

Einführung in die Programmiersprache C++

... FÜR FORTGESCHRITTENE ...

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Gliederung

- 1.Einführung in C
- 2.Einführung in C++
- 3.C++ für Fortgeschrittene

3.1 Templates

- 3.1.1. Grundlagen
- 3.1.2. Traits
- 3.1.3. Funktoren
- 3.1.4. Template Meta Programming

3.2 STL

3.3 C++ Strings

Problem der Woche: Traits (diesmal auf Folien!)

Traits (1)

- ▶ Oft werden zur Laufzeit Informationen über den Typ mit dem eine Template-Klasse instanziiert wurde
- ▶ Beispiel: Dreieck mit generischem Punkttyp

```
template<typename VertexT>
class Triangle
{
public:
    VertexT a;
    VertexT b;
    VertexT c;

    float getArea();    // Get area
    short getColor();   // Get (grey) color value
};
```

Traits (2)

- ▶ `getArea()` ist klar, Punkte müssen Koordinaten haben
- ▶ `getColor()` soll eine Farbe zurückgeben.
- ▶ Solange die Punkte keine Farbinformationen haben, soll ein Default-Wert zurück gegeben werden
- ▶ Wenn die Punkte Farbinformationen haben, soll der Mittelwert der drei Farben geliefert werden?
- ▶ Wie kann ich erfahren, dass die Punkte Farbinformationen haben?
- ▶ Einfach probieren?

```
template<typename VertexT>
short Color<VertexT>::getColor()
{
    return (short)(a.color + b.color + c.color) / 3;
}
```

- ▶ Gefahr eines Compiler-Fehlers!

Traits (3)

- ▶ Lösung: Traits-Objekte
- ▶ (partielle) Template-Spezialisierungen, die Meta-Informationen über verschiedene Typen enthalten.
- ▶ Beispiel:

```
template<typename T>
struct VertexTraits
{
    static string name;
    static bool has_color;
    static bool has_normal;
    ...
};
```

- ▶ Für dieses Template werden eine Default-Implementierung und Spezialisierungen mit den benötigten Informationen angelegt

Traits (4)

- ▶ Beispiel für Default-Implementierung:

```
template<typename T>
struct VertexTraits
{
    static string name;
    static bool has_color;
    static bool has_normal;
    ...
};

template<typename T>
string VertexTraits<T>::name = "unknown";

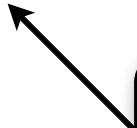
template<typename T>
bool VertexTraits<T>::has_color = false;

template<typename T>
bool VertexTraits<T>::has_normal = false;
```

Traits (5)

- Spezialisierung (benötigt eigene Deklaration):

```
template< >
struct VertexTraits<ColorVertex>
{
    static string name;
    static bool has_color;
    static bool has_normal;
    ...
};
```



```
class ColorVertex : public Vertex
{
public:
    short color;
}
```

- Spezialisierung (Implementierung in .cpp-Datei!):

```
string VertexTraits<ColorVertex>::name = "ColorVertex";
bool VertexTraits<ColorVertex>::has_color = true;
bool VertexTraits<ColorVertex>::has_normal = false;
```

Traits (6)

- Erweiterung für Triangle:

```
template<typename VertexT>
class Triangle
{
public:
    VertexT a;
    VertexT b;
    VertexT c;

    float getArea();    // Get area
    short getColor();  // Get (grey) color value

private:
    static VertexTraits<VertexT> traits;
};
```


Traits (7)

► Benutzung:

```
template<typename VertexT>
short Triangle<VertexT>::getColor()
{
    if(VertexTraits<VertexT>::has_color)
    {
        return (short) (a.color + b.color + c.color) / 3;
    }
    else
    {
        return 128;
    }
}
```

Funktoren (1)

- ▶ Wir wollen eine *flexible* Template-Funktion `sort ()` schreiben
- ▶ Problem: Die Funktion kennt den Anwendungskontext nicht
- ▶ Sortierreihenfolge und Sortierkriterium sollten steuerbar sein
- ▶ Eine allgemeine Implementierung scheint wegen der zweiten Forderung nicht möglich
- ▶ Die Eigenschaften sind für beliebige T nicht bekannt
- ▶ Beispiel:

```
template <class T>
void sort (T* first, int num);
```

```
class Student {
    string name;
    int semester;
    /* ... */
};
```

Funktoren (2)

