ASR5 - Système d'Exploitation

Prérequis

Fabien Rico

Univ. Claude Bernard Lyon 1

Séance 1

Fabien RICO	fabien.rico@univ-lyon1.fr	CM+TD+TP
Jacques BONNEVILLE	jacques.bonneville@univ-lyon1.fr	TP
Adil KHALFA	adil.khalfa@cc.in2p3.fr	TD + TP
Yves CANIOU	yves.caniou@univ-lyon1.fr	TP
Dorra BOUGHZALA	dorra.boughzala@ens-lyon.fr	TP

1 Introduction

2 C++

3 Compilation/exécution

4 Prérequis Unix

Prérequis : UE Système d'Exploitation

- Utilisation d'Unix
- Programmer en C/C++
- Compilation d'un code

Comportement

C'est une UE technique

- On la réussit si on comprend des concepts et si on apprend à faire des opérations.
- On comprend si on pose des questions (en TP, TD **et** CM).
- L'ensemble est sans doute trop complexe pour être vu en si peu de temps.
  - Il y aura du code fourni (et incompréhensible).
  - Vos camarades ne comprennent pas plus que vous.
  - On peut copier du code si on est capable de l'utiliser.
  - Les corrections vont parfois plus loin que ce qui est demandé.

Prérequis en C++

- `C++11` :
  - développé à partir de 2003
  - standard en 2011
- *Standard Template Library*
  - surtout les `vector`, `string`, `stream`
  - éventuellement les *expressions régulières*

Un site important : `http://fr.cppreference.com/w/`

Standard Template Library

Bibliothèque d'objets disponible sur tous les compilateurs C++, elle utilise des objets basés sur des modèles (*template*). *Par exemple les conteneurs :*

```
std::vector<char> tableau;
std::list<int> liste;
```

qui correspondent aux tableaux dynamiques ou aux listes que vous avez programmés en LIFAP3. Ils sont beaucoup *plus efficaces et stables que vos productions.*
Certains types sont des template *cachés, par exemple les* `std::string` *(voir* `http://fr.cppreference.com/w/cpp/string/basic_string`*).*

Problèmes avec les *templates*

Ils compliquent beaucoup les messages d'erreur :

```
string chaine = "coucou";
cout << chaine + 4 << endl;
```

Donne :

```
template.cpp:9:18:draw between picture
   erreur : no match for « operator+ » (operand types are
  « std::__cxx11::string
       {aka std::__cxx11::basic_string<char> » and « int »
   cout << chaine + 4 << endl;
   ...
   /usr/include/c++/6.3.1/bits/stl_iterator.h:341:5: note :
   ...
```

Cela signifie que, à la ligne 9 du fichier, l'opération plus entre un `string` et un `int` n'existe pas.

Problèmes avec les *templates*

Ils compliquent beaucoup les messages d'erreur :

```
string chaine = "coucou";
cout << chaine + 4 << endl;
```

Donne :

```
template.cpp:9:18:draw between picture
   erreur : no match for « operator+ » (operand types are
  « std::__cxx11::string
        {aka std::__cxx11::basic_string<char> » and « int »
   cout << chaine + 4 << endl;
   ...
   /usr/include/c++/6.3.1/bits/stl_iterator.h:341:5: note :
   ...
```

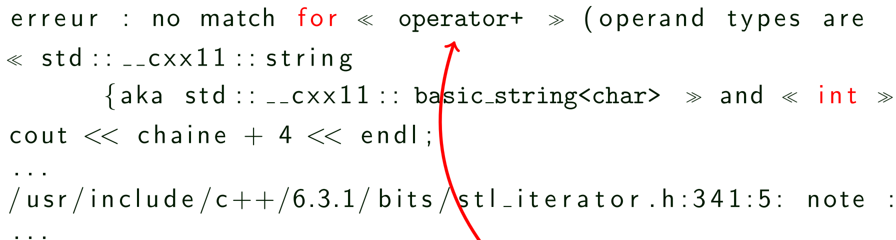
Cela signifie que, à la ligne 9 du fichier, l'opération plus entre un `string` et un `int` n'existe pas.

