Programmieren / Algorithmen & Datenstrukturen 2

Templates

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Templates

Inhalt.

- Templates
- Abgeleitete Klassen
- Testgetriebene Programmierung
- Container, Iteratoren und Algorithmen der StdLib
- Fortgeschrittenes Suchen
- Fortgeschrittenes Sortieren
- Grafische Benutzeroberflächen

Die Mythen über das Programmieren

Sie haben inzwischen erkannt, dass die folgenden Vorurteile nicht stimmen.

- Programmieren setzt eine besonders seltene Begabung voraus.
- Programme werden von sozial auffälligen Männern in völliger Einsamkeit und vorwiegend nachts geschrieben.
- Programmieren ist nichts Ernstes, es geht vor allem um irgendwelche Zahlenspielereien.
- Programme haben keine reale Bedeutung für die Gesellschaft.
- Programmieren setzt höhere Mathematik voraus.
- Die Denkweise beim Programmieren unterscheidet sich völlig vom alltäglichen Denken.

Stimmt nicht!

Hardware und Software

Wie kann man die von der Hardware bereit gestellten Grundlagen durch Software möglichst sicher und produktiv nutzbar machen?

- Die Hardware stellt uns wenige Grundoperationen und Speicherplatz / Adressen zur Verfügung.
  - "Low Level" Operationen, Speicherzugriff mit Zeiger und Datenfeld (Array).
  - Keine Typen, nur Bits/Bytes ohne Überprüfung des Zugriffs.
  - Entweder feste Größen oder fehlerträchtige Speicherverwaltung im Heap.
  - Performance so schnell, wie es die Hardware hergibt.
- Produktive Programmierer brauchen mehr.
  - Operationen "höherer" Art.
  - Typen, wie z.B. für Zeichenketten, einschl. Überprüfungen für die Operationen.
  - Einfaches Hinzufügen/Entfernen von Elementen zur Laufzeit.
  - Performance fast so schnell wie die Hardware.
- Container-Typen wie `vector<T>` aus der StdLib sind hier äußerst nützlich.
  - Wie werden solche Sprachmittel implementiert?
  - Die Techniken, mit denen wir unseren `myVector` Container von Grund auf konstruiert haben, und die wir im Folgenden weiter ausbauen, gehören zu allen Programmierarbeiten mit höheren Datenstrukturen.

Der `void*`

Ein `void*` ist das C++ Sprachelement, das einer bloßen Maschinenadresse am nächsten kommt.

- Ein `void*` ist ein Zeiger auf einen Speicherbereich, dessen Typ der Compiler nicht kennt.
- `void` ist *kein* Typ, es gibt *keine* "`void` Objekte".
  - `void v;   // Fehler`
  - `void f(); // f() hat keinen Rueckgabetyp`
`// f() gibt nicht "ein void Objekt" zurueck`
- An einen `void*` kann jeder Zeiger auf ein beliebiges Objekt zugewiesen werden.
  - `int* pi    { new int{} };`
  - `double* pd { new double[10]{} };`
  - `void* pv1  { pi };`
  - `void* pv2  { pd };`

Der `void*`

Um einen `void*` zu verwenden muss man dem Compiler mitteilen, worauf er zeigt.

```
void f( void* pv ) {
  void* pv2{ pv };  // kopieren geht (void* sind zum Kopieren da)
  double* pd{ pv }; // Fehler: keine Umwandlung fuer void* moeglich
  *pv = 7;          // Fehler: keine Inhaltsoperation fuer void*
  pv[2] = 9;        // Fehler: keine Indexoperation fuer void*
  pv++;             // Fehler: keine Inkrementierung fuer void*

  int* pi { static_cast<int*>( pv ) };  // explizite Umwandlung geht
   // …
}
```

- `static_cast` kann verwendet werden, um einen (beliebigen) Zeiger explizit auf einen Typ umzuwandeln.
  - "static_cast" ist ein absichtlich so hässlich gewählter Name für eine gefährliche Operation.
  - Verwenden Sie `static_cast` nur, wenn es nicht anders geht.
  - `static_cast` ist i.Allg. als Templatefunktion definiert.

Explizite Typumwandlungen

Diese Sprachmittel sind nur nach gründlichen Überlegungen einzusetzen, da sie die Typsicherheit drastisch verletzen können, d.h. sehr fehleranfällig sind.

- **`static_cast<T>(a)`** konvertiert `a` in den Typ `T`,
  - wird zur Build-Time ausgewertet, und funktioniert für alle vorgesehenen Typumwandlungen.
- `reinterpret_cast<T>(a)` konvertiert `a` in den Typ `T`,
  - wie `static_cast`, setzt aber Typprüfungen weitestgehend außer Kraft und konvertiert im Prinzip beliebig (d.h. gleiches Bitmuster).
  - Der Compiler geht davon aus, dass Sie hier genau wissen, was Sie tun...
- `const_cast<T>(a)` entfernt (oder ergänzt) `const`.
  - Der Compiler geht davon aus, dass Sie auch hier genau wissen, was Sie tun...
- `T(a)` sog. *funktionaler Cast* , konvertiert `a` in den Typ `T`, wenn es irgendwie durch `static_cast` oder `reinterpret_cast`, und in Kombination mit `const_cast` geht.
- `(T)a` sog. *C-Cast*, konvertiert wird wie beim funktionalen Cast.