- ▶ `sort()` braucht eigentlich nur eine Problemabhängige Vergleichsfunktion
- ▶ Idee: Vergleichsoperatoren überschreiben
- ▶ Löst hier das Problem nicht (wenn wir z.B. die Sortierreihenfolge ändern wollen)
- ▶ Lösung: Wir stellen einen so genannten Funktor zur Verfügung, der leicht ausgetauscht werden kann:

```
struct compName
{
    bool operator()(const Student& a, const Student& b)
    {
        return (a.name < b.name);
    }
};
```

Funktoren (3)

- ▶ Funktoren sind Klassen, die den ()-Operator implementieren
- ▶ Diese Klassen können als Parameter übergeben werden, um das Verhalten von Algorithmen zu beeinflussen
- ▶ Benutzung in der Version der `sort ()`-Funktion:

```
template <class T, class Comp>
void sort (T* begin, int num, Comp cmpFunc)
{
    ...
    if (cmpFunc(a, b)) ...
}
```

- ▶ Benutzung von `sort ()` mit verschiedenen Funktoren:

```
Student students[100];
// ...
sort(students, 100, compName); // sort by Name
sort(students, 100, compTime); // sort by Semester
```

Template Meta Programming (1)

- ▶ In C haben wir Makros benutzt, um bestimmte Funktionen zu definieren:

```
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

- ▶ Makros benutzen Mittel der Textersetzung
 - keine Informationen über Semantik
 - keine Typprüfung
- ▶ Makros ermöglichen optimierten Code, da Berechnungen bereits zur Compilezeit ausgeführt werden können
- ▶ Einige Tricks erlauben Berechnungen zur Compile-Zeit mit Templates durchzuführen
- ▶ „Template Meta Programming“

Template Meta Programming (2)

- ▶ Beispiel Fakultätsberechnung
- ▶ Rekursive Version:

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}

void foo()
{
    int x = factorial(4); // (4 * 3 * 2 * 1 * 1) = 24
    int y = factorial(0); // 0! = 1
}
```

- ▶ Nun wollen wir das mit Templates machen....

Template Meta Programming (3)

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};
```

- Verankerung mittels spezialisierten Templates

```
template <>
struct Factorial<0>
{
    enum { value = 1 };
};
```

- Benutzung:

```
void foo()
{
    int x = Factorial<4>::value; // == 24
    int y = Factorial<0>::value; // == 1
}
```

Template Meta Programming (4)

- ▶ Hier wird der Wert der Fakultät zur Compilezeit bestimmt und als Konstante in den Code eingefügt
- ▶ Durch den Template-Parameter wurden Fakultäten als `ints` festgelegt
- ▶ Vergleichsweise geringer Nutzen
- ▶ Weiteres Beispiel: Vektor-Addition

```
template <int dimension>
Vector<dimension>& Vector<dimension>::operator
+=(const Vector<dimension>& rhs)
{
    for (int i = 0; i < dimension; ++i)
        value[i] += rhs.value[i];
    return *this;
}
```


Template Meta Programming (5)

- ▶ Sobald der Compiler das Template instanziert, wird kann der Compiler den Code optimieren, da `dimension` eine Konstante ist
- ▶ Der Compiler sollte daher in der Lage sein, die `for`-Schleife zu vektorisieren:

```
template <>
Vector<2>& Vector<2>::operator+=(const Vector<2>& r)
{
    value[0] += r.value[0];
    value[1] += r.value[1];
    return *this;
}
```

- ▶ Aber es kommt noch besser...

Template Meta Programming (6)

- ▶ ... statischer Polymorphismus
- ▶ Wir kennen die Probleme mit den „Virtual Function Tables“
- ▶ Man kann dieses Problem umgehen
- ▶ „Curiously Recurring Template Pattern“ (CRTP)
- ▶ Beobachtung: Man kann den Typ einer abgeleiteten Klasse als Template-Parameter der Oberklasse benutzen.
- ▶ Wie kann das Aussehen?

Template Meta Programming (7)

► CRTP:

```
template <class Derived> struct Base
{
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }

    static void static_func() {
        Derived::static_sub_func();
    }
};

