B5 - C++ avancé

B-RPC-500

Amorçage de type R

Écrivez-moi un ECS

SYSTÈME DE COMPOSANTS D'ENTITÉ

QU'EST-CE QU'UN ECS ?

Un ECS est un modèle de conception architecturale, principalement utilisé dans le développement de jeux vidéo. Ce modèle suit le principe de composition plutôt que d'héritage. Au lieu de définir un arbre d'héritage comme d'habitude dans la programmation orientée objet, les types sont divisés en petits mais hautement réutilisables *Composants*.

Un modèle de composition courant dans la POO serait de mettre des fonctions modifiant ces composants avec les composants, par exemple des composants physiques contenant la masse, la vitesse, l'accélération et la hitbox avec des fonctions pour déplacer et heurter des entités qui ont ce composant.
Dans un ECS, nous essayons de garder les composants aussi légers que possible (nous en reparlerons plus tard). Cependant, un problème se pose si nous essayons de mettre une logique à l'intérieur des composants, car une partie de cette logique pourrait dépendre de plusieurs composants d'attributs s'ils sont trop "petits". Une façon d'éviter ce problème tout en gardant les composants aussi petits que possible est de prendre la logique (fonctions) en dehors des composants. Ces fonctions autonomes sont alors libres de dépendre d'un nombre arbitraire de composants. Nous appelons ces fonctions *systèmes*.
Dans notre exemple de composant physique, il serait divisé en sa partie (position, rapidité, Masse, accélération et hitbox), et un tableau de fonctions interagirait avec toutes les entités qui ont un sous-ensemble spécifique de composants.

On aurait ainsi les systèmes suivants :
- **mouvement_système**: met à jour toutes les positions et vitesses des entités si elles ont les composants suivants : rapidité, position et accélération.
- **système_de_gravité**: mise à jour des accélérations des entités, en tenant compte de leur masse. Les composants suivants doivent être présents : accélération et Masse.
- **collision_system**: gère les collisions entre toutes les entités qui ont une hit-box. Les entités doivent être composées des composants suivants : position et hitbox.

Ce modèle permet une architecture plus flexible, avec la possibilité d'ajouter/supprimer des composants à la volée. Vous pouvez par exemple gérer l'effet de statut en ajoutant un composant à une entité et en le gérant dans le système approprié (c'est-à-dire en ajoutant un lent composante à une entité, qui serait prise en compte par le mouvement_système , ou un slow_system qui mettrait à jour le rapidité avant neuf positions sont calculés).

Il y a deux autres avantages avec ce modèle :
- la façon dont les données sont organisées est proche de la façon dont elles pourraient être stockées dans une base de données (chaque composant est une ligne dans une table)
- la façon dont les données sont organisées joue bien avec le cache CPU (plus à ce sujet dans un instant).

ATTENDEZ, OÙ SONT LES ENTITÉS ?

Nulle part. **Les entités n'ont pas besoin d'exister réellement ou de contenir quoi que ce soit**.

Dans un ECS, les entités ne sont pas quelque chose dont vous devriez vraiment vous soucier. Il n'y a pas de type d'entité qui contiendrait ses composants, car il n'y en a pas besoin. Vous pouvez considérer une entité comme un index ou un ID. Certains systèmes n'ont pas besoin de savoir sur quelles entités ils travaillent, tant qu'ils obtiennent tous les composants

dont ils ont besoin pour une entité donnée en même temps. D'autres systèmes ont besoin d'un moyen de modifier une entité donnée (y ajouter un composant, pour qu'un autre système interagisse avec elle). Ces systèmes n'auraient qu'à appeler l'objet ou la fonction gérant toutes les entités et leurs composants, en lui donnant l'ID (ou index) de l'entité.

QU'EST-CE QUE LE CACHE CPU ? EST-CE IMPORTANT ?

