Techniques

- Assertions de compilation
- Spécialisation partielle de template
- classes locales
- association de type et de valeurs
- Meta-programmation : select
- Détecter la convertibilité et le lien d'héritage a la compilation
- Utilisation de std::type_info
- le mécanisme de Trait

Assertions de compilation

```
template <class To, class from>
To safe_reinterpret_cast(From from)
{
assert(sizeof(From) <= sizeof(To));
return reinterpret_cast<To>(from);
}

int i=...;
char *p=safe_reinterpret_cast<char *>(i);
```

Quel dommage d'attendre l'exécution pour détecter une erreur.
Première version d'une détection d'erreur a la compile :
utilisons le fait que les tableau de taille zéro sont illégales.

```
#define STATIC_CHECK(expr) {char unnamed[(expr)?1:0];}
```

on peut donc ecrire

```
template <class To, class from>
To safe_reinterpret_cast(From from)
{
STATIC_CHECK(sizeof(From) <= sizeof(To));
return reinterpret_cast<To>(from);
}
```

Assertions de compilation 2

```
void *unptr=...;
char c=safe_reinterpret_cast<char>(unptr);
```

Le défaut est d'avoir un message d'erreur bizare (tableau de taille zéro invalide) alors que ce qu'il nous faut soit : le type char est trop petit pour stocker un pointeur.

Essayons :

```
template<bool> struct ErreurDeCompile;
template<> struct ErreurDeCompile<true> {};
#define STATIC_CHECK(expr) (ErreurDeCompile<(expr)!=0>())
```

le message devient Undefined specialization ErreurDeCompile<false>

C'est mieux mais c'est pas encore idéal, ajoutons un paramètre a notre static_check, le problème est qu'il faut passer un texte qui soit un identifier du C et donc pas d'espaces, mais bon ....

```
template<bool> struct Verification {Verification(...){} };
template<> struct Verification<false> {/* rien */};
#define STATIC_CHECK(expr,msg) \
    {\
    class ERREUR_##msg {};\
    (void)sizeof(Verification<(expr) != 0>((ERREUR_##msg())));\
    }
```

```
template <class To, class from>
To safe_reinterpret_cast(From from)
{
 STATIC_CHECK(sizeof(From)<=sizeof(To), Destination_trop_petite);
    return reinterpret_cast<To>(from);
}
```

nous donne l'erreur :

```
Error: cannot convert ERREUR_Destination_trop_petite to Verification
```

Ce qui est on ne peut plus clair !

Spécialisation partielle de template

```
template <class Window , class Controleur>
class Widget { /* implenetation generique */  };
```

Spécialisation Explicite

```
template <>
class Widget<ModalWindow ,MyControleur> {
/* implementation spécifique */
  };
```

Spécialisation partielle :

```
template <class Window>
class Widget<Window,MyControleur> {
/* implementation spécifique dans le cas de MyControleur  */
  };
```

on peut faire des truc tordus et puissant :)

```
template <class ButtonArg>
class Widget<Button<ButtonArg>,MyControleur> {
/* implementation spécifique dans le cas de MyControleur et d'un Button */
  };
```

Le compilateur fait une étude complète de toutes les spécialisation et vous trouve la plus proche.

Outil fondamental ! Dommage ne fonctionne que pour une classe entière e(pas possible de spécilisé une seule fonction membre par exemple), le rôle du template générique devient plus proche d'une interface munie d'un comportement par default. La spécialisation partielle ne fonctionne pas pour les fonctions.

Classes Locales

```
 void fifi()
{
class Local {
    membres donnée fonctions etc....
    };
    Local x;
    ...
}
```

Limites : pas de membres static, pas d'accès aux variables locales non statiques.
Elles sont très utile comme classes (locales) dans des fonctions template qui sont construites avec les types fournis en paramètre de la fonction template.

```
class Interface
{ public: virtual void Fun() = 0;
}
template <class T,class P>
Interface *MakeAdpater(const T&  obj,  const P& arg )
{
class Local : public Interface
{
public: Local(const T&  obj,  const P& arg ) : _obj(obj), _arg(arg) {}
    virtual void Fun() { obj.Call(_arg); } // substitution d'interface call
       en fun
    private:
    T _obj; P _arg;
}
return new Local(obj, arg);
}
```

Associer des entiers et des types

```
template <int v>
struct Int2Type {  enum { value = v }; } ;
```

Pour chaque valeur de `v` nous avons la création d'un nouveau type différent de tous les autres
`exemple`
`Int2Type<1552678>`
`C`ela vous permet par exemple de choisir une fonction par le type d'un de ses paramètre, Ce paramètre est construit avec le résultat d'un calcul effectué pendant la compilation.

Pourquoi tout ce tracas un bon vieux `if-else` résout rapidement le problème au moment de l'exécution.
Et bien non Il faut prendre une décision a la compilation car une seul branche du test est valide !

Container d'objets polymorphes ou non

```
template <class T> class MonSuperContaineur {};
```

Ce containeur contient des pointeurs, nous voulons fournir des copies des objets contenus, de deux choses l'une :

- soit se sont des objets standards qui ont un constructeur de copie
- soit des objets polymorphes (de type inconnu) et il faut donc appeler la fonction clone pour avoir une copie.

