Optimisation

Voir cours.

Boost/C++11

1 – Les (principaux) conteneurs ajoutés à la bibliothèque standard du C++11 sont les `unordered_set`, `unordered_multiset`, `unordered_map` et `unordered_multimap`. Comme leurs prédécesseurs, ces conteneurs sont des conteneurs associatifs, qui associent une clé à une valeur (les deux étant confondues dans le cas des `set/multiset`). Les set et les map associent une clé à une valeur, les multiset et les multimap associent une clé à une ou plusieurs valeurs. Les clés sont uniques.

La caractéristique principale des nouveaux conteneurs est qu'ils ne sont pas ordonnés (comme leur nom l'indique), à l'inverse de leurs prédécesseurs qui étaient ordonnés selon la clé. Cela permet d'utiliser des tables de hachages, plus efficaces que les arbres bicolores généralement utilisés précédemment. Ainsi, la complexité des opérations de recherche, d'insertion et de suppression ont une complexité constante alors qu'elle était logarithmique auparavant.

Les opérations à fournir sur les éléments contenus sont également différentes : pour pouvoir être stocké dans un (multi)(set/map), un objet doit pouvoir être comparé (selon une relation d'ordre strict, pour être précis), soit en définissant un operateur '<', soit en spécifiant une fonction de comparaison lors de l'instanciation du conteneur ; pour les unordered_(multi)(set/map), un objet doit pouvoir être comparé pour l'égalité (en définissant un operateur '==' ou bien avec une fonction donnée en paramètre) et haché (généralement en spécialisant `std::hash<T>`).

2 – La bibliothèque de génération de nombres aléatoires repose sur deux types de classes :

- des *générateurs*, qui génèrent des nombres répartis de façon uniforme dans un intervalle donné (par exemple Mersenne Twister, lagged fibonacci, etc.) ;
- des *distributions*, qui transforment une distribution (généralement une distribution uniforme fournie par un générateur) en une autre (binomiale, Bernouilli, exponentielle, etc.).

Implémentation des concepts objets

1 – Fonctions potentiellement générées pour chaque constructeur (on suppose pouvoir utiliser des fonctions surchargées, même si en réalité leur nom sera décoré (« manglé ») :

```
void Derivee_Derivee(Derivee * this)
{
    Base_Base(this);              // construction par défaut de la base : Base::Base()
    this->vptr = Derivee_vtable;

    string_string(&this->s_);     // construction par défaut de s_ : string::string()
}

void Derivee_Derivee(Derivee * this, int i, std::string s)
{
    Base_Base(this, i);            // construction paramétrée de la base : Base::Base(int)
```

```
    this->vptr = Derivee_vtable;

    string_string(&this->s_, s);   // construction par copie de s_ : string::string(string)
}

void Derivee_Derivee(Derivee * this, std::string s)
{
    Base_Base(this);           // construction par défaut de la base : Base::Base()
    this->vptr = Derivee_vtable;

    string_string(&this->s_);  // construction par défaut de s_ : string::string()

    operator=(&this->s_, s);   // assignement de s à s_ : string::operator=(string)
}
```

Les points importants sont :

- Dans tous les cas, le constructeur de la classe de base est appelé. Si la liste d'initialisation du constructeur ne contient pas d'appel explicite, c'est le constructeur par défaut qui est appelé (cf. 1 et 3).
- Dans tous les cas, le vptr doit être assigné pour pointer sur la bonne table des méthodes virtuelles. (En C++, cela est fait après l'appel du constructeur de la classe de base. Par conséquent, l'invocation d'une méthode virtuelle dans le constructeur de la classe de base appelle l'implémentation de la classe de base et pas celle de la classe dérivée (ce qui paraît logique, vu que la partie dérivée n'est pas encore construite)).
- Dans tous les cas, les membres de l'objet doivent être construits. Si la liste d'initialisation ne contient pas d'appel explicite, c'est le constructeur par défaut des membres qui est appelé (cf. 1 et 3).
- Une fois que tout cela est fait, le corps du constructeur est exécuté (appel de l'opérateur d'affectation par exemple (cf. 3))

Remarque : La plupart d'entre vous ont supposé l'existence de métaclasses telles que celles que nous avons utilisé dans notre implémentation ; je n'ai bien évidemment pas compté ça comme une erreur (même si j'ai précisé plusieurs fois que les compilateurs C++ existants n'utilisent jamais de telles métaclasses).

2 – Le compilateur crée des identifiants uniques en accolant des suffixes/préfixes dépendants du contexte (namespace, classe, type des paramètres, etc.). Cette technique s'appelle name mangling (décoration de nom).

3 – Voir cours. Les points importants sont :

- une table des méthodes virtuelles par classe, contenant les adresses des implémentations des méthodes virtuelles ;
- un pointeur de table des méthodes virtuelles dans chaque objet, qui pointe vers la table correspondante au type de l'objet ;
- un appel de méthode virtuelle consiste à récupérer le pointeur de table dans l'objet, puis l'adresse de la fonction dans la table, et enfin à invoquer cette fonction.