Le cache CPU est la mémoire la plus rapide (hors registres) disponible pour votre CPU. Plus le processeur peut compter dessus, plus il peut fonctionner rapidement. Le cache du processeur est organisé en petits morceaux de mémoire (appelés lignes de cache) avec lesquels le processeur fonctionne. Si la donnée à laquelle le CPU veut accéder se trouve dans son cache, il peut travailler avec. Si ce n'est pas le cas (nous appelons cela un manque de cache) et que le processeur doit supprimer toute une ligne de cache et charger la mémoire pour la remplir. Lors du chargement d'une adresse RAM dans son cache, le CPU récupère toujours plus de mémoire afin de remplir la ligne de cache. Ainsi, mettre les données qui seront consultées ensemble, ou plus tôt que les autres données, est essentiel pour optimiser l'utilisation du cache.

Est-ce important pour le projet ? Pas vraiment, donc nous n'essaierons pas de faire un ECS optimisé. Le but du bootstrap est de vous faire écrire un ECS simple afin que vous sachiez à quoi il ressemble et comment travailler avec.

En utilisant le modèle d'héritage OOP habituel (ou en rassemblant toutes les données d'un type), nous remplissons la ligne de cache avec moins d'objets que nous ne le pourrions si nous n'utilisions que ce qui est nécessaire pour le calcul en cours.

Par exemple, dans un jeu vidéo, on utilise généralement une boucle principale qui comporte plus ou moins ces étapes :

- obtenir une entrée
- mise à jour des entités (elle-même composée de plusieurs étapes)
- redessiner

Si vous devez mettre à jour toutes les positions avant de vérifier les collisions, vous mettrez à jour les positions de chaque entité en utilisant leurs vitesses et accélérations, avant de les charger à nouveau, cette fois en utilisant leurs positions et leurs hitbox. Dans un ECS, les composants de position et de vitesse seraient chargés uniquement, ce qui signifie que plus de données pourraient tenir dans une seule ligne de cache (donc moins de charge de la RAM).
En utilisant des modèles OOP traditionnels, une entité complète est chargée, ce qui prend plus d'espace (car nous chargeons également des parties de l'objet dont nous n'avons pas besoin), ainsi moins d'entités peuvent être chargées à la fois dans une ligne de cache (ainsi plus de cachemiss se produisent).

ÉDITONS UN ECS

Dans cette partie, nous allons écrire un ECS simple. Le but est de vous faire comprendre comment il est structuré, pas d'écrire le meilleur/le plus optimisé, ni d'écrire le meilleur moteur en le rendant modulaire.

Dans ce bootstrap, nous utiliserons certaines fonctionnalités de c++17. Bien qu'elles puissent être réimplémentées, il est fortement recommandé de lire d'abord la documentation sur les classes que nous utiliserons et de compiler le bootstrap en utilisant c++17.

Notre ECS sera composé de plusieurs parties :
- **Enregistrement**:Cette classe est le noyau de l'ECS. Il créera et gérera des entités, contiendra des conteneurs de composants et gérera des systèmes.
- **Tableaux clairsemés**:ceux-ci contiendront un type de composants. Il y aura un "trou" lorsqu'une entité n'a pas le composant spécifique. Ceux-ci sont stockés dans le registre.

Dans ce bootstrap, nous éviterons l'héritage en utilisant un concept connu sous le nom d'effacement de type.

Pense**annuler***.

En c++17, un type spécial a été ajouté (std :: tout), qui permet l'effacement de type sécurisé, en stockant la valeur à côté de son type et en la vérifiant lorsque nous essayons de la retransformer.

Nos composants seront stockés dans ce qu'on appelle un tableau creux. Au lieu de conserver l'ID de l'entité dont un composant fait partie, nous utiliserons l'index dans le tableau dans lequel le composant est stocké. Cela signifie que nous avons besoin d'un moyen de représenter un composant inexistant. Nous utiliseronsstandard :: facultatifdans cet effet, car c'est la manière c++ d'y parvenir.

Prenez le temps de lire la documentation destd :: toutetstandard :: facultatif.

ÉTAPE 0 - DÉFINITION DES ENTITÉS

L'entité peut être définie comme un`taille_t`, ou en tant que type nommé, convertible en`taille_t`. Dans ce bootstrap, nous allons implémenter ce dernier.
Ecrire une classe`entité`qui stocke un`taille_t`, et peut être converti en`taille_t`. Cette classe doit être implicitement convertible en `taille_t`, mais ne devrait pas être implicitement constructible à partir de`taille_t`.

Nous voulons que la construction soit**explicite**.

