Kapitel 9
Ausdrücke

9.1 Einführung

Um es als erstes zu sagen: wer meint, jetzt etwas über „reguläre Ausdrücke“ zu lesen, der liegt falsch. Die behandeln wir hier nicht. Es geht um die direkte Umsetzung komplexer mathematischer Ausdrücke. Die Umsetzung eine mathematischen Formel wie

$$\int_1^5 \frac{1-x}{1+x} dx$$

in ein Programm, das das Integral auswertet, führt in der Regel zur Definition einer Funktion, die den Ausdruck unter dem Integral darstellt und die als Funktionszeiger in eine Integrationsmethode eingebunden wird. Eleganter wäre eine direkte Implementation der Funktion unter dem Integral in der Form

```
r=integrate((1-x)/(1+x),1,5);
```

in der der Ausdruck direkt als Übergabeparameter angegeben wird.

Eine einfachere Schreibweise ist jedoch nicht die einzige Motivation, solche eine Umsetzung zu realisieren. Bei einer mit einer Tabellenkalkulation vergleichbaren Arbeit mit Feldern können Ausdrücke wie

```
A = (1 - B) * C / (5 - C);
```

auftreten, die zwar durch das Überladen von Operatoren abgewickelt werden können, aber bei genauerem Hinsehen die Rechnung in mehrere Einzelschritte zerlegen und jeweils ein komplettes temporäres Feld für jeden Einzelschritt erzeugen. Effektiver wäre eine elementweise Abwicklung in der Form[1]

[1] Der Leser mache sich klar, dass dies in der Regel nur auf Tabellenkalkulationen anwendbar ist. Feldoperationen in der linearen Algebra mit Vektoren, Matrizen oder Tensoren unterliegen meist anderen Gesetzmäßigkeiten, die nicht so einfach aussehen.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_9, © Springer-Verlag Berlin Heidelberg 2010

```
for(i=0;i<A.size();i++)
     A[i] = (1 - B[i]) * C[i] / (5 - C[i]);
```

Möglichkeiten zur Lösung dieser Probleme bestehen in der Verwendung von Template-Techniken, die die Formeln als Ausdrücke interpretieren und in ihre Bestandteile zerlegen. Techniken dieser Art werden unter anderem auch in der boost++ und der blitz++ Bibliothek verwendet. Einige der im Weiteren diskutierten Techniken haben wir bereits mehrfach in einer etwas einfacheren Form in anderen Kapiteln verwendet, so dass Sie hoffentlich an der einen oder anderen Stelle ein Aha-Erlebnis haben werden.

Aufgabe. Implementieren Sie Rechenoperatoren +,-,∗,/ für Container und machen Sie sich daran den Unterschied zwischen der Wirkung solcher Operatoren und der elementweisen Abwicklung klar.[2]
Versuchen Sie, die Rechnung mit Hilfe von Containeralgorithmen zur implementieren. Liefert dieser Ansatz den effizienteren Rechenweg? Welche Probleme entstehen beim Verfolgen dieses Ansatzes?

9.2 Zerlegung der Ausdrücke

9.2.1 Überführung von Methoden in Objekte

So komplex ein Ausdruck auch aussehen mag, letzten Endes führen die üblichen Ausführungsregeln zu einer rekursiven Zerlegung in einen Operator und seine beiden Operanden (*sofern es sich um einen binären Operator handelt; einem unären Operator ist nur ein Operand zugeordnet; wie viele Operanden zu einem bestimmten Typ von Operator gehört, ist dem Compiler bekannt*). Die Operanden sind jedoch hier ziemlich abstrakt, und was bei einer Operation nun tatsächlich erfolgen soll, muss erst noch über einige Stufen von Templateauswertungen ermittelt werden. Da Templatefunktionen nicht mit der notwendigen Flexibilität aufwarten, überführen wir die Operationen zunächst in Objekte:

```
template <class T1,class T2>
BinaryExpression <T1,T2,plus>operator+(T1 const& t1,
                                       T2 const& t2){
     return BinaryExpression<T1,T2,plus>(t1,t2);
}//end operator
```

Der Compiler kennt die mathematischen Regeln zur Auswertung von Formeln und reagiert nach Auflösung von Klammern und Berücksichtigung der Prioritäten auf den Rechenoperator, hier beispielsweise auf den Additionsoperator. Ist dieser

[2] Anmerkung: da wir für Matrizen und Vektoren eigene Klassen definiert haben, können Sie die Operatoren ohne Bedenken bezüglich entstehender Konflikte in Rechenoperationen implementieren.

zwischen den beiden Operanden definiert, wird er ausgeführt, und wir sind bei der normalen Abwicklung von Rechnungen.

