# Proposal for adding tuple types into the standard library

Programming Language C++
Document no: N1382=02-0040

Jaakko Järvi
Indiana University
Pervasive Technology Laboratories
Bloomington, IN
*jajarvi@cs.indiana.edu*

September 10, 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.

There are at least a few different approaches for implementing tuples underneath a common interface.

For these reasons we do not suggest that a fixed implementation be standardized, but rather that the proposed text for the standard state the requirements for a tuple implementation in the form of valid expressions and the semantics of those expressions. Also, this approach 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 a most useful.[1]

In sum, this proposal defines the requirements for a standard tuple type. The requirements are stated so as to allow a library implementation with the current language, but also a future transition to a more feature-rich built-in tuple type, or to a library implementation that can take advantage of new language features, such as variable length template argument lists or templated typedefs.

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 and should thus be described elsewhere in the standard, possibly together with the *type traits* library [1, 5], which has been proposed for standardization [4].

[1] Truly 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.

We propose two new standard headers. The basic tuple definitions are to be included by including the `<tuple>` header. Operators for reading tuples from a stream and writing tuples to a stream introduce a dependency to `<istream>` and `<ostream>`, and so tuple input and output operators will be defined in a separate header `<tupleio>`.

# 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
  ```

- 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
  ```

  The argument to this constructor does not actually have to be of the standard tuple type, but can be any *tuple-like* type that acts like the standard tuple type, in the sense of providing the same element access interface. For example, `std::pair` is such a tuple-like type. 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)
```

Array arguments to `make_tuple` result in the corresponding tuple element being a reference to a const array. This is to avoid the need to wrap arrays with `cref`, as arrays cannot be copied as such anyway. 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, it suffices that the right-hand side of the assignment operator is a tuple-like object.

## 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 `N`th 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 `N`th 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:]

— 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. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values.

For certain tuple operations, the argument type does not have to a be tuple type. Instead, it suffices that the argument type is *tuple-like*. The following subclauses describe the requirements for tuple types and for types that are tuple-like. All tuple types are tuple-like.

#### Valid Expressions for all types

`is_tuple<T>::value`

**Type:** `static const bool` (integral constant expression).
**Value:** `true` if `T` is, or derives from, an instantiation of the `tuple` template. Users may not specialize this trait. An implementation may define this trait for any tuple-like type.

`is_tuple_like<T>::value`

**Type:** `static const bool` (integral constant expression).
**Value:** `true` if `T` is a tuple type or an instance of `pair`. Users may specialize this class template to indicate the conformance of a type to the requirements for a tuple-like type. Setting the value to `true` for a type that does not conform to all of the requirements for a tuple-like type causes undefined behavior.

`tuple_size<T>::value`

**Type:** `static const int` (integral constant expression).
**Value:** Number of elements in `T`. The number of elements in any non-tuple-like type is 1. The number of elements in a tuple-like type must be nonnegative.

`tuple_element<N, T>::type`

**Requires:** $0 \leq$ `N` $<$ `tuple_size<T>::value`
**Value:** The type of the `N`th element of `T`, where indexing is zero-based. If `T` is a non-tuple-like type and `N == 0`, the value is `T`. A diagnostic must be produced for a value of `N` that is out of bounds.

## 5.1 Tuple-like requirements

A type `P` is tuple-like if

- `is_tuple_like<P>::value == true`, and
- `tuple_size<P>::value` is a valid integral constant expression with a non-negative value, and
- `tuple_element<N, P>::type` is a valid expression that designates a type whenever `N` is in bounds, and
- `get<N>(p)`, where `p` is either of type `P` or `const P`, has the semantics defined in the *Element access* subclause.

## 5.2 Notation

| | |
|---|---|
| `T`, `U` | are tuple types |
| `t`, `u` | objects of types `T` and `U` |
| `tc`, `uc` | objects of types `const T` and `const U` |
| `P` | a tuple-like type |
| `p`, `pc` | objects of types `P` and `const P` |
| $\texttt{X}_i$ | the type of the $i$th element in `X`, where `X` is a tuple-like type |
| $\texttt{x}_i$ | the $i$th element of `x`, where `x` is of a tuple-like type |
| For all $i$ in `X` | For all indices $i$ from 0 to `tuple_size<X>::value - 1` for a tuple-like type `X`. |

The notation `T0, T1, ..., TN` stands for a comma separated list of types which may contain any number of elements from 0 to `N` + 1, where `N` + 1 $\leq$ maximum number of allowed tuple elements. The notation `t0, t1, ..., tn` is an analogous list of objects.

