Programmation Avancée

Cours 9 : Faire un terminal

Simon Forest

1 avril 2021

Introduction

Parfois, on se retrouve à recompiler en permanence un code pour faire des calculs légèrement différents :

```
// ...
int main(){
  data *donnees = charger_donnees("fichier1.data");
  // data *donnees = charger_donnees("fichier2.data");

  faireUnCalcul(donnees);
  // faireUnAutreCalcul(donnees);
  // faireEncoreUnAutreCalcul(donnees);
}
```

Introduction

Parfois, on se retrouve à recompiler en permanence un code pour faire des calculs légèrement différents :

```
// ...
int main(){
  data *donnees = charger_donnees("fichier1.data");
  // data *donnees = charger_donnees("fichier2.data");

  // faireUnCalcul(donnees);
  faireUnAutreCalcul(donnees);
  // faireEncoreUnAutreCalcul(donnees);
}
```

Introduction

Parfois, on se retrouve à recompiler en permanence un code pour faire des calculs légèrement différents :

```
// ...
int main(){
  data *donnees = charger_donnees("fichier1.data");
  // data *donnees = charger_donnees("fichier2.data");

  // faireUnCalcul(donnees);
  // faireUnAutreCalcul(donnees);
  faireEncoreUnAutreCalcul(donnees);
}
```

Introduction

Parfois, on se retrouve à recompiler en permanence un code pour faire des calculs légèrement différents :

```
// ...
int main(){
  // data *donnees = charger_donnees("fichier1.data");
  data *donnees = charger_donnees("fichier2.data");

  // faireUnCalcul(donnees);
  // faireUnAutreCalcul(donnees);
  faireEncoreUnAutreCalcul(donnees);
}
```

Introduction

Il peut alors être utile d'introduire une meilleure façon d'interagir avec le programme.

On peut alors penser à un **terminal** : des **commandes** sont passées au programme qui y répond.

```
$ charger fichier1.data
Chargement du fichier OK.
$ executer calcul1
Résultats du calcul:
..
```

Structure

Pour avoir un terminal dans un programme, il faut *a priori* avoir une boucle infinie qui :

- récupère les lignes écrites par l'utilisateur
- les décompose en éléments (nom de commande, arguments)
- appelle les fonctions qui correspondent à ces commandes

Récupérer les lignes

Comme on l'a déjà vu, il y a plusieurs façons de récupérer des lignes en `C` :

Récupérer les lignes

Comme on l'a déjà vu, il y a plusieurs façons de récupérer des lignes en `C` :

- avec `scanf` (mais dangereux)

```c
char ligne[1024];
scanf("%[^\n]",ligne);
```

Récupérer les lignes

Comme on l'a déjà vu, il y a plusieurs façons de récupérer des lignes en `C` :

- avec `scanf` (mais dangereux)

```
char ligne[1024];
scanf("%[^\n]",ligne);
```

- avec `gets` (mais dangereux)

```
char ligne[1024];
gets(ligne);
```

Récupérer les lignes

Comme on l'a déjà vu, il y a plusieurs façons de récupérer des lignes en C :

- avec `scanf` (mais dangereux)

```c
char ligne[1024];
scanf("%[^\n]",ligne);
```

- avec `gets` (mais dangereux)

```c
char ligne[1024];
gets(ligne);
```

- avec `getline`

```c
char *ligne = 0;
size_t capacite = 0;
ssize_t taille = 0;
taille = getline(&ligne,&capacite,stdin);
```

`size_t` et `ssize_t`

Au fait, les types `size_t` et `ssize_t` utilisés par `getline` sont juste des `typedef` vers des types connus.

Ces `typedef` sont faits dans `stddef.h`.

Par exemple, on peut avoir

```
typedef long unsigned int size_t;

typedef long ssize_t;
```

Récupérer la commande

Une fois la ligne lue, il faut récupérer la commande.

En supposant qu'il n'y a que des commandes sans arguments, on peut utiliser `sscanf`.
Avantage : les espaces parasites sont enlevés du `char*` récupéré.

```
char cmd[100];
int r = sscanf(ligne,"%s",cmd);
if(r == 1)
  printf("Commande lue: \"%s\"",cmd);
```

```
executer_commande
```

```
Commande lue: "executer_commande"
```

Récupérer la commande

Une fois la ligne lue, il faut récupérer la commande.

