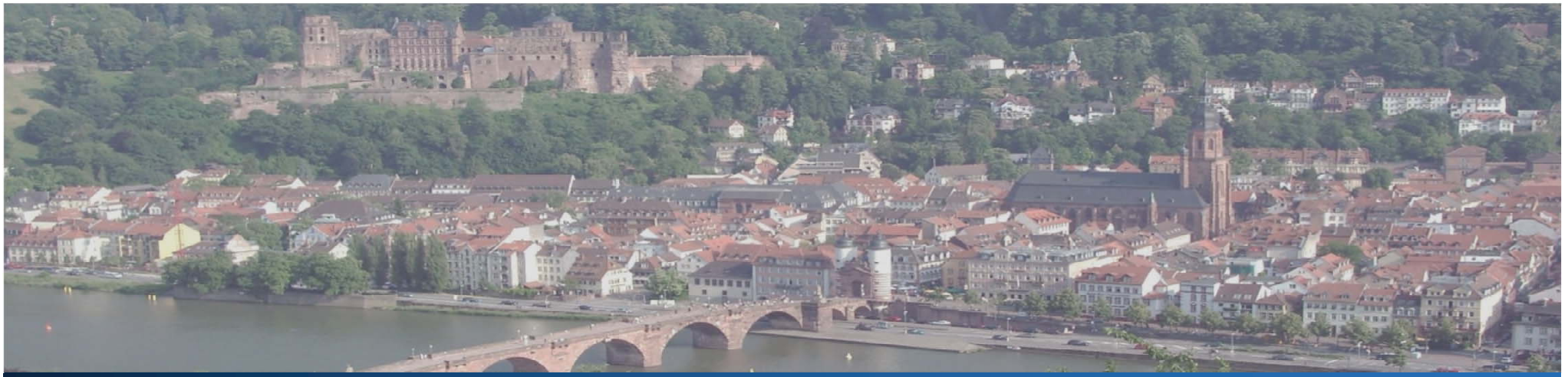
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Generische Programmierung

- Funktionsschablonen
- Klassenschablonen
- Effizienz generischer Programmierung
- Zusammenfassung

Funktionsschablonen

Motivation: Auswechseln von Datentypen in streng typgebundenen Sprachen.

Definition: Eine Funktionsschablone (*Function Template*) entsteht, indem man die Präambel

```
template <class T>
```

bzw.

```
template <typename T>
```

einer Funktionsdefinition voranstellt. In der Schablonendefinition kann man `T` dann wie einen vorhandenen Datentyp verwenden. Dieser Typ muss keine Klasse sein, was die Einführung der äquivalenten Definition mittels **typename** motivierte.

Programm: Vertauschen des Inhalts zweier gleichartiger Referenzen:

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

int main()
{
  int a = 10, b = 20;
  swap( a, b );
}
```

Bemerkung:

- Bei der Übersetzung von `swap(a,b)` generiert der Übersetzer die Version `swap(int& a, int& b)` und übersetzt sie (es sei denn, es gibt schon genau so

eine Funktion).

- Wie beim Überladen von Funktionen wird die Funktion nur anhand der Argumente ausgewählt. Der Rückgabewert spielt keine Rolle.

- Im Unterschied zum Überladen generiert der Übersetzer für jede vorkommende Kombination von Argumenten eine Version der Funktion (keine automatische Typkonversion).

- Dies nennt man automatische Template-Instanzierung.

Programm: Beispiel: Maximum

```
template <class T> T max( T a, T b )
{
  if ( a < b ) return b; else return a;
}
```

Bemerkung: Hier muss für den Typ `T` ein **operator**< definiert sein.

Beispiel: wieder funktionales Programmieren

Problem: Der Aufruf virtueller Funktionen erfordert Entscheidungen zur Laufzeit, was in einigen (wenigen) Fällen zu langsam sein kann.

Abhilfe: Verwendung von Funktionsschablonen.

Programm: Funktionales Programmieren mit Schablonen **(Funktional-statisch.cc)**:

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

class Inkrementierer
{
public:
  Inkrementierer( int n ) { inkrement = n; }
  int operator()( int n ) { return n + inkrement; }
```

```
private:
  int inkrement;
};

class Quadrat
{
public:
  int operator()( int n ) { return n * n; }
};

template <class T> void schleife( T& func )
{
  for ( int i=1; i<10; i++ )
    cout << func(i) << " ";
  cout << endl;
}
```

```
int main()
{
  Inkrementierer ink( 10 );
  Quadrat quadrat;
  schleife( ink );
  schleife( quadrat );
}
```

Bemerkung:

- Es werden die passenden Varianten der Funktion `schleife` erzeugt.
- Unterschied zur Variante mit der gemeinsamen Basisklasse Function:
  - Statt genau einer gibt es nun mehrere Varianten der Funktion `schleife`.
  - Methodenaufrufe am Argument `func` erfolgen **nicht** über virtuelle Funktionen.
- Nachteil: Leider haben wir aber keine Schnittstellendefinition mehr.

Bezeichnung: Man nennt diese Technik auch statischen Polymorphismus, da die Methodenauswahl zur Übersetzungszeit erfolgt. Im Gegensatz dazu bezeichnet man die Verwendung virtueller Funktionen als dynamischen Polymorphismus.

Empfehlung: Wenden Sie diese oder ähnliche Techniken (wie etwa die sogenannten *Expression Templates*) nur an, wenn es unbedingt notwendig ist. Untersuchen Sie auch vorher das Laufzeitverhalten (Profiling), denn laut Donald E. Knuth (ursprünglich wohl von C. A. R. Hoare) gilt:

> Premature optimization is the root of all evil!

Klassenschablonen

Problem: Unsere selbstdefinierten Felder und Listen sind noch zu unflexibel. So hätten wir beispielsweise auch gerne Felder von **int**-Zahlen.