Problèmes avec les *templates*

Ils compliquent beaucoup les messages d'erreur :

```
string chaine = "coucou";
cout << chaine + 4 << endl;
```

Donne :



```
template.cpp:9:18:draw between picture
    erreur : no match for « operator+ » (operand types are
   « std::__cxx11::string
         {aka std::__cxx11::basic_string<char> » and « int »
    cout << chaine + 4 << endl;
    ...
    /usr/include/c++/6.3.1/bits/stl_iterator.h:341:5: note :
    ...
```

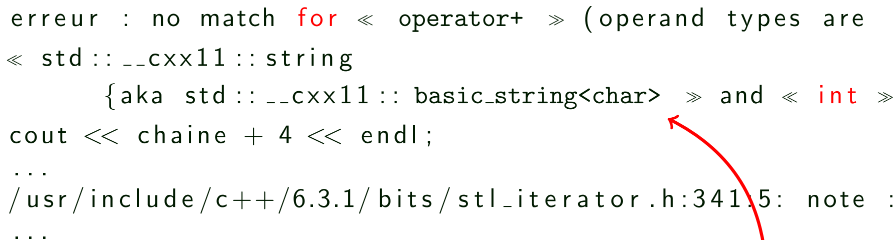
Cela signifie que, à la ligne 9 du fichier, l'opération plus entre un `string` et un `int` n'existe pas.

Problèmes avec les *templates*

Ils compliquent beaucoup les messages d'erreur :

```
string chaine = "coucou";
cout << chaine + 4 << endl;
```

Donne :



```
template.cpp:9:18:draw between picture
    erreur : no match for « operator+ » (operand types are
   « std::__cxx11::string
         {aka std::__cxx11::basic_string<char> » and « int »
    cout << chaine + 4 << endl;
    ...
    /usr/include/c++/6.3.1/bits/stl_iterator.h:341:5: note :
    ...
```

Cela signifie que, à la ligne 9 du fichier, l'opération plus entre un `string` et un `int` n'existe pas.

STL ce qu'il faut connaître - std::vector

```
std::vector<T> tab;
std::vector<T> tab(int taille);
```

Un tableau dynamique contenant des éléments de type T. Si on crée un tableau avec une taille de départ, *il faut pouvoir construire les éléments sans argument*.

- vector<T>::size() : la taille du tableau ;
- T∗ vector<T>::data() : un accès au tableau C des données ;
- vector<T>::push_back(T e) : ajoute une case et un élément au tableau ;
- ... voir ici

Attention : Quand on remplit une case de tableau on *copie la valeur*, cela peut poser des problèmes pour les objets non copiables.

vector : exemple

```
int main(int argc, char* argv[]) {
  std::vector<int> tab; // tableau vide
  std::cout << "Donnez des nombre, 0 pour arrêter SVP" << std::endl;
  while (true) {
    int c;
    std::cin >> c;
    std::cout << "lu " << c << std::endl;
    if (c == 0) break;

    tab.push_back(c); // remplisage du tableau
  }
  // taille du tableau
  int nb = tab.size();
  std::cout << "Vous avez fournis "
            << nb << "valeurs." << std::endl;

  // création d'un tableau de char avec le même nombre d'octet
  std::vector<unsigned char> tab2(nb*sizeof(int));
  //copie du tableau en utilisant une fonction C
  memcpy(tab2.data(), tab.data(), nb*sizeof(int));
  // Affichage en hexadécimal
  for (uint i=0; i<tab2.size(); i++) {
    fprintf(stdout, "tab[%d] = %2x\n", i, tab2[i]);
  }
}
```

`vector` : gestion de la mémoire

- Un `vector` est une zone mémoire modifiable de taille dynamique.
- L'agrandissement et la suppression en fin est *en moyenne en temps constant*.
- Lorsqu'une fonction retourne un `vector`, cela *ne génère pas de copie* des données contenues (voir l'élision de copie).
- La mémoire d'un `vector` est *automatiquement libérée* lorsque l'objet est détruit.
- Attention cependant à la copie de `vector` ($\neq$ référence java).