ÉTAPE 1 - TABLEAUX ÉPARSÉS

1.1 - ÉCRITURE D'UN CONTENANT PERSONNALISÉ

Notre conteneur de composants sera implémenté sous la forme d'un tableau clairsemé. Ce type de conteneur fonctionne comme un tableau traditionnel (ou vecteur). La principale différence est qu'une valeur peut ne pas exister à un index spécifique. Cela peut être très utile pour stocker des composants définis pour la plupart des entités, car vous n'avez pas besoin de stocker l'ID d'entité à côté du composant.

Implémentez la classe suivante :

```
modèle<nom de typeComposant >// Vous pouvez également refléter la définition de std :: vector ,
      qui prend un alternateur supplémentaire. classer
sparse_array {
      Publique:
      utilisanttype_valeur = ???;// type de composant facultatif utilisant
      type_référence = type_valeur &;
      utilisantconst_reference_type = value_typeconstante& ;
      utilisantconteneur_t = std :: vecteur < type_valeur > ;// ajouter éventuellement votre alternateur
            modèle ici.
      utilisanttype_taille =nom de typecontainer_t :: size_type ;

      utilisantitérateur =nom de typeconteneur_t :: itérateur ;
      utilisantconst_iterator =nom de typeconteneur_t :: const_iterator ;

      Publique:
      sparse_array () ;// Vous pouvez ajouter plus de constructeurs .

      sparse_array ( sparse_array          constante&) ;// copie le constructeur &&)
      sparse_array ( sparse_array ~        noexcept ;// déplace le constructeur
      sparse_array ());

      sparse_array      &  opérateur=( sparse_array      constante&) ;// copier l'opérateur d'affectation &&)
      sparse_array      &  opérateur=( sparse_array      noexcept ;// déplacer l'opérateur d'affectation

      Type de référenceopérateur[]( size_t idx );
      const_reference_typeopérateur[]( taille_t          idx )constante;

      itérateur commence() ;
      const_iterator       commencer ()   constante;
      const_iterator       commencer ()   constante;

      fin de l'itérateur() ;
      const_iterator       fin ()    constante;
      const_iterator       fin ()      constante;

      taille_type taille ()constante;

      reference_type insert_at ( size_type pos , Composantconstante&) ; reference_type
      insert_at ( size_type pos , Composant &&) ;

      modèle<classer... Paramètres >
      reference_type emplace_at ( size_type pos , Params &&...) ;// optionnel

      annulereffacer ( size_type pos );

      size_type get_index ( value_typeconstante&)constante;
```

```
        privé:
        conteneur_t _data ;
} ;
```

N'hésitez pas à ajouter des fonctions au besoin. La plupart des fonctions sont simples, il vous suffit d'appeler la fonction correspondante du sous-jacentstd :: vecteur. Voici les fonctions que vous**ont**mettre en œuvre vous-même :

- **insert_at**: Ces fonctions doivent insérer le composant à un index spécifique, en effaçant l'ancienne valeur et en redimensionnant le vecteur si nécessaire.
- **emplace_at**: Identique à insert_at, mais le composant est construit sur place, oustd :: déplacer'd dans le conteneur.
- **effacer**: efface une valeur à un index spécifié.
- **get_index**: prend une référence à un composant facultatif et renvoie son index.

Vous ne pourrez pas toujours comparer deux composants, car les entités peuvent avoir des composants identiques.

std :: adressede

Laemplace_atfonction construit l'objet composant sur place. Pour y parvenir, il faut détruire l'objet déjà présent, puis construire le nouveau au même endroit. Heureusement,std :: vecteurpermet d'accéder à leur répartiteur, qui peut être utilisé pour le faire. La norme C++ recommande que le développeur interagisse avec l'allocateur via l'utilisation d'une classe spécialestd :: allocator_traits.

Lisez d'abord la documentation !std :: allocator_traits

Vous pouvez récupérer le type d'un objet au moment de la compilation en utilisantdecltype

ÉTAPE 2 - ENREGISTREMENT

Dans cette étape, vous devrez écrire la classe principale de l'ECS. Lors de la première étape, le but est de stocker chaque sparse_arraydans un conteneur, de sorte que nous puissions les récupérer en utilisant leurs types.

