✅ Prérequis & Rappels — Langage C

class: lead

Résumé concis des notions essentielles pour le cours

📝 Changelog — V0.0.2

- 01/02/2026 18:03 — Corrections et améliorations : ajout de diagrammes explicatifs, ajustement de leur taille pour les slides, ajout des étapes d'installation pour GDB/Valgrind, compléments pour `Makefile`.
- Mise à jour de la version des decks affectés : `compiler-first-c-debian12.md`, `prerequis-rappels-c.md`, `seance1-outillage-qualite.md`.

Types & Qualificateurs

- Types entiers: `char`, `short`, `int`, `long`, `long long` (signed / unsigned)
- Virgule flottante: `float`, `double`
- Booléen: `_Bool` (C99)
- Qualificateurs: `const`, `volatile`, `static`, `extern`
- Taille dépend du compilateur — préférez `sizeof` et types fixes (`stdint.h`)

Contrôle & Structures

- `if` / `else`, `switch`, `for`, `while`, `do - while`
- Toujours utiliser `{}` pour éviter les erreurs d'ambiguïté
- `enum` pour états symboliques, `struct` pour regrouper des données

Fonctions & Headers

- Prototypes dans `.h`, définitions dans `.c`
- Include guards: `#ifndef / #define / #endif` ou `#pragma once`
- `static` pour linkage interne, `inline` pour optimisation
- Favoriser signatures claires et responsabilité unique

Pointeurs & Tableaux

- Tableau décaye en pointeur; conservez la longueur séparément
- Notation: `int *p`, `char s[]`, `const char *msg`

```c
int a[10];
int *p = a; // p == &a[0]
```

- Attention à l'arithmétique de pointeurs et à l'aliasing

Tableau → pointeur : explication simple

- En C, la plupart du temps un *tableau* utilisé dans une expression est automatiquement converti ("décaye") en pointeur vers son premier élément. Ainsi, l'information sur la **taille du tableau** n'est pas propagée.

```c
int a[10];
int *p = a; // équivalent à &a[0]
printf("sizeof(a) = %zu, sizeof(p) = %zu\n", sizeof(a), sizeof(p));
// sizeof(a) == 10 * sizeof(int)  (si utilisé dans le même scope)
// sizeof(p) == taille d'un pointeur (ex. 8)
```

- Conséquence : dans une fonction `void f(int *p)`, on **ne connaît pas** la longueur de `p` → passez toujours la `size_t n` en argument.
- Exceptions (pas de "décay") : `sizeof(a)`, l'opérateur `&a` (donne un `int (*)[N]`) et lors de l'initialisation d'un tableau.
- Mini-exo : que retourne `sizeof(a)/sizeof a[0]` vs `sizeof(p)/sizeof p[0]` ?

Allocation dynamique

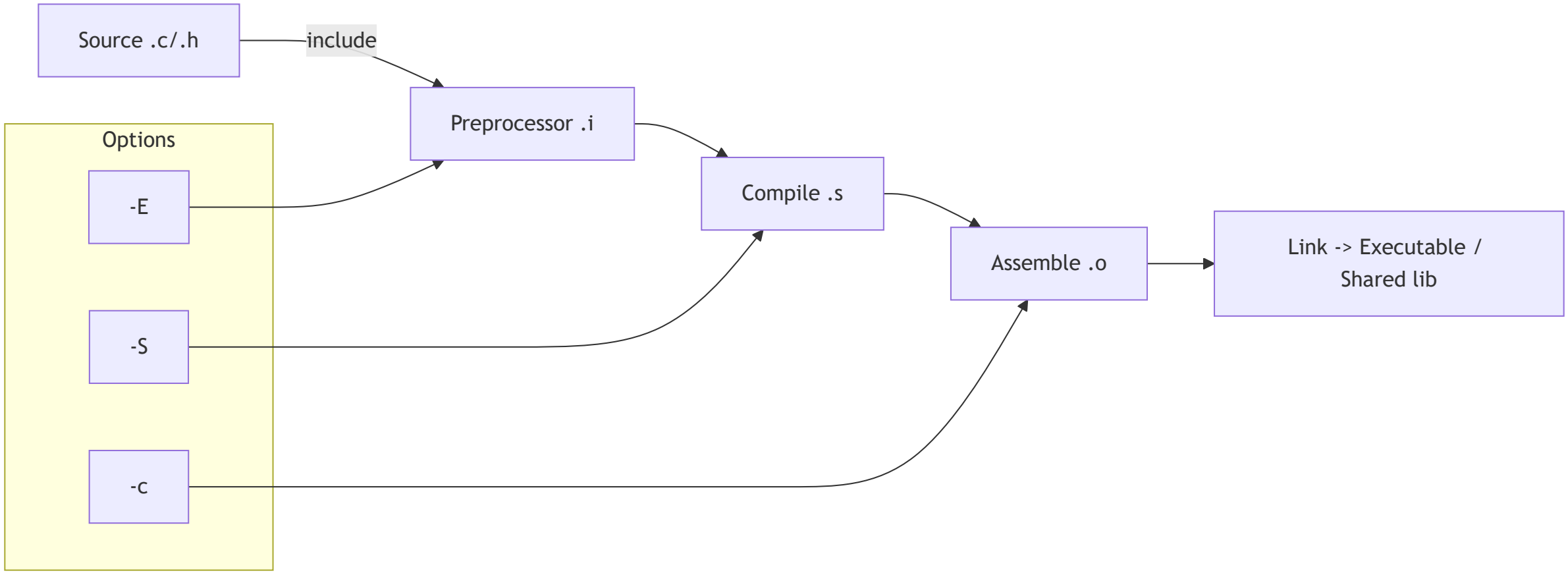
```c
int *arr = malloc(n * sizeof *arr);
if (!arr) return NULL; // gérer l'erreur
// ...
free(arr);
```

