Syntaxe du Langage C32

Un C simplifié pour apprendre la compilation

> "Le C est le langage assembleur portable." — Dennis Ritchie

Qu'est-ce que C32 ?

C32 = sous-ensemble de C adapté pour Seed

- Syntaxe familière du C
- Compilation vers assembleur A32
- Pas de bibliothèque standard complexe

> **Pourquoi C32 ?**
>
> Assez complet pour écrire de vrais programmes, assez simple pour comprendre la compilation.

Structure d'un Programme C32

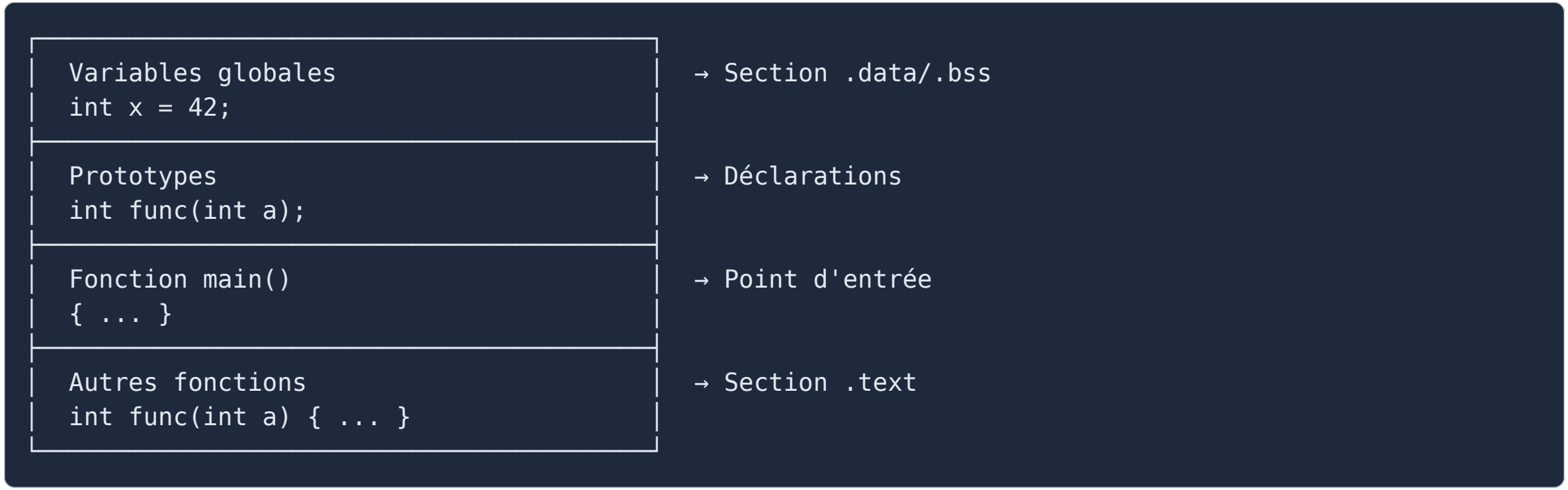
```
// Déclarations globales
int compteur;
char *message = "Hello";

// Prototypes de fonctions
int add(int a, int b);

// Fonction principale
int main() {
    int result = add(5, 3);
    return result;
}

// Définition de fonction
int add(int a, int b) {
    return a + b;
}
```

Anatomie d'un Programme



Exemple Annoté Ligne par Ligne

```
int x;                  // 1
int y = 10;             // 2

int double_val(int n) {  // 3
    return n * 2;        // 4
}                        // 5

int main() {             // 6
    x = 5;               // 7
    int z = x + y;       // 8
    z = double_val(z);   // 9
    return z;            // 10
}                        // 11
```

Ligne 1 : `int x;`

```
int x;
```

Signification :

- `int` : type entier 32 bits signé
- `x` : nom de la variable
- `;` : fin de déclaration

Pourquoi globale ?

