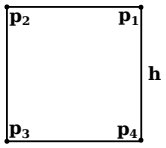
# PROG2 : Programmation avancée et C++

Examen du 11 mai 2022 (2 h)

Tous documents interdits à l'exception d'un A4 recto verso manuscrit de votre main. Calculatrices, téléphones, ordinateurs et montres connectées interdites. La correction tiendra compte de la qualité de l'argumentaire et de la présentation. Un code illisible et/ou incompréhensible est un code faux.

★ **Exercice 1 :** Classes `Point` et `Carre` (3pts).

Écrivez une classe `Point` permettant de représenter un point **constant** du plan, sous forme de deux doubles $x$ et $y$. Écrivez une classe `Carre` permettant de représenter un carré **constant** dans le plan. La classe `Carre` sera dotée de deux constructeurs. L'un prendra un point (correspondant à $p_1$ dans la figure ci-contre) ainsi que longueur du coté du carré, tandis que l'autre prendra deux points opposés $p_1$ et $p_3$.




Vous doterez votre classe `Carre` d'une méthode `aire()` ainsi que de la méthode `translate()` permettant de calculer le carré résultant de l'application de la translation donnée en paramètre sous forme de deux double $x$ et $y$. Vous doterez aussi votre classe de l'opérateur permettant de trouver un point représentant l'angle $p_2$ du carré $c$ avec la notation `c[2]`, ainsi que l'opérateur permettant d'afficher un carré sous forme textuelle.

**Réponse**

```cpp
#include <iostream>
#include <stdexcept>

class Point {
public:
    const double x,y;
    explicit Point(double x_, double y_) : x(x_), y(y_){}
};
std::ostream& operator<<(std::ostream& out, const Point& p) {
    out << "Point(" << p.x << ", " << p.y<<")";
    return out;
}

class Carre {
    const Point p1;
    const double h;
public:
    Carre(Point p1_, double h_) : p1(p1_), h(h_) {}
    Carre(Point p1_, Point p3) : p1(p1_), h(p1.x-p3.x) {
        if (p1.y-p3.y != h)
            throw std::invalid_argument("Provided p1 and p3 do not form a square.");
        // It's possible to make it work if other opposed angles are provided, but that's not the po
    }
    double aire() const {
        return h*h;
    }
    double hauteur() const { return h; }
    Carre translate(double x, double y) const {
        return Carre(Point(p1.x + x, p1.y + y), h);
    }
    Point operator[](int i) const {
        switch (i) {
        case 1: return p1;
        case 2: return Point(p1.x - h, p1.y);
        case 3: return Point(p1.x - h, p1.y - h);
        case 4: return Point(p1.x    , p1.y - h);
        default: throw std::out_of_range("You can only access the points from p1 to p4");
        }
    }
};
std::ostream& operator<<(std::ostream& out, const Carre& c) {
    out << "Carre(" << c[1] << ", " << c.hauteur()<<")";
    return out;
}

```

```
46 int main() {
47     auto c1 = Carre(Point(1,2), 4);
48     std::cout << "c1: " << c1 <<".aire()=" << c1.aire()<< "\n";
49     auto c2 = c1.translate(10, 20);
50     std::cout << "c2: " << c2 <<".aire()=" << c2.aire()<< "\n";
51     std::cout << "c1: " << c1 <<".aire()=" << c1.aire()<< "\n";
52 }
```

**Fin réponse**

★ **Exercice 2 :** Surcharge et redéfinition de méthodes (5pts).

