
5.1. Pairs and Tuples

In C++98, the first version of the C++ standard library, a simple class was provided to handle value pairs of different types without having to define a specific class. The C++98 class was used when a value pair was returned by standard functions and the container elements were key/value pairs.

TR1 introduced a tuple class, which extended this concept for an arbitrary but still limited number of elements. Implementations did portably allow tuples with up to ten elements of different types.

With C++11, the tuple class was reimplemented by using the concept of variadic templates (see Section 3.1.9, page 26). Now, there is a standard tuple class for a heterogeneous collection of any size. In addition, class `pair` is still provided for two elements and can be used in combination with a two-element tuple.

However, the `pair` class of C++11 was also extended a lot, which to some extent demonstrates the enhancements that C++ as a language and its library received with C++11.

5.1.1. Pairs

The class `pair` treats two values as a single unit. This class is used in several places within the C++ standard library. In particular, the container classes `map`, `multimap`, `unordered_map`, and `unordered_multimap` use `pair`s to manage their elements, which are key/value pairs (see Section 7.8, page 331). Other examples of the use of `pair`s are functions that return two values, such as `minmax()` (see Section 5.5.1, page 134).

The structure `pair` is defined in `<utility>` and provides the operations listed in Table 5.1. In principle, you can create, copy/assign/swap, and compare a `pair<>`. In addition, there are type definitions for `first_type` and `second_type`, representing the types of the first and second values.

Table 5.1. Operations of `pair`s

Operation	Effect
`pair<T1,T2>` *p*	Default constructor; creates a pair of values of types `T1` and `T2`, initialized with their default constructors
`pair<T1,T2>` *p*(*val1*,*val2*)	Creates a pair of values of types `T1` and `T2`, initialized with *val1* and *val2*
`pair<T1,T2>` *p*(*rv1*,*rv2*)	Creates a pair of values of types `T1` and `T2`, move initialized with *rv1* and *rv2*
`pair<T1,T2>` *p*(`piecewise_construct`, *t1*,*t2*)	Creates a pair of values of types `T1` and `T2`, initialized by the elements of the tuples *t1* and *t2*
`pair<T1,T2>` *p*(*p2*)	Copy constructor; creates p as copy of *p2*
`pair<T1,T2>` *p*(*rv*)	Move constructor; moves the contents of *rv* to p (implicit type conversions are possible)
p = *p2*	Assigns the values of *p2* to *p* (implicit type conversions are possible since C++11)
p = *rv*	Move assigns the values of *rv* to *p* (provided since C++11; implicit type conversions are possible)
p`.first`	Yields the first value inside the pair (direct member access)
p`.second`	Yields the second value inside the pair (direct member access)

`get<0>(p)`	Equivalent to `p.first` (since C++11)
`get<1>(p)`	Equivalent to `p.second` (since C++11)
`p1 == p2`	Returns whether *p1* is equal to *p2* (equivalent to `p1.first==p2.first && p1.second==p2.second`)
`p1 != p2`	Returns whether *p1* is not equal to *p2* (`!(p1==p2)`)
`p1 < p2`	Returns whether *p1* is less than *p2* (compares `first` or if equal `second` of both values)
`p1 > p2`	Returns whether *p1* is greater than *p2* (`p2<p1`)
`p1 <= p2`	Returns whether *p1* is less than or equal to *p2* (`!(p2<p1)`)
`p1 >= p2`	Returns whether *p1* is greater than or equal to *p2* (`!(p1<p2)`)
`p1.swap(p2)`	Swaps the data of *p1* and *p2* (since C++11)
`swap(p1,p2)`	Same (as global function) (since C++11)
`make_pair(val1,val2)`	Returns a pair with types and values of *val1* and *val2*

Element Access

To process the values of the `pair` direct access to the corresponding members is provided. In fact, the type is declared as `struct` instead of `class` so that all members are public:

```
namespace std {
    template <typename T1, typename T2>
    struct pair {
        // member
        T1 first;
        T2 second;
        ...
    };
}
```

For example, to implement a generic function template that writes a value pair to a stream, you have to program:[2]

Click here to view code image

```
// generic output operator for pairs (limited solution)
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm,
                           const std::pair<T1,T2>& p)
{
    return strm << "[" << p.first << "," << p.second << "]";
}
```

[2] Note that this output operator does not work where *ADL* (*argument-dependent lookup*) does not work (see Section 15.11.1, page 812, for details).

