Formation C++ Ubisoft - Module 2

Romain Arcila[1,2]
Charles de Rousiers[1]

14 mars 2009

[1] INRIA Grenoble
[2] Liris -CNRS Lyon

Plan

Gestion de la mémoire

Contexte

- De quoi va-t-on parler ?
- Que va-t-on voir ?

- De gestion de la mémoire en C++ : allocation en générale
- L'allocation statique et dynamique
- Les opérateurs d'allocation d'une manière générale
- De techniques de gestion de mémoire
- D'erreurs typiques, de fuites mémoire ...

Plan de la séance

Déroulement de la séance

- Séparation de la journée en **deux parties**

Matin

1. Rappels sur la mémoire
2. Notion d'allocation et les outils fournis par le C++ pour sa gestion
3. Notion d'exception dans le cadre de l'allocation de la mémoire

Après-midi

1. Notion d'allocateur, gestion personnalisée de l'allocation
2. Présentation des pointeurs intelligents
3. Quelques astuces et idioms pour faciliter la gestion de la mémoire

Déroulement de la séance

- Exercices proposés

Matin

1. Debug de programme (fuites mémoire, ...)
2. Surcharges d'opérateurs d'allocation

Après midi

1. Implémentation d'un allocateur de mémoire : pool mémoire
2. Implémentation d'un pointeur intelligent

Plan

Fonctionnement CPU

- Le CPU est une machine à état : exécution pas à pas d'instructions qui composent le programme

Fonctionnement (Sur plusieurs cycles

1. Lecture de l'instruction
2. Décodage
3. Récupération des valeurs (opérandes)
4. Exécution de l'instruction

- Besoin de données (instructions, adresses, valeurs, ...)
- Récupération de ces données dans différentes zones de mémoire : fait appel à une hiérarchie de mémoire

Types de mémoire

- Existance de plusieurs types de mémoire au sein d'un ordinateur :
  - Registres
  - Caches
  - Mémoire centrale (RAM)
  - Mémoire de stockage - swap (DD)
- Différenciation de ces mémoires :
  - Par leur taille
  - Par leur rapidité

Raison

Plus c'est rapide, plus ça coûte cher !

Types de mémoire

Registres

- Type de mémoire manipulé directement pas le CPU.
- Extrêmement rapide.
- Aucune latence d'accès.
- Taille très limitée.

Types de mémoire

Caches

- Type de mémoire accédée par le CPU quand les données ne sont pas dans les registres
- Tampon entre le CPU et la mémoire centrale
- Latence très faible.
- En général réparti sur 2 à 3 niveaux
- Taille limitée.
- Sur les CPU multi-cores peuvent être partagés (permet communication inter-core)

Types de mémoire

Mémoire centrale (RAM)

- Mémoire où se situe principalement les données accédées par les programes
- Tailles importante
- Latence moyenne

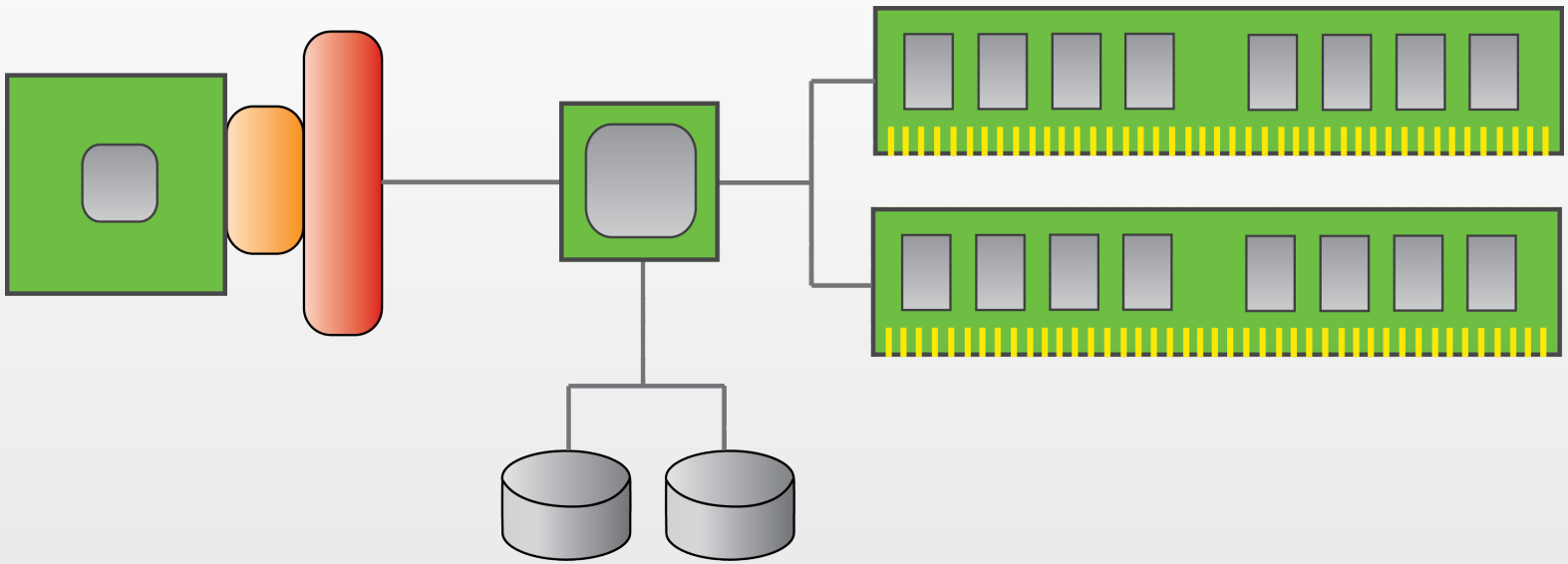
Types de mémoire

Mémoire de stockage - swap (DD)

- Sert pour le stockage des données
- Latence très importante
- Taille très importante
- Peut servir de swap quand la mémoire centrale est surchargée (principale de la mémoire virtuelle)
- Utilisation directe entraîne des chutes de performances

Types de mémoire

- **Schéma** : Communication entre tous les types de mémoire



Durée de vie

Définition

Toute variable dans une programme a une durée de vie déterminée : Caratérise la validité de manupulation d'une variable

- La durée de vie d'une variable dépend :
  - Du modificateur : auto, static, ...
  - Du type d'allocation : statique / dynamique

Pour les allocations statiques

- auto, register, volatil : durée de bloc (pour les variables situées dans le bloc principal, hors main, durée de vie du programme)
- static : durée du programme

Pour les allocations dynamiques

- Existence à partir l'allocation
- Fin d'existence à la désallocation

Durée de vie

Détail des principaux cas de figures

```
int a;

void foo() { int d; static e; }

int main()
{
        int b;
        {
                int c;
                foo();
        }
        foo();
        return 0;
}
```

Visibilité

Définition

Détermine les zones d'utilisation d'une variable : Caratérise la validité d'accès d'une variable

- Dépend principalement :
  - Des blocs (fonctions, boucles, conditions, namespace, ...)
  - Des modificateurs au niveau fichier
  - Des déclarations
  - ...

