0 structs and classes

Nur als Info, nicht ultra relevant für euch

179

It returns true for both classes and structs. I know that in C++ they are almost the same thing, but I'd like to know why there's not a distinction between them in the type trait.

Unfortunately this is a common misconception in C++. Sometimes it comes from fundamental misunderstanding, but at other times it comes from an ambiguity in English. It can come from inaccurate compiler diagnostics, badly-written books, incorrect SO answers...

You've probably read something like this:

"There is no difference in C++ between a struct and a class except the default visibility of members and bases."

This passage can be interpreted in a sense that is misleading, because the notions of *identity* and *equality* are hard to distinguish when using phrases like "no difference".

In fact, C++ has not had structs since 1985. It only has classes.

The kind of types that you declare with the keyword `class` and the keyword `struct` are *classes*. Period. The keyword `struct`, and the visibility rules that are the default when defining a class using that keyword, were kept only for backward compatibility with C ... but that's a syntax thing. It doesn't make the resulting types actually be of a different kind.

The type trait makes no distinction because there literally isn't one to make.

https://stackoverflow.com/questions/54585/when-should-you-use-a-class-vs-a-struct-in-c

→ rule of thumb:

↳ structs zum mehrere Datentypen gruppieren; zu einem neuen Typ zusammen zu fassen.
→ eher keine Memberfkt

↳ classes für "richtiges" OOP mit Memberfkt etc

=> ABER beides macht das Gleiche

Einziger Unterschied gegenüber structs: Member in structs sind per default öffentlich (public).

→ backward compatability mit C

C: 1972

C++: 1985

. operator greift auf Member zu

=> Zugriffsrechte werden durch public/privat Regionen geregelt.

```
class my_class {
public: // public section
   double some_public_member;
private: // private section
   double some_private_member;
};
...
my_class inst;
inst.some_public_member = 1.0;
inst.some_private_member = 0.0; // ERROR
```

OK: Ihr greift auf ein public member zu (public heisst ihr dürft von ausserhalb der Class darauf zugreifen)

Geht nicht! privat member → kein Zugriff von ausserhalb der class.

hier wird ein neues Objekt vom Typ my_class und mit Namen "inst"

man sagt auch eine Instanz des Typs my_class

=> Entweder man möchte gar nicht, dass ein member sichtbar ist oder man implementiert getter/setter Fkt o.Ä.

e.g.
```
get_private_member(){
    return private_member;
}
```

→ member functions

Aus Summary (10):

"Memberfunktionen stellen Funktionalität auf einer Klasse bereit. Sie ermöglichen den kontrollierten Zugang zu den privaten Daten und priva- ten Memberfunktionen. Die Deklaration einer Memberfunktion erfolgt immer in der Klassendefinition, die Definition der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Klasse explizit erwähnt werden mittels der ::-Schreibweise.
Der Aufruf einer Memberfunktion ist obj.mem_func(arg1, arg2, ..., argN). Der Teil obj. kann weggelassen werden, falls aus der Class heraus auf einen Member des aufrufenden Objekts zugegriffen wird."

→ Deklaration ≙ "Hey Compiler, diese ... Fkt existiert"

e.g. `void set_private_member ( int a);`

→ Definition ≙ tatsächliche Implementation der Fkt i.e. "was sie macht"

e.g.
```
void set_private_member (int a ){
        private_member = a ;
}
```

⇒ Definition ausserhalb der class: wir müssen dem Compiler sagen "wo die Fkt" lebt resp. wo sie existiert.

ungefähres schema:

```
return-type    class_name :: function_name ( args ){
    // stuff
}
```

→ const member functions

Das const bezieht sich auf *this. Es verspricht, dass durch die Funktions- ausführung das implizite Argument nicht im Wert verändert wird.

siehe pointers

⇒ Innerhalb einer Memberfkt (d.h. wir sind innerhalb der Class) haben wir immer Zugriff auf einen Pointer der auf das Objekt zeigt mit/auf der die Memberfkt aufgerufen wurde.

e.g. `Insurance my_ins;` erstellt uns ein Objekt/ Instanz von Typ "Insurance" und mit Namen "my_ins".

⇒ wenn wir jetzt `my_ins.get_value()` aufrufen, verspricht uns das const, dass die Instanz mit/auf der die Memberfkt `get_value()` aufgerufen wurde nicht verändert wird d.h. "my_ins" wird nicht verändert.

(Das const macht dann aus dem *this pointer einen const ptr d.h. ein ptr, der auf etwas zeigt, was nicht verändert werden kann.)