateur d'un champ entier de la structure pointée par `s`, alors l'expression `((*s).x)` est équivalente à l'expression `(s->x)`.

4.11 Conversion de type explicite

L'opérateur de conversion de type prend deux opérandes : un type de « destination » et une valeur dont le type va être converti.

type-conv-expr ::= `(` id-type ptr-pref `)` expr

4.12 Paire d'expressions

pair-expr ::= expr `,` expr

La valeur d'une paire d'expressions est la valeur de la deuxième expression de la paire.

Par exemple, l'expression `(1, 2)` a pour valeur `2`.

4.13 Taille d'un type

sizeof-expr ::= `sizeof` `(` expr | id-type ptr-pref arr-suff `)`

L'opérateur `sizeof` prend comme opérande une valeur à gauche ou un type et a comme valeur la taille en octets du type associé à la valeur à gauche ou du type spécifié.

Par exemple, `sizeof(int *)` a pour valeur le nombre d'octets nécessaire pour représenter un pointeur d'entier.

Chapitre 5

Instructions

Contrairement à l'expression, l'instruction n'a pas de valeur.

base-instr ::= bloc-instr | cond-instr | loop-instr | `break` | `continue`

5.1 Choix conditionnels

Le langage C reprend deux types de conditions.

cond-instr ::= if-cond | switch-cond

5.1.1 Choix conditionnel binaire

Une expression est utilisée comme gardien. Si cette expression a la valeur *true*, l'instruction qui suit le gardien est exécutée. Sinon, si elle est spécifiée, c'est l'instruction qui suit le mot-clé `else` qui est exécutée.

if-cond ::= `if` `(` expr `)` instr [`else` instr]

Par exemple, la suite d'instructions suivante implique que la variable `b` aura toujours comme valeur `1`.

```
int b;
if(1)
    b = 1;
else
    b = 0;
```

5.1.2 Choix conditionnel multiple

switch-cond ::= `switch` `(` expr `)` `{` switch-cases [switch-default] `}`
switch-cases ::= (`case` cst `:` instr)*
switch-default ::= `default` `:` instr

Une expression est utilisée comme gardien. L'instruction à exécuter est sélectionnée parmi une liste de « cas » proposés. Si la valeur de la constante associée à un cas est égale à la valeur du gardien, alors les instructions qui suivent le cas sont exécutées jusqu'à rencontrer la fin du bloc « switch » (c'est-à-dire l'accolade fermante du bloc switch) ou le mot-clé `break`.

Par exemple, les instructions ci-dessous impliquent la définition suivante pour la variable `x` :

$$\texttt{x} = \begin{cases} 3-n & \text{si } 0 \leq n \leq 2 \\ -1 & \text{si } n < 0 \text{ ou } n > 2 \end{cases}$$

```
int x = 0;
switch(n) {
    case 0 : ++x;
    case 1 : ++x;
    case 2 : ++x; break;
    default : x = -1;
}
```

5.2 Boucles

Il y a essentiellement 3 types de boucles en C.

loop-instr ::= while-loop | do-loop | for-loop

5.2.1 Boucle while

while-loop ::= `while` `(` expr `)` instr

Une expression est utilisée comme gardien. Une instruction est exécutée tant que le gardien est *true*.

Par exemple, la boucle suivante incrémente la variable `x` jusqu'à ce qu'elle atteigne la valeur `10`.

```
int x = 0;
while(x < 10)
    ++x;
```

5.2.2 Boucle do

do-loop ::= `do` instr `while` `(` expr `)`

Une expression est utilisée comme gardien. Une instruction est exécutée tant que le gardien est *true*. Le gardien est évalué après l'exécution de l'instruction, ce type de boucle implique donc toujours au moins une itération.

