Mehr Sicherheit und Komfort in C-Anwendungen

Karl Nieratschker

C ist aufgrund der vielen Möglichkeiten für Embedded Systeme besonders gut geeignet. Mitunter ist aber gerade dies der Grund dafür, dass Compilern Programmierfehler nicht frühzeitig erkennen können, oder dass Programme schwer lesbar sind. Hier kann der Einsatz von C++ hilfreich sein, denn viele der rückwärtskompatiblen und neuen Spracheigenschaften sind deutlich leistungsfähiger, ohne zusätzliche Systemressourcen zu verbrauchen.

Grundsätzliche Konzepte zur Qualitätsverbesserung

Es gibt verschiedene Möglichkeiten, die Qualität von Software zu verbessern:

- Die Idee des Konzeptes *„Sicherheit durch Einschränkung“* besteht darin, auf gefährliche Konstrukte entweder zu verzichten oder diese nicht zuzulassen.
- Durch das Konzept der *„Syntaxvereinfachung“* wird das Schreiben und Lesen von Programmen vereinfacht. Die Wichtigkeit dieser Vorgehensweise wird häufig unterschätzt und mit Begriffen der Art „syntaktischer Zucker“ verharmlost. Dabei besitzt gerade dieses Konzept ein beträchtliches Potenzial, Fehler erst gar nicht entstehen zu lassen.
- Das Konzept der *„Gruppierung zusammengehörender Dinge“* sorgt für eine Reduzierung der Komplexität, was zu einem leichteren Verständnis und einem besseren Überblick führt.

Anwendung der Konzepte mit den Bordmitteln von C

C bietet bereits einige Möglichkeiten für die Umsetzung dieser Konzepte. Unsichere Sprachelemente, wie Makros, Unions und Cast-Operationen sollte man generell möglichst vermeiden. Eine Liste von #define-Statements für Zustände eines Zustandsautomaten könnte man z.B. durch einen entsprechenden Aufzählungstyp ersetzen. Makros gelten als unsicher, weil der damit verbundene Textersetzungsmechanismus überall, und damit auch an unsinnigen Stellen im Programm angewendet werden kann. Ein Enum-Typ dagegen ist nur da erlaubt, wo C dies zulässt.

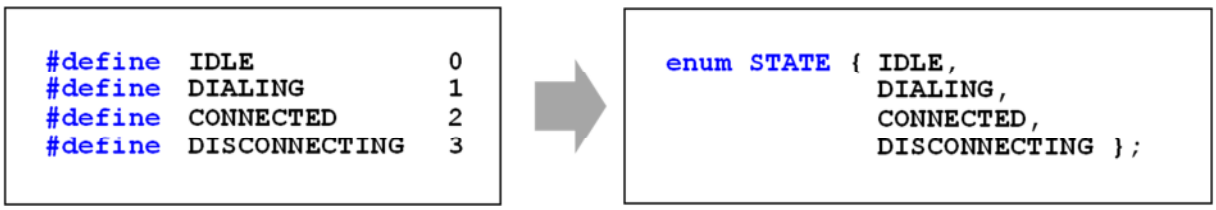


Abb. 1: Aufzählungstyp statt Makros

Ein anderes Beispiel ist die Verwendung von const bei der Deklaration von Pointern, wenn von dem zugehörigen Speicher nur gelesen werden soll. Durch diese Einschränkung verhindert man ein versehentliches Schreiben auf diesen Speicher.

Die Programmiersprache C++

Da C++ rückwärtskompatibel ist, können C-Programme grundsätzlich auch mit einem C++-Compiler übersetzt werden. Allerdings stellt C++ höhere Anforderungen an die Typsicherheit, was dazu führen kann, dass der C++-Compiler bei gefährlicher oder fehlerhafter Anwendung von C-Konstrukten Warnungen oder sogar Fehler meldet. Dies wirkt sich z.B. bei Aufzählungstypen aus. In C ist es z.B. zulässig, einer Variablen eines Aufzählungstyps STATE ein Symbol eines anderen Enum-Typs DIRECTION zuzuweisen, was aber sicher nicht im Sinne des Anwenders ist. Der C++-Compiler generiert in diesem Fall einen Fehler.