struct Derived : Base<Derived>{
    void implementation();
    static void static_sub_func();
};
```

Gliederung

- 1. Einführung in C
- 2. Einführung in C++
- 3. C++ für Fortgeschrittene
 - 3.1 Templates
 - 3.2 STL**
 - 3.2.1 Einleitung
 - 3.2.2 Container / Algorithmen / Iteratoren
 - 3.2.3 Funktoren

C++ Standard Template Library (1)

► Entstehung:

- 1971 erste Entwürfe generischer Bibliotheken (Dave Musser)
- 1979 erste Überlegungen von Alexander Stepanow
- Stepanow entwickelt bei HP erste Version von STL
- 1993 wurde die STL dem Standardisierungskomitee vorgestellt
- Basierend darauf wurden Teile der Bibliothek in die C++-Standardbibliothek übernommen

► Die C++-Standardbibliothek weicht in Teilen von der ursprünglichen Implementierung ab

► STL ist keine Untermenge der C++-Standardbibliothek

► Der Begriff wird aber häufig mit ihr in Zusammenhang gebracht

► Was ist die STL?

C++ Standard Template Library (2)

► Antwort von Wikipedia:

Ursprünglich wurde mit Standard Template Library eine in den 1980er Jahren bei Hewlett-Packard (kurz: HP) entwickelte, in C++ verfasste Bibliothek bezeichnet, die weitgehend auf generischer Programmierung mit dem Schwerpunkt Datenstrukturen und Algorithmen basierte. Diese Bibliothek beeinflusste maßgeblich die so genannte C++-Standardbibliothek, die heute fester Bestandteil der Programmiersprache C++ ist.

- Es gibt verschiedene Bibliotheken namens STL
- Wir werden uns um den Teil kümmern, der in die C++-Standardbibliothek eingegangen ist

C++ Standard Template Library (3)

► STL ist

- Generisch
 - Stark parametrisiert, viele Templates
- Enthält so genannte Container
 - Zusammenstellung von Objekten mit verschiedener Charakteristik
- Enthält Algorithmen
 - Zur Manipulation der in Containern gespeicherten Daten
- Enthält Iteratoren
 - Iteratoren sind eine Generalisierung von Pointern
 - Damit ist es möglich Algorithmen sauber von Containern zu trennen

► Erste Kategorie von Containern: Sequenzen

- Verwendung eines Index zum Zugriff
- Werte haben eine „Ordnung“

C++ Standard Template Library (4)

- ▶ Weitere Kategorien?
- ▶ Dazu später mehr!
- ▶ Beispiel:

```
vector<int> v(3);           // Vector of 3 elements
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];
```

- ▶ Oder:

```
vector<int> v;               // Without predetermined size
v.push_back(7);
v.push_back(v[0] + 3);
v.push_back(v[0] + v[1]);
```

- ▶ Nun möchten wir die Reihenfolge der drei Zahlen vertauschen

```
reverse(v.begin(), v.end());
```


C++ Standard Template Library (5)

- ▶ `vector<int>` ist ein generischer Container
- ▶ `reverse()` ist ein Algorithmus
- ▶ `reverse()` benutzt *Iteratoren*, die mit `v` assoziiert sind
- ▶ STL bietet generische Funktionen
- ▶ Parametrisiert über den Iterator-Typ, nicht über den Container
- ▶ Beispiel: Der `find()`-Algorithmus

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                  InputIterator last,
                  const T& value)
{
    while (first != last && *first != value) ++first;
    return first;
}
```

- ▶ Sucht nach `value` im Bereich `[first, last)`

C++ Standard Template Library (6)

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                  InputIterator last,
                  const T& value)
```

- ▶ InputIterator hat keinen speziellen Typ

```
while (first != last && *first != value) ++first;
```

- ▶ Nur die Unterstützung für * (Dereferenzieren), ++ (Inkrement) und == (Gleichheit) muss vorhanden sein!
- ▶ Pointer erfüllen diese Zwänge