Für unsere Zwecke gehen wir aber nun davon aus, dass zwischen den beiden Templateparametern kein Additionsoperator definiert ist. In diesem Fall wird der Compiler nun versuchen, ein Objekt der Klasse `BinaryExpression` zu konstruieren. Die Rolle des `operator+` wird durch die Klasse `plus` übernommen, die später innerhalb des Objektes `BinaryExpression` die Durchführung der Operation zwischen den beiden Operanden übernimmt. Dieses speichert Verweise auf Instanzen der beiden Operatoren ab. Da hier noch einiges andere zu erledigen ist, kommen wir erst später auf die Details zurück.

Diese Aufspaltung erfolgt im Compiler rekursiv, wenn einer der Operanden weitere Operatoren aufweist. Die Anweisung

```
template <class U> struct T { U val; };
T<double> x;
T<int> i,j;
...
(x+i)*j;
```

wird somit beispielsweise zerlegt in[3]

```
BE<
 BE<T<double>,T<int>,plus>,T<int>,multiply>
   (BE<T<double>,T<int>,plus>(x,i),j);
```

Glücklicherweise können wir die Auflösung derartiger Bandwürmer dem Compiler überlassen. Die erste Aufgabe besteht mithin in der Implementation einer hinreichenden Menge binärer sowie einiger unärer Operatoren, die ähnlich aufgebaut sind (vergleiche Compileralgorithmen):

```
template <class T>
UnaryExpression<T,negate> operator-(T const& t){
     return UnaryExpression<T,negate>(t);
}//end function
```

9.2.2 Typkonversion

Um mit Hilfe der Objekte die Operationen ausführen zu können, müssen innerhalb der Ausdrucksklassen (`BinaryExpression`, `UnaryExpression`) auch die Rechenklassen (`plus`, `multiply`) definiert sein. Diese haben wir in ähnlicher Form in der STL kennengelernt. Wir definieren:

---

[3] Dieser Umweg über die Klasse T, die uns schon einmal bei den Objektfabriken begegnet war, ist notwendig, um die Überführung in Objekte der BE-Klasse zu erzwingen, da ansonsten die Standardoperatoren ausgeführt werden.

```
struct plus {
  template <typename T1,typename T2>
  static typename Supertype<T1,T2>::value_type
     eval(T1 const& t1, T2 const& t2){ return t1+t2;}
};
```

Das sieht sicher komplizierter aus, als Sie im ersten Moment vermutet haben, denn hier taucht als Besonderheit die Typkonversionsklasse `Supertype` auf, die den passenden Rückgabetyp erzeugt (und auch dafür sorgt, dass die Rechnung mit diesem Typ ausgeführt wird). Ähnliches ist uns ja schon bei verschiedenen `trait`- oder Eigenschaftsklassen begegnet, etwa wenn die Summe von `char`-Variablen auf einem `int`-Typ ausgeführt werden soll.

Das ist hier nicht ganz trivial, denn wir haben es mit zwei Typen zu tun. Der Ergebnistyp kann im Templateparametersatz nicht angegeben werden, kann aber auch nicht einfach einer der beiden Parametertypen sein, denn würde bei einer möglichen Typmischung von `<int,double>` der Rückgabetyp `int` ausgewählt, bei `<double,int>` aber `double`, wäre man von den Ergebnissen sicherlich nur recht selten wirklich begeistert. Eine Serie von Spezialisierungen

```
template <> struct Supertype<int,double>{
   typedef double value_type;
};
template <> struct Supertype<double,int> {
   typedef double value_type;
};
```

wäre zwar ein möglicher Ausweg, würde aber einen ziemlichen Rattenschwanz an Spezialisierungen mit sich bringen, insbesondere, wenn wir berücksichtigen, dass wir es mit eigenen Typen zu tun haben, die in irgendeiner Form erst einmal aufgelöst werden müssten. Versuchen wir es also mit einer Automatisierung und übertragen wir die Arbeit dem Compiler.

