Programmation « orientée système »
Langage C – Compilation (1/2)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- passer des arguments à son programme
- directives de précompilation

Jean-Cédric Chappelier

Mon programme dans son environnement

Votre programme est exécuté par la machine dans un **environnement** :

interpréteur de commandes / *système d'exploitation*.

Il peut donc interagir avec eux (cf par exemple les flots).

Votre programme est un *processus* du système, une sorte de « fonction » du système.

En fait, `main()` **est une fonction** presque comme les autres.
Elle a juste les spécificités :

- de toujours être appelée en premier ;
- de n'avoir que deux prototypes possibles.

C'est donc à elle de gérer les interactions avec l'environnement d'appel.

Jean-Cédric Chappelier

Valeur de retour de `main()`

Que signifie cet entier retourné par `main()` ?

☞ C'est le « *statut* » retourné au système d'exploitation par le processus correspondant au programme.

Choisir :

- la valeur `0` (ou mieux : `EXIT_SUCCESS`, qui est défini dans `stdlib.h`) si tout va bien,
- autre chose, s'il y a une erreur (un code à vous ou alors `EXIT_FAILURE`).

Jean-Cédric Chappelier

Arguments de `main()`

On a vu le prototype

```
int main(void);
```

Quel est l'autre prototype de `main()` ?

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

Ces arguments sont les **paramètres** donnés par l'interpréteur de commandes appellant la fonction `main`.

Exemple : passer une option « `-v` » et un fichier à un programme

```
monprogramme -v fichier
```

Arguments de `main()` (2)

Dans le prototype

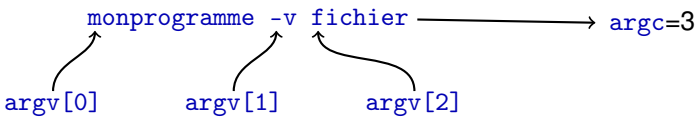
```
int main(int argc, char* argv[])
```

`argc` est un entier comptant le nombres d'arguments (+1) passés au programme

`argv` est un tableau de pointeurs sur des caractères : tableau des arguments

`argv[0]` correspond au nom du programme.

Exemple :



Traitement des arguments de `main()`

```
int main(int argc, char* argv[])
{
  int erreur;
  erreur = traite_arguments(&argc, argv);
  if (erreur != OK) {
    ...
  }
  return erreur;
}
```

un `const int` défini au préalable, par exemple `EXIT_SUCCESS`

On peut distinguer 3 types d'arguments

- obligatoires
  p.ex. un nom de fichier : `rm fichier`
- optionnels
  p.ex. une option d'affichage : `ls -l`
- optionnels avec arguments
  p.ex. changer une valeur par défaut : `dvips -o masortie.ps`

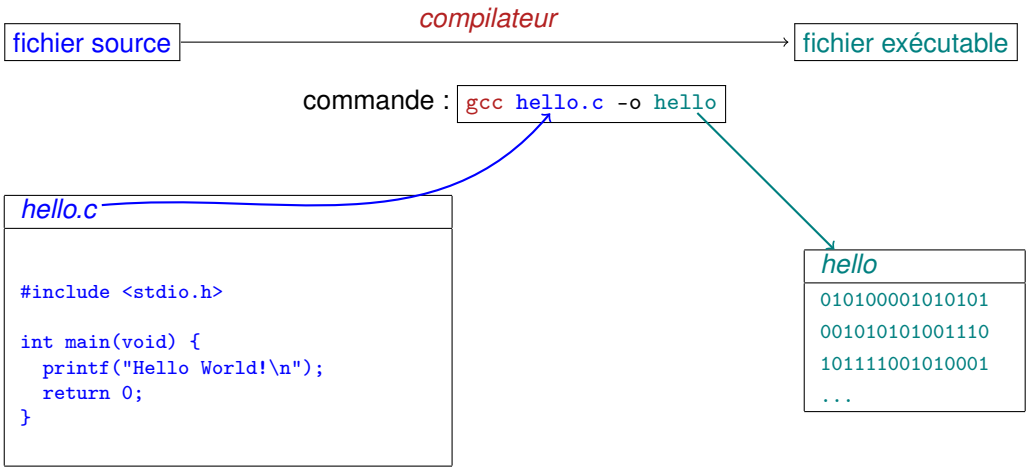
Exemple

```
int traite_arguments(int* nb, char** argv)
{
  int required = 0; // nb d'arguments obligatoires déjà traités
  char const * const pgm_name = argv[0]; // le nom du programme

  ++argv; --(*nb); // passe à l'argument suivant

  while ((*nb) > 0) { // tant qu'il y a des arguments
    if (!strcmp(argv[0], "-P")) { //  option -P
      /* par exemple, avec option_P une variable globale,
       * ou mieux : le champ d'une structure passée en paramètre  */
      option_P = 1;

    } else if (!strcmp(argv[0], "-i")) {
      // une option avec 1 argument : par exemple -i nom
      option_I = 1;
      ++argv; --(*nb); // passe à l'argument suivant
      if (*nb == 0) { // si l'argument de l'option n'est pas là...
        fprintf(stderr,
                "ERREUR: pas d'argument pour l'option -i\n");
        return ERREUR_I; // une constante définie globalement
      } else {
```