Wiederholung / Vertiefung: Datenfelder (Arrays)

Datenfelder müssen nicht im Heap sein.

- Ein Datenfeld ist eine Folge von Objekten des selben Typs, die in einem zusammenhängenden Speicherbereich liegen.
- Zwischen den Objekten im Speicher gibt es keine Lücken, so dass man die Elemente beginnend mit 0 durchnummerieren kann.
  - Deklaration durch quadratische Klammern: `int ai[100]{};`
  - Elementzugriff durch Indexoperator oder Zeigerarithmetik (*ohne* Bereichsüberprüfung) möglich:
    ```
    int i1 { ai[0] };
    int i100 { *(ai+99) };
    ```
- Dynamische Datenfeld-Längen mit zu-/abnehmender Elementzahl sind (nur) mittels `new` / `delete` möglich.

```
char ac[7]{};   // globales Datenfeld im statischen Datenspeicher,
                // Adressbereich liegt meist zwischen Heap und Maschinencode

int f( int n ) {
   char lc[20]{}; // lokales Datenfeld, verschwindet mit f() wieder aus dem Stack
   double lx[n];  // Fehler: die Anzahl n ist zur Build-Time unbekannt
                  // Bessere (und klare) Alternative: std::vector<double> lx(n);
}
```

Wiederholung / Vertiefung: Adressoperator &

Zeiger können auf beliebige Objekte verweisen, nicht nur auf Objekte, die im Heap liegen.

```
int a{}; char ac[20]{}; // im globalen Scope
void f( int n ) {
    int b{};
    int* p{&b};      // p zeigt auf die lokale Variable namens b
    p = &a;          // p zeigt auf die globale Variable namens a
    char* pc{ac};    // Array-Namen sind Zeiger auf ihr erstes Element
    pc = &ac[0];     // gleichbedeutend mit pc = ac
    pc = &ac[n];     // zeigt auf das n-te Element, es erfolgt
                     // keine Pruefung der Zugriffsgrenzen
    // …
}
```

- Datenfeld-Namen werden in C++ "beim kleinsten Anlass" implizit in Zeiger auf ihr erstes Element umgewandelt, Beispiele:

```
char ch[100]{}; // sizeof( ch ) ist 100
char* p{ch};    // sizeof( p ) ist z.B. 4, denn p wird mit &ch[0] initialisiert
void f( char* cp ) {/*…*/ }; f( ch ); // &ch[0] wird uebergeben
```

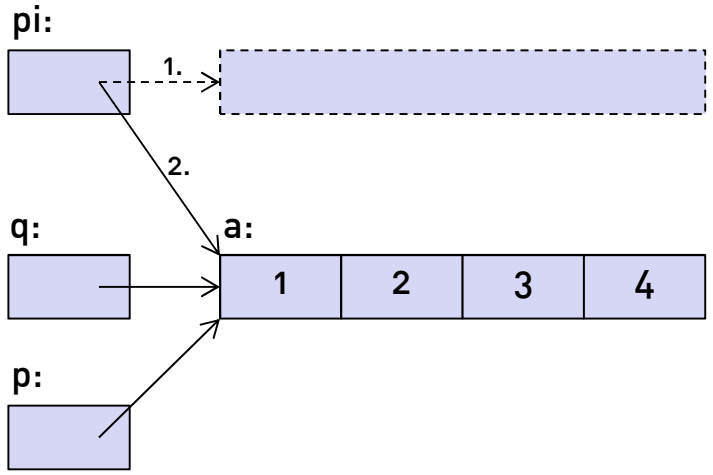
  - Das Verhalten beim Funktionsaufruf ist hier überraschend, weil in jedem andern Fall, in dem ein Objekt übergeben und nicht ausdrücklich anders deklariert wird, das Argument als Referenz übergeben wird (call-by-value Prinzip)…

Wiederholung / Vertiefung: Datenfelder und Zeiger

Datenfelder werden als Parameter in Funktionsaufrufen *nicht* kopiert (kein pass-by-value), sondern es wird ein Zeiger übergeben (pass-by-reference).

```
01  void f( int pi[] ) { // gleichbedeutend mit void f( int* pi )
02    int a[] { 1,2,3,4 };
03    //int b[] { a }; // Fehler: Datenfelder kennen keine Kopieroperation
04    pi = a;          // OK, ist aber keine Kopie: pi zeigt jetzt
05                     // auf das erste Element von a
06    int* p {a};      // p zeigt auf das erste Element von a
07    int* q {pi};     // q zeigt auf das erste Element von a
08  }
```

- *Aufgabe* :
  Könnte hier ein Speicher-problem entstehen?



Bildquelle: Stroustrup

Wiederholung / Vertiefung: Zeiger und Referenzen

Stellen Sie sich eine Referenz als alternativen Namen für ein Objekt oder als unveränderlichen, automatisch dereferenzierten Zeiger vor.

```
int a {10};
int* z {&a}; // der Adressoperator & gibt uns einen Zeiger auf a
*z = 7;      // Zuweisung an a durch z
             // Der Dereferenzierungsoperator * (oder []) gibt
             // uns Zugriff auf das, worauf der Zeiger verweist
int x1 {*z}; // lesender Zugriff auf a durch z

int& r {a};  // r ist ein Synonym für a
r = 9;       // Zuweisung an a durch r
int x2 {r};  // lesender Zugriff auf a durch r

z = &x1;     // ein Zeiger kann seinen Wert (eine Speicheradresse)
             // aendern, d.h. auf ein anderes Objekt verweisen
//r = &x1;   // Fehler: eine Referenz kann ihren Wert nicht aendern
```

Wiederholung / Vertiefung: Zeiger-/Arrayprobleme

Eine Zusammenfassung wichtiger Fallstricke bei der Arbeit mit Zeigern und Datenfeldern.

