

2. C++ vertieft (II): Templates

Was lernen wir heute?

- Templates von Klassen
- Funktionentemplates
- Smart Pointers

Motivation

Ziel: generische Vektor-Klasse und Funktionalität.

```
Vector<double> vd(10);  
Vector<int> vi(10);  
Vector<char> vi(20);  
  
auto nd = vd * vd; // norm (vector of double)  
auto ni = vi * vi; // norm (vector of int)
```

Typen als Template Parameter

1. Ersetze in der konkreten Implementation einer Klasse den Typ, der generisch werden soll (beim Vektor: **double**) durch einen Stellvertreter, z.B. **T**.
2. Stelle der Klasse das Konstrukt **template<typename T>** voran (ersetze **T** ggfs. durch den Stellvertreter)..

Das Konstrukt **template<typename T>** kann gelesen werden als **“für alle Typen T”**.

Typen als Template Parameter

```
template <typename ElementType>
class Vector{
    std::size_t size;
    ElementType* elem;
public:
    ...
    Vector(std::size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator[] (std::size_t pos){
        return elem[pos];
    }
    ...
}
```

Template Instanziierung

`Vector<typeName>` erzeugt Typinstanz von `Vector` mit `ElementType=typeName`.

Bezeichnung: **Instanziierung**.

```
Vector<double> x;           // vector of double
Vector<int> y;              // vector of int
Vector<Vector<double>> x;   // vector of vector of double
```

Type-checking

Templates sind weitgehend Ersetzungsregeln zur Instanziierungszeit und während der Kompilation. Es wird immer so wenig geprüft wie nötig und so viel wie möglich.

Beispiel

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){
        return left < right ? left : right;
    }
};

Pair<int> a(10,20); // ok
auto m = a.min(); // ok
Pair<Pair<int>> b(a,Pair<int>(20,30)); // ok
auto n = b.min(); // no match for operator<!
```


Generische Programmierung

Generische Komponenten sollten eher als **Generalisierung eines oder mehrerer Beispiele** entwickelt werden als durch Ableitung von Grundprinzipien.

```
template <typename T>
class Vector{
public:
    Vector();
    Vector(std::size_t);
    ~Vector();
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    Vector (Vector&&);
    Vector& operator=(Vector&&);
    const T& operator[] (std::size_t) const;
    T& operator[] (std::size_t);
    std::size_t size() const;
    T* begin();
    T* end();
    const T* begin() const;
    const T* end() const;
}
```

Funktionentemplates

1. Ersetze in der konkreten Implementation einer Funktion den Typ, der generisch werden soll durch einen Namen, z.B. **T**,
2. Stelle der Funktion das Konstrukt **template<typename T>** voran (ersetze **T** ggfs. durch den gewählten Namen).

Funktionentemplates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Typen der Aufrufparameter determinieren die Version der Funktion, welche (kompiliert und) verwendet wird:

```
int x=5;
int y=6;
swap(x,y); // calls swap with T=int
```

Sicherheiten

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Eine unverträgliche Version der Funktion wird nicht erzeugt:

```
int x=5;
double y=6;
swap(x,y); // error: no matching function for ...
```

.. auch mit Operatoren

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){ return left < right? left: right; }
    std::ostream& print (std::ostream& os) const{
        return os << "("<< left << "," << right<< ")";
    }
};
```

```
Pair<int> a(10,20); // ok
std::cout << a; // ok
```

```
template <typename T>
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){
    return pair.print(os);
}
```

Praktisch!

```
// Output of an arbitrary container
template <typename T>
void output(const T& t){
    for (auto x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

int main(){
    std::vector<int> v={1,2,3};
    output(v); // 1 2 3
}
```

Explizite Typangabe

```
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
    T left;
    T right;
    std::cin << left << right;
    return Pair<T>(left,right);
}
...
```

```
auto p = read<double>();
```

Wenn der Typ bei der Instanzierung nicht inferiert werden kann, muss er explizit angegeben werden.

Mächtig!

```
template <typename T> // square number
T sq(T x){
    return x*x;
}

template <typename Container, typename F>
void apply(Container& c, F f){ // x <- f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}

int main(){
    std::vector<int> v={1,2,3};
    apply(v,sq<int>);
    output(v); // 1 4 9
}
```


Nochmal Speichermanagement

Richtlinie “Dynamischer Speicher”

Zu jedem **new** gibt es ein passendes **delete**!

Vermeide:

- *Speicherlecks*: “alte” Objekte, die den Speicher blockieren
- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit **delete**.

Wie?

Smart Pointers

- Können sicherstellen, dass ein Objekt gelöscht wird genau dann, wenn es nicht mehr genutzt wird
- Basieren auf dem RAII (Resource Acquisition is Initialization) Paradigma.
- Können an die Stelle jedes gewöhnlichen Pointers treten: sind als Klassentemplates implementiert.
- Es gibt `std::unique_ptr<>`, `std::shared_ptr<>` (und `std::weak_ptr<>`)

```
std::unique_ptr<Node> nodeU(new Node()); // unique pointer  
std::shared_ptr<Node> nodeS(new Node()); // shared pointer
```

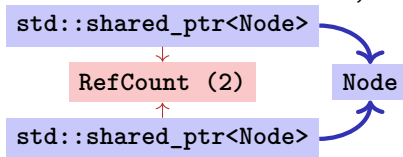
Unique Pointer

- Der Dekonstruktor von `std::unique_ptr<T>` löscht den enthaltenen Zeiger.
- `std::unique_ptr<T>` hat exklusiv Zugriff auf den enthaltenen Zeiger auf `T`.
- Kopierkonstruktor und Assignment Operator sind gelöscht. Ein Unique Pointer kann nicht als Wert kopiert werden. Movekonstruktor ist vorhanden: der Zeiger kann verschoben werden.
- Kein Zusatzaufwand zur Laufzeit im Vergleich zu einem normalen Zeiger.

```
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::unique_ptr<Node> node2 = std::move(nodeU); // ok
std::unique_ptr<Node> node3 = nodeU; // error
```

Shared Pointer

- `std::shared_ptr<T>` zählt die Anzahl von Besitzern eines Zeigers (Referenzzähler). Wenn der Referenzzähler auf 0 fällt, wird der Zeiger gelöscht.
- Shared Pointers können kopiert werden.
- Shared Pointers haben zusätzlichen Speicher- und Laufzeitbedarf: sie verwalten den Referenzzähler zur Laufzeit und enthalten jeweils einen Zeiger auf den Referenzzähler.



Shared Pointer

```
std::shared_ptr<Node> nodeS(new Node()); // shared pointer, rc = 1  
std::shared_ptr<Node> node2 = std::move(nodeS); // ok, rc unchanged  
std::shared_ptr<Node> node3 = node2; // ok, rc = 2
```

Smart Pointers

Einige Regeln

- Niemals **delete** auf einen Zeiger im Smart Pointer aufrufen.
- **new** vermeiden, stattdessen:

```
std::unique_ptr<Node> nodeU = std::make_unique<Node>()  
std::shared_ptr<Node> nodeS = std::make_shared<Node>()
```

- Wo möglich, **std::unique_ptr** verwenden.
- Bei der Verwendung von **std::shared_ptr** sicherstellen, dass es keine Zyklen im Zeigergraphen gibt.

Übung heute

- Implementieren Sie einen **generischen** dynamischen Vektor, der Indexoperatoren anbietet, über effizientes Speichermanagement verfügt und **Iteration** unterstützt.
- Wandeln Sie einen Sortieralgorithmus auf Indizes eines int-Vektors in einen **generischen Sortieralgorithmus auf einem Iterator** um.