CM 18 - PORTABILITÉ ET LIMITATION DU C

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

PORTABILITÉ

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
  - Eviter les trucs exotiques comme `#pragma`
  - Respecter les normes (`-std=c99 -Wall`)
  - Cerner le code qui dépend du compilateur, du système ou de la machine
  - Eviter 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

▷ Le respect de la casse est importante sous certains systèmes

motor.c

```
#include ”Motor.h”
/* ... */
```

motor.h

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

▷ ⚠Cet exemple compile sous Windows mais pas sous Linux

Dépendances au système

▷ Il existe deux types de dépendances

  ▷ Vis-à-vis de valeurs ou types spécifiques (e.g., les séparateur de fichier – '\' ou '/')

  ▷ Vis-à-vis de comportements (e.g., le respect strict de la casse)

▷ Le premier type de dépendances peut se gérer à l'aide de macros conditionnelles (i.e., `#ifdef`)

▷ Pour le second type, ce n'est pas toujours suffisant

DÉPENDANCES AU SYSTÈME

- Toute dépendance au système doit être explicitée
- Exemple: La limitation des noms de fichiers
  - Mauvaise solution: utiliser une constante arbitraire
  - Bonne solution: Utiliser la constante `FILENAME_MAX` définie dans **stdio.h** qui est adaptée au système courant

TAILLE DES TYPES

- Il faut anticiper les différences entre architectures en utilisant
  - Les constantes de `limits.h`
  - L'opérateur `sizeof`

badBoy.c

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

goodBoy.h

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

TAILLE DES TYPES

▷ Si on a besoin d'un type avec une taille en octets fixe, il faut utiliser les constantes de `stdint.h`

▷ Exemple: si on souhaite gérer l'encodage Unicode non étendu, nous avons besoin d'un type non signé sur 2 octets

```
#include <stdint.h>

typedef uint16_t unichar;
```

MAKEFILE PORTABLE

- Dans la mesure du possible, il faut écrire des Makefile portables utilisant une variable dépendant du système
- Les informations concernées sont
  - Le compilateur à utiliser
  - Les options de compilation
  - Les répertoires de travail (include, lib, ...)
  - Les commandes d'installation et de nettoyage
  - Les noms de sorties (e.g., **.exe** ou pas ?)
  - ...

MAKEFILE PORTABLE

▷ Exemple: compilation portable d'une bibliothèque

```
ifeq ($(strip $(PATHEXT)),) # test if PATHEXT is empty
SYSTEM=linux-like  # strip removes leading, trailing and duplicated
else               # whitespace
SYSTEM=windows
endif
...
ifeq($(SYSTEM),linux-like)
CLEAN=rm-f
OUTPUT=libutf8.so
else
CLEAN=del
OUTPUT=utf8.dll
endif
```

▷ **PATHEXT** est une variable d'environnement contenant la liste des extensions de fichiers exécutables

MAKEFILE PORTABLE

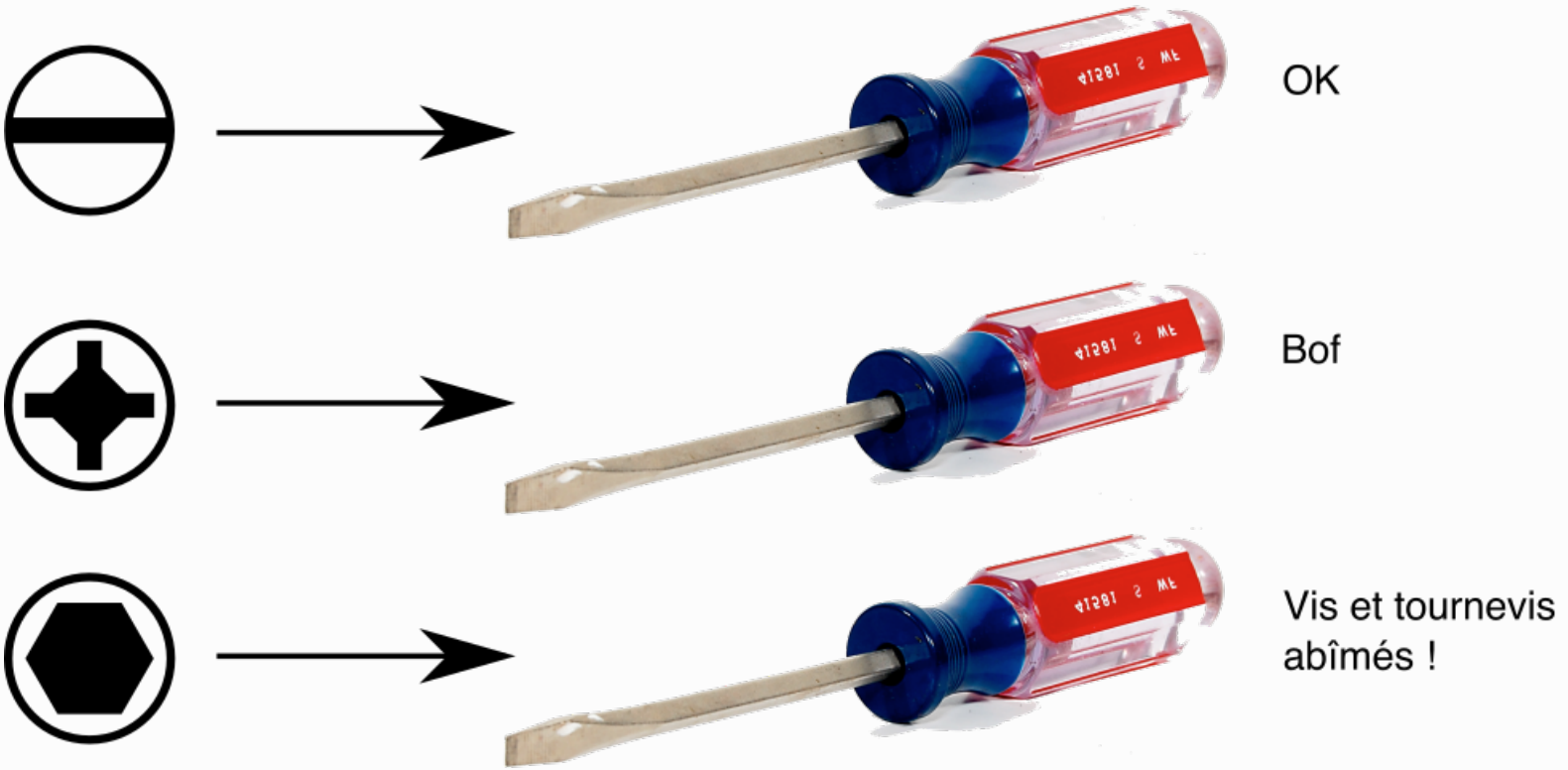
▷ Exemple: compilation portable d'une bibliothèque

