

Proposal for adding tuple types into the standard library

Programming Language C++

Document no: N1403=02-0061

Jaakko Järvi

Indiana University

Pervasive Technology Laboratories

Bloomington, IN

jajarvi@cs.indiana.edu

November 8, 2002

1 Motivation

Tuple types exist in several programming languages, such as Haskell, ML, Python and Eiffel, to name a few. Tuples are fixed-size heterogeneous containers. They are a general-purpose utility, adding to the expressiveness of the language. Some examples of common uses for tuple types are:

- Return types for functions that need to have more than one return type.
- Grouping related types or objects (such as entries in parameter lists) into single entities.
- Simultaneous assignment of multiple values.

This proposal describes tuple types for C++. The standard library provides the `pair` template which is being used throughout the standard library, demonstrating the usefulness of tuple-like constructs. The proposed tuple type is basically a generalization of the `pair` template from two to an arbitrary number of elements. In addition to the features and functionality of pairs, the proposed tuple types:

- Support a wider range of element types (e.g. reference types).
- Support input from and output to streams, customizable with specific manipulators.
- Provide a mechanism for ‘unpacking’ tuple elements into separate variables.

2 Impact on the standard

All features described in this document can be implemented in a library, without requiring any core language changes. However, the implementation would benefit from a set of core language changes:

- Adding support for variable-length template argument lists.
- Making a reference to a reference to some type `T` equal to a reference to `T` (core language issue 106).
- Allowing default template arguments for function templates (core language issue 226).
- Ignoring cv-qualifiers that are added to function types (core language issue 295).
- Adding support for templated typedefs.

The proposal is written in such a way that it leaves room for a built-in tuple type, or for a tuple template with special support from the compiler, which we see as being worth considering. Compared to built-in tuple types in other languages, a library solution still falls short in some aspects. For instance, in the programming language Python, the function argument list is implicitly a tuple. This is a feature that cannot be added to C++ as a library, but would be most useful.¹

The concrete additions and changes to the standard are:

- A new section describing the requirements for the tuple template.
- Backwards-compatible changes to `pair` to allow pairs to act as tuples.
- A utility class and two utility function templates to be used in passing reference arguments through a pass-by-copy interface. These templates have uses outside of tuples we suggest then to be included in section 20.2 (Utility components).

We propose a new new standard header `<tuple>`. Operators for reading tuples from a stream and writing tuples to a stream introduce a dependency to `<istream>` and `<ostream>`, so it is a quality-of-implementation issue to not include definitions from these headers unnecessarily.

3 Tuples in a nutshell

The purpose of this section is to give an informal overview of the features that the tuple types provide. The feature set is largely based on the Boost Tuple Library [2, 3].

3.1 Defining tuple types

The `tuple` template can be instantiated with any number of arguments from 0 to some predefined upper limit. In the Boost Tuple library, this limit is 10. The argument types can be any valid C++ types. For example:

```
typedef tuple<A, const B, volatile C, const volatile D> t1;
typedef tuple<int, int&, const int&, const volatile int&> t2;
typedef tuple<void, int()(int)> t3;
```

Note that even types of which no objects can be created (cf. `void`, `int()(int)`), are valid tuple elements. Naturally, an object of a tuple type with such an element type cannot be constructed.

3.2 Constructing tuples

An n -element tuple has a default constructor, a constructor with n parameters, a copy constructor and a *converting copy constructor*. By converting copy constructor we refer to a constructor that can construct a tuple from another tuple, as long as the type of each element of the source tuple is convertible to the type of the corresponding element of the target tuple. The types of the elements restrict which constructors can be used:

- If an n -element tuple is constructed with a constructor taking 0 elements, all elements must be default constructible. For example:

```
tuple<int, float> a; // ok
class no_default_constructor { no_default_constructor(); };
tuple<int, no_default_constructor, float> b; // error
tuple<int, int&> c; // error, no default construction for references
```

¹Truly generic forwarding functions that could take any number of parameters would be supported. For example, one constructor definition in a derived class could cover a large set of base class constructors with different arities and argument types.