ÉTAPE 2.0 - TYPES COMME INDICES

Nous voulons pouvoir stocker et récupérer des tableaux de composants dans leenregistrement. Une approche naïve consisterait à utiliser unstd :: vecteuren tant que conteneur, mais nous aurions alors besoin de savoir quel tableau est stocké à quel index. Nous pourrions le faire en enregistrant nos tableaux dans le même ordre à chaque fois, ou en attribuant un ID à chaque type de composants. Cependant, les deux solutions nous empêcheraient de charger de nouveaux types de composants à partir de bibliothèques dynamiques tout en garantissant qu'il n'y a pas de collisions.
Une autre façon de gérer cela serait d'utiliserstd ::chaînedans un conteneur associatif, mais cela n'empêcherait pas non plus la possibilité de collision entre deux identifiants de composants.
Heureusement, depuis C++11, une classe est disponible qui nous permet de créer un index unique à l'intérieur d'un conteneur associatif, qui utilise le type réel d'un objet.

Prenez le temps de lire sa documentation :std :: type_index

L'utilisation de ce type garantit que les composants peuvent être chargés au moment de l'exécution, tout en conservant un moyen simple de les récupérer. Tout ce dont nous avons besoin maintenant est de choisir le meilleur conteneur associatif et de trouver un moyen de stocker nos tableaux de composants.

Lorsque vous essayez de décider du type de conteneur associatif dont vous avez besoin, vous devez toujours penser aux fonctionnalités dont vous avez besoin. Si vous avez besoin de trier des clés ou de rechercher un élément, vous pouvez utiliser un conteneur ordonné (std :: carte). Cependant, si vous ne souhaitez accéder qu'à un seul élément spécifique à la fois, un conteneur non ordonné serait plus efficace.
(std :: unordered_map).

ÉTAPE 2.1 - STOCKAGE DE PLUSIEURS TYPES DANS UN RÉCIPIENT

En C, nous pourrions passer un pointeur non typé à n'importe quelle classe en utilisantannuler *. Un style plus orienté objet consisterait à utiliser une interface ou une classe abstraite. Bien que les deux méthodes fonctionnent en C++, ce ne sont pas nécessairement les meilleures façons de le faire, carannuler *s sont de type non sécurisé, etdiffusion_dynamiques vous obligent à essayer chaque type ou à ajouter des identifiants. Alors que dans notre cas, une interface simple et un static_cast fonctionneraient (nous utilisons de toute façon nos types pour récupérer les données), nous allons introduire un moyen sûr de type pour stocker des données non liées dans un conteneur. Depuis C++17, un wrapper spécial existe dans la bibliothèque standard qui fait exactement cela :std :: tout. Un exemple destd :: toutstocke une référence à un objet de manière opaque. Une référence à l'objet stocké peut être récupérée à l'aide destd :: any_cast. Le cast échouera et une exception sera levée si le type de destination n'est pas compatible avec l'objet stocké.

ÉTAPE 2.2 - ÉCRITURE DU REGISTRE.

Vous devriez maintenant être capable d'écrire la classe suivante :

```
classerregistre {
    Publique:
        modèle<classerComposant >
        sparse_array < Composant > & register_component () ;

        modèle<classerComposant >
        sparse_array < Composant > & get_components () ;

        modèle<classerComposant >
        sparse_array < Composant >constante& get_components ()constante;

    privé:
    /* type de conteneur associatif */_components_arrays ;
}
```

registre_composantsajoute un nouveau tableau de composants à notre conteneur associatif. **get_components** sont des fonctions pour récupérer le tableau qui stocke un type spécifique de composants.
Vous devez écrire ces fonctions de manière à ce que l'index d'un tableau soit créé à l'aide destd :: type_index, et le std :: any_castest faite par leenregistrementau lieu de cela, le système qui utilisera lesparse_arrays.

ÉTAPE 2.3 - ENTITÉS DE GESTION

Notre registre devrait également être responsable de la gestion des entités. En créer un nouveau est assez facile, car il suffit de garder une trace du nombre d'entités existantes. Une façon plus avancée de le faire serait de garder une trace des entités mortes afin que vous puissiez réutiliser leurs identifiants au lieu de toujours augmenter un nombre. (N'oubliez pas que nos tableaux clairsemés ont des trous, et bien que cela puisse être plus efficace que de stocker un tableau d'index ou d'utiliser un conteneur associatif, plus il y a de trous, moins ils deviennent efficaces, à la fois en termes de vitesse et d'empreinte mémoire).

