Programmation Générique
Session 2 - De la surcharge aux Concepts

Joel Falcou

LRI

9 septembre 2013

Rappel sur la surcharge

Principe

- En C : une fonction = un symbole
- En C++ : une fonction = une signature
- Une signature = symbole + liste de paramètres

Exemples :

- `double f()`
- `double f(int)` - OK
- `double f(double,int)` - OK
- `double f(...)` - OK
- `int f()` - KO

Règles de résolution

Processus Général [1]

- Les variantes du symbole sont recherchées pour créer l'*Overload Set* (Ω).
- Toutes variantes n'ayant pas le nombre de paramètres adéquat est éliminés pour obtenir le *Viable Set*.
- On recherche dans cet ensemble la *Best Viable Function*.
- On vérifie l'accéssibilité et l'unicité de la sélection.

Que faire de tout ça ?

- Comment définir (Ω) ?
- Quels critères pour la *Best Viable Function* ?

[1] *C++ Templates : The Complete Guide* – David Vandevoorde, Nicolai M. Josuttis

All Glory to the Overload Set

Construction de Ω

- Ajouter toutes les fonctions non-template avec le bon nom
- Ajouter les variantes template une fois la subsitution des paramètres templates est effectuée avec succés

Notes

- Ω est un treillis : les fonctions non-template dominent les fonctions template
- Définition de "substituer avec succés"
- L'ADL met bien sur son petit nez la dedans

Finding `nemo()`

Sélection de la *Best Viable Function*

- Chaque argument est associé à un Séquence de Conversion Implicite (ICS)
- Chaque argument est trié par rapport à son ICS
- Si un argument n'est pas triable, le compilateur boude.

Finding `nemo()`

Les Séquence de Conversion Implicite

- Conversions standards (SCS)
  - Correspondance exacte
  - Promotion
  - Conversion
- Conversion utilisateur (UCDS) , composée comme :
  - une séquence stadnard
  - une conversion définies explicitement
  - une deuxiéme séquence stadnard
  - Un UDCS est meilleur qu'un autre si sa seconde SCS est meilleur que l'autre
- Séquence de conversion par ellipse

assert(Mind==Blown)

Un example

```
void f(int)           { cout << "void f(int)\n"; }
void f(char const*)   { cout << "void f(char const*)\n"; }
void f(double)        { cout << "void f(double)\n"; }

int main()
{
  f(1); f(1.); f("1");  f(1.f);  f('1');
}
```

Output

- `f(1)` → `void f(int)`
- `f(1.)` → `void f(double)`
- `f("1")` → `void f(char const*)`
- `f(1.f)` → `void f(double)`
- `f('1')` → `void f(int)`

assert(Mind==Blown)

Un example

```
void f(int)          { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double)       { cout << "void f(double)\n"; }
template<class T> void f(T) { cout << "void f(T)\n"; }

int main()
{
  f(1); f(1.); f("1");  f(1.f);  f('1');
}
```

Output

- f(1) → void f(int)
- f(1.) → void f(double)
- f("1") → void f(char const*)
- f(1.f) → void f(T)
- f('1') → void f(T)

Cool story Bro ! Et maintenant ...

```
template<typename Container>
typename Container::size_type f(Container const&)
{
  return c.size();
}

int main()
{
  std::vector<double> v(4);
  f(v);

  f(1); /// OMG Incoming Flaming Errors of Doom
}
```

Cool story Bro ! Et maintenant ...

```
template<typename Container>
typename Container::size_type f(Container const&)
{
  return c.size();
}

int main()
{
  std::vector<double> v(4);
  f(v);

  f(1); /// OMG Incoming Flaming Errors of Doom
}
```

```
error: no matching function for call to 'f(int)'
```

Substitution Failure Is Not An Error

Que se passe-t-il ?

