Base III

Programmation séquentielle en C, 2020-2021

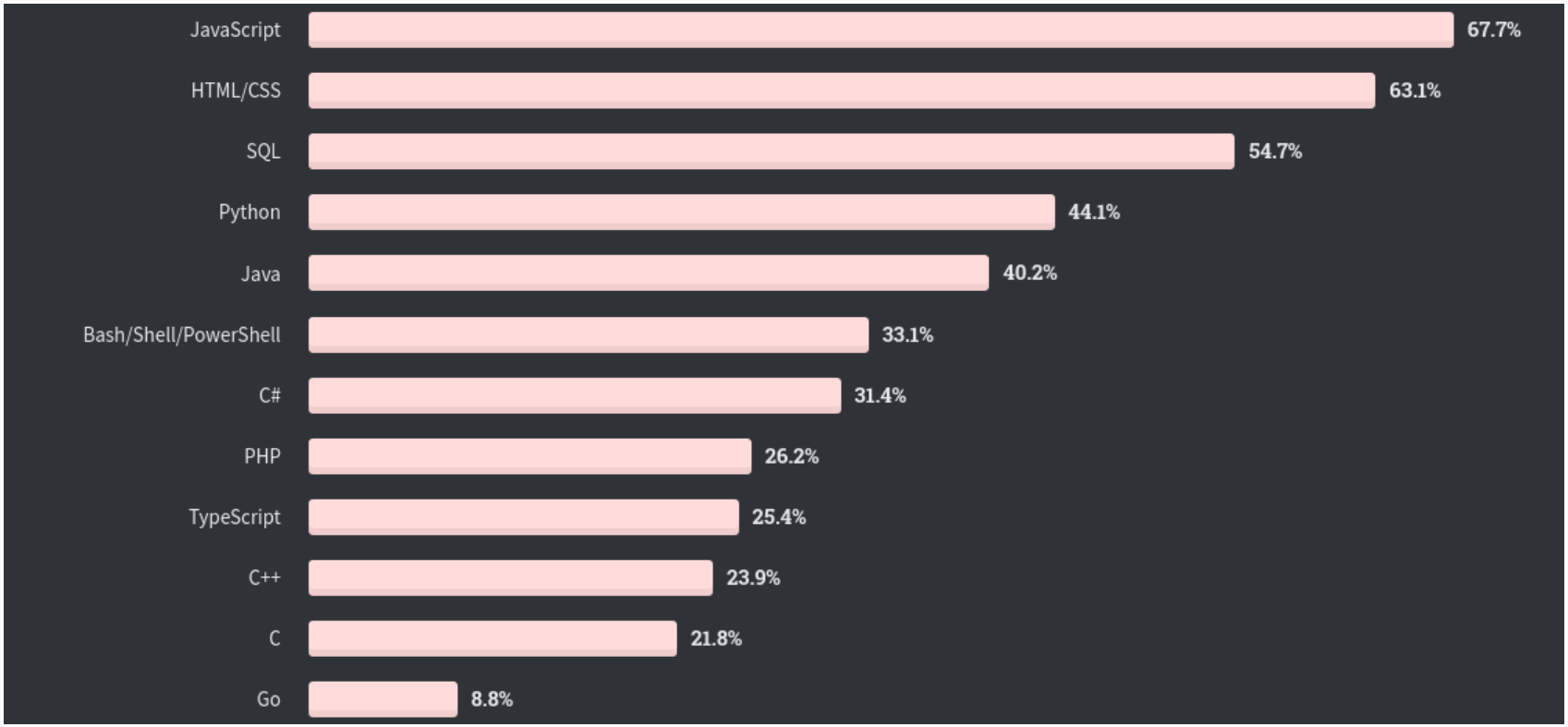
Orestis Malaspinas (A401), ISC, HEPIA

2020-09-30

Inspirés des slides de F. Glück

Le C sur stackoverflow

Le C est-il un langage encore utilisé (ou juste une vieille croûte inutile)?



