Tuples and multiple return values in C++

Jaakko Järvi

Turku Centre for Computer Science
TUCS Technical Report No 249
March 1999

ISBN 952-12-0401-X
ISSN 1239-1891

Abstract

A generic *tuple* class, capable of storing an arbitrary number of elements each being of arbitrary type, is presented. The class offers a concise means to return multiple values from a function. Instead of using refrerence parameters to pass data out of functions, the function return type is an instantiation of the tuple template, grouping the values to be returned. The semantics of the tuple class is analogous to the pair template in the C++ standard library, thus the usage of tuples is convenient and intuitive. The implementation is described in details.

Efficiency considerations are addressed. It is shown that with modern C++ compilers, there is no extra runtime overhead caused by using tuples. However, due to excessive template instantiatons, there is some effect on compilation times and memory usage during compilation. This effect is unlikely to be significant in real programs.

Keywords: generic programming

TUCS Research Group
Algorithmics group

1 Introduction

In most common programming languages (including C++) functions can return only a single value. This is a reasonable restriction, although quite often there is a need to pass several related values from a function to the caller. For example, consider the numerous cases, where in addition to the actual result, one or more status codes are given. Particularly typical multiple return values are in numerical libraries, where besides the extensive need for status codes, the actual results themselves often have several components. E.g., a common matrix decomposition operation, singular value decomposition (SVD), takes a matrix and decomposes it to two matrices and a vector. Still another example is the usage of the `pair` template with STL maps.

To circumvent the restriction of a single return value, reference or pointer parameters can be used to pass data out of functions. Alternatively a struct or class type can be defined to group the values to be returned into a single object. In the first approach, the distinction between input and output parameters is not always obvious and one may need to add extra comment lines just for clarifying this (although disciplined usage of common idioms, such as using non-const pointers and references only for output parameters does help). The second approach is conceptually better: there are no multiple return values, just a single value, which happens to be a composition of several related values. All the parameters are input parameters and the return value is the sole output.

Consider the SVD example. One would either write

```
void SVD(const Matrix& M, Matrix& U, Vector& S, Matrix& V);
```

and leave it to the callers responsibility to provide `U`, `S` and `V` suitably initialized, or use the latter approach:

```
struct SVD_result {
  Matrix U, V;
  Vector S;
  // possibly a constructor etc.
};
SVD_result SVD(const Matrix& M);
```

Even though in this case the distinction between input and output values is clear, it is a burden to write small classes just to serve as a collection of return values of some individual function. New unnecessary names are added to the namespace, and in addition to the name and prototype of a function, the caller must also know the return class and its behavior. In this article, *tuples* are proposed as an alternative way of returning multiple values from a function.

1.1 Tuples

Tuples are fixed size collections of objects of arbitrary types. They provide a concise means for multi-valued returns. The return type of a function can simply be defined as a tuple, e.g.:

```
<Matrix, Vector, Matrix> SVD(Matrix); // pseudo code
```

The intention of the declaration is instantly clear and there is no need to specify an artificial wrapper class for the result.

Some programming languages, such as Python[1], Scheme[2] or ML[3], have tuple constructs or analogous mechanisms for multiple return values. There is also an ongoing effort to include tuples to Eiffel[4]. C++ has no direct support for tuples. However, the generic features of the language offer potential solutions. The pair template in the standard library implements 2-tuples, allowing us to write, for example

```
pair<Matrix, Matrix> LU(Matrix); // another matrix decomposition
```

Although the pair suffices for 2-tuples, it does not provide a general solution. It is of course possible to write templates for triples, quadruples, and so on, but where to stop?

This article describes a solution, which covers all lengths of tuples with a single template. The solution is completely type-safe requiring no runtime checks. Furthermore, with modern optimizing C++ compilers, the usage of tuples (creation and element access) has no performance penalty compared with using normal structs as return values. Also, independent of the number of elements, *tuple* is the common name for all tuple types (instead of tuple2, tuple3, etc.). The template definitions set a certain predefined upper limit for the length of tuples, but in any case tuples of several dozens of elements are allowed, which should be enough for most of the imaginable uses.

