Programmation impérative

Cours 1

Catalin Dima

Objectifs du cours

- Rappels de C :
  - Structure d'un programme C.
  - Types de variables, tableaux, structures.
  - Fonctions.
- Approfondissement de la programmation en C
  - Pointeurs et organisation de la mémoire, listes chaînées.
  - Gestion des fichiers dans la bibliothèque standard C.
  - Modularité, fonctions, passage de paramètres et valeurs de retour.
- Introduction au système d'exploitation Linux.

Ressources

- Cours en ligne sur `http://lacl.fr/dima/progimp` (à venir).
- D'innombrables livres de programmation...
- Notes de cours
- Pages du manuel `man 3 printf`
- En ligne : `http://code-reference.com/c`

Ressources

- Cours en ligne sur `http://lacl.fr/dima/progimp` (à venir).
- D'innombrables livres de programmation...
- Notes de cours
- Pages du manuel `man 3 printf`
- En ligne : `http://code-reference.com/c`
- Déroulement du cours : 5 premières semaines du semestre.
- Evaluation
  - Examen : 40% de la note finale.
  - 3/4 sujets = programmes à écrire !
- 2 controle en TD.
- Note TP :
  - Comptes rendus après chaque TP.
  - Projet à rendre à la mi-novembre.
  - Triche dans les comptes-rendus ou projets = note 0 en TP !

Rappels de C : structure d'un programme

- Notion de base = fonction.
  - Exemple : `main()`, fonction principale du programme.
- Déclarations de fonctions.
- Déclarations de variables.
- Instructions modifiant les valeurs des variables ou utilisant les fonctions.
- Structures de contrôle.
- Instructions d'interaction avec le système hôte.

Rappels de C : déclarations de variables

- Cas le plus simple : `type nom;`
- Types primitifs :
  - Entiers : `char` (généralement 1 octet), `short` (2 octets), `int` (2 ou 4), `long` (4 ou 8), `long long` (8).
    - Signés ou non-signés !
    - Les caractères alphanumériques sont codés en tant que `unsigned char` – donc tout nombre entre 0 et 255 est un nombre ou un caractère, en fonction de son utilisation !
    - Pas de type booléen ! Toute valeur entière non-nule vaut "`vrai`" !
  - Réels : `float` (4 octets), `double` (8 octets).
- On peut enchaîner les déclarations de variables de même type :
  - `int a,b,c,d,e;`
  - Ne pas oublier le point-virgule ! (erreur de syntaxe – compilo se fâche !)

N'oublions pas les constantes !

- Numériques : 0, 1234, 225.17, 1.3e-100, 0x35, 076...
- Caractère : `'a'`, `'\n'`,...
  - Codage ASCII : une constante numérique entre 0 et 255 représente un caractère.
  - D'autres codages existants : UTF8, Windows-1252, ...
- Déclarations de nouvelles constantes avec `#define`.
- On peut les utiliser lors de la déclaration des variables :
  - `unsigned long var = 12345678`.
  - On parle alors de déclaration *avec initialisation*.
- Constantes particulières : plus grand entier `short` (`SHRT_MAX`), plus petit flottant (`FLT_MIN`), ...
- Certaines variables peuvent être déclarées comme *constantes* !
  - Exemple : `const char mon_a='a';`

Instructions : affectation et expressions

- Affectation simple : `variable = valeur;` (le cas le plus simple !)
- `valeur` peut être une constante...
- ... mais le plus souvent est une *expression* !
- **Expressions** : construites avec des **opérateurs**
  - Opérateurs arithmétiques : `+` , `/` , `%` ,...
    - Attention : division entière sur les types entiers, division réele sur les types flottants !
  - Opérateurs logiques : `||` , `&&` , `!`.
    - Attention : toute valeur entière est une valeur logique !
    - ... mais pas les valeurs flottantes ! !
  - Opérateurs logiques par bits : `|` , `&` , `^`
  - Opérateurs de test : `==` , `<=` , `!=` , le résultat est une valeur booléenne.
    - Attention : ne pas confondre test d'égalité avec affectation !
  - Même l'affectation est un opérateur ! !
    - Cela permet d'enchaîner des affectations : `a=b=c`.
    - Combinaisons : `a+=b`.
  - Incréments et décréments : `++` , `--`

Structures de contrôle

- Les instructions peuvent être groupées en **blocs** :
  - `{x=1; y=2} z=3;`
- Test simple :

```
if (x<5) x+=100;
if (x){
  z=x+y;
  x*=x;
}
else x=y-z;
```