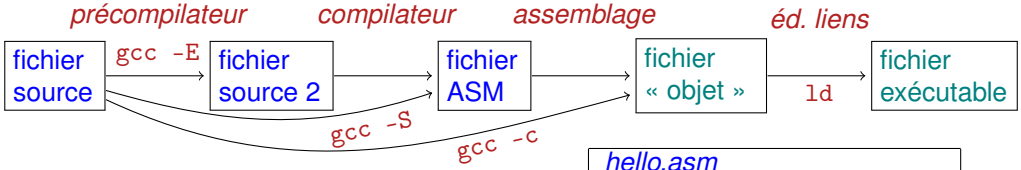
```
      } else {
        // traite l'argument de l'option
        fait_ce_qui_faut(argv[0]);
      }
    } else { // traite les arguments obligatoires
      if (required >= NB_REQUIRED) {
        fprintf(stderr, "ERREUR: je ne comprend pas "
                "l'option %s\n", argv[0]);
        return ERREUR_UNK;
      } else {
        soccupe_argument_obligatoire(argv[0]);
        ++required;
      }
    }
    ++argv; --(*nb); // passe à l'argument suivant
  }

  // vérifie qu'on a bien eu tous les arguments obligatoires
  if (required != NB_REQUIRED) {
    fprintf(stderr, "ERREUR: il manque des arguments\n");
    return ERREUR_LESS;
  }
  return OK;
}
```


Jean-Cédric Chappelier

Compilation d'un programme C (version simple)



Compilation d'un programme C (réalité)



hello_E.c

```
typedef long unsigned int size_t;

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
...
```

hello.asm

```
        .file   "hello.c"
        .section .rodata
.LC0:   .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:  .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
...
```

Etapes de compilation

La compilation est, en fait, une étape un peu plus compliquée que ce que nous avons vu jusque maintenant.

Le compilateur (par abus de langage) effectue en effet plusieurs opérations successives :

- La **précompilation**, dont le rôle est de
  - substituer les macros (récriture)
  - choisir les lignes de codes en compilation conditionnelle
  - inclure les fichiers demandés (directive `#include`)
- la **compilation** proprement dite, qui produit du code assembleur
- l'*assemblage* du code assembleur en code objet
- l'**édition de liens** entre différents codes objets pour en faire un code exécutable (un code « chargeable », en toute rigueur).

Nous ne parlerons pas du tout de la troisième étape (ni d'assembleur) dans ce cours.

Jean-Cédric Chappelier

Macros de précompilation

La commande `#define`, déjà rencontrée au moment des tableaux, est, en fait, un moyen (trop ?) puissant de **récriture de code**.

Attention ! Il faut bien comprendre que cela ne fait **que** récrire du code avant de le passer au compilateur. En se rappelant bien cela, on évitera peut être de commettre des erreurs classiques (voir plus loin).

La syntaxe de `#define` est la suivante :

```
#define alias ( arguments ) séquence_à_récrire
```

où la portion `( arguments )` est optionnelle.

Par exemple

```
#define TAILLE_MAX 12
```

ne fait que dire au compilateur de *remplacer* chaque occurrence de la chaîne `TAILLE_MAX` par la **séquence de caractères** `12`.

Si celle-ci a ensuite un sens pour le compilateur, tant mieux ; sinon un message d'erreur intervient lors de la compilation, **mais concernant la chaîne remplacée !** Ce qui est parfois source de confusion.

#define (suite)

Autre exemple (avec arguments) :
```
#define affiche_entier(x) printf("%d\n", x)
```

Ici l'expression à remplacer sera également **récrite** mais en récrivant également le `x` par la chaîne de caractères donnée entre parenthèse à `affiche_entier`

Ainsi `affiche_entier(i);` sera vu par le compilateur comme `printf("%d\n", i);`

`affiche_entier(12);` comme `printf("%d\n", 12);`

et `affiche_entier(affiche_entier(i));` (pourquoi pas ?)
comme `printf("%d\n", printf("%d\n", i));`

Jean-Cédric Chappelier

`#define` (3)

On peut passer autant d'arguments à la macro que l'on veut, séparés par des virgules.

Exemple (**à améliorer**) :

```
#define mult(x,y) x*y
```

Autre exemple, plus classique :

```
#define max(x,y) ((x) < (y) ? (y) : (x))
```

Note :
`?:` est un opérateur ternaire, tel que « `A ? B : C` »
vaut `B` si `A` est non nul (c.-à-d. « vrai ») et `C` sinon.

#define (4)

Qu'affiche le bout de code suivant ?

```
#define mult(x,y) x*y

printf("%d\n", mult(5-5, 7-2));
```

Réponse :

Pourquoi ?

