ILV Datenstrukturen und Algorithmen

05: Templates

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

Kapitel

Templates

Motivation

```
1  double mean(double a, double b)
2  {
3    return (a+b)/2.0;
4  }
5  int mean(int a, int b)
6  {
7    return (a+b)/2.0;
8  }
9  Vector mean(const Vector& a, const Vector& b)
10 {
11   return (a+b)/2.0;
12 }
```

- Die Funktion `mean` berechnet das arithmetische Mittel.
- C++ erlaubt zwar die Überladung der Funktion `mean` für die unterschiedliche Datentypen, aber die Duplizierung von Code ist unschön.
- Idee: **generic** programming
- Implementiere `mean` allgemein, d.h. ohne den konkreten Datentyp zu spezifizieren.

Funktionen Templates

```
1  template<class T>
2  T mean(const T& a, const T& b)
3  {
4    return (a+b)/2;
5  }
6  void test()
7  {
8    int i = mean(1,2);
9    double f = mean(1.0, 2.3);
10   Vector c = mean(Vector(1,0), Vector(2,3));
11 }
```

- Anhand von `template<class T>` sagt man dem Compiler, dass man ein „function template“ definiert. (Oft auch Template Funktion genannt.)
- Der Compiler erstellt („instanziiert“) eine „echte“ Funktion und ersetzt dabei `T` durch den konkreten Typ, etwa `int`.
  - D.h. die Instantiierung findet zur compile time statt!
- Anstatt `class` kann auch `typename` verwendet werden.
  - Meist wird `class` verwendet. Trotzdem muss der konkrete Typ nicht zwingend ein Klassentyp sein.

Automatische Typ-Herleitung

```
1 template<class T>
2 T mean(const T& a, const T& b)
3 {
4   return (a+b)/2;
5 }
```

- Der konkrete Typ kann beim Aufruf explizit angegeben werden:
  - `mean<float>(1, 2);` wird nicht zu `int` instantiiert.
- Wird der Typ nicht angegeben, dann versucht der Compiler den Typ automatisch herzuleiten.
  - Bei `mean(1.2, 15)` sagt g++:
    `error: no matching function for call to 'mean(double, int)'`
  - `mean(1.2, 15.)` geht natürlich.

```
1 template<class A, class B, class C>  void func(A a, B b, C c);
2
3 func(1, 2.0, 'c');                   // deduce all types
4 func<>(1, 2.0, 'c');                 // like before
5 func<int, float>(1, 2.0, 'c');       // deduce only type C
```

Klassen Templates

```
1 template<class T>  class A
2 {
3   //Use T as a type...
4 };
5 template<class T>  struct Vector
6 {
7   T x;
8   T y;
9 };
```

- Beispiel: Vector mit `double`, `float` oder `int` Koordinaten wird gleich implementiert → Template!
- Beispiel: Dynamisches Array für Elemente von einem beliebigen Typ `T` → Template!
  Code demo...
- Für Klassen Templates wird keine automatische Herleitung der template Argumente vorgenommen. Die Template Argumente müssen immer angegeben werden:
  - `Vector<float> position;`

Member Funktionen von Klassen Templates

```
1  template<class T>
2  class Vector
3  {
4    private:
5      T x, y;
6    public:
7      Vector(T x, T y);
8      T length() const;
9  };
10
11 template<class T>
12 Vector<T>::Vector(T x, T y) :
13   x(x), y(y)
14 {
15 }
16
17 template<class T>
18 T Vector<T>::length() const
19 {
20   return x*x + y*y;
21 }
```

Source organization

Problem: Die Definition muss in allen translation units (`.cpp` files) bekannt sein, wo das Template verwendet wird, da dort das Template instanziiert wird.

- **Variante 1**: Trennung von Definition und Deklaration in `.cpp` und `.h` Dateien.
  - Theoretisch möglich durch das Keyword `export`.
  - Wird kaum von Compilern unterstützt:
    ```
    g++: warning: keyword ‘export’ not implemented, and will be ignored
    ```
  - Ist in C++11 wieder entfernt worden.
- **Variante 2**: Alles in die Header files.
  - Oder: Alle member functions gleich inline (in der Klasse) definieren und nicht nur deklarieren.
  - Auch Funktionen Templates in Header files geben.