- Déclarée hors fonction
- Accessible partout
- Stockée en `.bss` (initialisée à 0)

Variable globale non initialisée

Va dans la section .bss (Block Started by Symbol), automatiquement à zéro.

Ligne 2 : `int y = 10;`

```
int y = 10;
```

Signification :

- `int y` : déclare une variable entière
- `= 10` : initialisation
- Valeur connue à la compilation

Pourquoi .data ?

- Variable initialisée
- Valeur stockée dans l'exécutable
- Section `.data` (lecture/écriture)

Ligne 3 : `int double_val(int n) {`

```
int double_val(int n) {
```

Décomposition :

Élément	Signification
`int`	Type de retour
`double_val`	Nom de la fonction
`int n`	Paramètre entier
`{`	Début du corps

> **Convention d'appel**
>
> Le paramètre `n` arrive dans R0.

Ligne 4 : `return n * 2;`

```
    return n * 2;
```

Signification :

- `return` : termine la fonction
- `n * 2` : expression à retourner
- Résultat placé dans R0

Code généré :

```
; n est dans R0
ADD R0, R0, R0  ; ou LSL #1
BX LR           ; retour
```

Lignes 7-8 : Variables et Expressions

```
    x = 5;              // Assignation globale
    int z = x + y;      // Déclaration locale + init
```

Différences :

`x = 5`	`int z = x + y`
Variable existante	Nouvelle variable
Globale (mémoire)	Locale (pile ou registre)
Assignation	Déclaration + initialisation

Ligne 9 : `z = double_val(z);`

```
    z = double_val(z);
```

Ce qui se passe :

1. `z` est mis dans R0 (argument)
2. `BL double_val` (appel)
3. Résultat récupéré de R0
4. Stocké dans `z`

> **Convention ARM**
>
> Arguments dans R0-R3, retour dans R0.

Les Types de Données

Type	Taille	Range	Usage
`int`	32 bits	-2^{31} à $2^{31}-1$	Entiers signés
`uint`	32 bits	0 à $2^{32}-1$	Entiers non signés
`char`	8 bits	0 à 255	Caractères ASCII
`bool`	8 bits	0 ou 1	Booléens
`void`	0	—	Pas de valeur
`T*`	32 bits	Adresse	Pointeurs

Pourquoi ces Types ?

int (32 bits) :

- Taille native du CPU
- Registres 32 bits
- Calculs efficaces

char (8 bits) :

- ASCII standard
- Économie mémoire
- Manipulation texte

Pointeurs (32 bits) :

- Adresses 32 bits
- Même taille que int
- Arithmétique possible

> **Pas de float !**
>
> C32 n'a pas de virgule flottante (pas de FPU).

Déclaration de Variables

```
// Syntaxe générale
type nom;                    // Non initialisée
type nom = valeur;           // Initialisée
type a, b, c;                // Multiples

// Exemples
int compteur;
int x = 42;
char c = 'A';
int *ptr = &x;
int tableau[10];
```

Portée des Variables

```
int globale = 1;                // Globale : partout

int func() {
    int locale = 2;             // Locale : dans func seulement

    if (1) {
        int bloc = 3;           // Bloc : dans le if seulement
    }
    // bloc n'existe plus ici

    return locale;
}
// locale n'existe plus ici
```

Pourquoi la Portée ?

Globales :

- En mémoire (.data/.bss)
- Toujours accessibles
- Risque de conflits de noms

Locales :

- Sur la pile (ou registres)
- Libérées automatiquement
- Isolation entre fonctions

Règle

Préférer les variables locales. Les globales sont pour l'état partagé.

Opérateurs Arithmétiques

Opérateur	Signification	Exemple
`+`	Addition	`a + b`
`-`	Soustraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division entière	`a / b`
`%`	Modulo	`a % b`

```c
int sum = 10 + 5;       // 15
int diff = 10 - 3;      // 7
int prod = 4 * 5;       // 20
int quot = 17 / 5;      // 3 (pas 3.4 !)
int rest = 17 % 5;      // 2
```

Opérateurs de Comparaison

Opérateur	Signification	Résultat
`==`	Égal	0 ou 1
`!=`	Différent	0 ou 1
`<`	Inférieur	0 ou 1
`>`	Supérieur	0 ou 1
`<=`	Inférieur ou égal	0 ou 1
`>=`	Supérieur ou égal	0 ou 1

Attention

`==` (comparaison) ≠ `=` (assignation) !