Bemerkung: Dieses Problem rührt von der statischen Typbindung von C/C++ her und tritt bei Sprachen mit dynamischer Typbindung (Scheme, Python, . . .) nicht auf. Allerdings ist es für solche Sprachen viel schwieriger hocheffizienten Code zu generieren.

Abhilfe: Die C++-Lösung für dieses Problem sind parametrisierte Klassen, die auch Klassenschablonen (*Class Templates*) genannt werden.

Definition: Eine Klassenschablone entsteht, indem man der Klassendefinition die Präambel **template** <**class** T> voranstellt. In der Klassendefinition kann dann der Parameter `T` wie ein Datentyp verwendet werden.

Beispiel:

```
// Schablonendefinition
template <class T> class SimpleArray
{
public:
  SimpleArray( int s, T f );
...
};
// Verwendung
SimpleArray<int> a( 10, 0 );
SimpleArray<float> b( 10, 0.0 );
```

Bemerkung:

- `SimpleArray` alleine ist kein Datentyp!
- `SimpleArray<int>` ist ein neuer Datentyp, d. h. Sie können Objekte dieses Typs

erzeugen, oder ihn als Parameter/Rückgabewert einer Funktion verwenden.

- Der Mechanismus arbeitet wieder zur Übersetzungszeit des Programmes. Bei *Übersetzung* der Zeile

```
SimpleArray<int> a( 10, 0 );
```

generiert der Übersetzer den Programmtext für `SimpleArray<int>`, der aus dem Text der Klassenschablone `SimpleArray` entsteht, indem alle Vorkommen von `T` durch **int** ersetzt werden. Anschließend wird diese Klassendefinition übersetzt.

- Da der Übersetzer selbst C++-Programmcode generiert, spricht man auch von generischer Programmierung.

- Den Vorgang der Erzeugung einer konkreten Variante einer Klasse zur Übersetzungszeit nennt man auch Template-Instanzierung. Allerdings gibt es bei Klassen, im Gegensatz zu Funktionen, keine automatische Instanzierung.

- Der Name Schablone (*Template*) kommt daher, dass man sich die parametrisierte Klasse als Schablone vorstellt, die zur Anfertigung konkreter Varianten benutzt wird.

Programm: **(SimpleArray.hh)**

```
template <class T>
class SimpleArray
{
public:
   SimpleArray( int s, T f );
   SimpleArray( const SimpleArray<T>& );
   SimpleArray<T>& operator=( const SimpleArray<T>& );
   ~SimpleArray();

   T& operator[]( int i );
   int numIndices();
```

```
    int minIndex();
    int maxIndex();
    bool isMember( int i );

private:
    int n;    // Anzahl Elemente
    T* p;     // Zeiger auf built-in array
};
```

Bemerkung: Syntaktische Besonderheiten:

- Wird die Klasse selbst als Argument oder Rückgabewert im Rumpf der Definition benötigt, schreibt man `SimpleArray<T>`.

- Im Namen des Konstruktors bzw. Destruktors taucht *kein* `T` auf. Der Klassenparameter parametrisiert den Klassennamen, nicht aber die Methodennamen.

- Die Definition des Destruktors (als Beispiel) lautet dann:

```
template <class T>
SimpleArray<T>::~SimpleArray() { delete[] p; }
```

Programm: Methodenimplementierung **(SimpleArrayImp.cc)**:

```
// Destruktor
template <class T>
SimpleArray<T>::~SimpleArray()
{
  delete[] p;
}

// Konstruktor
template <class T>
SimpleArray<T>::SimpleArray( int s, T v )
{
  n = s;
  p = new T[n];
  for ( int i=0; i<n; i=i+1 ) p[i] = v;
}
```

```
// Copy-Konstruktor
template <class T>
SimpleArray<T>::SimpleArray( const SimpleArray<T>& a )
{
  n = a.n;
  p = new T[n];
  for ( int i=0; i<n; i=i+1 )
    p[i] = a.p[i];
}

// Zuweisungsoperator
template <class T>
SimpleArray<T>& SimpleArray<T>::operator=
( const SimpleArray<T>& a )
{
  if ( &a != this )
```

```
  {
    if ( n != a.n )
    {
      delete[] p;
      n = a.n;
      p = new T[n];
    }
    for ( int i=0; i<n; i=i+1 ) p[i] = a.p[i];
  }
  return *this;
}

template <class T>
inline T& SimpleArray<T>::operator[]( int i )
{
  return p[i];
}
```

```
template <class T>
inline int SimpleArray<T>::numIndices()
{
  return n;
}

template <class T>
inline int SimpleArray<T>::minIndex()
{
  return 0;
}

template <class T>
inline int SimpleArray<T>::maxIndex()
{
  return n - 1;
```

```
}

template <class T>
inline bool SimpleArray<T>::isMember( int i )
{
  return ( i >= 0 && i < n );
}
```

Operator für ostream überladen
und nicht für SimpleArray

```
template <class T>
std::ostream& operator<<( std::ostream& s,
                          SimpleArray<T>& a )
{
  s << "#( ";
  for ( int i=a.minIndex(); i<=a.maxIndex(); i=i+1 )
    s << a[i] << " ";
  s << ")" << std::endl;
  return s;
```

}

Programm: Verwendung **(UseSimpleArray.cc)**:

```
#include <iostream>

#include "SimpleArray.hh"
#include "SimpleArrayImp.cc"

int main()
{
  SimpleArray<float> a( 10, 0.0 ); // erzeuge
  SimpleArray<int> b( 25, 5 );     // Felder

  for ( int i=a.minIndex(); i<=a.maxIndex(); i++ )
    a[i] = i;
  for ( int i=b.minIndex(); i<=b.maxIndex(); i++ )
    b[i] = i;
```

```
    std::cout << a << std::endl << b << std::endl;

    // hier wird der Destruktor gerufen
}
```

Beispiel: Feld fester Größe

Bemerkung:

- Eine Schablone kann auch mehr als einen Parameter haben.
- Als Schablonenparameter sind nicht nur Klassennamen, sondern z. B. auch Konstanten von eingebauten Typen erlaubt.