Visibilité

Détail des principaux cas de figures

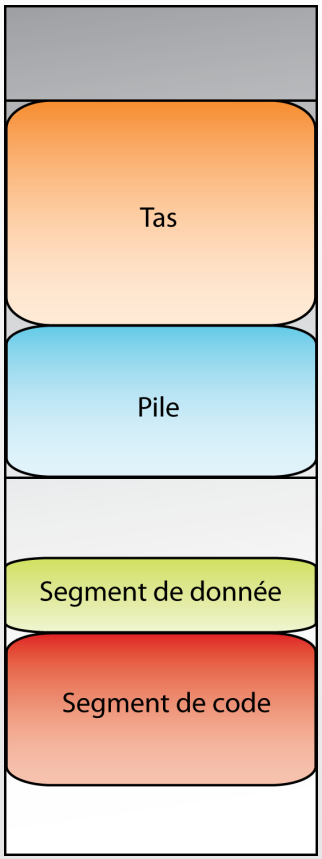
```
static int a;

int f;
extern int g;

void Foo() {
        int b;
}

namespace Ns {
        int c;
}

int main() {
        int d;
        {
                int e = Ns::c;
        }
        return 0;
}
```

Accès mémoire

- Du point de vue du programme : plusieurs type/zones de mémoires utilisable
  - Le tas
  - La pile
  - Le segment de code
- Distribution des variables dans les différentes zones suivant leur type d'allocation ou le type de variable
  - Variables allouées statiquement : sur la pile
  - Variables allouées dynamiquement : sur le tas
  - Constantes : dans le segment de données
  - Constantes : dans le segment de code

Mémoire dans le code

- **Schéma** :Les différentes zones que l'on retrouve dans la mémoire



Plan

Généralités

- Les processeurs alignent les données sur leur taille native (32/64 bits)
  - Raison de performance : car optimisés pour récupérer ce type de données
- Sur un processeur 32 bits : Alignement sur les adresses multiples de 4 octets (32bits)

Code

```
struct A {
        char a;
        int b;
        char c;
        short d;
};
struct B {
        char a, c;
        short d;
        int b;
};
```

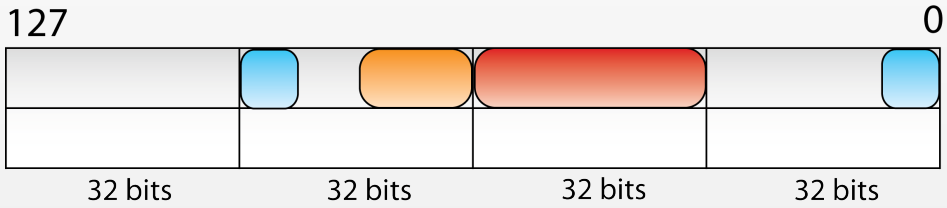
Conséquence

sizeof(A) != sizeof(B)

Exemple d'alignement

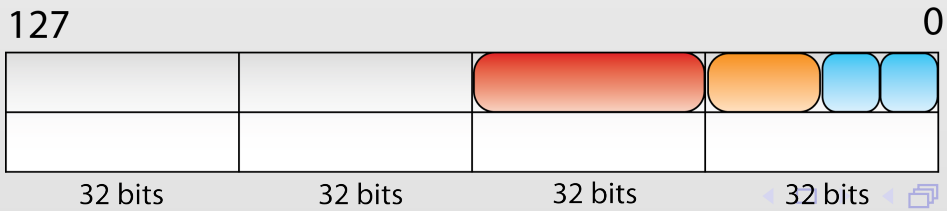
Code

```
struct A {
        char a;
        int b;
        char c;
        short d;
};
```



Code

```
struct B {
        char a, c;
        short d;
        int b;
};
```



Les alignements types

- Taille des types primitifs (sur un processeur 32 bits) :
  - sizeof(char) = 1
  - sizeof(short) = 2
  - sizeof(int) = 4
  - sizeof(float) = 4
  - sizeof(double) = 8
- Alignement par défaut des types primitifs (sur les processeurs 32 bits) :
  - Double : multiple de 8 octets
  - Short : multiple de 2 octets
  - Int : multiple de 4 octets
  - Float : multiple de 4 octets
  - Char : n'importe où

Conséquences

Conséquences directes

- Si l'on souhaite qu'une structure soit la plus compacte possible : alignement sur 1 octets mais pas performant
- Si l'on souhaite rapididité et efficacité : alignement sur 4 octects

Changement d'alignement

- Possibilité d'influer l'alignement du code généré : utilisation de **'pragma'** du compilateur

Remarque

L'influence peut avoir lieu sur les deux types d'allocation

- Allocation statique : change l'alignement de la pile
- Allocation dynamique : change l'alignememt de la mémoire allouée par malloc

- Changement de l'alignement influence
  - La taille du code généré
  - La vitesse d'exécution du code

Remarque

Requis pour certains jeux d'instructions vectoriels (SSE1, SSE2, SSE3...)

Padding

- Possiblité de 'forcer' un alignement : notion de Padding

Principe

Ajout de données inutiles (uchar, ushort, uint) pour forcer un alignement particulier sur une structure de donnée

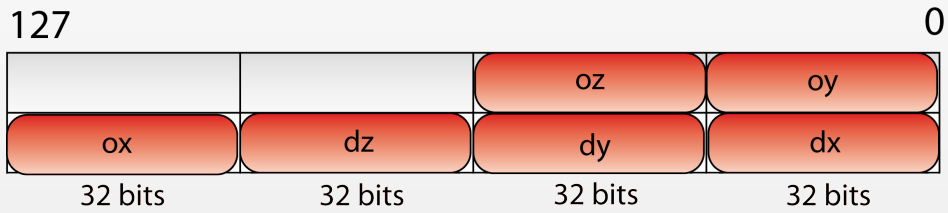
- Permet ainsi de combler les trous lié à l'alignement.

Exemple de padding

- **Exemple** : utilisation du padding pour avoir un alignement sur 128 bits

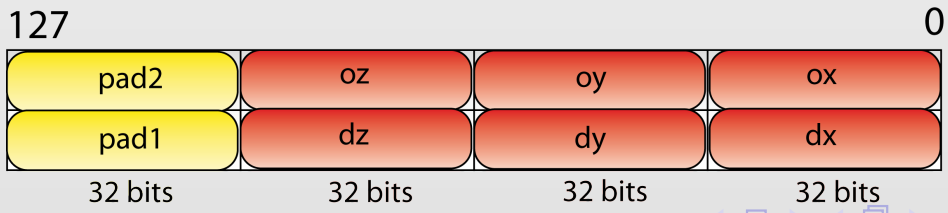
Code

```
struct Ray {
        float dx, dy, dz;
        float ox, oy, oz;
};
```



Code

```
struct Ray {
        float dx, dy, dz, pad1;
        float ox, oy, oz, pad2;
};
```



Contraintes d'alignement

Problème

- Une classe contient un attribut alloué statiquement qui nécessite un alignement particulier
- Que se passe-t-il si on alloue un objet de cette classe dynamiquement ?

- **Réponse** : dépend de l'alignement de l'opérateur d'allocation
- Peut conduire à des problèmes car potentiellement ne respect pas l'alignement souhaité

Solution

Redéfinition des opérateurs d'allocation de la classe en utilisant un allocateur qui respecte les contraintes d'alignement

Bonnes pratiques

Bonnes pratiques

Surcharge systématique des opérateurs d'allocation dynamique globaux / locaux

- Connaissances tardives
  - Des fréquences d'allocation
  - De l'occupation mémoire
  - ...

Remarque

Permet de réagir rapidement, en mettant en place les allocateurs appropriés

- Facilite le déploiement des allocateurs

Plan

Allocation : différence entre C et C++

- Les fonctionnalités de l'allocation en C et en C++ :

En C

- Allocation de la mémoire

En C++

- Allocation de la mémoire
- Initialisation de la mémoire allouée en appelant le constructeur

- Les fonctionnalités de la désallocation en C et en C++ :

En C

- Désallocation de la mémoire

En C++

- Traitement de la mémoire désallouée en appelant le destructeur
- Désallocation de la mémoire

Attention

Attention aux types primitifs : pas de constructeur par défaut donc initialisation **'aléatoire'**

Allocation statique

- Pour l'allocation de mémoire statique : mémoire allouée sur la pile

Construction/Allocation

- Pour les types non primitifs : appel du constructeur
- Pour les types primitifs : pas d'initialisation de la valeur 'aléatoire'

Destruction/Désallocation

- Pour les types non primitifs : appel du destructeur

Allocation statique

- **Exemple** : Construction d'un objet simple

Code

```
int main()
{
        A a;
        A aprime;
        return 0;
}
```

Allocation statique

- Cas des tableaux statiques :
  - Allocation des objets de manière contiguë en mémoire (à l'alignement près)
  - En C++ : initialisation de tous les objets en appelant leur constructeur

Problème

L'objet doit impérativement posséder un constructeur par défaut : Nécessaire pour l'initialisation de chaque objet

- Possibilité de passer outre le constructeur par défaut : utilisation d'une liste d'initialisation

Attention

Encore une fois cette initialisation n'est vrai que pour les types non primitifs ! Pour les types primitifs, initialisation avec une valeur 'aléatoire'

Allocation statique

- **Exemple** : Construction d'un tableau d'objets simples

Code

```
int main()
{
        A a[10];
        A b[] = {A(0), A(9), A(45)};
        return 0;
}
```

Allocation dynamique

Contexte

Pourquoi a-t-on besoin d'une allocation dynamique ?