- Die meisten Probleme mit Zeigern und Datenfeldern sind darauf zurück zu führen, dass über Zeiger versucht wird, auf etwas zuzugreifen, das kein Objekt des erwarteten Typs ist:
  - Zugriff über einen Nullzeiger,
  - Zugriff über einen nicht initialisierten Zeiger,
  - Zugriff hinter das Ende (oder vor den Anfang) eines Datenfelds,
  - Zugriff auf ein gelöschtes Objekt,
  - Zugriff auf ein Objekt, dessen Gültigkeitsbereich bereits verlassen wurde.
- Der Zugriff selbst sieht im Quellcode fast immer völlig harmlos aus,
  - der Zeiger enthält aber keinen Wert, der einen *gültigen* Zugriff sicherstellt.
- Viele Programmierer unterschätzen diese Probleme,
  - obwohl deren unzählige Variationen und Kombinationen selbst erfahrene Profis immer wieder überraschen.
- Allgemeiner Tipp: den eigenen Quellcode nicht unnötig mit Zeigern und Datenfeldern verkomplizieren, ggf. auf Zeigerarithmetik komplett verzichten...

Wiederholung / Vertiefung: Zeiger-/Arrayprobleme

Eine Zusammenfassung wichtiger Fallstricke bei der Arbeit mit Zeigern und Datenfeldern.

- Zeiger initialisieren (wie überhaupt alle Variablen!)
  - ```
    int* ip; // jetzt werden wueste Fehler moeglich
    *ip = 7; // grober Fehler
    ```
  - Denken Sie vor allem daran, Zeiger zu initialisieren, die Member einer Klasse sind
- Die Objekte, auf die die Zeiger verweisen, initialisieren.
  - ```
    int* ip {new int};
    int i {*ip + 1}; // grober Fehler
    ```
  - Denken Sie immer daran, dass Sie es bei Zeigern mit *zwei* Speicherstellen zu tun haben.
- Nicht über Nullzeiger zugreifen.
  - ```
    int* ip{};
    *ip = 7; // grober Fehler
    ```
  - Denken Sie auch über die Verwendung von Referenzen und den Einsatz von Ausnahmen nach, um Nullzeiger-Zugriffe zu vermeiden.
    ```

Wiederholung / Vertiefung: Zeiger-/Arrayprobleme

Eine Zusammenfassung wichtiger Fallstricke bei der Arbeit mit Zeigern und Datenfeldern.

- Nur auf existierende Elemente eines Datenfelds zugreifen.
  - ```
    int a[100]{};
    int* ip {&a[100]}; // grober Lesefehler
    *ip = 42; // grober Schreibfehler
    ```
  - Achten Sie vor allem in Schleifen auf das erste und letzte Element.
  - Übergeben Sie möglichst mit einem Datenfeld zusätzlich seine Größe.
- Nicht über einen mit `delete` gelöschten Zeiger zugreifen.
  - ```
    int* piZwerge { new int{7} };
    //…
    delete piZwerge;
    //…
    *piZwerge = 6; // grober Fehler
    ```
  - In der Praxis das wohl kniffligste Problem, die sinnvollste Verteidigungsstrategie besteht oft darin, `new` und `delete` möglichst nur in Konstruktoren und Destruktoren zu verwenden.
  - Ggf. nach einem `delete` zur Sicherheit den Zeiger "nullen":
    ```
    delete piZwerge; piZwerge = nullptr;
    ```

Wiederholung / Vertiefung: Zeiger-/Arrayprobleme

Eine Zusammenfassung wichtiger Fallstricke bei der Arbeit mit Zeigern und Datenfeldern.

- Keine Zeiger auf lokale Variablen als Rückgabewert liefern.
  - ```
    int* f( ) {
        int x {42};
        //…
        return &x;
    }
    //…
    int* pi { f() }; // Fehler
    *pi = -1; // grober Fehler
    ```
  - Problem: beim Verlassen der Funktion `f()` wird der Speicherbereich der Funktion im Stack freigegeben, und der zurückgegebene Zeigerwert zeigt irgendwo wüst in den Stack.
  - Könnte im Prinzip vom Compiler abgefangen werden, doch die wenigsten Compiler unterstützen dies.
    ```

Wiederholung / Vertiefung: Zeiger-/Arrayprobleme

Eine Zusammenfassung wichtiger Fallstricke bei der Arbeit mit Zeigern und Datenfeldern.

- Ein weiteres, abschließendes Negativbeispiel:

```
char* f( ) {
   char ch[20]{};
   char* p {&ch[20]};
   *p = 'a';          // wuestes Schreiben in den Speicher hinter ch
   char* q;           // nicht initialisiert
   *q = 'b';          // wuestes Schreiben irgendwo in den Speicher
   return &ch[10];    // ch verschwindet aber mit f()
}

void g( ) {
   char* pp { f() }; // haengender Zeiger
   *pp = 'c';         // wuestes Schreiben irgendwo in den Speicher
}
```

Wiederholung / Vertiefung: `myVector`

Unser `myVector` Typ ist mit einem Datenfeld namens `elem` vom Typ `double[]` implementiert.