En supposant qu'il n'y a que des commandes sans arguments, on peut utiliser `sscanf`.
Avantage : les espaces parasites sont enlevés du `char*` récupéré.

```
char cmd[100];
int r = sscanf(ligne,"%s",cmd);
if(r == 1)
  printf("Commande lue: \"%s\"",cmd);
```

```
    faire_calcul
```

```
Commande lue: "faire_calcul"
```

Exécuter la commande

Une fois la commande récupérée, on peut lancer la fonction qui lui correspond :

```
if(strcmp(cmd,"faire_calcul") == 0)
  commande_faire_calcul();
else if(strcmp(cmd,"saluer") == 0)
  commande_saluer();
else if (...)
...
```

Invite

Par commodité, on peut afficher une **invite** avant de récupérer la ligne entrée par l'utilisateur.

Cela permet de voir facilement si une commande a fini de s'exécuter et si une autre commande peut être entrée.

```
printf("$ "); // <- invite
char *ligne = 0;
size_t capacite = 0;
ssize_t taille = 0;
taille = getline(&ligne,&capacite,stdin);
// ...
```

```
$ faire_calcul
Calcul terminé. Résultat : 42.
$
```

Arguments

Comment faire maitenant si on veut que certaines fonctions aient des **arguments** ?

```
$ additionner 10 32
Résultat : 42.
$ lire liste-fruits.txt
- pomme
- poire
- raisin
$
```

Découpage

Il faut déjà décomposer une ligne de commande en les différents éléments qu'elle contient.

Découpage

Il faut déjà décomposer une ligne de commande en les différents éléments qu'elle contient.

Par exemple, il faut pouvoir décomposer

```
additionner 10 32
```

en `additionner`, `10` et `32`.

Découpage

Il faut déjà décomposer une ligne de commande en les différents éléments qu'elle contient.

Par exemple, il faut pouvoir décomposer

```
additionner 10 32
```

en `additionner`, `10` et `32`.

On ne peut pas utiliser `sscanf(ligne,"%s",...)` en boucle car on ne sait pas où continuer la recherche après un premier appel.

Découpage

Il faut déjà décomposer une ligne de commande en les différents éléments qu'elle contient.

Par exemple, il faut pouvoir décomposer

```
additionner 10 32
```

en `additionner`, `10` et `32`.

On ne peut pas utiliser `sscanf(ligne,"%s",...)` en boucle car on ne sait pas où continuer la recherche après un premier appel.

À la place, on peut utiliser `strtok`.

strtok

```
char *strtok(char *str, const char *delim);
```

- `str` est une C-chaîne à découper selon des **délimiteurs**
- `delim` est une C-chaîne qui représente un **ensemble de délimiteurs**

Lors du premier appel, on passe la C-chaîne en premier argument.

Lors des appels suivants, on met `NULL` en premier argument pour avoir la suite du découpage.

La fonction renvoie un `char*` vers la chaîne suivante entre délimiteurs (`NULL` s'il n'y en a plus).

Attention : `strtok` modifie `str` dans l'opération.

Exemple

```
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne," ");
          str != NULL;
          str = strtok(NULL," "))
  printf("\"%s\n\"",str);
```

Exemple

```
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne," ");
          str != NULL;
          str = strtok(NULL," "))
  printf("\"%s\n\"",str);
```

```
Bonjour à vous !
```

```
"Bonjour"
"à"
"vous"
"!
"
```

Exemple

```
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne," ");
          str != NULL;
          str = strtok(NULL," "))
  printf("\"%s\n\"",str);
```

```
Bonjour à vous !
```

```
"Bonjour"
"à"
"vous"
"!
"
```

Le retour à la ligne `'\n'` n'étant pas un délimiteur, il est inclus dans le dernier bloc.

