Programmation avancée en C :

Portabilité, maintenabilité et réutilisabilité

Licence informatique 3e année

Université Gustave Eiffel

Portabilité

- Qu'est-ce qu'un code portable ?
- Indépendance vis-à-vis :
  - du compilateur
  - du système
  - de la machine
- Pourquoi le faire ?
  - force à coder proprement en prenant du recul
  - facilite la diffusion des applications

Un code qui compile partout

- Pour qu'un code compile toujours, il faut :
  - éviter les choses exotiques comme `#pragma`
  - respecter les normes `-ansi -Wall`
  - cerner le code qui dépend du compilateur, du système, de la machine
  - éviter les dépendances sur des bibliothèques non portables
  - faire des Makefile portables

Cerner le code variable

- Tout code pouvant varier doit être cerné avec des directives préprocesseurs, et si possible, isolé dans des fichiers à part

```
1 /* System-dependent function that compares files names */
2 int fcompare(const char* a, const char* b){
3 #ifdef WINDOWS_LIKE
4     return strcmp_ignore_case(a, b);
5 #else
6     return strcmp(a, b);
7 #endif
8 }
```

Cerner le code variable

- Même chose pour les inclusions, les types, les constantes, etc.

```
#ifdef WINDOWS_LIKE
#include "mygetopt.h"
#else
#include <getopt.h>
#endif
```

```
#ifdef WINDOWS_LIKE
#define PATH_SEPARATOR_CHAR '/'
#define PATH_SEPARATOR_STR "/"
#else
#define PATH_SEPARATOR_CHAR '\\'
#define PATH_SEPARATOR_STR "\\"
#endif
```


Noms de fichiers

- Casse importante sous certains systèmes
- Il faut être soigneux :

motor.c :

```
#include "Motor.h"

/* ... */
```

motor.h :

```
#ifndef motorH
#define motorH
/* ... */
#endif
```

Ce code compile sous `Windows`, mais pas sous `Linux`.

Dépendances au système

- 2 types de dépendances :
  - valeurs/types (séparateur de fichier `/` ou `\`)
  - comportements (casse des noms de fichiers)
- Pour le premier type, la gestion par `#ifdef` suffit.
- Pour le second, pas toujours.

Dépendances au système

- Chaque fois que quelque chose dépend du système, il faut l'expliciter.
- Exemple : longueur des noms de fichiers
  - Mauvaise solution : constante arbitraire
  - Bonne solution : utiliser la constante `FILENAME_MAX` (`stdio.h`) adaptée au système courant.

Tailles des types

- Anticiper les différences d'architecture
- Utiliser les constantes de `limits.h`
- Utiliser `sizeof`
- Exemple non portable :

```
1 void** new_ptr_array(int n){
2   void** ptr = malloc(n * 4);
3   if (ptr == NULL){
4     /* ... */
5   }
6   return ptr;
7 }
```


Tailles des types

- Si on a besoin d'un type avec une taille en octets fixe, utilisez les constantes et types de `stdint.h`.
- Exemple : si on veut gérer Unicode non étendu, on a besoin d'un type non signé sur 2 octets.

```
1 #include <stdint.h>
2
3 typedef uint16_t unichar;
```


Les sauts de ligne

- Windows : `\r \n`
- Linux : `\n`
- Anciennes versions de MacOS : `\r`
- Il ne suffit pas que le programme soit portable.
- Est-ce que les données doivent l'être ?
  - Comment faire pour lire/écrire un fichier texte multi-plateforme ?

Endianness

- x86-like : little-endian
- Motorola-like : big-endian
- Il ne suffit pas que le programme soit portable.
- Est-ce que les données doivent l'être ?
  - comment lire/écrire un fichier binaire contenant des `int` d'une façon multi-plateforme ?
  - Exemple de solution : encodage UTF8

Makefile

- Écrire des Makefile portables avec une variable qui dépend du système
- Informations concernées :
  - Compilateur à utiliser
  - Options de compilation
  - Répertoire (include, lib, etc)
  - Commandes d'installation et de nettoyage
  - Noms des sorties (exemple : `.exe` ou pas ?)
  - Et bien d'autres...