Note that the code presented relies on the new template features of the C++ standard[5], such as member templates, partial specialization and explicit specialization of function templates. Not all compilers currently support these features. The code in this article was successfully compiled with the Egcs and KAI C++ compilers, both being on the leading edge with respect to the conformance to the new standard.

2 Tuple Template

A tuple template must be able to store an arbitrary number of elements of arbitrary types. A template having this capability is surprisingly simple:

```
struct nil {};

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

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

As the name suggests, the definition corresponds to the LISP dot pair. Instantiating `TT` with a `cons` recursively leads to a list structure (see [6] for a more detailed description of these *compile time recursive objects*). E.g., the instantiation representing the tuple type `<Matrix, Vector, Matrix>` is

```
cons<Matrix, cons<Vector, cons<Matrix, nil> > > aTuple;
```

This instantiation defines a nested structure of classes, containing the tuple elements as nested member variables. The `nil` class is just an empty placeholder class. The elements can be accessed with the usual syntax (e.g. `aTuple.tail.tail.head` refers to the third element). This `cons` template is the underlying construct for representing tuples.

Even though the `cons` template is sufficient for representing the structure of tuples, the syntax is not usable. Directly defining tuples as dot pair lists with the `cons` template would be rather awkward, and indeed a better syntax can be attained. The objective is to be able to write type declarations, such as `tuple<int>`, `tuple<float, double, A>` etc. Since the number of template parameters of a generic class can not vary and neither is the definition of several templates with the same name possible, the solution is to use default arguments for template parameters:

```
template <class T1,class T2=nil,class T3=nil,class T4=nil>
struct tuple ...
```

Now this definition allows between 1 to 4 template arguments. The unspecified arguments are of type `nil`. As pointed out above, we need to set an upper limit for the number of elements in any tuple. To make the code sections short, the limit of four elements is used here. It is, however, straightforward to extend the tuple to be able to handle more elements.

Now the `tuple` template provides the desired interface, whereas the `cons` template implements the underlying structure of tuples. The interface and implementation can be connected via inheritance. A tuple inherits a suitable instantiation from the cons template: `tuple<float, int, A, nil>` for example inherits from `cons<float, cons<int, cons<A, nil> > >`. It is reasonable to assume that a tuple only stores its non-nil elements, hence `nil` classes are excluded from the `cons` instantiation. To be able to implement this inheritance relation, we need a means to express the type to be inherited using the template parameters of the tuple.

2.1 Mapping tuple types

A recursive *traits* (more on traits, see [5]) class is needed to define the mapping from tuple parameters to a suitable cons instantiation:

```
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;
};
```

The sole purpose of this class is to define a mapping from the 4 template parameters of `tuple` to a corresponding instantiation of the `cons` template. This is given by the typedef `U` in the `tuple_to_cons` class. It defines a dot pair, where the head type is the first template parameter `T1` and the tail type is defined recursively. The recursive definition instantiates the `tuple_to_cons` template again, with `T1` dropped from and `nil` added to the parameter list. This is repeated, until the partial specialization of `tuple_to_cons` matches.

As an example, consider `tuple<float, int, A, nil>`. To resolve the underlying type of the typedef `tuple_to_cons<float, int, A, nil>::U`, the following chain of instantiations occurs:

```
tuple_to_cons<float, int, A, nil>::U
≡ cons<float, tuple_to_cons<int,A,nil,nil>::U >
≡ cons<float, cons<int, tuple_to_cons<A,nil,nil,nil>::U > >
≡ cons<float, cons<int, cons<A,nil> > >.
```

In the last step, the partial specialization is applied.

Using the `tuple_to_cons` type mapping the tuple class can now be defined as:

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

Now it is clear that tuple instantiations up to four elements with arbitrary types are valid and inherit a cons instantiations capable of storing the elements. For example, the following definitions are all correct instantations of the tuple template.