Member Funktionen von Klassen Templates

- Variante 2 mit inline Definition der member functions:

```
1  template<class T>
2  class Vector
3  {
4    private:
5      T x, y;
6
7    public:
8
9      Vector(T x, T y) :
10       x(x), y(y)
11     {
12     }
13
14     T length() const
15     {
16       return x*x + y*y;
17     }
18 };
```

Vererbung von Klassen Templates

- Klassen und Klassen Templates können von Klassen oder Klassen Templates erben:

```
1  class A { };
2
3  template<class U>
4  class B : public A
5  {
6  };
7
8  template<class T>
9  class C : public B<T>
10 {
11 };
12
13 class D : public C<float>
14 {
15 };
```

Spezialisierung

- Man kann Templates (Funktionen und Klassen) für spezielle Typen extra definieren: „specialization“
  - Etwa um für bestimmte Typen effizienteren Code zu schreiben.
  - Oder um für bestimmte Typen einen anderen Code zu schreiben.
    `mean<char>('a', 'C')` könnte etwa `'C'` zu `'c'` ändern um `'b'` zu erhalten und nicht `'R'`.

```
1  template<class A, class B, class C>  void func(A a, B b, C c)
2  {
3    cout << "func<A, B, C>" << endl;
4  }
5
6  template<>  void func<char, int, double>(char a, int b, double c)
7  {
8    cout << "func<char, int, double>" << endl;
9  }
10
11 void test()
12 {
13   func(1, 1, 1);        // --> func<A, B, C>
14   func('a', 2, 3.0);    // --> func<char, int, double>
15 }
```

Partielle Spezialisierung

- Klassen Templates können auch partiell (teilweise) spezialisiert werden.
  - D.h., man kann ein oder mehrere Template Parameter spezialisieren.

```
1  template<class A, class B, class C>  class ClassX
2  {
3    public:
4      void func()
5      {
6        cout << "X<A, B, C>" << endl;
7      }
8  };
9  template<class A, class C>  class ClassX<A, int, C>
10 {
11   public:
12     void func()
13     {
14       cout << "X<A, int, C>" << endl;
15     }
16 };
17
18 ClassX<char, int, long>().func();     // --> X<A, int, C>
19 ClassX<char, char, long>().func();    // --> X<A, B, C>
```

Template Parameter

Es gibt drei Arten von Template Parametern:

- **Types:**
  - `template<class T, class U, typename V> class A {};`
- **Values:**
  Erlaubt sind integrale Typen (int, float, long, ...), Pointer, Referenzen und Enumerationen.
  - `template<int a, char* b, Vector& c> class A {};`
- **(Class Templates)**

Value Template Parameter

- Value Template Parameter müssen konstant sein.
- Man könnte hier z.B. eine partielle Spezialisierung für `dim=3` durchführen und etwa das Kreuzprodukt von zwei Vektoren implementieren.

```
1  template<class T, int dim>
2  class Vector
3  {
4      T coord[dim];
5  };
6
7  void test(int foo)
8  {
9    Vector<double, 3> v1;     // OK
10   Vector<double, foo> v2;   // Error
11 }
```

Default arguments

- Template Parameter von Klassen Templates können default Argumente haben:

  ```
  template<class T=double, int dim=2, T initial=T()> class Vector {};
  ```

  - Nur die am weitesten rechts stehenden template Parameter können default Argumente haben.
  - Function templates können keine default arguments haben.

```
1 template<class T=double, int dim=2>  class Vector {};
2
3 void test(int foo)
4 {
5   Vector v1; // two-dimensional double vector
6   Vector<float> v2; // two-dimensional float vector
7   Vector<float, 3> v2; // three-dimensional float vector
8 }
```

Templates vs. OOP

- OOP und Templates lösen manchmal ähnliche Probleme: Abstraktion von konkreten Datentypen.
  - Array, welches Elemente von „beliebigem“ Typ aufnimmt.
    `Array<T>::add(const T& a);`
  - Zeichnen von verschiedenen Formen:
    polymorphes `Shape::draw()`
- Vorteile Templates:
  - Keine Einführung von (unnötigen?) Vererbungshierarchien.
  - Kein Overhead durch Polymorphismus (Zeit, Speicher).
  - Funktioniert auch für Basistypen wie `int`.
  - Strenge Überprüfung des Typs möglich.
- Vorteile OOP:
  - Code ist meist lesbarer und einfacher zu programmieren.
  - Es wird nicht für jede Verwendung eines unterschiedlichen Typs eigener Code erzeugt (Speicher, kürzere Compile-Zeiten).

Templates vs. OOP

Wann verwende ich OOP, wann Templates?

- Sind Vererbungsbeziehungen im Sinne von „ist ein“ erkennbar, dann deutet dies auf OOP hin.
  - Beispiel: Zeichnen von verschiedenen Formen: `drawAll(shapeArray)` mit Klassen `Rectangle, Circle` welche von `Shape` abgeleitet sind. `Shape::draw()` ist polymorph.
- Ist auch die Verwendung von Basis-Datentypen denkbar, dann deutet dies auf Templates hin.
  - Beispiel: `Array<int>, mean(2.0, 5.6)`
- Abstraktion zur Laufzeit: OOP (Polymorphismus)
- Abstraktion zur Compile-time: eher templates.

```
1 // Abstraction at compile time: Array<Shape*>
2 void drawAll(Array<Shape*> shapes)
3 {
4   for(unsigned i=0; i< shapes.size(); i++)
5     // Abstraction at runtime: polymorphic Shape::draw()
6     shapes[i]->draw();
7 }
```