Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Tuple Types and Multiple Return Values

There are several ways to return multiple values from a function in C++. Here is an elegant way that will give your template-grokking brain a real workout.

August 01, 2001
URL:http://www.drdobbs.com/cpp/tuple-types-and-multiple-return-values/184401421

Introduction

In most common programming languages, including C++, functions can return only a single value. This is a reasonable restriction, but often there is a need to pass several related values from a function to the caller. For example, what if one or more status codes need to be returned in addition to the actual result? Or what if the result itself has several components? A common example of multiple return values is the use of the `std::pair` template with STL `map`s.

There are two common approaches to circumventing the restriction of returning only a single value. First, you can use extra reference or pointer parameters to pass more data out of the functions. With this approach, the distinction between input and output parameters is not always obvious, and extra comment lines may be needed just for clarifying this. (Although disciplined usage of common idioms, such as using non-const pointers and references only for output parameters, does help.) Alternatively, you can define a separate class to group the return values into a single returned object. This second approach is conceptually better because there are no multiple return values; instead, there is just a single value that happens to be a composition of several related values. All the parameters are input parameters and the return value is the sole output.

I'm going to take an example from the numerical domain and use it throughout the article. The SVD (singular value decomposition) is a common matrix decomposition operation. It takes a matrix (A) and decomposes it into two matrices (U, V) and a vector (S): SVD(A) = U S VH. Thus, the operation produces three result values.

Using the first approach, a typical function prototype and invocation might be:

```
void SVD(const Matrix* A, Matrix* U,
         Vector* S, Matrix* V);
...
Matrix *A, *U, *V; Vector *S;
...
SVD(A, U, S, V);
```

These lines of code leave open questions: which arguments are for input and which for output? Should the client initialize the output arguments `U, S`, and `V` in some way?

In the second approach, these questions are not relevant:

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

The distinction between input and output values is now clear. However, it's a lot of work to write small classes just to serve as a collection of return values of individual functions. New unnecessary names are added to the namespace. Furthermore, in addition to memorizing the name and prototype of a function, the programmer must also remember the return class and its behavior. Note that for a straightforward `Matrix` and `Vector` implementation, copying such a return class might be considerably less efficient than using pointer parameters. Here I assume `Matrix` and `Vector` to be handle classes with inexpensive copy operations.

Tuples provide a third approach. They allow a natural way to define the return value class on-the-fly as part of the function prototype, while avoiding the problems described above.

Tuples

A tuple type is the *cartesian product* of its element types; that is, a tuple type represents the set of ordered, fixed-size lists of objects, where the *i*th object in the list can have any value of the *i*th element type. More concretely, a tuple object contains a fixed number of other objects as elements, and these element objects can be of different types. Consequently, a tuple object contains a fixed number of other objects as elements. These element objects may be of different types. Some programming languages, such as Python [1] and ML [2], are equipped with tuple constructs. There is also an ongoing effort to include tuples in Eiffel [3]. Tuples provide a concise means for defining multiple return values: the return type of a function can simply be defined as a tuple, for example:

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

The intent of the declaration is instantly clear and there is no need to specify an artificial wrapper class for the result. Furthermore, with an appropriately defined assignment operation, tuples make the call sites clearer as well. For example, consider the following function calls written in Python and ML:

```
(U, S, V) = SVD(A);      #Python
val(U, S, V) = SVD(A);   (* ML *)
```

In these expressions, the returned tuple is *unpacked;* the components are assigned, or bound, to variables `U`, `S`, and `V`.

C++ has no direct support for tuples. However, the generic features of the language offer potential solutions for implementing them. The `pair` template in the standard library implements 2-tuples, though in a restricted form. For example, a prototype of a function returning two values can be defined as:

```
// Namespace std is assumed
// LU is another matrix decomposition
pair<Matrix, Matrix> LU(Matrix);
```

However, the assignment semantics of `pair`s does not allow the element unpacking described above.

