CSC2 - Correction

Q1. Mauvaise idée, polynome n'est pas modifié, il faut le qualificateur const (danger d'intégrité).

float eval(**const** struct polynome* p,float x);

rem. On peut accepter un passage par copie de structure. C'est valide pour certains projet. Mais cela contredit les règles du cours: toute structure est passée par pointeur (évite la copie inutile d'une structure > 4 octets).

Q2.
Fonction calculant la valeur du polynome pour un nombre x quelconque.

Prérequis:
Un pointeur (const) vers une structure polynome non NULL.
Une valeur flottante quelconque.

Garantie:
Renvoie la valeur du polynome (flottant) appliquée au point x.

Q3.

void calcul_racines(**const** struct polynome* p,struct racine* r);

Q4.

Mauvaise idée: mélange entre valeur et cas particulier, problème de lisibilité.
De plus: Impossible de différencier avec la racine double -1 (polynome (x+1)^2 par ex).

Q5.

Fonction calculant les deux racines réelles du polynome.

Prérequis:
Un pointeur (const) vers une structure polynome non NULL. Le discriminant du polynome doit être >= 0.
Un pointeur (non const) vers une structure racine non NULL.
Un polynome du second degre (c.a.d dont le coefficient a (celui des x^2) ne vaut pas 0).

Garantie:
La structure racine est mise à jour avec les 2 racines du polynome.
La fonction eval appliquée aux deux valeurs des racines trouvées doit retourner 0

Q6.

```
void calcul_racines(const struct polynome* p,struct racine* r)
{
  assert(p!=NULL);
  assert(r!=NULL);
  assert(abs(p->a)>=1e-5);

  float delta=p->b*p->b-4*p->a*p->c;
  assert(delta>0);

  r->r0=(-p->b+sqrt(delta))/(2*p->a);
  r->r1=(-p->b-sqrt(delta))/(2*p->a);

  assert(fabs(eval(p,r->r0))<1e-5);
  assert(fabs(eval(p,r->r1))<1e-5);
}
```

rem. Ne pas oublier les assertions. Attention à la comparaison entre flottants (pas d'égalité stricte) Pas besoin de différencier les cas Delta>0 et Delta valant 0. Le calcul converge vers les même valeurs. Le cas a proche d'être 0 est plus gênant numériquement.

Q7.

int discriminant_est_positif(**const** struct polynome* p);

Q8.

```
int main()
{
  struct polynome p={1,2,-1};
  struct racine r={0,0};

  if(discriminant_est_positif(&p))
  {
     calcul_racines(&p,&r);
     printf("poly: %fx^2+%fx+%f=0 : (%f,%f)\n",p.a,p.b,p.c,r.r0,r.r1);
  }
  else
     puts("Disciminant negatif");

  return 0;
}
```

Q9.

Au minimum 2 tests (un dans chaque cas)

```
struct polynome p={1,2,-1};
if(discriminant_est_positif(&p) != 1)
  puts("Erreur");

struct polynome p2={1,0,1};
if(discriminant_est_positif(&p) != 0)
  puts("Erreur");
```

Q10.

1. Faux, on inclue l'ensemble des en-têtes. Le préprocesseur ne fait qu'un "copié-collé" du fichier.
2. Vrai, il s'agit d'une pratique améliorant la lisibilité. Ex. Habitant[France]=55 est plus lisible que Habitant[3]=55
3. Faux, cela rend généralement le code moins lisible. Après compilation, il n'y a pas de différence au niveau du code assembleur entre le fait de l'écrire sur une ligne ou deux. Bonne pratique: une seule instruction par ligne.
4. Faux, git est logiciel de gestion de version tout comme svn.
5. Faux, un freeware est un logiciel gratuit donc le code est propriétaire et n'est pas fourni.
6. Faux, les assert sont éliminées lors de la compilation en mode release (macro NDEBUG)
7. Vrai, cela fait partie des bonnes pratiques de programmation (développement par tests). Permet de cadrer les entrees/sorties de la fonction avant d'écrire son code. Permet de la tester directement.
8. Vrai, un pointeur n'est qu'une adresse, peut importe le type. On peut donc caster tout type de pointeur en void*. Cela peut être utile pour réaliser des fonctions génériques. (Il faut pouvoir le caster à nouveau dans le type d'origine avant de le déférencer par contre).
9. Faux. Les caches sont de tailles limités, il permettent l'optimisation pour les accès mémoires. De base, le code est situé en RAM, pas dans le cache.
10. Faux, le .o est le fichier objet obtenu après compilation. La sortie du préprocesseur reste du code C plus simple à analyser pour le compilateur (les macros sont remplacées).

