C++ : De l'Antiquité à nos jours

Joel Falcou

Laboratoire de Recherche en Informatique

30 mai 2012

Un peu d'histoire

- Le C++ est initialement utilisé comme un "C avec des classes"
  - allocation dynamique et pointeurs nus
  - polymorphisme dynamique parfois à outrance
  - gestion artisanale des erreurs
- Le C++ moderne lui, mets l'accent sur :
  - une utilisation accrue de la pile au détriment du tas ;
  - l'inférence de type automatique ;
  - des pointeurs sémantiquement riche ;
  - le polymorphisme statique via les *templates* ;
  - une large utilisation des conteneurs et algorithmes de la STL ;
  - les exceptions comme systèmes de détection et remonté des erreurs ;
  - des fonctions anonymes locales.

le C++ avant et après

Le C++ *circa* 1999

```
circle*         p = new circle( 42 );
vector<shape*> v = load_shapes();

for(vector<circle*>::iterator i = v.begin(); i != v.end(); ++i)
  if( *i && **i == *p ) cout << **i << " is a match\n";

for( vector<circle*>::iterator i = v.begin(); i != v.end(); ++i)
  delete *i;

delete p;
```

Le C++ *circa* 2010

```
auto p = make_shared<circle>( 42 );
vector<shared_ptr<shape>> v = load_shapes();

for_each( begin(v), end(v)
        , [&]( const shared_ptr<shape>& s )
          {
            if( s && *s == *p ) cout << *s << " is a match\n";
          }
        );
```

Développé surement en C++

- Objectifs :
  - Ecrire un code correct le "jour 0"
  - Gestion de la mémoire
  - Gestion des erreurs à l'exécution
- Moyens mis en œuvre :
  - La "Rule of Three"
  - Le principe RAII
  - Pointeurs sémantiquement riches
  - Exception et Garantie d'exceptions

La "Rule of Three"

- Etend la forme canonique de Coplien
- Définie la structure correct d'une classe
  - un constructeur par copie
  - un opérateur d'affectation
  - un destructeur
- Si une de ces méthodes est définie, les autres sont obligatoires

Le principe RAII

- En C++, chaque objet a une durée de vie bien définie
  - Allouer sur le tas, il disparait lors d'une libération mémoire
  - Allouer sur la pile, il disparait à la sortie de la portée courante
- Allocation des ressource via une construction d'objet
- Toute ressource est automatiquement libérée en sortie de portée
- *Ressource Allocation In Initialisation*
- Exemple : mutex & vérrouillage.

Pointeurs sémantiquement riches

- Les pointeurs "nus" sont dangereux
  - Pas de notion de propriété
  - Pas de sémantique de partage
  - *To Delete or not to Delete ?*
- Solution : Enrichir la sémantique des pointeurs
  - Gestion de la propriété
  - Gestion de la durée de vie

Pointeurs sémantiquement riches - `smart_ptr`

- `smart_ptr<T>` : pointeur sur T à durée de vie "managée"
- Implemente un `reference-counter`
- Libération de la ressource lorsque celle ci est définitivement abandonnée
- Comportement proche des références JAVA

```
Song* UseRawPointer()
{
  Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");
  pSong->play();
  return pSong;
}
```

```
shared_ptr<Song> UseSmartPointer()
{
  shared_ptr<Song> pSong(new Song(L"Nothing on You", L"Bruno Mars"));
  pSong->play();
}
```

Pointeurs sémantiquement riches - `unique_ptr`

- `unique_ptr<T>` : pointeur exclusif sur T
- Un `unique_ptr<T>` a un unique propriétaire
- Toute tentative de copie est une erreur
- Garantie l'unicité de la ressource
- Plus léger que `shared_ptr<T>` et à privilégier
- Exemple : L'idiome Handle/Body (ou PIMPL)

Exception et Garantie d'exceptions

- Le concept d'exception en C++
  - Permet de signaler un comportement aberrant en post-conditions
  - Effectue une sortie prématurées de la portées courantes
  - Le bloc `try ... catch` surveille ces remontées et les traite
- Les garanties d'exceptions
  - Introduite par Abrahams et al. (`http://www.boost.org/community/exception_safety.html`)
  - Objectif : Minimiser les blocs try/catch explicites
  - Pour chaque fonction, spécifier un type de gestion des exception
  - Par composition, les propriétés dde gestion se conservent

