C++ Bit-Access

Detlef Wilkening

10.03.2016

Einfache Aufgabe:

- **Klasse, die ein Int kapselt**
  - Genau genommen “hier-und-heute” einen “uint32_t”
  - Mit Templates könnte man das natürlich verallgemeinern
    - Aber das ist heute nicht das Thema
- **Und einen lesenden und schreibenen Bit-Zugriff mit [] unterstützt**
  - Dabei ist als Schnittstellen-Typ “bool” gewünscht
- **Und dabei natürlich “const” korrekt unterstützt**
  - Daher schreibende Zugriffe bei Const-Objekten erzeugen Compiler-Fehler
  - Lesende Zugriffe funktionieren natürlich auch bei Const-Objekten
- **Und eine Ausgabe-Funktion und „str()“ Funktion hat**
  - Ausgabe-Format siehe Assertions nächste Folien

- **Anwendungs-Beispiele auf den folgenden beiden Folien**

```
Integer x(0b101);
assert(x.str()=="5 -> 0000:0000:0000:0000:0000:0000:0000:0101");
assert(x[0]==true);
assert(x[1]==false);
assert(x[2]==true);
assert(x[3]==false);
assert(x[4]==false);

x[0] = false;
x[1] = true;
assert(x.str()=="6 -> 0000:0000:0000:0000:0000:0000:0000:0110");
assert(x[0]==false);
assert(x[1]==true);
assert(x[2]==true);
assert(x[3]==false);
assert(x[4]==false);
```

```
const Integer cx(0b1001);
assert(cx.str()=="9 -> 0000:0000:0000:0000:0000:0000:0000:1001");
assert(cx[0]==true);
assert(cx[1]==false);
assert(cx[2]==false);
assert(cx[3]==true);
assert(cx[4]==false);

cx[0]=false;                => Compiler-Fehler
```

Fangen wir mit den einfachen Dingen an:

- **Klasse “Integer”**
  - Klasse mit einem „uint32_t“ Attribut
  - Konstruktor mit Default-Wert „0“
  - Ausgabe Operator << für alle „ostreams“
    - Abbildung auf Funktion „print“
      - Könnte man „private“ machen und den Operator zum „friend“
      - Heute aber nicht
  - Abbildung „str()“ auf „operator <<“
  - Konstruktor und so könnten auch „constexpr“ sein
    - Aber machen wir heute nicht
    - Heute nicht unser Thema

```
class Integer
{
public:
   explicit Integer(uint32_t value = 0) : mValue(value) {}

   ostream& print(ostream& out) const;

   string str() const;

private:
   uint32_t mValue;
};

inline ostream& operator<<(ostream& out, const Integer& arg)
{
   return arg.print(out);
}
```

```
ostream& Integer::print(ostream& out) const
{
   out << mValue << " -> ";
   uint32_t mask = uint32_t(0x80000000);
   for (int i=1; ; ++i)
   {
      out << (mask & mValue ? '1' : '0');
      if (i==32) break;
      if (i%4==0) out << ':';
      mask >>= 1;
   }
   return out;
}
```

```
string Integer::str() const
{
   ostringstream oss;
   oss << (*this);
   return oss.str();
}
```

Kommen wir zum ersten interessanten Thema:

- **Lesender Bit-Zugriff mit dem Operator [ ]**

```
Integer x(0b101);
bool bx0 = x[0];            // => true
bool bx1 = x[1];            // => false

const Integer cx(0b1001);
bool bcx0 = cx[0];          // => true
bool bcx1 = cx[1];          // => false

=>
```

- **Index: int**
  - Assertions fangen Fehl-Benutzung ab
- **Rückgabetyp: bool**
- **Element-Funktion muß „const“ sein**
  - Semantisch ja nur ein „lesend“
  - Syntaktisch notwendig für die Anwendung mit Const-Objekt

```
class Integer
{
public:
   explicit Integer(uint32_t value = 0) : mValue(value) {}

   bool operator[](int idx) const;

   ostream& print(ostream& out) const;
   string str() const;

private:
   uint32_t mValue;
};

inline ostream& operator<<(ostream& out, const Integer& arg)
{
   return arg.print(out);
}
```

```
bool Integer::operator[](int idx) const
{
   assert(idx>=0);
   assert(idx<32);
   return (uint32_t(1)<<idx) & mValue;
}
```

- **Das war ja einfach**
- **Dann kommen wir zum schreibenden Zugriff**
- **Kann ja auch nicht schwerer sein**
- **Wir lieferen einfach eine Referenz auf das Bit zurück**
  - Dann kann der Nutzer einfach das Bit setzen

```
<bit>& operator[](int idx);
```

- **Das war ja einfach**
- **Dann kommen wir zum schreibenden Zugriff**
- **Kann ja auch nicht schwerer sein**
- **Wir lieferen einfach eine Referenz auf das Bit zurück**
  - Dann kann der Nutzer einfach das Bit setzen

```
<bit>& operator[](int idx);
```