Exemple

```
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne," \n"); // ajout de '\n'
          str != NULL;
          str = strtok(NULL," \n"))  // ajout de '\n'
  printf("\"%s\n\"",str);
```

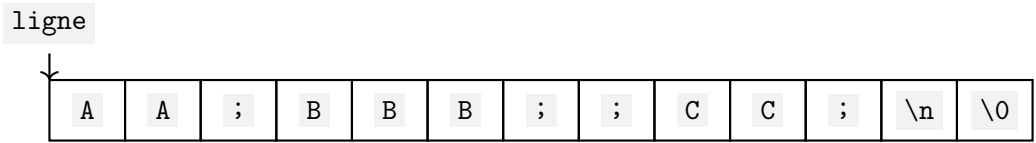
Exemple

```
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne," \n"); // ajout de '\n'
          str != NULL;
          str = strtok(NULL," \n"))  // ajout de '\n'
  printf("\"%s\n\"",str);
```

```
Bonjour à vous !
```

```
"Bonjour"
"à"
"vous"
"!"
```

Fonctionnement

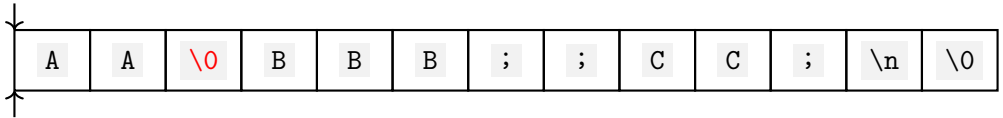



```
char *ligne = NULL; size_t capa = 0;
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne,";\n");
          str != NULL;
          str = strtok(NULL,";\n"))
  printf("\"%s\"\n",str);
```

```
AA;BBB;;CC;
```

Fonctionnement

ligne



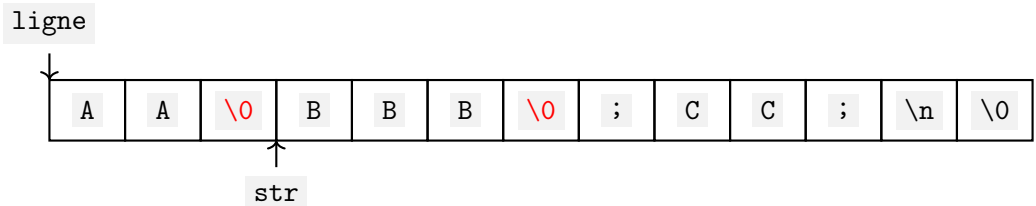
str

```
char *ligne = NULL; size_t capa = 0;
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne,";\n");
          str != NULL;
          str = strtok(NULL,";\n"))
  printf("\"%s\"\n",str);
```

Premier tour :

```
"AA"
```

Fonctionnement

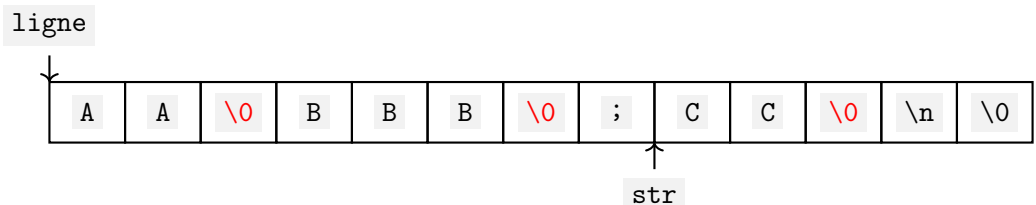



```
char *ligne = NULL; size_t capa = 0;
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne,";\n");
          str != NULL;
          str = strtok(NULL,";\n"))
  printf("\"%s\"\n",str);
```

Deuxième tour :

```
"BBB"
```

Fonctionnement

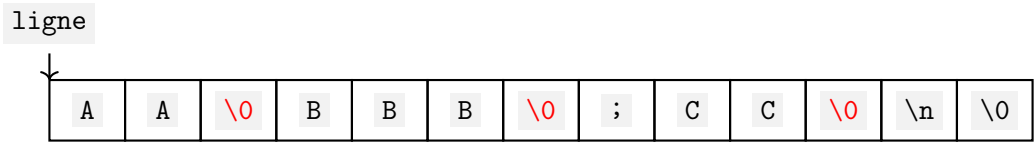



```
char *ligne = NULL; size_t capa = 0;
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne,";\n");
          str != NULL;
          str = strtok(NULL,";\n"))
  printf("\"%s\"\n",str);
```

Troisième tour :

```
"CC"
```

Fonctionnement




```
char *ligne = NULL; size_t capa = 0;
getline(&ligne,&capa,stdin);
for(char *str = strtok(ligne,";\n");
          str != NULL;
          str = strtok(NULL,";\n"))
  printf("\"%s\"\n",str);
```

Pas de quatrième tour.