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

LIMITATION DU C

PRENDRE UN OUTIL ADAPTÉ

▷ Quand on bricole, on doit toujours prendre un outil adapté



▷ Quand on programme, c'est pareil

▷ Cela donne accès à la machine physique au prix d'un plus grand risque d'erreurs

▷ Exemple: débordement des tableaux

```
float get1(float t[], int n, int size){
  if(t==NULL){
    fprintf(stderr,"NULL array in get\n");
    exit(EXIT_FAILURE);
  }
  if(n<0||n>=size){
    fprintf(stderr,"Index %d out of bounds in get\n",n);
    exit(EXIT_FAILURE);
  }
  return t[n];
}
```

▷ Même exemple mais avec une sortie moins sauvage

```
int get2(float t[], int n, int size, float* res){
  if(t==NULL || n<0 || n>=size){
    return 0;
  }
  (*res)=t[n];
  return 1;
}
```

▷ Générification de ma fonction pour éviter d'avoir une fonction par type

```
int get3(void* t, int n, int size, int size_n, void* res){
  if(t==NULL || n<0 || n>=size){
    return 0;
  }
  char* foo=(char*)t;
  memcpy(res,foo+n*size_n,size_n);
  return 1;
}
```

▷ The memcpy() function copies len bytes from memory area src to memory area dst

▷ Version autorisant `t[i]=t[i]`

```
int get4(void* t, int n, int size, int size_n, void* res){
  if(t==NULL || n<0 || n>=size){
    return 0;
  }
  char* foo=(char*)t;
  memmove(res,foo+n*size_n,size_n);
  return 1;
}
```

▷ The memcpy() function copies len bytes from memory area src to memory area dst. If src and dst overlap, behavior is undefined. Applications in which src and dst might overlap should use memmove(3) instead

▷ Version évitant le débordement de la zone **`res`**

```
int get5(void* t, int n, int size, int size_n, void* res,
                                                int size_res){
  if(t==NULL || n<0 || n>=size || size_res<size_n){
    return 0;
  }
  char* foo=(char*)t;
  memmove(res,foo+n*size_n,size_n);
  return 1;
}
```

▷ Version évitant le débordement de la zone **res**

```
int get5(void* t, int n, int size, int size_n, void* res,
                                                int size_res){
  if(t==NULL || n<0 || n>=size || size_res<size_n){
    return 0;
  }
  char* foo=(char*)t;
  memmove(res,foo+n*size_n,size_n);
  return 1;
}
```

▷ Si je me trompe de paramètres, cela ne marchera pas ...

▷ Oui ... tout comme dans la version **get1**

▷ Version évitant le débordement de la zone `res`

```
int get5(void* t, int n, int size, int size_n, void* res,
                                                int size_res){
  if(t==NULL || n<0 || n>=size || size_res<size_n){
    return 0;
  }
  char* foo=(char*)t;
  memmove(res,foo+n*size_n,size_n);
  return 1;
}
```

▷ Toutes ces modifications n'ont servi à rien alors ?

  ▷ Oui

▷ Version évitant le débordement de la zone **`res`**

```
int get5(void* t, int n, int size, int size_n, void* res,
                                                int size_res){
  if(t==NULL || n<0 || n>=size || size_res<size_n){
    return 0;
  }
  char* foo=(char*)t;
  memmove(res,foo+n*size_n,size_n);
  return 1;
}
```

▷ Comment fait-on en C ?