- Les paranthèses font partie de la syntaxe !
- Instructions simples ou blocs !
- Ne pas oublier alors les point-virgule ou les accolades !
- Toute expression *entière* peut être utilisée – même une affectation !
  - Se rapeller de l'interprétation d'une valeur entière comme booléen !
- Instructions conditionnelles à plusieurs ramifications : `switch-case`

Boucles for et while

- Un exemple simple :

```
s=0;
for (i=0;i<10;i++)
    s+=i;
```

- Quelle est la valeur finale de `s` ?
- Autre écriture du même programme :

```
for (s=0, i=0; i<10; s+=i,i++);
```

- Ou encore :

```
s=i=0;
while (i<10)
    s+=i++;
```

Compléments boucles

- Boucle avec test après exécution du corps :

```
s=i=0;
do{
   s+=i++;
}
while(s<1000);
```

- Attention ! Si expression vraie, on continue de boucler !
- Sortie d'une boucle sans terminer son corps :

```
// ici on lit s et i -- on re-verra comment plus tard !
do{
   s+=i++;
   if (s>1000) break;
}
while(i<1000);
```

- Passer à l'itération suivante sans terminer toutes les instructions du corps :

```
// ici on lit s et i, qui peut être négatif !
do{
   if (i++==0) continue;
   s+=i++;
}
while(i<10);
```

Entrées-sorties sur la “console”

Comment affiche-t-on les résultats ? Comment lit-on des valeurs initiales pour nos variables ?

- `printf, scanf` – **fonctions** n'appartenant pas au langage !
  - Le noyau du langage est indépendent du système d'exploitation.
  - ... mais les entrées-sorties sont très dépendantes.
  - Donc c'est aux concepteurs du compilateur d'implémenter les entrées-sorties, en fonction des *appels système* !
- Sous linux : `man 3 printf` – déscription (assez complète, donc compliquée) !
- Il faut inclure `<stdio.h>` dans votre programme !
- Quelques examples :

```
int n=24;
float a=123.45;
printf("%d",n);
printf("%f",a);
printf("n=%u et a=%e",n,a);
```

Entrées avec scanf

```
int s,i;
float a;
scanf("%i%i%f",&s,&i,&a);
```

- Attention ! L'opérateur & est essentiel !

```
char c1,c2;
scanf("%c%d",&c1,&c2);
```

- Un caractère peut être lu en tant que tel, ou en tant que code ASCII !
- Se rappeler de l'identité entre caractères et entiers codés sur 1 octet !

EXEMPLE INCORRECT :

```
scanf("Introduire un entier %d",&i);
```

Fonctions – déclaration

```
type_val_retour   nom_fct(type1 param1, type2 param2....)
```

- Fonctions d'une variable :

  ```
  long factoriel(int parametre){...}
  ```

- Fonctions de deux variables :

  ```
  int max_de_deux_entiers(int parametre1, int parametre2){...}
  ```

- Fonctions ayant un tableau comme paramètre :

  ```
  int max_de_tableau(int tableau[], int dim){...}
  ```

- Fonctions sans variable :

  ```
  float valeur_de_pi(){...}
  ```

- Fonctions sans valeur de retour :

  ```
  void affiche_entier(int quoi_afficher){...}
  ```

Et encore d'autres...

Fonctions – déclaration

- Corps de fonction = suite d'instructions représentant les calculs à faire lors de chaque appel.
  - Peut contenir des déclarations – comme `main`.
  - **Variables locales** – invisibles à l'exterieur !
  - On peut avoir deux variables de même nom dans deux fonctions différentes !
- Une fonction peut être déclarée (**prototype**) sans avoir à détailler le corps.
  - Mais alors le corps devra être décrit ailleurs !
- Instruction spéciale : `return`
  - `return` sans paramètre = terminaison d'une fonction `void`.
  - `return expression` = terminaison d'une fonction non-`void` avec retour de valeur.
  - La valeur retournée doit avoir le même type que celui declaré dans l'entête de la fonction.
- Dans les fonctions ayant un type de retour $\neq$ `void`, `return` est obligatoire !
- Avoir un `return;` dans une fonction `int f(...)` – erreur de compilation !
- Pareil pour un `return 2.5` dans une fonction `int g(...)` !