On compile le programme ci-dessous avec `g++` avec les paramètres `-Wall -Wextra`. Attention, ce n'est pas exactement le code étudié en cours. Les lignes 1 à 15 compilent et s'exécutent sans erreur.

```
1  #include <iostream>
2
3  struct A {
4    virtual void f1()  { std::cout << "A::f1\n"; }
5  };
6  struct B : public A {
7    void f1() override { std::cout << "B::f1\n"; }
8  };
9
10 struct UU {
11    virtual A foo()  { return A();     }
12    virtual A* bar() { return new A(); }
13    virtual void qux(A a)   { a.f1();  }
14    virtual void xyz(A* a) { a->f1(); }
15 };
16 struct DD : public UU {
17    B foo()  { return B(); }
18    B* bar() { return new B(); }
19    void qux(B b)  { b.f1();  }
20    void xyz(B* b) { b->f1(); }
21 };
```

```
22 int main() {
23    UU u;
24    DD d;
25    A a;  B b;
26
27    u.foo().f1();
28    d.foo().f1();
29    u.bar()->f1();
30    d.bar()->f1();
31
32    u.qux(a);
33    u.qux(b);
34    u.xyz(&a);
35    u.xyz(&b);
36
37    d.qux(a);
38    d.qux(b);
39    d.xyz(&a);
40    d.xyz(&b);
41    return 0;
42 }
```

▷ **Question 1 :** Pour chacune des lignes 17 à 20, indiquez (en justifiant très brièvement) si elle compile ou non (rappel : le mot-clé *override* est optionnel en C++).

**Réponse**

```
16 struct DD : public UU {
17    //B foo()  { return B(); }   // ERROR: invalid covariant return type
18    B* bar() { return new B(); } // OK (redéfinition. Type retour = pointeur, covariance OK)
19    void qux(B b)  { b.f1();  }  // OK (surcharge sans redefinition)
20    void xyz(B* b) { b->f1(); }  // OK (surcharge sans redefinition)
21 };
```

Dans l'exemple vu en cours, les deux dernières lignes ne compilaient pas car on avait ajouté `override` aux méthodes `B::qux` et `B::xyz` alors qu'il n'y a pas redéfinition. Comme le code de cet exercice n'a pas `override`, la réponse n'est pas la même.

**Fin réponse**

Dans la suite de l'exercice, on supposera que les lignes ne compilant pas sont commentées.

▷ **Question 2 :** Pour chacune des lignes 27 à 40, indiquez (en justifiant très brièvement vos réponses) :

1. si elle compile ou non, en explicitant les éventuelles erreurs et warning produits ;
2. le message affiché à l'exécution (si la compilation est possible).

**Réponse**

```
27 u.foo().f1(); // OK: A::f1 (u est UU, foo renvoie du A)
28 d.foo().f1(); // OK: A::f1 (foo n'est pas surchargé ni redéfini, A::foo invoqué)
29 u.bar()->f1();// OK: A::f1 (u est UU, bar renvoie du A)
30 d.bar()->f1();// OK: B::f1 (d est DD, DD::bar renvoie du B, usage de la surcharge B::f1)
31
32 u.qux(a); // OK: A::f1 (pas de B en vue, A::f1 seule possibilité)
33 u.qux(b); // OK: A::f1 (b est amputé -transtypage violent- lors du passage de params)
34 u.xyz(&a);// OK: A::f1 (pas de B en vue, A::f1 seule possibilité)
35 u.xyz(&b);// OK: B::f1 (pas d'amputation sur un passage de pointeur, B s'exprime)
36
37 //d.qux(a); // ERR: cannot convert 'A' to 'B' (DD::qux attend un paramètre B, impossible à downcaster)
38 d.qux(b);   // OK: B::f1 (compil: B attendu et obtenu; exec: b passé sans dommage)
39 //d.xyz(&a);// ERR: invalid conversion from 'A*' to 'B*' [-fpermissive] (*B attendu, downcast dangereux)
40 //             mais g++ peut être permissif si on ajoute -fpermissive (ce qu'il ne faut JAMAIS faire)
41 d.xyz(&b);  // OK: B::f1 (*B attendu et obtenu)
```

**Fin réponse**

▷ **Question 3 :** Expliquez la différence entre *surcharge de méthode* et *redéfinition de méthode* en utilisant des exemples dans le code présenté.

**Réponse**

`B* DD::bar()` est une redéfinition (*override*). Le type des paramètres ne changeant pas, ce n'était pas une surcharge *(overload)*.

La tentative de redéfinition `B DD::foo()` ne compile pas car les types de retour n'étant pas des pointeurs doivent être invariants. Ca ne pouvait pas non plus être une surcharge car le type des paramètres ne change pas (le type de retour ne rentre pas en ligne de compte dans la détermination de la surcharge).

