Cours de C - IR1
2007-2008

Portabilité, maintenabilité
&
réutilisabilité

Sébastien Paumier

MCours.com

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 trucs exotiques comme `#pragma`
  - respecter les normes (`-ansi -Wall`)
  - cerner le code qui dépend du compilateur, du système ou de la machine
  - éviter les dépendances vers des bibliothèques non portables
  - faire des Makefile portables

Cerner le code variable

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

```
/**
 * System-dependent function that compares
 * file names.
 */
int fcompare(const char* a,const char* b) {
#ifdef WINDOWS_LIKE
/* case doesn't matter under windows */
return strcmp_ignore_case(a,b);
#else
return strcmp(a,b);
#endif
}
```

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
```

```
#ifndef WINDOWS_LIKE
#define PATH_SEPARATOR_CHAR '/'
#define PATH_SEPARATOR_STRING "/"
#else
#define PATH_SEPARATOR_CHAR '\\'
#define PATH_SEPARATOR_STRING "\\"
#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
```

compile sous Windows,
pas sous Linux

Dépendances au système

- 2 types de dépendances:
  - valeurs/types (séparateur de fichier `/` ou `\`)
  - comportements (casse des noms de fichiers)
- pour le 1er 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:

```
void** new_ptr_array(int n) {
    void** ptr=malloc(n*4);
    if (ptr==NULL) {
        /* ... */
    }
    return ptr;
}
```

sur une machine 64 bits, **`sizeof(void*)`** vaut 8

Tailles des types

- si on a besoin d'un type avec une taille en octets fixe, utiliser les constantes de **stdint.h**
- exemple: si on veut gérer Unicode non étendu, on a besoin d'un type non signé sur 2 octets

```
#include <stdint.h>

typedef uint16_t unichar;
```

Les sauts de lignes

- Windows: `\r\n`
- Linux: `\n`
- MacOS: `\r`
- il ne suffit pas que le programme soit portable
- est-ce que les données doivent l'être ?
  - comment 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

* (toute ressemblance avec un transparent existant serait forfuite et involontaire)

Makefile

- écrire des Makefile portables avec une variable qui dépend du système
- informations concernées:
  - compilateur à utiliser
  - options de compilation
  - répertoires (include, lib, etc)
  - commandes d'installation et de nettoyage
  - noms des sorties (exemple: `.exe` ou pas?)
  - etc.

Makefile

- exemple: compilation portable d'une bibliothèque

test sur une variable d'environnement pour savoir si on est sous Windows (à vérifier...)

définitions dépendant du système

```
ifeq ($(strip $(PATHEXT)),)
SYSTEM = linux-like
else
SYSTEM = windows
endif

CC = gcc
CFLAGS = -fPIC
OBJS = utf8.o

ifeq ($(SYSTEM),linux-like)
CLEAN = rm -f
OUTPUT = libutf8.so
else
CLEAN = del
OUTPUT = utf8.dll
endif
```

Makefile

instructions pour Linux

instructions pour Windows

```
all: $(OUTPUT)

%.o: %.c
    $(CC) -c $< $(CFLAGS)

ifeq ($(SYSTEM),linux-like)
$(OUTPUT): $(OBJS)
    $(CC) -shared $(OBJS) -o $(OUTPUT)
else
$(OUTPUT): $(OBJS)
    $(CC) -shared $(OBJS) -Wl,--export-all-symbols -o $(OUTPUT)
endif

clean:
    $(CLEAN) *.o
    $(CLEAN) $(OUTPUT)
```

Maintenabilité & réutilisabilité
ou
introduction au génie logiciel

Référence: *L'art de la programmation UNIX*, Eric S. Raymond, Vuibert

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

Proverbe étudiant

"Quand on est con, on est con"

Georges Brassens

Qu'est-ce que le GL ?

- ensemble de bonnes pratiques et de conseils à méditer
- 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.

*(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
- principe d'encapsulation en bibliothèques *boîtes noires*

Règle de clarté

préférer la clarté à l'intelligence

- penser à celui qui relira le code
- un algorithme très rusé a moins de chances d'être lisible qu'un classique
  - risque plus élevé de bugs
  - moins facile à entretenir
- style gourou à bannir absolument!!!

Règle de composition

concevoir des programmes à connecter à d'autres programmes

- éviter les programmes "dieu" qui font tout
- ne pas réinventer la roue:
  - utiliser les choses faites par ceux qui savent (vous gérez les `.pnm` ? utilisez `anytopnm` pour les autres formats)
  - utiliser les dépendances

Règle de séparation

séparer méthode et mécanismes;
séparer moteur et interfaces

- bonne pratique: ne rendre visible que les interfaces et pas les implémentations en utilisant `static`
- exemple: frontal GUI dissocié des tâches de fond (affichage et calcul=tâches différentes)
- 1 module=1 responsabilité

Règle 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
- ne pas lésiner sur les tests
- ne pas cacher les bugs