C

```
enum STATE { IDLE, DIALING,
                CONNECTED, DISCONNECTED };
enum DIRECTION { UP, DOWN };

int main()
{
    enum STATE state = IDLE;
    ...
    state = DIALING;
    ...
    state = CONNECTED;
    ...
    state = DISCONNECTED;
    ...
    state = 5;     // Evtl. Warnung
    state = UP;    // Evtl. Warnung
    ...
    int i = IDLE; // ok
}
```

C++

```
enum STATE { IDLE, DIALING,
                CONNECTED, DISCONNECTED };
enum DIRECTION { UP, DOWN };

int main()
{
    STATE state = IDLE;     // kein enum!
    ...
    state = DIALING;
    ...
    state = CONNECTED;
    ...
    state = DISCONNECTED;
    ...
    state = 5;              // Fehler
    state = UP;             // Fehler
    state = (STATE)5;       // möglich!
    ...
    int i = IDLE;           // ok
}
```

Abb. 2: Vergleich der Enum-Typen

Die höhere Typgenauigkeit wirkt sich auch auf const aus. Möchte man z.B. die Dimension eines Arrays mithilfe eines konstanten Integers statt mit einem (unsicheren) Makro definieren, dann meldet der C-Compiler einen Fehler, während der C++-Compiler dies akzeptiert.

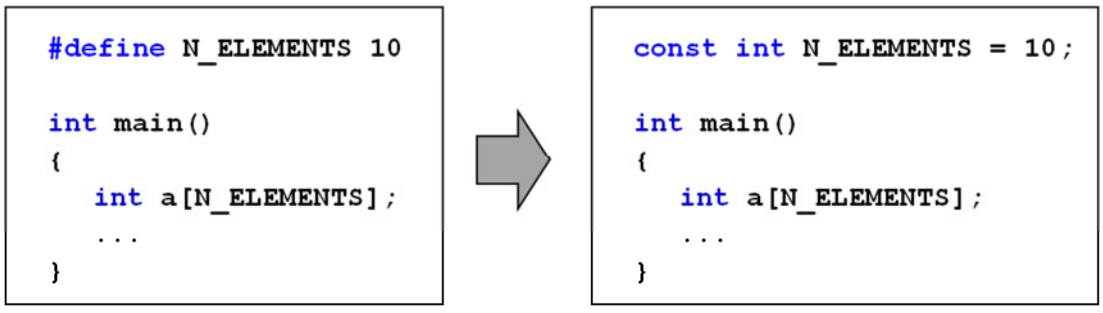


Abb. 3: const int statt Makro

Selbstverständlich bietet C++ auch viele neue Möglichkeiten. Die wesentlichste Neuerung stellt die Unterstützung der objektorientierte Programmierung (OOP) dar. Die damit mögliche Datenkapselung fällt in die Kategorie *„Gruppierung zusammengehörender Dinge“* und bietet z.B. Einschränkungsmöglichkeiten der Sichtbarkeit, die weit über das hinaus gehen, was C anzubieten hat.
C++ bietet aber auch unabhängig von OOP interessante neue Spracheigenschaften, wobei immer darauf geachtet wurde, dass die von C bekannte Effizienz in Bezug auf Speicherplatz- und Laufzeitbedarf erhalten bleibt. Ein Beispiel hierfür ist das C++-Schlüsselwort inline, durch dessen Hilfe es möglich ist, den Aufruf einer Funktion durch deren Funktionskörper zu ersetzen, um den Overhead des Hin- und Rücksprungs zu vermeiden. In C kann dies nur mithilfe einer typunsicheren Makro-Funktion oder einem Pragma erreicht werden.