- On veut construire Ω pour une fonction donnée
- Certaines surcharges sotn des résultats de substitutions templates
- Si la substitution échoue, la surcharge est retirée de Ω
- Aucune erreur n'est levée par le compilateur
- On parle de **SFINAE**

La SFINAE en pratique : enable_if

```
template<bool Condition, typename Result = void>
struct enable_if;

template<typename Result>
struct enable_if<true,Result>
{
  typedef Result type;
};
```

La SFINAE en pratique : enable_if

```
template<typename T>
typename enable_if<(sizeof(T)>2)>::type
f( T const& )
{
  cout << "That's a big type you have there !\n";
}

template<typename T>
typename enable_if<(sizeof(T)<=2)>::type
f( T const& )
{
  cout << "Oooh what a cute type!\n";
}
```

La SFINAE en pratique : enable_if_type

```
template<typename Type, typename Result = void>
struct enable_if_type
{
  typedef Result type;
};

template<typename T, typename Enable = void> struct size_type
{
  typedef std::size_t type;
};


template<typename T> struct size_type<T,typename
    enable_if_type<typename T::size_type>::type
{
  typedef typename T::size_type type;
};
```

La SFINAE en pratique : Détection de propriétés

```
template<typename T>
struct is_class
{
  typedef char yes_t
  typedef struct { char a[2]; } no_t;

  template<typename C> static yes_t test(int C::*);
  template<typename C> static no_t test(...);

  static const bool value = sizeof(test<T>(0)) == 1;
};
```

Tag Dispatching

Limitation de la SFINAE

- Les conditions d'exclusion doivent être disjointes
- Difficile à étendre hors des fonctions
- La compilation est en $O(N)$ avec le nombre de cas

Le Tag Dispatching

- Catégorisation des types selon des familles de propriétés
- Facile à étendre : une famille = un type
- La sélection se fait via la surcharge normale

Tag Dispatching - std::advance

```
namespace std
{
  struct input_iterator_tag {};
  struct bidirectional_iterator_tag : input_iterator_tag {};
  struct random_access_iterator_tag
       : bidirectional_iterator_tag {};
}
```

Tag Dispatching - std::advance

```
namespace std
{
  namespace detail
  {
    template <class InputIterator, class Distance>
    void advance_dispatch( InputIterator& i
                         , Distance n
                         , input_iterator_tag const&
                         )
    {
      assert(n >=0);
      while (n--) ++i;
    }
  }
}
```

Tag Dispatching - std::advance

```
namespace std
{
  namespace detail
  {
    template <class BidirectionalIterator, class Distance>
    void advance_dispatch( BidirectionalIterator& i
                         , Distance n
                         , bidirectional_iterator_tag const&
                         )
    {
      if (n >= 0)
        while (n--) ++i;
      else
        while (n++) --i;
    }
  }
}
```

Tag Dispatching - std::advance

```
namespace std
{
  namespace detail
  {
    template <class RandomAccessIterator, class Distance>
    void advance_dispatch( RandomAccessIterator& i
                         , Distance n
                         , random_access_iterator_tag const&
                         )
    {
      i += n;
    }
  }
}
```

Tag Dispatching - std::advance

```
namespace std
{
  template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n)
  {
    typename iterator_traits<InputIterator>::iterator_category
         category;
    detail::advance_dispatch(i, n, category);
  }
}
```

Du TD au Concepts

Renversons le problème

- Le Tag Dispatching agrége des **implantations** dont la **synatxe** est correcte
- Association entre "Famille" de type et synatxe concrête
- On parle de
  textitDuck Typing

Que faire de mieux ?

- Définissons l'interface de nos objets comme un série d'éléments syntaxiques cohérents
- Remplaçons la vérification basé sur l'héritage par un contrat "mou" entre concepteur et utilisateur
- Notion de **Concept**

Concepts vs POO

POO

- Un comportement = une interface
- Raffinement des comportements = héritage public
- Résolution basée sur les types dynamiques

Concepts

- Un comportement = un Concept
- Raffinement des comportements = Ajout de contraintes
- Résolution basée sur la validité syntaxique

Programmez avec les Concepts

Les 4 pilliers

- Lifting
- Conceptualisation
- Modelisation
- Specialisation

Programmez avec les Concepts : Le Lifting

```
int sum(int* array, int n)
{
  int result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Programmez avec les Concepts : Le Lifting

```
float sum(float* array, int n)
{
  float result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Programmez avec les Concepts : Le Lifting

```
template<typename T>
T sum(T* array, int n)
{
  T result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Programmez avec les Concepts : Le Lifting

```
std::string concatenate(std::string* array, int n)
{
  std::string result = "";
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Programmez avec les Concepts : Le Lifting

```
// Requirements:
//    T must have a default constructor that produces the
     identity value
//    T must have an additive operator +
//    T must have an assignment operator
//    T must have a copy constructor
template<typename T>
T sum(T* array, int n)
{
  T result = T();
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```