Conseil : Depuis C++11, les `vector` ont pour vocation de remplacer les anciens tableaux C/C++ que vous avez utilisés ! Ils permettent l'utilisation de tableaux *sans se soucier de la libération de la mémoire*.

C++ 11 ce qu'il faut connaître - copie

Certains objets ne doivent pas être copiés comme ceux liés à des éléments du système qui ne doivent pas être partagés (fichiers, threads, zone mémoire, ...), mais avec les actions implicites du C++, il est parfois difficile de savoir lorsqu'on fait une copie.
Le C++11 permet *d'interdire la copie*

```
objetNonCopiable &operator=(const objetNonCopiable&) = delete
```

Ainsi le compilateur est capable de vérifier que l'on ne fait pas d'erreur.
Mais on obtient cela (avec gcc) :

```
Exemple/copie.cpp: Dans la fonction « int main(int, char**) »:
Exemple/copie.cpp:26:20: erreur : use of deleted function «
std::basic_ifstream<_CharT, _Traits>
  ::basic_ifstream(const std::basic_ifstream<_CharT, _Traits>&)
  [with _CharT = char; _Traits = std::char_traits<char>] »
  ifstream fich1 = fich;
                   ^~~~

In file included from Exemple/copie.cpp:2:0:
/usr/include/c++/6.3.1/fstream:519:7: note : declared here
       basic_ifstream(const basic_ifstream&) = delete;
```

C++11 ce qu'il faut savoir - copie d'objet temporaire

Un objet qui ne doit pas être copié peut l'être dans un cas particulier : *quand il disparaît*. C'est le cas de tous les objets dont on voudrait transmettre les données.

- retour de fonction ;
- objet temporaire ;
- objet qu'on transmet de manière explicite : std::move.

Il est donc en général possible de copier un objet lorsqu'il est temporaire ou que sa durée de vie se termine.

```
ifstream fich("toto.txt");
//création d'un tableau avec 2 cases
vector<ifstream> tab_fich(2);

tab_fich[0] = fich; // ERREUR c'est une copie
tab_fich[0] = std::move(fich); // BON
// transmission explicite (mais fich devient invalide)
tab_fich[1] = ifstream("toto.txt"); // BON
// copie d'objet temporaire
tab_fich.push_back(ifstream("toto.txt")); // BON
// ajout d'objet temporaire
```

C++ 11 ce qu'il faut connaître - type auto

Type retrouvé automatiquement par le compilateur.

```
  std::map<std::string, float> notes;

  notes["Chaprot"] = 1.5;
  notes["Raffin"] = 4;
  notes["Gotlib"] = 10;

  // Pas besoin de se souvenir du nom du type
  for (auto p: notes) {
    std::cout << p.first << " : " << p.second << std::endl;
  }
}
```

Compilation

La compilation est le fait de traduire votre code pour en faire un programme éxécutable.

- Il y a 3 phases :
  - *Précompilation* : utilisation de directives pour compléter le code ou adapter au système.
  - *Compilation* : traduction du code en langage machine.
  - *Édition de liens* : ajout de liens avec les fonctions utilisées (celles que vous avez programmées comme celles qui sont fournies).
- À chaque phase il y a des options et des erreurs différentes.
- Le résultat est souvent un programme qui dépend encore des bibliothèques dynamiques.

Précompilation

Le compilateur va modifier le code fourni en fonction des ordres donnés par les directives :

- #include <fichier> = recopie le fichier ici ;
- #pragma once = ne fait qu'une seule inclusion du fichier concerné
- #define TEXTE VALEUR dans la suite remplace tous les `TEXTE` par `VALEUR`
- #ifdef NOM ... #endif = ne conserve le code suivant que si `NOM` est défini.

Les erreurs lors de la précompilation sont rares, mais une mauvaise utilisation peut générer un code incorrect, ce qui apparaîtra dans la 2e phase.

