ML-style Tuple Assignment in Standard C++ — Extending the Multiple Return Value Formalism

Jaakko Järvi

Turku Centre for Computer Science
TUCS Technical Report No 267
April 1999
ISBN 952-12-0434-6
ISSN 1239-1891

Abstract

It has been shown previously that tuples can be implemented with templates in standard C++. This article extends the author's previous work on tuples and multiple return values by defining generic functions and classes to support a kind of a multivalued assignment. The return values of a function returning a tuple can be assigned to separate variables with a single statement. E.g., assuming that foo is a function returning a 3-tuple and that a, b and c are variables of correct type, the expression tie(a, b, c) = foo() assigns the three returned elements to a, b and c.

The presented template definitions result in a typesafe and intuitive tuple formalism, resembling the the tuple constructs in ML, for defining and calling functions having multiple return values. The produced code is efficient. Resource consumption of compiling increases to some extent, but hardly significantly on real programs.

Keywords: generic programming

TUCS Research Group
Algorithmics group

1 Introduction

In a previous article [1], the problem of multiple return values was addressed. It was argued that *tuples*, as found in the programming languages ML[2] or Python[3], provide a concise and intuitive means for returning multiple values from functions. To compensate the lack of direct support for tuples in C++, a set of generic functions and classes implementing a tuple formalism were described in the article. Instead of using reference parameters to pass data out of functions, or writing separate classes for grouping return values, the presented templates definitions make it possible to use tuple types as return values.

We shortly revisit the SVD example from [1]. SVD is a matrix decomposition operation decomposing a matrix into two matrices U, V and one vector S, representing a typical situation, where a function produces several results. The following code snippets show the am. three possibilities for returning the three resulting objects.

```
// 1. Using reference parameters:
void SVD(const Matrix& m, Matrix& U, Vector& S, Matrix& V);
```

```
// 2. Using a separate class:
struct SVD_result {
  Matrix U, V;
  Vector S;
  // constructor, etc.
};
SVD_result SVD(const Matrix& m);
```

```
// 3. Using tuples:
tuple<Matrix, Vector, Matrix> SVD(const Matrix& m);
```

As demonstrated by this example, the prototypes of functions returning multiple values can be defined very naturally using tuples.

Functions returning tuples can called as follows (see [1] for the details of the element access in tuples):

```
Matrix U, V; Vector S;
    ...
tuple<Matrix, Vector, Matrix> result = SVD(aMatrix);
U = get<1>(result); S = get<2>(result); V = get<3>(result);
```

This is rather convenient. However, it would be even more convenient to be able to assign the tuple elements to variables in a single statement. This is possible, for example, in ML and Python. In numerical programming, a similar construct is used extensively in the scripting language of MATLAB [4]. Using these languages, the SVD example could be written as:

```
(U, S, V) = SVD(aMatrix)        # Python
val (U, S, V) = SVD(aMatrix);   (* ML *)
[U, S, V] = SVD(aMatrix);       % Matlab
```

Though it may not be apparent, this powerful construct is attainable in standard C++ as well. In C++ syntax, the above example can be written as:

```
Matrix U, V; Vector S;
        ...
tie(U,S,V) = SVD(aMatrix);
```

The idea of redirecting return values in the above manner was originally presented in a usenet article by Ian McCulloch. The technique was proposed to be used with functions returning *std::pair* objects, being thus limited to only two elements.

This article shows how the techique can be generalised to tuples of arbitrary length and describes the implementation in detail. The implementation extends the tuple formalism presented in [1]. New class and function tempates are added to allow the simultaneous assignment of elements. The existing definitions require no changes.

2 Assigning tuple elements to variables

How does the above example C++ code work? Fig. 1 depicts the workings of the example graphically. The right-hand side of the assignment expression tie(U,S,V) = SVD(aMatrix) is an ordinary call to a function returning a tuple object. In the left-hand side, tie is a function template. The evaluation of the whole expression comprises of the following steps, where the order of steps 1 and 2 is implementation dependent:

1. The function call SVD(aMatrix) returns a tuple object, the type of which is tuple<Matrix, Vector, Matrix>.

2. tie(U, S, V) is a call to an instantiation of a generic function. This generated function creates and returns a *tier* object, a kind of a tuple as well. The elements of this tier-tuple are references to variables U, S and V.