Types complexes: `struct` (1/5)

Généralités

- Plusieurs variables qu'on aimerait regrouper dans un seul type: `struct`.

```
struct complex { // déclaration
    double re;
    double im;
};

struct complex num; // déclaration de num
```

- Les champs sont accessible avec le sélecteur " . ".

```
num.re = 1.0;
num.im = -2.0;
```

Types complexes: `struct` (2/5)

Simplifications

- `typedef` permet de définir un nouveau type.

```
typedef unsinged int uint;
typedef struct complex complex_t;
typedef struct complex {
    double re, im;
} complex_t;
```

- L'initialisation peut aussi se faire avec

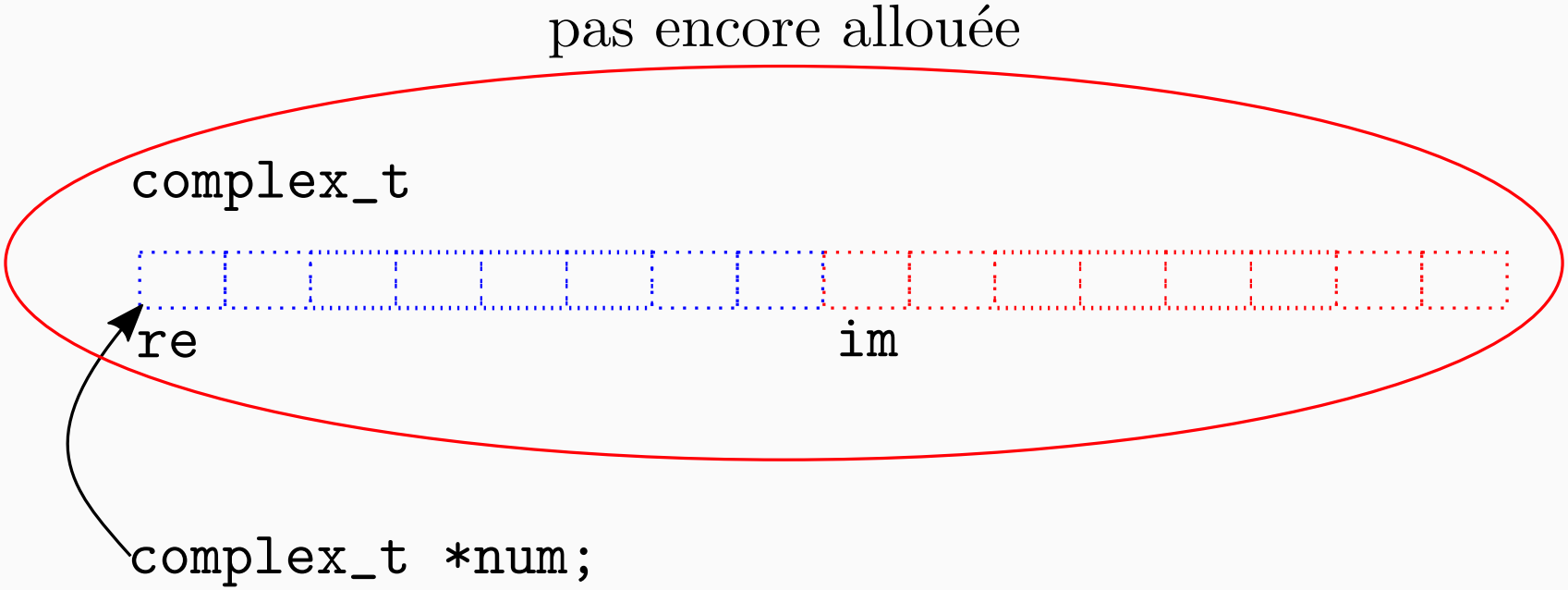
```
complex_t num = {1.0, -2.0}; // re = 1.0, im = -2.0
complex_t num = {.im = 1.0, .re = -2.0};
complex_t num = {.im = 1.0}; // argl! .re non initialisé
complex_t num2 = num; // copie
```

Types complexes: `struct` (3/5)

Pointeurs

- Comme pour tout type, on peut avoir des pointeurs vers un **`struct`**.
- Les champs sont accessible avec le sélecteur `->`

```
complex_t *num; // on crée un pointeur
num->re = 1.0;  // seg fault...
num->im = -1.0; // mémoire pas allouée.
```