- Der einfache Container für `double`, den wir uns definiert hatten:

```
class myVector {
    int sz;         // size
    double* elem;   // Zeiger auf die Elemente
  public:
    myVector( );                          // Standardkonstruktor
    explicit myVector( int s );           // ein Konstruktor
    ~myVector( );                         // Destruktor
    int size( ) const;                    // Anzahl Elemente liefern
    double get( int ) const;              // Element lesen
    void set( int, double );              // Element schreiben
    // da waren doch noch…
    myVector( const myVector& );          // Kopier-Konstruktor
    myVector& operator=( const myVector& ); // Zuweisungsoperator
};
```

- Die Techniken, die wir für unseren `myVector` Container verwenden, liegen im Prinzip allen Programmierarbeiten mit höheren Datenstrukturen zugrunde…

Anzahl der `myVector` Elemente änderbar

Im Arbeitsspeicher gibt es nur feste Größen, wir benötigen aber zur Laufzeit dynamisch änderbare Größen.

- Unser Typ `myVector` soll solche Operationen können:

```
v.push_back( 7.1 );  // fuegt ein Element mit dem Wert 7.1 an
                     // und inkrementiert den Elementzaehler
v.resize( 10 );      // v vom Typ myVector hat jetzt 10 Elemente
```

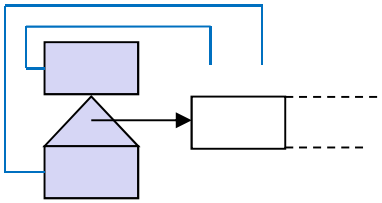
- Lösungsansatz:

```
class myVector {
    int sz;            // Anzahl Elemente
    double* elem;      // Zeiger auf den Anfang des Datenfelds
    int space;         // Platz für weitere Elemente
  public:
    // …
};
```

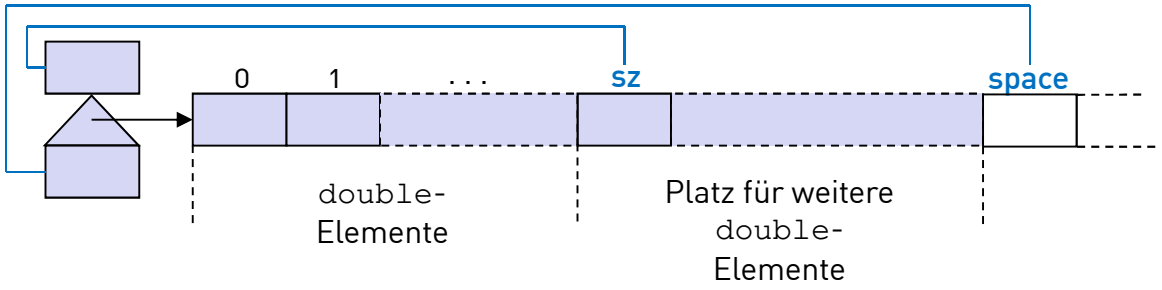
Anzahl der `myVector` Elemente änderbar

Im Arbeitsspeicher gibt es nur feste Größen, wir benötigen aber zur Laufzeit dynamisch änderbare Größen.

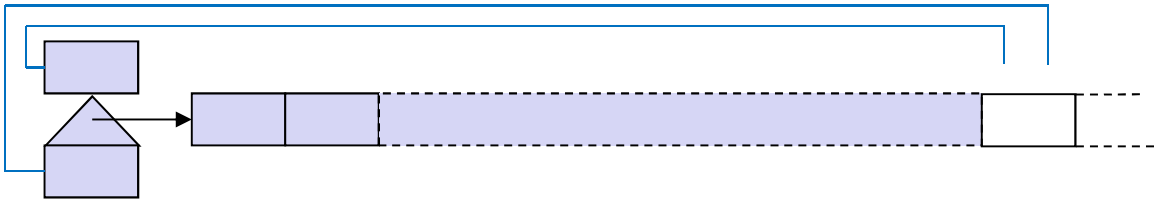
- Ein leeres Objekt vom Typ `myVector`:



- Ein Objekt vom Typ `myVector` füllt sich:



- Ein "volles" Objekt vom Typ `myVector`:



Bildquelle: Stroustrup

void myVector::reserve(int);

Die Methode namens `reserve()` beschafft neuen Speicher im Heap und kümmert sich nicht um die Elemente oder deren Anzahl.

```
void myVector::reserve( int newspace )
{
  if( newspace <= space ) return;       // nie weniger Platz holen

  double* p { new double[newspace] };       // mit new Speicher allokieren
  for( int i{}; i<sz; ++i )                 // Elemente kopieren
    p[i]=elem[i];

  delete[] elem;            // alten Speicher freigeben
  elem = p;                 // Zeiger umhaengen

  space = newspace;         // Platz korrekt setzen
}
```

