Spécification et réalisation d'une classe générique

Romuald THION

2007

Résumé

Ce document est un exemple de document de spécification et de réalisation d'une classe générique simple en C++. À partir d'un cahier des charges sommaire, nous définissons les fonctionnalités proposées par la classe. Une particularité de cette classe est qu'elle est paramétrée selon deux types de données. La spécification doit donc faire abstraction des types et n'utiliser que des propriétés supposées connues pour ces types. La réalisation fait appel aux templates C++ et aux concepts de la Standard Template Library (STL). La classe *paire* que nous spécifions est disponible dans les utilitaires fournis par la STL.

Table des matières

1 Introduction

Vérifier la correction d'un programme, d'un algorithme, c'est s'assurer qu'il fait bien ce que l'on a prévu qu'il fasse. L'étape de spécification d'un programme est donc fondamentale en génie logiciel : un programme n'est correct que vis-à-vis de ses spécifications. C'est donc par là qu'il faut commencer pour réussir un projet informatique, c'est probablement pourquoi tout le monde s'en passe.

Ce document a deux objectifs :

- d'une part, illustrer ce qu'est une spécification sur un cas relativement simple. Cet exemple reste *un* exemple et n'est en aucun cas une forme canonique qu'il faut suivre les yeux fermés,
- d'autre part, introduire des notions de programmation générique (c'est à dire l'utilisation de templates) issues du standard C++ [1] et utilisées dans les spécifications de la STL.

Le cas d'étude choisi est une classe générique simple : la *paire*. C'est une classe auxiliaire définie dans le chapitre 20.2.2 du standard C++ ISO 14882 :2003 [1].

2 Cahier des charges

Le cahier des charges peut être exprimé en des termes confus, ambigus voire contradictoires. Nous considérons pour cet exemple que le client a donné le cahier des charges lapidaire suivant :

> *Nous souhaitons un outil C++ permettant de gérer des paire de données, dans le but de simplifier certains de nos développements. Nous voulons un type de données qui à partir de données existantes, crée un doublet d'éléments. Nous voulons pouvoir manipuler les tuples avec simplicité, comme des entiers par exemple, sans toutefois proposer toutes les opérations des entiers, on ne souhaite pas ajouter ou soustraire des doublets directement par exemple. De plus nous souhaitons que la classe soit générique, performante et qu'un minimum de conditions soient imposées quant aux données du tuple. Des outils facilitant la création de doublet arbitrairement choisis devraient être proposés.*

Cet exemple de cahier des charges est relativement artificiel.

Un échange avec le client s'avère ici inévitable, la profusion et la diversité des termes employés à plus ou moins bon escient nécessite de clarifier la pensée de la maîtrise d'ouvrage. La notion de *paire* semble prévaloir dans les besoins, le produit que nous allons spécifier va donc être « une paire », mais de quels types de données ? Avec quelles fonctionnalités offertes ? Quelles sont les conditions acceptables sur les types d'entrée ? Nous supposons par la suite que l'étape de dialogue/négociation préalable a été menée a bien, et que le client accepte les spécifications qui suivent.

C'est l'étape où des choix doivent être proposés au client.

3 Spécification

L'action de *spécifier* consiste à décrire avec précision les caractéristiques du produit que nous allons réaliser, dans notre cas un produit logiciel et même plus précisement une *classe générique*.

Nous allons tâcher de clarifier les notions évoqués, en prenant en compte les critères demandés par le client comme la possibilité de manipuler simplement la classe et de définir un minimum de conditions en contrat. Cette section va commencer par établir les définitions des *concepts* préalables à la spécification de la classe générique *paire*.

Ne pas aller à l'encontre du cahier des charges.

3.1 Définitions des concepts

Les *concepts* sont des propriétés abstraites définies sur des types de données. Ces propriétés sont utilisées pour définir les contrats sur les méthodes de la paire. Les concepts que nous présentons ici sont issus de la STL et utilisés courament dans la définition de classes de la bibliothèque standard (exemple : dans les conteneurs ou les algorithmes). Les définitions proposées permettent d'utiliser sans ambiguïtés des notions abstraites, évitant ainsi les répétitions et les risques de confusion.

Les spécifications pourraient être réalisées dans un autre langage de programmation que le C++, pourvu qu'il soit orienté objet à base de classes et supportant la programmation générique.