Types complexes: `struct` (4/5)

Initialisation

- Avec le passage par **référence** on peut modifier un struct *en place*.
- Les champs sont accessible avec le sélecteur ->

```
void complex_init(complex_t *num,
                  double re, double im)
{
    // num a déjà été allouée
    num->re = re;
    num->im = im;
}
int main() {
    complex_t num; // on alloue un complexe
    complex_init(&num, 2.0, -1.0); // on l'initialise
}
```

Types complexes: `struct` (5/5)

Initialisation version copie

- On peut allouer un complexe, l'initialiser et le retourner.
- La valeur retournée peut être copiée dans une nouvelle structure.

```
complex_t complex_create(double re, double im) {
    complex_t num;
    num.re = re;
    num.im = im;
    return num;
}
int main() {
    // on crée un complexe et on l'initialise
    // en copiant le complexe créé par complex_create
    // deux allocation et une copie
    complex_t num = complex_create(2.0, -1.0);
}
```

Les fonctions (1/2)

Arguments de fonctions par copie

- Les arguments d'une fonction sont toujours passés **par copie**.
- Les arguments d'une fonction ne peuvent **jamais** être modifiés.

```
void do_something(complex_t a) { // a: nouvelle variable
    // valeur de a est une copie de x
    // lorsque la fonction est appelée, ici -1
    a.re += 2.0;
    a.im -= 2.0;
} // a est détruite

int main() {
    complex_t x;
    do_something(x); // x est passé en argument
    // x sera inchangé
}
```

- Que pourrait-on faire pour modifier `x`?

Les fonctions (2/2)

Arguments de fonctions par référence

- Pour modifier une variable, il faut passer son **adresse mémoire** (sa référence) en argument.
- L'adresse d'une variable, `a`, est accédée par `&a`.
- Un **pointeur** vers une variable entière a le type, `int *`.
- `*a` sert à **déréférencer** le pointeur (accéder la mémoire pointée).

```
void do_something(complex_t *a) {
    // a: un nouveau pointeur
    // valeur de a est une copie de du pointeur
    // passé en argument, mais
    // les données pointées sont les données originales
    a->re += 2.0;
    a->im -= 2.0;
} // le pointeur a est détruit,
  // *a est toujours là et a été modifié
```

Prototypes de fonctions (1/3)

Principes généraux de programmation

- Beaucoup de fonctionnalités dans un code $\Rightarrow$ Modularisation.
- Modularisation du code $\Rightarrow$ écriture de fonctions.
- Beaucoup de fonctions $\Rightarrow$ regrouper les fonctions dans des fichiers séparés.

Mais pourquoi?

- Lisibilité.
- Raisonnement sur le code.
- Débogage.

Exemple

- Libraire `stdio.h`: `printf()`, `scanf()`, `...`

Prototypes de fonctions (2/3)

- Prototypes de fonctions nécessaires quand:
  1. Utilisation de fonctions dans des fichiers séparés.
  2. Utilisation de librairies.
- Un prototype indique au compilateur la signature d'une fonction.
- On met les prototypes des fonctions **publiques** dans des fichiers *headers*, extension `.h`.
- Les *implémentations* des fonctions vont dans des fichier `.c`.

Prototypes de fonctions (3/3)