- Nombre/taille figé mais dépendant de paramètres entrés
- Nombre/taille variable durant la durée de vie d'un objet
- Éviter un débordement de pile (Portabilité : taille de pile inconnue)
- Eviter de déclarer des objets énormes
  - Surconsommation mémoire
  - Ralentissement de l'allocateur
- Durée de vie découplée de la pile
- ...

Allocation dynamique

- **Principe** : l'allocation des objets se fait sur le tas.
  - Nécessite de récupérer un morceau de mémoire du tas (appel aux fonctions l'OS)
- En C, utilisation des primitives (fonctions masquant les fonctionnalités d'allocation de l'OS) :
  - malloc : pour allouer la mémoire
  - free : pour désallouer la mémoire

Prototypes

```
// Retourne un pointeur sur la zone memoire allouee de taille _size
void* malloc(size_t _size);

// Libere la zone memoire referencee par _ptr
void free(void* _ptr);
```

Allocation dynamique

- En C++, ajout de fonctionnalitées d'initialisation pour l'allocation d'objet
  1. Allocation d'une zone mémoire de la taille de l'objet
  2. Appel du constructeur pour initialiser la zone mémoire
- Vue macro de la fonction new : l'allocation de la zone mémoire s'appuie sur la fonction 'malloc'

New

```
T* new<T>(params)
{
        T* mem = static\_cast<T>(T::new(sizeof(T));
        mem->T(params);
        return mem;
}
```

Allocation dynamique

- De la même manière : ajout de fonctionnalités de finalisation des objets pour la désallocation
  1. Appel du destructeur pour effectuer les derniers traitements
  2. Désallocation de la zone mémoire
- Vue macro de la fonction new : l'allocation de la zone mémoire s'appuie sur la fonction 'free'

Delete

```
void delete<T>(T* _mem)
{
        mem->~T();
        T::delete( static_cast<void*>(mem));
}
```

Allocation dynamique

- Extension aux tableaux d'objets
  - Allocation : Utilisation de l'opérateur new[]
  - Désallocation : Utilisation de l'opérateur delete]
- Assure l'initialisation de chaque objet du tableau en appelant le constructeur par défaut

Attention

Éviter les mélanges :

- new / delete[]
- new[] / delete

$\Rightarrow$ Fuites mémoire assurées !

Allocation dynamique

- Possibilité d'utiliser new / delete pour l'allocation
  - Fonctionnalité 'native' pour le C++
  - Assure une gestion automatique de l'initialisation et de la destruction
- Mais également possibilité d'utiliser malloc / free
  - Issue du C mais tout à fait utilisable en C++
  - Permet une gestion moins 'contrainte' de la mémoire avec un contrôle plus bas niveau
  - Possibilité d'utiliser realloc pour étendre la mémoire
  - Attention cependant car pas d'appel automatique au destructeur !

Attention

Éviter les mélanges :

- malloc / delete
- new / free

⇒ Comportement indéterminé !

Opérateur d'allocation

- À présent définition de tous les opérateurs d'allocation dynamique du C++
  - Définition de leur prototype
  - Vue d'une implémentation pseudo standard

But

Avoir une idée exacte du prototype et d'une implémentation pour pouvoir réaliser ces propres opérateurs

Opérateur d'allocation

- Opérateur new

Prototype

```
void* operator new(std::size_t _size);
```

Implémentation pseudo standard

```
void* operator new(std::size_t _size)
{
        void * ptr = malloc(_size);
        return ptr;
}
```

Opérateur d'allocation

- Opérateur delete

Prototype

```
void operator delete(void* _ptr)
```

Implémentation pseudo standard

```
void operator delete(void* _ptr)
{
        free(_ptr);
}
```

Opérateur d'allocation

- Opérateur new[]

Prototype

```
void* operator new[](std::size_t _size);
```

Implémentation pseudo standard

```
void* operator new[](std::size_t _size)
{
        void * ptr = malloc(_size);
        return ptr;
}
```

Opérateur d'allocation

- Opérateur delete[]

Prototype

```
void operator delete [](void* _ptr);
```

Implémentation pseudo standard

```
void operator delete [](void* _ptr)
{
        free(_ptr);
}
```

Surcharge opérateur d'allocation

- Possibilité de surcharger ou remplacer ces opérateurs
- Plusieurs raisons possibles pour cette redéfinition ou cette surcharge :
  - Permet d'avoir une politique d'allocation différente afin d'optimiser les perfomances
  - Prendre en compte certain type de matériel de stockage
  - ...

Remarque

Plus fréquent que ce que l'on pourrait croire !

Surcharge opérateur d'allocation

- **Redéfinition** : utilisation du prototype de base
- **Surcharge** : Ajout de nouveaux parmètres,
  - Présence obligatoire de la taille mémoire à allouer
  - Change la syntaxe d'appel

Exemple

```
// Surcharge de l'operateur global new
void* operator new(std::size_t _size, A _a, B _b);

// Exemple d'appel pour utiliser cet operateur
C* c = new(_a,_b) C(params_pour_C);
```

Astuce

Si les paramètres sont des constants ou des objets globaux : utiliser une macro

- Automatise le remplacement des appels à new
- Rend transparent l'utilisation de cet opérateur

Surcharge opérateurs d'allocation

- Existence de plusieurs niveaux de surcharge / redéfinition

Niveau global

- Remplace/surcharge des opérateurs globaux par défaut
- Nécessite de modifier la syntaxe

Niveau classe

- Modification de l'allocation pour un type d'objet seulement
- Les opérateurs new, delete new[] et delete[] sont méthodes static !

Surcharge opérateur d'allocation

- **Exemple** : surcharge des opérateurs new et delete globaux

Exemple

```
void* operator new(std::size_t _size) {
        return Memory::Allocator::Instance().Allocate(_size);
}

void operator delete(void* _ptr) {
        Memory::Allocator::Instance().Desallocate(_ptr);
}

class A {
        public: void Foo() { ... }
};

int main() {
        A* a = new A();
        a->Foo();
        delete a;
        return 0;
}
```

Surcharge opérateur d'allocation

- **Exemple** : surcharge des opérateurs new et delete dans une classe

Exemple

```
class A
{
        public:
                static void* operator new(std::size_t _size) {
                        return Memory::Pool<A>::Instance().Allocate(_size);
                }

                static void operator delete(void* _ptr) {
                        Memory::Pool<A>::Instance().Desallocate(_ptr);
                }

                void Foo() { ... }
};

int main() {
        A* a = new A();
        a->Foo();
        delete a;
        return 0;
}
```

Surcharge opérateur d'allocation

- Existance d'une surcharge par défaut de l'opérateur new : placement new
  - Place l'objet à un endroit choisi
  - Permet d'utiliser une allocation non standard
  - Permet de conserver une homogénéité dans la syntaxe d'appel

Prototype

```
// _ptr pointe sur une zone memoire dont la taille est d'au moins _size
void* operator new(std::size_t _size, void* _ptr);
```

- Attention cependant pour la désallocation :
  - Pas possible d'appeler opérateur delete
  - Nécessite d'appeler le destructeur à la main
  - Puis faire une désallocation à la main

Surcharge opérateur d'allocation

- **Exemple** : utilisation de l'opérateur new placement

Code

```
{
        // Allocation d'un bloc de memoire de la taille d'un objet A
        unsigned char memBlock[sizeof(A)];

        // Appel a l'operateur placement new (assure l' initisation de l'objet)
        A* a = new(memblock) A();

        // Utilisation de a
        a->Foo();

        // Appel a la main du destructeur pour assure la finalisation de l'objet
        a->~Foo();
}
```

Plan

Rappel allocation dynamique en C++

Rappel

Rappel sur les étapes d'allocations dynamiques

- Phase d'allocation
  1. Allocation de la mémoire sur le tas
  2. Appel de constructeur pour initialiser la mémoire
- Phase de libération
  1. Appel du destructeur pour libérer les ressources
  2. Désallocation de la mémoire
- Pour les tableaux :
  1. Construction : Appel du constructeur par défaut pour tous les objets
  2. Destruction : Appel du destructeur pour tous les objets

Exceptions

Contexte

Gestion des exceptions lors de la manipulation d'allocations dynamiques peut devenir primordiale

- Risque de fuite mémoire
- Risque de pointeurs mal/non initialisés
- ...

