Information, Calcul et Communication

Composante Pratique: Programmation C++

Semaine 13: entrée-sortie conversationnelle

Le role des buffers d'entrée-sortie / la redirection

Lecture : détection et traitement d'erreur

Conversion chaine de caractères

Sortie : formatage

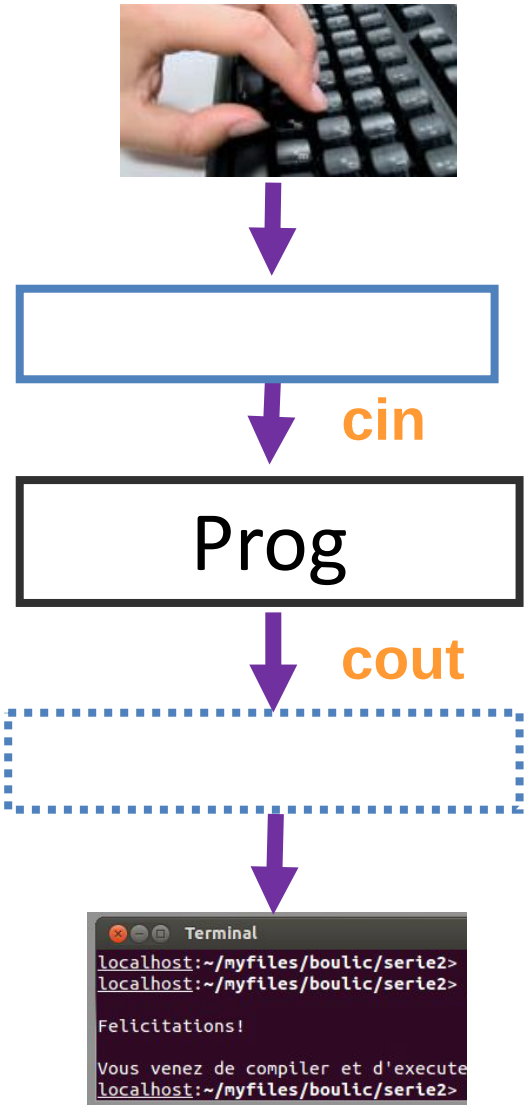
Sortie et bug

Vue d'ensemble

Entrées-sorties conversationnelles standard

obtenues avec le lancement:

`./Prog`



Dans le cas général des entrées-sorties
on parle de **flot** (*stream*)

cin est ***l'entrée standard***
= la variable associée par défaut au **flot** d'entrée

cout est la ***sortie standard***
= la variable associée par défaut au **flot** de sortie

Vue d'ensemble (2)

Pour l'utilisateur: ***Entrée*** *au clavier*
Pour le programme *c'est comme si on lisait le contenu d'un fichier* **cin**

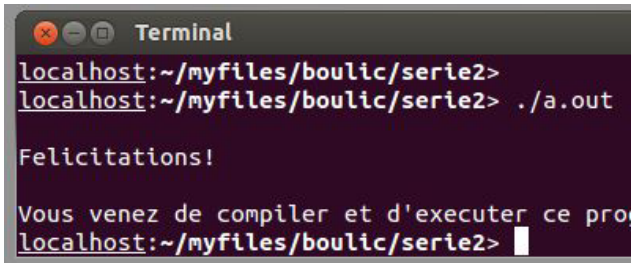
Synchronisation grâce à un ***buffer*** *d'entrée*

gestion des buffers par le système d'exploitation

Programme en cours d'exécution

Stockage dans un ***buffer*** *de sortie*

buffer = mémoire tampon = mémoire intermédiaire



Pour l'utilisateur: ***Sortie*** *= affichage dans terminal*
Pour le programme: *c'est comme si on écrivait dans le fichier* **cout**

Rappel: principe de la redirection

Quoi & Pourquoi ?