Par exemple, la boucle suivante incrémente la variable `x` jusqu'à ce qu'elle atteigne la valeur `10`.

```
int x = 0;
do
    ++x;
while(x < 10);
```

5.2.3 Boucle for

for-loop ::= `for` `(` [for-init] `;` [expr] `;` [expr] `)` instr
for-init ::= for-var-decl | expr
for-var-decl ::= id-type ptr-pref id arr-suff (`,` ptr-pref id arr-suff)*

La boucle for implique 4 éléments :

– for-init : les instructions exécutées avant l'exécution de la boucle – ces instructions peuvent être des déclarations de variables ;
– premier expr : le gardien évalué avant chaque itération – si le gardien est *true*, la boucle continue son exécution ;
– second expr : les instructions exécutées à la fin de chaque itération ;
– instr : les instructions exécutées à chaque itération (avant celles du second expr).

Les 3 premiers éléments sont optionnels. L'absence de gardien revient à remplacer l'expression par une constante différente de 0 (donc *true*).

Par exemple, la boucle suivante incrémente la variable `x` jusqu'à ce qu'elle atteigne la valeur `10`.

```
for(int x = 0; x < 10; ++x);
```

5.2.4 Instructions de rupture de séquence

L'instruction `break` peut être utilisée pour interrompre l'exécution d'une boucle. L'exécution reprend à la première instruction suivant la boucle.

L'instruction `continue` peut être utilisée pour interrompre l'exécution d'une itération de boucle. L'exécution reprend au début de l'itération suivante de la boucle si le gardien est vérifié et à la première instruction suivant la boucle sinon.

Mis à part `break` qui peut être utilisé dans le contexte d'un bloc switch, les instructions de rupture de séquence ne peuvent être utilisées que dans le corps d'une boucle.

Deuxième partie

Outils de base

Chapitre 6

Préprocesseur

Les compilateurs du langage C incluent un outil appelé « préprocesseur ». Cet outil permet notamment la définition de variables (qui ne sont pas des variables du programme C cependant) et une forme de manipulation de fichiers.

Les directives données au préprocesseur tiennent généralement sur une ligne préfixée par le symbole # suivi d'une commande.

6.1 Inclusion de fichier

La directive `#include` permet l'inclusion du contenu d'un fichier dans un autre. Soit un fichier `file1` dont le contenu est noté A. Soit un autre fichier `file2` contenant la directive `#include "file1"` ou `#include <file1>`. Soient B_1 le contenu du fichier `file2` avant la directive d'inclusion et B_2 le contenu du fichier `file2` après la directive d'inclusion.

Si le fichier `file2` est fourni en entrée à un compilateur C, celui-ci compilera B_1AB_2, à savoir le contenu du fichier `file1` inclus dans le fichier `file2`.

Les directives d'inclusion de fichier permettent de répartir un programme C dans plusieurs fichiers, d'où une possibilité de structurer son code.

L'utilisation des <> implique que le fichier à inclure fait partie de la librairie standard du C ou d'une autre librairie considérée comme faisant partie du système de compilation. L'utilisation des `""` indique que le fichier à inclure à été défini par le programmeur dans le contexte de son programme.

6.1.1 Séparation interface / implémentation

Pour écrire du code générique et réutilisable, la séparation entre interface et implémentation est un outil de base. L'interface représente une description de comment les fonctionnalités proposées par une bibliothèque peuvent être utilisées tandis que l'implémentation décrit la manière dont ces fonctionnalités seront exécutées.

Une bibliothèque C se présente donc généralement sous la forme de son interface, répartie dans un ou plusieurs fichiers *en-tête*, et son implémentation répartie dans un ou plusieurs fichiers *source*.

Les fichiers en-tête ont généralement l'extension `h` et contiennent les *déclarations* de type, fonctions et variables globales.

Les fichiers source ont généralement l'extension `c` et contiennent les *définitions* de fonctions et variables globales ainsi que des définitions de types « privés » c'est-à-dire des types qui ne sont pas accessibles via l'interface.

Toutes les fonctions définies dans un fichier source ne doivent pas être déclarées dans le fichier en-tête associé. En effet, certaines fonctions sont propres à l'implémentation de la bibliothèque et ne devraient pas être accessibles à l'utilisateur.

6.2 Définition de macros

La directive `#define` permet la définition de macros. Une définition de macro consiste en l'association d'un texte de substitution à un nom de macro. Les occurences du nom de macro dans le programme C sont remplacées par le texte de substitution.