Exception et Garantie d'exceptions

- La garantie de base (Basic Garantee)
  - Les invariants sont préservés
  - Aucune ressource ne fuit
- La garantie forte (Strong Garantee)
  - La fonction est un succés ou lance une exception
  - L'état du programme est intact
  - Comportement type COMMIT/ROLLBACK
- La garantie *No Throw*
  - Ne lancent jamais d'exception. Ex : `swap`
- Exemple : Buffer memoire

Ecrire du C++ performant

- Objectifs :
  - Maitriser le temps d'exécution d'un programme
  - Optimiser l'utilisation de la mémoire
  - Quid des architectures multi-core ?
- Moyens mis en œuvre :
  - La sémantique de mouvement
  - La "Rule of Five"
  - Support pour le multithreading

La sémantique de mouvement

- Problème numéro 1

```
std::vector<double> foo()
{
  return std::vector<double>(4096*4096);
}

std::vector<double> x = foo();
```

  - Comment eviter le coût de la copie ?

- Problème numéro 2 :

```
void foo( double const& x);
void bar( double& x);

void select( bool which, ??? x )
{
  which ? foo(x) : bar(x);
}
```

  - Quel type pour `x` ?

- Réponse : les `rvalue` references

Augmenter l'ergonomie de votre C++

- Objectifs :
  - Limiter le code "d'aisance"
  - Augmenter le niveau d'abstraction
  - Reutiliser plutôt que recoder
- Moyens mis en œuvre :
  - Inférence de type automatique
  - Algorithmes standards
  - Object Appelable et Fonctions Lambda

Inférence de type automatique

- Problème :

```
template<class T> typename T::const_iterator
next_to_begin( T const& x )
{
  typename T::const_iterator c = x.begin();
  return c++;
}

template<class T1, class T2>
???? average( T1 const& t1, T2 const& t2 )
{
  return (t1+t2)/2.;
}
```

- "Fuite d'abstraction" sur le typage des variables
- Calcul de type génériques complexes et peu extensibles

Inférence de type automatique

- auto

```
template<class T> auto next_to_begin( T& x ) -> decltype(++x.begin())
{
  auto c = x.begin();
  return ++c;
}
```

- decltype

```
template<class T1, class T2>
decltype( (T1()+T2())/2.f ) average( T1 const& t1, T2 const& t2 )
{
  return (t1+t2)/2.f;
}
```

Inférence de type automatique

- `auto` et `decltype`

```
template<class T> auto next_to_begin( T const& x ) -> decltype(x.begin()++)
{
  auto c = x.begin();
  return c++;
}

template<class T1, class T2>
auto average( T1 const& t1, T2 const& t2 ) -> decltype((t1+t2)/2.f)
{
  return (t1+t2)/2.f;
}
```

- `auto` indique un calcul différé du type de retour
- `->` indique le calcul effectif du type de retour
- `decltype` peut utiliser les arguments nommés après leur déclaration

Algorithmes standards

- Problèmes :
  - Les boucles issues du C manque de sémantique
  - Notion de séparation entre algorithmes et données
  - Répondre aux évolutions du matériel sans modifier le code
- Solution : La STL (Stepanov 1999)
  - Trois piliers : Algorithme/Iterateur/Conteneur
  - Programmation Générique
  - Extensibilité via des Objets Appelables (CallableObject)

Object Appelable & Fonctions Lambda

- Objets Appelables
  - Extension de la notion de fonction
  - Possibilité de stocker un état (*stateful function*)
  - Stockable comme un objet classique
  - `std::function`
- Fonctions Lambda
  - Fonctions anonymes locales
  - Remplace une définition externe déportée
  - Stockable via `std::function`

Ouvrages et Webographie

- Programming - Principles and Practices using C++, B. Stroustrup, 2009
- Modern C++ Design, A. Alexandrescu, 2003
- Le blog de Herb Sutter : `http://herbsutter.com/`
- Microsoft "Welcome back to C++" : `http://msdn.microsoft.com/en-us/library/hh279654(v=vs.110)`
- BOOST : `http://www.boost.org`
- C++Next : `http://cpp-next.com/`