- Polymorphismus bedeutet, dass die jeweilige Methode bzw. Funktion in Abhängigkeit der Parametertypen (u.a. auch nur von einem einzigen Parametertyp) ausgewählt wird.
- Dies kann statisch (also zur Übersetzzeit) oder dynamisch (zur Laufzeit) erfolgen.
- Dynamische Polymorphismus wird grundsätzlich von objekt-orientierten Programmiersprachen unterstützt.
- In einigen Fällen lässt sich die Auswahl der Methode oder Funktion auch im Rahmen der Optimierung zur Übersetzzeit treffen.
- Bei statischem Polymorphismus wird der Übersetzer gezwungen, die Auswahl zur Übersetzzeit durchzuführen.

- Für C++ werden seit einiger Zeit auch Optimierer angeboten, die beim Zusammenbau des Programms aktiv werden und Optimierungen über die einzelnen Übersetzungseinheiten hinweg vornehmen können.
- Bei neueren GCC-Versionen wird dies durch die Option *-flto* möglich (LTO = *link time optimization*). Die Option muss zur Übersetz- und Zusammenbauzeit angegeben werden.
- Zur Übersetzzeit wird dann die internen Datenstrukturen mit in der Ausgabe abgelegt, die dann zur Zusammenbauzeit ausgewertet werden kann.
- Dies eröffnet die Möglichkeit, dynamischen Polymorphismus durch statischen Polymorphismus zu ersetzen, wenn feststeht, dass bei dem Aufruf einer virtuellen Methode an einer Stelle immer die gleiche Implementierung aufgerufen wird.
- Beim kommenden GCC 5 können mit dieser Optimierung 50% der virtuellen Methodenaufrufe bei Firefox durch statische Aufrufe ersetzt werden.

- Neben der Frage, ob die Entscheidung zur Übersetzzeit oder Laufzeit fällt, lässt sich unterscheiden, ob die Typen irgendwelchen Beschränkungen unterliegen oder nicht.
- Dies lässt sich prinzipiell frei kombinieren. C++ unterstützt davon jedoch nur zwei Varianten:

	statisch	dynamisch
beschränkt	(z.B. in Ada)	virtuelle Methoden in C++
unbeschränkt	Templates in C++	(z.B. in Smalltalk oder Perl)

- Mit den *concepts* gibt es Bestrebungen, auch in C++ die Möglichkeit von Beschränkungen einzuführen. Bislang wurden die bisherigen Vorschläge noch nicht in einen Standard übernommen und auch für C++17 ist dies nicht geplant.

Statischer vs. dynamischer Polymorphismus in C++ 219

Vorteile dynamischen Polymorphismus in C++:

- Unterstützung heterogener Datenstrukturen, etwa einer Liste von Widgets oder graphischer Objekte.
- Dynamisches Nachladen unbekannter Implementierungen ist möglich.
- Die Schnittstelle ist durch die Basisklasse klarer definiert, da sie dadurch beschränkt ist.
- Der generierte Code ist kompakter.

Vorteile statischen Polymorphismus in C++:

- Erhöhte Typsicherheit.
- Die fehlende Beschränkung auf eine Basisklasse erweitert den potentiellen Anwendungsbereich. Insbesondere können auch elementare Datentypen mit unterstützt werden.
- Der generierte Code ist effizienter.

```
class StdRand {
   public:
      void seed(long seedval) { std::srand(seedval); }
      long next() { return std::rand(); }
};

class Rand48 {
   public:
      /* note srand48 & lrand48 are part of the
         POSIX standard but neither of the C or
         C++ standard and thereby not in std:: */
      void seed(long seedval) { srand48(seedval); }
      long next() { return lrand48(); }
};
```