```
void foo(int t[],int n) {
if (n<=0) {
   /* should not happen */
   fprintf(stderr,"n<=0 in foo!\n");
   return;
}
/* ... */
}
```

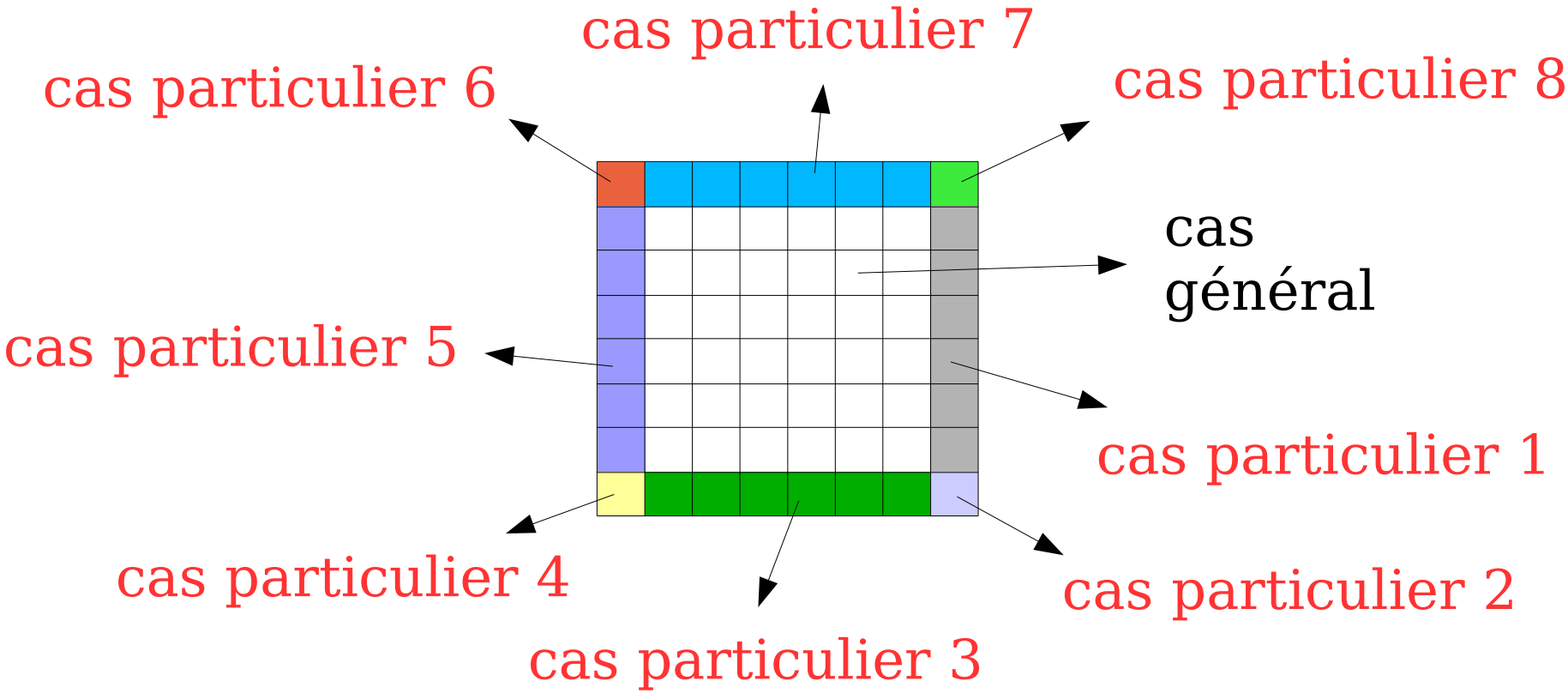
Règle de robustesse

engendrer de la robustesse par la transparence et la simplicité

- penser aux conditions d'utilisation non normales:
  - tester, tester, tester
- pas de données cachées en dur:
  - fichiers de configuration
- éviter les cas particuliers dans le code

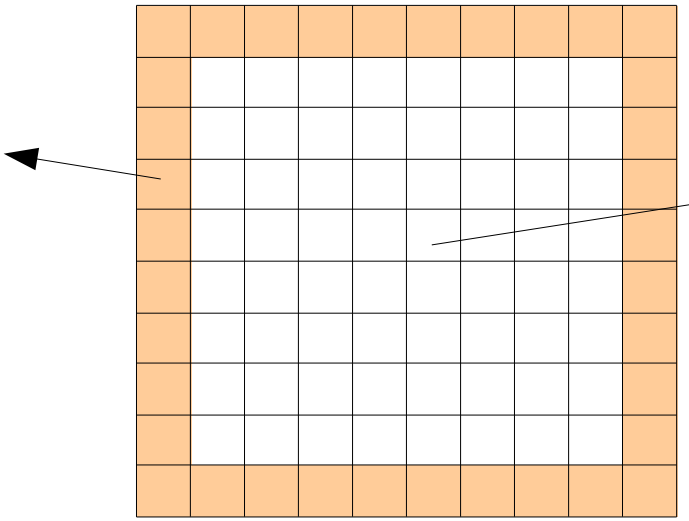
Règle de robustesse

- 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 ne modifie pas le code

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