```
tuple<int, int>
tuple<Matrix, Vector, Matrix>
tuple<A, B, tuple<C, D>, E>
```

3 Construction semantics

The templates above define just the structure of tuple types. For tuples to be usable, we need convenient mechanisms for constructing tuples and accessing their individual elements. The construction and element access semantics can be made consistent of the semantics of the pair template in the standard library:

```
template< class T1, class T2>
struct pair {

  T1 first; T2 second;

  pair() : first(T1()) , second(T2()) {}
  pair(const T1& x, const T2& y) : first(x), second(y) {}

  template<class U, class V>
    pair(const pair<U, V>& p)
      : first(p.first), second(p.second) {}
};
```

Hence, by default a pair is initialized to the default values of its element types. The elements may also be given explicitly in the construction. The member template is a 'copy constructor', which allows type conversions between elements. This is the construction semantics a general tuple template should have as well. Furthermore, the element access should be as straightforward as with pairs (`p.first`, `p.second`, etc.). Let us first focus on the construction semantics.

3.1 Tuple constructor

The tuple template above allows up to four elements. It is therefore obvious that the tuple constructor should have four parameters. But tuples can also be shorter. For an n-tuple, there should be a constructor taking n parameters. To avoid writing a separate constructor for each different length, default arguments can be used. The definition of the tuple template becomes:

```
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) {}
};
```

The constructor has a distinct parameter for each element to be stored. The parameters are passed directly to the inherited cons template instantiation. We have not yet defined any constructors for the cons template, so just assume for now that the constructor works reasonably. Each parameter is given a default value. Though looking somewhat odd, the default arguments are expressions that merely create and return a new default object of the current element type. Given this definition, a constructor for an n-element tuple can be called with 0 to n parameters and the elements left unspecified are constructed using their default constructors. Especially, the unused `nil` objects are created by the default argument expressions, hiding their existence entirely from the user of the tuple template.

3.1.1 Default arguments

Why not just use `T()` instead of `wrap<T>()` as the default argument? Suppose `A` is a class with no public default constructor. Then the constructor of the `tuple<A>` instantiation is `tuple<A>(const A& t1 = A(), ...)`. This results in a compile time error, since the constructor call `A()` is invalid. Hence a type without a default constructor would not be allowed as an element type of a tuple. This is the case even if the default argument is never used. The solution is to wrap the default constructor call to a function template[1]:

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

Now the constructor of `tuple<A>` becomes:

```
tuple<A>(const A& t1 = wrap<A>(), ...)
```

[1]In the egcs version 1.0.2. this is not yet supported. A class template with a static function: `template<class T> struct wrap { static T f() { return T(); }; ` must be used instead of a function template. The call becomes: `wrap<T>::f()`.

If the first parameter is always supplied, the `wrap<A>` template is never instantiated.[2] The compiler only checks the semantic constrains of the default argument and finds out that `wrap<A>()` is indeed a valid expression. Only if the default argument is really used, the compiler is allowed to instantiate the body of the wrap function template (and flag the error if there is no default constructor for the type in question)[5, section 14.7.1.].

3.2 Constructing dot pairs

The tuple constructor passes all four parameters to the constructor of the inherited `cons` instantiation. Hence, the `cons` constructor also has four parameters. This constructor is a member template, where only the type of the first parameter is fixed and the remaining parameter types are deduced. Here are the definitions of the cons templates:

```
template<class HT, class TT> struct cons {
  HT head; TT tail;

  template <T2, T3, 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) {}
};
```

The constructor initializes the head with the first parameter and passes the remaining parameters to the tail's constructor recursively, until all but the first parameters are nil. At each level, one element is initialized. Note that even though only the first parameter's type is directly bound, the types of the remaining parameters are in fact also mandated by the instantiation of the `cons` template. Hence, a constructor call with erroneous argument types result in a compile time error at some point during the recursive instantiations.

As an example, let us consider in detail a particular constructor call `tuple<Matrix, Vector, Matrix>(U,S,V)`. When the call is encountered,

[2]Version 3.3.c of KAI C++ incorrtecly instantiates the default argument. According to their support, this is to be addressed in their next major release. For a workaround, contact the author.

the tuple template is instantiated: the missing fourth template parameter is first added and `tuple<Matrix, Vector, Matrix, nil>` becomes the complete instantiated type. The `tuple_to_cons` traits class computes the list type `cons<Matrix, cons<Vector, cons<Matrix, nil> > >` to be inherited. The tuple constructor is called with arguments `U`, `S` and `V`. The fourth parameter is not given, so the default argument is applied: `wrap<nil>()` is used to construct an empty object of type `nil`. Now the cons constructor is called with arguments `U`, `S`, `V` and `nil()`, which recursively initializes the elements of the tuple with `U`, `S` and `V`.

3.3 Constructor allowing element-wise conversions

The constructors analogous to the third constructor of the pair template are still to be defined. The intention is to allow 'copy' construction from another tuple with different element types, if the elements can be implicitly converted. For example, `tuple<int, double, int>` could be initialized with an object of type `tuple<char, int, int>`. For pairs, this is useful mostly in situations, where the pairs are created using the `make_pair` function template [8]. There does not seem to be a generic way to implement a corresponding `make_tuple` function (other than explicitly writing a function for each tuple length). It is thus a matter of taste, whether a 'converting' copy constructor is needed for tuples, but nevertheless it is not difficult to implement. An additional member template constructor is required in the tuple template:

```
template <class T1, class T2, class T3, class T4>
  template<class U1, class U2>
  tuple<T1,T2,T3,T4>::tuple(const cons<U1, U2>& p)
    : base(p) {}
```

In both of the cons templates (primary and specialization), a new constructor is needed.

```
//primary template
template<class HT, class TT>
  template<class HT2, class TT2>
  cons<HT,TT>::cons(const cons<HT2, TT2>& u)
    : head(u.head), tail(u.tail) {}

//specialization
template<class HT>
  template<class HT2>
  cons<HT, nil>::cons(const cons<HT2, nil>& u)
    : head(u.head) {}
```

The tuple constructor merely delegates the copy to the base class. In the base class, the copy constructor of each member is called along the recursion. Hence, the converting copy is allowed, if the types are element-wise compatible. Otherwise a compile time error results.

4 Accessing tuple elements

With the above definitions, the element access is tedious. For example, one must write `aTuple.tail.tail.tail.tail.head` to refer to the fifth element of a tuple. However, as stated above, element access should be as convenient as with pairs. Since the elements of tuples have no names (such as first, second, etc.), numbers are used to refer to them. With the aid of some additional template definitions, the `N`th element can be referred with the expression `get<N>(aTuple)`.[3]

Here `get` is a function template, where the index of the element to be accessed is given as an explicitly specified integral template argument. Obviously, the access mechanism must be recursive. However, the `get` function template can not directly be defined recursive. It is not possible to define a specialization with respect to a template parameter, which must be explicitly specified. Therefore, the recursion is implemented as a static member template of the class `nth`:

```
template<int N> struct nth {
  template<class HT, class TT>
  static void* get(cons<HT, TT>& t) {
    return nth<N-1>::get(t.tail);
  }
};