- If an n -element tuple is constructed with a constructor taking n elements, all elements must be copy constructible and convertible (default initializable) from the corresponding argument. For example:

```
tuple<int, const int, std::string>(1, 'a', "Hi")
tuple<int, std::string>(1, 2); // error
```

- If an n -element tuple is constructed with the converting copy constructor, each element type of the constructed tuple type must be convertible from the corresponding element type of the argument.

```
tuple<char, int, const char(&)[3]> t1('a', 1, "Hi");
tuple<int, float, std::string> t2 = t1; // ok
```

Construction works from `std::pair` as well. For example:

```
tuple<int, int> t3 = make_pair('a', 1); // ok
```

3.3 make_tuple

Tuples can also be constructed using the `make_tuple` (cf. `make_pair`) utility function templates. This makes the construction more convenient, saving the programmer from explicitly specifying the element types:

```
tuple<int, int, double> add_multiply_divide(int a, int b) {
    return make_tuple(a+b, a*b, double(a)/double(b));
}
```

By default, the element types are plain non-reference types. E.g., the `make_tuple` invocation below creates a tuple of type `tuple<A, B>`:

```
void foo(const A& a, B& b) {
    ...
    make_tuple(a, b);
    ...
}
```

This default behavior can be changed with to utility functions `ref` and `cref`. An argument wrapped with `ref` will cause the element type to be a reference to the argument type, and `cref` will similarly cause the element type to be a reference to the const argument type. For example:

```
A a; B b; const A ca = a;
make_tuple(cref(a), b);      // constructs tuple<const A&, B>(a, b)
make_tuple(ref(a), b);      // constructs tuple<A&, B>(a, b)
make_tuple(ref(a), cref(b)); // constructs tuple<A&, const B&>(a, b)
make_tuple(cref(ca));       // constructs tuple<const A&>(ca)
make_tuple(ref(ca));        // constructs tuple<const A&>(ca)
```

Note that `make_tuple` cannot be made to accept references to function types without the `ref` wrapper, unless core language issue 295 is resolved.

3.4 Assignment

The assignment operation is defined as element-wise assignment. Consequently, two tuples are assignable as long as they are element-wise assignable. For example:

```
tuple<char, int, const char(&)[3]> t1('a', 1, "Hi");
tuple<int, float, std::string> t2;
t2 = t1; // ok
```

Analogously to the converting copy constructor, assignment is defined from pairs as well.

3.5 The tie function templates

The `tie` functions are a short-hand notation for creating tuples where all element types are references. A `tie` call corresponds to an invocation of `make_tuple` where all arguments have been wrapped with `ref`. For example, the `tie` and `make_tuple` invocations below both return the same type of tuple object, namely `tuple<int&, char&, double&>`:

```
int i; char c; double d;
tie(i, c, d);
make_tuple(ref(i), ref(c), ref(d));
```

A tuple that contains non-const references as elements can be used to ‘unpack’ another tuple into variables. For example:

```
int i; char c; double d;
tie(i, c, d) = make_tuple(1, 'a', 5.5);
```

After the assignment, `i == 1`, `c == 'a'` and `d == 5.5`. A tuple unpacking operation like this is found, for example, in ML and Python. It is convenient when calling functions which return tuples.

3.5.1 Ignore

The library provides an object called `ignore` which allows one to ignore elements in an assignment to a tuple. Any assignment to `ignore` is a no-operation. For example:

```
char c;
tie(ignore, c) = make_tuple(1, 'a');
```

After this assignment, `c == 'a'`.

3.6 Number of elements

The number of elements in a tuple type is accessible as a compile-time constant:

```
tuple_size<tuple<int, int, int, int> >::value; // equals 4
```

3.7 Element type

The type of the Nth element of a tuple type is accessed using the `tuple_element` template:

```
tuple_element<2, tuple<int, char, float, double> >::type // float
```

Indexing is zero-based. The index must be an integral constant expression and using an index that is out of bounds results in a compile time error.