Makefile

```
CCFLAGS=-Wall -ansi
ifeq ($(OS),Windows_NT)
    CCFLAGS += -D WIN32
    ifeq ($(PROCESSOR_ARCHITECTURE),AMD64)
        CCFLAGS += -D AMD64
    endif
    ifeq ($(PROCESSOR_ARCHITECTURE),x86)
        CCFLAGS += -D IA32
    endif
else
    UNAME_S := $(shell uname -s)
    ifeq ($(UNAME_S),Linux)
        CCFLAGS += -D LINUX
    endif
    ifeq ($(UNAME_S),Darwin)
        CCFLAGS += -D OSX
    endif
    UNAME_P := $(shell uname -p)
    ifeq ($(UNAME_P),x86_64)
        CCFLAGS += -D AMD64
    endif
    ifneq ($(filter %86,$(UNAME_P)),)
        CCFLAGS += -D IA32
    endif
endif
```


Makefile

- Les versions de Windows depuis 2000 possèdent une variable d'environnement `OS` dont la valeur est `Windows_NT`.
- Les systèmes basés sur `Unix` ont une commande `shell uname` qui donne une chaîne décrivant le système.
- En stockant dans une variable du Makefile la description de l'OS, on peut alors donner des instructions dédiées pour chaque OS (`-fPIC` est inutile sous Windows pour les bibliothèques dynamiques par exemple...).
- Une entreprise commercialisant des logiciels a tout intérêt à ne pas développer deux fois le même logiciel.

De l'art de développer rapidement et efficacement

- Science appelée le génie logiciel
- Pour développer un logiciel :
  - **On réfléchit,**
  - et puis on produit du code.
- La production du code :
  - Implantation de l'application avec ses fonctionnalités (10% du temps de travail)
  - Débogage, documentation, affinement, tests, optimisations (90% du temps de travail)

Fait : La seconde partie de la production est d'autant plus réduite que l'on a bien réfléchit avant de passer à l'action

Dans l'imaginaire collectif

"Pour être un bon développeur en langage *foo*, il suffit d'être bon en algo et de maîtriser la syntaxe de *foo*."

Proverbe étudiant

"Quand on est con, on est con."

Georges Brassens

Qu'est ce que le génie logiciel ?

- Ensemble de bonnes pratiques et de conseils à méditer
- Puis un autre ensemble de mauvaises pratiques à éviter
- Objectifs : développer mieux et plus vite*

  - mieux = moins de bugs, code réutilisable, évolutif, etc.

  - plus vite = accélérer le débogage, éviter de tout casser tout le temps, etc.
- **Qui se hâte n'atteint pas le but.** (*Confucius*)

*(comment devenir des feignants efficaces)

Règle de modularité

Écrire des éléments simples et les relier par des interfaces propres

- Pour faire des applications complexes, il faut pouvoir contrôler la complexité du code.
- Principes d'encapsulation en bibliothèques *boîtes noires*

Règles de séparation

Séparer méthodes et mécanismes
Séparer moteur et interfaces

- Bonne pratique : ne rendre visible que les interfaces et pas les implémentations en utilisant `static`
- Exemple : frontal GUI (Graphical User Interface) dissocié des tâches de fond (affichages et calculs sont deux choses différentes)
- 1 module = 1 responsabilité

Règle de clarté

Préférer la clarté à l'intelligence

- Penser à celui qui relira le code (y compris vous-même à l'avenir).
- Un algorithme trop rusé a moins de chance d'être lisible qu'un classique.
  - risque plus élevé de bugs
  - moins facile à entretenir
- Style obscurantiste à bannir absolument

Règle de composition

Concevoir des programmes à connecter à d'autres programmes

- Éviter les programmes "Dieu" qui font tout
- Esprit d'`Unix` : de multiples petits programmes bien taillés qui pourraient être assemblés facilement via des pipes
- Ne pas réinventer la roue, la plupart du temps on en fait une roue carrée.
  - Utiliser les choses faites par ceux qui savent
  - Utiliser les dépendances proprement

Règles de transparence

Concevoir un comportement lisible pour faciliter l'investigation et le débogage

- N'utiliser que de bonnes interfaces
- Tout ce qui facilite le débogage est bon.
- Mettre des sorties de debug (avec `stderr`).
- Ne pas lésiner sur les tests
- Ne pas cacher les bugs