Le texte de substitution peut être paramétré par des arguments de macro. Ces arguments sont représentés par des identificateurs qui peuvent apparaître dans le texte de substitution. Lorsqu'une macro est mentionnée dans le programme, des valeurs sont associées à ces paramètres et sont « intégrées » dans le texte de substitution qui va remplacer la macro.

La syntaxe d'une définition de macro est la suivante :

#define name [(id (, id)*)] text

Chapitre 7

Gestion de la mémoire

Les fonctions de gestion de la mémoire présentées dans ce chapitre sont accessibles via le fichier en-tête `<stdlib.h>`.

7.1 Allocation dynamique

La fonction `void *malloc(unsigned int n)` alloue un bloc de `n` octets sur le tas et retourne un pointeur vers ce bloc ou `NULL` en cas d'erreur.

La fonction `void *realloc(void *p, unsigned int n)` tente de redimensionner le bloc pointé par `p` (qui doit précédemment avoir été alloué) afin de lui donner une nouvelle taille `n`. Le bloc peut être déplacé. La fonction retourne un pointeur vers le nouvel emplacement du bloc ou `NULL` en cas d'erreur.

7.2 Désallocation dynamique

La fonction `void free(void *p)` libère le bloc pointé par `p` (qui doit précédemment avoir été alloué). Si `p` vaut `NULL`, l'appel n'a aucun effet.

Chapitre 8

Entrées/sorties

Les fonctions d'entrée/sortie sont accessibles via le fichier `<stdio.h>`.

8.1 Entrées/sorties standard formatées

Généralement, la sortie standard est la console et l'entrée standard le clavier. Écrire quelque chose sur la sortie standard revient donc à afficher des données à l'écran et lire à partir de l'entrée standard revient à récupérer des données fournies par l'utilisateur via le clavier.

8.1.1 Sorties

La fonction `int printf(const char *format, ...)` permet d'écrire des données formatées sur la sortie standard. Elle retourne le nombre de caractères écrits en cas de succès et un nombre négatif sinon. Cette fonction prend au moins un argument : une chaîne de caractères représentant le format des données à afficher. Cette chaîne peut contenir des marqueurs qui indiquent où et comment des valeurs passées en argument devront être affichées.

Ces marqueurs ont la syntaxe suivante :

```
%[drapeau][largeur][.précision][longueur]spécificateur
```

Les différents arguments d'un marqueur servent à :
- spécifier le type de donnée à afficher (spécificateur et longueur, voir tables 8.5 et 8.4 respectivement),
- contrôler la quantité de caractères à afficher (largeur et précision, voir tables 8.2 et 8.3 respectivement),
- contrôler la manière dont le remplissage est effectuer (voir Table 8.1).

8.1.2 Entrées

La fonction `int scanf(const char *format, ...)` permet de lire des données formatées à partir de l'entrée standard. Elle retourne le nombre de données qui ont été correctement lues. Ce nombre peut être inférieur au nombre de données qui devraient être lues en cas d'erreur. Si une erreur d'entrée intervient avant même qu'une donnée ne soit lue, une valeur négative est retournée.

Drapeau	Description
`+`	Justification à gauche (par défaut, justification à droite).
`-`	Force l'affichage du signe.
espace	Si le nombre est positif, un espace est introduit à la place du signe.
`#`	– Avec les spécificateurs `o`, `x` et `X`, préfixe la valeur à afficher avec respectivement `0`, `0x` ou `0X` si la valeur est différente de 0. – Avec les spécificateurs `e`, `E`, `g`, `G` et `f`, force l'affichage du point, même en l'absence de chiffres après la virgule.
`0`	Les espaces ajoutés à gauche du nombre pour remplir l'espace spécifié sont remplacés par des 0.

TABLE 8.1 – Drapeaux de format.

Largeur	Description
nombre	Le nombre minimum de caractères à afficher. Si la valeur à afficher est trop courte, des espaces ou des zéros sont ajoutés à gauche de celui-ci.
`*`	Le nombre minimum de caractères à afficher n'est pas spécifié dans la chaîne de format mais via un argument entier passé avant l'argument associé à ce marqueur.

TABLE 8.2 – Largeur de format.