This article presents a template library that implements a tuple abstraction for C++. The library has the following properties:

1) The abstraction is typesafe and efficient.

2) The semantics of the abstraction is natural and fits well with other language features. The abstraction is more or less a generalized standard `pair` template.

3) The library allows objects of practically any C++ type to be used as tuple elements. This includes class objects, built-in types, pointers, references, arrays, and even other tuples.

4) The number of elements in a tuple can be arbitrary. This is true up to some limit defined by the number of template specializations provided.

5) An element can be accessed based on its index with no run-time overhead.

6) Tuples can be copied and assigned; even the unpacking assignment semantics described above are supported.

Before focusing on how to implement these properties, let's take another look at the SVD example to introduce how the tuple library works. Omitting all code unrelated to tuples, the SVD function could be defined as follows:

```
tuple<Matrix, Vector, Matrix> SVD(const Matrix& A) {
  ...
  Matrix U, V; Vector S;
  ...
  return make_tuple(U, S, V);
};
```

The above use is analogous to standard `pair`s. When the function is called, there are two ways to access the results. First, you can store the result to a tuple object and access the elements with special access functions (`get`):

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

Alternatively, you can assign the tuple elements directly to the variables `U`, `S`, and `V` with the unpacking assignment construct:

```
tie(U, S, V) = SVD(A);
```

Implementation of the Tuple Template

To start with, 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 `cons` suggests, the definition corresponds to LISP-style lists. Instantiating `TT` with a struct `cons` recursively leads to a list structure (see [4] for a more detailed description of such compile-time lists). For example, the instantiation representing the tuple type `(Matrix, Vector, Matrix)` is

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

This instantiation defines a nested structure of classes, containing the tuple elements as member variables of these classes. The elements can be accessed with the usual syntax (e.g., `aTuple.tail.tail.head` refers to the third element). The `nil` class is an empty class acting as the end mark of the list.

Even though the `cons` template is sufficient for representing the structure of tuples, the syntax is not usable. Defining tuples with the `cons` template directly would be awkward. Instead, we'd like to be able to write type declarations like:

```
tuple<int>
tuple<float, double, A>
```

Since the number of template parameters of a generic class can not vary, and it's impossible to define several templates with the same name, the solution is to use default values for template parameters:

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

Now this definition allows between one and four template arguments. The unspecified arguments are of type `nil`. As pointed out above, there is an upper limit for the number of elements in any tuple. This is set by the number of template arguments in the template definition. To make the code sections short, the limit of four elements is used here. It is, however, straightforward to extend the tuple to allow more elements. Currently, I've coded up to ten-element tuples.

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 of the `cons` template. As an example,

```
tuple<float, int, A, nil>
```

inherits

```
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, the type to be inherited must be expressed with the template parameters of the tuple.

Mapping Tuple Types

A recursive *traits* class (for more on traits, see [5, 6, 7]) provides a typedef that defines the mapping from tuple parameters to the corresponding `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>::type >
    type;
};

template <class T1>
struct tuple_to_cons<T1, nil, nil, nil>
{
  typedef cons<T1, nil> type;
};
```

The `type` typedef defines a list, where the head type is the first template parameter `T1` and the tail type is defined recursively by instantiating the `tuple_to_cons` template again, with `T1` dropped from and `nil` added to the parameter list. The partial specialization of `tuple_to_cons` ends the recursion.

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

```
tuple_to_cons<float, int, A, nil>::type
= cons<float, tuple_to_cons<int, A, nil, nil>::type>
= cons<float, cons<int, tuple_to_cons<A,nil,nil,nil>::type> >
= cons<float, cons<int, cons<A, nil> > >
```

Finally, with this type mapping machinery, we can define the `tuple` class itself as:

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

Public inheritance lets us treat tuples as `cons` lists as well, which is needed for the element access functions. Furthermore, since these functions allow a direct access to the individual elements (see Accessing Tuple Elements), there is no reason to restrict the access here.