Neue OOP-unabhängige Spracheigenschaften in C++

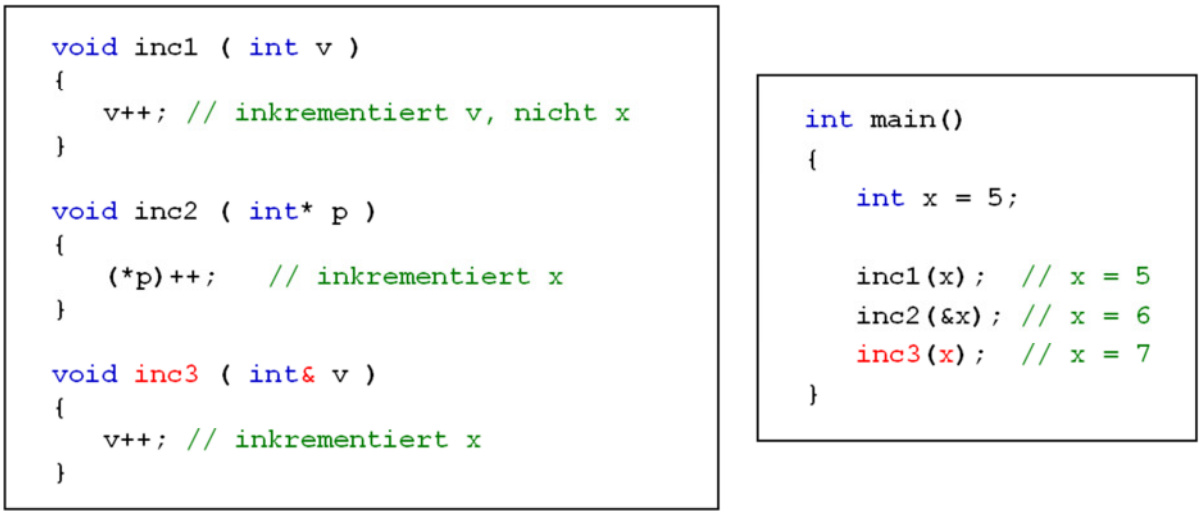
Strukturen und Aufzählungstypen sind in C++ vollwertige Datentypen. D.h., beim Deklarieren von Struktur- und Enum-Variablen dürfen die Schlüsselwörter struct und enum ohne explizite typedef-Anweisung weggelassen werden. Aufzählungstypen können innerhalb einer Strukturdefinition angelegt werden, ohne dass dies Speicherplatz kostet. Die Sichtbarkeit des Enum-Typs wird dadurch auf den Bereich der enthaltenden Struktur eingeschränkt. Der Enum-Typ kann sogar anonym sein, wodurch (als typsichere Alternative zu Makros) Symbole für Werte definiert werden können, die nur im Kontext („Scope“) der umfassenden Struktur Gültigkeit haben sollen.

```
struct Vector
{
      enum {SIZE = 3};
      int v[SIZE];
};
```

```
int main()
{
   Vector vec;     // KEIN struct!
   ...
   for(int i=0; i< Vector::SIZE; ++i)
          vec.v[i] = i;
   ...
}
```

Abb. 2: Anonymer Aufzählungstyp in einer Struktur

Nicht zuletzt wegen der Komplexität der Syntax gehört der Umgang mit Pointern zu den Dingen, die am häufigsten zu fehlerhaftem Code führen. In C++ können Pointer oft durch *Referenzen* ersetzt werden. Der Vorteil besteht darin, dass beim Referenzzugriff auf eine (Struktur-)Variable die einfache Syntaxform wie beim Direktzugriff verwendet wird, während der Compiler aber Code wie bei einem Pointerzugriff generiert. Da der Compiler außerdem auch sicherstellt, dass Referenzen beim Anlegen initialisiert werden, erübrigt sich die sonst übliche NULL-Pointer Prüfung, was nicht nur zur Sicherheit beiträgt, sondern gleichzeitig auch noch Code und Laufzeit einspart.