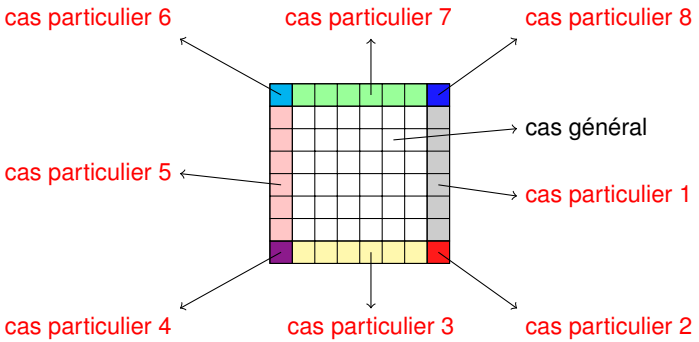
Règle de robustesse

Engendrer de la robustesse par la transparence et la simplicité

- Penser aux conditions d'utilisation non normales
  - Contrôles des plages de valeurs, cas limites, ...
  - Ne pas faire confiance aux données venant de l'utilisateur
  - Tester, tester, tester
- Pas de données cachées en dur :
  - Fichiers de configuration
- Éviter les cas particuliers dans le code

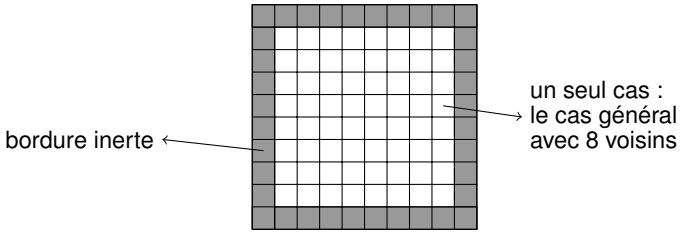
Règle de robustesse

- Exemple : les jeux de plateau
- Comment tester les voisins dans un 8x8 ?



Règle de robustesse

- Bonne solution : ajouter une bordure inerte (des cases vides par exemple)
- taille du jeu = 8x8
- taille du tableau = 10x10



Règle de représentation

Placer le savoir dans les données, afin d'obtenir des algorithmes disciplinés et robustes

- Prévoir de bonnes structures de données
- Si l'on utilise des structures, l'ajout d'une information (d'un champ...) ne modifie pas le code.

```
1  void minimize(State s[], int n_states, Transition t[]);
```

```
1  typdef struct{
2    State s[];
3    int n_states;
4    Transition t[];
5  } Automaton;
6
7  void minimize(Automaton* a);
```


Règle de la moindre surprise

Éviter les surprises dans la conception des interfaces

- Éviter les nouveautés inutiles
- Respecter les habitudes :

  - des programmeurs (pourquoi changer si `for(i=0;i<n;i++)` convient ?)

  - des utilisateurs (ne pas appeler `.txt` un fichier binaire)

Règle de silence

N'afficher des informations que si nécessaire

- Informations inutiles = pollution
- Les informations internes (debug) doivent pouvoir être désactivées (*e.g.* par macro).
- Cas exceptionnel : Un programme qui peut durer longtemps peut donner des signaux pour montrer qu'il n'est pas planté (ex : gros calcul qui affiche un progression...). Des logs ou des checkpoints sont aussi appropriés.

Règle de réparation

Réparer ce qui est possible, mais en cas de problème, faire échouer rapidement et clairement

- "Être tolérant avec ce qu'on reçoit, exigeant avec ce qu'on envoie." (J. Postel)
- Mais : à vouloir trop gérer les erreurs, on crée des usines à gaz.
- Ne pas hésiter à faire exit lors d'une erreur fatale
  - Erreur d'allocation du plateau pour un jeu de dames
  ⇒ Rien ne sert de continuer.
  - Erreur d'allocation du nom du joueur en fin de partie
  ⇒ Tant pis, on ne sauvegarde rien mais on termine normalement.

Règle d'économie

Préférer l'économie du temps de programmation à celle du temps machine

- Automatiser autant que possible
- Exemple : inutile de garder en cache la taille des chaînes de caractères
  - Dans l'immense majorité des cas, `strlen` suffit.
  - Mais il faut bien distinguer les cas où ça ne suffit pas.