Première version :

```
template <class t, bool polymorphe>
class MonContainer
{
...
void dotheRightThing()
{
    T* unObjet = ... ; // l'objet a copier
    if(polymorphe){
        T* copie = unObjet->Clone();
    } else {
        T* copie = new T(*unObjet);
    }
};
```

Et bien NON ! Le compilateur ne peut pas : soit Clone() n'est pas définie ou bien le constructeur de copie n'est pas public (normal pour un objet polymorphe bien écrit).

Deuxième tentative

Nous allons profiter du fait que les fonctions template et les fonctions membre de class template qui ne sont pas appelées ne sont pas traduites (il y a simplement une vérification syntaxique pas de tentative d'édition de liens).

Nous allons donc construire deux fonctions une pour les objet polymorphes une pour les objets non polymorphes, en fonction du paramètre du template nous appellerons l'une ou l'autre.

```
template <class T, bool polymorphe>
class MonContainer
{
private:
void dotheRightThing(T* pobj,Int2Type<true>)
{
        T* copie = pobj->Clone();
}
void dotheRightThing(T* pobj,Int2Type<false>)
{
        T* copie = new T(*unObjet);
}
public:
void dotheRightThing(T* pobj)
    { dotheRightThing(pobj,Int2Type<polymorphe>());  }
};
```

Voilà !

Spécialisation de fonction template !

La spécialisation partielle de fonction template n'existe pas !
Pas de problème nous allons fournis le service en utilisant un template qui créer une bijection entre l'espace des types et un autre espace de type en utilisant le template Type2Type.
Exemple de problème la fonction suivante permet la création de n'importe quel type a partir de n'importe quel argument, bien sur la fonction cible (le constructeur en question) doit exister pour que le programme compile;

```
template <class T , class U>
T* Create(const U&arg)
{
    return new T(arg);
}
```

Votre problème c'est le paquetage graphique que vous utilisez (legacy code) n'est bien sur pas modifiable et le constructeur de Widget prend deux argument le deuxième étant une valeur fixe comme par exemple (-1) .
Le code suivant est Interdit :

```
// illegal
template <class U>
Widget* Create<Widget, U>(const U&arg) { return new Widget(arg, -1); }
//
```

Une utilisation classique de la surcharge consiste a passer un argument bidon pour typer la fonction

```
template <class T , class U>
T* Create(const U&arg, T /*bidon */ ) {return new T(arg);}

template <class U>
Widget* Create<Widget, U>(const U&arg,Widget /*bidon */ )
 { return new Widget(arg, -1); }
```

Ok ca fonctionne mais il y a construction d'un Widget inutile a chaque fois pour l'argument bidon !
Comment éviter le sur-coût ?

Type2Type

```
template <class T>
struct Type2Type {
typedef T OriginalType;
};
```

Type2Type n' a pas de valeur une taille nulle, mais par contre pour deux type différents on obtient deux Type2Type différents.

```
template <class T , class U>
T* Create(const U&arg, Type2Type<T> /*bidon */ ) {return new T(arg);}

template <class U>
Widget* Create<Widget, U>(const U&arg,Type2Type<Widget> /*bidon */ )
 { return new Widget(arg, -1); }
```

Utilisation de Create :

```
String* pStr = Create(''Hello World'', Type2Type<String>());
Widget* pW= Create( 100, Type2Type<Widget>());
```

Sélection de Type

De nouveau nous voilà confronter à des types polymorphes et d'autre qui ne le sont pas, nous voulons fournir un service identique avec des implémentation différentes et cela sur un choix fait a la compilation (en effet savoir si un type est polymorphe ou non est une information que l'on connaît assez tôt dans la conception !).

Ici nous avons une class qui stocke un certain nombre d'objet dans un std::vector, si les types sont polymorphe nous allons faire un stockage indirect par pointeurs, sinon nous allons stocker directement dans le std::vector car c'est plus efficace (mémoire+vitesse).

Première version (STL 1998) utilisation de traits.

```
template  <typename T, bool polymorphe>
struct MonContainerValuesTraits {
    typedef T* ValueType;
};
template  <typename T>
struct MonContainerValuesTraits<T,false> {
    typedef T ValueType;
};
template <typename T, bool polymorphe>
class MonContainer
{
 typedef MonContainerValuesTraits<T,polymorphe> Traits;
 typedef typename Traits::ValueType ValueType;
 ...// utilisation de ValueType dans le container
}
;
```

Lourd car il faut pour chaque sélection de type créer un nouveau template de Traits .

Sélection de Type (2)

Utilisons l'opérateur de **Méta-programmation** SELECT (il pourrait s'appeler IF)

```
template <bool flag, typename T, typename U>
struct Select {
typedef T Result;
};
template <typename T, typename U>
struct Select<false,T,U> {
typedef U Result;
};
```

Il est alors plus facile d'écrire MonContainer :

```
template <typename T, bool polymorphe>
class MonContainer
{

 typedef typename Select<polymorphe,T*,T> ValueType;

}
;
```

Convertible ? IS-A ?