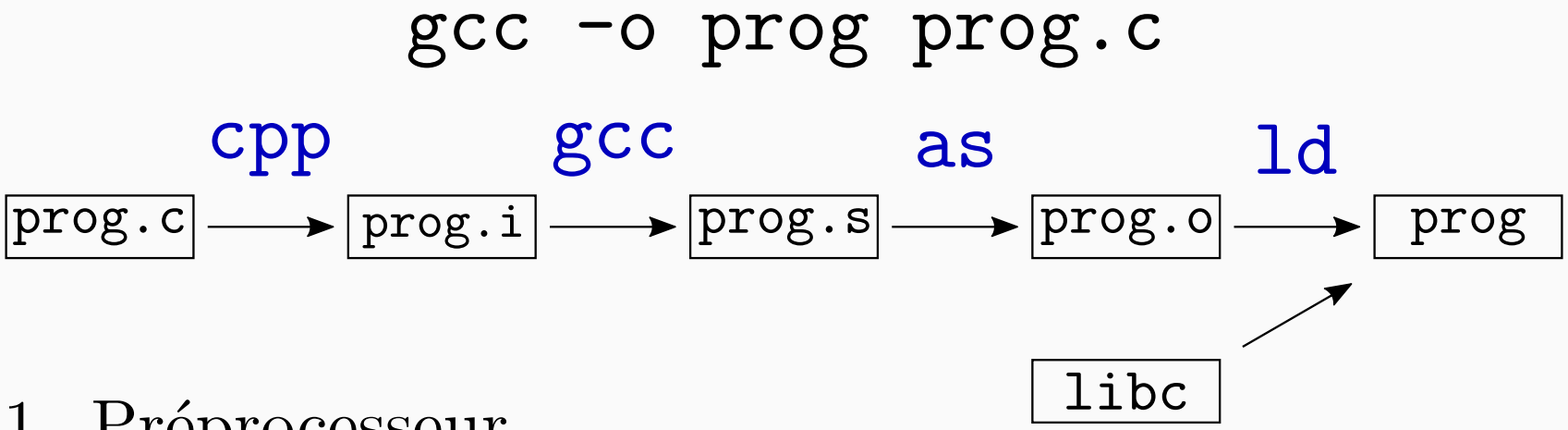
Fichier header

- Porte l'extension `.h`
- Contient:
  - définitions des types
  - prototypes de fonctions
  - macros
  - directives préprocesseur (cf. plus loin)
- Utilisé pour décrire **l'interface** d'une librairie ou d'un module.
- Un fichier `C` (extension `.c`) utilise un header en *l'important* avec la directive *`#include`*:

```
#include <stdio.h> // libraire dans LD_LIBRARY_PATH
#include "chemin/du/prototypes.h"// chemin explicite
```

Génération d'un exécutable (1/5)

Un seul fichier source



1. Préprocesseur
2. Compilation assembleur
3. Compilation code objet
4. Édition des liens

Figure 2 – Étapes de génération.

Génération d'un exécutable (2/5)

Un seul fichier source

```
gcc proc.c -o prog
```

1. **Précompilation:** `gcc` appelle `cpp`, le préprocesseur qui effectue de la substitution de texte (`#define`, `#include`, macros, `...`) et génère le code `C` à compiler, portant l'extension `.i` (`prog.i`).
2. **Compilation assembleur:** `gcc` compile le code C en code assembleur, portant l'extension `.s` (`prog.s`).
3. **Compilation code objet:** `gcc` appelle `as`, l'assembleur, qui compile le code assembleur en code machine (code objet) portant l'extension `.o` (`prog.o`).
4. **Édition des liens:** `gcc` appelle `ld`, l'éditeur de liens, qui lie le code objet avec les librairies et d'autres codes objet pour produire l'exécutable final (`prog`).

Les différents codes intermédiaires sont effacés.

Génération d'un exécutable (3/5)