In addition, a tuple-like interface (see Section 5.1.2, page 68) is available since C++11. Thus, you can use `tuple_size<>::value` to yield the number of elements and `tuple_element<>::type` to yield the type of a specific element, and you can use `get()` to gain access to `first` or `second` :

Click here to view code image

```
typedef std::pair<int,float> IntFloatPair;
IntFloatPair p(42,3.14);

std::get<0>(p)                              // yields p.first
std::get<1>(p)                              // yields p.second
std::tuple_size<IntFloatPair>::value        // yields 2
std::tuple_element<0,IntFloatPair>::type    // yields int
```

Constructors and Assignment Operators

The default constructor creates a value pair with values that are initialized by the default constructor of their type. Because of language rules, an explicit call of a default constructor also initializes fundamental data types, such as `int` . Thus, the declaration

```
std::pair<int,float> p;    // initialize p.first and p.second with zero
```

initializes the values of `p` by using `int()` and `float()` , which yield zero in both cases. See Section 3.2.1, page 37, for a description of the rules for explicit initialization for fundamental types.

The copy constructor is provided with both versions for a pair of the same types and as member template, which is used when implicit type conversions are necessary. If the types match, the normal implicitly generated copy constructor is called.[3] For example:

Click here to view code image

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
...
void foo() {
    std::pair<int,const char*> p(42,"hello");
    f(p);    // OK: calls implicitly generated copy constructor
    g(p);    // OK: calls template constructor
}
```

[3] A template constructor does not hide the implicitly generated copy constructor. See Section 3.2, page 36, for more details about this topic.

Since C++11, a `pair<>` using a type that has only a nonconstant copy constructor will no longer compile:[4]

[4] Thanks to Daniel Krügler for pointing this out.

Click here to view code image

```
class A
{
  public:
    ...
    A(A&);   // copy constructor with nonconstant reference
    ...
};

std::pair<A,int> p;    // Error since C++11
```

Since C++11, the assignment operator is also provided as a member template so that implicit type conversions are possible. In addition, move semantics — moving the first and second elements — are supported.

Piecewise Construction

Class `pair<>` provides three constructors to initialize the `first` and `second` members with initial values:

```
namespace std {
   template <typename T1, typename T2>
   struct pair {
       ...
       pair(const T1& x, const T2& y);
       template<typename U, typename V> pair(U&& x, V&& y);
       template <typename... Args1, typename... Args2>
         pair(piecewise_construct_t,
              tuple<Args1...> first_args,
              tuple<Args2...> second_args);
       ...
   };
}
```

The first two of these constructors provide the usual behavior: passing one argument for `first` and one for `second`, including support of move semantics and implicit type conversions. However, the third constructor is something special. It allows passing two tuples — objects of a variable number of elements of different types (see Section 5.1.2, page 68) — but processes them in a different way. Normally, by passing one or two tuples, the first two constructors would allow initializing a pair, where `first` and/or `second` are tuples. But the third constructor uses the tuples to pass their *elements* to the constructors of `first` and `second`. To force this behavior, you have to pass `std::piecewise_construct` as an additional first argument. For example:

Click here to view code image

```
// util/pair1.cpp

#include <iostream>
#include <utility>
#include <tuple>
using namespace std;

class Foo {
  public:
    Foo (tuple<int, float>) {
        cout << "Foo::Foo(tuple)" << endl;
    }
    template <typename... Args>
    Foo (Args... args) {
        cout << "Foo::Foo(args...)" << endl;
    }
```

```
};

int main()
{
    // create tuple t:
    tuple<int,float> t(1,2.22);

