Université Paris Cité **Langages à objets avancés**
M1 **Année 2023—2024**

TP Noté
Gestion de projet

1 Généralités

Ce TP noté compte pour 20% de votre note finale, il vous permet de faire le point sur les premières choses que nous avons vues ensemble et qui sont propres à c++. Il nous permettra aussi de vous fournir un premier retour avant le projet (40%) et l'examen (40%).

Le travail est à réaliser en binôme, c'est un format qui vous permettra d'avoir des échanges, un regard critique et éventuellement une aide tout le long de sa réalisation. Trouver un binôme fait partie du travail demandé !

Les créneaux du TP 5 (avant les vacances) seront utilisés pour vous accompagner. Vous avez environ 10 jours pour le réaliser, il faut compter 4 à 6 heures de travail par personne. Le dépôt est à faire sur Moodle avant dimanche 29/10/2023 à 23h59 (clôture automatique).

Derniers conseils : avant de vous lancer dans la programmation, lisez le sujet en entier, réfléchissez au problème, dessinez un premier diagramme de classes et envisagez les choses dans leur ensemble. Votre programmation ne se fera pas linéairement, relisez et retouchez votre code, posez vous les questions que nous avons abordées en cours (ce qui doit être déclaré constant, s'il faut utiliser des références, des pointeurs, comment se passent les destructions, les copies ...).

2 Description de l'objectif

On s'intéresse à la gestion de projets qui se décomposent en tâches dépendantes les unes des autres. Chacune a sa durée propre, supposée exacte ; elle ne pourra être lancée que si ses dépendances sont déjà réalisée. La durée d'une tâche est fixe, mais la durée totale du projet dépendra du degré de parallélisme que le gestionnaire envisagera.

On distingue deux natures de projets : ceux qui sont en cours d'élaboration, et ceux qui sont en cours d'exécution. Ces objets étant différents, avec leurs méthodes propres, nous les représenterons avec des classes différentes, appellées respectivement `ProtoProjet` et `RunProjet` qui sont conceptuellement suffisamment proches pour être considérées comme étant toutes deux des `Projets` (une notion plus générale). Un `RunProjet` se construira uniquement à partir d'un `ProtoProjet`. Il pourra avancer vers sa conclusion en prenant en compte la réalisation de telle ou telle tâche ponctuelle. Des `Gestionnaires` étudieront les `RunProjet` : ils recommanderont des ordonnancements pour la suite des exécutions à venir ; calculeront la durée totale restante, et demanderont un salaire pour leur expertise.

Pour chaque classe vous devrez pouvoir en afficher une instance en surchargeant l'opérateur <<, avoir une gestion satisfaisante des copies/affectation/destruction. Vous remettrez en plus de votre code une page rédigée montrant que vous avez tenu compte de certaines problématiques de copies/sécurité etc en expliquant en français non technique comment vous les avez traitées. Enfin, pour mettre à l'épreuve votre modélisation vous écrirez un petit jeu de test, clair.

3 Analyse plus détaillée

Nous passons ici en revue les classes à écrire, en précisant davantage ce que nous attendons de votre part.

Tâche

On appelle Tâche un élément atomique à réaliser au sein d'un projet. Elle possède un nom, et vous proposerez une façon de les numéroter automatiquement de sorte qu'on puisse l'identifier par ce numéro. Une tâche peut être réalisée ou en attente. Elle ne peut évoluer que dans un sens : il n'est pas possible de dé-réaliser une tâche. Une durée est prévue pour sa réalisation, on parle ici de sa durée propre. La tâche ne pourra être mise en oeuvre que si ses dépendances sont elles même réalisées. Ce qu'on appelle dépendances, ce sont d'autres tâches qui lui sont prioritaires. Une tâche a une vision locale de ses dépendances, vous utiliserez un `vector` pour la caractériser. Elle peut être incomplète : par exemple, si `t1` dépend de `t2` qui dépend de `t3`, la dépendance de `t1` à `t3` peut apparaître ou pas dans la vision locale qu'a `t1`.

En particulier il vous faudra écrire :

- `bool realise()` qui déclenche la réalisation d'une tâche après avoir vérifié que c'est bien possible.
- `bool depends_from(Tache & x)` indique si la tâche courante dépend transitivement de `x`
- `bool ajouteDependance(Tache & x)` qui ajoute la dépendance de `this` à `x` si celle-ci ne crée pas de cycle.
- `int dureeParal()` qui calcule la durée totale de réalisation d'une tâche et de toutes ses dépendances (au sens large) non encore réalisées. On suppose que les exécutions pourront se faire avec un degré de parallélisme arbitraire. [1].