Plusieurs fichiers sources

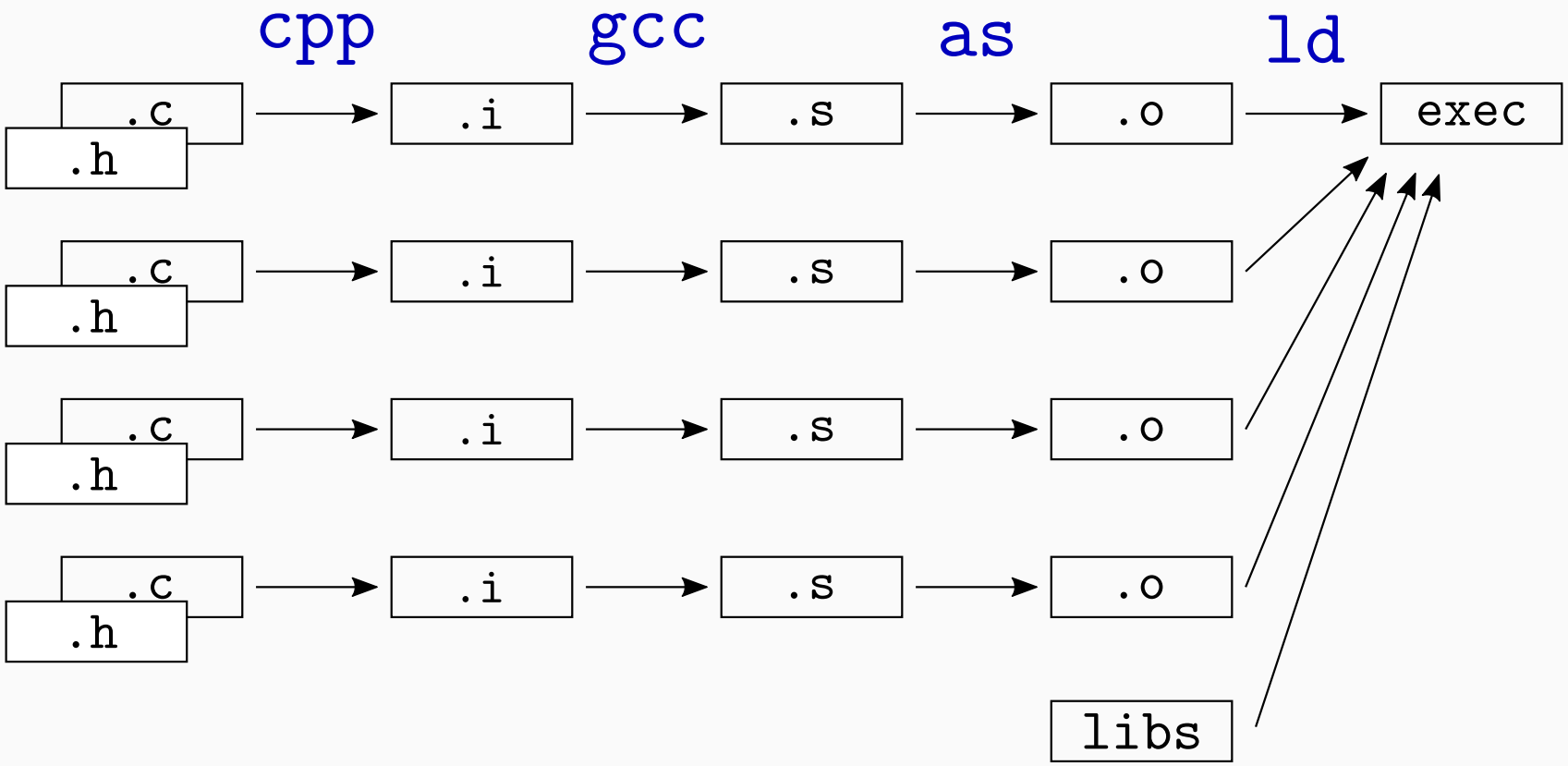


Figure 3 – Étapes de génération, plusieurs fichiers.

Génération d'un exécutable (4/5)

main.c

```c
#include <stdio.h>
#include "sum.h"
int main() {
  int tab[] = {1, 2, 3, 4};
  printf("sum: %d\n", sum(tab, 4));
  return 0;
}
```

sum.h

```c
#ifndef _SUM_H_
#define _SUM_H_

int sum(int tab[], int n);

#endif
```

sum.c

```c
#include "sum.h"
int sum(int tab[], int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s += tab[i];
  }
  return s;
}
```

Génération d'un exécutable (5/5)

La compilation séparée se fait en plusieurs étapes.

Compilation séparée

1. Générer séparément les fichiers `.o` avec l'option `-c`.
2. Éditer les liens avec l'option `-o` pour générer l'exécutable.

Exemple

- Création des fichiers objets, `main.o` et `sum.o`

```
$ gcc -Wall -Wextra -std=c11 -c main.c
$ gcc -Wall -Wextra -std=c11 -c sum.c
```

- Édition des liens

```
$ gcc main.o sum.o -o prog
```

Préprocesseur (1/2)

Généralités

- Première étape de la chaîne de compilation.
- Géré automatiquement par `gcc` ou `clang`.
- Lit et interprète certaines directives:
  1. Les commentaires (`//` et `/* ... */`).
  2. Les commandes commençant par `#`.
- Le préprocesseur ne compile rien, mais subtitue uniquement du texte.