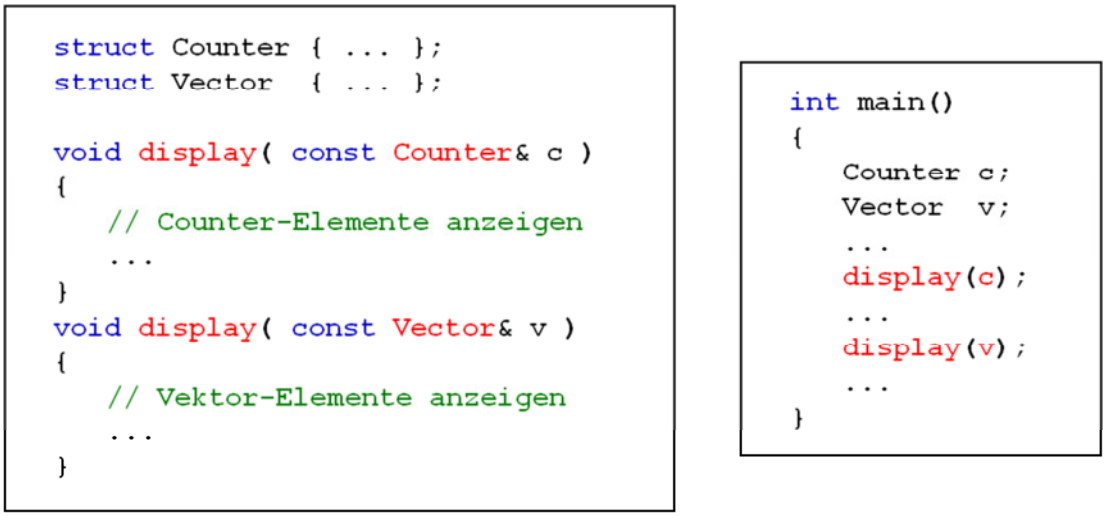


```
void inc1 ( int v )
{
   v++; // inkrementiert v, nicht x
}

void inc2 ( int* p )
{
   (*p)++;   // inkrementiert x
}

void inc3 ( int& v )
{
   v++; // inkrementiert x
}
```

```
int main()
{
   int x = 5;

   inc1(x);  // x = 5
   inc2(&x); // x = 6
   inc3(x);  // x = 7
}
```

Abb. 3: Referenzen

Häufig muss eine Operation mit unterschiedlichen Operanden ausgeführt werden. Z.B. könnte die Funktion displayCounter die Elemente einer Struktur Counter, und die Funktion displayVector die Elemente einer Struktur Vector anzeigen. In C++ ist es nicht nötig, die beiden Funktionen namentlich zu unterscheiden, d.h., beide Funktionen könnten display heißen. Beim Aufruf von display prüft der Compiler den Typ des Aufrufparameters (Counter bzw. Vector) und ruft dann die dazu gehörende display-Funktion auf. Diese Spracheigenschaft nennt sich *Funktionsüberladung* und leistet einen wesentlichen Beitrag zur Vereinfachung von Programmen.



```
struct Counter { ... };
struct Vector  { ... };

void display( const Counter& c )
{
   // Counter-Elemente anzeigen
   ...
}
void display( const Vector& v )
{
   // Vektor-Elemente anzeigen
   ...
}
```

```
int main()
{
   Counter c;
   Vector  v;
   ...
   display(c);
   ...
   display(v);
   ...
}
```

Abb. 4: Funktionsüberladung

Implementiert man die Struktur Vector als Array mit drei Integerelementen und möchte man dann zwei Vector-Variable v1 und v2 addieren, dann geht dies leider nicht mit der nahe liegenden Anweisung v1+v2. Da Vector ein benutzerdefinierter Datentyp ist, weiß der Compiler nicht, was im Fall einer Addition zu tun ist. Selbstverständlich könnte man eine Funktion Add schreiben, die zwei Vektoren entgegennimmt und den Ergebnisvektor zurückliefert. Aber der Umgang mit Vektoren wird dadurch deutlich unnatürlicher. In C++ gibt es für dieses Problem eine elegante Lösung: ersetzt man den Namen Add der Additionsfunktion durch das Schlüs-

selwort operator+, akzeptiert der Compiler das +-Zeichen für die Addition zweier Vektoren und ruft einfach die Additionsfunktion auf. Diese Spracheigenschaft nennt man *Operatorüberladung*.

