

Tuples

Tuples extend the principles of a pair to a broader perspective. Find out more in this lesson.

WE'LL COVER THE FOLLOWING ^

- `std::make_tuple`
- `std::tie` and `std::ignore`

You can create tuples of arbitrary length and types with `std::tuple`. The class template needs the header `<tuple>`. `std::tuple` is a generalization of `std::pair`. You can convert between tuples with two elements and pairs. The tuple has, like his younger brother `std::pair`, a default, a copy, and a move constructor. You can swap tuples with the function `std::swap`.

The *i*-th element of a tuple *t* can be referenced by the function template `std::get`: `std::get<i-1>(t)`. By `std::get<type>(t)` you can directly refer to the element of the type `type`.

Tuples support the comparison operators `==`, `!=`, `<`, `>`, `<=` and `>=`. If you compare two tuples, the elements of the tuples will be compared lexicographically. The comparison starts at index 0.

`std::make_tuple`

The helper function `std::make_tuple` is quite convenient for the creation of tuples. You don't have to provide the types. The compiler automatically deduces them.

```
// tuple.cpp
#include <iostream>
#include <tuple>
using std::get;

int main(){
    std::tuple<std::string, int, float> tup1("first", 3, 4.17f);
    auto tup2= std::make_tuple("second", 4, 1.1);
```



```

std::cout << get<0>(tup1) << ", " << get<1>(tup1) << ", "
        << get<2>(tup1) << std::endl; // first, 3, 4.17

std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
        << get<2>(tup2) << std::endl; // second, 4, 1.1

std::cout << (tup1 < tup2) << std::endl; // true

get<0>(tup2)= "Second";

std::cout << get<0>(tup2) << "," << get<1>(tup2) << ","
        << get<2>(tup2) << std::endl; // Second, 4, 1.1

std::cout << (tup1 < tup2) << std::endl; // false

auto pair= std::make_pair(1, true);
std::tuple<int, bool> tup= pair;

return 0;
}

```



The helper function `std::make_tuple`

std::tie and std::ignore

[std::tie](#) enables you to create tuples, whose elements reference variables. With [std::ignore](#) you can explicitly ignore elements of the tuple.

```

// tupleTie.cpp
#include <iostream>
#include <tuple>
using namespace std;

int main(){
    int first= 1;
    int second= 2;
    int third= 3;
    int fourth= 4;

    cout << first << " " << second << " "
        << third << " " << fourth << endl; // 1 2 3 4

    auto tup= tie(first, second, third, fourth) // bind the tuple
        = std::make_tuple(101, 102, 103, 104); // create the tuple
        // and assign it
    cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
        << " " << get<3>(tup) << endl; // 101 102 103 104

    cout << first << " " << second << " " << third << " "
        << fourth << endl; // 101 102 103 104

    first= 201;
    get<1>(tup)= 202;

```



```
get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
    << " " << get<3>(tup) << endl;           // 201 202 103 104

cout << first << " " << second << " " << third << " "
    << fourth << endl;                       // 201 202 103 104

int a, b;
tie(std::ignore, a, std::ignore, b)= tup;
cout << a << " " << b << endl;              // 202 104

return 0;
}
```



The helper functions `std::tie` and `std::ignore`

Now, let's move on to reference wrappers.