Anwendung: Ein Feld fester Größe könnte folgendermaßen definiert und verwendet werden:

```
template <class T, int m>
class SimpleArrayCS
{
public:
  SimpleArrayCS( T f );
  ...
private:
  T p[m]; // built-in array fester Groesse
};

...

SimpleArrayCS<int, 5> a( 0 );
SimpleArrayCS<float, 3> a( 0.0 );
...
```

Bemerkung:

- Die Größe ist hier auch zur Übersetzungszeit festgelegt und muss nicht mehr gespeichert werden.
- Da nun keine Zeiger auf dynamisch allokierte Objekte verwendet werden, sind für Copy-Konstruktor, Zuweisung und Destruktor die Defaultmethoden ausreichend.
- Der Compiler kann bei bekannter Feldgröße unter Umständen effizienteren Code generieren, was vor allem für kleine Felder interessant ist (z. B. Vektoren im $\mathbb{R}^2$ oder $\mathbb{R}^3$).
- Es ist ein wichtiges Kennzeichen von C++, dass Objektorientierung bei richtigem Gebrauch auch für sehr kleine Datenstrukturen ohne Effizienzverlust angewendet werden kann.

Beispiel: Smart Pointer

Problem: Dynamisch erzeugte Objekte können ausschließlich über Zeiger verwaltet werden. Wie bereits diskutiert, ist die konsistente Verwaltung des Zeigers (bzw. der Zeiger) und des Objekts nicht einfach.

Abhilfe: Entwurf mit einem neuen Datentyp, der anstatt eines Zeigers verwendet wird. Mittels Definition von **operator**$*$ und **operator**$->$ kann man erreichen, dass sich der neue Datentyp wie ein eingebauter Zeiger benutzen lässt. In Copy-Konstruktor und Zuweisungsoperator wird dann *reference counting* eingebaut.

Bezeichnung: Ein Datentyp mit dieser Eigenschaft wird intelligenter Zeiger (*smart pointer*) genannt.

Programm: **(Zeigerklasse zum *reference counting*, Ptr.hh)**

```
template <class T>
class Ptr
{
  struct RefCntObj
  {
    int count;
    T* obj;
    RefCntObj( T* q ) { count = 1; obj = q; }
  };
  RefCntObj* p;

  void report()
  {
    std::cout << "refcnt = " << p->count << std::endl;
  }
```

```
void increment()
{
  p->count = p->count + 1;
  report();
}

void decrement()
{
  p->count = p->count - 1;
  report();
  if ( p->count == 0 )
  {
    delete p->obj; // Geht nicht fuer Felder!
    delete p;
  }
}
```

```
public:
  Ptr() { p = 0; }

  Ptr( T* q )
  {
    p = new RefCntObj( q );
    report();
  }

  Ptr( const Ptr<T>& y )
  {
    p = y.p;
    if ( p != 0 ) increment();
  }

  ~Ptr()
```

```
{
  if ( p != 0 ) decrement();
}

Ptr<T>& operator=( const Ptr<T>& y )
{
  if ( p != y.p )
  {
    if ( p != 0 ) decrement();
    p = y.p;
    if ( p != 0 ) increment();
  }
  return *this;
}

T& operator*() { return *(p->obj); }
```

```
  T* operator->() { return p->obj; }
};
```

Programm: **(Anwendungsbeispiel, PtrTest.cc)**

```
#include <iostream>
#include "Ptr.hh"

int g( Ptr<int> p )
{
  return *p;
}

int main()
{
  Ptr<int> q = new int( 17 );
  std::cout << *q << std::endl;
  int x = g( q );
  std::cout << x << std::endl;
  Ptr<int> z = new int( 22 );
```

```
    q = z;
    std::cout << *q << std::endl;
    // nun wird alles automatisch geloescht !
}
```

Bemerkung:

- Man beachte die sehr einfache Verwendung durch Ersetzen der eingebauten Zeiger (die natürlich nicht weiterverwendet werden sollten!).
- Nachteil: mehr Speicher wird benötigt (das `RefCntObj`)
- Es gibt verschiedene Möglichkeiten, *reference counting* zu implementieren, die sich bezüglich Speicher- und Rechenaufwand unterscheiden.
- Die hier vorgestellte Zeigerklasse funktioniert (wegen **delete**[]) nicht für Felder!
- *Reference counting* funktioniert nicht für Datenstrukturen mit Zykeln $\rightsquigarrow$ andere Techniken zur automatischen Speicherverwaltung notwendig.

Wiederholung: Generische Programmierung

```
template <class T>
T min( T a, T b ) { if ( a < b ) return a; else return b; }

int a, b, c;
c = min( a, b );
double x, y, z;
z = min( x, y );
```

Ermöglicht es, eine Funktion für verschiedene Datentypen zu schreiben. Übersetzer generiert eigene Version der Funktion für jeden Datentypen.

Automatische Extraktion der Template-Parameter.

Geht auch: c = min<int>(2.0, 3.0);

```
template <class T>
class SimpleArray
{
public:
  ...
  T& operator[]( int i ) { ... }
private:
  int n; T* p;
};

SimpleArray<int> a;
SimpleArray<double> x;
```

Ermöglicht es, eine Klasse mit einem Datentyp zu parametrisieren.

Übersetzer erzeugt für jeden Datentyp `T` eine eigene Version `SimpleArray<T>`. Diese sind vollkommen separate Klassen.

Template-Argument muss explizit angegeben werden.

