# ToiletPaper #94

jambit
WHERE INNOVATION WORKS

## Strong Typedefs

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

### ✕ Problem

Many functions/methods that use an integer as a parameter have very special expectations regarding this integer. This could be an index in a certain structure, the ID of something, a certain physical unit or something made of chocolate. The problem is that f(`int x`) does not say "what meaning" `int x` has. The possible consequences range from "nobody will notice anyway" or "crash/restart" to "serious Ouch" (1).
In C++ there is the keyword `typedef`, but in the end it's not much more than `alias` in bash, so it doesn't help here. If someone comes along with "an extra class", many will find it "too bulky" and above all ... the performance!

### ✓ Solution

With its template system, C++ allows to create custom data types, whose usage can be checked by the compiler (e.g. no "implicit casts" (2)), which nevertheless behave like simple `ints` at runtime. In the simplest case, this helps to keep different types of indices apart (3); but it can also be extended to perform type-safe calculations with SI units (4) (even “English imperial” units if necessary). The compiler includes the necessary code to convert `millirad` to `deg` or `miles/h` to `m/s` (and this way would’ve saved the Mars orbiter).

### ➔ Example

The basic idea is to create a `struct` (or – for the sake of visibility – a `class`) with a single value as member. Additionally, it gets all the necessary operators required for its handling.
Starting point is the following "inconspicuous" code. The declarations for `PersonId` and `RoomId` are still missing.

```
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();
}
```

Using a `typedef` will compile … and do something different than intended …

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

Compiling the same code with `PersonId` as class, the compiler will complain and quit at `rsc.get_room(id)`

```
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;
```

This works quite well for small applications. But to keep the linker quiet and to optimize the method call, a `template` is made from this `class/struct`. This brings us to `BOOST_STRONG_TYPEDEF` (3).
At first glance, this seems quite cumbersome, but it saves you from "falsely wired" integers (without any bigger runtime penalties). Anyone who has ever searched the entire system for an off-by-one bug will value this aid.

### ✚ Further Aspects

(1) NASA lost its $125-million Mars Climate Orbiter because spacecraft engineers failed to convert from English to metric measurements: http://articles.latimes.com/1999/oct/01/news/mn-17288

(2) In 1996, an Ariane 5 exploded with its payload (worth about $500 million dollars) after a 64-bit double was put into a function that expected a 16-bit signed int. The value was too large for the 16 bits. http://www-users.math.umn.edu/~arnold/disasters/ariane.html

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

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