Opérateurs Logiques

Opérateur	Signification	Court-circuit
`&&`	ET logique	Oui
`||`	OU logique	Oui
`!`	NON logique	—

```
if (x > 0 && x < 10) { ... }    // Les deux vraies
if (x == 0 || y == 0) { ... }   // Au moins une vraie
if (!found) { ... }              // Inverse
```

Court-circuit : Si le résultat est connu, le reste n'est pas évalué.

Pourquoi le Court-circuit ?

```
// Sécurité : évite le déréférencement null
if (ptr != 0 && *ptr > 0) {
    // *ptr n'est évalué que si ptr != 0
}

// Performance : évite calculs inutiles
if (cache_hit || compute_expensive()) {
    // compute_expensive() non appelé si cache_hit
}
```

Opérateurs Bit à Bit

Opérateur	Signification	Exemple
`&`	ET bit à bit	`0xF0 & 0x0F` → `0x00`
`|`	OU bit à bit	`0xF0 | 0x0F` → `0xFF`
`^`	XOR bit à bit	`0xFF ^ 0x0F` → `0xF0`
`~`	Inversion	`~0x00` → `0xFFFFFFFF`
`<<`	Décalage gauche	`1 << 4` → `16`
`>>`	Décalage droite	`16 >> 2` → `4`

Usage des Opérateurs Bit à Bit

```c
// Masquage : extraire des bits
int low_byte = value & 0xFF;

// Setting : mettre un bit à 1
flags = flags | (1 << 3);

// Clearing : mettre un bit à 0
flags = flags & ~(1 << 3);

// Toggling : inverser un bit
flags = flags ^ (1 << 3);

// Multiplication/division par 2^n
int doubled = x << 1;    // x * 2
int halved = x >> 1;     // x / 2
```

Précédence des Opérateurs

```
1.  ()          Parenthèses (plus haute)
2.  ! ~ -       Unaires (droite → gauche)
3.  * / %       Multiplicatifs
4.  + -         Additifs
5.  << >>       Décalages
6.  < <= > >=   Relationnels
7.  == !=       Égalité
8.  &           ET bit à bit
9.  ^           XOR bit à bit
10. |           OU bit à bit
11. &&          ET logique
12. ||          OU logique
13. =           Assignation (plus basse)
```

Pourquoi Connaître la Précédence ?

```c
// Sans parenthèses - DANGER !
int result = a + b * c;      // a + (b * c), pas (a + b) * c

// Piège classique
if (flags & MASK == 0) { }   // flags & (MASK == 0) !
if ((flags & MASK) == 0) { } // Correct

// Recommandation
int result = a + (b * c);    // Explicite = clair
```

Conseil

En cas de doute, utilisez des parenthèses.

Structure if / else

```
if (condition) {
    // Exécuté si condition vraie (≠ 0)
}

if (condition) {
    // Si vrai
} else {
    // Si faux
}

if (cond1) {
    // Si cond1
} else if (cond2) {
    // Si cond2
} else {
    // Sinon
}
```

Boucle while

```
while (condition) {
    // Corps exécuté tant que condition vraie
}

// Exemple : somme de 1 à n
int sum = 0;
int i = 1;
while (i <= n) {
    sum = sum + i;
    i = i + 1;
}
```

Pas de i++ !

C32 n'a pas d'opérateur `++`. Utilisez `i = i + 1`.