Gemeinsame Verwendung von Klassen und Funktionsschablone:

```
template <class T>
T min( SimpleArray<T>& x )
{
  T m = x[ x.minIndex() ];
  for ( int i=x.minIndex()+1; i<=x.maxIndex(); i=i+1 )
    if ( x[i] < m ) m = x[i];
  return m;
}

SimpleArray<int> a( 10, 1 );
SimpleArray<double> x( 20, 3.14 );
int b = min( a );
```

Dies nennt man eine Spezialisierung (einer Funktionsschablone), da `min` nicht mit beliebigen Datentypen aufgerufen werden kann.

Effizienz generischer Programmierung

Beispiel: Bubblesort

Aufgabe: Ein Feld von Zahlen $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ ist zu sortieren. Die Sortierfunktion liefert als Ergebnis eine Permutation $a' = (a'_0, a'_1, a'_2, \ldots, a'_{n-1})$ der Feldelemente zurück, so dass

$$a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$$

Idee: Der Algorithmus Bubblesort ist folgendermaßen definiert:

- Gegeben sei ein Feld $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ der Länge n.

- Durchlaufe die Indizes $i = 0, 1, \ldots, n-2$ und vergleiche jeweils a_i und a_{i+1}. Ist $a_i > a_{i+1}$ so vertausche die beiden. Beispiel:

	17	10	8	16
$i = 0$	10	17	8	16
$i = 1$	10	8	17	16
$i = 2$	10	8	16	17

  Am Ende eines solchen Durchlaufes steht die größte der Zahlen sicher ganz rechts und ist damit an der richtigen Position.

- Damit bleibt noch ein Feld der Länge $n - 1$ zu sortieren.

Satz: t_{cs} sei eine obere Schranke der Kosten für einen Vergleich und einen swap (Tausch), und n bezeichne die Länge des Felds. Falls t_{cs} nicht von n abhängt, so hat Bubblesort eine asymptotische Laufzeit von $O(n^2)$.

Beweis:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} t_{cs} = t_{cs} \sum_{i=0}^{n-1} i = t_{cs} \frac{(n-1)n}{2} = O(n^2)$$

Programm: **(bubblesort_.cc)**

```
/* ist in namespace std schon enthalten:
   template <class T> void swap( T& a, T& b ) {
     T t = a;
     a = b;
     b = t;
   }
*/

template <class C> void bubblesort( C& a )
{
  for ( int i=a.maxIndex(); i>=a.minIndex(); i=i-1 )
    for ( int j=a.minIndex(); j<i; j=j+1 )
      if ( a[j+1] < a[j] )
        std::swap( a[j+1], a[j] );
}
```

Bemerkung:

- Die Funktion `bubblesort` benötigt, dass auf $Elementen$ des Feldes der Vergleichsoperator **operator**$<$ definiert ist.

- Die Funktion benutzt die öffentliche Schnittstelle der Feldklassen, die wir programmiert haben, d. h. für `C` können wir jede unserer Feldklassen einsetzen!

Mit folgender Routine kann man Laufzeiten verschiedener Programmteile messen:

Programm: (timestamp.cc)

```
#include <ctime>

// Setzt Marke und gibt Zeitdifferenz zur letzten
// Marke zurueck
clock_t last_time;
double time_stamp()
{
  clock_t current_time = clock();
  double duration =
    ((double) (current_time - last_time)) /
      CLOCKS_PER_SEC;
  last_time = current_time;
  return duration;
}
```

Dies wenden wir auf Bubblesort an:

Programm: Bubblesort für verschiedene Feldtypen (UseBubblesort.cc)

```
#include <iostream>

// SimpleFloatArray mit virtuellem operator[]
#include "SimpleFloatArrayV.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// templatisierte Variante mit variabler Groesse
#include "SimpleArray.hh"
#include "SimpleArrayImp.cc"

// templatisierte Variante mit Compile-Zeit Groesse
#include "SimpleArrayCS.hh"
#include "SimpleArrayCSImp.cc"
```

```
// dynamisches listenbasiertes Feld
#include "FloatArray.hh"
#include "ListFloatArrayDerived.hh"
#include "ListFloatArrayImp.cc"

// Zeitmessung
#include "timestamp.cc"

// generischer bubblesort
#include "bubblesort_.cc"

// Zufallsgenerator
#include "Zufall.cc"

const int n = 32000;
const int samples = 50; // Mittle "samples" Messungen
const int lowSamples = 10; // Anzahl fuer langsame Impl.

static Zufall z( 93576 );
```

```
template <class T>
void initialisiere( T& a )
{
  for ( int i=0; i<n; i=i+1 )
    a[i] = z.ziehe_zahl();
}

int main()
{
  SimpleArrayCS<float, n> a( 0.0 );
  SimpleArray<float> b( n, 0.0 );
  SimpleFloatArray c( n, 0.0 );
  ListFloatArray d;

  initialisiere( a ); initialisiere( b );
  initialisiere( c ); initialisiere( d );

  double duration;
```

```
duration = 0.0;
std::cout << "SimpleArrayCS ...";
time_stamp();
for ( int s=0; s<samples; s=s+1 )
{
  bubblesort( a );
  duration = duration + time_stamp();
}
std::cout << duration / samples << " sec" << std::endl;

duration = 0.0;
std::cout << "SimpleArray ...";
time_stamp();
for ( int s=0; s<samples; s=s+1 )
{
  bubblesort( b );
  duration = duration + time_stamp();
}
```

```
std::cout << duration / samples << " sec" << std::endl;

duration = 0.0;
std::cout << "SimpleFloatArray ...";
time_stamp();
for ( int s=0; s<samples; s=s+1 )
{
  bubblesort( c );
  duration = duration + time_stamp();
}
std::cout << duration / samples << " sec" << std::endl;

// duration = 0.0;
// std::cout << "ListFloatArray ...";
// time_stamp();
// for ( int s=0; s<lowSamples; s=s+1 )
// {
//   bubblesort( d );
//   duration = duration + time_stamp();
```

```
    // }
    // std::cout << duration / lowSamples << " sec" << std::endl;
}
```