Q11.
4,7 // v3 contient la copie du contenu de v0
4,7 , 0,1 //p1 pointe sur v0 (égalité de pointeurs), v1 n'est pas modifié
7,7 //p0 pointe sur v0, il modifie donc la valeur de v0
indéterminé, p4 pointe sur v4, mais v4 est une variable temporaire à la boucle.
1 // v0 contient 2 entiers contigus en mémoire. Adressage identique à un tableau de 2 entiers.

rem. La copie de structure est tout à fait licite (voir cours).
Avant dernière question: 1,2 accepté à condition de bien expliquer le problème (espace mémoire probablement non modifié, mais incertain).

Q12.

p1. ligne 7 et 9: utilisation d'énumération plutôt que des entiers dans les indices de tableau
p2. ligne 1: passage de la structure en **const** car non modifiée.
p3. ligne 3 et 4: variables trop globales et non de variable (t) non significatif. Déclarer k et t avant la boucle for, renommer t en population_totale par exemple.
p4. 12 et 14: réutilisation de la variable pays[k].climat. Utiliser une variable intermédiaire. Ex. struct type_climat* climat_courant=&pays[k].climat;

rem.
k++ ou ++k n'est qu'une question de d'habitude en C. L'un n'est pas moins lisible que l'autre.
a+=b n'est pas forcément moins lisible que a=a+b. C'est une question d'habitude.
Ne pas perdre de temps, sur des questions de détails de dénomination de variable, ou d'affichage de contenu de printf. La seule variable vraiment très mal nommée était t.

Q13.

p1. Correct
p2. l.6, égalité de flottants: utiliser abs(t-24,3)<epsilon
p3. l.10, tableau.pays est un pointeur. On passe ici l'adresse du pointeur, non correct. La ligne 9 est correct.
p4. l.9, on passe une struct et non l'adresse de cette struct en argument.
p5. l.8, on déclare un pointeur et non une structure. Pas d'espace mémoire reservé.

rem. Ne pas oublier la valeur absolue sur la comparaison entre flottants.

Q14.

```
/** climat_froid: Climat tel que la temperature min soit la plus basse
//
// Prerequis:
//   un pointeur constant vers un pays_info (non NULL)
// Garanties:
//   un pointeur vers le climat dont la temperature min est plus basse
//   que toutes les autres (pointeur non NULL)
*/
```

rem. L'aspect constant ou non dépend de l'application. pointeur non constant également accepté.

Q15.

```
const struct type_climat* climat_froid(const struct pays_info* pays)
```

ou bien

```
struct type_climat* climat_froid(struct pays_info* pays)
```

rem. Par contre, mix des deux (retour constant à partir d'un pointeur non constant) non accepté.

Q16.

```
//Algorithme:
// Initialise temperature minimale a +inf
// Parcours l'ensemble des pays
//
//   Si temperature courante > temperature minimale
//      temperature minimale = temperature courante
//      stocke temperature courante dans pointeur climat
//
// Retourne pointeur climat
```

rem.
Un algorithme n'est pas un code. On ne doit pas voir apparaitre pays[k].climat par exemple.
Il faut préciser comment vous faites la recherche du min (pas uniquement dire: on recherche le min).

Q17.

```
  assert(pays!=NULL);
  const struct type_climat* le_plus_froid=NULL;
  float temp_min_courante=+999.9f;

  int n_pays=0;
  for(n_pays=0;n_pays<NBR_MAX_PAYS;n_pays++)
  {
     const struct type_climat* climat_courant=&pays[n_pays].climat;
     float temp_courante=climat_courant->temperature.min;

    if(temp_courante<temp_min_courante)
    {
       temp_min_courante=temp_courante;
       le_plus_froid=climat_courant;
    }
  }

  assert(type_climat!=NULL);

  return le_plus_froid;
```

rem. Il est possible de commencer temp_min_courante avec pays[0].
Si le passage d'argument est du type tableau (et non pointeur), alors il n'y a pas de vérification d'assertion: un tableau statique ne peut pas être NULL.
Attention à ne pas retourner l'adresse d'une variable locale à la fonction.

Q18.

gcc -c p0.c -g -Wall -Wextra
gcc -c p1.c -g -Wall -Wextra
gcc -c p2.c -g -Wall -Wextra
gcc p0.o p1.o p2.o -o p0

rem. Egalement acceptable
gcc -c p0.c p1.c p2.c
gcc p0.o p1.o p2.o -o p0

ou bien
gcc p0.c p1.c p2.c - o p0

Q19.

p0.o p1.o p2.o p0

rem.
Il n'y a pas de p0.exe p1.exe par défaut sous Linux.

Q20.

p3.o p1.o p4.o p0 p3

rem.
p0 n'est pas éliminé par le make clean.

Q21

p5.o p6.o p0 p3 a.out p6

rem. La ligne gcc de la compilation pour p5 est écrite en dure. Il n'y a pas de -o p5, donc le nom par défaut est a.out.