Récupérer les arguments

```c
while(1){
  char *ligne = NULL; size_t capa = 0;
  getline(&ligne,&capa,stdin);
  char* arguments[100];
  int n_args = 0;
  for(char *str = strtok(ligne," \n");
            str != NULL;
            str = strtok(NULL," \n")) {
    arguments[n_args] = str;
    n_args++;
  }
  if(n_args == 0){ // si la ligne ne contient rien
    printf("Veuillez rentrer une commande\n");
    continue; // on reprend au début de la boucle du terminal
  }
  if(strcmp(arguments[0],"faire_calcul") == 0)
    commande_faire_calcul(arguments,n_args);
  else if(strcmp(arguments[0],"faire_autre_calcul") == 0)
  // ...
}
```

Utiliser les arguments

Une fois les arguments récupérés, on peut les utiliser dans le corps des fonctions associées aux commandes.

```
$ lire fruits.txt
- pomme
- poire
- raisin
```

Utiliser les arguments

Une fois les arguments récupérés, on peut les utiliser dans le corps des fonctions associées aux commandes.

```
void commande_lire(char **arguments,int n_args)
{
  if(n_args != 2) // le premier argument est le nom de la commande
  {
    printf("Erreur: mauvais nombre d'arguments.\n");
    return;
  }
  FILE *file = fopen(arguments[1],"r");
  for(int c = fgetc(file); c != EOF; c = fgetc(file))
    putchar(c);
  fclose(file);
}
```

Utiliser les arguments

Si ce sont des arguments entiers, il faudra les convertir dans un premier temps.

```
$ additionner 11 31
Résultat : 42.
```

Utiliser les arguments

Si ce sont des arguments entiers, il faudra les convertir dans un premier temps.

```
$ additionner 11 31
Résultat : 42.
```

On peut utiliser la fonction

```c
int atoi(const char *str);
```

pour convertir une C-chaîne en `int`.

Utiliser les arguments

Si ce sont des arguments entiers, il faudra les convertir dans un premier temps.

```
$ additionner 11 31
Résultat : 42.
```

```c
void commande_additionner(char **arguments,int n_args)
{
  if(n_args != 3) // le premier argument est le nom de la commande
  {
    printf("Erreur: mauvais nombre d'arguments.\n");
    return;
  }
  int a = atoi(arguments[1]);
  int b = atoi(arguments[2]);
  printf("Résultat : %d.\n",a+b);
}
```

Utiliser les arguments

Si ce sont des arguments entiers, il faudra les convertir dans un premier temps.

```
$ additionner 11 31
Résultat : 42.
```

Problème : avec `atoi`, on ne sait pas si l'utilisateur a entré n'importe quoi.

```
$ additionner pigeon carotte
Résultat : 0.
```

Utiliser les arguments

Si ce sont des arguments entiers, il faudra les convertir dans un premier temps.

```
$ additionner 11 31
Résultat : 42.
```

À la place, on peut utiliser

```
long int strtol(const char *str, char **fin, int base);
```

qui met à jour `fin` pour pointer sur le premier caractère qui ne fait pas partie d'un nombre.

En particulier, si `**fin == '\0'`, alors le nombre entré est correct.

Aussi, `base` permet de préciser la base utiliser. On prendra donc souvent `base == 10`.

Utiliser les arguments

Si ce sont des arguments entiers, il faudra les convertir dans un premier temps.

```
$ additionner 11 31
Résultat : 42.
```

```c
void commande_additionner(char **arguments,int n_args)
{
  if(n_args != 3) { /* traitement comme avant */ }
  char *fin;
  int a = strtol(arguments[1],&fin,10);
  if(*fin != '\0') {
    printf("Erreur: le premier argument n'est pas un nombre.\n");
    return;
  }
  int b = strtol(arguments[2],&fin,10);
  if(*fin != '\0') { /* cas symétrique */ }
  printf("Résultat : %d.\n",a+b);
}
```

Arguments avec espaces

On veut souvent pouvoir écrire des arguments avec des espaces.