Un problème survient lorsque nous pensons à supprimer une entité. Bien que nous stockionssparse_arrays dans un seul conteneur, nous ne pouvons pas y accéder sans connaître le type de composant dont nous avons besoin. Une solution serait alors de revenir à l'utilisation de l'héritage, afin que nous puissions appeler notreeffacerfonction sur lesparse_arrays, mais nous avons essayé d'éviter l'héritage jusqu'à présent, nous allons donc essayer de l'éviter ici si possible.

Nous avons accès au type de composant dans notreregistre_composantfonction

Nous allons modifier notre registre et sa fonction de registre pour pouvoir créer et stocker des fonctions capables d'effacer un composant de chaquesparse_arrays. La signature de nos fonctions sera la suivante :annuler (registre &, entity_t const &)**.**
Modifiez votre registre afin qu'il stocke un conteneur pour ces fonctions, puis mettez à jourregistre_composantdonc il crée et stocke lesdites fonctions.

Vous pouvez soit utiliser une fonction basée sur un modèle, soit créer une fonction lambda lors de l'enregistrement du composant. Utilisationstd :: fonctionpour les stocker à l'intérieur du conteneur.

Ajoutez les méthodes suivantes au registre, ou une classe de votre choix qui sera stockée dans le registre et gère les entités.

```
entité_t        spawn_entity () ;
entité_t        entity_from_index ( std :: size_t idx );
annulerkill_entity ( entité_t                constante&e);

modèle          <nom de typeComposant >
nom de type     sparse_array < Composant > :: type_référence                add_component ( entité_t          constante    &à ,
        Composant     && c);
modèle          <nom de type     Composant ,nom de type... Paramètres >
nom de type     sparse_array < Composant > :: type_référence                emplace_component ( entité_t          constante    &à
        , Paramètres     &&... p);

modèle<nom de typeComposant >
annulerremove_component ( entité_tconstante& de );
```

Laadd_componentetemplace_componentles fonctions doivent prendre leurs paramètres par référence lvalue ainsi que par référence rvalue. Vous pourriez surchargeradd_component, mais cela ne fonctionnera pas pouremplace_component. Cependant, lorsqu'il s'agit de types de modèles,Taper &&(généralement appelée "référence universelle") signifie que le paramètre sera soit une lvalue, soit une rvalue, selon la manière dont la fonction a été appelée. Vous pouvez ensuite appelerstd ::forward<Type>pour conserver cette propriété lors de l'appel d'une autre fonction.

Vous pouvez maintenant modifier les entités afin que seule leur classe de gestion puisse les créer.

Vous êtes-vous déjà demandé à quoi servaitami? C'est un excellent moyen d'appeler un constructeur privé en veillant à ce qu'une seule classe puisse créer un type.

ÉTAPE 3 - UTILISATION DU SEC

Vous devriez maintenant avoir un ECS de base. Vous l'utiliserez pour implémenter un petit programme qui déplace un carré 2D dans une fenêtre SFML. La première étape consistera à définir nos composants et à les enregistrer dans le registre. Une fois les composants définis, vous écrirez plusieurs systèmes qui feront le travail.

ÉTAPE 3.1 - LISTE DES COMPOSANTS

Voici les composants que nous utiliserons pour cet exercice :

- **position**: Ce composant contient la position 2D d'une entité.
- **rapidité**: Ce composant contient deux variables, représentant la vitesse actuelle d'une entité.
- **dessinable**: Ce composant rend notre entité dessinable.
- **contrôlable**: Un objet avec ce composant peut être contrôlé par l'utilisateur, à l'aide de son clavier.

Écrivez-les commestructures, puis enregistrez-les dans le registre. Implémentation decontrôlableetdessinabledépend de vous.