Tuples Are Generalized Pairs

So far, the templates define just the structure of tuple types. For tuples to be usable, we also need convenient ways to construct, copy, and assign tuples and access their elements. I introduced these mechanisms in the introductory section with the SVD example. This section describes the mechanism and their implementation in more detail.

Basically, tuples generalize the semantics of the standard `pair` template:

```
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) {}
};
```

The default `pair` constructor initializes its elements to the default values of its element types. The elements may also be given explicitly in the constructor call. The member template is a kind of a 'copy constructor,' which allows type conversions between elements. The `tuple` template has similar construction semantics. Furthermore, element access for `tuple` is almost as straightforward as for `pair` (`p.first`, `p.second`, etc.); `tuple` element access is accomplished just by index number `N`, via an overloaded function called `get<N>`.

Constructors

The `tuple` template above allows up to four elements, so it's 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, I use default arguments. 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>::type {
  tuple(typename convert<T1>::plain_to_cref t1 = def<T1>::f(),
        typename convert<T2>::plain_to_cref t2 = def<T2>::f(),
        typename convert<T3>::plain_to_cref t3 = def<T3>::f(),
        typename convert<T4>::plain_to_cref t4 = def<T4>::f())
    : tuple_to_cons<T1, T2, T3, T4>::type(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. I have not yet defined any constructors for the `cons` templates, so just assume for now that the `cons` constructors work reasonably. Let me first explain the parameter types and default arguments expressions.

Parameter Types

The standard library `pair` constructor takes its arguments as references to const. This prevents the use of a reference as an element of a pair. In such a case the constructor parameter would be of type reference to reference, which is an invalid type. (This may change in the future revisions of the C++ Standard.) In order to allow elements of reference types, the tuple constructor parameter types are mapped through the expression `convert<T>::plain_to_cref`. The `convert` template is a traits class which maps any non-reference type `T` to reference to const type `const T&`, but leaves reference types as such. This is an easy way to avoid reference to reference situations. The `convert` template can be written as follows (the `plain_to_ref` typedefs are needed later and thus defined here for convenience; they define a similar mapping to non-const reference types):

```
template<class T> struct convert {
  typedef const T& plain_to_cref;
  typedef T& plain_to_ref;
};
template<class T> struct convert<T&> {
  typedef T& plain_to_cref;
  typedef T& plain_to_ref;
};
```

Default Arguments

Each parameter in the constructor 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. The elements that are left unspecified are constructed using their default constructors. Particularly, the unused `nil` objects are constructed by the default argument expressions, hiding their existence entirely from the user of the `tuple` template.

Why not just use `T()` instead of `def<T>::f()` as the default argument? Suppose `A` is a class with no public default constructor. Then the constructor of the `tuple<A>` instantiation would be `tuple<A>(const A& t1 = A(), ...)`. This would result 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, even if the default argument, and thus the default constructor, were never used. To eliminate this restriction, the call to the default constructor is wrapped inside a function template.

```
template <class T> struct def {
  static T f() { return T(); }
};
```

Now the constructor of `tuple<A>` becomes `tuple<A>(const A& t1 = def::f<A>(), ...)`. If the first parameter is always supplied, the `def::f<A>` template is never instantiated. The compiler only checks the semantic constraints of the default argument and finds out that `def<A>::f()` is indeed a valid expression. Only if the default argument is really used may the compiler instantiate the body of the `f` function in the `def` template (and flag the error if there is no default constructor for the type in question) [8, Section 14.7.1].

Furthermore, reference parameters should obviously not have default values. A specialization for reference types takes care of this:

```
template <class T> struct def<T&> {
  static T& f()
    { return error<T&>::no_default_for_references; }
};
```

The `error` template does not have the requested member, so the instantiation intentionally results in a compile-time error.