    // pass the tuple as a whole to the constructor of Foo:
    pair<int,Foo> p1 (42, t);

    // pass the elements of the tuple to the constructor of Foo:
    pair<int,Foo> p2 (piecewise_construct, make_tuple(42), t);
}
```

The program has the following output:

```
Foo::Foo(tuple)
Foo::Foo(args...)
```

Only where `std::piecewise_construct` is passed as the first argument is class `Foo` forced to use a constructor that takes the *elements* of the tuple (an `int` and a `float`) rather than a tuple as a whole. This means that in this example, the varargs constructor of `Foo` is called. If provided, a constructor `Foo::Foo(int,float)` would be called.

As you can see, both arguments have to be a tuple to force this behavior. Therefore, the first argument, `42` , is explicitly converted into a tuple, using `make_tuple()` (you could instead pass `std::tuple(42)`).

Note that this form of initialization is required to `emplace()` a new element into an (unordered) map or multimap (see Section 7.8.2, page 341, and Section 7.9.3, page 373).

Convenience Function `make_pair()`

The `make_pair()` function template enables you to create a value pair without writing the types explicitly.[5] For example, instead of

[5] Using `make_pair()` should cost no runtime. The compiler should always optimize any implied overhead.

```
std::pair<int,char>(42,'@')
```

you can write the following:

```
std::make_pair(42,'@')
```

Before C++11, the function was simply declared and defined as follows:

Click here to view code image

```
namespace std {
    // create value pair only by providing the values
    template <template T1, template T2>
    pair<T1,T2> make_pair (const T1& x, const T2& y) {
        return pair<T1,T2>(x,y);
    }
}
```

However, since C++11, things have become more complicated because this class also deals with move semantics in a useful way. So, since C++11, the C++ standard library states that `make_pair()` is declared as:

```
namespace std {
    // create value pair only by providing the values
    template <template T1, template T2>
    pair<V1,V2> make_pair (T1&& x, T2&& y);
}
```

where the details of the returned values and their types `V1` and `V2` depend on the types of `x` and `y` . Without going into details, the standard now specifies that `make_pair()` uses move semantics if possible and copy semantics otherwise. In addition, it "decays" the arguments so that the expression `make_pair("a","xy")` yields a `pair<const char*,const char*>` instead of a `pair<const char[2],const char[3]>` (see Section 5.4.2, page 132).

The `make_pair()` function makes it convenient to pass two values of a pair directly to a function that requires a `pair` as its argument. Consider the following example:

Click here to view code image

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
```

```
...
void foo() {
    f(std::make_pair(42,"empty"));   // pass two values as pair
    g(std::make_pair(42,"chair"));   // pass two values as pair with type
conversions
}
```

As the example shows, `make_pair()` works even when the types do not match exactly, because the template constructor provides implicit type conversion. When you program by using maps or multimaps, you often need this ability (see Section 7.8.2, page 341).

Note that since C++11, you can, alternatively, use initializer lists:

```
f({42,"empty"});   // pass two values as pair
g({42,"chair"});   // pass two values as pair with type conversions
```

However, an expression that has the explicit type description has an advantage because the resulting type of the pair is not derived from the values. For example, the expression

```
std::pair<int,float>(42,7.77)
```

does *not* yield the same as

```
std::make_pair(42,7.77)
```

The latter creates a pair that has `double` as the type for the second value (unqualified floating literals have type `double`). The exact type may be important when overloaded functions or templates are used. These functions or templates might, for example, provide versions for both `float` and `double` to improve efficiency.