- Des exceptions peuvent être lancées à plusieurs niveaux :
  - Constructeur de l'objet
  - Destructeur de l'objet
  - Opérateur new / new[]
  - Opérateur delete / delete[]

Remarque

Les fonctions malloc et free ne levent pas d'exceptions : retour d'erreur uniquement

Rappels sur les exceptions

- Permet de gérer les cas d'erreur

Conseil

Lever les exceptions par référence (de préférence constante) : évite les copies d'objet non nécessaires

- **Fonctionnement** :
  - throw pour lever une exception
  - bloc try / catch pour capturer et traiter une exception
- Appel du callback terminate() qui tue le programme
  - Si exception levée et aucun block try/catch
  - Si la situation devient ingérable (cas des exceptions dans le destructeur)
- Modification possible du callback : méthode set_terminate
- Exception standard std : :exception
  - Fichier d'entête $<$ *exception* $>$
  - Base conseillée pour dériver ces classes d'exceptions
  - Méthode what() : description de l'erreur

Exception avec l'opérateur new

- Situation possible entrainant une erreur dans new :
  - Pénurie de mémoire
  - Pénurie d'adresses virtuelles
  - Aucune plage contigüe assez grande
- Suivant les compilateurs l'appel à new peut :
  - Lancement exceptions de type std : :bad_alloc (le standard !)
  - Renvoyer un pointeur NULL (ex : VC++6)

Exception avec l'opérateur new

- **Fonctionnement** :
  1. Lorsque l'opérateur new par défaut n'arrive pas à allouer la mémoire nécessaire : appel du callback new_handler
  2. Comportement par défaut : lève une exception de type std : :bad_alloc
- Possibilité pour l'utilisateur de définir un nouveau callback pour modifier le comportement par défaut de l'opérateur new
  - Spécifie le nouveau callback : set_new_handler (prototype : void CallBack(void))
- Le callback doit faire une des actions suivantes :
  - Tenter d'allouer à nouveau la mémoire
  - Lever une exception de type std : :bad_alloc (ou dérivée)
  - Sortir du programme

Remarque

Si la fonction ne termine pas le programme avec terminate() et ne lève pas d'exception, appel à nouveau du callback de manière cyclique

Exception avec l'opérateur new

- **Exemple**

Exemple

```
void CustomCallback() {
        printf("Allocation impossible\n");
}

int main(){
        set_new_handler(CustomCallback);
        int * a = new int[std::numeric_limits<int>::max()];
        delete [] a;
        return 0;
}
```

Exception avec l'opérateur new

- Existance d'un autre new (non présenté auparavant) : new (std : :nothrow)
  - Ne lève aucune exception
  - Renvoie NULL en cas d'impossibilité d'allocation de mémoire

Exemple

```
int main(){
        int * a = new (std::nothrow) int[std::numeric_limits<int>::max()];
        if (a == NULL)
                printf("Echec de l'allocation \n");
        else
                delete [] a;
        return 0;
}
```

Exception dans le constructeur

Contexte

- Pas de type de retour pour les constructeurs : pas possible d'utiliser des codes d'erreur
- Meilleure façon de signaler un problème dans le constructeur : jeter une exception

- Existence de deux situations 'délicates'
  - Cas où l'on doit lancer une exception dans le constructeur (pour signaler une erreur)
  - Cas où l'on peut recevoir une exception dans le constructeur

Exception dans le constructeur

Problème

Si le constructeur jette une exception, le destructeur de l'objet n'est pas exécuté : Peut poser des problèmes

- **Exemple** : Constructeur acquière des ressources devant être libérées à la destruction :
  - Allocation de blocs de mémoire
  - Ouverture de fichiers
  - Verrou sur un sémaphore

Exemple

```
class A {
        public:
                A()     {
                        mem = new int[100];
                        if (...)  // Fait quelque chose ...
                        else throw;
                }
                ~A() {  delete [] mem; }
        private:
                int * mem;
};
```

Exception dans le constructeur

- Cas de lancement d'exceptions dans le constructeur
  - Avant de lancer l'exception, veiller à : libérer toutes les ressources acquises

Code

```
class A {
        public :
                A()     {
                        mem = new int[100];
                        if (...)
                                // ...
                        else    {
                                delete [] mem;
                                throw;
                        }
                }
                ~A() {  delete [] mem; }
        private :
                int * mem;
};
```

Exception dans le constructeur

- Cas de réception d'exception dans le constructeur
  - Catcher les exceptions pouvant potentiellement être levées dans le constructeur
  - Libérer les resources acquises dans le constructeur ou traiter l'erreur
  - Relancer l'erreur (si erreur non traitée)

Remarque

Préférable de mettre les attributs pointeurs d'un objet à NULL (dans la liste d'initialisation)

- Permet un appel de delete / delete[] sans vérification : ptr != NULL
- Évite de provoquer des erreurs supplémentaires

Exception dans le constructeur

- **Exemple**

Code

```
class A {
        public :
                A():mem(NULL){
                        try{
                                mem = new int[100];
                                Foo(); // Peut lancer une exception
                        }
                        catch(std::exception& ex) {
                                delete [] mem;
                                throw;
                        }
                }

                ~A() { delete [] mem; }

        private :
                int * mem;
};
```

Exception dans le constructeur

- Cas des exceptions levées par un objet alloué dynamiquement

Problème

- Lors de l'allocation dynamique d'objet : le constructeur peut lever une exception
- Dans le catch faut-il détruire cet objet ?

Principe

1. Utilisation de l'opérateur void* operator new(std : :size_t _size) pour allouer l'espace mémoire nécessaire
2. Le constructeur de l'objet est appelé pour initialiser la mémoire

Si une exception est levée dans la seconde étape : attraper dans un bloc try / catch : assure aucune fuite mémoire

Exception dans le constructeur

- **Exemple**

Code

```
class A {
        public:
                A():mem(NULL) {
                        try{
                                mem = new BigObject();
                        }
                        catch(std::exception& ex) {
                                // delete non necessaire et aucun fuite memoire
                                throw;
                        }
                }

                ~A() { delete mem; }

        private:
                BigObject* mem;
};
```

Exception dans le destructeur

- Est-il possible de lever une exception dans le destructeur ?
- Est-ce une bonne idée ?

Problème

1. Quand une exception est levée : phase de déroulement de la pile ('stack unwinding')
2. Destruction de tous les objets entre le throw et le catch
3. Si le destructeur d'un objet lance une autre exception

⇒ Deux exceptions en parallèle : laquelle choisir ?

- Situation ambiguë : appel de la fonction terminate() !

À faire

- **Règle** : ne jamais lancer d'exception dans un destructeur qui pourrait être appelé durant la phase de déroulement de la pile
- **Plus simple** : ne jamais lancer d'exception dans un destructeur !

Exception dans le destructeur

- **Exemple**

Code

```
class A {
        public:
                A() { }
                ~A() { throw; }
};
class B {
        public:
                B() { }
                ~B() { throw; }
};
int main() {
        try     {
                B b;
                { A a; }
        }
        catch (...) {
                // ...
        }
}
```

Exception dans le delete

- Implémentation par défaut de delete / delete[] : aucune exception levée
- Possibilité de lever une exception dans vos implémentations personnalisées

Problème

- Quand une exception est levée : phase de déroulement de la pile ('stack unwinding')
- Destruction de tous les objets entre le throw et le catch
- Si le destructeur d'un objet lance une autre exception

⇒ Retour du problème de 'stack unwinding' !

- **Conclusion** : Éviter de lever une exception dans vos opérateurs delete

Exception et tableaux

- Allocation dynamique de tableaux
  - Initialisation de la mémoire
  - Appel du constructeur pour tous les objets

Problème

Que se passe-t-il si un des constructeurs lève une exception ?

