Information, Calcul et Communication

Composante Pratique: Programmation C++

Semaine 13: entrée-sortie conversationnelle

Les entrées-sorties: vue d'ensemble et rappels / la redirection

Sortie et bug

Lecture : détection et traitement d'erreur

Conversion chaine de caractères

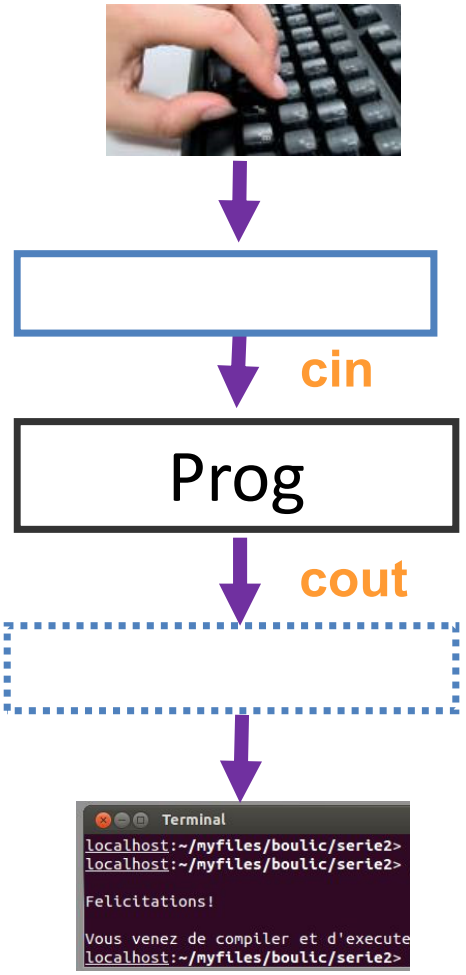
Sortie : formatage

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**

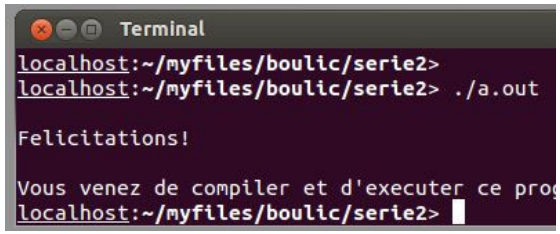
Synchronisation grâce à un ***buffer*** *d'entrée*

Programme en cours d'exécution

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

gestion des buffers par le système d'exploitation

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: La redirection

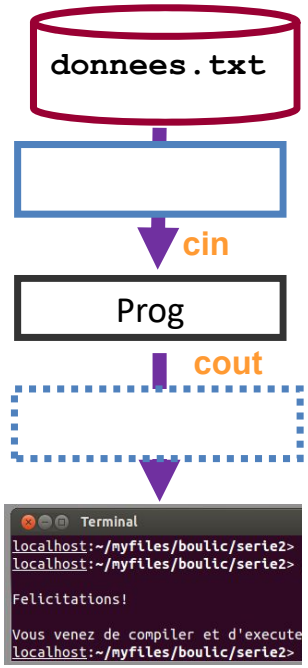
Quoi & Pourquoi ?
remplacer l'entrée et/ou la sortie standard par un fichier. Grande efficacité pour fournir des données (jeux de tests) et pour mémoriser les résultats.

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.

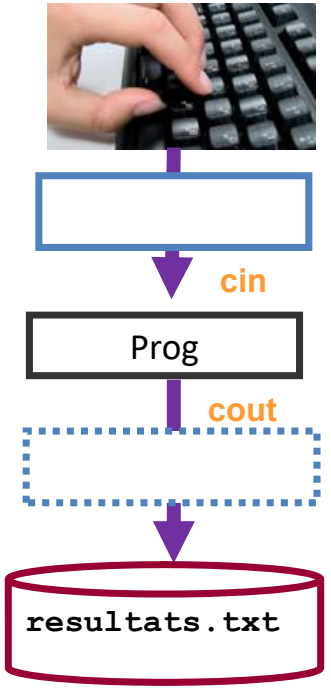
Qui ? responsabilité du système d'exploitation.

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

`./Prog < donnees.txt`



`./Prog > resultats.txt`

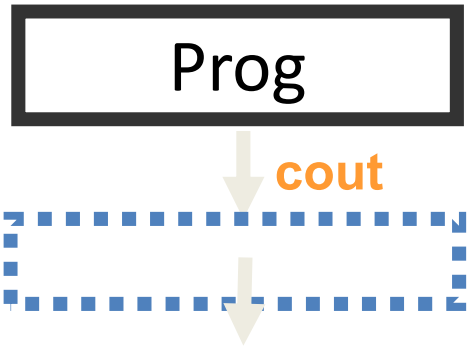


Si le fichier **resultats.txt** existe déjà son ancien contenu est effacé. Le symbole '>>' permet d'ajouter à la suite d'un fichier de résultats déjà existant.

EPFL

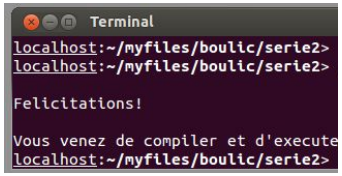
cout *et le buffer de sortie*

Du fait du temps relativement important requis pour les opérations d'affichage sur le terminal, ou plus généralement pour l'écriture dans un fichier (M3.L3), 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" = affichage effectif



seulement si :

- **fin** (normale) **du programme**
- **il est plein** (~plusieurs milliers de char = typiquement un bloc du disque = 4K)
- **caractère de contrôle** `endl`
- dès qu'il y a **appel d'une fonction de lecture**
  - `cin >>, getline, get,` etc
- s'il y a une demande explicite de le vider avec flush:
  - `cout.flush() ;`
  - `cout << flush ;`

SpeakUp: what is displayed in the terminal when running cout_quizz.cc ?