With the new semantics of C++11, you can influence the type `make_pair()` yields by forcing either move or reference semantics. For move semantics, you simply use `std::move()` to declare that the passed argument is no longer used:

```
std::string s, t;
...
auto p = std::make_pair(std::move(s),std::move(t));
... // s and t are no longer used
```

To force reference semantics, you have to use `ref()`, which forces a reference type, or `cref()`, which forces a constant reference type (both provided by `<functional>`; see Section 5.4.3, page 132). For example, in the following statements, a pair refers to an `int` twice so that, finally, `i` has the value `2`:

Click here to view code image

```
#include <utility>
#include <functional>
#include <iostream>

int i = 0;
auto p = std::make_pair(std::ref(i),std::ref(i));   // creates
pair<int&,int&>
++p.first;                                          // increments i
++p.second;                                         // increments i again
std::cout << "i: " << i << std::endl;               // prints i: 2
```

Since C++11, you can also use the `tie()` interface, defined in `<tuple>`, to extract values out of a pair:

Click here to view code image

```
#include <utility>
#include <tuple>
#include <iostream>

std::pair<char,char> p=std::make_pair('x','y');   // pair of two chars

char c;
std::tie(std::ignore,c) = p;   // extract second value into c (ignore first one)
```

In fact, here the pair `p` is assigned to a tuple, where the second value is a reference to `c` (see Section 5.1.2, page 70, for details).

Pair Comparisons

For the comparison of two pairs, the C++ standard library provides the usual comparison operators. Two value pairs are equal if both values are equal:

Click here to view code image

```
namespace std {
    template <typename T1, typename T2>
    bool operator== (const pair<T1,T2>& x, const pair<T1,T2>& y) {
        return x.first == y.first && x.second == y.second;
    }
}
```

In a comparison of pairs, the first value has higher priority. Thus, if the first values of two pairs differ, the result of their comparison is used as the result of the overall comparison of the pairs. If the members `first` are equal, the comparison of the members `second` yields the overall result:

Click here to view code image

```
namespace std {
   template <typename T1, typename T2>
   bool operator< (const pair<T1,T2>& x, const pair<T1,T2>& y) {
       return x.first < y.first ||
              (!(y.first < x.first) && x.second < y.second);
   }
}
```

The other comparison operators are defined accordingly.

Examples of Pair Usage

The C++ standard library uses pairs a lot. For example, the (unordered) map and multimap containers use `pair` as a type to manage their elements, which are key/value pairs. See Section 7.8, page 331, for a general description of maps and multimaps, and in particular Section 6.2.2, page 179, for an example that shows the usage of type `pair`.

Objects of type `pair` are also used inside the C++ standard library in functions that return two values (see Section 7.7.2, page 323, for an example).

5.1.2. Tuples

Tuples were introduced in TR1 to extend the concept of `pair`s to an arbitrary number of elements. That is, tuples represent a heterogeneous list of elements for which the types are specified or deduced at compile time.

However, with TR1 using the language features of C++98, it was not possible to define a template for a variable number of elements. For this reason, implementations had to specify all possible numbers of elements a tuple could have. The recommendation in TR1 was to support at least ten arguments, which meant that tuples were usually defined as follows, although some implementations did provide more template parameters:

Click here to view code image

```
template <typename T0 =  ..., typename T1 =  ..., typename T2 =  ...,
          typename T3 =  ..., typename T4 =  ..., typename T5 =  ...,
          typename T6 =  ..., typename T7 =  ..., typename T8 =  ...,
          typename T9 =  ...>
class tuple;
```

That is, class `tuple` has at least ten template parameters of different types, with an implementation-specific default value used to give unused tuple elements a default type with no abilities. This was in fact an emulation of variadic templates, which in practice, however, was quite cumbersome and very limited.

With C++11, variadic templates were introduced to enable templates to accept an arbitrary number of template arguments (see Section 3.1.9, page 26). As a consequence, the declaration for class `tuple`, which happens in `<tuple>`, is now reduced to the following:

```
namespace std {
  template <typename... Types>
  class tuple;
}
```

Tuple Operations

In principle, the tuple interface is very straightforward:

- You can create a tuple by declaring it either explicitly or implicitly with the convenience function `make_tuple()`.
- You can access elements with the `get<>()` function template.

