Templates

- Funktionstemplates
- Klassentemplates

Templates

Funktionstemplates

Mit **Schablonen** (engl. **templates**) können Funktionen mit parametrisierten Datentypen geschrieben werden.

Die allgemeine Form für Funktionstemplates ist

```
template < class TypBezeichner >
    Funktionsdefinition
```

(Statt `class` kann auch `typename` geschrieben werden.)

Beispiel:

```
template <class T>
void swap(T &a, T &b)
{
  T tmp = a;

  a = b;
  b = tmp;
}
```

Die Funktion swap verarbeitet Argumente jeden Typs, mit der Einschränkung, dass Variablen (oder Objekte) dieses Typs einander zugewiesen werden können und miteinander initialisiert werden können.

Funktionstemplates

Verwendung des Funktionstemplates:

```
int main()
{
  int a = 3, b = 16;
  double d = 3.14, e = 2.17;
  Person
    k("Karl", "Flanderstr. 37", "542644"),
    f("Frank", "Sichelstr. 17", "403223");

  swap(a, b);
  printf("a = %d, b = %d\n", a, b);

  swap(d, e);
  printf("d = %f, e = %f\n", d, e);

  swap(k, f);
  printf("k's name = %s, f's name = %s\n",
         k.getname(), f.getname());

  return 0;
}
```

Ein Template ist keine Funktionsdefinition im bisherigen Sinne, sondern eine Schablone, nach der der Compiler *erst bei Bedarf* eine Funktion zu einem konkreten Datentyp erzeugt.

Ähnlichkeit mit dem Makromechanismus (`#define`)

Templates

Klassentemplates

werden auch **parametrisierte Datentypen**
oder **generische Datentypen** genannt.

```
#include <cstdio>
#include <cstdlib>
using namespace std;

template <class T>
class Array
{
  public:
    // Konstruktoren, Destruktoren etc.
    virtual ~Array(void) { delete [] data; }
    Array(int sz = 10) { init(sz); }
    Array(Array<T> const &other);
    Array<T> const &operator=
      (Array<T> const &other);

    // Schnittstelle
    int size() const;
    T &operator[](int index);

  private:
    // Daten
    int n;
    T *data;
    // Initialisierungsfunktion
    void init(int sz);
};
```

Klassentemplates

```
template <class T>
void Array<T>::init(int sz)
{
  if (sz < 1)
  {
    fprintf(stderr,
            "Array: cannot create array of "
            "size < 1;\n requested was: %d\n",
            sz);
    exit(1);
  }
  n = sz;
  data = new T[n];
}

template <class T>
Array<T>::Array(Array<T> const &other)
{
  n = other.n;
  data = new T[n];
  for (register int i = 0; i < n; i++)
    data[i] = other.data[i];
}
```

Klassentemplates

```
template <class T>
Array<T> const &Array<T>::operator=
  (Array<T> const &other)
{
  if (this != &other)
  {
    delete [] data;
    n = other.n;
    data = new T[n];
    for (register int i = 0; i < n; i++)
      data[i] = other.data[i];
  }
  return *this;
}

template <class T>
int Array<T>::size() const
{
  return n;
}
```

Klassentemplates

```
template <class T>
T &Array<T>::operator[](int index)
{
  if (index < 0 || index >= n)
  {
    fprintf(stderr,
            "Array: index out of bounds, "
            "must be between 0 and %d;\n"
            "requested was: %d\n",
            n - 1,
            index);
    exit(1);
  }
  return data[index];
}
```

Klassentemplates

Verwendung des Klassentemplates:

```
#include <cstdio>
using namespace std;

#include "array.h"

#define PI 3.1415

int main()
{
  Array<int> intarr;
  register int i;

  for (i = 0; i < intarr.size(); i++)
    intarr[i] = i << 2;

  Array<double> doublearr;

  for (i = 0; i < doublearr.size(); i++)
    doublearr[i] = PI * (i + 1);

  for (i = 0; i < intarr.size(); i++)
    printf("intarr[%d]   : %d\n"
           "doublearr[%d]: %f\n",
           i, intarr[i],
           i, doublearr[i]);

  return 0;
}
```

Klassentemplates

```
#include <cstdio>
using namespace std;

#include "person.h"
#include "array.h"

int main()
{
  // Ein Array, das zwei Personen enthält.
  Array<Person> staff(2);

  Person one, two;

  // Hier fehlt noch der Code für die
  // Zuweisung von Namen, Adressen und
  // Telefonnummern.

  staff[0] = one;
  staff[1] = two;

  printf("%s\n%s\n",
         staff[0].getname(),
         staff[1].getname());

  return 0;
}
```