ÉTAPE 3.2 - SYSTÈMES

Nous voulons maintenant ajouter un peu de logique à notre "jeu". Nos systèmes peuvent être des fonctions libres, des lambdas ou des instances de classe qui surchargent leopérateur()fonction. La signature de nos systèmes sera la suivante :

void example_system(registre &);

Ajoutons les systèmes suivants :

- **position_system**: ce système utilise les deuxpositionetrapiditéComposants. Pour toutes les entités ayant les deux, il ajoute la vitesse actuelle à la position.
- **Système de contrôle**: pour les entités avecontrôlableetrapiditécomposants, il règle la vélocité sur une valeur fixe en fonction des touches enfoncées sur le clavier, ou sur 0 s'il n'y a pas d'entrées.
- **draw_system**: utilise ledessinablecomposant à connaître*Quel*dessiner, et lepositionà savoir*où*dessiner.

Voici un petit exemple, qui utilise à la foispositionetrapidité, et enregistrez-les surcerr

```
annulerlogging_system (registre &r) {
    auto constante & postes = r. get_components < composant :: position >>() ;
    auto constante & vitesses = r. get_components < composant :: vitesse >() ;

    pour( taille_t je = 0; je < positions . taille () && je < vitesses . taille () ; ++ je) {
        auto constante & pos = positions [i] ;
        auto constante & vel = vitesses [i ] ;

        si( pos && vel ) {
            std :: cerr << i <<" : Poste = { "<<pos. valeur () .x <<" , "<<pos. évaluer
                () .y <<" }, Vitesse = { "<< vel . évaluer () . vx <<" , "<< vel . évaluer () . v <<" }"<< std :: endl ;

        }
    }
}
```

ÉTAPE 3.3 - EMBALLER LE TOUT ENSEMBLE

Écrivez un programme qui enregistrera les composants définis précédemment, puis créez les entités suivantes :

- Une entité qui est mobile, utilisant tous les composants.
- Certaines entités statiques, qui ontdessinableetpositioncomposants, mais pasrapidité.

Vous pouvez maintenant essayer d'ajouter de nouveaux composants et systèmes. Serait-il facile d'ajouter une accélération ou une décélération (votreSystème de contrôleinteragirait avec la composante d'accélération) ? Ou des entités qui se déplacent sur l'écran, en boucle (à l'aide d'un composant spécial ?).

ÉTAPE 4 - ALLER PLUS LOIN

ÉTAPE 4.1 - DÉPLACEMENT DES SYSTÈMES À L'INTÉRIEUR DU REGISTRE

Bien que notre ECS soit utilisable pour le moment, les systèmes sont responsables de la récupération des composants, ce qui peut ajouter beaucoup de code passe-partout dans toutes les fonctions. Il serait préférable que le registre appelle directement nos systèmes, en passant des paramètres correctement typés.

Nous pouvons modifier ainsi notre exemple de système à partir de l'étape 3.2 :

```
annulerlogging_system ( registre &r ,
                        sparse_array < composant :: position >        constante   & postes ,
                        sparse_array < composant :: vitesse >         constante   & vitesses ) {
    pour( taille_t je = 0; je < positions . taille () && je < vitesses . taille () ; ++ je) {
        auto  constante  & pos  =  positions [i] ;
        auto  constante  & vel  =  vitesses [i ] ;

        si( pos && vel ) {
            std :: cerr << i <<" : Poste = { "<<pos. valeur () .x <<" , "<<pos. évaluer
                () .y <<" }, Vitesse = { "<< vel . évaluer () . vx <<" , "<< vel . évaluer () . v <<" }"<< std :: endl ;

        }
    }
}
```

Nous avons maintenant besoin d'un moyen d'enregistrer notre fonction dans notre registre. Nous utiliserons la même astuce que nous avons utilisée pour "tuer" une entité, en enveloppant l'appel à nos systèmes dans une fonction lambda ou basée sur un modèle.

Ajoutez les fonctions suivantes dans le registre :

```
modèle<classer... Composants ,nom de typeFonction >
annuleradd_system (Fonction && f);// transfert parfait en capture lambda, quelqu'un ? // ou

modèle<classer... Composants ,nom de typeFonction >
annuleradd_system ( Fonctionconstante&F);// en le prenant par référence .

annulerrun_systems () ;
```

Cette fonction stockera dans le registre un objet fonction avec l'opérateur suivant() :opérateur vide ()(registre &);qui appellera notre système en lui donnant les tableaux de composants dont il a besoin. Laadd_systemLa fonction prend notre système comme seul paramètre. Nous utilisons ici un modèle pour pouvoir soit écrire des systèmes en tant que lambdas, utiliser n'importe quel type avec un surchargéopérateur()implémenté, ou une fonction libre.