Règle de génération

Éviter le travail manuel, écrire plutôt des programmes de génération (de programme, de données, etc)

- Exemples : `flex`, `bison`, `automake`, `doxygen`, etc
- Si un programme utilise un fichier d'entiers, écrire un autre programme pour générer des fichiers de tests pour le premier.

Règle d'optimisation

Créer des prototypes avant d'affiner ; rechercher un bon fonctionnement avant d'optimiser

- "On devrait repousser les petites améliorations de l'efficacité dans environ 97% des cas, car une optimisation prématurée est la racine de tout le mal." (C. A. R. Hoare)
- "Dans 90% des cas, la meilleure optimisation consiste à ne rien faire."
- Les compilateurs modernes optimisent le code mieux que nous en général avec l'option `-O2`.
- L'intuition est mauvaise conseillère. Il faut utiliser des outils de profilage.
- Il faut comprendre la machine et le programme avant toute optimisation.

Règle de diversité

Se méfier de la bonne solution unique

- En utilisant des interfaces propres, on laisse la possibilité d'utiliser un jour une autre implantation (plus rapide, mois gourmande, dans un autre langage, etc).

```
1  void foo(Automaton* a){
2    State* s=a->states[0];
3    if (s->control & FINAL){
4      /* ... */
5    }
6  }
```

```
1  void foo(Automaton* a){
2    State* s=get_state(a, 0);
3    if (is_final(s)){
4      /* ... */
5    }
6  }
```


Règle de simplicité

Concevoir des systèmes simples ; n'introduire de la complexité que si nécessaire

- "La simplicité est la sophistication suprême" (Léonard de Vinci)
- Pas d'abstraction inutile

  - Pas de tableau de hâchage générique si on ne manipule que des entiers
- Ne pas trop anticiper

  - Pas de paramètres inutiles, au cas où, plus tard, ...

Règle d'extensibilité

Concevoir en pensant à l'avenir, qui sera là plus tôt qu'on ne l'imagine

- Format de fichiers génériques
- Utilisation de numéros de version
- Exemples : gestion des encodages de caractères

Règle d'extensibilité

- Bonne solution : utiliser une bibliothèque qui les gère avec possibilité d'en ajouter, éventuellement par plugins.

```
1  /* This librairy is designed to manage
2     various implementations of
3     fputc for various encoding. */
4  #ifndef encodings_H
5  #define encodings_H
6
7  typdef int (*encoder) (int, FILE*);
8
9  void add_encoder(char* name, encoder f);
10 encoder get_encoder(char* name);
11 #endif
```


Plugins automatiques

- Lorsque des plugins possèdent tous les mêmes spécificités (mêmes symboles pour le linker) visant à enrichir une application, ces derniers peuvent être chargés automatiquement.

  - On déclare les types pointeurs de fonctions correspondants aux symboles à récupérer dans les `.so`.

  - On récupère tous les fichiers `.so` contenu dans un joli répertoire `plugins` avec `scandir` (en vérifiant une signature si la sécurité est importante).

  - On remplit un tableau de symboles (pointeurs de fonctions) récupérés avec `dlsym`.

  - Lors de l'utilisation d'une fonctionnalité, on recherche si elle est disponible dans le tableau des plugins chargés.

Exemple : `dc` avec plugins

- `dc` pour Desk Calculator est la fameuse calculatrice en Polonais inversé d'`Unix`.
- Toutes les opérations arithmétiques de `dc` dépilent un certain nombre d'arguments, calculent un résultat et empilent ce résultat.
- On peut imaginer un mécanisme de plugins automatiques recherchant toute les opérations disponibles pour l'application.
- Une opération est définie par :

  - Une fonction d'évaluation : `int eval(int* args);`
  `return args[0] + args[1];`

  - Une arité : `int arity(void);`
  `return 2;`

  - Un symbole : `char symbole(void);`
  `return '+';`

Et tout le reste...

- Coder et commenter préférablement en anglais
- Utiliser des formats lisibles, et si possible standards et ouverts
- Toujours penser à ceux qui reliront le code (même si c'est vous...)
- Penser à l'utilisateur qui n'a pas forcement les mêmes repères
- Respecter les spécifications données