- Faut-il détruire tous les objets ?
- Faut-il libérer la mémoire ?

- Situation similaire à ce qui a déja été vu
  - Phase de construction des objets protégée dans un bloc try / catch
  - Destruction et désallocation de la mémoire dans new

Recommendation

Recommendation

Quelques recommendations pour éviter les problèmes liés aux exceptions

- Penser à vérifier les exceptions pouvant être lancées dans le constructeur
- Utiliser des pointeurs intelligents pour manipuler vos ressources allouées dynamiquement
  - Automatise la libérartion des ressources : évite les oublis
- Initialiser vos pointeur à NULL pour éviter les delete sur des zones non allouées

Plan

Généralités

- **Problème** : allocateur par défaut alloue selon même processus la mémoire dynamique indépendamment :
  - De la taille à allouée
  - De la fréquence des allocations
  - Des caractéristiques souhaitées de mémoire (alignement, fragmentation ...)
- **Conséquence** : gestion non optimale de l'allocation mémoire
  - Sur-appel des fonctions d'allocation et de désallocation de l'OS
  - Perte de performances
  - Impossible d'utiliser certaines instructions (SSE, ...)

Solution

Gérer d'une manière personnalisée l'allocation de sa mémoire en définissant des allocateurs personnalisés

Généralités

- **Exemple** : Allocation de nombreux petits objets
- Problèmes sous-jacents
  - Nécessite N appels à la fonction malloc
  - Nécessite N appels à la fonction free

Conséquence

Politique d'allocation non adaptée pour ce type d'objet : redéfinition des opérateurs new et delete pour cette classe

- **Fonctionnement**
  - Définition d'un pool de mémoire commum
  - Chaque demande d'allocation prend une petite partie du pool mémoire

Généralité

- **Exemple** : allocateur type 'memory pool' ultra simpliste

Code

```
class MemoryPoolA
{
        public:
                MemoryPool() {
                        memory = malloc(sizeof(A)*MAX_OBJECTS);
                        index  = 0;
                }

                ~MemoryPool() {
                        free(memory);
                }

                void* Allocate() {
                        assert(index<MAX_OBJECTS);    // Full
                        return memory + sizeof(A)*index;
                }

                void Desallocate(void* _ptr) {
                        assert(_ptr>memory);
                        assert(_ptr<&memory[MAX_OBJECTS]);
                }
        private:
                unsigned char* memory;
                int index;
};
```

Généralités

- **Autre exemple** : Alignement de la mémoire

Exemple du SSE

Traitement des données en parallèles

- Utilisation de type primitif de 128bits
- Opération attend alignement sur 128bits

(Plus de détail dans le cours d'optimisation)

- Avec une allocation standard
  - Alignement sur 32 bits
- Avec une allocation spécialisée
  - Force l'alignement sur une taille souhaitée

Allocateur

- **Autre exemple** : Alignement de la mémoire (suite)

Code

```
class SSEAllocator
{
        public:
                SSEAllocator(){}

                ~SSEAllocator() {}

                void* Allocate(std::size_t _size) {
                        return _mm_alloc(_size,16);
                }

                void Desallocate(void* _ptr) {
                        _mm_free(_ptr);
                }
};

void* operator new(std::size_t _size) {
        return SSEAllocator::Instance().Allocate(_size);
}

void operator delete(void* _ptr) {
       SSEAllocator::Instance().Desallocate(_ptr);
}
```

Statégie d'allocation

Context

Création gestionnaire de mémoire générique (non spécifique à un type objet : taille variable)

- Appropriation d'un pool de mémoire
- Distribution de ce pool de mémoire

- Plusieurs stratégies de distribution possibles :
  - First fit : prend le premier bloc de mémoire suffisamment grand
  - Worst fit : prend le plus grand bloc de mémoire disponible
  - Best fit : prend le plus petit bloc possible qui ait une taille au moins égale à la taille du bloc à allouer.

Statégie d'allocation

Contexte

Création gestionnaire de mémoire spécifique (taille fixe)

- Appropriation d'un pool de mémoire
- Distribution de ce pool de mémoire

- Plusieurs stratégies de distribution possible autour d'un partionnement du pool en zone ('chunk') :
  - Allocate & Delete :
    - Distribue les éléments libres du chunk **en avant**
    - Quand un chunk est plein construction d'un autre chunk
    - Quand un chunk est totalement libéré : recyclage ou destruction
  - Reuse : prends le plus grand bloc de mémoire disponible sur toute la mémoire
    - Distribue les éléments libre du chunk **par recherche**
    - Quand un chunk est plein construction d'un autre chunk

Stratégie d'allocation

Cas rupture de mémoire

- Création de nouveaux chunks
- Quelle nouvelle quantité allouée ? Plusieurs possibilités d'allocation :
  - Un simple nouveau chunk
  - Deux fois le nombre déjà allouée
  - ...

Cas de libération

- Réutilisation de la mémoire libre
- Destruction de la mémoire
- ...

Statégie d'allocation

- Solution utilisée dans la STL
  - Allocation d'un taille initiale (taille par défaut ou spécifiée par l'utilisateur)
  - En cas de manque de mémoire, agrandi la zone mémoire de deux fois la taille courante
- Stratégie intéressante
  - Performance en coût d'allocation
  - Performance en consomation mémoire

Stratégie d'allocation

- **Exemple** : Implémentation d'une classe Vector

Code

```
class IntVector
{
public:
        IntVector(int _initialSize):size(_initialSize) {
                memory = malloc(size*sizeof(int));
        }

        ~IntVector() {
                free(memory);
        }
        ...

        void push_back(int _value) {
                if(index>=size) {
                        size *= 2;
                        memory = realloc(memory, size*sizeof(int));
                }
                memory[index++] = _value;
        }

        int *memory;
        int size, index;
}
```

Allocateur personnalisé

Mise en place

- Création d'un allocateur personnalisé
- Dispose de méthode d'allocation / désallocation

⇒ Mise en place de l'allocateur : Quelle interface utiliser ?

- Utilisation de l'interface ad-hoc : revient à utiliser simplement 'malloc' et 'free'
  - Utilisation brute des méthodes Allocate et Desallocate
  - Appel constructeur et destruction à la charge de l'utilisateur
  - Possibilité de passer par un warpper person (intérêt ?)
- Utilisation des opérateurs d'allocation standards
  - Encapsulation
  - Automatisation l'appel au constructeur et destructeur
- Mise en place avec les opérateurs standards
  - Passage par un singleton : transparent
  - Passage par paramètre : nécessite une modification des appels

Allocateur personnalisé

- **Exemple** : illustration des possibiltés

Code - Avec utilisation brute

```
// Avec utilisation brute
Allocator alloc;
A* a = reinterpret_cast<A*>(alloc.Allocate(sizeof(A)));
a->A();
a->~A();
alloc.Desallocate( reinterpret_cast<void*>(a));

// Avec placement new
void * mem = alloc.Allocate(sizeof(A));
A* a = new(mem) A();
a->~A();
alloc.Desallocate(mem);
```

Allocateur personnalisé

- **Exemple** : illustration des possibiltés

Code - Avec opérateur new / delete

```
// Avec new surcharge
A* a = new(alloc) A();
delete a;

// Avec une utilisation transparente
A* a = new A();
delete a;
```

Principe

Contexte

La gestion de la mémoire peut devenir rapidement complexe : partage d'objets, échange d'objets, ...

- **Exemple** : gestion de ressources
  - Qui alloue ?
  - Qui détruit ?