Exemple :

```
$ ecrire_dans_fichier msg.txt Ceci est mon message.
```

Ici, on aimerait que `Ceci est mon message.` soit considéré comme le deuxième argument (espaces compris) de `ecrire_dans_fichier`.

Arguments avec espaces

Pour cela, il est standard de spécifier de telles chaînes avec des guillemets :

```
$ ecrire_dans_fichier msg.txt "Ceci est mon message."
```

Ici, la commande `ecrire_dans_fichier` a deux arguments qui sont `msg.txt` et `Ceci est mon message.` (sans les `"..."`).

Ainsi, on autorise deux syntaxes pour spécifier des arguments :

- une version sans `"..."` où l'argument tient en une chaîne sans espaces,
- une version avec `"..."` où l'argument peut contenir des espaces.

Arguments avec espaces

Pour gérer ces deux types de syntaxes, il n'y a pas de fonction toute faite et il faut faire la lecture **à la main**.

Pour cela, on va écrire une fonction

```
char* lire_argument(char* str);
```

qui renvoie le prochain argument dans `str`.

Arguments avec espaces

Pour gérer ces deux types de syntaxes, il n'y a pas de fonction toute faite et il faut faire la lecture **à la main**.

Pour cela, on va écrire une fonction

```
char* lire_argument(char* str);
```

qui renvoie le prochain argument dans `str`.

Ainsi, appeler `lire_argument` sur

```
msg.txt "Ceci est mon message."
```

renvoie `msg.txt`.

Arguments avec espaces

Pour gérer ces deux types de syntaxes, il n'y a pas de fonction toute faite et il faut faire la lecture **à la main**.

Pour cela, on va écrire une fonction

```
char* lire_argument(char* str);
```

qui renvoie le prochain argument dans `str`.

Ainsi, appeler `lire_argument` sur

```
"Ceci est mon message." msg.txt
```

renvoie `Ceci est mon message.`.

lire_argument

```
char* lire_argument(char* str){
  int i = 0;
  for(; str[i] == ' '; i++); // tant que str[i] == ' ', on incrémente i
  if(str[i] == '\n' || str[i] == '\0')
    return NULL;
  char *res = calloc(100,sizeof(char));
  int taille = 0;
  if(str[i] == '"') { // cas "..."
    for(i++; str[i] != '"'; i++) { // Noter le 'i++' au début
      res[taille] = str[i];
      taille++;
    }
  }
  // ...
}
```

lire_argument

```c
char* lire_argument(char* str){
  // ...
  else { // cas non-"..."
    for(; str[i] != ' ' && str[i] != '\n' && str[i] != '\0'; i++) {
      res[taille] = str[i];
      taille++;
      // ou plus simplement: res[taille++] = str[i];
    }
  }
  res[taille] = '\0'; // on termine par '\0' pour avoir une C-chaîne
  return res;
}
```

`for`

Au fait, on n'est pas obligé de définir une variable au début d'un `for` si on n'en a pas besoin :

```
int i = ...;
// ...
for(; str[i] != ' ' && str[i] != '\n' && str[i] != '\0'; i++) {
  // ...
}
```

for

On peut faire à la place une opération comme une incrémentation avant le début de la boucle :

```
if(str[i] == '"')
{
  for(i++; str[i] != '"'; i++) {
    // ...
  }
  i++;
}
```

`for`

On peut aussi spécifier plusieurs opérations à faire à chaque tour de boucle.