Fonctions – exemples

```
float valeur_de_pi(){
   return 3.1415926535;
}

void affiche(int val_a_afficher){
   printf("Valeur a afficher = %d", val_a_afficher);
}   // pas de return necessaire !

int max_de_tableau(int a[], int taille){
  int i=0, max = INT_MIN;
                    // i et max = variables locales a la fonction !
  while (i<taille){
     max = (max>a[i]) ? max : a[i] ;
     i++;
  }
  return max;
}
```

Appels de fonctions

Si `fonction` déclarée comme

```
type_de_retour fonction(type parametre);
```

alors on se doit de l'utiliser qu'à des endroits où le `type_de_retour` est utilisable !

- Cas le plus simple : `void`
  - On utilise alors `fonction` comme une instruction à part entière.

```
affiche_entier(2);
```

- Cas général :

```
a[taille++] = max_de_tableau(taille,a);
                // faut pas oublier alors taille++ !
```

Structure d'un programme C

Un petit programme :

```
#include<stdio.h>
main(){
  int s=0,i;
  printf("Donnez un entier");
  scanf("%d",&i);
  if (i<0) printf("J'aime pas les entiers négatifs !");
      else{
          while (s<1000){
             s+=i++;
          }
          printf("somme des %i premiers entiers = %i",i,d);
}
```

Compilation de programmes C

- Un programme C ne peut pas être exécuté en tant que tel sur une machie quelconque !
- Il doit être transformé en code machine (langage de la machine).
- Compilateur = transformateur du programme (en C) en code machine.
- Compilation = génération d'un exécutable qui peut être lancé en exécution sur la machine hôte.
- Erreurs de compilation : signalements du compilateur des erreurs d'écriture du code.
  - On dit que le code n'est pas correcte syntaxiquement !
  - Correction des erreurs syntaxiques : bien lire le message d'erreur affiché par le compilo, identifier l'instruction de l'erreur (pas toujours facile...), corriger, recompiler,...
- Erreurs d'exécution : après compilation (réussie !) et lancement en exécution du programme traduit en code machine, celui-là plante/donne un écran bleu/fait n'importe quoi/ne réagit pas de la façon qu'on attendait de lui.
  - Correction des erreurs d'exécution : faire tourner le programme au ralenti, l'arrêter aux moments importants, afficher des valeurs de variables supplémentaires, le déboguer.
  - Cela demande parfois de rajouter des instructions dans le code (`printf/scanf`), ou d'utiliser un débogueur spécialisé.

Petite initiation en UNIX/Linux

- UNIX : système d'exploitation créé en 1970, en parallèle avec C.
- UNIX développé en C.
- Linux = système d'exploitation construit autour d'une implémentation de la norme POSIX de UNIX (noyau Linux, initialement sur des architectures Intel).
- Système multi-tâche et multi-utilisateur.
- Shell = interface utilisateur "en ligne de commande".
  - Gestion des fichiers.
  - Lancement de programmes (e.g. éditeur et compilateur).

Fichiers et répertoires

- *Fichier* = objet sur lequel on peut lire/écrire/stocker des données/programmes.
- Organisés en *répertoires*,
  - **Arborescence** des répertoires,
  - La racine est /.
- Quelques répertoires standards : `/boot`, `/root`, `/bin`, `/usr`, `/lib`, `/etc`, `/sys`, `/home`...
- Caractéristiques des fichiers et répertoires :
  - **Nom** : `mon_fichier.txt`.
  - **Chemin d'accès absolu** : `/home/dima/mon_fichier.txt`
  - **Taille**.
  - **Droits d'accès** : en lecture, en écriture, exécution.
  - **Appartenance** : propriétaire, groupe d'utilisateurs.
  - Date de création, autres informations (à détailler plus tard).
  - On peut avoir *plusieurs noms pour un même fichier !* (**liens**)

Gestion des fichiers sous Bash

Commandes Linux :

- Affichage : `cat, more, less`
- Création : `> f`
- Destruction : `rm`
- Renommage/deplacement : `mv`
- Copie : `cp`
- Concaténation `cat f1 f2 > f3`
- Création d'un *lien* vers un fichier existant : `$ ln f1 f1bis`. Il y aura deux noms pour le même fichier !

Gestion des répertoires

- Création : `mkdir`
- Destruction : `rmdir`, destruction recursive : `rm -r`
- Copie : `cp -r`
- Affichage des caractéristiques : `ls [-al...]`
- Affichage du contenu : `cat` ou `less`.