`void myVector::push_back( double );`

Die Methode namens `push_back()` fügt ein Element hinzu und initialisiert es auch, um neuen Speicher im Heap kümmert sich `reserve()`.

```
void myVector::push_back( double d )
{
  if( space==0 )                    // Erstes Element
     reserve( 8 );                  // Platz aus dem Heap holen
  else
     if( sz==space )                // Kein Platz mehr?
        reserve( space*2 );         // Aus dem Heap holen
  elem[sz] = d;                     // Jetzt d anhaengen…
  ++sz;                             // …und den Elementezaehler erhoehen
}
```

- "Messungen haben bestätigt, dass es keinen merklichen Unterschied macht, ob man Speicher für Elemente vorab oder ad hoc reserviert." (Stroustrup 2010, S. 721)

void myVector::resize(int);

Die Methode namens `resize()` kümmert sich um die Elemente und deren Anzahl, und lässt neuen Speicher im Heap von `reserve()` beschaffen.

```
void myVector::resize( int newsize )
{
  reserve( newsize );                  // Speicher im Heap reservieren
  for( int i{sz}; i<newsize; ++i )       // Initialisierung der
    elem[i] = double{};                // zusaetzlichen Elemente
  sz = newsize;
}
```

Ergebnis: `class myVector`

Unser eigener, einigermaßen realitätsnaher, dynamischer Containertyp `myVector` für `double` Elemente.

```
class myVector {
    int sz;         // size
    int space;      // size + weiterer Platz
    double* elem;   // Zeiger auf die Elemente
  public:
    myVector( );                                  // Standardkonstruktor
    explicit myVector( int );                     // ein Konstruktor
    myVector( const myVector& );                  // Copy-Konstruktor
    myVector& operator=( const myVector& );       // Zuweisungsoperator
    ~myVector( );                                 // Destruktor
    double& operator[]( int );                    // Indexoperator
    int size( ) const;                            // Anzahl Elemente
    double get( int ) const;                      // read
    void set( int, double );                      // write

    void reserve( int );
    void push_back( double );
    void resize( int );
    int capacity( ) const { return space; }
};
```

Anpassung der Zuweisungsoperation

Die Methoden müssen ggf. angepasst werden, z.B. auch der Zuweisungs-operator `myVector& operator=( const myVector& ).`

```
myVector& myVector::operator=( const myVector& a ) {
  if( this == &a ) return *this;
  if( a.sz <= space ) {              // genug Platz, d.h. keine weitere Allokation
    for( int i{}; i<a.sz; ++i )
      elem[i] = a.elem[i];           // Elemente kopieren
                                     // Bem.: die beiden myVector Objekte sind
                                     // bzgl. space nicht unbedingt identisch
    sz = a.sz;
    return *this;
  }
  double* p {new double[a.sz]};      // copy & swap, Platz aus dem Heap holen
  for( int i{}; i<a.sz; ++i )
    p[i] = a.elem[i];                // Elemente kopieren
  delete[] elem;                     // alten Speicherplatz feigeben
  elem = p;                          // Zeiger umhaengen
  space = a.sz; sz = a.sz;           // Groessen richtig setzen
  return *this;                      // das (eigene) Objekt zurueckgeben
}
```

- Per Konvention liefert ein Zuweisungsoperator eine Referenz auf das Objekt zurück, dem zugewiesen wurde.

Typ der `myVector` Elemente als Parameter

Es soll `myVector` Container für *alle* Elementtypen geben.

- Wir wollen unseren `myVector` Container so anpassen, dass Elemente eines beliebigen Typs gespeichert werden können:

```
myVector<double>
myVector<int>
myVector<Month>
myVector<Liwanze*>                  // myVector mit Zeigerelementen
myVector< myVector<Liwanze> >       // myVector mit myVector-Elementen
myVector<bool>
```

- Ansatz: der Typ der Elemente soll ein Parameter des `myVector` sein.
  - `myVector` soll Objekte eingebauter Typen (wie `int`, `double`, ...) und auch benutzerdefinierter Typen (wie eigene Klassen) aufnehmen können.
- Also `myVector` als parametrisierter Typ.
  - Parametrisierte Typen heißen in C++ ***Templates***.
  - Das Fundamentalkonzept der "generischen Programmierung" wird in C++ wesentlich durch Templates umgesetzt.

Templates

Templates kann man sich als Schablonen zur Definition von *Funktionen* oder Klassen vorstellen.

- Deklarations- und Definitions-Syntax bei Funktions-Templates: es folgt auf den Template-Kopf die Funktion, z.B.:

```
template<class X> void myswap( X& a, X& b ) {
    X tmp{a}; a=b; b=tmp;
}
```

- Aufruf-Syntax bei Funktions-Templates: der Compiler leitet den Typ `X` aus dem Typ der Aufrufparameter ab, z.B..

```
int v{123}; int w{-7}; myswap( v, w ); // auch: myswap<int>(v,w);
```

Der Compiler erzeugt:

```
void myswap( int& a, int& b ) { int tmp{a}; a=b; b=tmp; }
```

```
einUDT p{einUDT()}; einUDT q{einUDT()};
myswap( p, q ); // auch: myswap<einUDT>(p,q);
```

Der Compiler erzeugt:

```
void myswap(  einUDT& a, einUDT& b ) { einUDT tmp{a}; a=b; b=tmp; }
```

Templates

Templates kann man sich als Schablonen zur Definition von Funktionen oder *Klassen* vorstellen.