3.8 Element access

Let `t` be a tuple object. The expression `get<N>(t)` returns a reference to the Nth element of `t`, where `N` is an integral constant expression.

```
tuple<int, float, char>(1, 3.14, 'a') t;
get<2>(t); // equals 'a'
```

Indexing is zero-based. Using an index that is out of bounds results in a compilation error.

3.9 Relational operators

Tuples implement the operators `==`, `!=`, `<`, `>`, `<=` and `>=` using the corresponding operators on elements. This means that if any of these operators is defined between all elements of two tuples, the same operator is defined between the tuples as well.

The `operator==` is defined as the logical AND of the element-wise equality comparisons. The `operator!=` is defined as the logical OR of the element-wise inequality comparisons. The operators `<`, `>`, `<=` and `>=` each define a lexicographical ordering. An attempt to compare two tuples of different lengths results in a compile-time error. The comparison operators are “short-circuited”: elementary comparisons start from the first elements and are performed only until the result is known. Elements after that are not accessed. For example:

```
tuple<int, float, char> t(1, 2, 'a');
tuple<int, char, int> u(1, 1, 1000);
t < u; // ok, false

tuple<int, int, int, int> x;
tuple<int, int, int> y;
x < y; // error, different sizes

tuple<int, int, complex<double>, int> x;
tuple<int, int, string, int> y;
x < y; // error, no operator< between complex<double> and string
```

3.10 Input and output

The library overloads the streaming operators `<<` and `>>` for tuples. Output is implemented by invoking `operator<<` for each element, and input similarly with invocations of `operator>>`. When writing a tuple to a stream, opening and closing characters are written around the body of the tuple. Additionally, a delimiter character is written between each two consecutive elements. Similarly, the opening, closing and delimiter characters are expected to be present when extracting a tuple from an input stream. The default delimiter between the elements is a space, and the default opening and closing characters are the parentheses. For example:

```
cout << make_tuple(1, 'a', "C++");
```

outputs (1 a C++).

The library defines three formatting manipulators for tuples, `tuple_open`, `tuple_close` and `tuple_delimiter` to change, respectively, the opening, closing and delimiter characters for a particular stream. For example:

```
cout << tuple_open('[') << tuple_close(']')
    << tuple_delimiter(',')
    << make_tuple(1, 'a', "C++");
```

outputs the same tuple as: [1,a,C++].

Note that in general it is not guaranteed that a tuple written to a stream can be extracted back to a tuple of the same type, since the streamed tuple representation may not be unambiguously parseable. This is true, for instance, for tuples with `string` or C-style string element types.

3.11 Performance

Based on the experience with the Boost Tuple library, it is reasonable to expect an optimizing compiler to eliminate any extra cost of using tuples compared to using hand-written tuple-like classes. Inlining and copy propagation are the optimizations required to attain this goal.

Concretely, accessing tuple members should be as efficient as accessing a member variable of a class. Further, constructing a tuple should have no other cost than the cost of constructing the elements as separate objects. The same should be true for assignment.

Text in the standard

Text enclosed with brackets and typeset in sans serif is a comment, not proposed standard text [This is a comment].

4 Annex B: Implementation quantities

[Add to the list of implementation quantities (which specifies the recommended minima for implementation quantities).]

— Number of elements in one tuple type [10].

5 Tuple library

This clause describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. An implementation can set an upper limit for the number of arguments. The minimum value for this implementation quantity is defined in Annex B. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values.

Header <tuple> synopsis

```

template <class T1 = implementation-defined,
          class T2 = implementation-defined,
          ...,
          class TM = implementation-defined> class tuple;

template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator==(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator!=(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator<(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator<=(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator>(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator>=(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

template <class T> class tuple_size;

template <int I, class T> class tuple_element;

template <int I, class T1, class T2, ..., class TN>
RI get(tuple<T1, T2, ..., TN>&);

template <int I, class T1, class T2, ..., class TN>
PI get(const tuple<T1, T2, ..., TN>&);

template<class T1, class T2, ..., class TN>
tuple<V1, V2, ..., VN>
make_tuple(const T1&, const T2& , ..., const TN&);

template<class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN&> tie(T1&, T2& , ..., TN&);

template<class CharType, class CharTrait, class T1, class T2, ..., class TN>
basic_ostream<CharType, CharTrait>&
operator<<(basic_ostream<CharType, CharTrait>&, const tuple<T1, T2, ..., TN>&);

template<class CharType, class CharTrait, class T1, class T2, ..., class TN>
basic_istream<CharType, CharTrait>&
operator>>(basic_istream<CharType, CharTrait>&, tuple<T1, T2, ..., TN>&);

tuple_manip1 tuple_open(char_type c);
tuple_manip2 tuple_close(char_type c);
tuple_manip3 tuple_delimiter(char_type c);