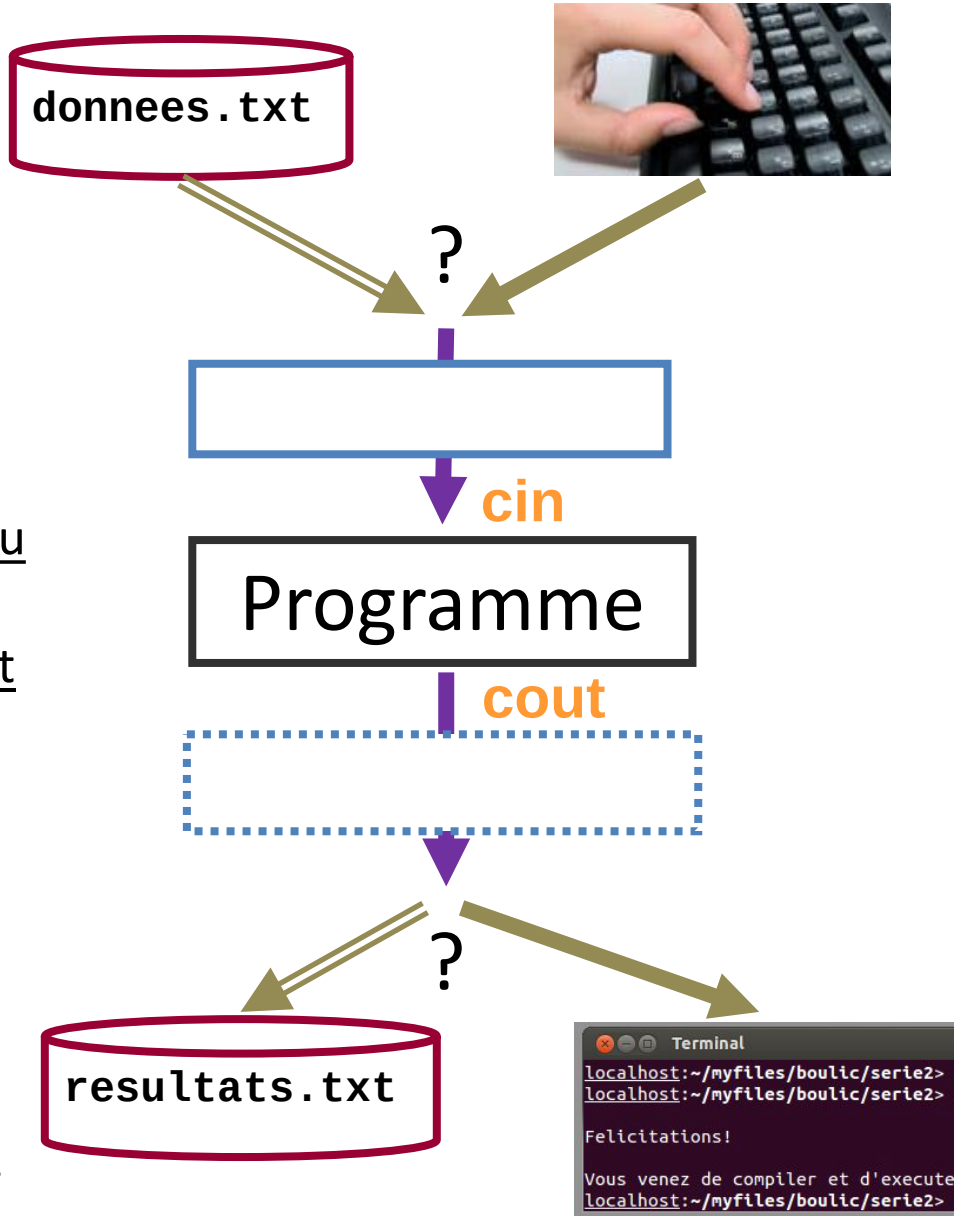
remplacer l'entrée et/ou la sortie standard par un fichier. Grande efficacité pour fournir des données (jeux de tests) et pour mémoriser les résultats.

Quand ?

indiquer au moment du lancement du programme. Valable pour TOUTE la durée de cette exécution ; on ne peut plus changer en cours d'exécution.

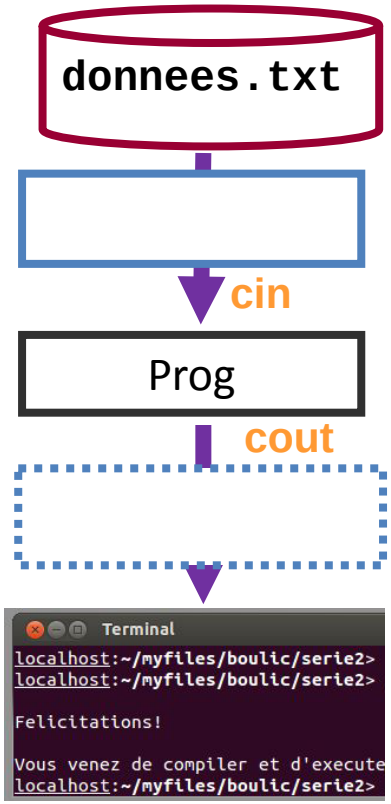
Qui ? responsabilité du système d'exploitation.