`DD::qux(B b)` est une surcharge *(overload)*. Ca ne pouvait pas être une redéfinition car les paramètres doivent être invariants, mais le compilateur s'en sort en prennant le code fourni comme une surcharge puisque le type des paramètres change. Le type `DD` a donc deux méthodes `qux`. `DD::qux(A a)` est celle héritée de `UU` (définie ligne 13) tandis que `DD::qux(B b)` est définie ligne 19.

`DD:xyz(B* b)` est une surcharge tout comme le cas précédent. Le fait que le paramètre soit un pointeur ne change rien à l'affaire d'invariance. Pour rappel, dans le cas contraire (si on autorisait la variance des paramètres pointés), le mangling et la recherche de la méthode à invoquer d'après le type des paramètres serait encore plus difficile.

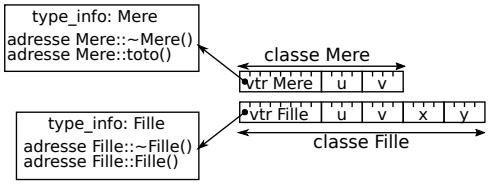
**Fin réponse**

▷ **Question 4 :** Expliquez le fonctionnement des différents mécanismes de sélection du code à exécuter (*code binding*) en C++. La réponse attendue est de l'ordre de la dizaine de lignes d'explications, éventuellement accompagnée de schémas explicatifs.

**Réponse**

Si la méthode n'est pas virtuelle, on a de la **liaison statique**. Tout se passe comme en C : le choix du code à exécuter se fait à l'édition de lien (pendant la compilation le plus souvent, plus tard dans le cas d'une bibliothèque dynamique mais c'est hors sujet en L3).

Si la méthode est virtuelle, on a de la **liaison dynamique**. Le compilateur génère une petite méthode bouchon (un *stub)* qui est appelée quand le symbole est utilisé. Le travail de cette fonction est de chercher la bonne méthode à exécuter en consultant la `vtable` de l'objet, qui est une table de pointeurs vers les bonnes méthodes à utiliser pour cet objet-là. C'est comme ça que le type dynamique d'un objet influe sur le code invoqué sur cet objet.




On peut noter qu'il n'y a qu'une vtable par classe, partagée entre toutes les instances de cette classe. C'est donc un pointeur au début de la zone allouée pour chaque instance, comme sur le schéma de gauche. Enfin, les choses sont plus compliquées dans le cas d'héritage multiple (mais c'est hors programme : nous n'en avons pas parlé en cours).

**Fin réponse**

★ **Exercice 3 :** Design de code orienté objet (6pts).

Vous trouverez sur l'autre feuille le code du jeu de UNO écrit en C++ [1], mais en suivant la philosophie du C. Ce code, relativement bien documenté, tire partie des conteneurs et algorithmes standards du C++, mais aucune classe d'objet n'est définie, il n'y a pratiquement pas de fonction pour découper et abstraire le code. L'objectif de cet exercice est de le refactorer pour le rendre plus lisible.

Exemple de partie

```
Joueur 1, à vous.
Carte sur la défausse: 7 bleu;
Votre jeu: a:3 rouge | b:4 rouge | c:7 vert | d:9 vert | e:change de sens vert | f:2 jaune | g:Joker |
Quelle carte jouer ('z' pour tirer une nouvelle carte)? g


Joueur 2, à vous.
Carte sur la défausse: Joker;
Votre jeu: a:+2 rouge | b:passe ton tour vert | c:3 jaune | d:2 jaune | e:+2 jaune | f:0 bleu | g:7 bleu |
Quelle carte jouer ('z' pour tirer une nouvelle carte)? a


2 cartes de pénalité pour le joueur 1!!
Carte sur la défausse: +2 rouge;
Votre jeu: a:3 rouge | b:4 rouge | c:9 rouge | d:7 vert | e:9 vert | f:change de sens vert | g:2 jaune | h:5 jaune |
Quelle carte jouer ('z' pour tirer une nouvelle carte)? ^C
```

▷ **Question 1 :** On souhaite modifier ce code pour que la main du joueur soit triée lorsque vient son tour. Indiquer la ou les ligne(s) à ajouter, ainsi que l'endroit du code où l'ajout doit avoir lieu.

