IFIPS Institut de Formation d'Ingénieurs UNIVERSITÉ PARIS-SUD 11

Introduction au langage C

Premier cours
Notions générales

Aurélien Max
aurelien.max@limsi.fr

Organisation du cours

- 12 séances de cours de 1h:
  mercredi 16-17h, salle A203
- 12 séances de TD de 1h:
  mercredi 10h30-12h30, salle Linux
- Evaluation
  - projet noté
  - exercices de TD (éventuellement)
  - examen final
- Contact
  aurelien.max@limsi.fr

Historique du langage C

- 1972: Dennis Ritchie (Bell Labs) conçoit un langage de programmation structuré mais « près de la machine » pour développer une version portable du système d'exploitation Unix.
- 1978: Brian Kernighan et Dennis Ritchie publient *The C programming language*, livre de référence du standard K&R C.
- 1983: l'ANSI commissionne une définition explicite et indépendante de la machine pour C (standard C ANSI).
- 1988: 2ème édition de *The C programming language,* qui devient la référence du C ANSI.
- 1983: émergence du langage C++ (Bjarne Stroustrup et al., AT&T), ajoutant au C ANSI la programmation orientée objet.

Tour d'horizon
Les avantages du langage C

- **Langage universel**: C n'est pas orienté vers un domaine d'application particulier, comme Fortran (applications scientifiques) ou COBOL (applications de gestion).
- **Langage compact**: le langage est constitué d'un noyau d'opérateurs et de fonctions prédéfinies limité (formulation simple et efficace): se limite aux fonctionnalités qui peuvent être traduites efficacement en langage machine.
- **Langage « près de la machine »**: C offre des opérateurs qui sont très proches de ceux du langage machine et des fonctions qui permettent un accès direct aux fonctions internes de l'ordinateur, notamment pour la gestion de la mémoire.
  → permet de développer des programmes efficaces et rapides.

Tour d'horizon
Les avantages du langage C

- **Langage indépendant de la machine**: C peut être utilisé sur n'importe quel ordinateur disposant d'un compilateur adéquat. C est devenu le *langage de programmation standard* pour les micro-ordinateurs.
- **Langage portable**: un même programme peut être recompilé sur un autre ordinateur utilisant un autre système d'exploitation si la norme C ANSI est respectée.
- **Langage extensible**: en plus des fonctions standard, de nombreuses bibliothèques de fonctions ont été développées.

Tour d'horizon
Les avantages du langage C

- **Langage modulaire:**
  - **Un programme peut être constitué de plusieurs modules, ou « fichiers sources ».**
    - **→Un module regroupe les déclarations de types et les définitions de fonctions relatives à un type de données particulier.**
    - **→Favorise la structuration et la compréhensibilité du code.**
    - **→Favorise la réutilisabilité du code.**
  - **Chaque fichier source peut être traduit ou *compilé* en un fichier objet propre à la machine.**
    - **→ La modification d'un fichier source ne nécessite pas la recompilation de tous les fichiers sources.**
  - **L'ensemble des fichiers objets définissant un programme sont associés pour constituer un fichier exécutable.**

Tour d'horizon
Limitations du langage C

- **Efficacité et compréhensibilité**: il peut être difficile de concilier ces deux exigences, ex.:

```
for (i = 0; i < n; i++)
        printf("%6d%c", a[i], (i%10 == 9) ? '\n':' ');
```

```
for (i = 0; i < n; i = i + 1) {
        printf("%6d", a[i]);
        if ((i%10) == 9) {
                printf("\n");
        }
        else {
                printf(" ");
        }
}
```


Tour d'horizon
Limitations du langage C

- **Efficacité et compréhensibilité** (suite):

```
void fonctionObscure(char t1[ ], char t2[ ]) {
        while (*t2++ = *t1++);
}
```

```
void copieTableau(char t1[ ], char t2[ ]) {
        int indice;
        indice = 0;
        while (t1[indice] != '\0') {
                t2[indice] = t1[indice];
                i = i + 1;
        }
        t2[indice] = '\0';
}
```