```
int a[5] = { 1.1, 2.3, -4.7, 3.6, 5.2 };
int *pVal;
pVal = find(a, a + 5, 3.6); // Use int* as iterators
```

C++ Standard Template Library (7)

- ▶ Die Menge der benötigten Funktionalität für einen Typ wird auch Konzept genannt.
- ▶ Hier wird das Konzept „InputIterator“ genannt.
- ▶ Bei einem Typ, der die Bedingungen eines Konzepts erfüllt, spricht man auch davon, dass er das Konzept modelliert.
- ▶ Beispiel:
 - `int*` ist ein Model des `InputIterators`, weil `int*` alle Operationen zur Verfügung stellt, die der `InputIterator` benötigt.
- ▶ Der `reverse()`-Algorithmus benötigt mehr!
 - Die Iteratoren brauchen auch eine `---`-Operation!
 - Man Spricht vom „`BidirectionalIterator`“
 - Analog wie `InputIterator`, aber mit mehr „Requirements“

C++ Standard Template Library (8)

- ▶ Der `BidirectionalIterator` erweitert (engl.: *refines*) das Konzept des `InputIterators`
 - Dies ist genau wie die Klassen-Hierarchie
 - Andere Namensgebung, weil es sich nicht um Klassen handelt
- ▶ Leider wird das Konzept der „Konzepte“ nicht von der Programmiersprache unterstützt
 - Keine Unterstützung bei der Deklaration von Konzepten
 - Keine Unterstützung ob ein Typ ein Konzept modelliert
- ▶ Mehr Arbeit für den Programmierer / Herausforderung

C++ Standard Template Library (9)

- ▶ Trivial Iterator
 - Unterstützt dereferenzieren
 - Das war's schon! ... It's trivial ...
- ▶ Input Iterator
 - Nur Lese-Unterstützung wird zugesichert
 - Nur „Single Pass“ Unterstützung wird zugesichert
- ▶ Forward Iterator
 - Ähnlich wie InputIterator
 - Unterstützt auch „Multi Pass“
- ▶ Bidirectional Iterator
 - Unterstützt Dekrement
- ▶ Random Access Iterator
 - Unterstützt Schritte beliebiger Größe in beliebiger Richtung

STL - Funktionsobjekte (1)

- ▶ Funktionsobjekte sind all das, was wie eine Funktion aufgerufen werden kann:
 - Eine Verallgemeinerung einer Funktion
 - Ein Funktionspointer
 - Eine Instanz einer Klasse, die () überlädt
- ▶ Erlauben die Veränderung von Algorithmen, bzw. der Operationen, die Algorithmen ausführen
- ▶ Diese Dinge können STL übergeben werden
- ▶ Auch bekannt unter dem Namen „Functors“

STL - Funktionsobjekte (2)

- ▶ In der Standard-Bibliothek werden verschiedene Konzepte unterstützt
- ▶ Beispiele:

Konzept	Beispiel	Bemerkung
Generator	$f()$	keine Argumente
Unäre Funktion	$f(x)$	ein Argument
Binäre Funktion	$f(x, y)$	zwei Argumente
Prädikat	<code>bool f(x)</code>	ein Argument
Binäres Prädikat	<code>bool f(x, y)</code>	zwei Argumente

- ▶ Und noch viele weitere mehr...

STL - Funktionsobjekte (3)

- ▶ Beispiel: Erzeugen von 100 Zufallszahlen

```
vector<int> values(100);  
generate(values.begin(), values.end(), rand);
```

- ▶ Mit Funktoren kann man sich eigene Funktionen erstellen:

```
int randomColorValue() {  
    return rand() & 0x00FFFFFF;  
}
```

...

```
vector<int> randColors(10);  
generate(randColors.begin(),  
         randColors.end(),  
         randomColorValue);
```


STL - Funktionsobjekte (4)

- ▶ Funktor mit Zustand
- ▶ Beispiel: Gesucht ist die Summe von Elementen
- ▶ Eine Klasse mit überladenen () – Operatoren ist perfekt

```
struct adder : public unary_function<int, void>
{
    int sum;
    adder() : sum(0) { }
    void operator()(int x) { sum += x; }
};
```

- ▶ Anwendung des Funktors mit dem for_each-Algorithmus

```
adder result = for_each(values.begin(),
                        values.end(),
                        adder());
cout << "Sum is " << result.sum << endl;
```

STL - Funktionsobjekte (4)

- ▶ Weiteres Beispiel: Ausgabe von Zahlen
- ▶ Anwendung des `copy ()`-Funktors und des Output-Iterators

```
copy(values.begin(),  
      values.end(),  
      ostream_iterator<int>(cout, ", "));
```

- ▶ Bemerkung: Der Template-Parameter von `ostream_iterator` muss zu dem Element-Type der Kollektion passen!

STL - Containerklassen

- ▶ Bisher: Sequenzen (z.B. Vektoren, Listen, ...)
- ▶ Weitere Container Kategorien sind:
- ▶ Assoziative Container
 - set
 - map
 - multiset / multimap
 - hash_set / hash_map / hash_multiset / hash_multimap / hash
- ▶ String package (kommt gleich)
- ▶ ropes (skalierbare Strings)
- ▶ Container-Adapter
 - stack
 - queue
 - priority_queue
 - bitset

STL-Beispiel: set-Vereinigung

```
struct ltstr {
    bool operator()(const char* s1, const char* s2) const {
        return strcmp(s1, s2) < 0;
    }
};

int main()
{
    const int N = 6;
    const char* a[N] = {"isomer", "ephemeral", "prosaic",
                        "nugatory", "artichoke", "serif"};
    const char* b[N] = {"flat", "this", "artichoke",
                        "frigate", "prosaic", "isomer"};

    set<const char*, ltstr> A(a, a + N);
    set<const char*, ltstr> B(b, b + N);

    cout << "Union: ";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
              ostream_iterator<const char*>(cout, " "), ltstr());
    cout << endl;
}
```