Soit T un *type de données* :

- T est dit *assignable* s'il est possible
  1. de *copier* des données de ce type, définition du constructeur de copie, `T(const T& t)` en C++,
  2. et d'*assigner* des donnése de ce type aux variables, définition de la surcharge de l'opérateur d'affectation, `T & operator=(const T& t)` en C++.
- T est dit *constructible par défaut* si le constructeur par défaut est défini pour T, `T(void)` en C++.
- T est dit *comparable* s'il est possible de comparer deux données de type T avec l'opérateur == et si == est une *relation d'équivalence* : transitive [1], symétrique [2] et réflexive [3].
- T est dit *ordonnable* s'il est possible de comparer deux données de type T avec l'opérateur < et si < est une *relation d'ordre strict* : transitive et irréflexive [4].

Nous « trichons » en utilisant la dialectique et les prototypes du C++.

Une type T est dit *convertible* en un type T′ s'il existe une application qui à toute donnée de type T fasse correspondre une donnée de type T′.

L'anti-symétrie forte , $a\mathcal{R}b$ implique $\neg(b\mathcal{R}a)$, d'une relation d'ordre strict est un théorème de la transitivité et de l'irréflexivité.

1. $a\mathcal{R}b \wedge b\mathcal{R}c$ implique $a\mathcal{R}c$
2. $a\mathcal{R}b$ implique $b\mathcal{R}a$.
3. $a\mathcal{R}a$ pour tout a.
4. $\neg(a\mathcal{R}a)$ a pour tout a.

3.2 Spécification de la paire

Le type paire *est générique.*

À partir des concepts précédemment définis, nous allons pouvoir définir la paire proprement dite. Nous désignerons par p la paire courante sur laquelle sont invoquées les méthodes.

Définition et construction

Ne connaissant pas les types de données à l'avance nous pouvons seulement supposer des propriétés que ces types respectent par contrat.

- Une *paire* est une *classe* paramétrée selon deux types T1 et T2, donnés à la déclaration de la paire.
  Contrat : les deux types T1 et T2 sont *assignables*.
- Une paire *contient* deux objets : un objet de type T1, nommé *premier membre*, et un autre de type T2, comme *second membre*.
- Les deux membres d'une paire sont accessibles publiquement.
- Une paire est constructible grâce :
  - *au constructeur par défaut*, qui invoque les constructeurs par défauts respectifs des types T1 et T2 pour le premier et le second membre de p.
    Contrat : ce constructeur requiert que les types T1 et T2 soient *constructibles par défaut*.
  - *au constructeur principal*, qui prend en paramètre un objet a constant de type T1 et un objet b constant de type T2. Ce constructeur copie a dans le premier membre de p et copie b dans le second membre de p,
  - *au constructeur de copie*, qui prend en paramètre une paire constante p' de types U1 et U2. Ce constructeur copie le premier membre de p' dans le premier membre de p et copie le second membre de p' dans le second membre de p. Les types U1 et U2 sont implicitement convertis en type T1 et T2.
    Contrat : le type U1 doit être convertible en type T1 et le type U2 doit être convertible en type T2.

On devrait également définir l'opérateur d'affectation pour les paires.

Relations entre paires

La relation d'infériorité ne paraît pas très naturelle mais a pour intérêt de ne pas faire intervenir la relation d'égalité dans sa définition.

- Une paire a est *égale* à une paire b (opérateur ==) si et seulement si le premier membre de a est égal au premier membre de b et que le second membre de a est égal au second membre de b.
  Contrat : l'égalité des paires requiert que les types T1 et T2 soient *comparables*.
- Une paire a est *inférieure* à une paire b (opérateur <) si et seulement si le premier membre de a est inférieur au premier membre de b ou si le premier membre de b n'est pas inférieur au premier membre de a et que le second membre de a est inférieur au second membre de b.
  Contrat : l'infériorité des paires requiert que les types T1 et T2 soient *ordonnables*.

Il s'agit d'un ordre lexicographique.