Précision	Description
nombre	– Avec les spécificateurs `d, i, o, u, x, X` : donne le nombre minimum de chiffres à afficher. Si la valeur à afficher est trop courte, des zéros sont ajoutés à gauche du nombre. – Avec les spécificateurs `e, E, F` : donne le nombre de chiffres à afficher après la virgule. – Avec les spécificateurs `g, G` : donne le nombre maximum de chiffres significatifs à afficher après la virgule. – Avec le spécificateur `s` : donne le nombre maximum de caractères à afficher.
`*`	Le nombre minimum de caractères à afficher n'est pas spécifié dans la chaîne de format mais via un argument entier passé avant l'argument associé à ce marqueur.

TABLE 8.3 – Précision de format.

Longueur	Description
`h`	Avec les spécificateurs `i, o, d, u, x, X`, l'argument est interprété comme un `short int` ou `unsigned short int`.
`l`	Avec les spécificateurs `i, o, d, u, x, X`, l'argument est interprété comme un `long int` ou `unsigned long int`.
`L`	Avec les spécificateurs `e, E, f, g, G`, l'argument est interprété comme un `long double`.

TABLE 8.4 – Longueur de format.

Spécificateur	Sortie	Exemple
`c`	Caractère	a
`d` ou `i`	Entier décimal signé	392
`e`	Notation scientifique utilisant `e`	3.9265e+2
`E`	Notation scientifique utilisant `E`	3.9265E+2
`f`	Nombre réel décimal	392.65
`g`	Notation la plus courte choisie entre `%e` et `%f`	392.65
`G`	Notation la plus courte choisie entre `%E` et `%f`	392.65
`o`	Entier octal signé	610
`s`	Chaîne de caractères	exemple
`u`	Entier décimal non signé	7235
`x`	Entier hexadécimal non signé	7fa
`X`	Entier hexadécimal non signé (avec lettres en majuscule)	7FA
`p`	Adresse mémoire (pointeur)	B800 :0000
`n`	Rien n'est affiché. L'argument associé doit être un pointeur d'entier qui contiendra le nombre caractères qui ont été affichés jusque là.	a
`%`	%	%

TABLE 8.5 – Spécificateurs de format.

Comme `printf`, cette fonction prend au moins un argument : une chaîne de caractères représentant le format des données à lire.

Cette chaîne peut contenir les éléments suivants :
- un caractère « espace » : indique qu'à ce point, la fonction va ignorer tous les espaces, retours à la ligne et tabulations qui seront lus à partir de l'entrée ;
- un caractère qui n'est ni un espace, ni un signe de pourcentage (%) : la fonction lit un caractère à partir de l'entrée standard et le compare au caractère de la chaîne de format. Si les caractères correspondent, la fonction passe au caractère suivant de la chaîne de formatage et de l'entrée. Sinon, la fonction interrompt son exécution ;
- un marqueur (préfixé par le signe de pourcentage) : la fonction va tenter de lire une donnée à partir de l'entrée standard et la stocker dans une variable dont le pointeur est passé en argument à `scanf` puis passera à la suite de la chaîne de formatage en cas de succès.

Un marqueur a la forme suivante :

```
%[*][largeur][longueur]spécificateur
```

où :
- `*` indique que la donnée associée au marqueur doit être lue à partir de l'entrée mais pas stockée dans la variable associée,
- `largeur` indique le nombre maximum de caractères à lire lors de l'acquisition de la donnée,
- `longueur` et `spécificateur` donne le type de donnée à lire (voir tables 8.6 et 8.7).

8.2 Fichiers

Le type `FILE` représente un « handle » de fichier. Un pointeur vers une instance de ce type est retournée par la fonction d'ouverture de fichier. Ce pointeur peut ensuite être passé en argument à la plupart des

Spécificateur	Sortie	Type de l'argument
`c`	Caractère	`char *`
`d` or `i`	Entier décimal signé (éventuellement précédé de + ou -)	`int *`
`e`	Notation scientifique utilisant `e` (éventuellement précédé de + ou -)	`float *`
`E`	Notation scientifique utilisant `E` (éventuellement précédé de + ou -)	`float *`
`f`	Nombre réel décimal (éventuellement précédé de + ou -)	`float *`
`g`	Notation choisie entre `%e` et `%f`	`float *`
`G`	Notation choisie entre `%E` et `%f`	`float *`
`o`	Entier octal signé	`int *`
`s`	Chaîne de caractères	`char *`
`u`	Entier décimal non signé	`unsigned int *`
`x`	Entier hexadécimal non signé	`int *`
`X`	Entier hexadécimal non signé (avec lettres en majuscule)	`int *`

TABLE 8.6 – Spécificateurs de format.