**Réponse**

À ajouter juste avant l'affichage de la main (entre les lignes 46 et 47)

```
  std::sort(hand.begin(), hand.end());
```

**Fin réponse**

▷ **Question 2 :** Proposez une fonction pour séparer et abstraire proprement les lignes 57, 58 et 59. Proposez un nom, donnez son implémentation et indiquez comment la fonction `main()` doit être modifiée pour utiliser cette nouvelle fonction.

**Réponse**

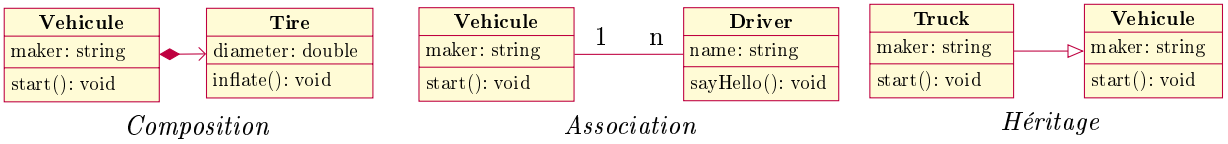
```
bool compat(int c1, int c2) {
  if (defausse.back() > 103 || hand[idx - 1] > 103) // l'une est joker
    return true;
  if (defausse.back() / 26 == hand[idx - 1] / 26) // même couleur
    return true;
  if ((defausse.back() % 26)%13 ==  (hand[idx - 1] % 26)%13) // même valeur
    return true;
  return false;
}
```

Cette fonction retourne si deux cartes sont compatibles. À l'usage, on ferait `break` si `idx==0` ou si la carte jouée est compatible avec le sommet de la défausse.

```
    if (idx == 0 || compat(defausse.back(), hand[idx - 1])
      break; // choix validé
```

**Fin réponse**

On rappelle les notations UML pour les relations classiques entre classes.




*Composition* *Association* *Héritage*

1. Les règles sont légèrement simplifiées par rapport au vrai jeu de UNO, mais l'esprit demeure.

▷ **Question 3 :** Proposez un découpage en classes de ce problème. Vous donnerez les responsabilités de chaque classe (au sens CRC) éventuellement avec les lignes du code existant correspondant à chaque responsabilité, avant de dessiner le diagramme UML liant les classes. Justifiez ce qui doit l'être.

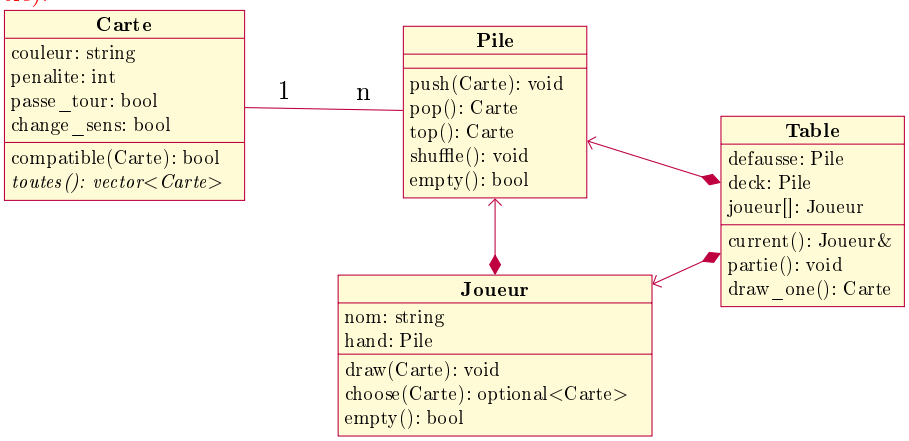
**Réponse**

La classe `Table` représente une table de jeu. Elle compte deux `Pile` : l'une pour la défausse et l'autre pour le talon dont on tire les cartes (relation de composition). Elle compte aussi deux `Joueurs`, et elle sait quel est le joueur dont c'est le tour. La méthode `draw_one()` retire la carte au sommet du talon. Au besoin, elle remet les cartes de la défausse dans le talon avant de le mélanger.