```
typedef struct {
    State* s;
    int n_states;
    Transition* t;
} Automaton;
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 nécessaires

- informations inutiles=pollution
- les informations internes (debug) doivent pouvoir être désactivées
- contre-exemple: un programme qui peut durer longtemps peut afficher pour montrer qu'il n'est pas planté (ex: gros calcul)

Règle de réparation

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

- "*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

```
if (n<=0) {
    /* should not happen */
    fprintf(stderr,"n<=0 in foo!\n");
    return;
}
```

prévenir clairement

si l'erreur est fatale, ne pas hésiter à faire **exit**

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

Règle de génération

éviter le travail manuel; écrire plutôt des programmes de génération (de programmes, de données, etc.)

- exemples: **`flex`**, **`bison`**, **`automake`**
- si un programme utilise un fichier d'entiers, écrire un autre programme pour générer les fichiers de test du 1er

Règle d'optimisation

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

- "*l'optimisation prématurée est la source de tout mal* " (C.A.R. Hoare)
- "*dans 90% des cas, la meilleure optimisation consiste à ne rien faire*"
- intuition mauvaise conseillère
  - utiliser des outils de profiling

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 implémentation (plus rapide, moins gourmande, dans un autre langage, etc.)

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

```
void foo(Automaton* a) {
    State* s=get_state(a,0);
    if (is_final(s)) {
        /* ... */
    }
    /* ... */
}
```

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 hashtable générique si l'on ne manipule que des entiers
- ne pas trop anticiper
  - pas de param. 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

- formats de fichiers génériques
- utilisation de numéros de version
- exemple: gestion d'encodages

```
typedef enum {ASCII,UTF8,UTF16LE} Encoding;
typedef int (*encoder) (int,FILE*);
encoder get_fputc(Encoding encoding) {
if (encoding==ASCII) return fputc_ascii;
if (encoding==UTF8) return fputc_utf8;
if (encoding==UTF16LE) return fputc_utf16le;
return NULL;
}
```

→ pas évolutif

Règle d'extensibilité

- bonne solution: utiliser une bibliothèque qui les gère avec possibilité d'en ajouter, éventuellement dynamiquement

encodings.h :

```
/* This library is designed to
 * manage various implementations
 * of fputc for various encodings.
 * Defaults = "ASCII" "UTF8" */
#ifndef encodings_H
#define encodings_H
#include <stdio.h>

typedef int (*encoder) (int,FILE*);

void add_encoder(char* name,encoder f);
encoder get_encoder(char* name);
#endif
```

interface très simple

Règle d'extensibilité

début de encodings.c :

```
#include "encodings.h"
#include "utf8.h"
#include <stdlib.h>
#include <string.h>

static int init();
static int n=init();
static int init() {
   n=0;
   add_encoder("ASCII",fputc);
   add_encoder("UTF8",fputc_utf8);
   return n;
}

#define ENC_MAX 256
typedef struct {
   char* name;
   encoder f;
} Encoding;
static Encoding enc[ENC_MAX];
```

initialisation automatique au chargement du code, pour mettre des encodages par défaut

implémentation cachée

Règle d'extensibilité

fin de encodings.c :

```
void add_encoder(char* s,encoder f) {
if (n==ENC_MAX) {
   fprintf(stderr,"Too much encodings!\n");
   exit(1);
}
enc[n].name=strdup(s);
enc[n].f=f;
n++;
}

encoder get_encoder(char* name) {
int i;
for (i=0;i<n;i++) {
   if (!strcmp(enc[i].name,name)) {
      return enc[i].f;
   }
}
return NULL;
}
```

est-ce la peine de gérer les doublons ?

pas la peine d'optimiser en utilisant des constantes au lieu de chaînes

Et tout le reste...

- coder et commenter en anglais
- utiliser des formats lisibles, et si possible standards et ouverts
- toujours penser à ceux qui reliront le code
- penser à l'utilisateur qui n'a pas forcément les mêmes repères
- respecter les spécifications données!