Structured Bindings in C++

```
auto [ a, b, c ] = obj
```

 smitesh-tamboli

Smitesh Tamboli

Structured Bindings

Structured Bindings help to extract or destructure members of a structure, class, tuple, or pair more conveniently and concisely.

```
struct Product
{
    int product_id;
    string name;
    string description;
    float price;
    string image_url;
};
```

```
Product pObj{ 1,"Coffee", "This is Coffee",20.00f,"Coffee-mage-url" };
```

Without Structured Bindings

```
cout << "Product Id :" << pObj.product_id<<endl;
cout << "Name       :" << pObj.name << endl;
cout << "Description:" << pObj.description << endl;
cout << "Price      :" << pObj.price << endl;
cout << "Image URL  :" << pObj.image_url << endl;
```

structured bindings

```
auto [id, name, desc, price, url] = pObj;
```

We can directly use id, name, desc...

```
cout << "Product Id :" << id << endl;
cout << "Name       :" << name << endl;
cout << "Description:" << desc << endl;
cout << "Price      :" << price << endl;
cout << "Image URL  :" << url << endl;
```

Smitesh-tamboli

structured-bindigs

Syntax

```
auto [var1, var2, var3, . . . ] = expression;

auto [ var1, var2, var3 , . . . ] { expression };

auto [ var1, var2, var3, . . . ] ( expression );
```

How Structured Bindings Works

When the compiler sees structured bindings, it might create a temporary tuple-like object with member variables corresponding to the expression.

```
auto tuple_like_obj = expression;
using var1 = tuple_like_obj.first;
using var2 = tuple_like_obj.second;
```

The variables *var1* and *var2* are just other names for the *first*, and *second*. The *decltype(var1)* is the type of the member *first*, and *decltype(var2)* is the type of *second*. Also, as the compiler creates a temporary tuple-like object, we cannot access it by name directly like *tuple_like_obj.first*.

Binding

By default structured bindings perform value bindings.

```
struct Employee
{
    string name;
    int age;
};
```

```
Employee jhon {"Jhon", 23};
auto [name, age] = jhon;
```

The change in *name* will not change in *jhon.name* and vice versa.

```
jhon.name = "New Jhon";
cout<<name<<endl;  // Jhon
cout<<jhon.name<<endl;  //New Jhon

name = "Robin";
cout<<name<<endl;  // Robin
cout<<jhon.name<<endl;  // New Jhon
```

References in Structured Bindings

```
auto & [ var1, var2, var3, . . . ] = expression;
```

```
Employee jhon {"Jhon", 23};
auto & [name, age] = jhon;

jhon.name = "New Jhon";
cout<<name<<endl; //New Jhon
cout<<jhon.name<<endl; //New Jhon

name = "Robin";
cout<<name<<endl; // Robin
cout<<jhon.name<<endl;  // Robin
```

Here, ***name*** and ***age*** are reference bindings to ***jhon.name*** and ***jhon.age***. Hence, any changes in ***name*** or ***age***, reflect changes in ***jhon.name*** or ***jhon.age***.

Structured Bindings with nested struct/class

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

struct Address{
    string street;
    string city;
    string pincode;
};

struct Employee{
    string name;
    int age;
    Address address;
};

int main()
{
    // structured bindings of nested struct or class
    Employee e1 { "Jhon", 23, { "Jasper Avenue", "Edmonton", "T2Y 4E5"} };

    auto [name, age, address] = e1;
    auto [street, city, pincode] = address;

    cout<<name << endl <<age<< endl  <<street<< endl <<city<< endl <<pincode<<endl;

    return 0;
}
```

Structured Bindings with nested tuple or pair

```cpp
#include <iostream>
#include <tuple>
#include <string>
#include <utility>

using namespace std;

std::tuple<string, int, float> GetProductData()
{
    // process data...
    return tuple<string, int, float> ("Apple 15",12,1500.00f);
}


std::pair<string, int> GetData()
{
    // process data..
    return pair<string, int> ("Mango",20);
}

int main()
{

    auto [name, qty, price] = GetProductData();

    cout<<name<<endl;
    cout<<qty<<endl;
    cout<<price<<endl;

    auto [fruit, amount] = GetData();
    cout<<fruit<<endl;
    cout<<amount<<endl;

    return 0;
}
```