Précompilation (suite)

- #include <iosteam> :

  ```
  /tmp/toto.cpp:1:19: erreur fatale : iosteam : Aucun fichi
  de ce type
  ```

  Le fichier n'existe pas où le compilateur ne le trouve pas.
- #define TAILLE 20; : le mot clef `TAILLE` va être remplacé par 20;. Donc

  ```
  int tab[TAILLE]; //devient int tab[20;];
  //La compilation donne :
        /tmp/toto.cpp:3:18: erreur : expected « ] »
        before « ; » token
    int tab[TAILLE];
  ```

Les options importantes du compilateur g++ sont :

- −I rep : répertoire où chercher les fichiers inclus
- −D NOM : définir la macro `NOM`
- −E : ne faire que la précompilation pour voir le résultat

Compilation

C'est la phase principale qui traduit le code dans un autre langage. Le compilateur en profite pour faire des vérifications, ce qui génère beaucoup de messages d'erreur. Attention,

- les messages sont là *pour vous aider* ;
- les corriger suppose *un choix éclairé du programmeur* ;
- si un algorithme était capable de corriger les erreurs, c'est ce qui se passerait.

Donc :

- En cours, si on ne comprend pas un message d'erreur, il faut poser une question *avant* de tenter une correction.
- Mal corrigée, une erreur de compilation simple peut se transformer en un problème de pointeur difficile à retrouver.
- « Je ne sais pas ... mets $*$ ici. » n'est jamais une solution.
- Il faut limiter la portée des erreurs et leur nombre donc *compilez souvent* et *testez dès que possible*.

Compilation

Les options utiles du compilateur g++ sont :

- −g rendre possible le *débogage* ;
- −std=c++11 prévenir que le code est du C++11 ;
- −Wall demander le plus possible de messages d'aide ;
- −O0 pas d'optimisation (pour aider au débogage) ;
- −o s'arrêter à la compilation pour faire un fichier objet.

Il faut savoir modifier les options, *même avec un IDE !*

Édition de liens

Pour faire un exécutable, il faut nécessairement que le système soit capable de retrouver le code de *TOUTES* les fonctions utilisées.

- les fonctions que vous avez écrites ;
- les fonctions que les autres ont écrites (enseignants, collègues, ...) ;
- les fonctions qui sont censées exister depuis toujours.

Par exemple :

```
/tmp/ccAUJJRT.o : Dans la fonction « main » :
lien.cpp:11 : référence indéfinie vers « operator+(
        std::__cxx11::basic_string<char, std::char_traits<
                std::allocator<char> > const&,
        int const&) »
collect2: erreur : ld a retourné 1 code d'état d'exécution
```

Signifie :

lors de l'édition de lien , je ne trouve pas le code de

la fonction + des `strings` et des `int`

Édition de liens

Pour faire un exécutable, il faut nécessairement que le système soit capable de retrouver le code de *TOUTES* les fonctions utilisées.

- les fonctions que vous avez écrites ;
- les fonctions que les autres ont écrites (enseignants, collègues, ...) ;
- les fonctions qui sont censées exister depuis toujours.

Par exemple :

```
/tmp/ccAUJJRT.o : Dans la fonction « main » :
lien.cpp:11 : référence indéfinie vers « operator+(
        std::__cxx11::basic_string<char, std::char_traits<
                std::allocator<char> > const&,
        int const&) »
collect2 erreur : ld a retourné 1 code d'état d'exécution
```

Signifie :

lors de l'édition de lien , je ne trouve pas le code de

la fonction + des `strings` et des `int`

Édition de liens

Pour faire un exécutable, il faut nécessairement que le système soit capable de retrouver le code de *TOUTES* les fonctions utilisées.

- les fonctions que vous avez écrites ;
- les fonctions que les autres ont écrites (enseignants, collègues, ...) ;
- les fonctions qui sont censées exister depuis toujours.