**Element access**

`get<N>(p)`

**Requires:** $0 \leq$ `N` $<$ `tuple_size<P>::value`
**Return type:** `tuple_element<N, P>::type&`
[Assuming core issue 106 is resolved to make a reference to a reference be just a reference instead of an error. Otherwise, the return type must be defined as: `add_reference<tuple_element<N, P>::type>::type`.]
**Returns:** A reference to the `N`th element of `p`, where indexing is zero-based.

`get<N>(pc)`

**Requires:** $0 \leq$ `N` $<$ `tuple_size<P>::value`
**Return type:** `const tuple_element<N, P>::type&`
[Assuming core issue 106 is resolved to make a reference to a reference be just a reference instead of an error. Otherwise, the return type must be defined as: `add_reference<const tuple_element<N, P>::type>::type` (see [4]).]
**Returns:** A reference to the `N`th element of `pc`, where indexing is zero-based.
**Notes:** Constness is shallow. If `element_type<N, P>::type` 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.]

[ There are alternative syntaces for element access. One particularly appealing syntax is something like: `t[index<N>]`, or even `t[_1]`, `t[_2]`, etc. We decided not to propose that syntax, because `operator[]` cannot be defined as a free function, and would thus prevent adding tuple-likeness into a type non-intrusively. ]

### Assignment

`t = pc`

**Requires:** `tuple_size<T>::value == tuple_size<P>::value`. For all $i$ in `T`, $\texttt{t}_i$ `=` $\texttt{pc}_i$ is a valid expression.
**Effects:** Performs $\texttt{t}_i$ `=` $\texttt{pc}_i$ for all $i$ in `T`.
**Return type:** `T&`
**Returns:** `t`

### Equality and inequality comparisons

`tc == uc`

**Requires:** `tuple_size<T>::value == tuple_size<U>::value`. For all $i$ in `T`, $\texttt{tc}_i$ `==` $\texttt{uc}_i$ is a valid expression returning a type that is convertible to `bool`.
**Return type:** `bool`
**Returns:** `true` iff $\texttt{tc}_i$ `==` $\texttt{uc}_i$ for all $i$ in `T`. 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`.

`t != u`

**Requires:** `T::size == U::size`. For all $i$ in `T`, $\texttt{t}_i$ `!=` $\texttt{u}_i$ is a valid expression returning a type that is convertible to `bool`.
**Return type:** `bool`
**Returns:** `true` iff $\texttt{t}_i$ `!=` $\texttt{u}_i$ for some $i$. 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`.

**<, >, <= and >= comparisons**

`t` $\odot$ `u`, where $\odot$ is either `<` or `>`

**Requires:** `T::size == U::size`. For all $i$ in `T`, $\texttt{t}_i$ $\odot$ $\texttt{u}_i$ and $\texttt{u}_i$ $\odot$ $\texttt{t}_i$ are valid expressions whose result types are convertible to `bool`.
**Return type:** `bool`
**Returns:** The result of a lexicographical comparison with $\odot$ between `t` and `u`, defined equivalently to:

`(bool)(`$\texttt{t}_0$ $\odot$ $\texttt{u}_0$`) || !((bool)(`$\texttt{u}_0$ $\odot$ $\texttt{t}_0$`)) && `$\texttt{t}_{\text{tail}}$ $\odot$ $\texttt{u}_{\text{tail}}$,

where $\texttt{r}_{\text{tail}}$ 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` $\odot$ `f` returns `false`.

`t` $\odot$ `u`, where $\odot$ is either `<=` or `>=`

**Requires:** `T::size == U::size`. For all $i$ in `T`, $\texttt{t}_i$ $\odot$ $\texttt{u}_i$ and $\texttt{u}_i$ $\odot$ $\texttt{t}_i$ are valid expressions whose result types are convertible to `bool`.
**Return type:** `bool`
**Returns:** The result of a lexicographical comparison with $\odot$ between `t` and `u`, defined equivalently to:

`(bool)(`$\texttt{t}_0$ $\odot$ $\texttt{u}_0$`) && (!((bool)(`$\texttt{u}_0$ $\odot$ $\texttt{t}_0$`)) || `$\texttt{t}_{\text{tail}}$ $\odot$ $\texttt{u}_{\text{tail}}$`)`,

where $\texttt{r}_{\text{tail}}$ 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` $\odot$ `f` returns `true`.

**Notes:** The above definitions for comparison operators do not impose the requirement that $\texttt{t}_{\text{tail}}$ (or $\texttt{u}_{\text{tail}}$) 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.

[ The comparison operators (as well as streaming operators defined below), are only defined for tuple types instead of tuple-like types. This is because it is somewhat tricky to overload operators to accept any tuple-like type and not to create ambiguities. A straightforward solution to provide comparison and streaming functionality for tuple-like types would be to define fully generic function templates, with names like `tuple_print` and `tuple_compare`, that implemented the same functionality as the comparison and streaming operators. ]

**Construction**

`tuple<T0, T1, ..., TN>()`

**Requires:** Each element type can be default constructed.
**Effects:** Default initializes each element.

`tuple<T0, T1, ... TN>(t0, t1, ..., tN)`

**Requires:** Each element type `Ti` is copy constructible. The type of each argument `ti` is convertible to `Ti`.

**Effects:** Copy initializes each element with the value of the corresponding parameter doing conversions as needed.

`tuple<T0, T1, ... TN>(pc)`

**Requires:** `tuple_size<P>::value == N+1`. For each $i$ in `P`, `P`$_i$ is convertible to `Ti`.
**Effects:** Copy initializes each element $i$ with `pc`$_i$.

[ There seem to exist (rare) conditions where the converting copy constructor and element-wise construction conflict. One example of this is if one is constructing a one-element tuple where the element type is another tuple type `T`. If the parameter passed to the constructor is not of type `T`, but rather a tuple type that is convertible to `T`, the conflict arises. It is possible to make this case fail, and provide another constructor, distinguished with an extra dummy parameter. Instead of