Le programme n'est pas modifié ; de son point de vue c'est toujours le clavier et le terminal qui sont utilisés.

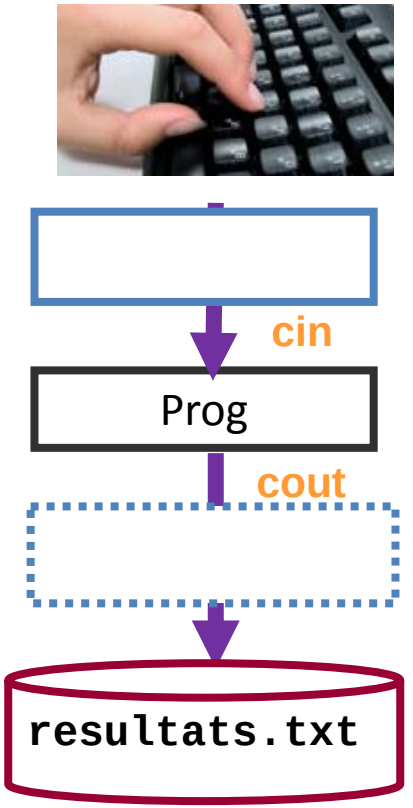


Rappel: principe de la redirection (2)

`./Prog < donnees.txt`



`./Prog > resultats.txt`



Si le fichier **resultats.txt** existe déjà son ancien contenu est effacé. Le symbole '**>>**' permet d'ajouter à la suite d'un fichier de résultats déjà existant.

La lecture

Qui fait Quoi ?

Le système d'exploitation gère la phase d'édition

Analogie avec un système de messagerie instantanée (*chat*):
l'expéditeur d'un message peut le modifier à volonté
tant qu'il ne l'a pas validé avec la frappe de "Enter" (**\n**) ;
le texte n'est pas visible par le destinataire ("le programme").

Le buffer d'entrée contient la suite des caractères (code ASCII).

Le programme travaille avec le contenu du buffer d'entrée

Si le buffer contient une suite de caractères validée par "Enter",
alors ils vont être consommés par les appels successifs des
fonctions de lecture du programme (**cin >>**, getline(), etc...)

La lecture (2) Rappel Topic4

Reprenons le code de **lecture_et_test.cc** :

Ce code lit successivement 3 nombres : un double, un int puis un float

La suite de **caractères alphanumériques** qui est fournie au clavier est **consommé selon le type de la variable** lue par les instructions **cin** successives du programme .

Que se passe-t-il avec: 0.1 1.1 suivi par Enter.

Il faut au moins un espace ou un passage à la ligne comme séparateur entre 2 valeurs destinées à 2 variable distinctes. Il peut y en avoir plusieurs : ils sont ignorés. Pour l'exemple ci-dessus, on a:

1) Lecture de la valeur **double 0.1** pour **a**

2) la lecture d'un **int** ne consomme QUE le **signe** et les **10 chiffres**.

PAS PLUS. La lecture consomme seulement **1** pour le **int b.**

3) la lecture se poursuit *avec les caractère non-consommés*.

Donc la lecture suivante récupère la valeur **double .1** pour **c**.

Cas général de la lecture **Rappel Topic6**

1) Tant qu'on n'a pas validé ce qui est tapé au clavier avec **Enter**, le *buffer* d'entrée est vide → il n'y a rien à «lire» pour le programme
2) Dès le première validation avec **Enter**, ce qui est validé est mémorisé par le système (*buffer d'entrée*).
3) Ce qui est mémorisé dans le *buffer d'entrée* est extrait et consommé par les appels successifs de >> ou getline() etc sur **cin.**
4) **cin >>** *filtre les* ***séparateurs*** = les espaces, tabulation, Enter sont ignorés.
5) En cas d'échec de lecture d'une donnée le caractère fautif reste dans le buffer d'entrée pour le prochain appel de >> sur cin.

Définition: les ***séparateurs*** sont appelés **whitespace** et noté **ws**

Alternatives à la lecture formatée avec >>

Rappel Topic6

La lecture formatée avec >> ne répond pas à tous les besoins.