Longueur	Description
`h`	Avec les spécificateurs `i, o, d, u, x, X`, l'argument attendu est de type `short int *` ou `unsigned short int *`.
`l`	– Avec les spécificateurs `i, o, d, u, x, X`, l'argument attendu est de type `long int *` ou `unsigned long int *`. – Avec les spécificateurs `e, E, f, g, G`, l'argument attendu est de type `double *`.
`L`	Avec les spécificateurs `e, E, f, g, G`, l'argument attendu est de type `long double *`.

TABLE 8.7 – Longueur de format.

fonctions de manipulation de fichier.

8.2.1 Ouverture et fermeture de fichier

Un fichier peut être ouvert en utilisant la fonction `FILE * fopen(const char *filename, const char *mode)` qui retourne un pointeur vers un handle si l'ouverture a été correctement effectuée et `NULL` sinon.

L'argument `filename` représente le nom du fichier à ouvrir. Le format de ce nom est dépendant du système d'exploitation.

L'argument `mode` indique le type d'accès de fichier :
- `"r"` ouvre un fichier en lecture seule ; le fichier doit exister.
- `"w"` ouvre un fichier en écriture ; si le fichier existe déjà, celui-ci est vidé. Sinon, un fichier vide est créé.
- `"r+"` ouvre un fichier existant en lecture et écriture ; ce mode peut être utilisé pour la mise à jour de fichiers.
- `"w+"` ouvre un fichier en lecture et écriture ; si le fichier existe, il est vidé de son contenu. Sinon, un fichier vide est créé.
- `"a+"` ouvre un fichier dans un mode « ajout de données » ; les données ne seront écrites qu'en fin de fichier. Par contre, les données peuvent être lues n'importe où.

Par défaut, les fichiers sont ouverts en mode « texte ». Lorsqu'un fichier doit être ouvert en mode « binaire », le suffixe `b` doit être ajouté en fin de mode.

La fonction `int fclose(FILE *stream)` fermer le fichier fourni en argument et retourne 0 en cas de succès. Une valeur négative est retournée en cas d'erreur. Lors de la fermeture d'un fichier, le contenu du tampon de sortie lié à ce fichier est écrit dans le fichier.

La fonction `int fflush(FILE *stream)` permet d'écrire le contenu du tampon de sortie dans le fichier et de vider celui-ci sans fermer le fichier. Cette fonction retourne 0 en cas de succès et une valeur différente de 0 sinon.

8.2.2 État d'un fichier ouvert

Certaines fonctions permettent d'obtenir une indication sur l'état d'un fichier ouvert : Est-on arrivé en fin de fichier ? Ene erreur est-elle survenue ? Etc.

`int feof(FILE *stream)` retourne une valeur différente de 0 si la fin de fichier a été atteinte et 0 sinon.

`int ferror(FILE *stream)` retourne une valeur différente de 0 si une erreur s'est produite lors de la manipulation du fichier et 0 sinon.

8.2.3 Déplacemement de la tête de lecture/écriture

La fonction `int fseek(FILE *stream, long int offset, int origin)` permet de déplacer la tête de lecture/écriture d'un fichier de `offset` positions à partir du point de départ `origin`. La fonction retourne 0 en cas de succès et une valeur négative sinon. `offset` est exprimé en octets et peut être négatif (déplacement « vers la gauche »). `origin` représente le point de départ du déplacement à effectuer. Voici ses 3 valeurs possibles :
- SEEK_SET : le point de départ est le début du fichier,
- SEEK_CUR : le point de départ est la position courante dans le fichier,

– SEEK_END : le point de départ est la fin du fichier.

