# 【C++】Day85(2)

| | |
|---|---|
| Class | C++ |
| Date | @April 26, 2022 |
| Material | |
| Series Number | |
| Summary | Defining and Initializing tuples |

## 【Ch17】Specialized Library Facilities

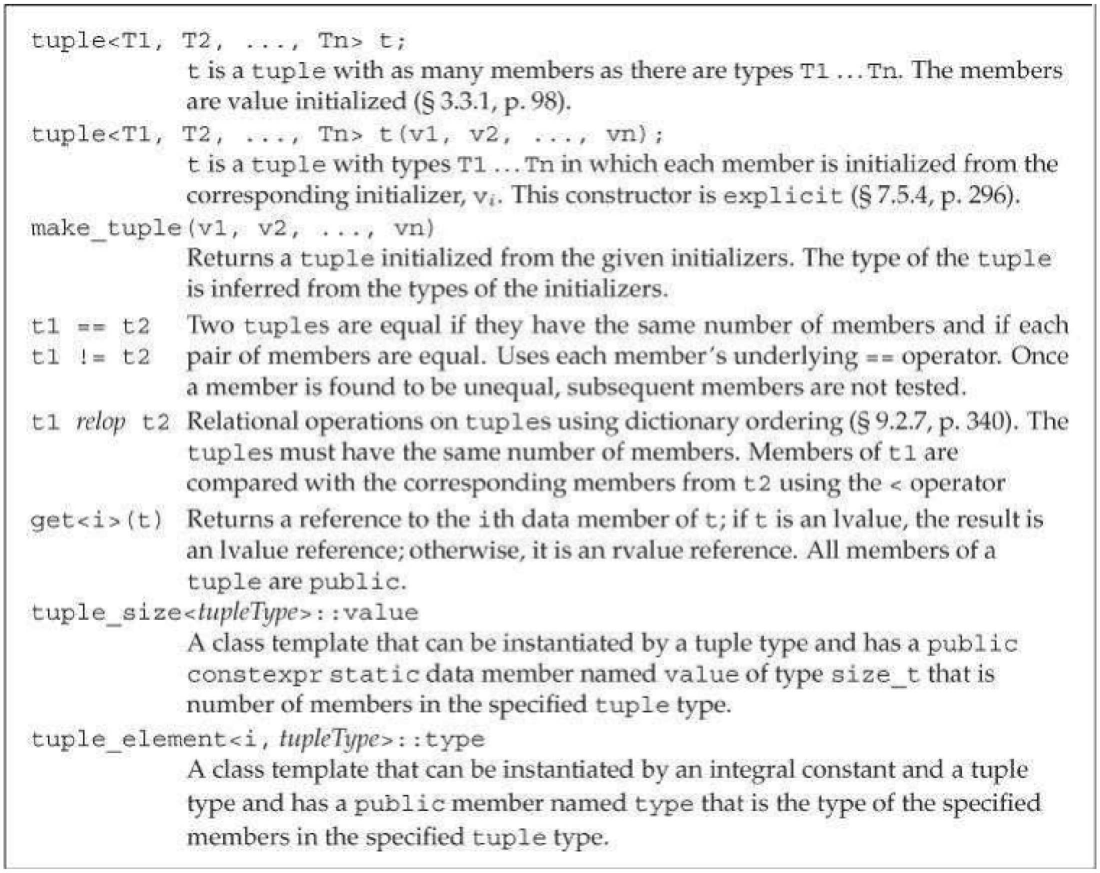
### 17.1 The tuple Type

A `tuple` is a template that is similar to a `pair`.

A `tuple` can have any number of members. Each distinct `tuple` type has a fixed number of members, but the number of members in one tuple type can differ from the number of members in another.

A `tuple` is most useful when we want to combine some data into a single object but do not want to bother to define a data structure to represent those data.

The following operations can be found in the `tuple` header.



| | |
|---|---|
| `tuple<T1, T2, ..., Tn> t;` | `t` is a `tuple` with as many members as there are types `T1`...`Tn`. The members are value initialized (§ 3.3.1, p. 98). |
| `tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);` | `t` is a `tuple` with types `T1`...`Tn` in which each member is initialized from the corresponding initializer, $v_i$. This constructor is `explicit` (§ 7.5.4, p. 296). |
| `make_tuple(v1, v2, ..., vn)` | Returns a `tuple` initialized from the given initializers. The type of the `tuple` is inferred from the types of the initializers. |
| `t1 == t2`<br>`t1 != t2` | Two `tuples` are equal if they have the same number of members and if each pair of members are equal. Uses each member's underlying `==` operator. Once a member is found to be unequal, subsequent members are not tested. |
| `t1` *relop* `t2` | Relational operations on `tuples` using dictionary ordering (§ 9.2.7, p. 340). The `tuples` must have the same number of members. Members of `t1` are compared with the corresponding members from `t2` using the `<` operator |
| `get<i>(t)` | Returns a reference to the `i`th data member of `t`; if `t` is an lvalue, the result is an lvalue reference; otherwise, it is an rvalue reference. All members of a `tuple` are `public`. |
| `tuple_size<`*tupleType*`>::value` | A class template that can be instantiated by a tuple type and has a `public constexpr static` data member named `value` of type `size_t` that is number of members in the specified `tuple` type. |
| `tuple_element<i,` *tupleType*`>::type` | A class template that can be instantiated by an integral constant and a tuple type and has a `public` member named `type` that is the type of the specified members in the specified `tuple` type. |

*Note: A tuple can be thought of as a "quick and dirty" data structure.*

### 17.1.1 Defining and Initializing tuples

When we define a `tuple`, we name the types of each of its members:

```
tuple<size_t, size_t, size_t> threeD; // All three members set to 0
tuple<string, vector<double> int, list<int>> someVal("constants", {3.14, 2.718}, 42, {0, 1, 2});
```

When we create a `tuple` object, we can use the default tuple constructor, which value initialize each member.

This tuple constructor is `explicit`, so we must use the direct initialization syntax.

```
tuple<size_t, size_t, size_t> threeD = {1, 2, 3}; // Error
```

The library defines a make_tuple function that generates a tuple object:

```
// tuple that represents a bookstore transaction: ISBN, count, price per book
auto item = make_tuple("0-999-8", 3, 20.00);
```

### *Accessing the Members of a tuple*

We access the members of a tuple through a library function template named `get`.

We pass a tuple object to get, which returns a reference to the specified member.

```
auto book_price = get<2>(item); // Returns the first member of item
book_price -= 5.0; // The book price is now 15.0
```

If we have a `tuple` whose precise type details we don’t know, we can use two auxilliary class templates to find the number and types of the `tuple`’s members:

```
typedef decltype(item) trans; // trans is the type of item
// Return the number of members in object's of type trans
size_t sz = tuple_size<trans>::value; // Returns 3
// cnt has the same type as the second member in item
tuple_element<1, trans>::type cnt = get<1>(item); // cnt is an int
```

### *Relational and Equality Operator*

We can compare two tuples only if they have the same number of members.

Moreoever, it must be legal to compare each pair of members using the `==` operator; to use the relational operators, it must be legal to use <. For example:

```
tuple<size_t, size_t> one(1, 2);
tuple<size_t, size_t> two;
bool ret = one < two; // OK: ret is false
```