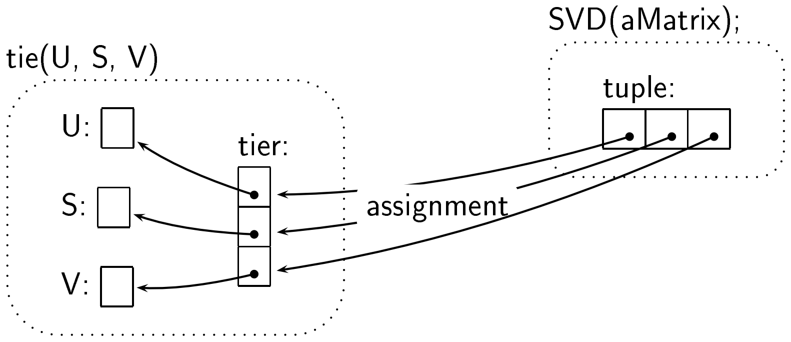

Figure 1: The evaluation of the expression tie(U, S, V) = SVD(aMatrix) illustrated graphically. The leftmost and rightmost dotted frames outline the results of the left-hand and right-hand sides of the expression.

3. The assignment operation assigns the tuple object to the tier object, performing an element-wise assignment from the tuple elements to the tier elements. Since the tier elements are references, the actual destinations of the assignments are the original variables referred to, that is, U, S and V.

4. The temporary tier object is destructed.

2.1 About terminology

The tier acts as an intermediary object *binding* the returned tuple elements to a set of variables, and indeed, as opposed to tying, binding is a more conventional term in the programming language literature. However, the terms *tie* and *tier* were chosen, since the C++ standard library contains a set of binder templates for a different purpose.

3 Tier template definitions

The structure of the tier template definitions correspond to the structure of the tuple template definitions in [1]. The tuple code is repeated in its relevant parts in appendix A.

The backbone of the tuple implementation is the *cons* template, implementing a compile time list structure. Different recursive instantiations of this template provides tuples the ability to store an arbitrary number of elements of arbitrary types. Tiers need the same capability, with the distinction that the elements are references. For this purpose the ref_cons template, a modifcation of the cons template, is defined:

```
struct nil {};

template <class HT, class TT> struct ref_cons {
  HT& head; TT tail;
};

template <class HT> struct ref_cons<HT, nil> { HT& head; };
```

The empty nil class represents the end-mark of the list. Now, for example the instantiation

```
ref_cons<Matrix, ref_cons<Vector, ref_cons<Matrix, nil> > >
```

defines a type containing a Vector reference and two Matrix references as member variables. To evade this unwieldy syntax, the ref_cons template is not instantiated directly. The tier template defines a more usable interface and inherits a suitable ref_cons instantiation. The tier template allows a variable number of template arguments, up to a predefined limit, using the technique presented in [1]. For terseness, this limit is chosen to be four here, but it is straightforward to increase it by extending the template parameter lists:

```
template <class T1, class T2 = nil, class T3 = nil, class T4 = nil>
struct tier : public tier_to_ref_cons<T1, T2, T3, T4>::U {...};
```

The tier_to_ref_cons template is a recursive traits class [5] mapping the tier template parameters to the correct ref_cons instantiation.

```
template <class T1, class T2, class T3, class T4>
struct tier_to_ref_cons {
  typedef ref_cons<T1, typename tier_to_ref_cons<T2, T3, T4, nil>::U > U;
};

template <class T1> struct tier_to_ref_cons<T1, nil, nil, nil> {
  typedef ref_cons<T1, nil> U;
};
```

These definitions allow, for example, the previous example to be written as tier<Matrix, Vector, Matrix>. For a more verbose description of recursive type mappings, see [1, 6].

3.1 Constructing tiers

The tier and ref_cons constructors are structurally equivalent to the tuple and cons constructors. First the tier constructor:

```
template <class T1, class T2 = nil, class T3 = nil, class T4 = nil>
struct tier : public tier_to_ref_cons<T1, T2, T3, T4>::U {
      ...
  tier( T1& t1, T2& t2=wrap_ref<T2>(),
        T3& t3=wrap_ref<T3>(), T4& t4=wrap_ref<T4>())
    : tier_to_ref_cons<T1, T2, T3, T4>::U(t1, t2, t3, t4) {}
      ...
```

The constructor takes four parameters (four being the upper limit chosen), and delegates them to the base class constructor defined in section 3.1.2. The intention is not to require four arguments in each constructor call. The number of arguments accepted by the constructor should equal the number of actual (non-nil) elements in the tier, ranging from one to the maximum number of allowed elements in a tuple. This is achieved with the wrap_ref template used as default arguments.