Problèmes

- Difficulté de la gestion de propriété
- Validité d'existance
- Risques majeurs de fuites mémoire

- **Solution** : automatiser la gestion mémoire pour le programmeur
  - Responsable seulement de l'allocation
  - Automatisation de la désallocation quand l'objet n'est plus utilisé

Principe

- La gestion automatique de la mémoire permet d'éviter :
  - Les fuites mémoire : oublie de faire une delete sur un objet alloué dynamiquement

Exemple

```
void readFile(char*);

void test(bool flag) {
        char* buf = new char[100];

        if (flag) {
                readFile(buf);
                delete [] buf;
        }
}
```

Principe

- Ceci doit permettre d'éviter (suite) :
  - Libération prématurée : utilisation d'un pointeur qui a été libérée auparavant

Exemple

```
void eval(int*);

void test() {
        int* p1 = new int;
        int* p2 = p1;
        *p1 = 1;
        eval(p1);
        delete p1;
        *p2 = 2;   // memoire deja supprimee
        eval(p2);
}
```

  - Fragmentation de la mémoire : arrive quand un nombre important d'allocation et de désallocation sont réalisées
    - Accés moins performant
    - Défauts de cache plus fréquent

Utilisation

- Utilisation importante :
  - Dans les langages managés (Java, .net, langage de script ...)
  - Dans les couches haut niveau où l'impact mémoire sur les performances est faible

  $\Rightarrow$ Facilite le développement
- Répond aux principaux problèmes :
  - Évite les fuites mémoires
  - Évite de libérer prématurément des ressources
  - Évite de libérer plusieurs fois une ressources
  - Évite la fragmentation de ma mémoire

Algorithmes

- Définition : objet atteignable
  - Distinction entre objet atteignable et non atteignable
  - Tout élément référencé par un objet atteignable est lui même atteignable
- Identification des objets non nécessaire :
  - Tourver les objets qui ne sont plus accédés
  - Désallouer ces objets
- Algorithme de base : Utilisation de trois ensemble (noir, gris, blanc)
  - Noir vide
  - Gris contient les racines
  - Blanc le reste

Principe de base

Invariant : l'ensemble noir ne référence pas l'ensemble blanc

- Place tous les éléments dans l'ensemble blanc
- Déplace les racines dans l'ensemble noir
- Déplace tous les objets référencés dans l'ensemble gris
- Détruit tout ce qui reste dans l'ensenble blanc

Types

- Existance de différents type d'algorithme de GC

Marquage et nettoyage (mark and sweep)

- Objet blanc ou noir
- Copie les objets noirs dans une autre zone et détruit tout la zone ⇒ Utile quand on a de nombreux objets alloués et détruit rapidement

Comptage de référence (references counting)

- Association d'un compteur de références
- Problème avec les cycles
- Coûteux en temps de comptage

- Et d'aute encore : Récupérateur à génération (generational GC), Conservatif et précis (conservative vs. precise) ...

Types

- Souvent utilisation mixte de plusieurs algorithmes

Principales stratégies

1. Périodiquement scanne les données du programme (objets du tas utilisés/référencés par un pointeur)
2. Libère tous les objets qui n'ont pas été marqués

- Excécution du GC : Lancement de la phase de scan en fonction de la quantité mémoire utilisée
  - En parallèle du programme
  - Par pauses successives pour effectuer le traitement

Implémentations

- Existance de plusieurs bibliothèques pour le C++

Bohem GC

- GC de type conservative
- Remplace malloc par GC_malloc
- Propose des fonctionnalités de vérification de fuites mémoires

libgc

- Mise en place simple : simple recompilation avec la bibliothèque
- Remplace les opérateurs malloc et free

$\Rightarrow$ Permet d'écrire tout un programme en n'utilisant que new et new[] sans aucun delete ou delete[] !

Avantages / inconvénients

Avantages

- Facilite la programmation
- Évite les fuites mémoire
- Peut rendre le code plus performant
- Simplifie le debuggage

Inconvénients

- Non déterminisme
- Performances (phase de destruction couteuse)
- Charge en dents de scie

Plan

Principe

Objectifs

Facilite la gestion de la mémoire en évitant principalement :

- Évite les fuites mémoires
- Évite de libérer prématurément des ressources
- Évite de libérer plusieurs fois une ressource (provoque des erreurs)

Solution facile

Utilisation des pointeurs intelligents (‘smart pointeur’)

- Destruction automatique de la ressource pointée selon certains critères
- Gestion générale plus sûre

Principe

- Objet encapsulant le pointeur brute ('wrapper')
  - Permet d'éviter la manipulation directe du pointeur
  - Permet d'effectuer des contrôles supplémentaires (vérification pointeur non nul ...)
- Définir un politique d'accès
  - Contrôles supplémentaires
  - Casts possible / interdit
- Désallocation du pointeur brute
  - À la destruction du wrapper (et sous condition)

Conséquences

- Durée de vie du pointeur intelligent liée au bloc !
- Différent de la durée de vie du pointeur brute.

Les ingrédients

But

Manipulation comme un pointeur traditionnel : transparence

- Construction
  - Par défaut : pointeur NULL
  - Autre : pointeur brute
- Constructeur de copie : dépendant de la politique choisie
  - Transfert de propriété
  - Ajout de référence
  - ...
- Opérateur d'affectation : identique au construteur de copie

Les ingrédients

- Opérateur * : opérateur de déférencement
  - Donne accès au pointeur brut
  - Peut permettre de faire des vérification préliminaire (pointeur non NULL, ...)
  - Peut permettre de faire des opérations autre (lazy evaluation, ...)
- Opérateur → : identique à l'opérateur de déférencement
- Opérateur T* : permet de donner accès au pointeur brute
  - Permet d'automatiser certaine opération de cast
  - Peut être également implémenté par une méthode Get

Attention à l'opérateur de cast T* !

- Donne accès au pointeur brute : remet en question l'encapsulation
- Peut corrompre l'utilisation initiale

Les types

- Existance de differents types de pointeurs intelligents
  - Les scoped pointers
  - Les shared pointers
  - Les intrusive pointers
  - ...
- Différenciation sur plusieurs niveaux
  - Autorisation de copie
  - Partage de ressources
  - Transfert de propriétés
  - Méthode de comptage de références
  - ...

Les types

- Les scoped pointers

Propriétés

- Construction avec le pointeur brut
- Pas de copie possible (pas de construteur de copie, ni d'opérateur d'affectation)
- Désallocation à la destruction

Les types

- **Exemple** : utilisation

Code

```
class A {
        public : void Foo();
};

typedef ScopedPointer<A> APtr;

int main(){
        APtr a();                               // Erreur
        APtr b(NULL);                   // Construction avec
        b->Foo();                               // Comportement ? (assert, exception, ...)
        {
                APtr c(new A());
                c->Foo();
        }
        return 0;
}
```

Les types

Attention

Forte ressemblance entre les ScopedPointeur et les std : :auto_ptr mais comportements différents !

- std : :auto_ptr
  - Gestion d'un seul pointeur
  - Destruction du pointeur à la fin du bloc
  - Mais possède un constructeur de copie : implémente un transfert de propriété !
- Source de confusion : copie d'un pointeur
  - Modification du pointeur source à NULL
  - Interpretation du pointeur source comme valide du point de vue du programmeur

Les types

- **Exemple** : confusion possible avec les std : :auto_ptr pointeurs

Code

```
class A {
        public : void Foo();
};

typedef std::auto_ptr<A> APtr;

int main(){
        APtr a();
        a->Foo();                       // Ok
        {
                APtr c = a;
                c->Foo();               // Ok
                a->Foo();               // Error
        }
        a->Foo();                       // Error
        return 0;
}
```

Les types

- Les shared pointers

Propriétés

- Encapsulation d'un pointeur brute et d'un compteur de référence
- Possibilité de partager une ressource pointée entre différents pointeurs
- Chaque copie ajoute une référence ($++$ *compteur*)
- Chaque destruction supprime une référence ($--$ *compteur*)
- Désallocation lorsque le nombre de références est 0

Attention

Ne jamais créer deux shared pointer directement à partir du même pointeur brut

Les types

- **Exemple** : utilisation des shared pointers

Code

```
class A {
        public : void Foo();
};

typedef SharedPointer<A> APtr;

int main(){
        APtr a();
        APtr c(new A());
        a = c;
        a->Foo();
        c->Foo();
        {
                APtr b = a;
                a->Foo();
                b->Foo();
        }
        a->Foo();
        return 0;
}
```

Les types

- Les instrusive pointers

Propriétés

- Encapsulation d'un pointeur brut
- Externalisation du compteur de références
- Possibilité de partager une ressource pointée entre différents pointeurs
- Chaque copie ajoute une référence ($++$ *compteur*)
- Chaque destruction supprime une référence ($--$ *compteur*)
- Désallocation sous la responsabilité du gestionnaire de références

Remarque

Forte similarité avec les shared pointers mais

- Utilise un compteur externe pour gérer les références.
- Utilise de callbacks pour contrôler la valeur du compteur
- Désallocation sous la responsabilité du gestionnaire de références
- Ex : Permet de gérer les objets COM, ...