Paramètre	Description
T1	Le type du premier membre de la paire
T2	Le type du second membre de la paire

TABLE 1 – Paramètres de type de la classe paire

Les autres relations (opérateurs `!=`, `>`, `<=` et `>=`) entre deux paires a et b sont définissables à partir des relations d'égalité et d'infériorité. Ces autres relations sont donc des formes syntaxiques de commodité dont les sémantiques sont définies par :

- $a!{=}b \equiv \neg(a{==}b)$
- $a{>}b \equiv b{<}a$
- $a{<=}b \equiv \neg(b{<}a)$
- $a{>=}b \equiv \neg(a{<}b)$

D'aucun préférerait $\neq$, $>$, $\leq$ et $\geq$.

La spécification que nous proposons au client est en fait la classe `pair<T1,T2>`[5], qui fait partie des outils proposés dans le header `<utility>` de la STL. Dans la suite de ce document, nous considérerons que la paire est celle de la STL définie dans le standard du C++.

Techniquement donc, la réalisation de cette classe peut se résumer à un `#include` et un `typedef`.

La section suivante est un exemple de manuel d'utilisation de la paire, nous y décrivons la classe et proposons des exemples d'utilisation.

4 Manuel d'utilisation

4.1 Définition

La classe paire de la catégorie utilitaires de la STL est définie dans le header standard `<utility>` et dans le header non-standard maintenu pour compatibilité `<pair.h>`.

4.2 Description

Une *paire* `pair<T1,T2>` est une classe paramétrée selon deux types T1 et T2 *assignables*, donnés à la déclaration de la paire, voir tableau 1. Une paire contient deux objets accessibles publiquement : un objet de type T1, nommé *premier membre*, et un autre de type T2, comme *second membre*. Le tableau 2 décrit les membres de la classe paire.

Une paire n'est pas un *conteneur* au sens de la STL car elle ne respecte pas les spécifications d'un conteneur, entre autres, la paire ne fournit pas d'itérateurs sur ses éléments.

5. `http://www.sgi.com/tech/stl/pair.html`

Membre	Description
`first_type`	alias pour le premier type
`second_type`	alias pour le second type
`first`	le premier membre public de la paire
`second`	le second membre public de la paire
`pair()`	constructeur par défaut, requiert que T1 et T2 soient constructibles par défaut
`pair(const T1& __a, const T2& __b)`	constructeur principal, requiert que T1 et T2 soient assignables
`pair(const pair<_U1, _U2>& __p)`	constucteur de copie. Convertit implicitement U1 en T1 et U2 en T2.
`pair& operator=(const pair&)`	opérateur d'affectation, par défaut
`template <class T1, class T2> bool operator==(const pair&, const pair&)`	relation d'égalité entre paires, requiert que T1 et T2 soient comparables
`template <class T1, class T2> bool operator<(const pair&, const pair&)`	relation d'infériorité entre paires, requiert que T1 et T2 soient ordonnables

TABLE 2 – Membres de la classe paire

4.3 Paramètres de type

Les types T1 et T2 doivent être *assignables*. Les opérations supplémentaires requièrent des conditions supplémentaires. Le constructeur par défaut de la paire requiert que T1 et T2 soient *constructibles par défaut*, l'opérateur == requiert que T1 et T2 soient *comparables*, l'opérateur < requiert que T1 et T2 soient *ordonnables*.

4.4 Exemple d'utilisation

Plus généralement tout conteneur de type Unique Associative Container dispose d'une méthode `insert` qui retourne une paire.

Dans la STL par exemple, la méthode `insert(const value_type& x)` de la classe générique `set` retourne comme type `pair<iterator, bool>`. Le second membre de la paire indique si l'élément a été ajouté ou s'il existe déjà, le premier membre est un itérateur qui pointe soit là où a eu lieu l'insertion soit sur l'élément déjà existant. Il s'agit d'un exemple d'utilisation où la paire est utilisée pour retourner deux valeurs, comme dans l'exemple suivant :

```
#include <utility>
...
pair<bool, double> result = do_a_calculation();
if (result.first)
  do_something_more(result.second);
else
  report_error();
```

4.5 Voir aussi

Concepts *assignable*, *constructible par défaut*, *ordonnable* et *comparable*.