Here is a basic example of this interface:

Click here to view code image

```
// util/tuple1.cpp

#include <tuple>
#include <iostream>
#include <complex>
```

```
#include <string>
using namespace std;

int main()
{
    // create a four-element tuple
    // - elements are initialized with default value (0 for fundamental types)
    tuple<string,int,int,complex<double>> t;

    // create and initialize a tuple explicitly
    tuple<int,float,string> t1(41,6.3,"nico");

    // "iterate" over elements:
    cout << get<0>(t1) << " ";
    cout << get<1>(t1) << " ";
    cout << get<2>(t1) << " ";
    cout << endl;

    // create tuple with make_tuple()
    // - auto declares t2 with type of right-hand side
    // - thus, type of t2 is tuple
    auto t2 = make_tuple(22,44,"nico");

    // assign second value in t2 to t1
    get<1>(t1) = get<1>(t2);

    // comparison and assignment
    // - including type conversion from tuple<int,int,const char*>
    //   to tuple<int,float,string>
    if (t1 < t2) {    // compares value for value
        t1 = t2;      // OK, assigns value for value
    }
}
```

The following statement creates a heterogeneous four-element tuple:

```
tuple<string,int,int,complex<double>> t;
```

The values are initialized with their default constructors. Fundamental types are initialized with `0` (this guarantee applies only since C++11).

The statement

```
tuple<int,float,string> t1(41,6.3,"nico");
```

creates and initializes a heterogeneous three-element tuple.

Alternatively, you can use `make_tuple()` to create a tuple in which the types are automatically derived from the initial values. For example, you can use the following to create and initialize a tuple of the corresponding types `int`, `int`, and `const char*`.[6]

[6] The type of `"nico"` is `const char[5]`, but it *decays* to `const char*` using the type trait `std::decay()` (see Section 5.4.2, page 132).

```
make_tuple(22,44,"nico")
```

Note that a tuple type can be a reference. For example:

```
string s;
tuple<string&> t(s);   // first element of tuple t refers to s

get<0>(t) = "hello";   // assigns "hello" to s
```

A tuple is no ordinary container class where you can iterate over the elements. Instead, for element access, member templates are provided so that you have to know the index of elements you want to access at compile time. For example, you get access to the first element of tuple `t1` as follows:

```
get<0>(t1)
```

Passing an index at runtime is not possible:

```
int i;
get<i>(t1)     // compile-time error: i is no compile-time value
```

The good news is that it is also a compile-time error to pass an invalid index:

```
get<3>(t1)      // compile-time error if t1 has only three elements
```

In addition, tuples provide the usual copy, assignment, and comparison operations. For all of them, implicit type conversions are possible (because member templates are used), but the number of elements must match. Tuples are equal if all elements are equal. To check whether a tuple is less than another tuple, a lexicographical comparison is done (see Section 11.5.4, page 548).

Table 5.2 lists all operations provided for tuples.

Table 5.2. Operations of `tuple`s

Operation	Effect
`tuple<T1,T2,...,Tn> t`	Creates a tuple with *n* elements of the specified types, initialized with their default constructors (0 for fundamental types)
`tuple<T1,T2,...,Tn> t(v1,v2,...,vn)`	Creates a tuple with *n* elements of the specified types, initialized with the specified values
`tuple<T1,T2> t(p)`	Creates a tuple with two elements of the specified type, initialized with the values of the passed `pair` *p* (*p*s types must match)
`t = t2`	Assigns the values of *t2* to *t*
`t = p`	Assigns a `pair` *p* to a tuple with two elements (the types of the pair *p* must match)
`t1 == t2`	Returns whether *t1* is equal to *t2* (`true` if a comparison with == of all elements yields `true`)
`t1 != t2`	Returns whether *t1* is not equal to *t2* `(!(t1==t2))`
`t1 < t2`	Returns whether *t1* is less than *t2* (uses lexicographical comparison)
`t1 > t2`	Returns whether *t1* is greater than *t2* `(t2<t1)`
`t1 <= t2`	Returns whether *t1* is less than or equal to *t2* `(!(t2<t1))`
`t1 >= t2`	Returns whether *t1* is greater than or equal to *t2* `(!(t1<t2))`
`t1.swap(t2)`	Swaps the data of *t1* and *t2* (since C++11)
`swap(t1,t2)`	Same (as global function) (since C++11)
`make_tuple(v1,v2,...)`	Creates a tuple with types and values of all passed values, and allows extracting values out of a tuple
`tie(ref1,ref2,...)`	Creates a tuple of references, which allows extracting (individual) values out of a tuple

Convenience Functions `make_tuple()` and `tie()`

The convenience function `make_tuple()` creates a tuple of values without explicitly specifying their types. For example, the expression

```
make_tuple(22,44,"nico")
```

creates and initializes a tuple of the corresponding types `int`, `int`, and `const char*`.