☞ Appliquez bêtement le fonctionnement de `#define` : il **récrit** !

Le bout de code précédent est donc vu par le compilateur comme

```
printf("%d\n", 5-5*7-2);
```

c'est-à-dire (en mettant plus en évidence les règles de priorités)

```
printf("%d\n", 5 - 5*7 - 2);
```

`#define` (suite)

CONSEIL (voire règle !) **:**

PARENTHÈSEZ les arguments de vos macros !

Écrivez

```
#define mult(x,y) ((x)*(y))
```

(les **deux** parenthèses sont nécessaires), plutôt que

```
#define mult(x,y) x*y
```

Autre conseil : **N'utilisez PAS de macros**, à moins de très bien savoir ce que vous faites

Leur utilisation contient encore d'autres pièges (évaluation multiple des arguments)

☞ **Préférez des fonctions ou des constantes lorsque c'est possible.**

Jean-Cédric Chappelier
EPFL


#define (prise de tête)


Il est possible de faire considérer les arguments d'une macro comme une chaîne de caractères au sens du langage C
(c.-à-d. `x` → `"x"`)... (Écrire pour cela `#x` au lieu de `x`)

...ou de concaténer un argument avec les caractères voisins, en utilisant `##`

Note : ce dernier moyen permet d'avoir un mécanisme sommaire similaire aux templates du C++ (mais à déconseiller, dans ce cas passez plutôt à C++ !)

Exemples :

```
#define affiche(fmt,var) printf("Ici, " #var "=" fmt "\n", var)

affiche("%d", i);                          ☞ printf("Ici, " "i" "=" "%d" "\n", i);

#define coupledef(type) \
    typedef struct { type x; type y; } couple_ ## type

coupledef(double);
                                   ☞ typedef struct { double x; double y; } couple_double;
```


Jean-Cédric Chappelier


#define (prise de tête... ...parfois bien pratique)


```c
#include <stdio.h>

#define SIZE 12
#define STR(X) #X
#define INPUT_FMT(X) "%" STR(X) "s"

int main(void)
{
  char lu[SIZE+1] = "";
  lu[SIZE] = '\0';
  int status;
  status = scanf(INPUT_FMT(SIZE), lu);
  printf(">%s< (%d)\n", lu, status);
  return 0;
}
```

Macros prédéfinies

`__LINE__`	numéro de ligne courante dans le code source
`__FILE__`	nom du fichier en train d'être compilé
`__DATE__`	date de la compilation
`__TIME__`	heure de la compilation
`__STDC__`	1 si conformité au C standard demandée

Exemples d'utilisation :

```
fprintf(stderr,
        "Erreur ligne %d de %s : message d'erreur\n",
        __LINE__, __FILE__ );

const char version[] =
  __FILE__ " du " __DATE__ " " __TIME__;
```

Compilation conditionnelle

La compilation conditionnelle permet de générer, avec le **même** code source, plusieurs programmes exécutables différents.

Un exemple classique (recommandé) est de générer la version «normale» et la version «mise au point» (debug). [voir plus loin]

La compilation conditionnelle se fait à l'aide de l'une des directives de précompilation suivante

```
#if expression
#ifdef identificateur
#ifndef identificateur
```

puis `#elif` ou `#else`, optionnels,
et le tout terminé par `#endif`.

Compilation conditionnelle : Exemples

Un exemple classique :

```
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
```

Note : la constante `M_PI` (et d'autres qui viennent du standard Unix98 et ont aussi été disponibles en 4.4BSD) n'est en toute rigueur pas définie dans le standard C (ni 89, ni 99, ni 11, ni 17) :-(.

Elle est souvent fournie « à bien plaire » par les compilateurs, mais est parfois supprimée si l'on demande de respecter strictement le standard.

Compilation conditionnelle : Exemples

Autre exemple :

```
#ifdef DEBUG
  printf("Ici nous avons i=%d\n", i);
#endif
...
```

Si le programme contient `#define DEBUG`, ou est compilé avec l'option `-DDEBUG` (strictement équivalent), alors le `printf` sera exécuté.
Sinon, c'est rigoureusement comme si la ligne n'existait pas.

Troisième exemple :

```
#ifdef DEBUG
#define affiche(fmt, var) \
   printf("Ici, " #var "=" fmt "\n", var)
#else
#define affiche(fmt, var)
#endif
```

C : divers

Prototype le plus général de `main` :

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

`argc` : nombre d'arguments, taille du tableau `argv`

`argv` : tableau de pointeur sur des caractères : tableau des arguments.

`argv[0]` est le nom du programme

Précompilation :

```
#define alias ( arguments ) sequence a reecrire
```

où la portion `( arguments )` est optionnelle

`#if` *expression*

ou

`#ifdef` *identificateur*

ou

`#ifndef` *identificateur*

puis `#elif` ou `#else`, optionnels,
et le tout terminé par `#endif`.