Aix-Marseille Université — Licence Info – Semestre 4
Faculté des Sciences de Luminy — Jeudi 9 Avril 2020

Programmation C et Système — Contrôle

Rendre le fichier-texte de réponse sur Ametice avant 18h

Ce quiz prend pour prétexte un programme `hello` dont voici le comportement :
Les exécutions suivantes affichent sur la **sortie standard** et terminent avec un **status de succès** :

```
$ ./hello -help
Usage: ./hello -help|-list|[LANGUAGE]
```

```
$ ./hello -list
en
fr
ja
```

```
$ ./hello
hello
$ ./hello en
hello
$ ./hello fr
bonjour
$ ./hello ja
konnichiwa
```

Les exécutions suivantes affichent sur la **sortie d'erreur** et terminent avec un **status d'échec** :

```
$ ./hello xx
no language 'xx'
```

```
$ ./hello xx xx
Usage: ./hello -help|-list|[LANGUAGE]
```

Question 1. Compléter ci-dessous pour que `echo` affiche le status de `./hello xx`.
Donner une valeur possible pour ce status :

```
$ ./hello xx
no language 'xx'
$ echo $?
1
```

Question 2. Si on exécute le même `echo` à nouveau, quelle valeur est affichée ? Justifier.

```
un nouveau echo $? affichera 0
car c'est le status d'erreur du echo précédent
```

Question 3. Pourquoi est-ce une mauvaise idée d'utiliser un status > 128 ?

```
Sous bash et sh, lors d'une terminaison anormale par le signal s,
$? contient la valeur 128+s, indistingable d'un status 128+s.
```

Question 4. Les commandes ci-dessous sont exécutées à la suite l'une de l'autre.
Compléter ces commandes pour obtenir les sorties affichées :

```
$ rm output.log
$ ./hello fr 1> output.log
$ cat output.log
bonjour
$ ./hello xx 2> output.log
$ cat output.log
no language 'xx'
$ ./hello en 1>> output.log
$ cat output.log
no language 'xx'
hello
$ ./hello xx xx 2>> output.log
$ cat output.log
no language 'xx'
hello
Usage: ./hello -help|-list|[LANGUAGE]
```

Toute la suite s'intéresse à l'implémentation en C du programme.

Question 5. Quelles directives d'inclusion d'entête écrire pour pouvoir utiliser :

(1) la fonction `exit()` ?
(2) la fonction `printf()` ?
(3) le type `bool`, les macros `true` et `false` ?
(4) les fonctions sur chaînes en `strxxx()` ?

```c
#include <stdlib.h>  // (1)
#include <stdio.h>   // (2)
#include <stdbool.h> // (3)
#include <string.h>  // (4)
```

Question 6. En utilisant une fonction de librairie, et sans utiliser de conditionnelle, compléter le prédicat `String_equals()` qui teste l'égalité de deux chaînes `string` et `other` :

```c
bool String_equals (char const string[], char const other[]) {
  return strcmp (string, other) == 0;
}
```

Question 7. Si le prédicat `String_equals()` est dans le module `String`, compléter l'entête `String.h` suivant avec sa garde et la directive d'inclusion nécéssaire :

```c
#ifndef STRING_H
#define STRING_H
#include <stdbool.h>
bool String_equals (char const string[], char const other[]);
#endif
```

Question 8. Écrire une fonction de test `StringTest_equals()` testant le prédicat `String_equals()` en utilisant deux assertions et trois variables tableaux de caractères :

```c
void StringTest_equals (void) {
   char str1[]= "string", str2[]= "string", str3[]= "strong";
   assert (    String_equals (str1, str2));
   assert ( !  String_equals (str1, str3));
}
```

Question 9. Dans le test `StringTest_equals()` de la question précédente, pourquoi ne pas utiliser directement des constantes littérales ou ne pas utiliser seulement deux variables ?

```
Afin d'avoir deux valeurs égales à des adresses différentes,
et faire échouer le test passant de String_equals() si la fonction
compare les adresses au lieu de comparer les valeurs.
Deux constantes littérales de même valeur peuvent avoir
la même adresse ou pas, à la discrétion du compilateur.
```

Question 10. Compléter la définition du type `Pair` qui est l'agrégat de deux pointeurs `key` et `value` sur (chaînes de) caractères constants :

```c
typedef struct Pair {
  char const * key, * value;
} Pair ;
```

Question 11. En utilisant la syntaxe C99 des *compound literals* pour les structures, compléter la fonction `Pair_make()` qui fabrique et retourne une paire à partir d'une clé `key` et d'une valeur `value`. Ces chaînes ne sont pas dupliquées :

```c
Pair Pair_make (char const key[], char const value[]) {
  return (Pair) { .key= key, .value= value };
}
```

Question 12. Compléter la fonction `PairArray_printKeys()` qui affiche sur le flux `file` toutes les clés d'un tableau `array` de `n` paires, à raison d'une clé par ligne.

```
void
PairArray_printKeys (Pair const array[], int n, FILE * file) {
  for (int k= 0; k < n; k++) {
    fprintf (file, "%s\n", array[k].key);
  }
}
```

Question 13. Compléter la fonction `PairArray_keyIndex()` qui recherche une clé `key` dans un tableau `array` de `n` paires. On retourne l'index de la première occurrence trouvée, où `-1` si la clé n'est pas trouvée :

```
int
PairArray_keyIndex (Pair const array[], int n, char const key[]) {
  for (int k= 0; k < n; k++) {
    if (String_equals (key, array[k].key))
      return k;
  }
  return -1;
}
```

Remarque : Les deux questions suivantes concernent la fonction `main()`.