By using the special `reference_wrapper<>` function object and its convenience functions `ref()` and `cref()` (all available since C++11 in `<functional>`; see Section 5.4.3, page 132) you can influence the type that `make_tuple()` yields. For example, the following expression yields a tuple with a reference to variable/object x:

```
string s;
make_tuple(ref(s))     // yields type tuple<string&>, where the element refers to s
```

This can be important if you want to modify an existing value via a tuple:

Click here to view code image

```
std::string s;

auto x = std::make_tuple(s);          // x is of type tuple<string>
std::get<0>(x) = "my value";          // modifies x but not s

auto y = std::make_tuple(ref(s));     // y is of type tuple<string&>, thus y
refers to s
std::get<0>(y) = "my value";          // modifies s via y
```

By using references with `make_tuple()`, you can extract values of a tuple back to some other variables. Consider the following example:

Click here to view code image

```
std::tuple <int,float,std::string> t(77,1.1,"more light");
int i;
float f;
std::string s;
// assign values of t to i, f, and s:
std::make_tuple(std::ref(i),std::ref(f),std::ref(s)) = t;
```

To make the use of references in tuples even more convenient, the use of `tie()` creates a tuple of references:

Click here to view code image

```
std::tuple <int,float,std::string> t(77,1.1,"more light");
int i;
float f;
std::string s;
std::tie(i,f,s) = t;   // assigns values of t to i, f, and s
```

Here, `std::tie(i,f,s)` creates a tuple with references to `i`, `f`, and `s`, so the assignment of `t` assigns the elements in `t` to `i`, `f`, and `s`.

The use of `std::ignore` allows ignoring tuple elements while parsing with `tie()`. This can be used to extract tuple values partially:

Click here to view code image

```
std::tuple <int,float,std::string> t(77,1.1,"more light");
int i;
std::string s;
std::tie(i,std::ignore,s) = t;   // assigns first and third value of t to i and s
```

Tuples and Initializer Lists

The constructor taking a variable number of arguments to initialize a tuple is declared as `explicit`:

Click here to view code image

```
namespace std {
    template <typename... Types>
    class tuple {
      public:
        explicit tuple(const Types&...);
        template <typename... UTypes> explicit tuple(UTypes&&...);
        ...
    };
}
```

The reason is to avoid having single values implicitly converted into a tuple with one element:

Click here to view code image

```
template <typename... Args>
void foo (const std::tuple<Args...> t);

foo(42);                  // ERROR: explicit conversion to tuple<> required
foo(make_tuple(42));      // OK
```

This situation, however, has consequences when using initializer lists to define values of a tuple. For example, you can't use the assignment syntax to initialize a tuple because that is considered to be an implicit conversion:

Click here to view code image

```
std::tuple<int,double> t1(42,3.14);     // OK, old syntax
std::tuple<int,double> t2{42,3.14};     // OK, new syntax
std::tuple<int,double> t3 = {42,3.14};  // ERROR
```

In addition, you can't pass an initializer list where a tuple is expected:

```
std::vector<std::tuple<int,float>> v { {1,1.0}, {2,2.0} };   // ERROR

std::tuple<int,int,int> foo() {
    return { 1, 2, 3 };   // ERROR
}
```

Note that it works for `pair<>`s and containers (except `array<>`s):

```
std::vector<std::pair<int,float>>  v1 { {1,1.0}, {2,2.0} };   // OK
```

```
std::vector<std::vector<float>>    v2 { {1,1.0}, {2,2.0} };   // OK

std::vector<int> foo2() {
    return { 1, 2, 3 };   // OK
}
```