Ergebnis:

Ergebnisse vom WS 2002/2003:

n	1000	2000	4000	8000	16000	32000
built-in array	0.01	0.04	0.14	0.52	2.08	8.39
`SimpleArrayCS`	0.01	0.03	0.15	0.58	2.30	9.12
`SimpleArray`	0.01	0.05	0.15	0.60	2.43	9.68
`SimpleArray` ohne inline	0.04	0.15	0.55	2.20	8.80	35.31
`SimpleFloatArrayV`	0.04	0.15	0.58	2.28	9.13	36.60
`ListFloatArray`	4.62	52.38	—	—	—	—

WS 2011/2012, gcc 4.5.0, 2.26 GHz Intel Core 2 Duo:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.0029	0.0089	0.034	0.138	0.557	2.205
`SimpleArray`	0.0031	0.0098	0.039	0.156	0.622	2.499
`SimpleFloatArrayV`	0.0110	0.0204	0.083	0.330	1.322	5.288

WS 2014/2015, gcc 4.9.2 -O3, 2.6 GHz Intel Core i7:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.00096	0.0032	0.013	0.070	0.328	1.4048
`SimpleArray`	0.00096	0.0034	0.013	0.0727	0.329	1.4095
`SimpleFloatArrayV`	0.0008	0.0027	0.011	0.0579	0.297	1.4353
`ListFloatArray`	1.056	8.999	—	—	—	—

WS 2015/2016, gcc 5.2.1 -O3, 4.0 GHz Intel Core i7-4790K:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.000238	0.000936	0.00372	0.0148	0.0594	0.240
`SimpleArray`	0.000237	0.000938	0.00371	0.0148	0.0593	0.240
`SimpleFloatArrayV`	0.000185	0.000716	0.00282	0.0112	0.0447	0.182
`ListFloatArray`	0.646912	6.68187	—	—	—	—

WS 2015/2016, gcc 5.2.1, 4.0 GHz Intel Core i7-4790K:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.00208	0.00837	0.0330	0.132	0.545	2.192
`SimpleArray`	0.00208	0.00833	0.0329	0.132	0.538	2.200
`SimpleFloatArrayV`	0.00241	0.00948	0.0379	0.152	0.613	2.511
`ListFloatArray`	1.846	14.2928	—	—	—	—

Bemerkung:

n	1000	2000	4000	8000	16000	32000
built-in array	0.01	0.04	0.14	0.52	2.08	8.39
`SimpleArrayCS`	0.01	0.03	0.15	0.58	2.30	9.12
`SimpleArray`	0.01	0.05	0.15	0.60	2.43	9.68
`SimpleArray` ohne inline	0.04	0.15	0.55	2.20	8.80	35.31
`SimpleFloatArrayV`	0.04	0.15	0.58	2.28	9.13	36.60
`ListFloatArray`	4.62	52.38	—	—	—	—

- Die ersten fünf Zeilen zeigen deutlich den $O(n^2)$-Aufwand: Verdopplung von n bedeutet vierfache Laufzeit.

- Die Zeilen fünf und vier zeigen die Laufzeit für die Variante mit einem virtuellem `operator[]` bzw. eine Version der Klassenschablone, bei der das Schlüsselwort `inline` vor der Methodendefinition des `operator[]` weggelassen wurde. Diese beiden Varianten sind etwa viermal langsamer als die vorherigen.

- Eine Variante mit eingebautem Feld (nicht vorgestellt, ohne Klassen) ist am schnellsten, gefolgt von den zwei Varianten mit Klassenschablonen, die unwesentlich langsamer sind.

- Beachte auch den Einfluss des Optimizers (gcc-Option -O3).

- `ListFloatArray` ist die listenbasierte Darstellung des Feldes mit Index-Wert-Paaren. Diese hat Komplexität $O(n^3)$, da nun die Zugriffe auf die Feldelemente Komplexität $O(n)$ haben.

