C++-Standard – Wer hält sich dran?

Stefan Kuhlins

Juli 1994*

Die Sprache C++ wird fortwährend weiterentwickelt. Keinesfalls genügen alle der hier untersuchten Produkte *einem* Standard. An einer derartigen Richtschnur zwirbelt zur Zeit ein ANSI-Gremium namens „X3J16“. Doch bis „ANSI C++“ den Weg der Instanzen genommen hat, bleiben als De-facto-Standard die AT&T-cfront-Versionen und vorab veröffentlichte ANSI-C++-Arbeitspapiere, aufgrund derer sich einige Compiler schon jetzt (vorläufiger) ANSI-Konformität rühmen.

Die „einfachen“ Erweiterungen von C++ gegenüber C (entsprechend cfront 2.1) gehören heute zum Standardrepertoire der Compiler – die Wichtigsten davon sind: Klassen, einfache wie mehrfache Vererbung, Polymorphismus, überladbare Operatoren und abstrakte Klassen. Doch schon bei Templates (cfront 3.0) scheiden sich die Compiler. Ausnahmebehandlung (nach cfront 3.1) weist nur noch die Hälfte der am Markt befindlichen Produkte auf. Die von ANSI unter anderem ins Spiel gebrachten Laufzeittypinformationen (RTTI) kann bis jetzt gar nur Borland vorzeigen.

Für weiteren in der ANSI-Debatte befindlichen Sprachzuwachs muß man auf die nächsten Compiler-Updates warten: getrennte Namensräume, um etwa eigentlich nicht für die Zusammenarbeit geschaffene Bibliotheken dennoch gleichzeitig einsetzen zu können, gehören ebenso dazu wie der „neue“ Datentyp `bool` für boolesche Werte. Darüber hinaus will ANSI der C++-Welt standardisierte Bibliotheken bescheren, die über die mittlerweile weitgehend einheitlichen Sammlungen für Dateizugriffe (*streams*) inklusive Bildschirm und Tastatur hinausgehen. Die ersten Ausläufer davon liefert Borland als String-Klassen.

Brot und Butter

Doch selbst in diesen einfachen Bibliotheken konnten wir Unterschiede ausmachen. Oftmals liegen sie derart im Detail verborgen, daß Inkompatibilitäten beim Compiler-Wechsel nicht zu befürchten sind; eher schon dürften sich die Seiteneffekte bei der täglichen Arbeit zeigen: Bei Watcom, Borland und Symantec fehlt zum Beispiel in der Deklaration des Ausgabeoperators

*c't 7/1994, S. 162–163

„<<“ für Zeiger das Schlüsselwort „`const`“, was die Ausgabe konstanter Zeiger behindert – C++-Frischlinge könnte das stutzig machen.

Ebenso bewältigten die Compiler im großen und ganzen die C++-Grundlagen, auch wenn sie sich dabei die eine oder andere Blöße gaben – jedes Produkt wurde mit in [1] und [2] veröffentlichten Quelltexten konfrontiert: Die gröbsten Ungereimtheiten geben die Listing-Fragmente im Anhang ab Seite 5 wieder. Zum Einsatz kamen jeweils die „abgespeckten“ Compiler-Versionen für die DOS-Kommandozeile. Die Schwachstellen betreffen vor allem die Ausgabe von Enumeratoren, die Bindung von Elementfunktionen, Standardargumente, das Erkennen von Mehrdeutigkeiten sowie benutzerdefinierte Typumwandlungen. Für GNU, Symantec und Salford dürfte peinlich sein, daß sie an einem mehrfach angewandten Präfix-Inkrementoperator scheitern.

In einer weiteren Runde haben wir neben den neuen Techniken wie Templates, Exception Handling und RTTI die Detailverbesserungen abgeklopft, beispielsweise ob sich die Operatoren `new[]` und `delete[]` überschreiben lassen und ob das Initialisieren statischer Datenelemente von Template-Klassen möglich ist. Die Tabelle auf Seite 4 zeigt das Ergebnis dieser Durchsicht.

Wer einen Codegenerator benutzt, den könnten Längenbeschränkungen bei den Bezeichnern stören. Visual C++ quittiert lange Bezeichner mit “fatal error C1066: compiler limit: decorated name length exceeded”. Borland macht bei 250 Zeichen dicht, obwohl es im Handbuch anders nachzulesen ist. Watcom, Symantec und GNU haben damit keine Probleme. GNU stellt auf Wunsch gar eine diesbezügliche Warnung zur Vorbereitung auf “certain obsolete, brain-damaged compilers” bereit – komisch, daß ausgerechnet diese Option nicht funktioniert.

Ausnahmen als Regel

Die Umsetzung der neueren Konstrukte wie Templates, Ausnahmebehandlung, Laufzeittypinformationen und Namespaces ist keinesfalls selbstverständlich. Der einzige Compiler, der Templates nicht unterstützt, ist Visual C++. Lediglich ein sogenanntes „Template Definition Utility“ versucht primitive Templates nachzuahmen. Borland, Watcom und Salford können Ausnahmen zur Regel erklären; Visual C++ kennt immerhin schon die Schlüsselwörter. Borland allein verwaltet Typinformationen auch während der Laufzeit (RTTI).

