Chapter 22. Boost.Tuple

The library Boost.Tuple provides a class called `boost::tuple`, which is a generalized version of `std::pair`. While `std::pair` can only store exactly two values, `boost::tuple` lets you choose how many values to store.

The standard library has provided the class `std::tuple` since C++11. If you work with a development environment supporting C++11, you can ignore Boost.Tuple because `boost::tuple` and `std::tuple` are identical.

Example 22.1. `boost::tuple` replacing `std::pair`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int> animal;
  animal a{"cat", 4};
  std::cout << a << '\n';
}
```

To use `boost::tuple`, include the header file `boost/tuple/tuple.hpp`. To use tuples with streams, include the header file `boost/tuple/tuple_io.hpp`. Boost.Tuple doesn't provide a master header file that automatically includes all others.

`boost::tuple` is used in the same way `std::pair` is. In Example 22.1, a tuple containing one value of type `std::string` and one value of type `int` is created. This type is called `animal`, and it stores the name and the number of legs of an animal.

While the definition of type `animal` could have used `std::pair`, objects of type `boost::tuple` can be written to a stream. To do this you must include the header file `boost/tuple/tuple_io.hpp`, which provides the required operators. Example 22.1 displays `(cat 4)`.

Example 22.2. `boost::tuple` as the better `std::pair`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a{"cat", 4, true};
  std::cout << std::boolalpha << a << '\n';
}
```

Example 22.2 stores a name, the number of legs, and a flag that indicates whether the animal has a tail. All three values are placed in a tuple. When executed, this program displays `(cat 4 true)`.

You can create a tuple using the helper function `boost::make_tuple()`, which works like the helper function `std::make_pair()` for `std::pair` (see Example 22.3).

Example 22.3. Creating tuples with `boost::make_tuple()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <iostream>

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << boost::make_tuple("cat", 4, true) << '\n';
}
```

A tuple can also contain references, as shown in Example 22.4.

Example 22.4. Tuples with references

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <boost/ref.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "cat";
  std::cout.setf(std::ios::boolalpha);
  std::cout << boost::make_tuple(boost::ref(s), 4, true) << '\n';
}
```

The values 4 and `true` are passed by value and, thus, are stored directly inside the tuple, However, the first element is a reference to the string **`s`**. The function `boost::ref()` from Boost.Ref is used to create the reference. To create a constant reference, use `boost::cref()`.

Usually, you can use `std::ref()` from the C++11 standard library instead of `boost::ref()`. However, Example 22.4 uses `boost::ref()` because only Boost.Ref provides an operator to write to standard output.

`std::pair` uses the member variables **`first`** and **`second`** to provide access. Because a tuple does not have a fixed number of elements, access must be handled differently.

Example 22.5. Reading elements of a tuple

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a = boost::make_tuple("cat", 4, true);
  std::cout << a.get<0>() << '\n';
  std::cout << boost::get<0>(a) << '\n';
}
```

There are two ways to access values in a tuple. You can call the member function `get()`, or you can pass the tuple to the free-standing function `boost::get()`. In both cases, the index of the corresponding element in the tuple must be provided as a template parameter. Example 22.5 accesses the first element of the tuple **a** in both cases and, thus, displays `cat` twice.

Specifying an invalid index results in a compiler error because index validity is checked at compile time.

The member function `get()` and the free-standing function `boost::get()` both return a reference that allows you to change a value inside a tuple.

Example 22.6. Writing elements of a tuple

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a = boost::make_tuple("cat", 4, true);
  a.get<0>() = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

Example 22.6 modifies the animal's name and, thus, displays `(dog 4 true)`.

Boost.Tuple also defines comparison operators. To compare tuples, include the header file `boost/tuple/tuple_comparison.hpp`.

Example 22.7. Comparing tuples

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a1 = boost::make_tuple("cat", 4, true);
  animal a2 = boost::make_tuple("shark", 0, true);
  std::cout << std::boolalpha << (a1 != a2) << '\n';
}
```

Example 22.7 displays `true` because the tuples **a1** and **a2** are different.

The header file `boost/tuple/tuple_comparison.hpp` also contains definitions for other comparison operators such as greater-than, which performs a lexicographical comparison.

Boost.Tuple supports a specific form of tuples called *tier*. Tiers are tuples whose elements are all reference types. They can be constructed with the function `boost::tie()`.

Example 22.8. Creating a tier with `boost::tie()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, int&, bool&> animal;
  std::string name = "cat";
  int legs = 4;
  bool tail = true;
  animal a = boost::tie(name, legs, tail);
  name = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

Example 22.8 creates a tier **a**, which consists of references to the variables **name**, **legs**, and **tail**. When the variable **name** is modified, the tier is modified at the same time.

Example 22.8 could have also been written using `boost::make_tuple()` and `boost::ref()` (see Example 22.9).

Example 22.9. Creating a tier without `boost::tie()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, int&, bool&> animal;
  std::string name = "cat";
  int legs = 4;
  bool tail = true;
  animal a = boost::make_tuple(boost::ref(name), boost::ref(legs),
    boost::ref(tail));
  name = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

`boost::tie()` shortens the syntax. This function can also be used to unpack tuples. In Example 22.10, the individual values of the tuple, returned by a function, are instantly stored in variables.

Example 22.10. Unpacking return values of a function from a tuple

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

boost::tuple<std::string, int> new_cat()
{
  return boost::make_tuple("cat", 4);
}

int main()
{
  std::string name;
  int legs;
  boost::tie(name, legs) = new_cat();
  std::cout << name << ", " << legs << '\n';
}
```

`boost::tie()` stores the string “cat” and the number 4, both of which are returned as a tuple from `new_cat()`, in the variables **`name`** and **`legs`**.