6. Custom Struct Keys

A notable limitation of the C++ STL's `std::unordered_map` and `std::unordered_set` is their inability to natively support user-defined types (such as a `struct` or `class`) as keys. An attempt to instantiate a map with a custom struct, as shown below, will result in a compile-time error.

```
// This declaration will fail to compile
std::unordered_map<Mahasiswa, int> nilaiUjian;
```

This failure occurs because the STL hash containers have two fundamental requirements for their key types, which C++ cannot automatically generate for a custom `struct`:

- **A Hashing Function**: The container must have a mechanism to convert a `Key` object (in this case, `Mahasiswa`) into a `size_t` value. This resulting "hash code" is what determines the bucket where the element will be stored.
- **An Equality Function**: The container must know how to compare two Key objects for equality. This is not for sorting, but specifically for handling hash collisions. The map must be able to determine if the key it's looking for is truly present.

The compiler cannot, for instance, assume whether two `Mahasiswa` objects are equal if only their `nim` matches, or if both `nim` and `nama` must match. To resolve this, the programmer must explicitly "teach" C++ how to perform these two operations for the custom type.

6.1. Defining the Equality Operation

The first requirement is met by **overloading the equality operator** (`operator==`) for the custom struct. This is the standard C++ mechanism for defining identity and is used directly by the `unordered_map` to compare keys within a bucket.

```
struct Mahasiswa {
    std::string nama;
    int nim;
};

// This function defines what makes two Mahasiswa objects
// "equal" in the eyes of the hash map.
bool operator==(const Mahasiswa& a, const Mahasiswa& b) {
    return a.nama == b.nama && a.nim == b.nim;
```

```
}
```

6.2. Providing a Hashing Function via `std::hash` Specialization

The second requirement is met by providing a custom hash function. The standard mechanism for this is to create a **template specialization** of the `std::hash` struct, which resides in the std namespace.

This specialization "injects" our custom hashing logic into the STL, allowing `unordered_map` to find and use it automatically whenever it needs to hash a `Mahasiswa` object. This is achieved by defining `operator()` within the specialized struct.

```
#include <functional> // Required for std::hash

// We "inject" our hashing logic into the std namespace
namespace std {

// We declare a specialization of the 'hash' template for our type
template<>
struct hash<Mahasiswa> {

    // operator() is what the unordered_map will call.
    // It must be 'const' and return a 'size_t'.
    size_t operator()(const Mahasiswa& m) const {

        // Get the hash for each member individually.
        size_t hashNama = std::hash<std::string>{}(m.nama);
        size_t hashNim = std::hash<int>{}(m.nim);

        // Combine the individual hashes into one final hash.
        return hashNama ^ (hashNim << 1);
    }
};

} // End of namespace std
```

6.3. Usage Example

With both the equality operator and the `std::hash` specialization provided, the `std::unordered_map` now has all the components necessary to manage the Mahasiswa key. The following code will now compile and function as expected

```
int main() {
    std::unordered_map<Mahasiswa, int> nilaiUjian;

    Mahasiswa budi = {"Budi", 12345};
    nilaiUjian[budi] = 90;

    // The map can now hash "budi" to find a bucket
    // and use operator== to check for collisions.
}
```

Revision #2
Created 2025-11-04 15:28:03 UTC by RE
Updated 2025-11-04 15:53:33 UTC by RE