  ▷ On ne fait pas

C - LANGAGE COMPILÉ

- Un programme C existe en 2 versions
  - Les sources
  - Le binaire exécutable
- Sans les sources, le programme ne pourra pas être conservé dans le temps
  - Problème inexistant dans les langages interprétés (e.g., perl, python)

C - PRÉCISION LIMITÉE

▷ La taille des nombres est limitée et conduit à des problèmes d'arrondi

```c
#include <stdio.h>
#include <stdlib.h>
int main(void){
  int i;
  float sum=0.f;
  for(i=0;i<999999997;i++){
    sum=sum+0.001f;
  }
  printf("sum=%f\n",sum);
  return EXIT_SUCCESS;
}
```

```
$>time ./a.out
sum=32768.000000
real    0m3.208s
user    0m3.161s
sys     0m0.022s
```

▷ Solution : outil de calcul formel (e.g. mupad ou mathlab)

C - Typage statique et faible

▷ Toute variable a un type unique

  ▷ Nécessitant des conversions explicites

▷ Dans le cadre d'un typage dynamique, la conversion est faite automatiquement quand c'est nécessaire (e.g. en bash)

▷ Toute variable a un type déclaré

▷ Dans le cadre d'un typage fort, le type est déterminé automatiquement en fonction de l'utilisation (e.g. inférence de types en Caml)

C - OPÉRATEURS FIGÉS

▷ Il n'existe pas de surcharge d'opérateurs: on ne peut pas écrire **a+b** pour deux nombres complexes **a** et **b**

▷ Ce mécanisme existe en C++

```
class Complex{
  public:
    float re, im;
    Complex(){
      re=0;
      im=0;
    }
    Complex(float a, float b){ re=a; im=b;}
    friend Complex operator+(const Complex &a, const Complex &b){
      Complex c;
      c.re=a.re+b.re;
      c.im=a.im+b.im;
      return c;
    }
};
```

C - PAS D'EXCEPTIONS

- ▷ Seule une gestion manuelle des erreurs est possible
- ▷ La propagation d'erreur est lourde à mettre en oeuvre

```
int fprintf_utf8(FILE* f, unichar* s){
  while(*s){
    if(!fputc_utf8(*s, f)){
      return 0;
    }
    s++;
  }
  return 1;
}
int save_HTML(FILE* f, unichar* s){
  if(!fprintf_utf8(f,begin_tag)){return 0;}
  if(!fprintf_utf8(f,s)){return 0;}
  if(!fprintf_utf8(f,end_tag)){return 0;}
  return 1;
}
```

C - PAS D'EXCEPTIONS

▷ Solution: le mécanisme d'exception (e.g., C++, Java, ...)

```
void fprintf_utf8(FILE* f, unichar* s){
  while(*s){
    if(!fputc_utf8(*s, f)){
      throw IOException;
    }
    s++;
  }
}
int save_HTML(FILE* f, unichar* s){
  try{
    fprintf_utf8(f,begin_tag);
    fprintf_utf8(f,s);
    fprintf_utf8(f,end_tag);
    return 1;
  } catch (IOException exception){
    return 0;
  }
}
```

C - NOMS GLOBAUX

▷ Il n'y a pas d'espaces de nommage paramétrables

algo.h

```
#ifndef _algo_H
#define _algo_H

typedef struct noeud{
  int val;
  struct noeud *g, *d;
} ARBRE;

typedef struct file{
  struct file* premier;
  struct file* dernier;
}FILE;

void largeur(ARBRE* a);
#endif
```

nommage.c

```
#include <stdlib.h>
#include <stdio.h>
#include "algo.h"
int main(void){
  /*...*/
  return EXIT_SUCCESS;
}
```

```
$>gcc -std=c99 nommage.c
In file included from nommage.c:2:
algo.h:12:conflicting types for 'FILE'
/usr/include/stdio.h:159:previous
                    declaration of 'FILE'
```

C - NOMS GLOBAUX

▷ Il ne peut également y avoir qu'une seule fonction `main`

▷ Si une bibliothèque `foo` contient une fonction `main` de test oubliée, plus rien ne fonctionne

▷ Solution: avoir un contrôle sur la portée des symboles (e.g., `foo.toto()`)

▷ Possible en programmation Objet, en encapsulant des choses ou en utilisant des espaces de noms (notion de package)

C - n'est pas un paradigme objet

▷ Le C n'offre pas

▷ Des contraintes de visibilité propres

▷ D'héritage permettant le partage de propriétés (e.g., nombre de sommets/arêtes entre carré et trapèze)

▷ De polymorphisme permettant de définir des fonctions de même nom mais avec des définitions dépendant du contexte (e.g. calcul de l'aire d'une forme)

DOGGY BAG

TO TAKE AWAY ...

▷ Le langage C n'est pas un langage adapté à tous les besoins

▷ Quand les caractéristiques du langage ne conviennent plus, il faut savoir en changer

Questions?