Les types

- **Exemple** : Utilisation des intrusive pointers

Code

```
class A {
        public :
                A():compteur(0) {}
                void Foo();
                void AddReference() { ++compteur; }
                void Release() { if(--compteur == 0) delete *this; }
        private :
                int compteur;
};

void add_reference(A* _obj)   { _obj->AddReference(); }
void remove_reference(A* _obj) { _obj->RemoveReference(); }
typedef IntrusivePointer<A> APtr;

int main(){
        APtr a();
        APtr c(new A());
        a = c;
        a->Foo();
        c->Foo();
        {
                APtr b = a;
                a->Foo();
                b->Foo();
        }
        a->Foo();
        return 0;
}
```

Bilan

En résumé

Les pointeurs intelligents présentent une certaine facilté d'utilisation

- Utilisation transparente grâce aux opérateurs * et →
- Application transparente du polymorphisme (voir après)
- Conversion automatique possible avec operator T*()

Bilan

En résumé

.. ou apporte une source de confusion

- Conversion operator T*() donne accès direct au pointeur brute
- Conversion implicit donc non maîtrisée
- Danger car suppression possible du pointeur

Exemple

```
typedef SharedPointer<A> APtr;

void OtherFoo(A* _a) {
        delete _a;
}

int main(){
        APtr a(new A());
        a->Foo();
        OtherFoo(a);
        a->Foo();       // Erreur d'appel !
        return 0;       // Erreur double delete !
}
```

Bilan

- Conduit à un comportement erroné/indéterminé car :
  - Accès à une ressource inexistante
  - Double libérations

Autre exemple

```
typedef SharedPointer<A> APtr;

void OtherFoo(A* _a) {
        APtr b(_a);
        b->Foo();
}

int main(){
        APtr a(new A());
        a->Foo();
        OtherFoo(a);
        a->Foo();       // Erreur d'appel !
        return 0;       // Erreur double delete !
}
```

Attention

- Donc éviter tant que possible d'avoir un opérateur de conversion T*()
- Toutefois si nécessaire :
  - Transformer en conversion 'explicit'
  - Mettre un accès avec get()

Polymorphisme et cast

- Et le polymorphisme dans tout cela ... ?

Appel de fonctions

Fonctionne au travers des opérateurs * et $\rightarrow$ avec les fonctions virtuals

Up-cast

- **Problème** : $xxxPointer < A >! = xxxPointer < B >$
- **Solutions** immédiates au problème :
  1. Mettre un pointeur de type classe dérivée : Comment passer en paramètre de tel pointeur ?
  2. Définir tous les pointeurs comme des pointeurs de classe mère : Comment avoir accès aux méthods dérivées ?

- Trouver une solution pérenne et générale !

Polymorphisme et cast

- **Solution** : Définition d'opérateur de cast automatique :

Code

```
template<typename A>
class xxxPointer
{
public:

        ...

        template<Ancestor>
        operator xxxPoint<Ancestor>() {
                return xxxPoint<Ancestor>(pointer, ...);
        }

        ...
private:
        xxxPointer(A* _ptr, ...);
};
```

Conslusion

Permet de gérer le polymorphisme de manière transparente, sans compromis sur l'intégrité du pointeur.

Polymorphisme et cast

- **Exemple** : Application des opérateurs de cast

Code

```
class A {
        public :
                void FooA();
                virtual void Foo();
};
class B : public A {
        public:
                void FooB();
                virtual void Foo();
};

typedef xxxPointer<A> APtr;
typedef xxxPointer<B> BPtr;

void FooExtern(const APtr& _a) { }

int main(){
        APtr a(new A()); APtr b(new B());

        a->FooA();    a->Foo();
        FooExtern(a);

        b->FooA();    b->FooB();    b->Foo();
        FooExtern(b);

        APtr c = b;
        c->FooA();
        c->FooB();    // Erreur mais normal
        c->Foo();
        FooExtern(c);
        return 0;
```

Les tableaux

Et les tableaux ...

- Avec les pointeurs normaux : possibilité d'avoir des pointeurs tableaux
- Possibilités avec les pointeurs intelligents ?

- Les équivalents :
  - Equivalent des ScopedPointer : ScopedArray
  - Equivalent des SharedPointer : SharedArray
- Les pincipales différences :
  - Le pointeur brut fourni doit avoir été alloué par new []
  - Implémentation de l'opérateur []
  - Utilisation de delete[] pour la désallocation

Les tableaux

- **Exemple** : utilisation des shared pointeurs array

Exemple

```
typedef SharedArray<A> AAPtr;

int main() {
        AAPtr array(new A[100]);
        A& a = array[1];
        a.Foo();
        {
                AAPtr b = a;
                b[99].Foo();
        }
        return 0;
}
```

Comparaison

Problème

Les pointeurs intelligents ne sont pas des pointeurs donc :

- Comparaison avec NULL non possible
- Conversion en booléen non possible

- **Exemple** : impossibilité d'utilisation comme des pointeurs normaux

Code

```
typedef xxxPointer<A> APtr;

int main()
{
        APtr a(new A());
        if (a == NULL)
                // ...
        if (a)
                // ...
        if (!a)
                // ...
        return 0;
}
```

Possibilité : implémentation de l'opérateur void* pour faire la comparaison avec 0

Comparaison

But

- Éviter l'implémentation de opérateur T* pour les raisons évoquées précédemmment
- Éviter l'implémentation de opérateur void*

- **Solution** : Implémentation de l'opérateur bool

Code

```
typedef xxxPointer<A> APtr;

int main()
{
        APtr a(new A());
        if (a == NULL)
                // ...
        if (a)
                // ...
        if (!a)
                // ...
        return 0;
}
```

Conclusion

Avantages

- Facilité de mise en place
- Permet d'abstraire en partie la gestion de la mémoire
- Évite de faire des bétises

Inconvénients

- Non déterminisme sur la désallocation des ressources
- Rend un peu plus lourd la manipulation des pointeurs
- Erreur possible lors des casts ou manipulations directes du pointeur

Plan

Outils

Problème

Lors de la mise au point d'un programme : nécessaire de s'occuper de son fonctionnement à l'exécution

- Chemins d'éxécution inattendus
- Comportements aléatoires
- Bugs en tout genre
- ...

- Concernant la gestion de la mémoire, l'intérêt est porté sur :
  - Occupation mémoire
  - Fréquence des allocations / désallocations
  - Tailles des allocations
  - Les classes les plus allouées ...
  - Et principalement : Les fuites mémoires !

Outils

Problème

Comment détecter les fuites mémoire ?

- Pour faciliter cette tâche : Utilisation d'outils spécialisés
  - Valgrind(Linux)
  - Purify(Win/Linux)
  - Visual Studio
  - ...
- Permet de détecter principalement les mauvaises allocation / désallocation, parfois les accès erronés

Comment fonctionne ces outils ?