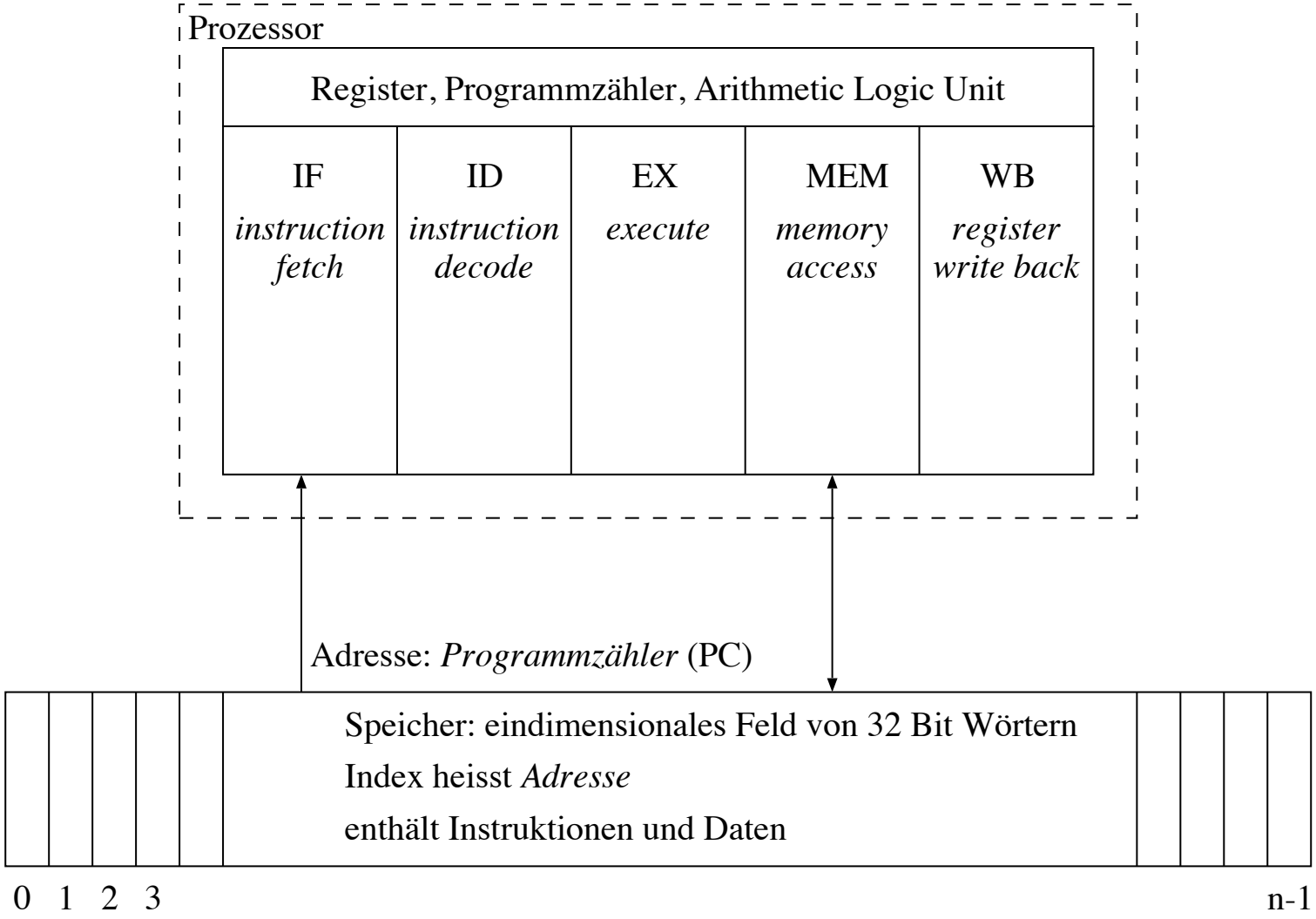
Frage: Warum sind die Varianten auf Schablonenbasis (mit inlining) schneller als die Variante mit virtueller Methode?

RISC

Bezeichnung: RISC steht für *Reduced Instruction Set Computer* und steht für eine Kategorie von Prozessorarchitekturen mit verhältnismäßig einfachem Befehlssatz. Gegenpol: CISC=*Complex Instruction Set Computer*.

Geschichte: RISC stellt heutzutage den Großteil aller Prozessoren dar (vor allem bei eingebetteten Systemen (Handy, PDA, Spielekonsole, etc.), wo das Verhältnis Leistung/Verbrauch wichtig ist). Für PCs ist allerdings noch mit den Intel-Chips die CISC-Technologie dominant (mittlerweile wurden aber auch dort viele RISC-Techniken übernommen).

Aufbau eines RISC-Chips


Prozessor
Register, Programmzähler, Arithmetic Logic Unit
IF
instruction fetch
ID
instruction decode
EX
execute
MEM
memory access
WB
register write back
Adresse: Programmzähler (PC)
Speicher: eindimensionales Feld von 32 Bit Wörtern
Index heisst Adresse
enthält Instruktionen und Daten
0 1 2 3
n-1

Befehlszyklus

Bezeichnung: Ein typischer RISC-Befehl lässt sich in Teilschritte unterteilen, die von verschiedener Hardware (in der CPU) ausgeführt werden:

1. IF: Holen des nächsten Befehls aus dem Speicher. Ort: Programmzähler.
2. ID: Dekodieren des Befehls, Auslesen der beteiligten Register.
3. EX: Eigentliche Berechnung (z. B. Addieren zweier Zahlen).
4. MEM: Speicherzugriff (entweder Lesen oder Schreiben).
5. WB: Rückschreiben der Ergebnisse in Register.

Dies nennt man Befehlszyklus (*instruction cycle*).

Pipelining

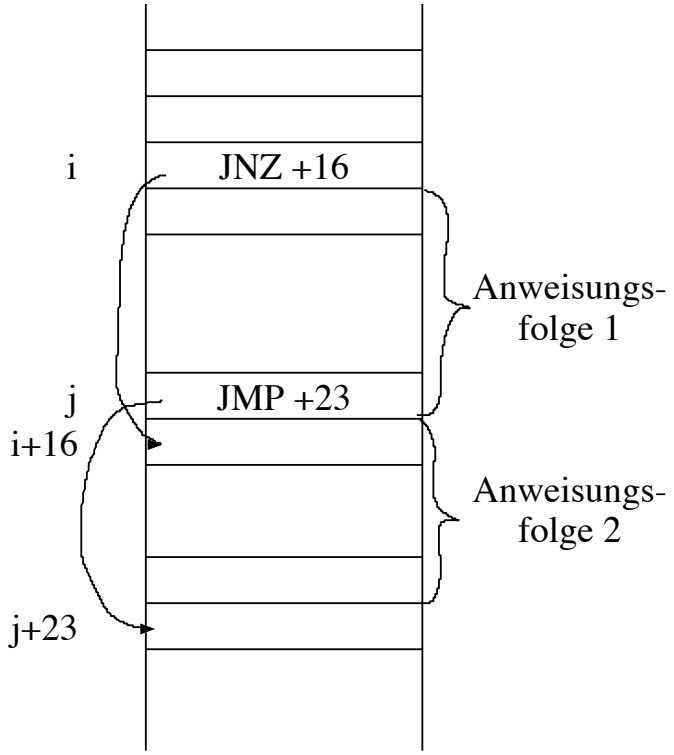
Diese Stadien werden nun für aufeinanderfolgende Befehle überlappend ausgeführt (Pipelining).