3.1.1 Default arguments

The actual arguments to the tier constructor are the variables to which the tuple elements should be assigned. There are thus no reasonable default values for the actual arguments. On the other hand, default values are needed for the unspecified arguments of type nil. Since the number of nil-arguments is not know beforehand, the default argument expressions should return a suitable default value if the type of the argument is nil, and generate a compile-time error for other types. The following definitions fulfill these requirements:

```
template <class T> struct ct_error {};

namespace { nil dummy_nil; }

template <class T> struct wrap_ref_struct {
  static T& f() {
    return ct_error<T>::missing_argument; // Yields a compile time error
  }
};

template <>
struct wrap_ref_struct<nil> { static nil& f() { return dummy_nil; } };

template <class T> T& wrap_ref() { return wrap_ref_struct<T>::f(); }
```

The separation between nil and non-nil template arguments can not be accomplished directly with the wrap_ref template, since the template parameter must be explicitely specified, prohibiting thus specialisations. Consequently, the function template wrap_ref is just an interface, delegating the task to a static function f of a template class wrap_ref_struct, which can be specialized.

In the specalisation wrap_ref_struct<nil>, the static function f returns correctly a reference to a nil object. Since the parameters to the tier constructor are passed as references, even the nil parameters must refer to some existing object, hence the dummy_nil object. It is placed in an unnamed namespace to make it local to a translation unit. As becomes clear in the sequel, the need for the dummy_nil object is temporary, the final tier object does not contain any references to this object.

The function f in the primary template is instantiated whenever a default argument is needed for any other type than nil. This introduces a compile time error, as required, by referring to a non-existing member of a class template.

3.1.2 Constructing ref_cons objects

The tier constructor passes its parameters, nil and non-nil ones, directly to the constructor of its base class:

```
template <class HT, class TT> struct ref_cons {
      ...
  template <class T2, class T3, class T4>
    ref_cons( HT& t1, T2& t2, T3& t3, T4& t4)
      : head(t1), tail(t2, t3, t4, dummy_nil) {}
      ...
};
template <class HT> struct ref_cons<HT, nil> {
      ...
  ref_cons(HT& t1, nil, nil, nil) : head (t1) {}
      ...
};
```

This constructor recursively initializes each element of the tier. At each recursive call, a reference to the first argument is stored in the current head and the remaining arguments are passed to the constructor of the tail.

3.2 Tie function templates

The above definitions allow the definition of tier types and construction of tier objects. For example, tier<Matrix, Vector, Matrix>(U, S, V) creates a tier holding references to U, S and V. The objective was, however, to be able to construct tier objects using a simpler syntax tie(U, S, V), avoiding thus the explicit specification of the element types. This can be attained by wrapping the constructor call inside a function template, analogously to the *make_pair* template in the standard library, and letting the compiler deduce the argument types. However, this is somewhat problematic.

The C++ standard bans default template arguments in function templates. Neither do default arguments have any effect on the template argument deduction [7, section 14.8.2.4]. Due to these restrictions, there seems to be no way to define a tier function template, which would be generic with respect to the number of arguments. Hence, a separate function template must be written for each allowed tier length. Albeit not very elegant, this is however perfectly feasible. In any case, the maximum number of elements in tuples is not very high. As functions with very long parameter lists tend to be error-prone and impractical, same is true for very long tuples as return values. The need for tuples longer than few dozens of elements is hardly justified. Consequently, the amount of code duplication is quite tolerable. Futhermore, as the code duplication does not affect the client code in any way, except of course by providing the desired functionality, the duplication remains merely as a source of discontent to the author.

As an example, the two and three argument tie function templates are defined as follows:

```
template<class T1, class T2>
inline tier<T1, T2> tie(T1& t1, T2& t2) {
  return tier<T1, T2>(t1, t2);
};

template<class T1, class T2, class T3>
inline tier<T1, T2, T3> tie(T1& t1, T2& t2, T3& t3) {
  return tier<T1, T2, T3>(t1, t2, t3);
};
```

4 Assigning tuples to tiers

We now have means to conveniently create tier objects, but the assignment operators from tuples to tiers are yet to be defined. As stated in section 2, these operators perform an element-wise assignment from tuple to tier elements.