Tour d'horizon
Limitations du langage C

- **Limites de la portabilité**:
  - **La portabilité est un des grands avantages de C…**
  - **or le répertoire des fonctions C ANSI est assez limité…**
  - **Si l'on souhaite faire appel à des fonctions spécifiques d'une machine on risque de perdre la portabilité.**
- **Discipline de programmation:**
  - **C offre beaucoup de libertés au programmeur, qui doit donc veiller à adopter un style de programmation solide et compréhensible.**
- **Conclusions (partielles):**
  - **Programmer de façon efficace en langage C nécessite de l'expérience.**
  - **Les commentaires et le choix judicieux des noms des identificateurs sont indispensables pour rendre les programmes compréhensibles, et donc utilisables.**

Pourquoi enseigner le C aujourd'hui? (en 2004)

- C est un langage important de l'histoire de l'informatique, dont dérive en partie les langages C++ et Java:
  - nombre des notions de C sont transposables dans ces langages.
  - important de passer par la « case C » avant d'aborder la programmation orientée objet.
- Beaucoup de programmes existants sont écrits en C, et doivent être maintenus et mis-à-jour.
- C est encore largement utilisé, notamment pour le développement proche de la machine, ex:
  - systèmes d'exploitation (plus de 90% du noyau d'Unix)
  - compilateurs C et outils de programmation

Programmation modulaire

- Un module `Module` est constitué de deux fichiers:
  - un fichier entête, `M.h`:
    - déclaration d'un type de données utilisé dans le programme
    - déclaration d'entêtes de fonctions relatives à ce type
  - un fichier source, `Module.c`:
    - définition des fonctions déclarées dans le fichier entête

Notion d'inclusion

- Tout fichier source doit inclure son fichier entête (directive `#include`): ainsi, les définitions des fonctions correspondent à leur déclaration:

Module1.c
```
#include "Module1.h"
/* définition des fonctions déclarées dans Module1.h */
```

- Tout module utilisant un autre module doit inclure le fichier entête de ce dernier dans son fichier source ou dans son fichier entête:

Module1.c
```
#include "Module1.h"
#include "Module2.h"
/* définition des fonctions déclarées dans Module1.h
et utilisation de celles déclarées dans Module2.h */
```

Exemple de programmation modulaire

- Application: gestion d'un ensemble d'étudiants (définis par un nom, un prénom, une date de naissance et une adresse).
- Modélisation possible:
  - Module « chaîne de caractères » (`chaine.c`, `chaine.h`): définition du type chaîne de caractères et des fonctions associées (saisie, comparaison, affichage, etc.)
  - Module « date » (`date.c`, `date.h`): définition du type date et des fonctions associées.
  - Module « étudiant » (`etudiant.c`, `etudiant.h`): définition du type étudiant et des fonctions associées.

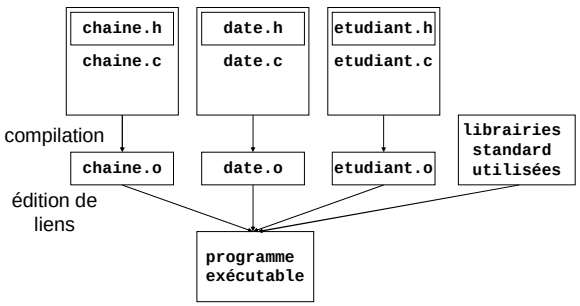
Traduction en langage machine

Traduction des fichiers sources en un fichier exécutable par la machine:

1) compilation:
   - prétraitement par le préprocesseur (inclusions et remplacements)
   - analyse lexicale et syntaxique et optimisation
   - conversion en langage machine (fichier objet)

   `Module.c, Module.h` → `Module.o`
2) édition de liens:
   - réunit les différents modules objets et les fonctions de la bibliothèque standard afin de constituer un programme exécutable

Traduction en langage machine Exemple



Traitements du préprocesseur

- Enlève les commentaires du texte d'un fichier source, compris entre **`/* … */`**
  - la notation **`// commentaire`** sur une ligne est un ajout de C++
- Traite les lignes précédées d'un caractère **#** pour créer le texte que le compilateur analysera:
  - inclusion de fichiers (directive **`#include`**)
  - définition de synonymes et de macro-fonctions (directive **`#define`**)
  - sélection de partie de texte (autour de la directive **`#ifdef`**)

Compilation

- Prend en entrée le texte produit par le préprocesseur.
- Analyse le texte source en une seule lecture du fichier du début à la fin
  - Ce type de lecture conditionne les contrôles que le compilateur peut faire: explique pourquoi toute variable ou fonction doit être déclarée avant d'être utilisée.
- Le compilateur crée des lignes d'assembleur (langage machine) correspondant au texte analysé.