- Bei Klassen-Templates folgt auf den Template-Kopf die Klasse, z.B.:

```
template<class X> class Name_value {
    string nam;
    X val;
  public:
    Name_value();
    Name_value( string name, X value );
    Name_value( const Name_value& );
    Name_value& operator=( const Name_value& );
    // …
};
```

- Der Programmierer gibt den Typ `X` beim Erzeugen von Objekten des Klassen-Templates an, z.B.:

```
Name_value<double> nvPi( string{"Pi"}, 3.1415 );
Name_value<char> nvTab( string{"Tabulator"}, '\t' );
Name_value<bool> nvOn( string{"On"}, true );
Name_value<bool> nvOff( string{"Off"}, false );
```

Templates am Beispiel: myVector<T>

myVector für double.

```
class myVector {
    int sz;        // Anzahl Elemente
    int space;     // Anzahl Elemente plus weiterer Platz
    double* elem; // Zeiger auf die Elemente
  public:
    myVector( )
      : sz{0}, space{0}, elem{nullptr} { } // Standardkonstruktor
    explicit myVector( int s )
      : sz{s}, space{s}, elem{ new double[s] } { } // ein Konstruktor
    myVector( const myVector& ); // Kopier-Konstruktor
    myVector& operator=( const myVector& ); // Zuweisung
    ~myVector(); // Destruktor
    double& operator[]( int n ); // Zugriff
    int size() const; // aktuelle Anzahl von Elementen
    // …
};
```

Templates am Beispiel: myVector<T>

myVector für double.

```
class myVector {
    int sz;         // Anzahl Elemente
    int space;      // Anzahl Elemente plus weiterer Platz
    char* elem;     // Zeiger auf die Elemente
  public:
    myVector( )
      : sz{0}, space{0}, elem{nullptr} { } // Standardkonstruktor
    explicit myVector( int s )
      : sz{s}, space{s}, elem{ new char[s] } { } // ein Konstruktor
    myVector( const myVector& ); // Kopier-Konstruktor
    myVector& operator=( const myVector& ); // Zuweisung
    ~myVector(); // Destruktor
    char& operator[]( int n ); // Zugriff
    int size() const; // aktuelle Anzahl von Elementen
    // …
};
```

Templates am Beispiel: `myVector<T>`

`myVector` für `double`.

```
template<class T> class myVector {
    int sz;         // Anzahl Elemente
    int space;      // Anzahl Elemente plus weiterer Platz
    T* elem;   // Zeiger auf die Elemente
  public:
    myVector( )
      : sz{0}, space{0}, elem{nullptr} { } // Standardkonstruktor
    explicit myVector( int s )
      : sz{s}, space{s}, elem{ new T[s] } { } // ein Konstruktor
    myVector( const myVector& ); // Kopier-Konstruktor
    myVector& operator=( const myVector& ); // Zuweisung
    ~myVector(); // Destruktor
    T& operator[]( int n ); // Zugriff
    int size() const; // aktuelle Anzahl von Elementen
    // …
};
```

Bemerkung: es ist bei Templates i.Allg. *nicht* ohne weiteres möglich, Deklarationen und Definitionen in unterschiedliche Dateien zu schreiben.

Templates am Beispiel: `myVector<T>`

`template<class T>` **bedeutet "für alle Typen** `T`**".**

- `myVector` angewendet:

```
template<class T> class myVector {  // d.h. "fuer alle Typen T"
    // …
};
```

```
myVector<double> vd{};                      // T ist double
myVector<int> vi{};                         // T ist int
myVector< myVector<int> > vvi{};            // T ist myVector<int>
myVector<char> vc{};                        // T ist char
myVector<double*> vpd{};                    // T ist double*
myVector< std::vector<double>* > vvpd{};    // T ist std::vector<double>*
```

Templates am Beispiel: `myVector<T>`

Verallgemeinerung von `myVector` zu `myVector<T>`: wie wird ein Index-geprüfter Zugriff für Elemente vom Typ `T` definiert?

- Der Indexoperator für den geprüften Zugriff auf die Elemente in einem Objekt vom Typ `myVector` kann so implementiert werden:

```
template<class T> class myVector {
   T& operator[]( int );
   // …
};

// ein bereichsgepruefter Indexoperator:
template<class T> T& myVector<T>::operator[]( int n ) {
   if( n<0 || sz<=n )
      error( "myVector::operator[](), bad index" );
   return elem[n];
}
```

Templates am Beispiel: `myVector<T>`

Verallgemeinerung von `myVector` zu `myVector<T>`: was passiert, falls es keinen Standardkonstruktor für Elemente vom Typ `T` gibt?

- Was passiert bei `myVector<T>`, wenn es für den Typ `T` keinen Standardkonstruktor `T()` gibt?
- Üblicher Lösungsansatz: man lässt den Benutzer des `myVector` Typs einen Wert angeben, der als Ersatz für den fehlenden Standardwert dient.
  - Die Umsetzung mit C++ Syntax ist zunächst unkompliziert, weil Default-Parameter in Deklarationen zugelassen sind.
- Beispiel `resize()`

```
template<class T> void myVector<T>::resize( int nsize, T def = T{} )
{ /* …  */ };
```

  - Nun kann ein Wert beim Aufruf von `resize` vorgegeben werden.
  - Nur, wenn kein Wert vorgegeben ist, wird der Standardwert `T()` verwendet, wie er vom Standardkonstruktor des Typs `T` vorgegeben ist.