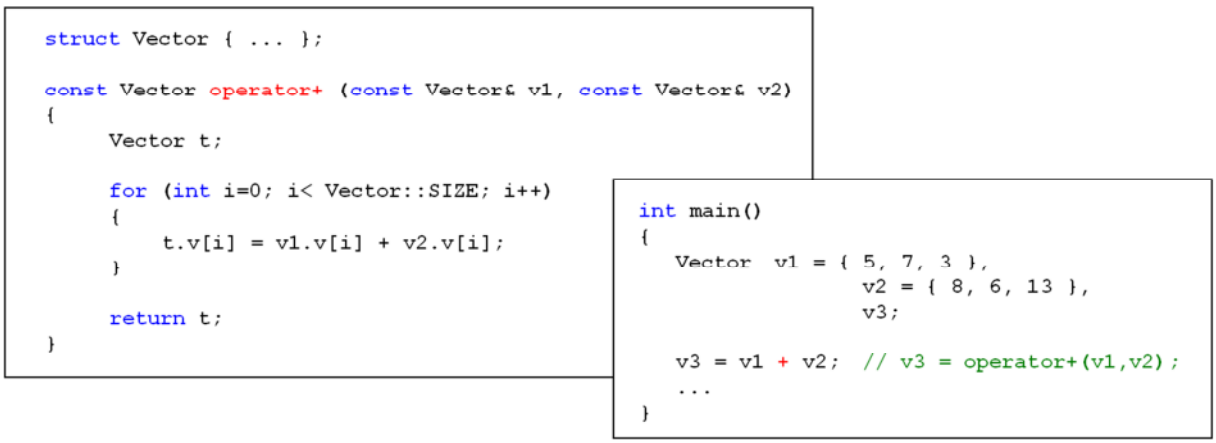


Abb. 5: Operatorüberladung

Beispiel: Erkennung eines Zahlbereichsüberlaufs

Die Addition zweier Integerwerte kann zu einem Zahlbereichsüberlauf führen, der weder von C, noch von C++ geprüft wird. Um einen Überlauf erkennen zu können, könnte der Anwender die Addition wieder durch eine Add-Funktion ersetzen und dort die Prüfung vornehmen. Eine Operatorüberladung kommt in diesem Fall nicht infrage, da Operatoren für Standarddatentypen wie int nicht überladen werden können. Allerdings könnte man einen eigenen Datentyp SafeInt16 mit einem Element vom Typ int definieren, für den dann auch operator+ und falls nötig auch weitere arithmetische Operationen überladen werden können. Die Überlaufprüfung kann in eine eigene Funktion ausgelagert werden, die von allen inline-definierten Operatoren aufgerufen wird.
Soll die Überprüfung nur bedingt gemacht werden, kann man im Programm mit einem Datentyp INT16 arbeiten, der mittels typedef je nach Bedarf als int oder als SafeInt16 definiert wird. Um eine vollständige Austauschbarkeit zwischen int und SafeInt16 zu erzielen muss man eine kleine Anleihe aus dem OOP-Bereich machen: die Struktur SafeInt16 benötigt dann einen Konstruktor für die Wandlung eines int in einen SafeInt16, und einen operator int() für die Wandlung eines SafeInt16 in einen int.

```
struct SafeInt16
{
  enum { MIN_VAL = 0x8000,
         MAX_VAL = 0x7FFF };

  int v;

  SafeInt16(int iv = 0) { v = iv; }

  operator int() { return v; }
};

void CheckOverOrUnderflow(...)
{
  ...
}

inline SafeInt16 operator+(
          SafeInt16 a, SafeInt16 b)
{
  CheckOverOrUnderflow(a,b);

  return SafeInt16(a.v + b.v);
}
```

```
//typedef int      INT16;
typedef SafeInt16 INT16;

void main()
{
  INT16 a = 0x7FFF;
  INT16 b = -1;
  int   c;

  c = a + b;

  printf("a: %d\n", (int)a);
  printf("b: %d\n", (int)b);
  printf("c: %d\n",      c);

  printf("Sizeof INT16: %d\n",
         sizeof(INT16));
}
```

