ToiletPaper #94

jambit WHERE INNOVATION WORKS

Strong Typedefs

Autor: Hannes Lerchl / Senior Software Architect / Business Division New Business

✕ Problem

Viele Funktionen/Methoden, die einen Integer als Parameter nehmen, knüpfen an diesen Integer ganz besondere Erwartungen. Das kann sein: Ein Index in einer bestimmten Struktur, die ID von irgendwas, eine bestimmte physikalische Maßeinheit oder irgendwas aus Schokolade. Das Problem ist, dass f(`int x`) nicht sagt, "welchen Sinn" `int x` hat. Die möglichen Konsequenzen reichen von "merkt eh keiner" oder "Absturz/Neustart" bis zu "serious Autsch" (1).
In C++ gibt es das Schlüsselwort `typedef`, aber letztendlich ist es nicht viel mehr als `alias` in der bash, hilft also hier nicht weiter. Kommt jemand mit "ne eigene Klasse" um die Ecke, ist das vielen "zu klobig" und außerdem ... die Performance!!

✓ Lösung

C++ erlaubt es mit seinem Template-System eigene Datentypen zu erschaffen, deren Verwendung vom Compiler überprüft werden kann (z.B. keine "impliziten casts" (2)), die sich aber zur Laufzeit weitgehend wie simple `int`s verhalten. Im einfachsten Fall dient das dazu, verschiedene Arten von Indizes auseinander zu halten (3); das kann aber auch erweitert werden, um typsichere Berechnungen mit SI-Einheiten durchzuführen (4) (wenn's sein muss auch "englisch imperiale" Einheiten). Der Compiler baut dabei den nötigen Code ein, um `millirad` in `deg` oder `miles/h` in `m/s` umzurechnen (und hätte so den Mars-Orbiter gerettet).

➔ Beispiel

Die grundlegende Idee ist, ein `struct` (oder der Sichtbarkeit wegen eine `class`) mit einem einzigen Wert als Member zu erstellen. Dazu gibt es noch alle nötigen Operatoren, die man für die Handhabung braucht.
Ausgangspunkt ist folgender "unauffälliger" Code. Es fehlen noch die Deklarationen für `PersonId` und `RoomId`.

```
class Room
{
  public:
    virtual ~Room();
    virtual int get_number() const;
};
class Person { /* like Room but for persons */ };

class Resources
{
  public:
    virtual ~Resources();
    virtual const Room& get_room(RoomId id) const = 0;
    virtual const Person& get_person(PersonId id) const = 0;
};

int do_something(PersonId id, const Resources& rsc)
{
      return rsc.get_room(id).get_number();
}
```

Der Weg über `typedef` wird compilieren ...
und was anderes tun, als gedacht war

```
typedef unsigned int PersonId;
typedef unsigned int RoomId;
```

Beim gleichen Code mit `PersonId` als Klasse wird der Compiler bei `rsc.get_room(id)` aussteigen und jammern

```
class PersonId
{
  public:
    explicit PersonId(const unsigned int t_) : t(t_) {};
    PersonId() : t(0) {}
    PersonId next() const { return PersonId(t + 1); }
    unsigned int value() const { return t; }

  private:
    unsigned int t;
};
// same for RoomId;
```

Das funktioniert für kleine Anwendungsfälle ganz gut. Um aber sowohl den Linker ruhig zu halten, als auch den Methodenaufruf wegzuoptimieren, wird aus dieser `class/struct` noch ein `template` gemacht. Damit sind wir bei `BOOST_STRONG_TYPEDEF` angekommen (3).
Das wirkt auf den ersten Blick recht umständlich, rettet einen aber (ohne große Laufzeit-Strafen) vor "falsch verdrahteten" Integern. Wer schon mal das komplette System nach einem off-by-one Bug durchsucht hat, der weiß Bescheid.

✚ Weiterführende Aspekte

(1) Die NASA verlor ihren 125 Mio. $ teuren Mars Climate Orbiter, weil die Ingenieure vergaßen von englischen auf metrische Einheiten umzuwandeln: http://articles.latimes.com/1999/oct/01/news/mn-17288

(2) 1996 explodierte eine Ariane 5 mitsamt ihrer Nutzlast (im Wert von ca 500 Mio $ Dollar) nachdem ein 64 Bit double in eine Funktion gesteckt wurde, die einen 16 Bit signed int erwartet. Der Wert war zu groß für die 16 Bit. http://www-users.math.umn.edu/~arnold/disasters/ariane.html

(3) Siehe BOOST_STRONG_TYPEDEF https://www.boost.org/doc/libs/1_67_0/boost/serialization/strong_typedef.hpp

(4) Siehe boost::units https://www.boost.org/doc/libs/1_67_0/libs/units/example/kitchen_sink.cpp oder nholthaus/units https://github.com/nholthaus/units#documentation