La classe `Joueur` représente un joueur. Elle a un nom (chaîne de caractères) et une main représentant ses cartes. Elle sait tirer une nouvelle carte (pour l'ajouter à sa main), et choisir une carte de sa main à jouer en demandant à l'humain quelle carte jouer en fonction de la carte au sommet de la défausse (si l'humain décide de tirer une carte, on retourne un `std::optional` invalide – charge à l'appelant de me donner une carte). Cette seconde méthode reprend les lignes 48 à 61 du code proposé. Une méthode `empty()` retourne si la main du joueur est vide.

La classe `Pile` représente une pile de cartes (la défausse, le talon ou la main d'un joueur). On peut y ajouter une carte, ou retirer celle au sommet de la pile. On peut aussi la mélanger. L'accesseur top permet de consulter la carte au sommet sans modifier la pile, tandis que empty retourne si la pile est vide.

La classe `Carte` représente une carte donnée. Elle a une couleur, un nombre de cartes de pénalité à ramasser quand l'adversaire joue cette carte, elle peut passer le tour et/ou changer de sens. Une méthode permet de savoir si la carte courante est compatible avec celle passée en paramètres. Une méthode statique *toutes()* retourne un vecteur de toutes les cartes existantes (pour fabriquer le talon en début de partie).




**Fin réponse**

▷ **Question 4 :** Écrivez le code de la boucle principale en supposant que toutes les autres méthodes de vos classes sont écrites. Si vous avez correctement découpé le projet, la boucle principale ne devrait pas dépasser une quinzaine de lignes.

**Réponse**

```
std::vector<Joueur> joueurs /* initialisation omise */;
int joueur_courant = 0;
int sens_de_jeu = 1;
while (not joueurs.empty()) {
  auto joueur = joueurs[joueur_courant];
  while (not joueur.empty()) {
    for (int i=0; i<defausse.top().penalite(); i++)
      joueur.draw(draw_one());
    auto choice = joueur.choose(defausse.top());
    if (choice.has_value()) {
      defausse.push(choice.value());
      if (defausse.top().change_sens())
        sens_de_jeu *= -1;
      std::swap(joueur, other);
```

```
  } else {
    joueur.draw(draw_one());
    std::swap(joueur, other);
  }
}
std::cout << joueur.name << "a gagné!!\n";
```

**Fin réponse**

★ **Exercice 4 :** Comprendre et modifier du code C++ (reprise du TP5 – 6pts)

Observez le code intitulé *Code mystère* en dernière page. Les entêtes et la méthode `sol_to_str()` sont omises, mais le code complet compile et s'exécute sans problème. Attention, ce n'est probablement pas exactement le code que vous avez utilisé dans le TP 5.

▷ **Question 1 :** Expliquez le principe général de ce code. L'objectif n'est pas d'expliquer l'algorithme permettant de résoudre le problème, mais plutôt d'expliquer les constructions C++ utilisées : précisez le statut de `tplate()`, son fonctionnement et son utilisation dans `main()`.

**Réponse**

(2pts) `tplate` est une fonction templatée prenant une lambda en paramètre, et renvoyant une autre lambda en résultat. Le résultat renvoyé par `tplate` est stocké dans la variable `fn` à la ligne 34, tandis que le paramètre est le corps de la lambda défini entre les lignes 36 et 54.

Cette fonction templatée est utilisée pour mémoiser de façon automatique toute lambda à deux paramètres passée en paramètre à `tplate`. La lambda retournée par `tplate` vérifie n'effectue un calcul appelant la fonction récursive `a` que si le résultat n'est pas encore stocké dans la table `b`. Si le résultat est déjà connu, il est retourné sans être recalculé.

Lors l'instanciation de la template (lignes 34 à 54), le résultat de `tplate()` (nommé `fn`) est passé dans la closure de la lambda passée à `tplate()` (ligne 36). Lorsque la lambda définie entre les lignes 35 à 54 fait un appel récursif (à la ligne 44), elle invoque `fn` au lieu de s'appeler elle-même directement. La mémoisation est donc appliquée avant de réinvoquer la lambda appelante si le résultat n'est pas encore connu.