Für die Auswahl des korrekten Obertyps schaffen wir uns zunächst eine Methode zur Feststellung, ob überhaupt eine Konversion möglich ist. Diese funktioniert ähnlich wie die Feststellung eines Vererbungsverhältnisses und besitzt drei Rückgabekonstante, die angeben, ob eine Konversion möglich ist, die Typen A und B gleich sind oder A speicherplatztechnisch größer als B ist:

```
template <typename A, typename B> class
FirstConvertsToSecond{
private:
     typedef char One;
     typedef struct {char a[2];} Two;
     static One Test(B);
     static Two Test(...);
     static A Make();
public:
     enum { yes=
```

```
        (sizeof(FirstConvertsToSecond<A,B>::Test(Make()))
                                          ==sizeof(One)),
               same_type=0,
               bigger=sizeof(A)>=sizeof(B) };
};//end struct

template <typename A>
class FirstConvertsToSecond<A,A>{
public:
     enum { yes=1, same_type=1, bigger=1 };
};//end struct
```

Die Auswahlklasse `Supertype` sucht nun mit Hilfe der Konversionsmöglichkeit nach den Regeln

- bei gleichen Templateparametern nehme den ersten,
- bei Konvertierbarkeit in eine Richtung nehme den Typ, in den konvertiert werden kann (beide Richtungen sind zu testen),
- bei beidseitiger Konvertierbarkeit nehme den größeren Datentyp.

den passenden Rückgabetyp heraus.

```
template <typename A, typename B> class Supertype {
private:
  template <class U, class V, int, int, int>
     struct tester;

  template <class U, class V>
     struct tester<U,V,1,1,1>{ typedef U stype; };

  template <class U, class V>
     struct tester<U,V,0,1,0>{ typedef V stype;};

  template <class U, class V>
     struct tester<U,V,0,0,1>{ typedef U stype; };

  template <class U, class V>
     struct tester<U,V,0,1,1>{
        template <class UU, class VV, int>
           struct tester2{  typedef VV stype;  };

        template <class UU, class VV>
           struct tester2<UU,VV,1>{typedef UU stype;};

        typedef typename
      tester2<U,V,
        FirstConvertsToSecond<U,V>::bigger>::stype
            stype;
     };
```

```
public:
     typedef typename tester<A,B,
         FirstConvertsToSecond<A,B>::same_type,
         FirstConvertsToSecond<A,B>::yes,
         FirstConvertsToSecond<B,A>::yes>::stype
               value_type;
};//end struct
```

Die ersten beiden Regeln sind recht einfach zu überprüfen (drei Spezialisierungen der internen Klasse `tester`), die dritte macht etwas mehr Aufwand, lässt sich aber rekursiv mit der internen Klasse regeln. Bei Typunverträglichkeit endet der Versuch mit einem Compilerfehler.

Wenn Sie sich nun die Ausdrucksklassen und die Operatorenklassen anschauen, werden Sie feststellen, dass hier immer noch eine große Lücke klafft, denn die Templateparameter der Ausdrucksklasse sind nicht für das Einsetzen in die Operatorenklasse geeignet, sondern hierfür werden die inneren Typen der Templateparameter benötigt.

9.2.3 Gerüste für binäre und unäre Ausdrücke

Die Ausdrucksklassen müssen die Operanden als Attribute aufnehmen, wie wir oben schon festgestellt haben. Dabei müssen wir den Attributen allerdings die Eigenschaft `mutable` geben, weil bei der Auflösung der Operatoren `const`-Operanden auftreten und die `const`-Eigenschaft auch bei der Auflösung durch den Compiler erhalten bleibt. Andererseits sind die Objekte aber während der Ausführung alles andere als `const`, sondern werden (abhängig von ihrem Typ) laufend mit neuen Werten versorgt, wie wir gleich begründen werden und was durch die Deklaration `mutable` berücksichtigt ist.

```
template <class T1 , class T2 , class op>
struct BinaryExpression: publicNonLiteral {
private:
     mutable typename
        LiteralDiscriminator<T1>::variable_type v1;
     mutable typename
        LiteralDiscriminator<T2>::variable_type v2;
```

Die korrekte Ermittlung des Typs ist nicht ganz so trivial, wie man das im ersten Moment vielleicht meint, was durch den Analysetyp `LiteralDiscriminator` angedeutet wird. Wenn man sich die Art der Ausdrücke in der Einführung noch einmal anschaut, können durch die Operatoren nämlich Konstante, Variable, Funktionen oder Ausdrücke miteinander verknüpft werden, die in der Rechnung auch korrekt bedient werden müssen. Deshalb erbt der Ausdruckstyp auch von einem speziellen Basistyp `NonLiteral`. Wir klären diesen Sachverhalt im nächsten Teilkapitel detailliert auf.