```

Class template tuple

M is used to denote the implementation-defined number of template type parameters to the tuple class template, and N is used to denote the number of template arguments specified in an instantiation.

[*Example:* Given the instantiation `tuple<int, float, char>`, N is 3. *—end example*]

```
template <class T1 = implementation-defined,
         class T2 = implementation-defined,
         ...,
         class TM = implementation-defined> class tuple {
public:
    tuple();
    explicit tuple(P1, P2, ..., PN); // iff N > 0

    tuple(const tuple&);

    template <class U1, class U2, ..., class UN>
    tuple(const tuple<U1, U2, ..., UN>&);

    template <class U1, class U2>
    tuple(const pair<U1, U2>&);

    tuple& operator=(const tuple&);

    template <class U1, class U2, ..., class UN>
    tuple& operator=(const tuple<U1, U2, ..., UN>&);

    template <class U1, class U2>
    tuple& operator=(const pair<U1, U2>&);
};
```

Construction

```
tuple();
```

Requires: Each tuple element type T_i can be default constructed.

Effects: Default initializes each element.

```
tuple(P1, P2, ..., PN);
```

Where, if T_i is a reference type then P_i is T_i , otherwise P_i is `const  $T_i$ &`.

Requires: Each tuple element type T_i is copy constructible.

Effects: Copy initializes each element with the value of the corresponding parameter.

```
tuple(const tuple& u);
```

Requires: all types T_i shall be copy constructible.

Effects: Copy constructs each element of `*this` with the corresponding element of `u`.

```
template <class U1, class U2, ..., class UN>
tuple(const tuple<U1, U2, ..., UN>& u);
```

Requires: Each type T_i shall be constructible from the corresponding type U_i .

Effects: Constructs each element of `*this` with the corresponding element of `u`.

[In an implementation where one template definition serves for many different values for N, `enable_if` can be used to make the converting constructor and assignment operator exist only in the cases where the source

and target have the same number of elements. Another way of achieving this is adding an extra integral template parameter which defaults to N (more precisely, a metafunction that computes N), and then defining the converting copy constructor and assignment only for tuples where the extra parameter in the source is N.]

```
template <class U1, class U2> tuple(const pair<U1, U2>& u);
```

Requires: T1 shall be constructible from U1, T2 shall be constructible from U2. N == 2.

Effects: Constructs the first element with u.first and the second element with u.second.

```
tuple& operator=(const tuple& u);
```

Requires: All types Ti are assignable.

Effects: Assigns each element of u to the corresponding element of *this.

Returns: *this

```
template <class U1, class U2, ..., class UN>
tuple& operator=(const tuple<U1, U2, ..., UN>& u);
```

Requires: Each type Ti shall be assignable from the corresponding type Ui.

Effects: Assigns each element of u to the corresponding element of *this.

Returns: *this

```
template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);
```

Requires: T1 shall be assignable from U1, T2 shall be assignable from U2. N == 2.

Effects: Assigns u.first to the first element of *this and u.second to the second element of *this.

Returns: *this

[There seem to exist (rare) conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type T and if the parameter passed to the constructor is not of type T, but rather a tuple type that is convertible to T. The effect of the converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, it is possible to compare the 'nesting depths' of the source and target tuples and decide to select the element-wise constructor if the source nesting depth is smaller than the target nesting-depth. This can be accomplished using an enable_if template or other tools for constrained templates.]