Constructing cons Objects

The `tuple` constructor passes all four parameters to the constructor of the inherited `cons` instantiation. This constructor is a member template whose template arguments are to be deduced. Each of the `cons` templates needs a constructor:

```
template<class HT, class TT> struct cons {
  ...
  template <class T1, class T2,
    class T3, class T4>
  cons(T1& t1, T2& t2, T3& t3, T4& t4)
    : head (t1) , tail (t2, t3, t4,
      the_nil) {}
};

// specialization to end the recursion
template <class HT> struct cons<HT, nil> {
  ...
  template<class T1>
  cons(T1& 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 parameters but the first are `nil`. At each level, one element is initialized and one `nil` object (`the_nil`) is added to the end of the parameter list. Even though the parameter types are deduced and in that sense unconstrained, the construction is typesafe; if the constructor is called with an argument of invalid type, the initialization of the head at some recursion level results in a compile-time error.

As an example of the construction mechanism, let's step through the effect of the constructor call `tuple<Matrix, Vector, Matrix>(U,S,V)` in detail. First, the `tuple` template is instantiated: the default value is first substituted for the missing fourth template argument and the complete instantiated type becomes `tuple<Matrix, Vector, Matrix, nil>`. The `tuple_to_cons` traits class computes the list type `cons<Matrix, cons<Vector, cons<Matrix, nil> > >`, from which the tuple inherits. Next, the `tuple` constructor is called with arguments `U`, `S`, and `V`. The fourth parameter is not given, so the default argument is applied: the invocation `def::f<nil>()` returns an object of type `nil`. Now the `cons` constructor is called with arguments `U`, `S`, `V`, and the temporary `nil` object. Finally, the constructor recursively initializes the elements of the list with `U`, `S`, and `V`.

Copy and Assignment

The constructors corresponding to `pair`'s templated constructors are still to be defined. The intent is to allow copy construction from another tuple with different element types, provided that there is an implicit conversion between the types of the corresponding elements. For example, `tuple<int, double, int>` could be initialized with an object of type `tuple<char, int, int>`. To attain this functionality, an additional member template constructor is required in the `tuple` template:

```
template <class T1, class T2, class T3,
  class T4>
struct tuple ... {
  ...
  template<class U1, class U2>
  tuple(const cons<U1, U2>& p)
    : tuple_to_cons<T1,T2,T3,T4>::type(p)
    {}
  ...
};
```

A new constructor is needed in both `cons` templates (primary and specialization) as well:

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

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

The `tuple` constructor merely delegates the copying work 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 and only if the types are element-wise compatible. Otherwise a compile-time error results.

The implementation of the assignment operation is structurally identical and thus not shown.

Accessing Tuple Elements

With the definitions described so far, element access is still tedious. For example, to refer to the fifth element of a tuple `x`, you would have to write `x.tail.tail.tail.tail.head`. This section describes the template definitions that let you refer to the `N`th element by writing `get<N>(x)`. (See note [9].)

The implementation of the access mechanism is based on recursive template definitions: the *n*th element of a list is the (*n*-1)th element of the tail of the list. The implementation consists of three components: the `get` function template provides the interface; the actual access functions are defined as member templates of the `tuple_element` class templates; and the traits class `tuple_element_type` resolves the type of the element to be accessed.

Let's start with the last component. The traits class is recursive: the type of the *n*th element of a list equals the type of the (*n*-1)th element of the tail of the list. With this definition in mind, we can now write the templates:

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

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

template<class HT, class TT>
struct tuple_element_type<1, cons<HT, TT> > {
  typedef HT type;
};
```

The type of the `N`th element of some list type `T` can be written as `tuple_element_type<N, T>::type`. For example, the type of `tuple_element_type<2, cons<int, cons<float, nil> > >::type` is `float`.

Next, consider the access functions. They are not defined as ordinary function templates, but rather as static member function templates, because the index parameter `N` is not deducible and must therefore be explicitly specified. It is not possible to define a specialization with respect to such template parameter, so I made the index `N` a template parameter of the class template, which allows the specialization that ends the recursion.