Für die Operatorauflösung (Supertyp) werden noch die (internen) Werttypen benötigt, die wir ebenfalls vom LiteralDiscriminator liefern lassen. BinaryExpression setzt sich also wie folgt fort:

```
      typedef typename
       LiteralDiscriminator <T1>::value_type
            value_type_1;
      typedef typename
         LiteralDiscriminator<T2>::value_type
            value_type_2;
      typedef op operator_var;
public:
      typedef typename
         Supertype<value_type_1,value_type_2>::value_type
            value_type;
```

An Methoden sehen wir neben dem Konstruktor eine Initialisierung, eine Berechnung und eine Inkrementierung vor, was uns formal zu Konstruktionen mit einer for-Schleife in die Lage versetzt, die in der Einleitung beschrieben wurden. Die Inkrementierungsmöglichkeit erfordert nun auch zwingend die mutable-Deklaration. Den genauen Zusammenhang klären wir ebenfalls etwas später auf.

```
      BinaryExpression(T1 const& t1, T2 const& t2):
               v1(t1),v2(t2){}

      void initialise()const{
            v1.initialise();
            v2.initialise();
      }//endfunction

      value_type const& value()const{
            return
             operator_var::eval(v1.value(),v2.value());
      }//end function

      void inkrement()const{
            v1.inkrement();
            v2.inkrement();
      }//end function

};//end class
```

Aufgabe. Implementieren Sie in gleicher Weise UnaryExpression.

9.3 Datenobjekte in den Ausdrücken

Kommen wir nun zu dem ominösen Typ LiteralDiscriminator. Die Ausdrucksobjekte am Ende der rekursiven Zerlegung enthalten nun eine Reihe unterschiedlicher konkreter Datenobjekte, als da wären:

(a) **Felder**. Auf Felder wird bei der Auswertung des Ausdrucks elementweise über einen Index oder – allgemeiner und auf alle Feldtypen anwendbar – sequentiell über einen Iterator zugegriffen.
(b) **Variablen**. Variablen werden bei der Auswertung verschiedene Werte zugewiesen, wobei sich die Werte in der Regel systematisch ändern.
(c) **Konstante**. Konstante behalten ihren Wert während der gesamten Auswertung bei.
(d) **Funktionen**. Funktionen wie beispielsweise $sin(x)$ enthalten Variablen, denen wiederum wie in (b) systematisch verschiedene Werte zugewiesen werden.

Für diese Datenobjekte sind spezielle Typisierungen notwendig, um sie in Ausdrücken korrekt verwenden zu können. Beliebige Klassen werden die Methoden `initialise`, `value` und `inkrement` in der Regel nicht unterstützen. Wir werden daher ein Feld

```
double x[100];
```

nicht einfach in dieser Form in einen Ausdruck einsetzen können, sondern es mit einem Rahmen umgeben müssen, der die fehlenden Methoden implementiert. Das ist an sich noch nicht besonders gravierend, wie wir gleich nachweisen werden, jedoch existieren einige grundsätzliche Unterschiede zwischen Variablen und Konstanten in Ausdrücken, deren Rahmenerzeugung wir zusätzlich mittels der Klasse `LiteralDiskriminator` automatisieren. Hier kommt dann auch der schon erwähnte Basistyp `NonLiteral` nebst seinem Gegenstück `Literal` zum Einsatz.[4]

9.3.1 Felder

Für Felder folgt die Notwendigkeit einer speziellen Typisierung in Ausdrücken bereits durch die Spezifizierung des gewünschten Implementationsziels, denn die Felder sollen ja elementweise und nicht als komplettes Objekt behandelt werden. Dies lässt sich nur durch ein Überschreiben des Zuweisungsoperators erreichen. Wir definieren einen Feldtyp durch

```
template <class T>
class array: public T, public NonLiteral {
public:
     ....
  template <class T1, class T2, class op>
```