Templates am Beispiel: `myVector<T>`

Verallgemeinerung von `myVector` zu `myVector<T>`: was passiert, falls es keinen Standardkonstruktor für Elemente vom Typ `T` gibt?

- Ausprobieren:

```
struct NoDefault {
  NoDefault( int ) { }; // der einzige Konstruktor
};

// StdLib

std::vector<NoDefault> v1{};                    // ok, weil leer
std::vector<NoDefault> v2(10);                  // Fehler, Standardkonstruktor fehlt
std::vector<NoDefault> v3( 10, NoDefault(7) ); // ok
v3.resize( 200 );                               // Fehler, kein Standardkonstruktor
v1.resize( 100, NoDefault(7) );                 // ok

// unser myVector<T>

myVector<std::string> mv(10);
mv.resize( 15 );                  // 15 Kopien von std::string{}, also von ""
mv.resize( 20, "aha" );           // 20 Kopien von "aha"

myVector<NoDefault> mv1{};                    // ok, weil leer
myVector<NoDefault> mv2(10);                  // Fehler (noDefault Standardkonstruktor fehlt)
myVector<NoDefault> mv3( 10, NoDefault(7) ); // Fehler (passender myVector Konstruktor fehlt)

mv1.resize( 100, NoDefault(7) );        // Fehler: aus resize wird reserve aufgerufen, und
                                        // reserve braucht fuer das "new" zur Initialisierung
                                        // der neuen Elemente den NoDefault Standardkonstruktor…
```

Templates am Beispiel: `myVector<T>`

Verallgemeinerung von `myVector` zu `myVector<T>`: was passiert, falls es keinen Destruktor für Elemente vom Typ `T` gibt?

- Ähnliche Probleme tauchen auf, falls es für den Typ `T` keinen Destruktor `~T()` gibt
  - Es muss trotzdem sicher gestellt sein, dass der dynamisch angeforderte Speicher zurück gegeben wird.
- Diese Probleme sind nicht einfach zu lösen, denn man muss nun auch nicht initialisierten Speicher manipulieren können.
- Mit `new` / `delete` ist das nicht möglich,
- der Typ `allocator` aus der StdLib (Header `memory`) kann aber mit "rohem", nicht initialisiertem Speicher umgehen.

Templates am Beispiel: `myVector<T,A>`

Ausblick.

- Der Typ `allocator` aus der StdLib stellt nicht-initialisierten Speicher in etwa so bereit:

```
template<class T> class allocator {
 public:
  // …
  T* allocate( int n );
 // reserviert Speicher fuer n Objekte vom Typ T
  void deallocate( T* p, int n );
 // gibt n Objekte vom Typ T frei, beginnend bei p
  void construct( T* p, const T& v );
 // erzeugt in p ein Objekt vom Typ T mit dem Wert v
  void destroy( T* p );
 // loest das Objekt vom Typ T in p auf
};
```

Templates am Beispiel: `myVector<T,A>`

Ausblick.

- Zunächst müssen wir dem `myVector` Template einen `allocator` als weiteren Parameter hinzufügen:

```
template< class T, class A = allocator<T> > class myVector {
    A alloc; // verwende alloc, um den Speicher fuer die Elemente zu handhaben
    // …
};
```

  - Nun kann für `myVector<T>` Objekte statt `new` der Allokator namens `alloc` vom Typ `allocator<T>` verwendet werden.

- Alle `myVector` Methoden, die direkt mit dem Speicher umgehen, müssen dafür noch entsprechend umdefiniert werden, wie z.B. der Konstruktor:

```
template<class T, class A>
myVector<T,A>::myVector( int s, T val = T() )
    : sz{s}, space{s} {
    elem = alloc.allocate( space );
    for( int i{}; i<sz; ++i ) alloc.construct( &elem[i], val );
}
```

Templates am Beispiel: `myVector<T,A>`

Ausblick.

```
template<class T, class A> void myVector<T,A>::reserve( int newspace )
{
  if( newspace <= space ) return;
  T* p { alloc.allocate( newspace ) };
  for( int i{0}; i<sz; ++i ) alloc.construct( &p[i], elem[i] );
  for( int i{0}; i<sz; ++i ) alloc.destroy( &elem[i] );
  alloc.deallocate( elem, space );
  elem = p;
  space = newspace;
}

template<class T, class A> void myVector<T,A>::push_back( const T& val )
{
  if( space==0 ) reserve( 8 );
  else
    if( sz==space ) reserve( space*2 );
  alloc.construct( &elem[sz], val );
  ++sz;
}

template<class T, class A> void myVector<T,A>::resize( int newsize, T val )
{
  reserve( newsize );
  for( int i{sz}; i<newsize; ++i ) alloc.construct( &elem[i], val );
  for( int i{newsize}; i<sz; ++i ) alloc.destroy( &elem[i] );
  sz = newsize;
}
```

Templates am Beispiel: `myArray<T,N>`

Templates können auch mit eingebauten Typen parametrisiert werden.