- Lecture «bas-niveau» de tous les caractères, y compris les séparateurs

➔ demander explicitement chaque caractère

```
char c;
cin.get(c);
```

➔ Variante permettant de détecter la fin de fichier avec **EOF** *(End Of File)*

```
int c;
while((c = cin.get())!= EOF)
{
  // traiter le caractère lu c
  // ici affichage avec put()
  cout.put(c);
}
```

On peut produire **EOF** au clavier avec **Ctrl-D**

Alternatives à la lecture formatée avec >> *(2)* Rappel Topic8

- Lecture *non-formatée* d'une ligne entière dans une **string**

➔ La fonction **getline()** prend tous les caractères jusqu'au prochain passage à la ligne (qui est extrait du buffer d'entré):

```
string ligne;
getline(cin,ligne);
```

- Attention: une lecture non-formatée ne filtre pas les ***séparateurs***
  - Scénario typique: si un appel à **getline**() suit immédiatement une entrée formatée avec **>>** alors **getline**() récupère une ligne contenant seulement le caractère de passage à la ligne qui validait la donnée pour l'opérateur **>>**.

  - Solution : demander un filtrage explicite des ***séparateurs*** AVANT l'appel à getline() avec l'une ou l'autre des syntaxes suivantes:

```
cin >> ws;
getline(cin,ligne);
```

```
getline(cin >> ws,ligne);
```

Echec de la lecture

Qu'est-ce qu'un échec de la lecture ?

```
double x(-1.0);
int n(-1);
cin >> n >> x ;
```

input clavier	conséquence et valeur de **n** et de **x**
3 0.7	succès **n** vaut 3 et **x** vaut 0.7
0.7 3	succès **n** vaut 0 et **x** vaut 0.7 et 3 reste dans le buffer d'entrée car arret dès le caractère '.' non-accepté pour **int**
.7 3	**ECHEC** **n** a une valeur indéterminée et x devrait être inchangé. Tout reste dans le buffer d'entrée car **aucun** caractère acceptable pour **int**

Comment supprimer ce qui reste dans le buffer d'entrée?

```
double x(-1.0);
int n(-1);
cin >> n >> x ;
```

input clavier ¦ conséquence et valeur de **n** et de **x**

`0.7  3` succès **n** vaut 0 et **x** vaut 0.7 et **3 reste dans le buffer d'entrée** car arret dès le caractère '.' qui n'est n'est pas accepté pour **int**

`cin.ignore();` // supprime le prochain caractère du buffer d'entrée

```
#include <limits>
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Le premier paramètre de la fonction **ignore()** indique le nombre max de caractères à supprimer dans le buffer d'entrée **jusqu'à trouver le second paramètre**. Si on donne l'expression ci-dessus, TOUS les caractères sont supprimés jusqu'à trouver le second paramètre.

Détection de l'échec de la lecture et reprise en main

```
include <limits>
```

```
double x(-1.0);
int n(-1);

// L'expression de lecture « cin >> n >> x »  vaut false en cas d'échec,
// sa négation est donc true

if(not(cin >> n >> x))  // ECHEC !
{
   cin.clear();  // refaire passer cin dans l'état d'accepter une lecture

   cin.ignore(numeric_limits<streamsize>::max(),'\n');
}

// Autre possibilité: tester immédiatement après la lecture avec fail()
cin >> n >> x;
if(cin.fail())  // ECHEC !
{
   ….
```

Détection de l'échec causée par une fin de fichier avec eof()

```
double x(-1.0);
int n(-1);

if(not(cin >> n >> x))  // ECHEC !
{
  if(cin.eof()) // eof() est vrai si l'échec est causée par une fin de fichier
  {
    cout << "Plus rien en entrée !" << endl;
    exit(0);
  }
  else
  {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(),'\n');
  }
}
```

Conversion chaîne de caractère vers types numériques

Ayant une suite de char qui représente un nombre comment obtenir la valeur numérique avec un type tel que **int**, **double**, etc ?

On dispose d'une famille de fonctions «string-to-numeric-type» → sto… ()
Ex: conversion vers double // existe aussi avec int, float, unsigned, long…

```
double stod(string& s, size_t *p_nb=0);
```

Si on sait que la string **s** est correcte
inutile d'utiliser le second paramètre

Si par contre la string à convertir **s** est un input, le second paramètre permet de récupérer le *nombre de caractères traités par la conversion*. Ce nombre devrait être égal à la taille de la chaine pour un succès complet.