Tuple creation functions

```
template<class T1, class T2, ..., class TN>
tuple<V1, V2, ..., VN>
make_tuple(const T1& t1, const T2& t2, ..., const TN& tn);
```

where Vi is X&, if the cv-unqualified type Ti is reference_wrapper<X>, otherwise Vi is Ti.

Returns: tuple<V1, V2, ..., VN>(t1, t2, ..., tn).

Notes: The make_tuple function template must be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.

[Example:

```
int i; float j;
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type

```
tuple<int, int&, const float&>
```

—end example]

```
template<class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN> tie(T1& t1, T2& t2, ..., TN& tn);
```

Returns: tuple<T1&, T2&, ..., TN>(t1, t2, ..., tn)

Notes: The tie function template must be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.

[Example:

tie functions allow one to create tuples that unpack tuples into variables. **ignore** can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++";
```

–end example]

Valid expressions for tuple types

```
tuple_size<T>::value
```

Requires: T is an instantiation of class template tuple.

Type: integral constant expression.

Value: Number of elements in T.

```
tuple_element<I, T>::type
```

Requires: $0 \leq I < \text{tuple_size}<T>::\text{value}$. The program is ill-formed if I is out of bounds.

Value: The type of the Ith element of T, where indexing is zero-based.

Element access

```
template <int I, class T1, class T2, ..., class TN>
RI get(tuple<T1, T2, ..., TN>& t);
```

Requires: $0 \leq I < N$. The program is ill-formed if I is out of bounds.

Return type: RI. If TI is a reference type, then RI is TI, otherwise RI is TI&.

Returns: A reference to the Ith element of t, where indexing is zero-based.

```
template <int I, class T1, class T2, ..., class TN>
PI get(const tuple<T1, T2, ..., TN>& t);
```

Requires: $0 \leq I < N$. The program is ill-formed if I is out of bounds.

Return type: PI. If TI is a reference type, then PI is TI, otherwise PI is const TI&.

Returns: A const reference to the Ith element of t, where indexing is zero-based.

[Constness is shallow. If TI is some reference type X&, the return type is X&, not const X&. However, if the element type is non-reference type T, the return type is const T&. This is consistent with how constness is defined to work for member variables of reference type.]

[Implementing get as a member function of tuple, would require using the template keyword in invocations where the type of the tuple object is dependent on a template parameter. For example: t.template get<1>();]

Equality and inequality comparisons

```
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator==(const tuple<T1, T2, ..., TM>& t,
               const tuple<U1, U2, ..., UM>& u);
```

Requires: tuple_size<tuple<T1, T2, ..., TM>>::value == tuple_size<tuple<U1, U2, ..., UM>>::value == N. For all i, where $0 \leq i < N$, get<i>(t) == get<i>(u) is a valid expression returning

a type that is convertible to `bool`.

Return type: `bool`

Returns: `true` iff `get<i>(t) == get<i>(u)` for all `i`. For any two zero-length tuples `e` and `f`, `e == f` returns `true`.

Effects: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

```
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator!=(const tuple<T1, T2, ..., TM>& t,
               const tuple<U1, U2, ..., UM>& u);
```

Requires: `tuple_size<tuple<T1, T2, ..., TM>>::value == tuple_size<tuple<U1, U2, ..., UM>>::value == N`. For all `i`, where `0 <= i < N`, `get<i>(t) != get<i>(u)` is a valid expression returning a type that is convertible to `bool`.

Return type: `bool`

Returns: `true` iff `get<i>(t) != get<i>(u)` for any `i`. For any two zero-length tuples `e` and `f`, `e != f` returns `false`.

Effects: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first inequality comparison that evaluates to `true`.

<, > comparisons

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
bool operator<(const tuple<T1, T2, ..., TN>&, const tuple<U1, U2, ..., UN>&);
```

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
bool operator>(const tuple<T1, T2, ..., TN>&, const tuple<U1, U2, ..., UN>&);
```

Requires: `tuple_size<tuple<T1, T2, ..., TM>>::value == tuple_size<tuple<U1, U2, ..., UM>>::value == N`. For all `i`, where `0 <= i < N`, `get<i>(t) ⊙ get<i>(u)` is a valid expression returning a type that is convertible to `bool`, where $\odot$ is either `<` or `>`.