```
tuple<tuple<A> >(tuple<B>())
```

where `B` is convertible to `A`. The programmer would then have to write something like:

```
tuple<tuple<A> >(tuple<B>(), ignore)
```

]

### Input and output

In this section, a return type specified as `istream` or `ostream` is understood to mean an instance of `basic_istream` or `basic_ostream`, which is the type of the returned stream object, or from which this type derives.

`os << t`, where `os` is an instance of `basic_ostream`.

**Requires:** For all $i$ in `T`, `os << t`$_i$ is a valid expression.
**Effects:** Inserts `t` into `os` as $L$`t`$_0 d$`t`$_1 d \ldots d$`t`$_n R$, where $L$ is the opening, $d$ the delimiter and $R$ the closing character, set by tuple formatting manipulators. Each element `t`$_i$ is output by invoking `os << t`$_i$. A zero-element tuple is output as $LR$ and a one-element tuple is output as $L$`t`$_0 R$.
**Return type:** `ostream&`
**Returns:** `os`

`is >> t`, where `is` is an instance of `basic_istream`.

**Requires:** For all $i$ in `T`, `is >> t`$_i$ is a valid expression.
**Effects:** Extracts a tuple of the form $L$`t`$_0 d$`t`$_1 d \ldots d$`t`$_n R$, where $L$ is the opening, $d$ the delimiter and $R$ the closing character set by tuple formatting manipulators. Each element `t`$_i$ is extracted by invoking `is >> t`$_i$. A zero-element tuple expects to extract $LR$ from the stream and one-element tuple expects to extract $L$`t`$_0 R$.

If bad input is encountered, calls `is.set_state(ios::failbit)` (which may throw `ios::failure` (27.4.4.3).
**Return type:** `istream&`

**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 type designated *tuple_manip* is implementation-specified and may be different for each function.

*tuple_manip* `tuple_open(char_type c)`
*tuple_manip* `tuple_close(char_type c)`
*tuple_manip* `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` are `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 `", "`. ]

### Utility functions for tuple construction

The library provides the class template `any_holder` that can hold objects or references of any type. The observable behavior of `any_holder` must be as if implemented:

```
template <class T>
class any_holder {
  T data;
public:
  typedef T type;

  operator T() { return data; }
  T unwrap() { return data; }

  any_holder(const T& t) : data(t) {}
};
```

[If core issue 106 is not resolved to make a reference to a reference be just a reference instead of an error, the constructor parameter type must be defined as: `typename add_reference<typename add_const<T>::type>::type`] (see [4])

Reference types wrapped in `any_holder` can be passed by copy, and as const references, without affecting the constness of the wrapped object. Two template functions, `ref` and `cref`, are provided to create `any_holder` objects.

```
template <class T> inline any_holder<T&> ref(T& t);
```

**Returns:** `any_holder<T&>(t)`.

```
template <class T> inline any_holder<const T&> cref(const T& t);
```

**Requires:** `T` cannot be a function type.
[This requirement may be unnecessary depending on the resolution of core language issue 295]
**Returns:** `any_holder<const T&>(t)`.

```
template<class V0, class V1, ..., class VN>
... make_tuple(const V0& v0, const V1& v1, ..., const VN& vn);
```

**Return type:** `tuple<T0, T1, ..., TN>`, where

- if `Vi` is an array type, then `Ti` is a reference to `const Vi`.
- if the cv-unqualified type `Vi` is `any_holder<X>`, then `Ti` is `X`.
- otherwise `Ti` is `Vi`, with any const qualifications removed.

**Example:**

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

creates a tuple of type

```
tuple<int, const char (&) [4], int&, const float&>
```

**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. To construct a tuple which contains a reference to a function, the function reference must be wrapped inside `ref`.

```
tie(t0, t1, ..., tn)
```

**Effects:** As if implemented:

```
template<class T0, class T1, ..., class TN>
tuple<T0&, T1&, ..., TN&> tie(T0& t0, T1& t1, ..., TN& tn) {
  return tuple<T0&, T1&, ..., TN&>(t0, t1, ..., tn);
}
```

for each different number of arguments from 0 to the maximum number of allowed tuple elements.

The library provides the class `swallow_assign`, as if implemented:

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

The library provides an object `ignore` of type `swallow_assign`. It must be possible to use `ignore` in multiple translation units in a program.

**Example:** `tie` functions allow one to create tuples that unpack tuple-like objects 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++";
```

# 6 Pairs

[Wording to make pairs tuple-like]

Pairs are tuple-like types (see [number of the section where tuple-like is defined]). The following subclauses define the semantics of the expressions pair types must support to comply to the tuple-like requirements.

### Notation

| | |
|---|---|
| `P` | is an instance of the `pair` template |
| `p`, `pc` | are objects of types `P` and `const P` |

```
is_tuple_like<P>::value
```

**Type:** `static const bool` (integral constant expression).
**Value:** `true`.

```
tuple_size<P>::value
```

**Type:** `static const int` (integral constant expression).
**Value:** 2

```
tuple_element<0, P>::type
```

**Values:** `P::first_type`

`tuple_element<1, P>::type`
**Values:** `P::second_type`

`get<0>(p)`
**Return type:** `P::first_type&`
**Returns:** `p.first`

`get<1>(p)`
**Return type:** `P::second_type&`
**Returns:** `p.second`

`get<0>(pc)`
**Return type:** `const P::first_type&`
**Returns:** `pc.first`

`get<1>(pc)`
**Return type:** `const P::second_type&`
**Returns:** `p.second`

# 7 Acknowledgements

The author is indebted to Jeremiah Willcock for his suggestions and 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 Gary Powell, Douglas Gregor, Jens Maurer, Jeremy Siek, William Kempf, Vesa Karvonen, John Max Skaller, Ed Brey, Beman Dawes, David Abrahams 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. Dobb's Journal*, October 2000.