À chaque instant, le shell est associé à un **répertoire de travail**

- Le répertoire où, par défaut, nos commandes manipulent les fichiers.
- **Chemin d'accès relatif** au répertoire de travail.

Commandes du shell (ce ne sont pas de commandes Linux !)

- Affichage du répertoire de travail : `pwd`
- Changement du répertoire de travail : `cd`

Expansion des noms de fichiers

- Arguments permettant le traitement de plusieurs fichiers à la fois, en utilisant des *wildcards* : `ls -l *txt` : affiche tout fichier dont le nom se termine en txt
  - `?` = n'importe quel caractère.
  - `*` = n'importe quelle chaîne de caractères.
  - `[abc-f]` = n'importe quel caractère de l'ensemble (intervalle).
  - `[!abc-f]` = n'importe quel caractère absent de l'ensemble.
  - `[[:digit:]]`, `[[:alpha:]]`, `[[:alnum:]]`, etc.
- Les caractères spéciaux `?.*,[,]` doivent être *protégés* pour ne pas être interprétés : `cat 'pourquoi ?'` ou `cat pourquoi\ ?` ou encore `cat "pourquoi ?"`, mais pas `cat pourquoi?`.
- Même problème pour `< > | ,` (virgule) `' "` (apostrophes ou guillemets) et tant d'autres.

Utilisateurs

- Linux (UNIX) = multi-utilisateur !
- Chaque utilisateur identifié par son nom et par son **UID** = descripteur numérique.
- Liste d'utilisateurs (et informations complémentaires) : /etc/passwd.
- Protection entre utilisateurs :
  - Mots de passe.
  - **Droits d'accès** : les attributs de lecture/écriture/exécution sont différents pour des utilisateurs différents.
- Utilisateurs organisés en **groupes**.
- **GID** = descripteur numérique du groupe.

Attributs des fichiers et droits d'accès

```
user> ls -l f.c repertoire/
-rw-r--r-- 1 dima   users   486  Feb  5 16:53  f.c
drwxr-xr-x 2 dima   users  4096  Feb 12 20:19  repertoire/
```

- Chaque fichier possède des attributs signifiant les **droits d'accès** au fichier – 9 bits.
- Lecture/écriture/exécution `[rwx]`.
- Utilisateur, groupe, autres.
- Fichiers/répertoires/liens.
- Taille du fichier, nombre de liens, date/heure de création.
- Changement des droits d'accès : `chmod, chown, chgrp`

Ligne de commande

- Un ou plusieurs mots :
  - nom de commande – 1er mot, peut être aussi un nom de programme !
  - arguments : impératifs ou optionnels.
- Un ou plusieurs séparateurs (espace, tabulation).
- Argument = souvent, le(s) fichier(s) cible de la commande.
  - Mais peut avoir aussi une signification spécifique : nom d'utilisateur/groupe/service, etc.
- Options : paramètres de fonctionnement.
  - Certaines options prennent à leur tour des arguments : `tr -s ' ' fichier.txt`

Commandes Linux utiles

- `head, tail` – affichage d'un nb. de lignes du debut/fin du fichier (implicit 10).
- `file` – type de fichier.
- `cut` – découpe le fichier sur les colonnes (option -c) ou sur les champs (option -f) selon des délimitateurs (option -d).
- `sort` – trie les lignes du fichier, en écrivant le résultat sur la sortie std.
- `find` – recherche conditionnelle et récursive de fichiers.
- `comm, cmp, diff, diff3` – différence entre 2(3) fichiers, `dircmp` – différence entre répertoires.
- `grep` – recherche *dans fichiers* des lignes contenant un *motif*.
- `tr` – substitution de caractères lus de l'entrée std. avec affichage à la sortie std.
- `echo` – affiche le paramètre sur la sortie std.
- `du` – espace alloué dans la sous-arborescence d'un répertoire.
- `date`
- `man,learn`.
  - Recherche la page du manuel d'une commande : `man cp` ou `man 1 cp`.
  - Recherche de la page du manuel d'une fonction C : `man 3 fonction` (mais parfois c'est `man 2 fonction`, ou autre page du manuel !).

Édition et compilation de programmes sous Linux

- Compilateur : `gcc`.
- `gcc monprog.c -o monprog.exe` – compilation, génère exécutable `monprog.exe` !
  - Sans l'option `-o monprog.exe`, un fichier exécutable `a.out` est créé !
- `./monprog.exe` – lancement en exécution du programme compilé !
  - Affichage des résultats et interaction avec le programme lancé – sur la console.