Abb. 6: Überlaufprüfung mit SafeInt16

Ein Blick auf den Ressourcenbedarf des vom IAR-Compiler generierten Codes für den16-Bit-Prozessor M16C von Renesas zeigt, dass sowohl der Konstruktor als auch operator int vollständig wegoptimiert wurden und jeweils nur ein MOV-Befehl erzeugt wurde. operator+ benötigt auch nur einen ADD- und 2 MOV-Befehle. Der Hauptaufwand, bestehend aus dem Aufruf und der Durchführung der Überlaufprüfung, ist in C und C++ identisch und für den Vergleich somit irrelevant.

```
 void main()
        main:
 {
7CF207        ENTER   #0x7
   INT16 a = 0x7FFF;
75CBFBFF      MOV.W   #0x7fff, -0x5[FB]
7F
   INT16 b = -1;
D9FBF9        MOV.W   #-0x1, -0x7[FB]
   int   c;

   c = a + b;
73BBF9FD      MOV.W   -0x7[FB], -0x3[FB]
73BBFBF9      MOV.W   -0x5[FB], -0x7[FB]
754BFD        PUSH.W  -0x3[FB]
754BF9        PUSH.W  -0x7[FB]
F5....        JSR.W   ??CheckOverOrUnderflow
7DB4          ADD.B   #0x4, SP
73B0FD        MOV.W   -0x3[FB], R0
A1B0F9        ADD.W   -0x7[FB], R0
730BFB        MOV.W   R0, -0x5[FB]
73BBFBF9      MOV.W   -0x5[FB], -0x7[FB]
```

```
   printf("a: %d\n", (int)a);
7DE2FF7F      PUSH.W  #0x7fff
75C4....      MOV.W   #`?<Constant "a: %d\\n">`, A0
F5....        JSR.W   ?Subroutine0
          ??CrossCallReturnLabel_3:
7DB2          ADD.B   #0x2, SP
   printf("b: %d\n", (int)b);
7DE2FFFF      PUSH.W  #-0x1
75C4....      MOV.W   #`?<Constant "b: %d\\n">`, A0
F5....        JSR.W   ?Subroutine0
          ??CrossCallReturnLabel_2:
7DB2          ADD.B   #0x2, SP
   printf("c: %d\n",      c);
754BF9        PUSH.W  -0x7[FB]
75C4....      MOV.W   #`?<Constant "c: %d\\n">`, A0
F5....        JSR.W   ?Subroutine0
          ??CrossCallReturnLabel_1:
7DB2          ADD.B   #0x2, SP

   printf("Sizeof INT16: %d\n",
          sizeof(INT16));
7DE20200      PUSH.W  #0x2
75C4....      MOV.W   #`?<Constant "Sizeof INT16: %d\\n">`,
A0
F5....        JSR.W   ?Subroutine0
          ??CrossCallReturnLabel_0:
7DB2          ADD.B   #0x2, SP
 }
7DF2          EXITD
```

Abb. 7: Assemblercode der Überlaufprüfung

Zusammenfassung

C++ bietet sehr viel bessere Möglichkeiten als C, wenn es darum geht, Fehler zu entdecken bzw. diese erst gar nicht entstehen zu lassen. Wesentliche Ursachen dafür sind die größere Typsicherheit, Syntaxvereinfachungen und das objektorientierte Programmiermodell (OOP). Viele dieser Möglichkeiten können unabhängig von OOP eingesetzt werden. Aufgrund der Rückwärtskompatibilität können somit auch bestehende C-Applikationen von den Vorteilen profitieren, wenn ein passender C++ - Compiler zur Verfügung steht. Dies ist aufgrund der zunehmenden Leistungsfähigkeit moderner Prozessoren immer häufiger der Fall. Der Ressourcenbedarf für die hier betrachteten C++ - Eigenschaften ist mit dem von C nicht nur vergleichbar, sondern mitunter sogar niedriger.

Autor

Dipl. Inf. Karl Nieratschker verfügt über eine mehr als 25-jährige Erfahrung in den Bereichen Softwareentwicklung, Support, Beratung und Training für Embedded-/Realtime- und Windows-Programmierung. Seit 1999 ist er als freiberuflicher Trainer, Softwareberater und Coach tätig. Seine besonderen Schwerpunkte sind „Objektorientierte Programmierung in ressource-limitierten Systemen“ und „Multithreading/Multicore-Programmierung“.

Internet: www.skt-nieratschker.de
Email: office@skt-nieratschker.de