Utilise le principe du memory tracer,

- Localise toutes les informations concernant les allocations et les désallocations
- Fournit un bilan à la fin de l'exécution

Outils

Context

Comprendre le principe memory tracer

- Pour tracer les allocations / désallocations
  - Redefinition des opérateurs new / delete globaux
  - Redefinition des opérateurs new[] / delete[] globaux
- Besoin d'informations sur la localisation des allocations et désallocations
  - Ajout d'informations de type fichier, ligne (__LINE__, __FILE__, ...)
- Dans l'opérateur new : enregistrement d'informations
  - Pointeur
  - Taille allouée
  - Fichier, ligne
  - Type d'allocation : pointeur simple, tableau
- Dans l'opérateur delete : vérification de la désallocation
  - Existance du pointeur
  - Type de désallocation

Outils

- **Exemple** : Tracer

Code

```
class MemoryTracer {
public:
        static MemoryTracer& Instance();
        ~MemoryTracer() {
                for(MemMap::iterator it = infos.begin();it != infos.end(); ++it) {
                        LogError(it->second);
                        delete it;
                }
        }

        void Allocate(void* _ptr, const std::string& _file, int _line, bool _array)     {
                MemBblock mBLock;
                // Remplir les informations ...
                infos[_ptr] = mBLock;
        }

        void Desallocate(void* _ptr, bool _array)       {
                current.Ptr   = _ptr;
                current.Array = _array;
                MemMap::iterator it = infos.find(_ptr);
                if(it!=infos.end())
                        LogError(current);
                else {
                        if(current.Array != it->second.Array)
                                LogError(current);
                        else
                                LogValid(current);
                        infos.erase(it);
                }
        }
};
```

Outils

- **Exemple** : Tracer (suite)

Code

```
class MemoryTracer {
public :
        void PreDesallocate(const std::string& _file, int _line) {
                current.Line     = _line;
                current.Filename = _file;
        }

private :
        // Private methods
        MemoryTracer() { }
        MemBlock{
                unsigned int Ptr;
                int Line;
                std::string Filename;
                bool ArrayAllocation;
        }
        void LogValid(MemBlock& _block) { ... }
        void LogError(MemBlock& _block) { ... }
        // Attributes
        typedef std::map<MemBlock> MemMap;
        MemMap infos;
        MemBlock current;
};
```

Outils

- **Exemple** : Redéfinition des opérateurs

Code

```
void* operator new(std::size_t _size) {
        void * ptr = malloc(_size);
        MemoryTracer::Instance().Allocate(ptr, _size, false);
        return ptr;
}

void* operator new[](std::size_t _size) {
        void * ptr = malloc(_size);
        MemoryTracer::Instance().Allocate(ptr, _size, true);
        return ptr;
}

void operator delete(void* _ptr) {
        MemoryTracer::Instance().Desallocate(_ptr, false);
        free(_ptr);
}

void operator delete[](void* _ptr) {
        MemoryTracer::Instance().Desallocate(_ptr, true);
        free(_ptr);
}
```

Outils

- **Exemple** : Utilisation

Code

```
int main() {
        int *a = new(__LINE__,__FILE__) int();
        MemoryTracer::Instance().PreDesallocate(__LINE__,__FILE__), delete a;

        int* b = new(__LINE__,__FILE__) int[10];
        MemoryTracer::Instance().PreDesallocate(__LINE__,__FILE__), delete[] b;
}
```

- **Exemple** : Utilisation avec une macro

Code

```
#define new      new(__LINE__, __FILE__)
#define delete   MemoryTracer::Instance(). PreDesallocate(__LINE__, __FILE__ ),  delete

int main() {
        int *a = new int();
        delete a;

        int* b = new int[10];
        delete [] b;
}
```

Restriction d'allocation

- Pour un type d'objet donné, l'utilisateur est libre de demander son allocation
  - Sur le tas
  - Sur la pile
- Besoin d'ajouter parfois des contraintes :
  - Interdiction d'allouer un objet sur la pile car trop volumineux
  - Interdiction d'allouer sur le tas pour ne pas ralentir le programme

Problème

Comment interdire un type d'allocation pour un objet donné ?

Restriction d'allocation

Contexte

Interdiction d'objet sur la pile

- **Solution** : Mettre tous les constructeurs en privée / protected :
  - Tous les constructeurs doivent être en privé
  - Fournir des méthodes static pour allouer l'objet

Problème

Relativement contraignant et fastidieux à mettre en place !

- **Alternative** : mettre le destructeur en privée / protected :
  - Plus rapide / concis
  - Ajout d'une seule méthode non static pour la destruction
  - Facilement 'templatisable'

Diagnostic

Détection à la compilation : permet de repérer rapidement les erreurs

Restriction d'allocation

- **Exemple** : interdiction sur la pile

Exemple

```
class A {
        public :
                void Foo() {...}
                void Destroy() { delete *this; }
        private :
                ~A() { }
};

int main() {
        A a; // Erreur
        A b = new A();
        delete b; // Erreur : destructeur privee
        b->Destroy();
        return 0;
}
```

Restriction d'allocation

Contexte

Interdiction d'objet sur le tas

- Mise en place simple, au niveau de la classe
  - Mettre les opérateurs d'allocation en private / protected
  - Mettre les opérateurs désallocation en private / protected

Diagnostic

Détection à la compilation : permet de repérer rapidement les erreurs

Restriction d'allocation

- **Exemple** : interdiction sur la pile

Exemple

```
class A {
        public :
                void Foo() {...}
        private :
                void* operator new(std::size_t _size);
                void* operator new[](std::size_t _size);
                void operator delete(void* _ptr);
                void operator delete[](void* _ptr);
};

int main() {
        A a;
        A b = new A(); // Erreur
        A b = new A[10]; // Erreur
        return 0;
}
```

Idiom NULL Pointer

Problème

En C++, le pointeur NULL pose des problèmes d'ambiguité conduisant à des comportement inattendu

- **Exemple**

Code

```
int Foo(int*); // Foo1
int Foo(int ); // Foo2

int main{
        Foo(2); // Foo2
        int a;
        Foo(&a); // Foo1
        Foo(0); // Foo2
        Foo(NULL); // Foo2
        return 0;
}
```

- Lors d'une ambiguité entre un entier et un pointeur sélection automatique de la surcharge avec entier

Idiom NULL Pointer

Contexte

Trouver une solution pour lever cette ambiguité.

- Pour cela possibilité de définir une classe spéciale qui dispose des bons opérateurs de conversion
  - Conversion pour les pointeur de valeur
  - Conversion pour les pointeur de fonction membre
  - Conversion pour les pointeur de fonction non membre

Information

Problème réglé dans la prochaine norme C++0x, en définissant une nouvelle constante intitulée ptrnull

Idiom NULL Pointer

- **Exemple** : implémentation de nullptr

Code

```
class nullptr_t {
  public:
    template<class T> operator T*() const { return 0; }

    template<class C, class T> operator T C::*() const    { return 0; }

  private:
    void operator&() const;

};
```

Idiom NULL Pointer

- **Exemple** : utilisation

Code

```
nullptr_t nullptr;

struct A {
  void Foo();
};

void Foo(double *) { }
void Foo(int) { }

int main(void)
{
        char * ch = nullptr;
        Foo(nullptr);
        Foo(0);

        void (*pmf2)(int) = nullptr;
        void (A::*pmf)() = nullptr;
        if (nullptr == ch) {}
        if (nullptr == pmf) {}
        if (nullptr == pmf2) {}

        return 0;
}
```

Idiom - RAII

- Utilisation fréquente de ressources nécessitants :
  - Une phase d'acquisition
  - Une phase de libération
- Exemple de ressources avec un tel mécanisme
  - Mutex
  - File
  - ...
- Toutes ces resources agissent sur un scoped limité !

Problèmes

- Si non respect de la libération après une acquisition, comportement indésirable / non déterminé
- Oublis aisés
- Arrive dans des cas non prévu (levé d'exception, ...)

Idiom - RAII

- **Exemple** : problèmes

Code

```
void Foo() {
        char * ch = new char [100];
        if (...)
                // ...
        else
                throw "ERROR";
        delete [] ch;
}

void Foo(Mutex& _m) {
        _m.Lock();
        if (...)
                // ...
        else
                throw "ERROR";
        _m.Unlock();
}
```

Idiom - RAII

Solution

Idiom RAII : Resource acquisition is initialization

- L'acquisition se fait à la construction de l'objet
- La libération se fait à la destruction de l'objet

- Grâce à ce principe évite tous les problèmes identifiés plus haut
- **Remarque** : Principe sur lequel repose les smart-pointers !

Idiom - RAII

- **Exemple** : utilisation

Code

```
class Lock {
        public :
                Lock(Mutex& _m):mutex(_m) { mutex.Lock(); }
                ~Lock() { mutex.Unlock(); }
        private :
                Mutex& mutex;
}

void Foo(Mutex& _m) {
        Lock lock(m);
        if (...)
                // ...
        else
                throw "ERROR";
}
```