Return type: `bool`

Returns: The result of a lexicographical comparison with $\odot$ between `t` and `u`, defined equivalently to:

`(bool)(get<0>(t) ⊙ get<0>(u)) || !((bool)(get<0>(u) ⊙ get<0>(t)) && ttail ⊙ utail),`

where `rtail` for some tuple `r` is a tuple containing all but the first element of `r`. For any two zero-length tuples `e` and `f`, `e ⊙ f` returns `false`.

<= and >= comparisons

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
bool operator<=(const tuple<T1, T2, ..., TN>&, const tuple<U1, U2, ..., UN>&);
```

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
bool operator>=(const tuple<T1, T2, ..., TN>&, const tuple<U1, U2, ..., UN>&);
```

Requires: `tuple_size<tuple<T1, T2, ..., TM>>::value == tuple_size<tuple<U1, U2, ..., UM>>::value == N`. For all `i`, where `0 <= i < N`, `get<i>(t) ⊙ get<i>(u)` is a valid expression returning a type that is convertible to `bool`, where $\odot$ is either `<=` or `>=`.

Returns: The result of a lexicographical comparison with $\odot$ between `t` and `u`, defined equivalently to:

`(bool)(get<0>(t) ⊙ get<0>(u)) && (!((bool)(get<0>(u) ⊙ get<0>(t)) || ttail ⊙ utail),`

where `rtail` for some tuple `r` is a tuple containing all but the first element of `r`. For any two zero-length tuples `e` and `f`, `e ⊙ f` returns `true`.

Notes: The above definitions for comparison operators do not impose the requirement that `ttail` (or `utail`) must be constructed. It may be even impossible, as `t` (or `u`) is not required to be copy constructible. Also, all comparison operators are short circuited to not perform element accesses beyond what is required to determine the result of the comparison.

Input and output

```
template<class CharType, class CharTrait, class T1, class T2, ..., class TN>
basic_ostream<CharType, CharTrait>&
operator<<(basic_ostream<CharType, CharTrait>& os,
          const tuple<T1, T2, ..., TN>& t);
```

Requires: For all $i = 0, 1, \dots, N-1$ in `os << get<i>(t)` is a valid expression.

Effects: Inserts `t` into `os` as `Lt0dt1d...dtnR`, where `L` is the opening, `d` the delimiter and `R` the closing character set by tuple formatting manipulators. Each element `ti` is output by invoking `os << get<i>(t)`. A zero-element tuple is output as `LR` and a one-element tuple is output as `Lt0R`.

Returns: `os`

```
template<class CharType, class CharTrait, class T1, class T2, ..., class TN>
basic_istream<CharType, CharTrait>&
operator>>(basic_istream<CharType, CharTrait>& is,
          tuple<T1, T2, ..., TN>& t);
```

Requires: For all $i = 0, 1, \dots, N-1$ in `is >> get<i>(t)` is a valid expression.

Effects: Extracts a tuple of the form `Lt0dt1d...dtnR`, where `L` is the opening, `d` the delimiter and `R` the closing character set by tuple formatting manipulators. Each element `ti` is extracted by invoking `is >> get<i>(t)`. A zero-element tuple expects to extract `LR` from the stream and one-element tuple expects to extract `Lt0R`. If bad input is encountered, calls `is.set_state(ios::failbit)` (which may throw `ios::failure` (27.4.4.3)).

Returns: `is`

Notes: It is not guaranteed that a tuple written to a stream can be extracted back to a tuple of the same type.

Tuple formatting manipulators

The library defines the following three stream manipulator functions. The types designated *tuple_manip1*, *tuple_manip2* and *tuple_manip3* are implementation-specified.

```
tuple_manip1 tuple_open(char_type c);
tuple_manip2 tuple_close(char_type c);
tuple_manip3 tuple_delimiter(char_type c);
```

Returns: Each of these functions returns an object `s` of unspecified type such that if `out` is an instance of `basic_ostream<charT,traits>`, `in` is an instance of `basic_istream<charT,traits>` and `char_type` equals `charT`, then the expression `out << s` (respectively `in >> s`) sets `c` to be the opening, closing, or delimiter character (depending on the manipulator function called) to be used when writing tuples into `out` (respectively extracting tuples from `in`).