In bezug auf die Umsetzung des C++-Standards bildet Visual C++ eindeutig das Schlußlicht. Symantec und GNU haben immerhin Templates im Programm. Eine Sonderrolle nimmt Salford ein: Templates und Ausnahmebehandlung funktionieren nur mit starken Einschränkungen. Der Compiler leidet an vielen Kinderkrankheiten gleichzeitig (über 30 Programme aus [2]

waren nicht übersetzbar oder wurden fehlerhaft übersetzt).

Borland gibt sich die wenigsten Blößen bei der Umsetzung der Sprachkonzepte und ist damit bezüglich des C++-Sprachumfangs der Konkurrenz derzeit mehr als eine Nasenlänge voraus. Zu Benchmarks bezogen auf C++ mußten die Kontrahenten nicht antreten. Aber mein rein subjektiver Eindruck, bestätigt durch eine nicht repräsentative Minimeßreihe während der C++-Experimente, brächte Symantec und Salford auf die Spitzenpositionen, gefolgt von Borland, Microsoft und Watcom. Schlußlicht ist GNU, aber das ist im Testfeld bekanntlich auch der einzig kostenlos erhältliche C++-Compiler.

Literatur

[1] Ellis, Margaret A. und Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley, 1991.

[2] Schader, Martin und Stefan Kuhlins: *Programmieren in C++ — Einführung in den Sprachstandard C++*. Springer-Verlag, 2. Aufl., 1994.

C++-Compiler im Vergleich

Version	Borland 4.0	GNU 2.5.7	Salford 1.54	Symantec 6.11	Visual 1.5	Watcom 9.5b
Basiskonstrukte	•	•	•	•	•	•
beliebig lange Bezeichner	○	•	•	•	○	•
Templates	•	•	(•)*	•	○	•
Ausnahmen	•	○	(•)*	○	○	•
RTTI	•	○	○	○	○	○
Überschreiben der Operatoren `new[]` und `delete[]`	•	○	○	○	○	○
Deklaration eingebetteter Klassen	•	○	○	○	○	○
Überschreiben virtueller Elementfunktionen mit anderem Rückgabetyp	•	○	○	○	○	•
Operatoren für Enumeratoren überladen	•	○	○	○	○	•
allgemeine Initialisierung statischer Datenelemente von Template-Klassen	•	○	○	•	○	•

• implementiert, ○ nicht implementiert, * siehe Text

C++-Kuriositäten

Ausgabe von Enumeratoren (vgl. [2] Kap. 10, Aufg. 2): Symantec C++ und Visual C++ verweigerten das Compilieren mit einer Fehlermeldung.

```
#include <iostream.h>
int main() {
  enum status { rot=0 };
  cout << rot;
  return 0;
}
```

Festlegung der Bindung von Elementfunktionen (vgl. [1] S. 104): Symantec C++, Visual C++, GNU und Watcom C++ übersahen den Fehler im Quelltext.

```
struct X { int f(); };

void k(X* p) {
  int i = p->f();    // jetzt hat X::f() external linkage
}

inline int X::f() {  // Fehler: aufgerufen vor der
  return 0;          // Definition als inline
}
```

Standardargumente (vgl. [1] S. 142f): Symantec C++, GNU und Watcom C++ ignorierten diese Fehler.

```
void f1() {
  int i1;
  extern void g1(int x1 = i1); // Fehler
}

int a2;
int f2(int a2, int b2 = a2);   // Fehler

typedef int I;
int g2(int I, int b2 = I(2));  // Fehler
```

Benutzerdefinierte Typumwandlung (vgl. [1] S. 271): Symantec C++, Visual C++, GNU und Watcom C++ generierten hier falschen Code.

```
class X { public: X(int); };
class Y { public: Y(X); };
Y a = 1;  // Fehler: Y(X(1)) wird nicht versucht
```

Erkennen von Mehrdeutigkeiten (vgl. [1] S. 327): Symantec C++, Visual C++, GNU, Salford und Borland C++ wiesen nicht auf diese Mehrdeutigkeit hin.

```
struct X { operator int(); };
struct Y { Y(X); };
Y operator+(Y, Y);

void f(X a, X b) {
  a+b;  // Fehler, mehrdeutig:
}       //   operator+(Y(a), Y(b)) oder
        //   a.operator int() + b.operator int()
```

Erkennen von Mehrdeutigkeiten (vgl. [2] Kap. 13, Aufg. 2): Symantec C++, Visual C++ und Salford übersahen diese Mehrdeutigkeit.

```
int g(int, int, ...);
int g(int, short, int*);
int x = 72;
int main() {
  g(0, 'X', &x);  // Fehler: mehrdeutig
  return 0;
}
```

Präfix-Inkrementor (vgl. [2] Kap. 10, Aufg. 6): Symantec C++, GNU und Salford weigerten sich, dieses Konstrukt zu übersetzen.

```
#include <iostream.h>
int a;
int main() {
  while (a<5) cout << ++++a << endl;
  return 0;
}
```