Separate assignment operations are required in the tier template, as well as in the ref_cons templates. Each of these assignment operations are defined as member templates. The parameter in the tier assignment is constrained to be a tuple.

```
template <class T1, class T2 = nil, class T3 = nil, class T4 = nil>
struct tier : public tier_to_ref_cons<T1, T2, T3, T4>::U {
    ...
  template <class V1, class V2, class V3, class V4>
  void operator=(const tuple<V1, V2, V3, V4>& t) {
    tier_to_ref_cons<T1, T2, T3, T4>::U::operator=(t);
  }
    ...
```

The tuple is redirected as such to the assignment operator of the base class, performing the actual assigning. Note, that even though the definition does not restrict the types of the tuple elements in any way, the assignment is typesafe. An attempt to assign a tuple with incompatible element types is caught at some of the assignments of the nested members of the ref_cons base class.

```
template <class HT, class TT> struct ref_cons {
    ...
  template <class HT2, class TT2>
  void operator=(const cons<HT2, TT2>& u) { head=u.head; tail=u.tail; }
    ...
};
template <class HT> struct ref_cons<HT, nil> {
    ...
  template <class HT2>
  void operator=(const cons<HT2, nil>& u) { head = u.head; }
    ...
};
```

The assignment operations in the ref_cons templates assign the elements recursively. Again, the parameters are only restricted to be of type cons, the element types are not confined explicitly. However, the assignment of the heads fails, if the types HT and HT2 are incompatible. In sum, the assignment leads to a compile time error, if the tuple and tier are not of the same length and if the types of the corresponding elements are not assignable.

4.1 Performance considerations

4.1.1 Runtime cost

As shown in [1], there is no performance penalty in returning tuples compared with returning explicitely defined classes from functions. The introduction of tiers does not change this, since the use of tiers induce no changes to functions returning tuples. However, the code for calling functions having multiple return values is different, as tiers add a new mechanism for redirecting the resulting values. In this section, we focus on the cost of this redirection. Consider the code:

```
// A, B, C, D are some classes
tuple<A, B, C, D> foo();
        ...
A a; B b; C c; D d;
        ...
tie(a, b, c, d) = foo();
```

The tie function creates the tier object. Since the tie function, tier constructor, and the constructors of the inherited ref_cons classes are all inlined, the effect of the tier construction process is roughly equivalent to:

```
A& a_ = a; B& b_ = b; C& c_ = c; D& d_ = d;
```

foo is a function returning a tuple. Right after foo has been executed, the resulting tuple resides somewhere in the memory (generally near the top of the stack) available for furher use. Now the assignment operator of the tier object is called. The operator assigns the elements of the tuple to the tied variables. Again, the assignment operations are all inlined, eliminating all overhead and leaving only the calls to the assignment operations of the individual elements. This step corresponds to the statements:

```
a_ = get<1>(foo_r); b_ = get<2>(foo_r);
c_ = get<3>(foo_r); d_ = get<4>(foo_r);
```

foo_r stands for the temporary tuple object returned from foo. As the left hand sides are references, the statements are assignments to a, b, c and d.

No destructor was defined for the tier template, the compiler generated destructor is trivial [7, chapter 12.4] and thus yields no code.

In sum, the cost arising from the tying mechanism is the cost of creating one reference variable and performing one assignment for each element of the tuple. The details are obviously dependent on the compiler, but this is more or less the behaviour one would expect from an optimising C++ compiler.

These assumptions were tested by studying the compiled code of several test programs. A diverse set of tuples and tiers was used: the elements ranged from built-in types to rather complex classes and the length of tuples was varied. Two compilers were used: KAI C++ (verion 3.3c) and Egcs (release 1.1.1).

Both compilers confirmed the expected behaviour. The code produced by Egcs was constistently in accordance with the above description. KAI C++ could in some cases even avoid allocating stack space for the reference variables by locating them into registers; the extra assignment was still performed. As mentioned in [1], KAI C++ was not capable of inlining very long tuples entirely, unless exceptions were turned off by a compiler switch.

Due to the extra assignment, there is some overhead in the tier mechanism compared with using reference parameters to pass data out from a function. For lightweight objects with low assignment cost the overhead is most likely not significant. For large objects with high copy or assignment semantics, the extra cost may be notable. However, returning such objects, inside tuples or otherwise, should be avoided anyhow. Instead, large objects ought to be wrapped inside auto pointers, reference counted pointers etc. providing cheap copy and assignment. Hence, tiers can be applied without a significant performance penalty in vast majority of the cases where multiple values need to be returned.

4.1.2 Compilation cost

Compiling code containing tuple and tier definitions is somewhat slower and requires more memory than compiling corresponding code not containing these definitions. There is discussion on the effect of tuples to compilation time and memory consumption in [1], showing that this effect is clear but on real programs probably not significant. The introduction of tiers to the multiple return value formalism does not change this, which was confirmed with two compilation tests.

The compilation times of tier constructs were compared with the compilation times of corresponding definitions using reference parameters to 'return' multiple values. This is a natural choise, since in both cases there is a set of existing variables to which the results should be bound. To measure the direct compilation speed difference of the constructs, five pairs of test programs were generated. Each pair contained a program using the tier mechanism and a functionally equivalent program using reference parameters in multivalued returns. The tier programs comprised solely of functions returning tuples and calls to these functions using the tier mechanism, hence the compilation times reflect the direct compilation cost difference between the two alternatives for returning multiple values.