Projet

Un projet peut être vu comme un graphe acyclique dont les sommets sont des tâches organisées par leur relation de dépendance. On fera en sorte qu'une tâche particulière (la "fin") serve de source à ce graphe.

1. il vous faut simplement prendre un max

Nous conservons dans notre modélisation un `vector` de l'ensemble de ses tâches. Nous supposons (et ferons en sorte) qu'il respecte un ordre topologique, c'est à dire que sa lecture de gauche à droite soit compatible avec la relation des dépendances. (En particulier son premier élément sera donc la tâche "fin").

La classe `Projet` sera mère des deux classes `ProtoProjet` et `RunProjet` présentées ci dessous. Elle factorise la modélisation qui leur est commune, et fourni quelques méthodes générales qui seront utiles à ses sous-classes uniquement.

En particulier :

- `pick_two_random_tasks()` retourne une paire `(id1,id2)` [2] d'identifiants de tâches prises au hasard parmi celles du projet en s'assurant que `id2` ne dépende pas transitivement de `id1`.
- `contains` qui indique pour une tâche désignée par son identifiant, ou par son nom, si elle fait partie du projet. Elle retourne un pointeur vers cette tâche si c'est le cas ou `nullptr` sinon.
- Pour l'affichage, vous vous contenterez de parcourir le vecteur de tâches.
- `consult_tasks()` retourne une version du vecteur de tâches qui ne sera en rien modifiable. Cette méthode sera utilisée par les gestionnaires pour leur travail d'analyse.
- `topologicalSort()` qui corrige les éventuelles anomalies du `vector` de tâches, (voir annexe).

ProtoProjet

Un protoProjet est un objet auquel on veut pouvoir ajouter de nouvelles tâches, mais qui n'est pas destiné à progresser, au sens où on ne pourra pas faire s'exécuter ses tâches.

Initialement il consistera toujours en deux tâches standards qu'on appellera `"Debut"` et `"Fin"` avec la dépendance naturelle que vous imaginez.

L'idée est qu'un concepteur interagisse et se décharge sur le `ProtoProjet` pour construire les tâches : il n'y aura donc pas de méthode d'ajout d'objet `Tache`, en revanche on mettra à sa disposition les 3 méthodes `bool ajoute (...)` suivantes :

- avec pour argument un nom et une durée : le protoProjet sélectionnera au hasard 2 de ses tâches `t1` et `t2` et il insérera une nouvelle tâche `t` en la plaçant "entre" `t1` et `t2` de sorte que `t1` ne puisse pas être réalisée avant `t` et que `t` ne puisse pas être réalisée avant `t2`.
- en donnant un nom, une durée, et un identifiant d'une tâche t. La nouvelle tâche créée devra être réalisée après la tâche t, et avant `"Fin"` ; Si t n'est pas déjà dans le `ProtoProjet` rien ne sera ajouté.
- en donnant un nom, une durée et deux identifiants de tâche. La nouvelle tâche crée se placera "entre" les deux tâches spécifiées.

N'oubliez pas qu'il faut maintenir invariante la propriété que "le vecteur de tâche respecte un ordre topologique" ; et s'il y a un risque de créer un cycle , l'ajout ne se fera pas et vous retournerez `false`.

2. utilisez `std::pair` de la librairie `utility`

RunProjet

Les `RunProjet` seront construits uniquement à partir de `ProtoProjet`. Pour faire en sorte que ses tâches et leurs dépendances ne puissent plus être modifiées par personne, lors de cette construction le `RunProjet` devra s'approprier les tâches du `ProtoProjet` et le vider de sa substance.

Ils disposeront en particulier de méthodes `run` :

— qui prendra un identifiant de tâche, et la fera évoluer si c'est possible.

— ou qui prendra une séquence de tâches et les exécutera dans cet ordre.

Gestionnaire

Les `Gestionnaire` ont pour spécialité l'analyse de `RunProjet` : ils affichent une facturation (faites ce que vous jugez nécessaire), donnent une trace de l'ordre dans lequel ils recommandent les exécutions à venir, et annoncent la durée totale qu'ils estiment nécessaire pour réaliser ce qu'il reste à faire dans le projet. Les deux éléments de réponse sont contenus dans la paire retournée par la méthode : `pair<vector<int>, int> avis(const RunProjet &)`

Ils peuvent être distingués en deux catégories selon la stratégie qu'ils envisagent : les `Expert` et les `Consultant`.