Boucle for

```
for (init; condition; increment) {
    // Corps
}

// Exemple
for (int i = 0; i < 10; i = i + 1) {
    sum = sum + i;
}

// Équivalent while
int i = 0;
while (i < 10) {
    sum = sum + i;
    i = i + 1;
}
```

Boucle do-while

```c
do {
    // Corps exécuté au moins une fois
} while (condition);

// Exemple : lire jusqu'à 'q'
char c;
do {
    c = getchar();
    process(c);
} while (c != 'q');
```

Différence avec while : Le corps est exécuté avant le test.

break et continue

```
// break : sort de la boucle
while (1) {
    if (done) break;    // Sort immédiatement
    process();
}

// continue : passe à l'itération suivante
for (int i = 0; i < 10; i = i + 1) {
    if (skip[i]) continue;  // Saute cette itération
    process(i);
}
```

Fonctions

```c
// Déclaration (prototype)
int add(int a, int b);

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

// Fonction sans retour
void greet() {
    print("Hello");
}

// Fonction sans paramètre
int get_count() {
    return counter;
}
```

Convention d'Appel

Arguments :

- R0 : 1er argument
- R1 : 2ème argument
- R2 : 3ème argument
- R3 : 4ème argument
- Pile : arguments suivants

Retour :

- R0 : valeur de retour

Sauvegarde :

- R4-R11 : callee-saved
- R0-R3 : caller-saved

Pourquoi cette Convention ?

```c
int func(int a, int b, int c, int d, int e) {
    return a + b + c + d + e;
}
```

```
; Arguments: R0=a, R1=b, R2=c, R3=d, [SP]=e
func:
    LDR R4, [SP]        ; e depuis la pile
    ADD R0, R0, R1      ; a + b
    ADD R0, R0, R2      ; + c
    ADD R0, R0, R3      ; + d
    ADD R0, R0, R4      ; + e
    BX LR               ; retour dans R0
```

Pointeurs

```c
int x = 42;
int *p = &x;     // p contient l'adresse de x

// Déréférencement
int val = *p;    // val = 42 (valeur pointée)
*p = 100;        // x devient 100

// Pointeur null
int *ptr = 0;    // Pointeur invalide
if (ptr != 0) {
    *ptr = 5;    // Sûr
}
```

Pourquoi les Pointeurs ?

1. Modification indirecte

```c
void increment(int *p) {
    *p = *p + 1;
}
int x = 5;
increment(&x);  // x = 6
```

2. Structures de données

```c
struct Node {
    int val;
    struct Node *next;
};
```

3. Tableaux

```c
int arr[10];
int *p = arr;  // Équivalent
```

Arithmétique des Pointeurs

```c
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;       // p pointe sur arr[0]

p = p + 1;          // p pointe sur arr[1]
                    // Avance de sizeof(int) = 4 octets

int val = *(p + 2); // arr[3] = 40

// Équivalences
arr[i]   ≡  *(arr + i)
&arr[i]  ≡  arr + i
p[i]     ≡  *(p + i)
```

Tableaux

```
// Déclaration
int arr[10];              // 10 entiers
char buffer[256];         // 256 caractères

// Accès
arr[0] = 42;              // Premier élément
arr[9] = 99;              // Dernier élément

// Initialisation manuelle
for (int i = 0; i < 10; i = i + 1) {
    arr[i] = i * 2;
}
```

Pas de vérification !

C32 ne vérifie pas les bornes. `arr[100]` compile mais plante.

Tableaux et Fonctions

```c
// Les tableaux sont passés par adresse
int sum(int *arr, int len) {
    int s = 0;
    for (int i = 0; i < len; i = i + 1) {
        s = s + arr[i];
    }
    return s;
}

int main() {
    int numbers[5];
    // ... remplir
    int total = sum(numbers, 5);
}
```

Chaînes de Caractères

```c
// Littéral de chaîne (read-only, en .rodata)
char *msg = "Hello";

// Parcours
char *s = "Hello";
while (*s != '\0') {
    putchar(*s);
    s = s + 1;
}

// Longueur manuelle
int len = 0;
char *p = "test";
while (*p != '\0') {
    len = len + 1;
    p = p + 1;
}
```

Structures

```c
// Définition
struct Point {
    int x;
    int y;
};

// Utilisation
struct Point p;
p.x = 10;
p.y = 20;

// Via pointeur
struct Point *ptr = &p;
ptr->x = 30;            // Équivalent à (*ptr).x
```

Pourquoi les Structures ?