The results are shown in Fig. 2, indicating clearly the slower compilation times of tier constructs. However, the compilation of parameter passing constructs is only a small part of the whole compilation process. Hence, to estimate the effect on the compilation times of real C++ programs, an existing

Tuple length	Egcs (T_{tier}/T_{ref})	KAI C++ (T_{tier}/T_{ref})
2	12.9	23.09
4	13	26.27
8	14.83	32.81
16	16.42	55.81
32	22.72	71.43

Figure 2: The relative compilation time of programs using tuples and tiers to equivalent programs using reference parameters to pass multiple values from a function (T_{tier} = compilation time of tuple implementation, T_{ref} = compilation time of reference parameter implementation). To eliminate the effect of any constant costs of the compilation process, the compilation time of a program containing an empty main function was subtracted from the results. The times are not comparable across compilers.

program was selected to represent an 'average' C++ program. The program included a few standard headers and used standard generic containers and algorithms quite frequently. A reasonable amount of tier constructs were added to the program. As a result, the program comprised of 150 functions, 20% of which returned tuples and 20% used the tier mechanism to call the tuple returning functions. The length of tuples ranged from 2 to 8, shorter tuples being more common than longer ones. The lenghts and percentages were selected based on the authors estimations about the frequency of functions with multiple return values. The program is intended to represent quite an extensive use of the tier mechanism.

The compilation of this program was compared with a functionally equivalent program using reference parameters instead of the tier mechanism. The increase in compilation time was 5% in Egcs and 11% in KAI C++. Memory consumption increased more, 27% (Egcs) and 22% (KAI C++).

5 Conclusion

An implementation of a generic tuple class was introduced in [1] to alleviate the definition and calling of functions with multiple return values in C++. This article extends the tuple implementation, describing the tier mechanism enabling a kind of a multivalued assignment, i.e., a means to assign the elements of a tuple to separate variables with a single statement.

Compared with using reference parameters as output parameters, the presented mechanism has a small performance penalty, basically an extra assignment is needed for each tuple element. Compiling code using the tier mechanism requires more resources than conventional code. The tests performed suggest that these increases are probably not significant, though possibly notable with very extensive usage of the mechanism.

Tuples and tiers together provide a convenient and intuitive formalism for defining and calling functions with multiple return values. The techniques are efficient and do not compromise type safety.

A Relevant parts of tuple implementation

```
struct nil {};

template<class HT, class TT> struct cons {
  HT head; TT tail;
  template <class T2, class T3, class T4>
  cons(const HT& t1, const T2& t2, const T3& t3, const T4& t4)
    : head(t1), tail(t2, t3, t4, nil()) {}
};
template <class HT> struct cons<HT, nil> {
  HT head;
  cons(const HT& t1, const nil&, const nil&, const nil&) : head (t1) {}
};

template <class T1, class T2, class T3, class T4>
struct tuple_to_cons {
  typedef cons<T1, typename tuple_to_cons<T2, T3, T4, nil>::U > U;
};
template <class T1>
struct tuple_to_cons<T1, nil, nil, nil> { typedef cons<T1, nil> U; };

template <class T> inline T wrap() { return T(); };

template <class T1,class T2=nil,class T3=nil,class T4=nil>
struct tuple : public tuple_to_cons<T1, T2, T3, T4>::U {
  tuple( const T1& t1=wrap<T1>(), const T2& t2=wrap<T2>(),
         const T3& t3=wrap<T3>(), const T4& t4=wrap<T4>())
    : tuple_to_cons<T1, T2, T3, T4>::U(t1, t2, t3, t4) {}
};
```

References

[1] Järvi J.: *Tuples and multiple return values in C++*, TUCS Techincal Report No 249, http://www.tucs.fi/publications.

[2] Paulson L. C.: *ML for the working programmer*, Cambridge University Press, 1991.

[3] http://www.python.org.

[4] The Mathworks Inc. (http://www.mathworks.com): *Using MATLAB, Version 5*, 1997.

[5] Myers, N. C.: *A new and useful template technique: 'traits'*, C++ Report, Vol. 7 no 5 pp. 32-35, 1995.

[6] Järvi J.: *Compile Time Recursive Objects in C++*, Proceedings of the TOOLS 27 conference, Beijing Sept. 1998, pp. 66-77, IEEE Computer Society Press.

[7] *International Standard, Programming Languages – C++*, ISO/IEC:14882, 1998.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**