Le dernier paramètre de modèle pouradd_systemest déduit automatiquement et permet d'utiliser ici n'importe quel type Callable.

ÉTAPE 4.2 - ADAPTATEUR DE CONTENEUR ET ITERATEUR PERSONNALISÉ

Jusqu'à présent, vous deviez vérifier si un composant était présent ou non, et vous deviez soit obtenir l'identifiant d'un composant, soit utiliser un standardpourboucle pour itérer sur plusieurs composants. Cela ajoute du code passe-partout inutile dans chaque système.

Pour pouvoir l'utiliser dans une base à distancepour, une classe doit exposercommenceretfinfonctions, qui**devoir**renvoie un type qui fonctionne comme un pointeur hérité. Voici les opérations qu'ils doivent prendre en charge :

- opérateur*():renvoie une référence ou une valeur.
- opérateur->():renvoie une valeur telle quetype->mest égal à(*type).m
- opérateur++():incrémente l'instance, puis lui renvoie une référence lvalue (++val)
- opérateur++(entier):incrémente l'instance et renvoie son état précédent (val++)
- opérateur!=(type const &, type const &):compare les deux valeurs.

Nous écrirons notre propre classe d'itérateur qui gérera au moins ces opérations. Comme nous écrivons notre propre itérateur, autant le faire fonctionner avec certains algorithmes standard, tels questd :: for_each. Pour ce faire, nous devons au moins implémenter C++LegacyInputIteratorexigences nommées. Les exigences nommées définissent une catégorie nommée de types qui peuvent être utilisés dans un contexte générique, sans être liés par héritage.

Vous devriez au moins lire surLegacyInputIterator, et pourrait être intéressé parLegacy-ForwardIterator

Pour que notre itérateur soit utilisé dans une base de gammepour, nous avons besoin d'un pseudo-conteneur, car les boucles basées sur des plages ne sont qu'un sucre syntaxique qui se transforme en une boucle traditionnelle à l'aide d'itérateurs.

```
pour(auto&v : conteneur ) {/* corps */} // est désucrisé
(c'est-à-dire réécrit) en
pour(autoça = conteneur. commencer () ,autofin = conteneur . fin () ; ça != fin ; ++ ça ) {
        auto&v = * il ; {/*
        corps */}
}
```

L'adaptateur de conteneur que nous allons écrire sera construit à partir de plusieurssparse_arrays, et produira des itérateurs qui peuvent itérer sur chacun d'eux en même temps, en sautant des valeurs si un composant est manquant. Deref-référencer l'itérateur nous donnera le type suivant :std :: tuple<Composant1 &, Composant2 &, Composant3 &, . . . >.

Écrivons d'abord notre classe d'itérateur :

```
modèle<classer... Conteneurs > classer
zipper_iterator {
    modèle<classerConteneur >
    utilisantiterator_t = ???// type de Conteneur :: begin () valeur de retour

    modèle<classerConteneur >
    utilisantit_reference_t =nom de typeiterator_t < Conteneur > :: référence ;

    Publique:
    utilisanttype_valeur = std :: tuple </* récupérer it_reference_t :: type de valeur () */
        &... >;// std :: tuple de références aux composants utilisantréférence
    = type_valeur ; utilisantpointeur =annuler; utilisantdifférence_type =
    taille_t ;

    utilisantiterator_category =/* balise d'itérateur appropriée */; utilisantiterator_tuple
    = std :: tuple < iterator_t < Container >... >;

    // Si nous voulons que zipper_iterator soit construit uniquement par zipper . ami
    conteneurs :: zipper < Conteneurs ... >; zipper_iterator ( iterator_tupleconst
    public:                                                    & it_tuple , size_t max );

        zipper_iterator ( zipper_iterator                constante&z);

        zipper_iterator          opérateur++() ;
        zipper_iterator          &opérateur++(entier);

        type de valeur   opérateur*() ;
        type de valeur   opérateur- >() ;

        ami bouffon      opérateur==( zipper_iterator          constante   &lhs ,   zipper_iterator          constante
            & droite );
        ami bouffon      opérateur!=( zipper_iterator          constante   &lhs ,   zipper_iterator          constante
            & droite );

    privé:
        // Incrémente chaque itérateur en même temps. Il passe également au suivant
            valeur si l'un des pointés vers std :: optionnel ne contient pas de valeur .

        modèle< taille_t ... Est >
        annulerincr_all ( std :: index_sequence < Est ... >) ;

        // vérifie si chaque std :: optional est défini . modèle<
        taille_t ... Est >
        bourdonnerall_set ( std :: index_sequence < Est ... >) ;

        // renvoie un tuple de référence aux composants . modèle<
        taille_t ... Est >
        value_type to_value ( std :: index_sequence < Est ... >) ; privé:

        iterator_tuple _current ;
        taille_t _max ;// compare cette valeur à _idx pour éviter une boucle infinie. size_t _idx ;


        statiqueconstexpr std :: index_sequence_for < Conteneurs ... > _seq {};
    } ;
}
```