Les différentes opérations sont séparées par `,` :

```
int i = ..., j = ..., k = ...;
for(; i + k <= j; i++, j--, k *= 2)
{
  // ...
}
```

Problème

La fonction

```
char* lire_argument(char* str);
```

telle qu'elle est écrite, ne permet pas de savoir où continuer la recherche après avoir lu un argument.

```
getline(&ligne,&capa,stdin);
char *arguments[100];
int i = 0, n_args = 0;
for(char *res = lire_argument(ligne);
          res != NULL;
          res = lire_argument(ligne + i)){
  arguments[n_args++] = res;
  i = ???; // lire_argument ne nous donne pas assez d'information
}
```

Solution

Pour résoudre ce problème, on change le prototype de la fonction en

```
char* lire_argument(char* str, int* offset);
```

où `offset` pointe vers une valeur spécifiant où commencer la recherche et qui sera modifiée pour dire où continuer la recherche.

Solution

```
char* lire_argument(char* str,int* offset){
  int i = *offset;
  for(; str[i] == ' '; i++); // tant que str[i] == ' ', on incrémente i
  if(str[i] == '\n' || str[i] == '\0')
    return NULL;
  char *res = calloc(100,sizeof(char));
  int taille = 0;
  if(str[i] == '"')
  {
    for(i++; str[i] != '"'; i++) { // Noter le 'i++' au début
      res[taille] = str[i];
      taille++;
    }
    i++; // ajouté : on dépasse le '"' terminant
  }
  // ...
}
```

Solution

```
char* lire_argument(char* str,int* offset){
  // ...
  else {
    for(; str[i] != ' ' && str[i] != '\n' && str[i] != '\0'; i++) {
      res[taille] = str[i];
      taille++;
      // ou plus simplement: res[taille++] = str[i];
    }
  }
  res[taille++] = '\0';
  *offset = i; // ajouté : on met *offset à jour
  return res;
}
```

Solution

Code pour récupérer les arguments :

```
getline(&ligne,&capa,stdin);
char *arguments[100];
int offset = 0, n_args = 0;
for(char *res = lire_argument(ligne,&offset);
          res != NULL;
          res = lire_argument(ligne,&offset)){
  arguments[n_args++] = res;
}
```

Autres fonctionnalités

Quitte à avoir des chaînes entre `"..."`, il peut être intéressant de permettre l'écriture de caractères spéciaux dans ce type de chaînes.

Autres fonctionnalités

Quitte à avoir des chaînes entre `"..."`, il peut être intéressant de permettre l'écriture de caractères spéciaux dans ce type de chaînes.

On peut pour cela réutiliser ce que fait le `C` :

- lettres précédées de `\` : `\n`, `\t`, `\r`, *etc.*
- format `\abc`, où `a`, `b`, `c` sont des chiffres entre 0 et 7

Pour cela, il faudrait adapter le code de `lire_argument` pour correctement interpréter ces séquences.

Autres fonctionnalités

Quitte à avoir des chaînes entre `"..."`, il peut être intéressant de permettre l'écriture de caractères spéciaux dans ce type de chaînes.

On peut pour cela réutiliser ce que fait le `C` :

- lettres précédées de `\` : `\n`, `\t`, `\r`, *etc.*
- format `\abc`, où `a`, `b`, `c` sont des chiffres entre 0 et 7

Pour cela, il faudrait adapter le code de `lire_argument` pour correctement interpréter ces séquences.

Ainsi, avec ces caractères spéciaux, on pourrait faire l'appel suivant :

```
ecrire_dans_fichier "mon message.txt" "Ligne 1\nLigne 2\nLigne 3"
```

Dynamiser

Pour simplifier, on peut se permettre d'utiliser des tableaux statiques dans un premier temps comme il a été fait :

```
getline(&ligne,&capa,stdin);
char *arguments[100];
int offset = 0, n_args = 0;
for(char *res = lire_argument(ligne,&offset);
          res != NULL;
          res = lire_argument(ligne,&offset)){
  arguments[n_args++] = res;
}
```

Mais, pour ne pas gâcher de mémoire et / ou que le programme ne crache, c'est mieux de passer au dynamique à un moment.

Dynamiser

```
getline(&ligne,&capa,stdin);
char **arguments = NULL;
int offset = 0, n_args = 0;
for(char *res = lire_argument(ligne,&offset);
          res != NULL;
          res = lire_argument(ligne,&offset)){
  arguments = realloc(arguments,(n_args + 1) * sizeof(char *));
  arguments[n_args++] = res;
}
// utilisation ...
free(arguments);
```