template<> struct nth<1> {
  template<class HT, class TT>
  static void* get(cons<HT, TT>& t) { return &t.head; }
};
```

Now `N` is a template parameter of a class and the specialization for `N`=1is thus allowed.

Since tuples are inherited from some `cons` instantiation, the element access functions can be defined to operate on `cons` dot pairs. The invocation

[3]The access functions can be defined as member templates as well, but since explicit specialisation is used, the `get` function must be explicitly qualified as a template. This would lead to the awkward syntax `aTuple.template get<N>()` instead of `aTuple.get<N>()`.

`nth<N>::get(aTuple)` returns a pointer to the `N`th element of the object `aTuple`. The access function works recursively returning the result of getting the (`N-1`)th element of the tail of `aTuple`. The specialization for `N`=1 merely returns the address of the head of the current dot pair. Note, that the access mechanism is safe with respect to the index parameter `N`, since an illegal index leads to a compile time error (no matching templates exist). Now we have a mechanism for getting the address of a given element, but not yet the type.

The type can be defined recursively: The type of the nth element of a tuple a equals the type of the $(n-1)$th element of the tail of a. With this definition in mind, we can write the corresponding recursive traits classes:

```
template <int N, class T> struct nth_type;

template <int N, class HT, class TT>
struct nth_type<N, cons<HT, TT> > {
  typedef typename nth_type<N-1,TT>::U U;
};

template<class HT, class TT>
struct nth_type<1, cons<HT, TT> > { typedef HT U; };
```

Now the type of the `N`th element of a dot pair type `T` can be written as `nth_type<N, T>::U`. Using this type definition, the actual `get` function template can be written as:

```
template<int N, class HT, class TT>
nth_type<N, cons<HT, TT> >::U&
get(cons<HT, TT>& c) {
   typedef
     typename nth_type<N, cons<HT, TT> >::U return_type;
   return *static_cast<return_type*>(nth<N>::get(c));
}
```

The `get<N>` function calls the `nth<N>::get` function to get the address of the `N`th element as a void pointer, casts it to the correct type and returns the result. Note that although the type information is seemingly lost, the cast from the void pointer is, however, entirely safe.[4]

As an example of the element access, consider accessing the third element of the tuple `a` defined as `tuple<Matrix, Vector, Matrix> a(U,S,V);` The invocation `get<3>(a)` triggers the following chain of instantiations:

[4]It is possible to carry the type information along in the `nth<N>::get` functions as well, but dropping it alleviates the task of the compiler considerably.

```
get<3,Matrix, cons<Vector, cons<Matrix, nil> > >(a)
→ nth<3>::get< Matrix, cons<Vector, cons<Matrix, nil> > >(a)
→ nth<2>::get<Vector, cons<Matrix, nil> >(a.tail)
→ nth<1>::get<Matrix, nil>(a.tail.tail).
```

Now the specialization `nth<1>` is used, and the head of `a.tail.tail`, which is the matrix `V`, is returned.

5 Efficiency

The tuple is intended to be an elementary utility template with widespread usage. Hence, efficiency is of utmost importance. However, the template definitions do not seem to be very efficient: The construction of tuples and accessing their elements requires several nested function calls. To access the `N`th element of a tuple with the `get<N>(aTuple)` function, `N`+1 functions altogether are invoked. However, the functions are all inlined 'one-liners'. In an optimizing C++ compiler, inline expansion eliminates the overhead of these functions and the address of the `N`th element is resolved at compile time.

The construction is analogous in this respect. Even though the construction of an n-tuple constructs n nested dot pairs, each constructor only effectively constructs its head and passes rest of the parameters forward. As the inline expansion is performed, the result is just the code performing the construction of the individual elements in a tuple. Particularly, the construction and destruction of the temporary `nil` objects are optimized away. This behavior was validated by experiments with the egcs (release-1.0.2, optimization flag -O) and KAI C++ (version 3.3c, flags +K3 -O2) compilers.

The obvious alternative for using the tuple template, is to explicitly write a struct containing an equivalent member variable for each tuple element. Suppose that a four-element tuple, composed of elements of types `A1`, `A2`, `A3` and `A4`, is needed. The alternative for `tuple<A1, A2, A3, A4>` is:

```
struct A {
  A1 first; A2 second; A3 third; A4 fourth;
  A(const A1& a1=A1(), const A2& a2=A2(),
     const A3& a3=A3(), const A4& a4=A4() )
      : first(a1), second(a2), third(a3), fourth(a4) {}
};
```

The comparisons between programs using these *explicitly written structs* vs. programs using tuples confirmed the anticipated behavior with both compilers. The compiled codes of object construction (e.g. the calls `A()` and `tuple<A1, A2, A3, A4>()`) were essentially identical for both approaches, consisting only of the code arising from the construction of the individual elements. Similarly, both compilers were capable of computing the relative address of a given element in a tuple at compile time, thus eliminating all overhead arising from the element access. E.g. in the example above, `get<3>(aTuple)` yields no code, just a reference to the third element of the tuple.

To assess the results, several tests were performed, varying the length and element types of tuples. Egcs eliminated all overhead in all cases. Even a tuple of 256 elements was compiled: the compiler eliminated the 257 nested function calls of the invocation `get<256>(aTuple)` and resolved the address of the 256th element of the tuple at compile time!

The optimization capabilities of the KAI C++ depended on whether exceptions were used or not (the compiler has a flag for turning exceptions on and off). With exceptions turned on, the compiler only reached the zero overhead for relatively short tuples. With exceptions off, there were no difficulties in optimizing longer tuples as well.

5.1 Effect on compilation time

Due to excessive template instantiations, it is inherently slower to compile tuples than corresponding explicitly written structs. Further, the compilation requires a greater amount of memory. The compilation speed difference was measured with some simple tests. All template definitions were included as a header file (this is natural, since every function is inlined). Four pairs of test programs were generated. Each pair had a program using tuples and another equivalent program using explicitly written classes. The intention was to measure the direct compilation speed difference between tuples and explicit classes, so the tuple programs consisted of nothing else than tuple definitions, element access and functions (basically empty) that returned tuples. The results are shown in Fig. 1.

Another test was performed to evaluate the effect of tuple usage to compilation speed in a realistic program. An existing program was modified and an'average' C++ program was generated. It consisted of 120 functions. 25% of the functions had tuples as return values. The lengths of the tuples varied from 3 to 8. The program included a few standard headers (iostream and some STL headers) and used STL containers and algorithms to some extent. The compilation of this program was now less than 4% slower than the compilation of an equivalent program, which used explicitly written structs instead of tuples. The increase in the amount of memory needed in the compilation was between 5-10%. Consequently, the use of tuples increase

Tuple length	Egcs (T_t/T_s)	KAI C++ (T_t/T_s)
3	7.70	5.33
5	8.06	5.90
10	10.8	7.40
32	13.4	16.8

Figure 1: The relative compilation time of programs using tuples to equivalent programs using explicitly written structs (T_t = compilation time of tuple implementation, T_s = compilation time of explicitly written struct implementation). The times are not comparable across compilers.

compilation times and memory consumption, but the increases are likely to be insignificant on real programs.

6 Conclusion

Tuples provide a clear and concise means to return multiple values from a function. Though C++ has no language constructs for tuples, they can be implemented in standard C++ using templates in a bit inventive fashion. The solution described requires only a few small generic classes and functions (and an up to date compiler) to achieve a completely type safe and efficient tuple implementation.

Up to a certain predefined limit, the proposed tuple template allows tuples to have an arbitrary number of elements, each element being of arbitrary type. The usage is simple and intuitive, semantics being analogous to the pair template in the standard library. The proposed tuple template simplifies the definition and use of functions which return multiple values and is worth of adding to the C++ programmer's basic toolbox.

References

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

[2] Ashley J. M. and Dybvig R. K.: *An Efficient Implementation of Multiple Return Values in Scheme*, Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, pp. 140-149, Orlando, June 1994.

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

[4] *Tuples, routine objects and iterators*, A draft proposal to NICE, http://eiffel.com (link 'Papers').

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

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

[8] Stroustrup, B.: *The C++ programming language, 3rd ed.*, p. 482, Addison-Wesley, 1997.