- Toujours vérifier le retour de `malloc` / `calloc`
- `free()` correspond à chaque `malloc` / `calloc` / `realloc`

Compilation & Linking

- Étapes: préprocesseur → compilation → assemblage → linking
- Découplez interface (`.h`) et impl. (`.c`)
- Flags utiles: `-Wall -Wextra -Werror -Wconversion -g -O2`

Pipeline de compilation (diagramme)



Undefined Behavior & Pièges courants

- Variables non initialisées, sorties hors-borne, use-after-free
- Débordement d'entiers signés = UB
- Mismatched `printf` format specifiers ⇒ crash / datas corruption

Bonnes pratiques & Outils

- Activez `-Wall -Wextra -Werror`, utilisez ASan/UBSan, Valgrind
- Tests unitaires, revue de code, CI/CD
- Documentez et écrivez des commits clairs

Checklist rapide (avant push)

- Code compile sans warnings
- Tests unitaires passent
- Pas de fuites détectées par Valgrind / ASan
- Coverage raisonnable pour les fonctions critiques

Exemples rapides / Mini-exercice

1. Écrire une fonction `int sum(const int *a, size_t n)` qui renvoie la somme d'un tableau.
2. Testez avec cas vide (`n==0`), overflow possible (attention au type)

Pointeurs avancés

- Pointeur vers pointeur, `const`-correctness et ownership

```c
/* const correctness */
void foo(const int *p); // ne modifie pas *p
void bar(int * const p); // p ne change pas, *p peut changer

/* Pointeur vers pointeur : créer une chaîne dupliquée */
char *strdup_safe(const char *s) {
    if (!s) return NULL;
    size_t n = strlen(s) + 1;
    char *d = malloc(n);
    if (!d) return NULL;
    memcpy(d, s, n);
    return d; // ownership transferred to caller
}
```

Allocation sécurisée & `realloc`

- Ne pas perdre le pointeur original si `realloc` échoue

```c
void *safe_realloc(void *ptr, size_t new_size) {
    void *tmp = realloc(ptr, new_size);
    if (!tmp) {
        // realloc failed, original ptr still valid
        return NULL;
    }
    return tmp;
}

/* Utilisation */
int *arr = malloc(n * sizeof *arr);
int *tmp = safe_realloc(arr, (n2) * sizeof *arr);
if (!tmp) {
    free(arr); // gérer l'erreur
} else {
    arr = tmp;
}
```

Gestion d'erreurs (pattern `goto cleanup`)

- Utiliser un point de sortie unique pour libérer les ressources

```c
int do_work(void) {
    int ret = -1;
    char *buf = NULL;
    FILE *f = NULL;

    buf = malloc(1024);
    if (!buf) goto out;

    f = fopen("data.bin", "rb");
    if (!f) goto out;

    // ... traiter
    ret = 0; // succès

out:
    if (f) fclose(f);
    free(buf);
    return ret;
}
```

Chaînes & sécurité

- Préférer `snprintf`, `strnlen`, `memmove` à `strcpy/strcat`

```c
char dst[32];
int n = snprintf(dst, sizeof dst, "%s %d", name, value);
if (n < 0 || (size_t)n >= sizeof dst) {
    // erreur ou troncature
}

/* memmove pour zones qui se chevauchent */
memmove(dst + 2, dst, 10);
```

Exemple : somme sûre (détection d'overflow)

- Accumuler dans un type plus large et vérifier

```c
#include <stdint.h>
#include <limits.h>

bool sum_safe(const int *a, size_t n, int *out) {
    if (!a || !out) return false;
    int64_t acc = 0;
    for (size_t i = 0; i < n; ++i) {
        acc += (int64_t)a[i];
        if (acc > INT_MAX || acc < INT_MIN) return false; // overflow
    }
    *out = (int)acc;
    return true;
}
```

Tests rapides (assert / Unity)

- Exemple minimal avec `assert`

```c
#include <assert.h>

void test_sum(void) {
    int a[] = {1,2,3};
    int out;
    assert(sum_safe(a, 3, &out) && out == 6);
}

int main(void) { test_sum(); return 0; }
```

Besoin d'exemples détaillés ?

Propose si tu veux des slides supplémentaires : pointeurs avancés, gestion d'erreurs, patterns mémoire.