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

template<> struct tuple_element<1> {
  template<class RET, class HT, class TT>
  inline static RET get(cons<HT, TT>& t) {
    return t.head;
  }
};
```

The invocation `tuple_element<N>::template get<RET>(x)` [10] returns the `N`th element of the object `x` and converts the result to type `RET`. What should be the value of this template parameter? The access functions should obviously return a reference to the accessed element. Suppose the element is of some non-reference type `T`. Then `RET` should equal to `T&`. If the element itself is a reference, say of type `T&`, then `RET` should obviously be `T&` as well. It is the task of the interface function template to instantiate the access template functions with an appropriate value for `RET`. The same value is the return type of the interface function as well:

```
template<int N, class HT, class TT>
typename
convert<typename
   tuple_element_type<N,cons<HT, TT> >
   ::type>::plain_to_ref
inline get(cons<HT, TT>& c) {
   typedef typename
   convert<typename
      tuple_element_type<N,
                         cons<HT, TT> >
      ::type>::plain_to_ref RET;
   return tuple_element<N>::template
      get<RET>(c);
}
```

The return type is specified with two traits classes. First, the type of the accessed element is resolved with the `tuple_element_type` traits and then converted with the `convert` templates to a reference type, if it is not that already. The resulting type is then used as the `RET` template parameter in the call to the access function [11]. The tuple library implements the interface and access functions for const tuples as well. They are analogous and not shown here.

As an example of element access, consider the definition:

```
tuple<Matrix, Vector, Matrix> x(U,S,V);
```

The invocation `get<3>(x)` first triggers the instantiation

```
get<3, Matrix, cons<Vector, cons<Matrix, nil> > >(x).
```

which instantiates

```
tuple_element_type<3,
  cons<Matrix, cons<Vector, cons<Matrix, nil> > > >::type
```

which resolves to the type `Matrix`, the type of the third element of `x`. The instantiation `convert<Matrix>::plain_to_ref` maps this type to `Matrix&`. Then the following instantiations are performed:

```
tuple_element<3>::get<Matrix&, Matrix,
                      cons<Vector, cons<Matrix, nil> > >(x)
tuple_element<2>::get<Matrix&, Vector,
                      cons<Matrix, nil> >(x.tail)
tuple_element<1>::get<Matrix&, Matrix, nil>(x.tail.tail)
```

The last instantiation is the specialization for `N=1`, which returns the head of `x.tail.tail`, which is the matrix `V`. Note that the access mechanism is safe with respect to the index parameter. An index to a non-existing element leads to a compile-time error (no matching templates exist).

make_tuple

For easy construction of `pair`s, the C++ Standard library contains the `make_pair` function template:

```
template<class T1, class T2>
pair<T1, T2> make_pair(const T1& a, const T2& b) {
  return pair<T1, T2>(a, b);
}
```

It is somewhat problematic to provide the same functionality for tuples. A `make_tuple` function which would be generic with respect to the number of arguments in the tuple cannot be defined. Consequently, a separate function template must be written for each allowed tuple length. While not very elegant, this is perfectly feasible since 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, the same is true for very long tuples as return values. For example, the two-argument `make_tuple` function template can be defined as:

```
template<class T1, class T2>
tuple<typename mt_convert<T1>::type,
      typename mt_convert<T2>::type>
make_tuple(const T1& a, const T2& b) {
  return tuple<typename mt_convert<T1>::type,
               typename mt_convert<T2>::type>(a, b);
}
```

The template is similar to `make_pair`, except for the `mt_convert` type mappings which exist to make the template more generally applicable. The standard `make_pair` template cannot be used with array types, including string literals. For example, `make_pair("doesn't", "work")` tries to construct an object of type `pair<char[8], char[5]>`, which fails in the element initialization. The foremost task of the `mt_convert` templates is to circumvent this restriction:

```
template<class T>
struct mt_convert {
  typedef T type;
};
template<class T, int n> struct mt_convert<T[n]> {
  typedef const T (&type)[n];
};
```

The primary template provides an identity mapping, but the specialization for arrays instructs the tuple to store arrays as references. With these definitions, `make_tuple("does", "work")` does work. In addition, the library defines specializations and helper templates that give the programmer more control over the tuple creation. For example, `ref` and `cref` are template functions which can be used to explicitly state that the tuple should store a reference to the argument, instead of copying it (`ref` stands for reference to non-const, `cref` for reference to const):

```
int i, j;
make_tuple(i, j);            // creates tuple<int, int>
make_tuple(ref(i), cref(j)); // creates tuple<int&, const int&>
```

The Unpacking Assignment

I showed the expression `tie(U, S, V) = SVD(A)` in the introduction. The effect of this expression is to assign the elements of the tuple returned by

`SVD(A)` to variables `U`, `S`, and `V`. The evaluation of the expression comprises the following steps (with the order of steps 1 and 2 being implementation dependent):

1. The function call `SVD(A)` returns a tuple object of type `tuple<Matrix, Vector, Matrix>`.
2. `tie(U, S, V)` constructs a tuple of type `tuple<Matrix&, Vector&, Matrix&>`. The elements of this tuple are references to variables `U`, `S`, and `V`.
3. The assignment operation performs the element-wise assignment from the right-hand side tuple to the left-hand side tuple. Since the elements of the left-hand side tuple are references, the actual destinations of the assignments are the variables `U`, `S`, and `V`.
4. The tuple objects constructed in steps 1 and 2 are destroyed.

The ability to store references as tuple elements gives us the unpacking assignment semantics almost for free. All that is needed is a set of `tie` function templates that make the definition of tuples with reference elements convenient. The definitions of the `tie` function templates are analogous to the `make_tuple` templates, where a separate template is needed for each allowed tuple length. Here is the two-argument case as an example:

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

I chose the name *tie* because the templates are *tying* or *binding* a set of variables to the tuple elements. I didn't use *bind* because that name already has an established meaning in the standard library. (To be precise, the same is true of `tie`. It's a `basic_ios` member function for tying an `istream` and `ostream` together.)

As a small additional example of the unpacking assignment operation, we can swap the values of two variables, say `a` and `b`, with the expression:

```
tie(a, b) = make_tuple(b, a);
```

I've also coded an assignment operator from `pair`s to tuples. This lets us use the unpacking assignment with functions returning standard `pair`s as well. [12]

Efficiency

Tuple is a basic utility that's intended to be widely used, so its efficiency is important. The template definitions do not seem like they would be very efficient, however: to construct tuples and access their elements requires several nested function calls. For example, to access the *n*th element of a tuple requires *n*+1 function calls. The functions, however, are all inlined 'one-liners' and so inline expansion can eliminate the overhead of calling these functions. For example, a call such as `get<N>(aTuple)` can reduce to the address of the `N`th element, producing no extra code at all.

The same efficiency is true of construction. Even though in the construction of an *n*-tuple, *n* nested classes are constructed, each constructor only effectively constructs its head and passes the remaining parameters forward. As the inline expansion is performed, the result is just the code performing the construction of the individual elements. The `nil` objects can be optimized away entirely.

The obvious alternative for using the tuple template is to explicitly write a struct containing an equivalent member variable for each tuple element. With an optimizing C++ compiler, constructing, copying, and assigning tuples is just as efficient as constructing, copying, and assigning the corresponding explicitly written structs. The same is true for element access. This behavior was validated by experiments with the gcc and KAI C++ compilers. As an extreme example, I even constructed a 256-tuple; The gcc C++ compiler eliminated the 257 nested function calls of the invocation `get<256>(aTuple)` and resolved the address of the 256th element at compile time. See [13] for detailed descriptions of the experiments.

The unpacking assignment, on the other hand, does have a small performance penalty compared to passing the return values as non-const pointer or reference parameters. Consider our SVD example `tie(U, S, V) = SVD(A);`. The `tie` function call creates a set of references, while the `SVD(A)` returns a temporary tuple object. It is unrealistic to expect that the compiler could avoid the creation of the temporary and assign the results directly to `U`, `S`, and `V` by performing some kind of return value optimization. After all, `U`, `S`, and `V` may reside anywhere in the memory space. Therefore, the cost arising from the tying mechanism is the cost of creating one reference variable and performing one assignment operation for each element of the tuple. The details are obviously dependent on the compiler, but this is more or less the behavior we would expect from an optimizing C++ compiler.

Effect on Compilation Time

Due to the excessive template instantiations, it is inherently slower to compile tuples than corresponding explicitly written structs. See [13, 14] for detailed results of compile time comparisons between different ways of returning multiple return values with different compilers. These tests show that a tuple can be an order of magnitude slower to compile than the corresponding conventional explicitly written construct. However, as tuple instantiations comprise only a small part of any real C++ program, the effect on the overall compilation time is not significant.

To validate this, I created a test program, which I considered to represent extensive but realistic use of tuples. The characteristics of the test program were as follows: 20% of functions returned tuples, with the number of elements ranging from 2 to 8; and 20% of function calls used `tie` templates and the unpacking assignment expressions. The increase in compilation time compared to a conventional program was between 5 and 11 percent, depending on the compiler. Memory consumption increased more, between 22 and 27 percent.

Conclusions

This article describes a generic tuple library for C++ — basically a generalized pair with a lot of extra features. At a general level, a tuple is a *typesafe* container for objects of arbitrary, and potentially different, types. I emphasized the use of tuples as return values, but this is not their only usage. They provide a means to pass a varying number of arguments *into* a template function as well. More generally, wherever you would need a template with a varying number of parameters, tuples may be part of the solution.

The library source code can be downloaded from the *CUJ*'s source code archive (see <www.cuj.com/code>). The current implementation supports tuples up to ten elements.

Acknowledgement

Many thanks to Gary Powell and Herb Sutter for their comments on the draft version of this paper.

References

[1] The Python home page: <http://www.python.org>.

[2] Lawrence C. Paulson. *ML for the Working Programmer* (Cambridge University Press, 1991).

[3] "Tuples, Routine Objects and Iterators," a draft proposal to NICE, <http://eiffel.com> (link Papers).

[4] Jaakko Järvi. "Compile Time Recursive Objects in C++," *Proceedings* of the TOOLS-27 conference, Beijing, September 1998 (IEEE Computer Society Press), pp. 66-77.

[5] Nathan C. Myers. "A New and Useful Template Technique: Traits," *C++ Report,* 7(5): 32-35, 1995.

[6] Andrei Alexandrescu. "Traits: the Else-If-Then of Types," *C++ Report,* 12(4), 2000.

[7] Andrei Alexandrescu. "Traits on Steroids,"*C++ Report,* 12(6), 2000.

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

[9] Note that it is also possible to define the access function template as a member of the `tuple` class. The syntax becomes then `x.get<N>()`. The current implementation implements both types of access functions.

[10] The template parameter is explicitly specified, hence the additional `template` keyword in the call `element<N-1>::template get<RET>`.

[11] It is possible to drop the `RET` parameter and resolve the return type similarly in the `element<N>::get` functions as well. However, resolving the type only once and not at every recursive instantiation alleviates the task of the compiler considerably.

[12] Ian McCulloch presented this idea of using a class with reference members for unpacking the elements of a pair in a Usenet article, 1998.

[13] Jaakko Järvi. "Tuples and Multiple Return Values in C++," *Technical Report 249, 1999,* Turku Centre for Computer Science, <www.tucs.fi/publications>.

[14] Jaakko Järvi. "ML-Style Tuple Assignment in Standard C++ — Extending the Multiple Return Value Formalism," *Technical Report 267, 1999*, Turku Centre for Computer Science, <www.tucs.fi/publications>.

Jaakko Järvi is a researcher at Turku Centre for Computer Science. He has a Ph.D. in Computer Science from University of Turku, Finland. He can be reached at `jaakko.jarvi@cs.utu.fi`.