Question 14. Dans le programme principal, on utilise un tableau `hellos` de `HELLO_COUNT` paires pour associer des langues et des messages de salutations. Définir la macro `HELLO_COUNT` et initialiser le tableau `hellos` pour qu'il corresponde à nos exemples d'éxecution en première page :

```
#define HELLO_COUNT 3

int main (int argc, char * argv[]) {
  Pair hellos [HELLO_COUNT]= {
    Pair_make ("en", "hello"),
    Pair_make ("fr", "bonjour"),
    Pair_make ("ja", "konnichiwa")
  };
  // main() continues in next question
```

Question 15. Dans la suite du programme principal, on suppose que l'on dispose d'une fonction `Hello_checkArgs()` qui nous débarrasse des cas des arguments `-help` et `-list`, ainsi que du cas des arguments trop nombreux, en traitant ces cas et en sortant du programme. Si la fonction ne termine pas le programme, elle retourne la chaîne de la langue spécifiée en ligne de commande, ou à défaut `"en"` si aucune n'est spécifiée. Continuer le programme principal pour qu'il traite le cas exceptionnel de la langue inconnue et le cas régulier de la langue connue :

```
  // main() continues from previous question
  char * lang= Hello_checkArgs (argc, argv, hellos, HELLO_COUNT);
  int index= PairArray_keyIndex (hellos, HELLO_COUNT, lang);
  if (index == -1) {
    fprintf (stderr, "no language '%s'\n", lang);
    exit (1);
  }
  fprintf (stdout, "%s\n", hellos[index].value);
  return 0;
}
```

Remarque : Les trois questions suivantes concernent la fonction `Hello_checkArgs()`.

Question 16. On se donne la fonction `Hello_printUsage()` ci-dessous. Compléter le premier tiers de la fonction `Hello_checkArgs()` pour qu'elle nous débarrasse du cas des arguments `-help` et `-list` en les traitant et en sortant du programme :

```
void
Hello_printUsage (char const cmd[], FILE * file) {
  fprintf (file, "Usage: %s -help|-list|[-lang LANGUAGE]\n", cmd);
}
```

```
char *
Hello_checkArgs(int argc, char* argv[], Pair const hellos[], int n){
  if (argc == 1+1 && String_equals (argv[1], "-help")) {
    Hello_printUsage (argv[0], stdout);
    exit (0);
  }
  if (argc == 1+1 && String_equals (argv[1], "-list")) {
    PairArray_printKeys (hellos, n, stdout);
    exit (0);
  }
  // Hello_checkArgs() continues in next question
```

Question 17. Compléter le deuxième tiers de la fonction `Hello_checkArgs()` pour qu'elle nous débarrasse du cas des arguments trop nombreux en traitant ce cas et en sortant du programme :

```
  // Hello_checkArgs() continues from previous question
  if (argc > 1+1) {
    Hello_printUsage (argv[0], stderr);
    exit (1);
  }
  // Hello_checkArgs() continues in next question
```

Question 18. Étant débarrassé de tous les cas d'usage précédents, compléter le dernier tiers de la fonction `Hello_checkArgs()` pour qu'elle retourne la langue spécifiée par l'utilisateur sur la ligne de commande, ou `"en"` à défaut si auncune langue n'est spécifiée. Pour ce faire, utiliser l'opérateur conditionnel ternaire.

```
  // Hello_checkArgs() continues from previous question
  return (argc == 1+1) ? argv[1] : "en";
}
```

Question 19. Dans un `Makefile`, quelle variable désigne :

(1) les options de compilation du compilateur C ?
(2) la cible de la règle courante ?
(3) la dépendance la plus à gauche de la règle courante ?
(4) l'ensemble des dépendances de la règle courante ?

```
(1) $(CFLAGS)
(2) $@
(3) $<
(4) $^
```

Question 20. Si le répertoire courant regroupe les fichiers `.c` et `.h` d'un programme, et que chaque fichier `.c` doit générer un fichier `.o`, quelle ligne de commande permet de lister les règles de dépendances de tous les fichiers `.o` dans un fichier `depend` :

```
clang -MM *.c > depend
clang -MM *.c | tee depend # si on veut afficher aussi
```