4.6 Exemple d'application

L'exemple suivant illustre l'utilisation de deux paires :

```
#include <iostream>                // cout
#include <cstdlib>                 // EXIT_SUCCESS
#include <utility>                 // pair<T1,T2>
#include <string>                  // string

using namespace std;
int main()
{
        pair<string,int> r;
        // Crée une paire vide qui contient une chaîne et un entier.
        pair<string,int> p("bye", 18);
        // Une paire initialisée chaîne-entier

        cout << "La paire p contient " << p.first << " et " << p.second << endl;
        // affichage :
        //La paire p contient bye et 18

        cout << "La paire r contient " << r.first << " et " << r.second << endl;
        // affichage :
        //La paire r contient  et 0

        r=pair<string,int>("hello", 17);
        // réaffectation de r
        cout << "La paire r contient " << r.first << " et " << r.second << endl;
        // affichage :
        //La paire r contient hello et 17

        cout << "p et r sont elles égales ? " << ((p==r)?"oui":"non")<<endl;
        // affichage :
        //p et r sont elles égales ? non

        p = r;
        //copie le contenu de r dans p
        cout << "p et r sont elles égales ? " << ((p==r)?"oui":"non")<<endl;
        // affichage :
        //p et r sont elles égales ? oui
        cout << "p est elle inférieure à r ? " << ((p<r)?"oui":"non")<<endl;
        // affichage :
        //p est elle inférieure à r ? non

        r.second++;
        //on modifie le second membre de r
        cout << "p est elle inférieure à r ? " << ((p<r)?"oui":"non")<<endl;
        // affichage :
        //p est elle inférieure à r ? oui

        return EXIT_SUCCESS;
}
```

5 Réalisation

L'étape de conception en informatique est celle où sont fait les choix pour la réalisation des spécifications, c'est dans cette étape qu'on détermine la meilleure structure de données pour avoir de bonnes performances par exemple. Dans cet exemple, la conception est quasiment inexistante, à part peut être le choix des méthodes en ligne (`inline`) pour efficacité.

À vrai dire, l'auteur à rétro-spécifié la classe paire et traduit des documents existants.

La réalisation est extraite du header standard de la STL `stl_pair.h`, situé (selon version de gcc, ici la 4.1.3) dans `/usr/include/c++/4.1.3/bits/`. Les modifications apportées pour l'exemple concernent la suppression des informations de licence, la traduction des commentaires et la suppression de la fonction globale `make_pair`. Ce code ne respecte pas les conventions du guide de style C++ IF.

En C++, la seule différence entre une classe et une structure est la visibilité par défaut des membres.

En C++, le mot-clef `class` utilisé dans les paramètres de template peut-être remplaçé par `typename`.

```
#ifndef _PAIR_H
#define _PAIR_H 1

namespace std
//la paire est définie dans le namespace standard de la stl
{
        // une paire contient deux objets de types arbitraires passés en paramètres
        template<class T1, class T2>
        struct pair{
                typedef T1 first_type;  //  alias pour le premier type
                typedef T2 second_type; //  alias pour le second type

                T1 first;               // le premier membre de la paire
                T2 second;              // le second membre de la paire

                pair() : first(), second() { }
                //constructeur par défaut, requiert que
                //T1 et T2 soient constructibles par défaut

                pair(const T1& __a, const T2& __b) : first(__a), second(__b) { }
                //constructeur principal, requiert que
                //T1 et T2 soient assignables

                template<class _U1, class _U2>
                pair (const pair<_U1, _U2>& __p) :
                        first(__p.first), second(__p.second) { }
                //constucteur de copie de la classe paire elle même
                //requiert que T1 et T2 soient assignables
        };//struct pair


        template<class T1, class T2>
        inline bool operator==(const pair<T1, T2>& __x, const pair<T1, T2>& __y)
                { return __x.first == __y.first && __x.second == __y.second; }
        //deux paires sont égales ssi leurs membres respectifs sont égaux
        //requiert que T1 et T2 soient comparables

        template<class T1, class T2>
        inline bool operator<(const pair<T1, T2>& __x, const pair<T1, T2>& __y)
                { return __x.first < __y.first
                        || (!(__y.first < __x.first) && __x.second < __y.second); }
        //définition membre à membre de la relation d'infériorité
        //requiert que T1 et T2 soient ordonnables
```

```
        template<class T1, class T2>
        inline bool operator!=(const pair<T1, T2>& __x, const pair<T1, T2>& __y)
                { return !(__x == __y); }
        //définition basée sur l'opérateur ==

        template<class T1, class T2>
        inline bool operator>(const pair<T1, T2>& __x, const pair<T1, T2>& __y)
                { return __y < __x; }
        //définition basée sur l'opérateur <

        template<class T1, class T2>
        inline bool operator<=(const pair<T1, T2>& __x, const pair<T1, T2>& __y)
                { return !(__y < __x); }
        //définition basée sur l'opérateur <

        template<class T1, class T2>
        inline bool operator>=(const pair<T1, T2>& __x, const pair<T1, T2>& __y)
                { return !(__x < __y); }
        //définition basée sur l'opérateur <
} // namespace std
#endif /* _PAIR_H */
```