La directive `define`

- Permet de définir un symbole:

```
#define PI 3.14159
#define _SUM_H_
```

- Permet de définir une macro.

```
#define NOM_MACRO(arg1, arg2, ...) [code]
```

Préprocesseur (2/2)

La directive `include`

- Permet d'inclure un fichier.
- Le contenu du fichier est ajouté à l'endroit du *#include*.
- Inclusion de fichiers "globaux" ou "locaux"

```
#include <file.h>       // LD_LIBRARY_PATH
#include "other_file.h" // local path
```

- Les inclusions multiples peuvent poser problème: définitions multiples. Les headers commencent par:

```
#ifndef _VAR_
#define _VAR_
/*
commentaires
*/
#endif
```

Les tableaux (1/2)

Généralités

- `C` offre uniquement des tableaux statiques
  - Un tableau est un "bloc" de mémoire contiguë associé à un nom
    - taille fixe déterminée à la déclaration du tableau
    - la taille ne peut pas être changée.
  - Pas d'assignation de tableaux.
  - Un tableau déclaré dans une fonction ou un bloc est détruit à la sortie de celle/celui-ci
    - $\Rightarrow$ Un tableau local à une fonction ne doit **jamais être retourné** (aussi valable pour toute variable locale)!
- Les éléments d'un tableau sont accédés avec `[i]` où `i` est l'index de l'élément.
- Le premier élément du tableau à l'index `0`!
- Lorsqu'un tableau est déclaré, la taille de celui-ci doit toujours être spécifiée, sauf s'il est initialisé lors de sa déclaration.

Les tableaux (2/2)

Exemple

```
float tab1[5]; // tableau de floats à 5 éléments
               // ses valeurs sont indéfinies

int tab2[] = {1, 2, 3}; // tableau de 3 entiers,
                        // taille inférée

int val = tab2[1]; // val vaut 2 à présent

int w = tab1[5]; // index hors des limites du tableau
                 // comportement indéfini!
                 // pas d'erreur du compilateur
```

La ligne de commande (1/4)

Point d'entrée d'un programme

- Le point d'entrée est la fonction `main()`.
- Elle peut être déclarée de 4 façon différentes:
  1. `void main()`.
  2. `int main()`.
  3. `void main(int argc, char **argv)`.
  4. `int main(int argc, char **argv)`.
- `argc` est le nombre d'arguments passés à la ligne de commande: **le premier est celui du programme lui-même**.
- `argv` est un tableau de chaînes de caractères passés sur la ligne de commande.

La ligne de commande (2/4)

Exemple d'utilisation

Pour la fonction dans le programme `prog`

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

Pour l'exécution suivante on a

```
$ ./prog -b 50 file.txt
```

```
argc == 4
argv[0] == "prog"
argv[1] == "-b"
argv[2] == "50"
argv[3] == "file.txt"
```

La ligne de commande (3/4)

Conversion des arguments

- Les arguments sont toujours stockés comme des **chaînes de caractère**.
- Peu pratique si on veut manipuler des valeurs numériques.
- Fonctions pour faire des conversions:

```
int atoi(const char *nptr);
long atol(const char *nptr);
long long atoll(const char *nptr);
double atof(const char *nptr);
int snprintf(char *str, size_t size,
             const char *format, ...);
// str: buffer, size: taille en octets max à copier,
// format: cf printf(), ret: nombre de char lus
```

La ligne de commande (4/4)

Exemple d'utilisation

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

int main(int argc, char **argv) {
    if (argc != 3) {
        char *progname = basename(argv[0]);
        fprintf(stderr, "usage: %s name age\n", progname);
        return EXIT_FAILURE;
    }

    char *name = argv[1];
    int age = atoi(argv[2]);

    printf("Hello %s, you are %d years old.\n", name, age);
    return EXIT_SUCCESS;
}
```

```
$ ./prog Paul 29
Hello Paul, you are 29 years old.
```