[4] Bei der Benennung der Typen folge ich hier dem in der Literatur beschriebenen Ansatz, der hier **Literal** und nicht **Constant** vorsieht. Da die Typen nur intern zum Einsatz kommen, ist eine Verwechslung mit reservierten Wörtern oder ähnlichem eigentlich weniger zu befürchten. Vielleicht ist es für den einen oder andern von Ihnen ganz reizvoll, dieser Begriffsbildung einmal aus sprachlich-philosophischer Sicht nachzugehen.

```
  array& operator=(BinaryExpression<T1,T2,op>
                          const& x){
     iterator itt;
     x.initialise();
     for(itt=T::begin();itt!= T::end();++itt){
           *itt=x.value();
           x.inkrement();
        }//endfor
        return *this;
     }//end function
```

Der Templateparameter ist das eigentliche Feld, wobei es sich allerdings um ein echtes Feld oder besser um einen Container handeln muss und nicht um den Grundtypen des Feldes. Der elementweise Zugriff wird (in der üblichen Art) mit Iteratoren realisiert, die gesamte Rechnung nun tatsächlich elementweise innerhalb des Zuweisungsoperators ausgeführt.

> **Aufgabe.** Der Zuweisungsoperator ist für ein Objekt eines binären Ausdrucks definiert. Definieren Sie den korrespondierenden Ausdrück für unäre Ausdrücke auf der rechten Seite der Zuweisung.

Wenn Objekte des Typs `array` auf der rechten Seite des Zuweisungsoperators als Attribute von binären oder unären Ausdrucksobjekten auftreten, sind die drei fehlenden Methoden `initialise`, `value` und `inkrement` zu definieren.

```
     void initialise() const  {it=T::begin();}
     value_type const& value() const { return *it;}
     void inkrement() const   {++it;}
private:
     mutable const_iterator it;
};//end class
```

Um somit ein Feld in einem Ausdruck verwenden zu können, müssen wir es folgendermaßen deklarieren:

```
array<vector<int> > A,B,C,D;
```

Da zwischen `array` und `vector` ein simples Vererbungsverhältnis besteht, können sämtliche Containerfunktionen außer den speziell überschriebenen außerhalb von Ausdrücken völlig normal verwendet werden.

9.3.2 Variablen

Variablen können von beliebigem Typ sein, so dass wir hier ebenfalls eine Template-Definition benötigen, die die fehlenden Methoden `initialise`, `value` und `inkrement` definieren (einen Zuweisungsoperator benötigen wir hier nicht.

Warum?). Auch hier wird die Elternklasse `NonLiteral` verwendet, auf die wir später zurückkommen

```
template <class T> struct Variable: public NonLiteral{
  Variable<T>& init(T const& start,
                            T const& delta)
     {t=start; dt=delta; return *this;}
  void  initialise()const{}
  value_type const& value()const {return t;}
  void  inkrement() const{t+=dt;}
private:
     mutable T t; T dt;
};//end class
```

Sie werden bemerkt haben, dass die Methode `initialise` in dieser Implementation gar nichts tut und die eigentliche Initialisierung durch die Methode `init` durchgeführt wird, mittels der der Startwert sowie das Inkrement festgelegt wird. Ein Zuweisungsoperator ist nicht vorgesehen, da Variablen nur einen Wert aufweisen und daher nicht, wie Felder, komplette Rechnungen kontrollieren können. Variablen können aber zusammen mit Feldern verwendet werden, wenn sie auf der rechten Seite der Zuweisung auftreten. Der folgende Ausdruck füllt ein Feld mit berechneten Werten.

```
array<vector<double> > v;
v.resize(20);
Variable<double> r1,r2;
v=r1.init(1.0,0.05)*r2.init(1.0,0.2);
```

Anmerkung. Abweichend von den Feldvariablen erbt eine normale Variable nicht von ihrem Grundtyp, weshalb der Zuweisungsoperator hier auch nicht definiert ist und eine Variable auch nicht als normale Typvariable verwendet werden kann. Bevor Sie sich Gedanken darüber machen, wie dies zu bewerkstelligen wäre, bedenken Sie, dass

- ein Erben von Grundtypen wie `int` usw. nicht möglich ist. Erbmechanismen greifen erst mit Klassen.
- auch die Feldtypen nicht von Grundtypen abstammen dürfen, sondern von Containertypen.

Für Variable in Ausdrücken sind daher spezielle Deklarationen notwendig.