Notes: Implementations are not required to support these manipulators for streams with `sizeof(charT) > sizeof(long)`; `out << s` and `in >> s` are required to fail at compile time if `out` and `in` are such streams and the implementation does not support tuple formatting manipulators for them.

[The constraint stated in the above **Notes** section allows an implementation where the delimiter characters are stored in space allocated by `xalloc`, which allocates an array of `longs`. A more general alternative is to store pointers to the delimiter characters in the `xalloc`-allocated array, and register a callback function (with `ios_base::register_callback`) for the stream to take care of deallocating the memory. If this approach is taken, the delimiters could be chosen to be strings instead of single characters. This might be worthwhile, such as to allow delimiters like `"`, `."`.]

6 Utility components

The library provides the class template `reference_wrapper` that stores a reference to an object in a Copy-Constructible wrapper, and two `reference_wrapper` construction functions that allow the user to express the intent in storing a reference instead of a copy.

```

template<typename T>
class reference_wrapper {
public:
    typedef T type;

    explicit reference_wrapper(T &);

    operator T& () const;
    T& get() const;
};

explicit reference_wrapper(T& t));

```

Postconditions: `this->get()` is equivalent to `t`.

Throws: will not throw.

```
operator T& () const;
```

Returns: `this->get()`

Throws: will not throw.

```
T& get() const;
```

Returns: the stored reference.

Throws: will not throw.

```
template<typename T> reference_wrapper<T> ref(T& t);
```

Returns: `reference_wrapper<T>(t)`

Throws: will not throw.

```
template<typename T> reference_wrapper<T const> cref(const T& t);
```

Returns: `reference_wrapper<T const>(t)`

Throws: will not throw.

The library provides the class `swallow_assign`:

```

struct swallow_assign {
    template <class T>
    swallow_assign& operator=(const T&) { return *this; }
};

```

The library provides an object `ignore` of type `swallow_assign`.

```
extern swallow_assign ignore;
```

7 Pairs

[Additions to `pair` to work with tuples]

```

template<class T1, class T2>
struct tuple_size<pair<T1, T2> > {
    static const int value = 2;
};

```

```

template<class T1, class T2>
struct tuple_element<0, pair<T1, T2> > {
    typedef T1 type;
};

template<class T1, class T2>
struct tuple_element<1, pair<T1, T2> > {
    typedef T2 type;
};

template<int I, class T1, class T2>
P& get(pair<T1, T2>&);

template<int I, class T1, class T2>
const P& get(const pair<T1, T2>&);

```

Return type: If I is 0 then P is T1, if I is 1 then P is T2, otherwise the program is ill-formed.

Returns: If I == 0 returns p.first, otherwise returns p.second.

8 Acknowledgements

The author is indebted to Jeremiah Willcock, Douglas Gregor and Gary Powell, as well as to Jeremy Siek and Dave Abrahams for their invaluable help in preparing this document. The Boost Tuple Library, the basis of this proposal, has benefited from suggestions by many in the Boost community, including Jens Maurer, William Kempf, Vesa Karvonen, John Max Skaller, Ed Brey, Beman Dawes, and Hartmut Kaiser.

References

- [1] The Boost Type Traits library. www.boost.org/libs/type_traits, 2002.
- [2] Jaakko Järvi. The Boost Tuple Library. www.boost.org/libs/tuple, 2001.
- [3] Jaakko Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 19:24–35, August 2001.
- [4] John Maddock. A Proposal to add Type Traits to the Standard Library. C++ Standards Committee Doc. no. J16/02-0003 = WG21/N1345, March 2002.
- [5] John Maddock and Steve Cleary. C++ type traits. *Dr. Dobbs's Journal*, October 2000.