5.1 Fonction outil supplémentaire

Par commodité, par exemple quand on manipule des dictionnaires, une fonction globale `make_pair( T1 x, T2 x)` est proposée dans la STL. Elle prend en paramètre un objet constant de type T1 et un objet constant de type T2 et retourne une paire de types T1 et T2 en utilisant le constructeur principal :

```
template<class _T1, class _T2>
inline pair<_T1, _T2> make_pair(_T1 __x, _T2 __y)
        { return pair<_T1, _T2>(__x, __y); }
```

Attention donc lors de l'utilisation de `make_pair` avec de « gros » objets.

Pour des raisons techniques, cette fonction prend désormais (version 2003 du standard C++ [1]) ses paramètres par valeur et non par référence constante. En effet le choix initial posait des problèmes lors du passage de chaînes de caractères constantes [6].

6 Tests fonctionnels

Au total la batterie comporte une trentaine de tests pour la classe paire.

La `libstdc++` du GNU dispose d'une batterie de tests pour la collection de compilateurs gcc et la STL. L'outillage et les options de configurations de cette batterie sont relativement complexes et ne peuvent être détaillés ici. Les vérifications effectuées sur la paire concernent principalement les types passés en paramètres, la correction des constructeurs et des deux opérations principales. Le plan de tests croise les vérifications selon trois aspects :

6. `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#181`

1. le fonctionnement avec comme type en paramètre :
   - `int`,
   - `float`,
   - `const char *`,
   - `gnu_obj`, un type construit minimal,
   - `gnu_t`, un type construit minimal générique,
2. selon que les types T1 et T2 soient :
   - homogènes,
   - hétérogènes,
   - `const`,
   - `const &`,
3. pour les méthodes :
   - le constructeur principal de `pair`,
   - la fonction `make_pair`,
   - l'algorithme générique `swap`, qui permet d'intervertir deux objets.

Le header standard `<cassert>` permet d'utiliser la fonction `assert(c)` qui évalue l'expression `c` passée en paramètre. Si `c` est évaluée à faux, alors `assert` affiche un message sur la sortie d'erreur standard et appelle `abort()` qui arrête l'exécution du programme. Si `c` est évaluée à vrai, l'exécution continue.

À titre d'exemple, voici la réalisation des tests[7] pour le cas où les types en paramètres sont hétérogènes. `VERIFY()` est une macro proche de la fonction `assert()` : le test est validé si la condition en paramètre est évaluée à vrai.

```
#include <utility>

class gnu_obj
{
  int i;
public:
  gnu_obj(int arg = 0): i(arg) { }
  bool operator==(const gnu_obj& rhs) const { return i == rhs.i; }
  bool operator<(const gnu_obj& rhs) const { return i < rhs.i; }
};

template<typename T>
  struct gnu_t
  {
    bool b;
  public:
    gnu_t(bool arg = 0): b(arg) { }
    bool operator==(const gnu_t& rhs) const { return b == rhs.b; }
    bool operator<(const gnu_t& rhs) const { return int(b) < int(rhs.b); }
  };

// heterogeneous
void test01()
{
  std::pair<bool, long> p_bl_1(true, 433);
  std::pair<bool, long> p_bl_2 = std::make_pair(true, 433);
  VERIFY( p_bl_1 == p_bl_2 );
  VERIFY( !(p_bl_1 < p_bl_2) );

  std::pair<const char*, float> p_sf_1("total enlightenment", 433.00);
  std::pair<const char*, float> p_sf_2 = std::make_pair("total enlightenment",
                                                        433.00);
```