Par exemple :

```
/tmp/ccAUJJRT.o : Dans la fonction « main » :
lien.cpp:11 : référence indéfinie vers « operator+(
        std::__cxx11::basic_string<char, std::char_traits<
              std::allocator<char> > const&,
        int const&) »
collect2: erreur : ld a retourné 1 code d'état d'exécution
```

Signifie :

lors de l'édition de lien , je ne trouve pas le code de

la fonction + des `strings` et des `int`

Édition de liens

Rappel : il faut que le compilateur sache où se trouve le code de toutes les fonctions. C'est-à-dire qu'il réponde à 2 questions :

- Comment s'appelle les fichiers contenant ce code compilé ?
- Où chercher ces fichiers ?

Une réponse fausse : ≪ *Je ne comprends pas, j'ai bien mis #include <LaBibliothequeQuOnMaDitDeMettre>* ≫

Pourquoi est-ce faux ?

Édition de liens

Dans certains langages (`python`, `java`, ...) le code est mélangé avec les entêtes, une seule directive import donne tout ce qu'il faut.
En C/C++, le code est souvent dans un fichier à part, il faut dire à l'éditeur de liens où il se trouve grâce à des options du compilateur ou alors directement lui donner les fichiers :

- −L REPERTOIRE : répertoire où chercher les fichiers des bibliothèques.
- −lNOM : nom du fichier à utiliser (le véritable nom est `libNOM.so` sous Unix) *Attention*, certains compilateurs demandent à ce que l'option −l soit **placée après** le code qui l'utilise.
- NOM1.o NOM2.o ... liste des fichiers à rassembler pour produire l'exécutable.

Édition de liens : exemple

```
g++ -g -Wall main.o bilioprof.o
       -L/opt/lib/ -lm -lpthread -o prog.exx
```

Signifie : *compile les 2 fichiers `main.o` et `bilioprof.o` en ajoutant le code des fonctions présentes dans `libm.so` et `libpthread.so` qui sont dans le répertoire `/opt/lib` ou dans un répertoire habituel*

Pour connaître les *endroits habituels*, il faut consulter la documentation de l'utilitaire `ldconfig`, du fichier `/etc/ld.so.conf` et des variables `LIBRAIRY_PATH` `LD_LIBRAIRY_PATH`).

Gestion des bibliothèques usuelles

Vous allez utiliser des bibliothèque assez courantes, cela signifie que :

- Le fichier contenant le code des fonctions est en général présent sur votre ordinateur.
- Ce n'est pas souvent le cas des fichiers nécessaires à la compilation (`.h`).

Pour installer sur votre ordinateur les fichiers nécessaires à la compilation, il faut installer les paquets de développement.

- Sous `Ubuntu/Debian`, les paquets de développement se terminent souvent par −dev.
- Sous `Fedora/RedHat`, les paquets de développement se terminent souvent par −devel.

Par exemple, pour utiliser la bibliothèque `boost`, il faut installer libboost−all sous Ubuntu et boost sous Fédora. Pour compiler des programmes qui utilisent cette bibliothèque, il faut installer libboost−all−dev sous Ubuntu et boost−devel sous Fédora.

Prérequis Unix

Unix est un système qui utilise beaucoup la ligne de commande de plus :

- les autres systèmes aussi :
  - MacOS repose sur Bsd ;
  - Windows utilise le Powershell il implémente maintenant un Bash complet.
- les commandes sont *plus stables* ;
- les commandes permettent les *scripts*.

Un grand nombre de commandes vont être utilisées dans l'UE, vous devez les connaître.

Objectifs sous Linux

1. Il faut savoir faire les tâches en ligne de commande ou avec une interface graphique.
   - Si vous utilisez un IDE, apprenez à le configurer.
   - Apprenez à utiliser un débogueur.
2. Il est préférable de savoir faire les tâches en ligne de commande.
   - Il faut connaître le nom de la commande.
   - Il faut savoir lire sa documentation.
3. Il est plus simple de connaître par cœur certaines commandes.

Pour réussir, il faut pratiquer.

Déroulement

- Des notes de TD :
  - retours sur les cours précédents ;
  - petits codes à écrire.
- Des notes de TPs :
  - ce que vous avez fait pendant le TP ;
  - ce que vous avez compris.
- Un CCF.