9.3.3 Konstante

Konstante werden ähnlich wie Variable definiert, indem sie einem Attribut der Klasse Literal zugewiesen werden.

```
template <typename T> struct Literal {
     Literal(T const& value):t(value)         {}
private:
     T t;
};//end struct
```

Allerdings stoßen wir hier auf ein Problem (und kommen damit auch zu dem noch immer ominösen `NonLiteral`): Felder und Variable, die ohnehin als Datentypen definiert werden müssen, werden für die Verwendung in Ausdrücken lediglich etwas anders als besondere Datentypen im Deklarationsteil des Programmkodes aufgeführt, was für Konstante aber kontraproduktiv ist, denn diese sind in der Regel direkt in die Ausdrucksformeln einzutragen, wenn die beabsichtigte Einfachheit beibehalten werden soll. Da aber auch Konstante die allgemeinen Funktionen unterstützen müssen (indem sie nichts machen), müssen sie in ein `Literal`-Objekt überführt werden.

Hier kommt nun die Klasse `NonLiteral` zum Einsatz, die bei Feldern und Variablen als Elternklasse verwendet wurde.

```
struct NonLiteral {};
```

Diese erlaubt nämlich, zwischen Konstanten und direkt in Ausdrücken verwendbaren Objekten zu unterscheiden. Dazu müssen wir lediglich die Vererbungsverhältnisse überprüfen, was wir mit der bereits bekannten Klasse `FirstInheritsFromSecond` durchführen können (siehe Tylisten). Hier kommt nun auch endlich die Klasse `LiteralDiskriminator` ins Spiel, die einfach einen Typ durchreicht oder einen Literalobjekt erzeugt

```
template <typename T>
struct LiteralDiscriminator{
private:
  template <class U, int i> struct tester{
     typedef          U              variable_type;
     typedef typename U::value_type  value_type;
  };
  template <class U> struct tester<U,0>{
     typedef Literal<U> variable_type;
     typedef U          value_type;
  };
public:
  typedef typename
      tester<T,FirstInheritsFromSecond<T,NonLiteral>
                  ::yes>::variable_type  variable_type;
  typedef typename
      tester<T,FirstInheritsFromSecond<T,NonLiteral>
                  ::yes>::value_type     value_type;
};//end struct
```

Die Unterscheidung zwischen Literal- und Nichtliteralobjekt übernimmt eine private Subklasse mit Spezialisierung, die als Ergebnistyp entweder den Typ durchreicht, wenn von `NonLiteral` geerbt wird, oder einen Literaltyp definiert. Wir versehen die Klasse auch noch mit einer weiteren Typisierung, nämlich der Festlegung des Datentyps für die durchzuführende Operation. Dieser Datentyp erhält, wie in den C++ Standardbibliotheken üblich, den Namen `value_type`. Damit ist der Kreis geschlossen und die komplizierten Typisierungen geklärt

Aufgabe. Ergänzen Sie die fehlenden Methoden in der Klasse `Literal`.

9.3.4 Funktionsobjekte

Funktionen von anderen Objekttypen zu unterscheiden ist zwar möglich, stößt jedoch auf Probleme bei der Ermittlung des Rückgabetyps. Wir gehen daher ähnlich vor wie bei der Implementation von Variablen und definieren einen speziellen Funktionstyp

```
template <class T> struct Function: public NonLiteral{
     typedef T (*function1)(T) ;
     Function(function1 f):func1(f),active_func(0){}

     Function<T>& init(T const& start,
                       T const& delta)
        {t=start; dt=delta; return *this;}
     void initialise()const{}
     value_type const& value()const
        {return func1(t);}
     void inkrement() const {t+=dt;}
private:
     mutable T t;
     T dt;
     int active_func;
     function1 func1;
};//end class
```

Bei Bedarf können auch andere Funktionstypen implementiert werden. Die Deklaration und Bedienung erfolgt ähnlich einer Variablen:

```
Function<double> Sin(sin); ...
```

9.4 Ein Beispiel

Das folgende Beispiel kombiniert alle drei Datentypen. Zwei Felder werden mit Daten gefüllt und anschließend mit einer Konstanten, einer Variablen und einer Funktion (Sinusfunktion) kombiniert.

```
Array<vector<double> > A,B,C;
for(int i=0;i<5;i++){
      B.push_back(i/3.0);
      C.push_back(i/4.0);
}

Variable<double> x;
Function<double> Sin(sin);

Sin.init(1.0,0.1);
x.init(0.0,1.5);

A=B+Sin+1.0+x+C;
```

Das Feld `A` enthält anschließend die fünf berechneten Werte.

Aufgabe. Implementieren Sie das in der Einleitung angegebene Beispiel eines Integrals.