7. http://gcc.gnu.org/svn/gcc/trunk/libstdc++-v3/testsuite/20_util/pair

```
  VERIFY( p_sf_1 == p_sf_2 );
  VERIFY( !(p_sf_1 < p_sf_2) );

  std::pair<const char*, gnu_obj> p_sg_1("enlightenment", gnu_obj(5));
  std::pair<const char*, gnu_obj> p_sg_2 = std::make_pair("enlightenment",
                                                          gnu_obj(5));
  VERIFY( p_sg_1 == p_sg_2 );
  VERIFY( !(p_sg_1 < p_sg_2) );

  std::pair<gnu_t<long>, gnu_obj> p_st_1(gnu_t<long>(false), gnu_obj(5));
  std::pair<gnu_t<long>, gnu_obj> p_st_2 = std::make_pair(gnu_t<long>(false),
                                                          gnu_obj(5));
  VERIFY( p_st_1 == p_st_2 );
  VERIFY( !(p_st_1 < p_st_2) );
}
```

Tester des classes génériques peut s'avérer délicat, en raison du niveau l'abstraction élevé des spécifications. Des méthodologies de tests appliquées à la programmation générique sont proposées dans [2]. La bibliothèque Boost propose notamment des outils facilitant les tests de programmes génériques [8] qui permettent de vérifier des concepts [3].

7 Remarques

7.1 Rigueur des spécifications

La rigueur des spécifications présentées dans ce document dépasse largement celle attendue dans un compte rendu de TP du département et même dans l'industrie. En revanche, elle est souhaitable, voire exigée, dans les publications scientifiques, les ouvrages de référence [4, 5] ou les standards [1].

7.2 Ressources en ligne

La majeure partie de ce document est une traduction de la documentation de la STL *originale* de Silicon Graphics disponible en ligne [9], c'est une ressource de référence. Le tableau 3 donne à ce propos les correspondances entre les termes que nous avons utilisé dans le document et ceux de la STL. La faq de la librarie standard GNU est également une documentation de référence exhaustive. Par exemple, l'implémentation de l'opérateur < y est commentée. [10]

8. http://www.boost.org/libs/libraries.htm#Correctness
9. http://www.sgi.com/tech/stl/
10. http://gcc.gnu.org/onlinedocs/libstdc++/20_util/howto.html#pairlt

Terme du document	Terme de la STL
paire	`pair`
assignable	`Assignable`[11]
constructible par défaut	`DefaultConstructible`[12]
comparable	`EqualityComparable`[13]
ordonnable	`LessThanComparable`[14]

TABLE 3 – Correspondance entre les termes du document et ceux de la STL

7.3 Édition du document

Ce document a été réalisé avec LaTeX 2_ε avec l'éditeur Kile de KDE. Il est sous licence GPL. Les sources sont disponibles auprès de l'auteur.

8 Références

[1] International Standardization Organization (ISO) and International Electrotechnical Commission (IEC). International standard ISO/IEC 14882 :2003. Programming languages - C++, second edition. Technical report, ANSI - American National Standards Institute, Octobre 2003.

[2] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates : The Complete Guide*. Addison-Wesley, 2002.

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[4] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[5] Ulrich Breymann. *Designing Components with the C++ STL, Third Edition*. Addison-Wesley, 2002.

[6] Jeff Cogswell, Christopher Diggins, Ryan Stephens, and Jonathan Turkanis. *C++ Cookbook*. O'Reilly, 2005.

[7] Scott Meyers. *Effective C++, Third Edition*. Addison-Wesley, 2005.

11. `http://www.sgi.com/tech/stl/Assignable.html`
12. `http://www.sgi.com/tech/stl/DefaultConstructible.html`
13. `http://www.sgi.com/tech/stl/EqualityComparable.html`
14. `http://www.sgi.com/tech/stl/LessThanComparable.html`

9 Annexe

9.1 Pourquoi un constructeur de copie template ?

Le constructeur de copie de la classe paire est défini ainsi :

```
template<class _U1, class _U2>
        pair (const pair<_U1, _U2>& __p) ...
```

Il s'agit d'une méthode membre générique définie dans une classe générique. La justification de ce choix est l'item 45 de [7] : *Use member function template to accept "all compatible types"*. Cette technique, appelée "generalized copy constructor" permet de construire une paire de type `pair<T1,T2>` à partir de types `pair<_U1, _U2>` *quelconques*, pourvu que U1 et U2 soient convertibles en types T1 et T2 respectivement.