Une autre fonction intéressante lorsqu'on déplace la tête de lecture/écriture est `long int ftell(FILE *stream)` qui retourne la position courante de la tête de lecture/écriture dans le fichier `stream` ou une valeur négative en cas d'erreur.

8.2.4 Entrées/sorties non formatées sur des fichiers

La fonction `int fgetc(FILE *stream)` lit le caractère présent sous la tête de lecture et avance celle-ci d'une position tant que la fin de fichier n'a pas été atteinte. La fonction retourne le caractère lu ou `EOF` si la fin de fichier a été atteinte ou une erreur a été rencontrée.

La fonction `int fputc(int character, FILE *stream)` écrit le caractère donné via l'argument `character` sous la tête de lecture et avance celle-ci d'une position. La fonction retourne le caractère écrit ou `EOF` si une erreur a été rencontrée.

Pour de meilleures performances, il est recommandé d'écrire ou lire des blocs de données plutôt que caractère par caractère. Les fonctions suivantes permettent ces opérations.

`size_t fread(void *ptr, size_t size, size_t count, FILE *stream)` permet de lire `count` éléments de taille `size` et de les stocker dans une zone pointée par `ptr`. La fonction retourne le nombre d'octets qui ont été lus. Cette valeur peut être inférieure à `(size * count)` si la fin de fichier a été atteinte ou une erreur s'est produite.

`size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)` permet d'écrire `count` éléments de taille `size` dans un fichier. Ces éléments sont issus de la zone pointée par `ptr`. La fonction retourne le nombre d'octets qui ont été écrits. Cette valeur peut être inférieure à `(size * count)` si une erreur s'est produite.

8.2.5 Entrées/sorties formatées sur des fichiers

La fonction `int fprintf(FILE *stream, const char *format, ...)` s'utilise de la même manière que la fonction `printf` (voir Section 8.1.1) mais requiert un argument supplémentaire `stream` spécifiant un fichier.

La fonction `int fscanf(FILE *stream, const char *format, ...)` s'utilise de la même manière que la fonction `scanf` (voir Section 8.1.2) mais requiert un argument supplémentaire `stream` spécifiant un fichier.

Chapitre 9

Manipulation de chaînes de caractères

La librairie standard du langage C inclus des fonctions de base de manipulation de chaînes de caractères : copie, comparaison, concaténation, etc. Celles-ci sont déclarées dans le fichier `<string.h>`.

9.1 Taille

`size_t strlen(const char *s)` retourne la longueur de la chaîne pointée par `s` c'est-à-dire le nombre de caractères qui précèdent le caractère de terminaison de chaîne `'\0'`.

9.2 Initialisation

`void *memset(void * ptr, int val, size_t num)` initialise les `num` premier octets de la zone pointée par `ptr` à `val` (interprété comme un `unsigned char`). La fonction retourne `ptr`.

9.3 Copie

`void *memcpy(void * dest, const void * src, size_t num)` copie une sous-chaîne de taille `num` (en octets) à partir d'une zone source `src` vers une zone de destination `dest`. La terminaison de chaîne n'est pas prise en compte lors de la copie. La fonction retourne `dest`.

`void *strcpy(char * dest, const char * src` copie les caractères, y compris le caractère de terminaison de chaîne, d'une chaîne pointée par `src` vers une zone de destination `dest`. La fonction retourne `dest`.

`void *strncpy(char * dest, const char * src, size_t num)` copie les `num` premiers caractères d'une chaîne pointée par `src` vers une zone de destination `dest`. Si la taille s de la chaîne `src` est plus petite que `num`, le caractère `'\0'` est inséré $(\texttt{num} - s)$ fois à la suite de la chaîne copiée dans `dest`. Le caractère de terminaison de chaîne n'est pas implicitement inséré à la fin de la chaîne dans la zone de destination. La fonction retourne `dest`.

`strncpy` devrait être préférée à `strcpy` car elle permet de se protéger explicitement contre un overflow (la chaîne pointée par `src` est trop grande pour tenir dans l'espace pointé par `dest`).