But for tuples, you have to explicitly convert the initial values into a tuple (for example, by using `make_tuple()`):

```
std::vector<std::tuple<int,float>> v { std::make_tuple(1,1.0),
                                        std::make_tuple(2,2.0) };   // OK

std::tuple<int,int,int> foo() {
    return std::make_tuple(1,2,3);   // OK
}
```

Additional Tuple Features

For tuples, some additional helpers are declared, especially to support generic programming:

- `tuple_size<`*tupletype*`>::value` yields the number of elements.
- `tuple_element<`*idx*,*tupletype*`>::type` yields the type of the element with index *`idx`* (this is the type `get()` returns).
- `tuple_cat()` concatenates multiple tuples into one tuple.

The following example shows the use of `tuple_size<>` and `tuple_element<>`:

```
typedef std::tuple<int,float,std::string> TupleType;

std::tuple_size<TupleType>::value        // yields 3
std::tuple_element<1,TupleType>::type    // yields float
```

You can use `tuple_cat()` to concatenate all forms of tuples, including `pair<>`s:

```
int n;
auto tt = std::tuple_cat (std::make_tuple(42,7.7,"hello"),
                          std::tie(n));
```

Here, `tt` becomes a tuple with all elements of the passed tuples, including the fact that the last element is a reference to `n`.

5.1.3. I/O for Tuples

The `tuple` class was first made public in the Boost library (see [Boost]). There, `tuple` had an interface to write values to output streams, but there is no support for this in the C++ standard library. With the following header file, you can print any tuple with the standard output operator `<<`:[7]

[7] Note that this output operator does not work where *ADL* (*argument-dependent lookup*) does not work (see Section 15.11.1, page 812, for details).

```
// util/printtuple.hpp

#include <tuple>
#include <iostream>

// helper: print elements with index IDX and higher of tuple t having MAX elements
template <int IDX, int MAX, typename... Args>
struct PRINT_TUPLE {
  static void print (std::ostream& strm, const std::tuple<Args...>& t) {
    strm << std::get<IDX>(t) << (IDX+1==MAX ? "" : ",");
    PRINT_TUPLE<IDX+1,MAX,Args...>::print(strm,t);
  }
};

// partial specialization to end the recursion
template <int MAX, typename... Args>
struct PRINT_TUPLE<MAX,MAX,Args...> {
  static void print (std::ostream& strm, const std::tuple<Args...>& t) {
  }
};

// output operator for tuples
template <typename... Args>
std::ostream& operator << (std::ostream& strm,
                           const std::tuple<Args...>& t)
{
```

```
        strm << "[";
        PRINT_TUPLE<0,sizeof...(Args),Args...>::print(strm,t);
        return strm << "]";
    }
```

This code makes heavy use of template metaprogramming to recursively iterate at compile time over the elements of a tuple. Each call of `PRINT_TUPLE<>::print()` prints one element and calls the same function for the next element. A partial specialization, where the current index `IDX` and the number of elements in the tuple `MAX` are equal, ends this recursion. For example, the program

```
// util/tuple2.cpp

#include "printtuple.hpp"
#include <tuple>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    tuple <int,float,string> t(77,1.1,"more light");
    cout << "io: " << t << endl;
}
```

has the following output:

```
io: [77,1.1,more light]
```

Here, the output expression

```
cout << t
```

calls

```
PRINT_TUPLE<0,3,Args...>::print(cout,t);
```

5.1.4. Conversions between tuples and pairs

As listed in Table 5.2 on page 71, you can initialize a two-element tuple with a `pair`. Also, you can assign a `pair` to a two-element tuple.

Note that `pair<>` provides a special constructor to use tuples to initialize its elements. See Section 5.1.1, page 63, for details. Note also that other types might provide a tuple-like interface. In fact, class `pair<>` (see Section 5.1.1, page 62) and class `array<>` (see Section 7.2.5, page 268) do.