- Gegeben seien zwei Klassen, die nicht miteinander verwandt sind, aber bei einigen relevanten Methoden die gleichen Signaturen offerieren wie hier etwa bei *seed* und *next*.

```
template<typename Rand>
unsigned int test_sequence(Rand& rg) {
   constexpr unsigned int N = 64;
   unsigned int hits[N][N][N] = {{{0}}};
   rg.seed(std::random_device()());
   unsigned int r1 = rg.next() / N % N;
   unsigned int r2 = rg.next() / N % N;
   unsigned int max = 0;
   for (unsigned int i = 0; i < N*N*N*N; ++i) {
      unsigned int r3 = rg.next() / N % N;
      unsigned int count = ++hits[r1][r2][r3];
      if (count > max) {
         max = count;
      }
      r1 = r2; r2 = r3;
   }
   return max;
}
```

- Dann können beide von der gleichen Template-Funktion behandelt werden.

```
int main() {
   StdRand stdrand;
   Rand48 rand48;
   std::cout << "result of StdRand: "
      << test_sequence(stdrand) << std::endl;
   std::cout << "result of Rand48: "
      << test_sequence(rand48) << std::endl;
}
```

- Hier verwendet *test_sequence* jeweils die passenden Methoden *seed* und *next* in Abhängigkeit des statischen Argumenttyps.
- Die Kosten für den Aufruf virtueller Methoden entfallen hier. Dafür wird hier der Programmtext für *test_sequence* für jede Typen-Variante zusätzlich generiert.

```
template<typename T>
T mod(T a, T b) {
   return a % b;
}

double mod(double a, double b) {
   return fmod(a, b);
}
```

- Explizite Spezialfälle können in Konkurrenz zu implizit instantiierbaren Templates stehen. Sie werden dann, falls sie irgendwo passen, bevorzugt verwendet.
- Auf diese Weise ist es auch möglich, effizientere Algorithmen für Spezialfälle neben dem allgemeinen Template-Algorithmus zusätzlich anzubieten.

```
template<typename T>
const char* tell_type(T* p) { return "is a pointer"; }

template<typename T>
const char* tell_type(T (*f)()) { return "is a function"; }

template<typename T>
const char* tell_type(T v) { return "is something else"; }

int main() {
   int* p; int a[10]; int i;
   cout << "p " << tell_type(p) << endl;
   cout << "a " << tell_type(a) << endl;
   cout << "i " << tell_type(i) << endl;
   cout << "main " << tell_type(main) << endl;
}
```

- Speziell konstruierte Typen können separat behandelt werden, so dass sich etwa Zeiger von anderen Typen unterscheiden lassen.

```
template<typename T>
constexpr std::size_t dim(T& vec) {
   return sizeof(vec)/sizeof(*vec);
}
```

- Funktionen, die mit *constexpr* deklariert sind, werden zur Überzeitzeit ausgeführt, wenn sie mit zur Übersetzzeit bekannten Werten arbeiten.
- Hier wird die Dimensionierung eines Arrays abgefragt.
- Obige Variante lässt sich auf beliebige Typen anwenden, scheitert aber, wenn *vec* sich nicht dereferenzieren lässt. Es werden auch Zeigertypen zugelassen, bei denen dann eine 1 zurückgeliefert wird.
- Diese Fassung funktioniert nur für die Arrays, für die es gedacht ist:

```
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) {
   return N;
}
```

- Traits sind Charakteristiken, die mit Typen assoziiert werden.
- Die Charakteristiken selbst können durch Klassen repräsentiert werden und die Assoziationen können implizit mit Hilfe von Templates oder explizit mit Template-Parametern erfolgen.
- Über <*type_traits*> können diverse Eigenschaften von Typen abgefragt oder getestet werden.

Sum.hpp

```
#ifndef SUM_HPP
#define SUM_HPP

template <typename T>
inline T sum(const T* begin, const T* end) {
   T result = T();
   for (const T* it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- Die Template-Funktion *sum* erhält einen Zeiger auf den Anfang und das Ende eines Arrays und liefert die Summe aller enthaltenen Elemente.
- (Dies ließe sich auch mit Iteratoren lösen, darauf wird hier jedoch der Einfachheit halber verzichtet.)

Summe eines Arrays

TestSum.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum.hpp"
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   float floats[] = {1.2, 3.7, 4.8};
   std::cout << "sum of floats[] = " <<
      sum(floats, floats + dim(floats)) << std::endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   std::cout << "sum of text[] = " << sum(text, text + dim(text)) <<
      std::endl;
}
```

- In den beiden Tests mit **int** und **float** klappt des problemlos, jedoch nicht mit **char**...
- Bei den ersten beiden Arrays funktioniert das Template recht gut. Weswegen scheitert es im dritten Fall?

```
thales$ testsum
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = ,
thales$
```

- Wieso wird „,“ ausgegeben, wenn wir eine numerische Summe erwartet hätten?