On a donc une sorte de récursivité croisée entre les deux lambdas, l'une générique consultant la table, et l'autre spécifique au problème considéré.

**Fin réponse**

▷ **Question 2 :** Expliquez les lignes 38 et 39 de ce code.

**Réponse**

(0.5pt) Si on cherche à découper une corde de longueur nulle, on a touché le cas de base de la récursion. Il faut donc retourner une paire formée du nombre 0 (la valeur de ce découpage) et un vecteur de $N$ booléens tous à `false`, avec $N =$ longueur du vecteur $pp$ passé en paramètre à la fonction.

**Fin réponse**

▷ **Question 3 :** Expliquez la ligne 45 de ce code.

**Réponse**

(0.5pt) C'est une affectation déstructurée du C++17. La `std::pair` retournée par la fonction `fn` est déconstruite, et ses éléments sont affectés automatiquement aux variables `cost` et `solution` dont le type est déduit automatiquement.

**Fin réponse**

▷ **Question 4 :** Replacez les lignes 22 à 31 par autre chose pour éviter la fuite mémoire liée au `new` de la ligne 24, dont la mémoire n'est jamais libérée dans l'état actuel.

**Réponse**

(1pt)

```cpp
template <typename X, typename Y, typename Z> class tplate {
  std::function<X(Y, Z)> a;
  std::map<std::pair<Y, Z>, X> b;

public:
  tplate(std::function<X(Y, Z)> a_) : a(a_) {}
  X operator()(Y y, Z z)
  {
    auto k = std::make_pair(y, z);
    if (b.find(k) == b.end())
      b[k] = a(y, z);
    return b[k];
  }
};
```

**Fin réponse**

▷ **Question 5 :** Écrivez la fonction `sol_to_str()` dont le prototype est donné ci-dessous. Pour obtenir tous les points de la question, vous devez l'écrire en utilisant une écriture aussi fonctionnelle que possible. Pour cela, vous aurez probablement besoin de `ranges::zip_view`, `ranges::iota_view(0)` et `ranges::views::filter()`, définis dans la bibliothèque range-v3. Pourquoi utiliser cette bibliothèque ?

**Réponse**

(2pt, moitié si pas de zip)

```cpp
#include <climits>
#include <functional>
#include <iostream>
#include <map>
#include <range/v3/all.hpp>

std::string sol_to_str(std::vector<bool>& sol, int value)
{
  std::string res = "[";
  bool first      = true;
  for (auto s : ranges::zip_view(sol, ranges::iota_view(0))
       | ranges::views::filter([](const std::pair<bool, int> elm){ return elm.first; })
       | ranges::views::transform([](const std::pair<bool, int> elm) {
           return std::to_string(elm.second);
         })) {
    res += (first ? "" : " + ") + s;
    first = false;
  }
  res += "]. Price: " + std::to_string(value) + "\n";
  return res;
}
```