```
string s("37.2");
double x(stod(s));
…
size_t nb(0);
x = stod(argv[1],&nb);

size_t all(strlen(argv[1]));
          // include <cstring>
if(nb != all)
    cout << echec ! ;
```

Sortie formatée : les outils

→ 2 outils pour agir sur le format d'affichage pour les sorties formatées:

- Les **manipulateurs** appliqués à l'opérateur **<<**
- Les options de configuration du flot **cout** (non-traité dans ce cours)

Pour utiliser les **manipulateurs** il faut ajouter: **`#include <iomanip>`**

Syntaxe:

```
cout << manipulateur  << expression << … ;
```

Deux scénarios de persistence selon les **manipulateurs** :

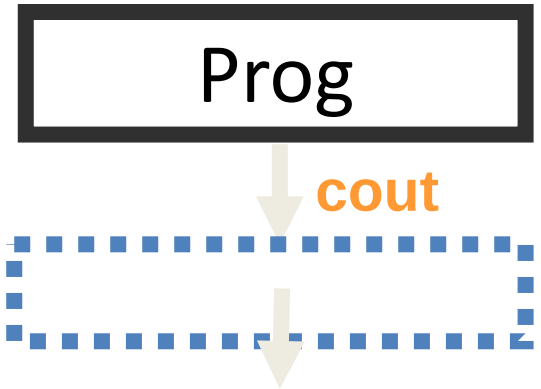
- Affecte seulement *l'expression* qui suit
- Affecte l'***état*** *de* ***cout*** pour *tous les affichages ultérieurs*
  - Il existe alors un **manipulateur** «inverse» qui annule son action

Formatage	manipulateur	permanent ?
	`cout << …`	
base **16**	**`hex`** // aussi valable // pour cin	oui
base **8**	**`oct`**	oui
base **10**	**`dec`**	oui
montrer base: **0** pour 8, **0x** pour 16	**`showbase`**	oui
désactivation avec	**`noshowbase`**	oui
n chiffres à droite du point décimal	**`setprecision(n)`**	oui
notation point décimal avec **n** chiffres à droite du point décimal	**`fixed`**	oui
notation scientifique avec **n** chiffres à droite du point décimal	**`scientific`**	oui
notation avec **6** chiffres significatifs	**`defaultfloat`**	oui
réserve **n** colonnes min	**`setw(n)`**	**non**
utilise **c** pour les colonnes vides	**`setfill(c)`**	oui
cadrage à gauche	**`left`**	oui
cadrage à droite	**`right`**	oui

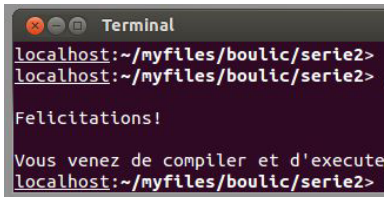
cout *et le buffer de sortie*

Du fait du temps relativement important requis pour les opérations d'affichage sur le terminal, ou plus généralement pour l'écriture dans un fichier, ce type d'opération n'est PAS immédiatement exécutée:

La chaîne à afficher est stockée dans le **buffer de sortie**



Le buffer de sortie est vidé "dans le terminal" si :



- **il est plein** (~plusieurs milliers de char)
- **caractère de contrôle** **endl**
- dès qu'il y a **appel d'une fonction de lecture**
  - cin >>, getline, get, etc

cout: *mes bugs et le buffer de sortie*

Exercice: voici un scénario dans lequel un bug fait planter votre programme. Les affichages ont été ajoutés pour trouver la portion de code incorrect. Cependant seul l'affichage de **`test1`** est fait.

Pourquoi ? Comment corriger ce problème d'affichage ?

```
cout << "test1" << endl;
...ici du code correct mais on ne le sait pas encore

cout << "test2" ;

...ici du code incorrect: l'exécution s'arrête ici sans
   affichage de «test2»

cout << "test3" << endl;
```

Résumé

les entrées se font par l'intermédiaire d'un buffer (mémoire tampon) qui permet de synchroniser le fonctionnement du programme avec l'utilisateur.

La lecture avec l'opérateur **>>** peut traiter les cas d'erreur sur les types de donnée à fournir.

L'affichage sur cout est immédiat *seulement* s'il se termine par endl. Sinon le **buffer** de sortie est rempli et n'est affiché que lorsqu'il est plein ou lorsqu'une lecture est effectuée.

En sortie formatée, les manipulateurs et les options sont deux moyens possibles pour jouer sur la presentation de l'affichage dans le terminal.

La redirection permet d'organiser efficacement les tests