9.4 Comparaison

`int memcmp(const void * p1, const void * p2, size_t num)` compare une les `num` premiers octets de deux zones pointées par `p1` et `p2`. La fonction retourne un entier x défini comme suit :

- $x = 0$: les octets comparés sont égaux,
- $x > 0$: deux octets différents ont été trouvés à la position la plus petite i et sont tels que `p1[i] > p2[i]`,
- $x < 0$: deux octets différents ont été trouvés à la position la plus petite `i` et sont tels que `p1[i] < p2[i]`.

`void *strcmp(const char * s1, const char * s2, size_t num)` compare au maximum les `num` premiers caractères des deux chaînes pointées par `s1` et `s2`. La fonction retourne un entier x défini comme suit :

- $x = 0$: les `num` premiers caractères des chaînes comparées sont égaux,
- $x > 0$: deux caractères différents ont été trouvés à la position la plus petite i et sont tels que `s1[i] > s2[i]`,
- $x < 0$: deux caractères différents ont été trouvés à la position la plus petite `i` et sont tels que `s1[i] < s2[i]`.

9.5 Concaténation

`char *strcat (char *dest, char *src, size_t num)` copie les `num` premiers caractères de la chaîne pointée par `src` plus le caractère de terminaison de chaîne à la suite de la chaîne pointée par `dest`. Si la chaîne pointée par `src` est plus petite que `num`, seuls les caractères précédent le caractère de terminaison de chaîne sont copiés. La fonction retourne `dest`.

9.6 Entrées/sorties formatées sur des chaînes de caractères

La fonction `int sprintf(char *str, const char *format, ...)` s'utilise de la même manière que la fonction `printf` (voir Section 8.1.1) mais requiert un argument supplémentaire `str` spécifiant une chaîne de caractères dans laquelle les données formatées seront écrites.

La fonction `int sscanf(const char *str, const char *format, ...)` s'utilise de la même manière que la fonction `scanf` (voir Section 8.1.2) mais requiert un argument supplémentaire `str` spécifiant une chaîne de caractères à partir de laquelle des données formatées seront lues.

Chapitre 10

Exemples de programme C

Ce chapitre reprend quelques exemples de programmes simples. Chaque exemple est accompagné d'un texte explicatif ainsi que de pointeurs vers les sections décrivant les outils utilisés.

10.1 Hello world

Ce programme se contente d'afficher le message « Hello world » sur la sortie standard.

Il commence par inclure (voir Section 6.1) le fichier en-tête définissant les fonctions d'entrée/sortie standard (voir Chapitre 8). Le comportement de la fonction `main` (voir Section 3.2) est constitué de 2 instructions : un appel (voir Section 4.7) à la fonction `printf` (voir Section 8.1.1) ainsi qu'un appel à l'instruction `return` (voir Chapitre 3) impliquant que le code de retour du programme est 0.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("Hello world\n");
   return 0;
}
```

10.2 Types

Ce programme définit quelques variables locales et affiche leur valeur sur la sortie standard.

Il commence par inclure (voir Section 6.1) le fichier en-tête définissant les fonctions d'entrée/sortie standard (voir Chapitre 8). Le comportement de la fonction `main` (voir Section 3.2) est constitué de 4 déclarations de variables (voir Section 3.1) suivies de 4 appels (voir Section 4.7) à la fonction `printf` (voir Section 8.1.1). Un appel à l'instruction `return` (voir Chapitre 3) impliquant que le code de retour du programme est 0.

Les 4 déclarations sont les suivantes (voir Section 2.3 pour les types et Section 2.1 pour les constantes) :
– une variable `ix` de type `int` (nombre entier) initialisée à la valeur 2.

– une variable `dx` de type `double` (nombre à virgule flottante en double précision) initialisée à la valeur 1.
– une variable `cx` de type `char` (caractère ou entier sur 8 bits) initialisée à la valeur 'c'.
– une variable `str` de type `const char *` (chaîne de caractères constante) initialisée à la valeur "string".

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int ix = 2;
   double dx = 1.;
   char cx = 'c';
   const char *str = "string";

   printf("ix=%d\n", ix);
   printf("dx=%f\n", dx);
   printf("cx=%c\n", cx);
   printf("str=%s\n", str);

   return 0;
}
```

Index

Table des matières