Optimisation de code

- Élimine les parties du code assembleur qui ne sont pas utiles.
- Remplace certaines séquences d'instructions par des instructions propres au processeur de l'ordinateur
  - → produit un code plus compact et optimisé pour le type d'ordinateur sur lequel a lieu la compilation

Assembleur

- Prend en entrée le code généré par le compilateur (éventuellement optimisé) et génère un fichier en format compréhensible pour la machine (code machine)
  - sur les systèmes de type Unix, ces fichiers ont l'extension **`.o`**
  - les fichiers objets contiennent des références insatisfaites (liens entre modules)

Édition de liens

- Produit un exécutable à partir de fichiers objets:
  - prend en entrée les fichiers objets générés par le compilateur
  - prend également en entrée des bibliothèques, en particulier **`libc.a`**
  - utilise un fichier objet contenant le code de démarrage du programme, **`crt0.o`**
  - sur les systèmes de type Unix, l'édition de liens produit par défaut un système exécutable appelé **`a.out`**

Production de fichier exécutable en pratique

- Sous Unix, utilisation de **`cc`** ou de **`gcc`** (version GNU), ex:

```
gcc <options> fichier_source.c
```

  - l'option **`-c`** arrête après la compilation (pas d'édition de liens), ex:

```
gcc –c module.c module.o
```

  - l'option **`-O`** optimise le code généré
  - l'option –o permet de spécifier le nom du fichier exécutable produit, ex:

```
gcc –o monProgrammeGenial module.c
```

- Options propres à **`gcc`**:
  - **`-ansi`** ne prend pas en compte les extensions du C GNU
  - **`-pedantic`** demande au compilateur de refuser la compilation de programmes non ANSI

Gestion de projet en C Makefile

- Un fichier **`Makefile`** permet:
  - d'assurer une compilation séparée à l'aide d'un compilateur C choisi;
  - de ne recompiler que le code modifié depuis la dernière compilation;
  - Contient les instructions indiquant à la commande **`make`** comment exécuter les instructions nécessaires à la création du fichier exécutable.
- Contraintes du fichier **`Makefile`**:
  - Le fichier doit se trouver dans le répertoire courant lorsqu'on invoque l'utilitaire **`make.`**
  - Les instructions contenues dans le fichier obéissent à une syntaxe stricte permettant de définir des règles.

Règles d'un fichier Makefile

- Les règles définissent ce qui doit être exécuté et comment pour construire une cible:

```
cible: dependance
       commandes
```

  - **`cible`**: nom du fichier généré par les commandes qui suivent, ou action générée par ces commandes
  - **`dependances`**: fichiers ou règles nécessaires à la création de la cible
    - Si la cible est un fichier, la cible n'est construite que si ce fichier est plus récent que la cible
  - **`commandes`**: suite de commandes du shell exécutées pour créer la cible:
    - les commandes sont toujours précédées d'une tabulation
    - utiliser \ à la fin de la ligne si les commandes dépassent une ligne de texte

Fichier Makefile Exemple

`Makefile`

```
# fichier Makefile
# pour le programme de gestion d'etudiants
all: chaine.o etudiant.c
      gcc –o gestetud chaine.o etudiant.c
chaine.o: chaine.c chaine.h
      gcc –c chaine.c
```

Exemples d'utilisation sous le shell Unix:

```
> make all
> make
> make chaine.o
```

Fichier Makefile
Macro-commandes

- Déclaration de macro-commande:
  `NOM = VALEUR`
- Appel de macro-commande:
  `$(NOM)`
- Utilisation possible de « jockers », ex.:
  `fichier?.c`

Fichier Makefile
Exemple plus complet

Makefile

```
# fichier Makefile
BIN = gestetud
OBJECTS = etudiant.o chaine.o
CC = gcc
all: $(OBJECTS)
      $(CC) $(OBJECTS) –o $(BIN)
chaine.o: chaine.c chaine.h
      $(CC) –c chaine.c
etudiant.o: etudiant.c etudiant.h
      $(CC) –c etudiant.c
clean: rm –f *.o $(BIN)
```