Grouper des données liées :

```
struct Player {
    int health;
    int x;
    int y;
    char name[16];
};
```

Organisation mémoire :

```
Offset 0:  health (4 bytes)
Offset 4:  x (4 bytes)
Offset 8:  y (4 bytes)
Offset 12: name (16 bytes)
Total: 28 bytes
```

Cast de Types

```
// Entier vers pointeur
int *p = (int*)0x1000;

// Pointeur vers entier
int addr = (int)p;

// Entre types entiers
int x = 1000;
char c = (char)x;     // Troncature (232)

// Signé vers non signé
int neg = -1;
uint pos = (uint)neg;  // 0xFFFFFFFF
```

Entrées/Sorties (MMIO)

```c
// Sortie caractère
void putchar(int c) {
    int *port = (int*)0xFFFF0000;
    *port = c;
}

// Entrée caractère
int getchar() {
    int *port = (int*)0xFFFF0004;
    return *port;  // -1 si pas de caractère
}

// Sortie programme
void exit(int code) {
    int *port = (int*)0xFFFF0010;
    *port = code;
}
```

Affichage de Texte

```
void print(char *s) {
    while (*s != '\0') {
        putchar(*s);
        s = s + 1;
    }
}

void println(char *s) {
    print(s);
    putchar(10);    // Newline
}

int main() {
    println("Hello, World!");
    return 0;
}
```

Affichage d'Entiers

```
void print_int(int n) {
    char buffer[12];
    int i = 0;

    if (n == 0) { putchar('0'); return; }

    int neg = 0;
    if (n < 0) { neg = 1; n = 0 - n; }

    while (n > 0) {
        buffer[i] = '0' + (n % 10);
        n = n / 10;
        i = i + 1;
    }

    if (neg) putchar('-');
    while (i > 0) { i = i - 1; putchar(buffer[i]); }
}
```

Limitations de C32

Non supporté :

- `float`, `double`
- `++`, `--`
- `?:` (ternaire)
- `enum`
- Préprocesseur complet
- `malloc` / `free`

Utilisez plutôt :

- Entiers uniquement
- `i = i + 1`
- `if/else`
- Constantes `#define`
- Allocation statique
- Gestion manuelle

Programme Complet : Fibonacci

```c
int fib(int n) {
    if (n <= 1) return n;

    int a = 0;
    int b = 1;
    int i = 2;
    while (i <= n) {
        int temp = a + b;
        a = b;
        b = temp;
        i = i + 1;
    }
    return b;
}

int main() {
    int result = fib(10);  // 55
    return result;
}
```

Compilation C32 → Assembleur

Utilisation CLI

```
# Compiler
cargo run -p c32_cli -- compile prog.c -o prog.s

# Compiler + Assembler
cargo run -p c32_cli -- build prog.c -o prog.bin

# Exécuter directement
cargo run -p c32_cli -- run prog.c
```

Questions de Réflexion

1. Pourquoi C32 n'a pas de `float` ?
2. Quelle est la différence entre `*p` et `&x` ?
3. Pourquoi les tableaux décroissent vers le bas en mémoire ?
4. Comment implémenter `malloc` sans OS ?
5. Pourquoi la convention met le retour dans R0 ?

Ce qu'il faut retenir

1. **Types** : `int`, `uint`, `char`, `bool`, pointeurs
2. **Pas de `++`** : utiliser `i = i + 1`
3. **Pointeurs** : `*` déréférence, `&` adresse
4. **Structures** : `.` direct, `->` via pointeur
5. **Fonctions** : R0-R3 arguments, R0 retour
6. **MMIO** : Entrées/sorties par adresses mémoire

Questions ?

Référence : `/book/references/carte_c32.md`

Exemples : `/demos/*.c`

Prochain : OS et gestion mémoire