**Fin réponse**

```cpp
std::string sol_to_str(std::vector<bool>& sol, int value);
```

Jeu de UNO en pseudo-C

```
std::string to_str(int card) { // Représentation textuelle d'une carte identifiée par son num.
  std::string couleur[] {"rouge", "vert", "jaune", "bleu"};
  if (card <= 103) {
    int v = card % 26;
    if (v > 12)
      v -= 13; // Toutes les cartes sont en double
    if (v < 10)
      return std::to_string(v % 10) + " " + couleur[card / 26];
    if (v == 10)
      return "+2 " + couleur[card / 26];
    if (v == 11)
      return "change de sens " + couleur[card / 26];
    if (v == 12)
      return "passe ton tour " + couleur[card / 26];
  } else if (card <= 107)
    return "Joker";
  return "Super joker +4";
}
int main() {
  srand(time(nullptr)); // Initialize the pseudo-random
  std::vector<int> deck, defausse;
  for (int i = 0; i < 112; i++)
    deck.push_back(i);
  std::random_shuffle(deck.begin(), deck.end()); // permutation aléatoire
  std::vector<int> hand, other;  // jeu de chaque joueur (hand: joueur actuel; other: l'autre)
  for (int i = 0; i < 7; i++) {
    hand.push_back(deck.back());     deck.pop_back();
    other.push_back(deck.back());    deck.pop_back();
  }
  defausse.push_back(deck.back());   deck.pop_back(); // Place une carte dans la défausse
  int player = 1;
  while (true) { // La partie continue
    int idx;
    int penalite = (defausse.back() % 26 == 10 || defausse.back() % 26 == 23) ? 2
                 : ((defausse.back() > 107) ? 4 : 0);
    if (penalite > 0) {
      std::cout << "\n\n" << penalite << " cartes de pénalité pour le joueur "<<player<<"!!\n";
      for (int i = 0; i < penalite; i++) {
        hand.push_back(deck.back());
        deck.pop_back();
    } } else std::cout << "\n\nJoueur " << player << ", à vous.\n";
    while (true) { // Attend une carte valide
      std::cout << "Carte sur la défausse: " << to_str(defausse.back()) << ";\nVotre jeu: ";
      for (unsigned i = 0; i < hand.size(); i++)
        std::cout << static_cast<char>('a' + i) << ":" << to_str(hand[i]) << " | ";
      std::cout << "\nQuelle carte jouer ('z' pour tirer une nouvelle carte)? ";
      char choix;
      std::cin >> choix;
      if (choix == 'z' || (choix >= 'a' && choix - 'a' < hand.size())) {
        idx = (choix == 'z') ? 0 : choix - 'a' + 1;
        if (idx == 0 // le joueur tire une carte
            || defausse.back() > 103 || hand[idx - 1] > 103 // l'une est joker
            || defausse.back() / 26 == hand[idx - 1] / 26 // même couleur
            || (defausse.back() % 26)%13 ==  (hand[idx - 1] % 26)%13) // même valeur
          break; // choix validé
    } }
    if (idx == 0) {
      hand.push_back(deck.back());      deck.pop_back();
    } else {
      defausse.push_back(hand[idx - 1]);
      hand.erase(hand.begin() + idx - 1); // Retire hand[idx] du vecteur hand
    }
    if (hand.empty()) {
      std::cout << "LE JOUEUR " << player << " A GAGNÉ!!\n";
      exit(0);
    }
    if (deck.empty()) {
      std::swap(deck, defausse); // inverse les deux piles
      std::random_shuffle(deck.begin(), deck.end()); // mélange
      defausse.push_back(deck.back());       deck.pop_back();
    }
    std::vector<int> pt{11, 12, 24, 25}; // Cartes qui passent son tour, dans chaque couleur
    if (std::none_of(pt.begin(), pt.end(),
                     [&defausse](auto i) { return defausse.back() % 26 == i; })) {
      player = (player == 1 ? 2 : 1);
      std::swap(hand, other);
} } }
```

Code mystère

```
std::function<X(Y, Z)> tplate(std::function<X(Y, Z)> a){
  auto* b = new std::map<std::pair<Y, Z>, X>();
  return [a, b](Y y, Z z) -> X {
    auto k = std::make_pair(y, z);
    if (b->find(k) == b->end())
      (*b)[k] = a(y, z);
    return (*b)[k];
  };
}

int main()
{
  std::function<std::pair<int, std::vector<bool>>(std::vector<int>, int)> fn =
    tplate<std::pair<int, std::vector<bool>>, std::vector<int>, int>
      ([&fn](std::vector<int> pp, int nn) {

        if (nn == 0)
          return std::make_pair(0, std::vector<bool>(pp.size(), false));

        int max_value = INT_MIN;
        std::vector<bool> best_solution;
        for (int i = 1; i <= nn; i++) {
          auto [cost, solution] = fn(pp, nn - i);

          if (pp[i - 1] + cost > max_value) {
            max_value     = pp[i - 1] + cost;
            best_solution = solution;
            best_solution[i] = true;
          }
        }

        return std::make_pair(max_value, best_solution);
      });

  std::vector<int> p {2, 5, 9, 9, 14, 17, 17, 25, 20, 22, 23, 2, 4, 6};
  auto [cost, sol] = fn(p, p.size());

  std::cout << "Best solution for n=" << p.size() << ": " << sol_to_str(sol, cost);
  return 0;
}
```