Note: this program compiles without any warning

A.
```
Segmentation fault
```

B.
```
1 test1
Segmentation fault
```

C.
```
1 test1
2 test2
3 test3
```

D.
```
1 test1
2 test2
Segmentation fault
```

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int  v1(1), v2(2), v3(0), *v4(nullptr);
7
8      cout << v1       << " " ;
9      cout << "test1"  << endl;
10
11     cout << v2       << " " ;
12
13     v3 = v1 + v2 ;
14
15     cout << "test2" ;
16
17     *v4 = 4 ;
18
19     cout <<  v3      << " "  ;
20     cout << "test3"  << endl ;
21
22     return 0;
23 }
```

cout: *mes bugs et le buffer de sortie*

Contexte: Quand on recherche un bug une pratique courante est d'effectuer des affichages pour trouver la portion de code incorrect. L'exemple suivant présente un défaut ; pourquoi ? Comment corriger ce problème ?

```
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;
```

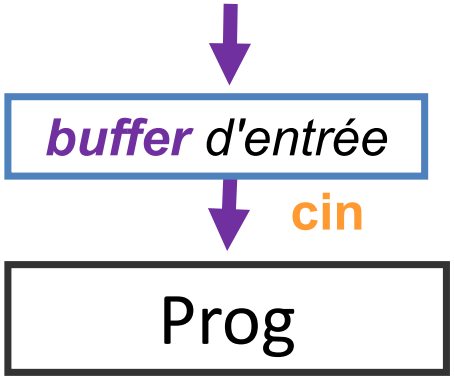
Conclusion: si on fait de la mise au point en affichant sur **cout**, il faut imposer l'affichage immédiat en ajoutant **<< endl**

La lecture

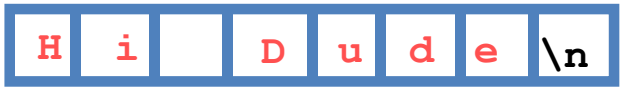
Qui fait Quoi ?

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

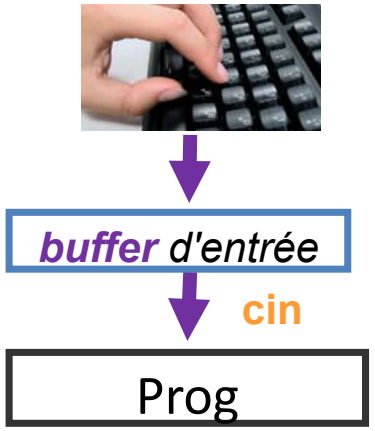
Si le buffer contient une suite de caractères validée par "Enter",
alors ils vont être consommés par les appels successifs des
fonctions de lecture du programme (**cin >>**, get(), getline(), etc...)

Rappel: la boucle et les outils pour effectuer la lecture

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 la première validation avec **Enter**,
ce qui est validé est mémorisé par le système (dans le ***buffer*** *d'entrée*).

3) Ce qui est mémorisé dans le *buffer d'entrée* est extrait et <u>consommé</u>
par les appels successifs des outils de lecture sur **cin :**



`cin >>` Lecture formatée: *filtre les* ***séparateurs*** = les espaces, tabulation, Enter

En cas d'arrêt de lecture d'une donnée, le caractère fautif reste dans le buffer d'entrée pour le prochain appel de >> sur cin.

`cin.get(c)` *ne filtre PAS les* ***séparateurs*** ; **tout** est lu avec la variable **char c** ;

`getline(cin,line)` lit une ligne entière jusqu'à Enter dans la string **line**

`getline(cin>>ws,line)` même chose en filtrant les séparateurs qui précèdent la ligne

<u>**Définition**</u>: les ***séparateurs*** sont appelés **<u>white</u>space** et noté **ws**

Rappel: exemple de lecture formatée avec arrêt partiel de décodage

```
double a(0); int b(0); float c(0);
cin >> a >> b >> c ;
```

Que se passe-t-il si on fournit **`0.1  1.1`** suivi par `Enter`.

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** ; le décodage s'arrête pour **int**.
3. la lecture se poursuit *avec les caractère non-consommés disponibles dans le buffer*.

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

Exemple d'erreur en lecture non-formatée: le cas de la redirection

Comme la lecture non-formatée peut lire tous les caractères, on pourrait penser qu'il n'y a jamais d'erreur.

Il reste le cas d'un fichier redirigé sur l'entrée et qui *ne contient pas assez de donnée*

Cette variante de **get()** obtient un **int** qui signifie «*fin de fichier / **End of File*** » représentée par le symbole **EOF**

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

get_eof_put.cc

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

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**

La méthode `ignore()` pour supprimer ce qui reste dans le buffer d'entrée

En cas de décodage partiel ou d'échec avec la lecture formatée,
=> il reste des caractères indésirables dans le buffer d'entrée

```
cin.ignore();  // supprime seulement 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

Le flot **cin** renvoie **true** en cas de succès de lecture formatée.

```
include <limits>
```

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

// test de l'expression de lecture « cin >> n >> x »  qui vaut false en cas d'échec

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');
}
```

Le flot **cin** met également à jour un mot **d'état** à chaque lecture pour documenter le type de problème.
On peut tester juste après la lecture si le problème provient d'une erreur de format avec la méthode **fail()**

```
cin >> n >> x;
if(cin.fail())   // ECHEC !
{
```

Détection de l'échec causée par une fin de fichier avec la methode `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              // problème de format => vider le buffer
  {
    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 => stod() // float => stof ; int => stoi ; unsigned long => stoul ; long => stol ; etc…

```
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

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