decltypepermet de récupérer le type d'une expression à la compilation,std ::declval() crée une pseudo-valeur de type T, à la compilation

std :: index_sequence_forsont des aides qui vous permettront de travailler sur chaque partie de votre tuple, en utilisantplier les expressions

En lisantstd :: cravatevous aidera à écrireévaluer

Ensuite, écrivez le pseudo-conteneur zipper :

```
modèle<classer... Conteneurs > classer
fermeture éclair {
    Publique:
        utilisantiterator = zipper_iterator < Conteneurs ... > ; utilisantiterator_tuple =nom
        de typeitérateur :: iterator_tuple ;

        fermeture à glissière ( Conteneurs &... cs );

        itérateur    commencer () ;
        itérateur    fin () ;

    privé:
        // fonction d'assistance pour connaître l'index maximum de nos itérateurs. statique
        size_t _compute_size ( Conteneurs &... conteneurs );
        // fonction d'assistance pour calculer un iterator_tuple qui nous permettra de
            construire notre itérateur de fin.
        statiqueiterator_tuple          _compute_end ( Conteneurs &... conteneurs );
    privé:
        iterator_tuple      _commencer ;
        iterator_tuple      _fin ;
        size_t _size ;
} ;
```

Si vous ne vous souciez pas de l'index de votre entité, vous pouvez maintenant utiliser la classe zipper dans lespour, ou un algorithme standard.

Vous devriez maintenant être en mesure d'écrire leindexed_zipperclasse par vous-même. Cela fonctionne de la même manière que la classe zipper, mais son itérateur a un size_t dans le tuple qu'il renvoie.

En utilisant cette classe, nous pouvons réécrire à nouveau notre exemple de système.

```
annuler  logging_system ( registre &r ,
                          sparse_array < composant :: position > sparse_array constante  & postes ,
                          < composant :: vitesse > pos , vel ] : indexed_zipper constante  & vitesses ) {
    pour(auto&&[je ,      ( positions ,                                    vitesses ))
        std :: cerr << i <<" : Poste = { "<<pos. valeur () .x <<" , "<<pos. valeur () .y
            <<" }, Vitesse = { "<< vel . évaluer () . vx <<" , "<< vel . évaluer () . v <<" }"<< std :: endl ;

}
```

4.3 - OPTIMISER L'EMPREINTE MÉMOIRE

Alors que notre`sparse_array`est utile, il peut devenir inefficace en mémoire si un composant est à peine utilisé (par exemple, un composant de balise, qui ne serait présent que sur les entités contrôlées par l'utilisateur). Dans votre type r actuel, vous voudrez peut-être avoir un type hybride, qui peut fonctionner soit comme un tableau clairsemé, soit comme deux`std :: vecteur`s (l'un stockant les identifiants des entités, l'autre stockant les composants), en fonction de la densité que vous pensez que les conteneurs seront (vous pouvez également calculer cette densité au moment de l'exécution et modifier le stockage sous-jacent à la volée).

Vous devriez lire un peu sur l'union de type sécurisé C++ et le modèle de visiteur (`std :: variante`et `std :: visite`).