- **Nur, wie liefert man eine Referenz auf ein Bit zurück?**
- **Das geht in C++ ja gar nicht**
  - Man kann nur Bytes adressieren
  - Bits haben keine Adresse

- **Was macht man denn nun?**

Regel für alle OO-Sprachen:

- **Hat man ein Problem, dann macht man eine Klasse daraus**

=>

- **Problem**
  - Adresse von Bit ist nicht möglich
- **Lösung**
  - Klasse daraus machen
  - Steht stellvertretend für das Bit
  - Kann mit einem Bool geschrieben werden

```
Integer x(0b101);
x[0] = false;
x[1] = true;
```

- **Hinweis**
  - Man nennt sowas auch einen „Proxy“
  - Mit einem so tollen Begriff ist der Vortrag jetzt noch viel wichtiger und besser und aufregender geworden...

Umsetzung

- **Klasse draus machen**
  - => Klasse „BitProxy“
- **Steht stellvertretend für das Bit**
  - Benötigt also die Adresse des Byte und die Bit-Nummer
    - Hier die Adresse des „uint32_t“ und die Bit-Nummer in dem „uint32_t“ Wert
- **Kann mit einem Bool geschrieben werden**
  - Zuweisungs-Operator mit „bool“ Schnittstelle

- **Das klingt doch einfach, also mal los...**

```
class Integer
{
public:
   explicit Integer(uint32_t value = 0) : mValue(value) {}

   bool operator[](int idx) const;
   BitProxy operator[](int idx);

   ostream& print(ostream& out) const;
   string str() const;

private:
   uint32_t mValue;
};

inline ostream& operator<<(ostream& out, const Integer& arg)
{
   return arg.print(out);
}
```

```
BitProxy Integer::operator[](int idx)
{
   assert(idx>=0);
   assert(idx<32);
   return BitProxy(mValue, idx);
}
```

```
class BitProxy
{
public:
   explicit BitProxy(uint32_t& value, int bit)
    : mValue(value), mBit(bit) {}

   BitProxy& operator=(bool);

private:
   uint32_t& mValue;
   int mBit;
};
```

```
BitProxy& BitProxy::operator=(bool setbit)
{
   if (setbit)
   {
      mValue |= uint32_t(1)<<mBit;
   }
   else
   {
      mValue &= uint32_t(0xFFFFFFFF) ^ (uint32_t(1)<<mBit);
   }

   return *this;
}
```

- **Super, das war es dann ☺**

- **Super, das war es dann ☺**
- **Wirklich?**

- **Super, das war es dann ☺**
- **Wirklich?**
- **Nein!**
  - Warum nicht?

- **Super, das war es dann ☺**
- **Wirklich?**
- **Nein!**
  - Warum nicht?

- **Bitte denkt daran**
  - Diese Funktion (Non-Const Operator [ ]) ist **nicht** für den Schreibzugriff
    - BitProxy operator[](int idx);
    - bool operator[](int idx) const;
  - Sondern für alle Zugriffe auf Non-Const Objekte
    - => Auch für die lesenden Zugriffe

- **Der Code unten lief bisher**
- **Und schlägt nun fehl**
- **Denn obwohl wir nur “lesen” wollen**
  - Der Compiler nimmt die Non-Const Funktion
  - Denn wir haben ein Non-Const Objekt
  - => BitProxy operator[](int idx);
- **Und BitProxy ist kein „bool“**
  - Und kann daher nicht zugewiesen werden
- **Und nun?**

```
Integer x(0b101);
bool bx0 = x[0];            // => Compiler-Fehler
```

Ganz einfach

- **Einfach eine Konvertierung zu “bool” in die Klasse “BitProxy” einbauen**

```
class BitProxy
{
public:
   explicit BitProxy(uint32_t& value, int bit)
    : mValue(value), mBit(bit) {}

   BitProxy& operator=(bool);

   operator bool() const
   {
      return (uint32_t(1)<<mBit) & mValue;
   }

private:
   uint32_t& mValue;
   int mBit;
};
```

Und sind wir jetzt fertig?

- **Lieber nochmal überprüfen, ob der Schreibzugriff auf Const-Objekte einen Compiler-Fehler liefert**

```
class Integer
{
   bool operator[](int idx) const;
   ...

const Integer cx(0b1001);
cx[0]=false;                => Compiler-Fehler
```

- **Klappt**
  - Denn eine Bool-Rückgabe ist ein R-Value und läßt sich nicht schreiben
  - Noch expliziter mit einer “const bool” Rückgabe – aber nicht notwendig

```
class Integer
{
public:
   explicit Integer(uint32_t value = 0) : mValue(value) {}

   const bool operator[](int idx) const;
   BitProxy operator[](int idx);

   ostream& print(ostream& out) const;
   string str() const;

private:
   uint32_t mValue;
};

inline ostream& operator<<(ostream& out, const Integer& arg)
{
   return arg.print(out);
}
```

Und fertig!
Fragen?