4 – Pas de bonne ou de mauvaise réponse, c'était une question de réflexion. Les points qu'il fallait prendre en considération : un objet peut implémenter plusieurs interfaces ; une interface ne contient pas de données, on peut donc imaginer des solutions qui n'impliquent pas de stocker un « sous-objet » dans l'objet.

La solution la plus simple consiste à faire comme de l'héritage multiple, c'est-à-dire stocker un sous-objet pour chaque interface (et donc avoir un vptr pour chaque interface implémentée). Pour passer l'objet quand l'interface est attendue, on passe simplement l'adresse du vptr correspondant (offset dans l'objet).

On pourrait également imaginer avoir un seul vptr dans chaque objet, qui pointerait vers une table contenant les vptr de chaque interface implémentée ; quand l'interface est attendue, on va chercher le bon vptr dans la table et on le passe. Le problème avec cette approche, c'est qu'on n'a plus l'adresse de l'instance manipulée ; il faut donc trouver un moyen de la conserver (par exemple en représentant les interfaces comme une paire (vptr, adresse instance)).

En .Net, chaque interface est associée à une table, propre à toute l'application. Chaque implémentation de l'interface possède une entrée dans la table. L'entrée pointe vers la table correspondante au type, contenue dans la métaclasse du type (oui, il y a des métaclasses en .Net). Pour ceux qui sont curieux, cet article très intéressant explique un peu les mécanismes internes du CLR : Drill Into .NET Framework Internals to See How the CLR Creates Runtime Objects.

CUDA

1 – Le problème concerne la libération de ressources, en l'occurrence la libération de mémoire sur le CPU (`bufferResultat`) et sur le GPU (`imageGpu`). Si jamais une exception est levée lors de la construction de `imageResultat`, le code de libération des ressources ne sera jamais exécuté, et les ressources jamais libérées.

2 – La meilleure façon de résoudre ce problème est d'utiliser le principe de RAII (Resource Acquisition Is Initialization), qui consiste à libérer les ressources dans les destructeurs d'objets ; dans tous les cas, les destructeurs seront invoqués et les ressources libérées.

Pour ce qui est de la mémoire sur CPU, on pourrait utiliser un smart pointer standard, par exemple `std::unique_ptr<T>`, qui fonctionne aussi bien avec les tableaux qu'avec les objets simples. Une autre solution, probablement meilleure, serait d'utiliser un « vrai » conteneur plutôt qu'un tableau brut, comme `std::vector<T>`.

En ce qui concerne la mémoire sur GPU, c'est plus compliqué, car la libération doit se faire avec la fonction `cudaFree`. Une première solution consiste à spécifier que l'on souhaite utiliser cette fonction, lors de la création du smart pointer :

```
const unsigned int taille = image.size();
const unsigned int tailleTotale = taille * sizeof(char);
char * tmpImageGPU;
cudaMalloc( (void **) &tmpImageGPU, tailleTotale );
std::unique_ptr< char, cudaError_t (*)(void *) > imageGPU( tmpImageGPU, cudaFree );
```

Cette solution n'est pas très pratique car elle implique d'avoir deux variables (pointeur nu + smart pointer), de connaître la signature de `cudaFree`, et de penser à passer `cudaFree` en paramètre. Et il faut être au courant que les smart pointers peuvent être paramétrés par une fonction de suppression ! Le mieux, c'est probablement de créer notre propre smart pointer pour CUDA. Voilà un exemple d'implémentation très simple :

```
template <typename T>
class cuda_array
{
  public:
    cuda_array( size_t size )
      : size_( size )
    {
        cudaMalloc( (void **) &data_, byteSize() );
    }

    ~cuda_array()
    {
        cudaFree( data );
    }

    T * get() const
    {
        return data_;
    }

    size_t size() const
    {
        return size_;
    }

    size_t byteSize() const
    {
        return size_ * sizeof(T);
    }

  private:
    T * data_;
    size_t size_;
};
```

(Comme on stocke la taille, on pourrait écrire des petites fonctions pour faciliter les copies CPU-> GPU et GPU->CPU.)

Le code devient alors :

```
CImg getImageTransformee( CImg<char> const & image )
{
    const unsigned int taille = image.size();

    cuda_array<char> imageGPU( taille );
    cudaMemcpy( imageGPU.get(), image.data(), imageGPU.byteSize(),
                cudaMemcpyDeviceToHost );

    operation_quelconque<<< taille / 512 + 1, 512 >>>( imageGPU.get(), taille );

    std::vector<char> bufferResultat( taille );
    cudaMemcpy( &bufferResultat[0], imageGPU.get(), imageGPU.byteSize(),
                cudaMemcpyDeviceToHost );

    return CImg( &bufferResultat[0], image.width(), image.height(),
                 image.depth(), image.spectrum(), false );
} // La mémoire CPU est libérée par std::vector, celle du GPU par cuda_array
```