- Weiteres Beispiel für Templates:

```
// Klassen-Template in Anlehnung an std::array<T,N>
template<class T, int N>
class myArray { /*…*/ };

// Funktions-Template:
template<class T, int N>
void fill( myArray<T,N>& r, const T& val ) { /*…*/ }
```

- C++ Syntax zur Verwendung der Templates im obigen Beispiel:

```
// Klassen-Template: Programmierer gibt Template-Argumente an
myArray<char,1024> buf;    // fuer das Objekt namens buf gilt:
                           // T ist char, N ist 1024

// Funktions-Template: Compiler leitet Template-Argumente ab
fill( buf, '\n' );         // fuer die Funktion namens fill() gilt:
                           // T ist char, N ist 1024, dies wird
                           // automatisch vom Compiler aus dem
                           // uebergebenen buf Objekt bestimmt
```

Templates am Beispiel: `myArray<T,N>`

Template-Parameter für Klassen.

- Die `myArray` Templateklasse:

```
template<class T, int N> class myArray {
  public:
    // Standardkonstruktor, Kopierkonstruktor, Kopierzuweisung:
    // die generierte Funktionalitaet verwenden
    T& operator[]( int n ); // Zugriff
    const T& operator[]( int n ) const; // Zugriff
    T* dataPtr( int n ); // Zeiger auf ein Element
    const T* dataPtr( int n ) const; // Zeiger auf ein Element
    int size() const { return N; }
    void printAll() const;
    // usw.
  private:
    T elem[N];
};

myArray<double,256> dmA{};
myArray<int,100> imA{};
```

Templates am Beispiel: `myArray<T,N>`

Template-Parameter für Methoden- und Funktionsdefinitionen.

- Beispielsweise:

```
template< class T, int N >
T& myArray<T,N>::operator[]( int n ) { return elem[n]; }

template< class T, int N >
T& myArray<T,N>::at( int n ) {
  if( n < 0 || N <= n ) throw std::out_of_range{ "bad index" }
  return elem[n];
}

template< class T, int N >
T* myArray<T,N>::dataPtr( int n ) {
  return &(elem[n]);
}
// etc.
```

- Die `fill()` Templatefunktion:

```
template<class T, int N>
void fill( myArray<T,N>& a, const T& val ) {
   for( size_t i{}; i<N; ++i ) a[i] = val;
}

fill( buf, 'x' ); // Kurzform fuer fill<char,1024>( buf, 'x' )
fill( dmA, 4.2 ); // Kurzform fuer fill<double,256>( dmA, 4.2 )
```

Templates am Beispiel: `myArray<T,N>`

Verbesserte Template-Klasse `myArray<T,N>`.

- Beispielsweise:

```
template< class T, int N > class myArray {
  public:
    myArray();
    explicit myArray( T );
    myArray( const myArray& );
    myArray( std::initializer_list<T> );
    myArray& operator=( const myArray& );
    T& operator[]( int n );
    const T& operator[]( int n ) const;
    int size() const { return N; }
    void fill( const T& x );
    void printAll() const;
    // etc.
  private:
    T elem[N];
};
```

Templates am Beispiel: `myArray<T,N>`

Verbesserte Template-Klasse `myArray<T,N>`.

- Beispielsweise:

```
template< class T, int N >
myArray<T,N>::myArray() : myArray{ T{} } {}

template< class T, int N > myArray<T,N>::myArray( T t ) {
  for( size_t i{}; i<N; ++i ) elem[i] = t;
}

template< class T, int N >
myArray<T,N>& myArray<T,N>::operator=( const myArray<T,N>& r ) {
  for( size_t i{}; i<N; ++i ) elem[i] = r.elem[i];
  return *this;
}

template< class T, int N >
myArray<T,N>::myArray( const myArray& r ) { *this = r; }

template< class T, int N >
myArray<T,N>::myArray( std::initializer_list<T> in ) {
  std::copy( in.begin(), in.end(), elem ); // Ausblick…
}

// etc.
```

Einige Beispielfragen

Templates.

- Was ist ein `void*`?
- Was macht ein `static_cast`? Warum ist das gefährlich?
- Wofür wird der `&` Operator verwendet?
- Welcher entscheidende Sachverhalt unterscheidet Datenfelder bei der Übergabe in Parameterlisten von anderen übergebenen Typen?
- Machen Sie sich die Unterschiede zwischen Zeiger und Referenz nochmals an einem *eigenen* Beispiel klar: implementieren Sie einige Quellcodezeilen und vergleichen Sie, wie sich unterschiedliche Operationen mit beiden Typen verhalten.
- Implementieren Sie zur Verdeutlichung des Problems drei *eigene* Beispiele für Speicherlecks.
- Welche Aufgabe haben `myVector::reserve()` und `myVector::resize()`?
- Erklären Sie mit eigenen Worten (und ohne Quellcode-Zitate), wie `myVector::push_back` arbeitet.

Einige Beispielfragen

Templates.

- Was versteht man unter parametrischer Polymorphie?
- Was hat parametrische Polymorphie mit den C++ Templates zu tun?
- Mit welcher Syntax definieren Sie Templates in C++?
- Erklären Sie allgemein, mit welcher Lösung man dafür sorgen kann, das ein Containertyp auch Elementtypen beinhalten kann, die keinen Standardkonstruktor besitzen, ohne dass diese nicht vernünftig initialisiert sind.
- Was ist ein *allocator* ?

Nächste Einheit:
Abgeleitete Klassen