IF	Instr1	Instr2	Instr3	Instr4	Instr5	Instr6	Instr7
ID	—	Instr1	Instr2	Instr3	Instr4	Instr5	Instr6
EX	—	—	Instr1	Instr2	Instr3	Instr4	Instr5
MEM	—	—	—	Instr1	Instr2	Instr3	Instr4
WB	—	—	—	—	Instr1	Instr2	Instr3

Probleme mit Pipelining

Sehen wir uns an, wie eine `if`-Anweisung realisiert wird:

```
if ( a == 0 )
{
  <Anweisungsfolge 1>
}
else
{
  <Anweisungsfolge 2>
}
```



Problem: Das Sprungziel des Befehls `JNZ +16` steht erst am Ende der dritten Stufe der Pipeline (EX) zur Verfügung, da ein Register auf 0 getestet und 16 auf

den PC addiert werden muss.

Frage: Welche Befehle sollen bis zu diesem Punkt weiter angefangen werden?

Antwort:

- Gar keine, dann bleiben einfach drei Stufen der Pipeline leer (pipeline stall).

- Man $rät$ das Sprungziel (branch prediction unit) und führt die nachfolgenden Befehle spekulativ aus (ohne Auswirkung nach aussen). Notfalls muss man die Ergebnisse dieser Befehle wieder verwerfen.

Bemerkung: Selbst das Ziel eines unbedingten Sprungbefehls stünde wegen der Addition des Offset auf den PC erst nach der Stufe EX zur Verfügung (es sei denn, man hat extra Hardware dafür).

Funktionsaufrufe

Ein Funktionsaufruf (Methodenaufruf) besteht aus folgenden Operationen:

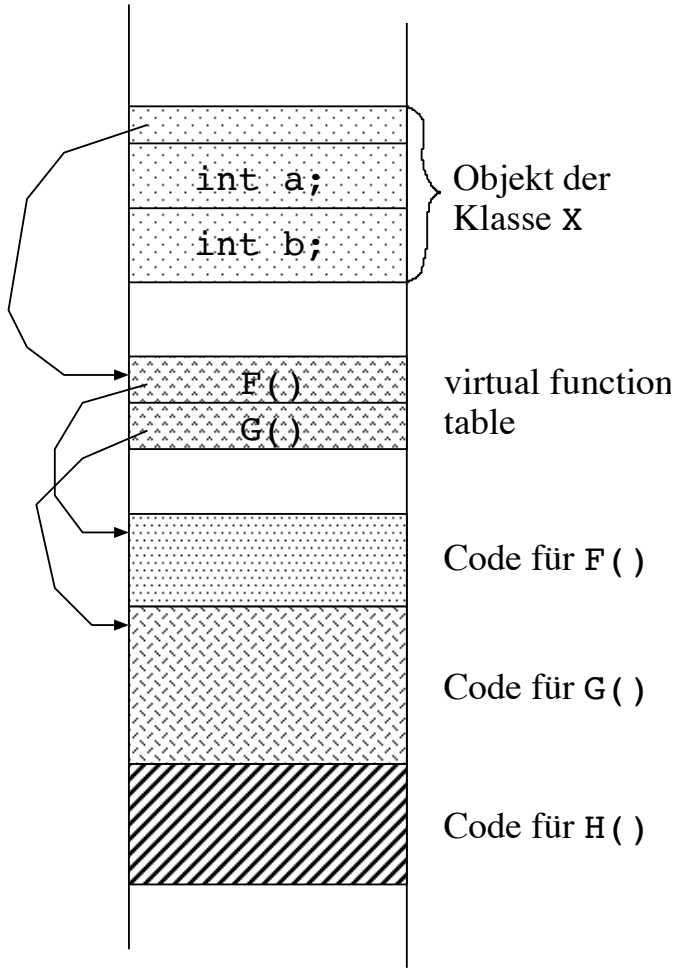
- Sicherung der Rücksprungadresse auf dem Stack
- ein unbedingter Sprungbefehl
- der Rücksprung an die gespeicherte Adresse
- + eventuelle Sicherung von Registern auf dem Stack

Diese Liste gilt genauso für CISC-Architekturen. Ein Funktionsaufruf ist also normalerweise mit erheblichem Aufwand verbunden. Darüberhinaus optimiert der Compiler nicht über Funktionsaufrufe hinweg, was zu weiteren Geschwindigkeitseinbussen führt.

Realisierung virtueller Funktionen

Betrachte folgende Klassendefinition und ein konkretes Objekt im Speicher:

```
class X
{
public:
   int a;
   int b;
   virtual void F();
   virtual void G();
   void H();
};

X x;
x.F();
x.H();
```



Bemerkung:

- Für jede Klasse gibt es eine Tabelle mit Zeigern auf den Programmcode für die virtuellen Funktionen dieser Klasse. Diese Tabelle heißt *virtual function table* (VFT).

- Jedes Objekt einer Klasse, die virtuelle Funktionen enthält, besitzt einen Zeiger auf die VFT der zugehörigen Klasse. Dies entspricht im wesentlichen der Typinformation, die bei Sprachen mit dynamischer Typbindung den Daten hinzugefügt ist.

- Beim Aufruf einer virtuellen Methode generiert der Übersetzer Code, welcher der VFT des Objekts die Adresse der aufzurufenden Methode entnimmt und dann den Funktionsaufruf durchführt. Welcher Eintrag der VFT zu entnehmen ist, ist zur Übersetzungszeit bekannt.

- Der Aufruf nichtvirtueller Funktionen geschieht ohne VFT. Klassen (und ihre

zugehörigen Objekte) ohne virtuelle Funktionen brauchen keinen Zeiger auf eine VFT.

- Für den Aufruf virtueller Funktionen ist immer ein Funktionsaufruf notwendig, da erst zur Laufzeit bekannt ist, welche Methode auszuführen ist.