Les `Consultant` envisagent qu'une seule personne devra réaliser l'ensemble des tâches. La durée totale est donc la somme des durées des tâches restantes.

Les `Expert` envisagent qu'une parallélisation totale est possible, et la durée restante est calculée à l'avenant.

Makefile et tests

Votre programme devra pouvoir être compilé avec la commande `make` seule. Tout programme qui ne respectera pas cette consigne aura une note très dégradée. Vous avez à votre disposition les machines de l'ufr pour vous assurer que votre dépôt est portable de votre machine à la notre.

Il faut présenter des tests pour que nous puissions constater que vos différentes constructions et destructions fonctionnent. Rédigez vos affichages afin que nous soyons bien guidés : ce qui est effectué / attendu doit être exprimé dans la console.

Si vous avez prévus des tests que vous souhaitez séparer, pour que nous ayons une ergonomie uniforme, faites en sorte que nous puissions les lancer avec la commande `make test_i`, et regroupez dans `make all` un descriptif court de la nature des tests que vous nous présentez.

Modalités de rendu

Vous rendrez **UNE SEULE** archive par binôme. Le dépôt est à faire sur Moodle sous forme d'une **archive tar** (utilisez impérativement la commande `tar`[3]). Les dépôts seront mécaniquement clos le dimanche 29/10/2023 à 23h59. Merci de respecter cette date limite, et d'anticiper les problèmes techniques. Au pire, si vous doutez de la compatibilité des logiciels de votre machine, utilisez celles de l'ufr.

Assurez vous de déposer les fichiers suivants :

- `Makefile`
- vos sources `.cpp` et `.hpp`,
- `binome.txt` qui contient deux lignes avec vos nom, prénom et numéro d'étudiant
- votre diagramme UML au format `pdf` ou `jpg`, `png`
- `explications.pdf` pour des explications complémentaires.

Exemple

```
int main() {
 ProtoProjet pp;
 pp.ajoute("a",10); // probablement numero 2
 cout << pp; // avec ses 3 taches
 cout << "----------" << endl;
 pp.ajoute("b",15,0,1); // en supposant que 0: fin et 1: début
 cout << pp;
 cout << "----------"<< endl;
 RunProjet rp{pp};
 cout << "----- verification : ProtoProjet vide " << endl;
 cout << pp << endl;
 Consultant c;
 cout << c.avis(rp).second << endl; // dira 25
 Expert e;
 cout << e.avis(rp).second << endl; // dira 15
 return EXIT_SUCCESS;
}
```

Annexe pour vector

Vous aurez ici à utiliser les vectors. Traditionnellement en c++ leurs manipulations se font via des itérateurs, sujet que l'on n'a pas encore abordé. Mais cela n'est pas l'objet principal de ce TP, vous devriez pouvoir écrire des solutions en utilisant les seuls éléments présents dans cet exemple :

3. `tar cvf mon_fichier.tar fic1 fic2 ...` pour créer l'archive `mon_fichier.tar` composée des fichiers `fic1`, `fic2`, ... Un fichier `tar` n'est en aucun cas un fichier `zip` ou autre dont vous auriez changé l'extension ! (vu l'an dernier...)

```
vector<string> v{"un","trois","cinq","sept"};
for (string s:v) cout << s << " ";  // boucle for each element s in v
cout << v.size()<< endl;            // taille du vecteur
cout << v[2] << endl;               // lecture à l'indice 2
v.push_back("neuf");                // ajout en fin au vecteur
v[2]="onze";                        // écriture à l'indice 2
cout << v[2] << endl;
```

Annexe pour `topologicalSort`

Notre graphe des relations de dépendance est acyclique, un tri topologique est donc toujours possible. Il s'obtient facilement par un parcours en profondeur, en partant de la source (`"Fin"`). Nous vous rappelons ici les étapes :

- Vous aurez besoin de marquer les sommets (les tâches) lors de votre parcours : ajoutez leur un attribut public booléen.
- Ecrivez dans `ProtoProjet` une méthode `cleanMarks()` qui remet tous les marquage à leur valeur initiale.
- Dans `Tache` écrivez une méthode de parcours en profondeur `void Tache::PP_postfixe` qui prenne en argument une référence d'un vecteur de Tache. Vous lui ajouterez une tache au moment postfixe de son parcours.
- `topologicalSort` récupèrera le calcul fait par `PP_postfixe` et construira son tri en reprenant simplement ce vecteur à l'envers.