Praxisorientierte Einführung in C++
Lektion: "Qualifier"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

Qualifier und Modifier

- C++ kennt eine Fülle von Qualifiern (aka Modifiern)
- Diese sind in 5 Gruppen aufgeteilt
- Qualifier werden in Deklaration integriert

Gruppe	Schlüsselwort	default
1	`auto`, `register`, `static`, `extern`, `typedef`	`auto`
2	`signed`, `unsigned`	`signed`
3	`short`, `long`	weder noch
4	`const`	nicht `const`
5	`volatile`	nicht `volatile`

Eine Variable kann optional aus jeder Gruppe einen Qualifier erhalten

```
int i = 0;  // auto, signed, weder short noch long, nicht const und nicht volatile
static short unsigned int j = 7;                    // ...
static unsigned long const volatile int crazy = 42; // ...
```

Zunächst aber nur ein paar

- Hier, zunächst nur
  - `const`
  - `volatile`
  - `static` und `extern`

```
const int i = 0;
volatile char x;
static char *y;
extern bool z;
```

Qualifier - const

- const -deklarierte Variablen sind konstant
  - Sie müssen also bei der Instanzierung initialisiert werden (später ist das ja wegen der Konstantheit nicht mehr möglich)

Fehler

```
const int i;
```

So geht's

```
const int i = 10;
const float j[3] = {2,3,4};
```

const - Referenzen

- Referenzen auf const Variablen müssen als const deklariert werden

Fehler

```
const int i = 10;
int &iref = i;
```

- Andersrum geht es allerdings: Es ist möglich const Referenzen auf nicht const Variablen zu deklarieren

Das geht

```
int i = 10;
const int &iref = i;
```

- const ist soetwas wie ein "Vertrag", den Wert von i nicht über die Referenz zu modifizieren - Erlaubt Compiler-Optimierungen (später hierzu mehr, bei Funktionen)

Einschub - Constant Folding

```
const int x = 3;
const int y = 2;

std::cout << (x + y) * x << std::endl;
```

- Compiler kann konstante Ausdrücke aus Werten, die zum Zeitpunkt der Übersetzung feststehen dann auch auswerten.
- Obiges ist also äquivalent zu

```
std::cout << 15 << std::endl;
```

const - Pointer

- Bei Pointern ist Position von const wichtig

```
int i = 0;
```

```
const int *pi = &i;
```

```
int const *pi2 = &i;
```

```
int *const pi3 = &i;
```

```
const int * const pi4 = &i;
```

- const "bindet nach links", ausser es ist ganz links, dann "bindet es nach rechts"
- Der const-Vertrag sieht hier jedesmal ein wenig anders aus (Nächste Folien)

const - Pointer

- Bei diesen Varianten kann Ziel des Pointers nicht beschrieben werden (Der Vertrag lautet "Pointer auf konstanten int")
- Die Zeigervariable selbst, kann aber verändert werden, nicht aber das auf was er zeigt

```
const int *pi = &i;  // zeiger auf const int
```

```
int const *pi2 = &i; // aequivalent zu oben
```

const - Pointer

- Bei dieser Variante lautet der Vertrag "Konstanter Pointer auf `int`"
- Der Zeiger selbst ist konstant (und muss initialisiert werden bei Deklaration und kann nicht umgebogen werden)
- Das worauf der Zeiger zeigt ist aber *nicht*

```
int *const pi3 = &i;
```

const - Pointer

- Und hier lautet er "Konstanter Zeiger auf konstanten int"

```
const int * const pi4 = &i;
```

const - Pointer

- Das Spielchen kann man beliebig weitertreiben.

```
const int ** const * i = CrazyStuff;
```

- Das wird aber so gut wie nie benötigt

const - Arrays

- Bei Arrays macht `const` nur an erster Stelle Sinn

```
const int myConstIntArray[3] = {1,2,3};
int const myConstIntArray[3] = {1,2,3};
```

- Denn sie sind sowieso äquivalent zu konstanten Zeigern

volatile

- Schaltet Optimierungen bezüglich der Variablen aus
- Sagt dem Compiler: Variable kann von externem Prozess verändert werden. Sie muss also bei jedem Zugriff wieder aus dem Hauptspeicher geladen/geschrieben werden

```
volatile char* z = 0;
```

- Bindet nach links genau wie `const`

`volatile` - Systemprogrammierung

- Nützlich z.B. für Gerätetreiberprogrammierung
- Memory Mapped I/O-Ports

volatile - Beispiel

```
#define TTYPORT 0x17755U
volatile char *port17 = (char*)TTYPORT;
*port17 = 'o';
*port17 = 'N';
```

- Ohne volatile wuerde der Compiler *port17 = 'o'; "wegoptimieren", oder z.B. in einem Register zwischenspeichern

static vs. extern

- Bis jetzt haben wir keinen Unterschied zwischen globalen und lokalen Variablen gemacht
- Lokale Variablen haben "natürlich" eingeschränkte Sichtbarkeit (später dazu mehr)
- Bei globalen Variablen und mehreren Translation Units ist die Sache nicht mehr ganz so klar
  - Es wird Mechanismus benötigt um...
    - ...Variablen in anderen Translation Units zu referenzieren
    - ...Die Sichtbarkeit von Variablen zu kontrollieren

extern

- `extern` besagt "Die Variable lebt in einer anderen Translation Unit"

foo.h

```
extern int foo;
```

foo.cc

```
int foo;
```

main.cc

```
#include "foo.h"
int main() {
  foo = 3;
}
```

- Vergleichbar mit Deklaration einer Funktion

Qualifier `static`

- `static` dagegen beschraenkt die Sichtbarkeit einer Variable auf die aktuelle Translation-Unit

foo.cc

```
static int foo = 0;
```

main.cc

```
#include "foo.h"

static int foo = 1;

int main() {
  foo = 3;
}
```

static vs. auto

- Im lokalen Scope einer Funktion hat `static` eine komplett andere Bedeutung
- Variablen, die im lokalen Scope `static` deklariert werden,
  - werden nur einmal initialisiert
  - werden nicht aus dem Speicher gelöscht, wenn die Funktion zuende ist

```
#include <iostream>
void foo(){
     int i = 4;
     static int j=4;
     i++; j++;
     std::cout << "i:" << i << "   j:" << j << std::endl;
}
int main(){
    foo();
    foo();
}
```

Ausgabe

```
i:5 j:5
i:5 j:6
```

static vs. auto

- Lokale `static` Variablen sind mit Vorsicht zu genießen
  - z.B. i.d.R. nicht Thread-safe
- Hin und wieder lassen sich damit aber Sachen sehr elegant Lösen
  - Funktionsaufrufszähler
  - Objektzähler
  - Singelton Pattern (dazu später noch was)
  - Profiling