SumTraits.hpp

```
#ifndef SUM_TRAITS_HPP
#define SUM_TRAITS_HPP

// by default, we use the very same type
template <typename T>
class SumTraits {
   public:
      using SumValue = T;
};

// special case for char
template <>
class SumTraits<char> {
   public:
      using SumValue = int;
};
#endif
```

- Die Template-Klasse *SumTraits* liefert als Charakteristik den jeweils geeigneten Datentyp für eine Summe von Werten des Typs T.
- Per Voreinstellung ist das T selbst, aber es können Ausnahmen definiert werden wie hier zum Beispiel für *char*.

Sum2.hpp

```
#ifndef SUM2_HPP
#define SUM2_HPP

#include "SumTraits.hpp"

template <typename T>
inline typename SumTraits<T>::SumValue sum(const T* begin,
      const T* end) {
   using SumValue = typename SumTraits<T>::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- Statt *T* wird hier jetzt *SumTraits*<*T*>::*SumValue* als Typ für die Summe verwendet.

TestSum2.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum2.hpp"
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   float floats[] = {1.2, 3.7, 4.8};
   std::cout << "sum of floats[] = " <<
      sum(floats, floats + dim(floats)) << std::endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   std::cout << "sum of text[] = "
      << sum(text, text + dim(text)) << std::endl;
}
```

```
thales$ testsum2
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = 3372
thales$
```

Sum3.hpp

```
#ifndef SUM3_HPP
#define SUM3_HPP

#include "SumTraits.hpp"

template <typename T, typename ST = SumTraits<T>>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   using SumValue = typename ST::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- C++ unterstützt voreingestellte Template-Parameter, seit C++11 auch bei Template-Funktionen.
- Diese Konstruktion ermöglicht dann einem Nutzer dieser Konstruktion die Voreinstellung zu übernehmen oder bei Bedarf eine eigene Traits-Klasse zu spezifizieren.

TestSum3.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum3.hpp"

template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }

struct MyTraits {
   using SumValue = long long int;
};

int main() {
   int numbers[] = {2147483647, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   std::cout << "sum of numbers[] = " <<
      sum<int, MyTraits>(numbers, numbers + dim(numbers)) << std::endl;
}
```

```
thales$ testsum3
sum of numbers[] = -2147483639
sum of numbers[] = 2.14748e+09
thales$
```

Sum4.hpp

```
template <typename ST, typename T>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   using SumValue = typename ST::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}
template <typename T>
inline typename SumTraits<T>::SumValue sum(const T* begin,
      const T* end) {
   return sum<SumTraits<T>, T>(begin, end);
}
```

- Den automatisierbar bestimmbaren Template-Parameter sollten diejenigen vorangehen, die u.U. abweichend bestimmt werden.
- Umgekehrt gilt, dass den Template-Parameter mit Voreinstellungen nicht solche ohne Voreinstellungen folgen dürfen.
- Der Konflikt lässt sich durch zwei Varianten lösen: einer generellen mit zwei Template-Parametern und dem Spezialfall mit nur einem Template-Parameter.

TestSum4.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum4.hpp"

template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }

struct MyTraits {
   using SumValue = long long int;
};

int main() {
   int numbers[] = {2147483647, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   std::cout << "sum of numbers[] = " <<
      sum<MyTraits>(numbers, numbers + dim(numbers)) << std::endl;
}
```

- Nun muss nur noch die Traits-Klasse angegeben werden, jedoch nicht mehr der Elementtyp des Arrays, der sich aus dem Parameter ableiten lässt.