Templates

C++ ist eine streng typisierte Sprache → Funktionen oder Klassen für jeden Datentyp neu definieren.

Bei einem Template in C++ handelt es sich um einen Mechnismus einen Algorithmus so zu implementieren, das der verwendete Datentyp nicht festgelegt werden muss, oder aber automatisch bestimmt werden kann.
Die Idee ist, den Datentyp als Parameter an Funktionen oder Klassen zu übergeben oder anders ausgedrückt der Datentyp wird als Parameter für Funktionen/Methoden erlaubt..
Dies führt zur generischen Programmierung und spielt bei der automatischen Erzeugung von Quellcode eine grosse Rolle.
In C++ werden 2 neue keywords hinzugefügt: `Template` und `Typename`

Wir unterscheiden 2 Typen:

- Function Templates (Template-Funktionen)
- Class Templates (Template-Klassen)

Bei Templates wird zur Compile Zeit festgelegt, welche Funktion aufgerufen wird. Sowohl das Überladen von Funktionen als auch Templates sind Beispiele für Polymorphie in OOP.

Function – Template

- Function Template Synthax

Deklaration:

```
template <template parameter> function definition;
```

Type Definition: `typename T`

Funktionssyntax ausgedrückt in Template-Parametern `T`

Aufruf:

```
function <template parameter> (argumente) ;
```

- Beispiel

funcTemplate.cc

```
…….
template <typename T>
T add(T A, T B) { return (A + B); }
…….
int main() {
    int result1; double result2;
    // calling with int parameters
    result1 = add <int> (2, 3);
    // calling with double parameters
    result2 = add <double> (2.2, 3.3);
    return 0; }
```

Function – Template

- Überladene Funktionen

```
// test function overloading in C++
#include <iostream>
using namespace std;
void print(int j) {
   cout << "Write integer: " << j << endl;
}
void print(double  x) {
   cout << "Write double: " << x << endl;
}
void print(char* c) {
   cout << "Write character: " << c << endl;
}
int main()
{
   int k = 5 ; double f = 2.3 ;
   char c[256] = "Hi there" ;
   // Write print to print integer
   print(k);
   // Call print to print float
   print(f);
   // Call print to print character
   print(c);
   return 0;
}
```

- Template-Funktionen

```
// test function Template in C++
#include <iostream>
#include <string>
using namespace std;

template <typename T>
void print(T val) {
   cout << "Write value: " << val << endl;
}
int main()
{
   int k = 5 ; double f = 2.3 ;
   string v = "Hi there" ;
   // Write print to print integer
   print<int>(k);
   // Call print to print float
   print<double>(f);
   // Call print to print character
   print<string>(v);
   return 0;
}
```

Class - Template

- Class Template Synthax

Deklaration:

```
template <template parameter> class MyClass{class definition};
```

Type Definition: `typename T` / `class T`

Class Definition ausgedrückt in Template-Parametern `T`

Aufruf des Konstruktors:

```
MyClass <template parameter> (argumente) ;
```

- Beispiel

classTemplate.cc

```
…...
template <class T>
class stdOutputData {
   public:
      void print(T pr){cout << "Write template: "<< pr << endl;}
   private:
      T val;  };
int main() {
  stdOutputData <int> p;  stdOutputData <string> v;
  int k = 5 ; string c = "Hi there" ;
  p.print(k);  v.print(c);
…..
```

Instanzieren

keine Template Parameter

Aufruf der Memberfunktion

Class - Template

- Class Template Synthax

Deklaration: Ein Operator oder Methode ausserhalb der Klassendeklaration

```
template <typename T>
   MyClass<T>::operator < const MyClass<T> &a_> {definition;}
```

die Schlüsselworte `typename` oder `class` sind äquivalent

Deklaration mit mehreren Type Parametern und anderen Parametern

```
template <typename Ta ,typename Tb, int k >
MyClass<..>::operator(<const MyClass< Ta a_ , Tb b , int j> &a_)
      { definition ;}
```

- Beispiel

classTemplateArray.cc

```
…...
public:
     Array(int s);
     Array operator+(const Array& v) ;
…...
template <class T> Array<T>::Array(int s){ code }

template <class T>
    Array<T> Array<T>::operator+(const Array<T>& v) { code }
```

Class - Template

- Class Template Spezialisierung

Templates definieren die Funktion von Familien von Klassen oder Funktionen. Manchmal ist es notwendig, dass fuer spezielle `typename` eine besondere Funktionalität gebraucht wird. Dazu gibt es die Spezialisierung von Templates.

```
template <typename T, int Zeile, int Spalte>     // (a)
class matrix { matrix implementation };

template <typename T>                            // (b)
class matrix<T, 2, 2>{ …....};

template <>                                      // (c)
class matrix<int, 2, 2>{…...};
```

zunehmende Spezialisierung

Instanzierung

```
matrix<int, 2, 2> x1;          // class matrix<int, 2, 2>

matrix<double, 2, 2> x2;       // class matrix<T, 2, 2>

matrix<string, 4, 3> x3;       // class matrix<T, Line, Column>
```

`x1` verwendet vollständige Spezialisierung, `x2` teilweise Spezialisierung und `x3` das primäre Template. Das primäre Template führt zu einem Fehler, da `string` nicht implementiert ist.

Class - Template

- C++ Unterstützung durch die Standard Template Library (STL)
  mit vielen Klassen aus den Bereichen
  - Container und Iteratoren, Iteratortypen
  - Algorithmen und Arrays
  - Vector, Listen, map/multimap
  -…… → Für Anfänger schwierig zu verwenden

- Vorteile
  Templateklassen können mit Typen T instanziert werden und sie garantieren, dass alle gleichen Typen auch die gleichen bleiben.
  Werden während des Compilierens aufgelöst. → Geschwingkeitsvorteil
  Durch expliziten Gebrauch `stdOutputData <int> p;` wird Typ-Sicherheit erreicht

- Nachteile
  Hohe Compile-Time weil `template code` mehrfach übersetzt wird
  Fehlermeldungen sind fast immer undurchsichtig
  Fehler treten zur Laufzeit des Programms auf

Arbeitsvorschlag:

Implementieren Sie die Basisfunktionalität unserer Übungsklasse `FourVector` mit Hilfe von Templates.

FourVectorTemplate.cc