Inlining

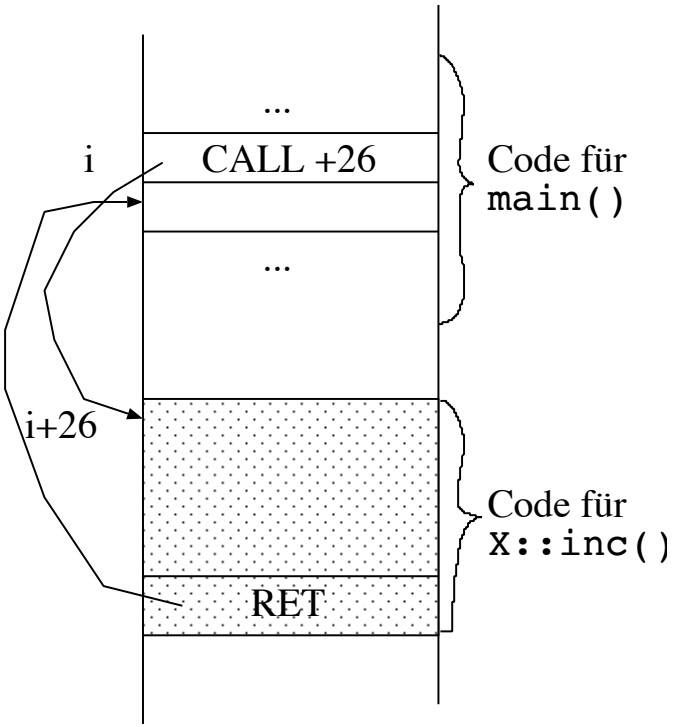
Problem: Der Funktionsaufruf sehr kurzer Funktionen ist relativ langsam.

Beispiel:

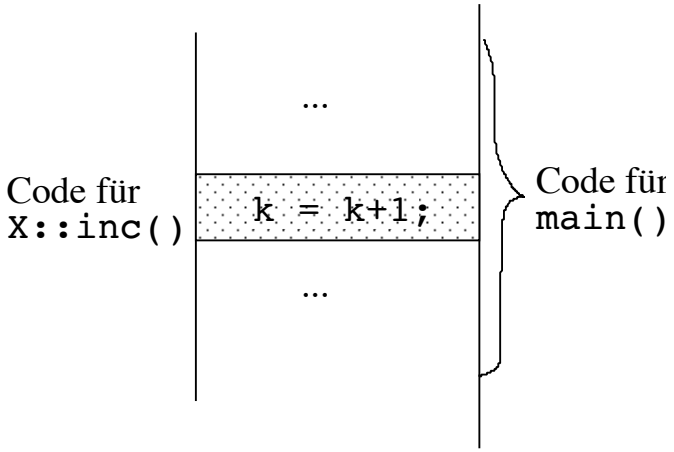
```
class X {
public:
  void inc();
private:
  int k;
};

inline void X::inc()
{
  k = k + 1;
}
```

```
void main()
{
  X x;
  x.inc();
}
```

Ohne das Schlüsselwort `inline` in der Methodendefinition generiert der C++-Übersetzer einen Funktionsaufruf für `inc()`:



Mit dem Schlüsselwort `inline` in der Methodendefinition setzt der Übersetzer den Code der Methode am Ort des Aufrufes direkt ein falls dies möglich ist:



Bemerkung:

- Inlining ändert nichts an der Semantik des Programmes.
- Das Schlüsselwort `inline` ist nur ein *Vorschlag* an den Compiler. Z. B. wird es für rekursive Funktionen ignoriert.
- Virtuelle Funktionen können nicht inline ausgeführt werden, da die auszuführende Methode zur Übersetzungszeit nicht bekannt ist.
- Aber: Änderungen der Implementation einer Inline-Funktion in einer Bibliothek machen normalerweise die erneute Übersetzung von anderen Programmteilen notwendig!

Bemerkung: Es sei auch nochmal eindringlich an Knuth's Wort *"Premature optimization is the root of all evil"* erinnert. Bevor Sie daran gehen, Ihr Programm durch Elimination virtueller Funktionen und Inlining unflexibler zu machen, sollten Sie folgendes tun:

1. Überdenken Sie den Algorithmus!

2. Messen Sie, wo der „Flaschenhals" wirklich liegt (Profiling notwendig).

3. Überlegen Sie, ob die erreichbare Effizienzsteigerung den Aufwand wert ist.

Beispielsweise ist die einzig sinnvolle Verbesserung für das Sortierbeispiel am Anfang dieses Abschnitts das Verwenden eines besseren Algorithmus!

Zusammenfassung

- Klassenschablonen definieren parametrisierte Datentypen und sind daher besonders geeignet, um allgemein verwendbare Konzepte (ADT) zu implementieren.
- Funktionsschablonen definieren parametrisierte Funktionen, die auf verschiedenen Datentypen (mit gleicher Schnittstelle) operieren.
- In beiden Fällen werden konkrete Varianten der Klassen/Funktionen zur Übersetzungszeit erzeugt und übersetzt (generische Programmierung).
- Diese Techniken sind für Sprachen mit dynamischer Typbindung meist unnötig. Solche Sprachen brauchen aber in vielen Fällen Typabfragen zur Laufzeit, was dazu führt, dass der erzeugte Code nicht mehr hocheffizient ist.

Nachteile der generischen Programmierung

- Es wird viel Code erzeugt. Die Übersetzungszeiten template-intensiver Programme können unerträglich lang sein.

- Es ist keine getrennte Übersetzung möglich. Der Übersetzer muss die Definition aller vorkommenden Schablonen kennen. Dasselbe gilt für Inline-Funktionen. Dies erfordert dann z. B. auch spezielle Softwarelizenzen.

- Das Finden von Fehlern in Klassen/Funktionenschablonen ist erschwert, da der Code für eine konkrete Variante nirgends existiert. Empfehlung: testen Sie zuerst mit einem konkreten Datentyp und machen Sie dann eine Schablone daraus.