Generisches Programmieren

Template-Funktionen, Template-Klassen, generisches Sortieren, Fibonacci- und Ackermann-Zahlen zur Kompilierungszeit

Generische Funktionalität

- Viele Funktionen braucht man für ganz verschiedene Typen

```
// POST: values of a and b are interchanged
void swap (int& a, int& b) {
  int c = a;
  a = b;
  b = c;
}
```

z.B. beim Sortieren ganzer Zahlen

Generische Funktionalität

- Viele Funktionen braucht man für ganz verschiedene Typen

```
// POST: values of a and b are interchanged
void swap (double& a, double& b) {
  double c = a;
  a = b;
  b = c;
}
```

z.B. beim Sortieren reeller Zahlen

Generische Funktionalität

- Viele Funktionen braucht man für ganz verschiedene Typen
- Ziel: Funktion nur einmal schreiben, und dann für alle passenden Typen aufrufen können.
- So eine Funktion heisst **generisch**.
- In C++: **Template-Funktion**

Generisches Swap

```
// POST: values of a and b are interchanged
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
```

- Template = „Vorlage, Formular"
- T ist Platzhalter für einen Typ und heisst **Template-Parameter**

Generisches Swap - Aufruf

```
int main() {
  int n1 = 1;
  int n2 = 2;
  swap (n1, n2);
  std::cout << n1 << ", " << n2 << std::endl;  // 2, 1

  double r1 = 1;
  double r2 = 2;
  swap (r1, r2);
  std::cout << r1 << ", " << r2 << std::endl;  // 2, 1

  return 0;
}
```

Instantiierung für `int`

Instantiierung für `double`

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
```

1. Compiler erkennt den Typ von n1, n2 (`int`)

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
```

→

```
void swap (int& a, int& b) {
  int c = a;
  a = b;
  b = c;
}
```

2. Der Compiler erzeugt aus der Vorlage Quelltext für den erkannten Typ, indem der Platzhalter T durch den erkannten Typ ersetzt wird.

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
```

→

```
void swap (int& a, int& b) {
  int c = a;
  a = b;
  b = c;
}
```

3. Der Compiler benutzt diesen Quelltext, um den Funktionsaufruf zu übersetzen.

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
…
swap (r1, r2);
```

1. Compiler erkennt den Typ von r1, r2 (`double`)

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
…
swap (r1, r2);
```

→

```
void swap (double& a, double& b)
{
  double c = a;
  a = b;
  b = c;
}
```

2. Der Compiler erzeugt aus der Vorlage Quelltext für den erkannten Typ, indem der Platzhalter T durch den erkannten Typ ersetzt wird.

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
…
swap (r1, r2);
```

→

```
void swap (double& a, double& b)
{
  double c = a;
  a = b;
  b = c;
}
```

3. Der Compiler benutzt diesen Quelltext, um den Funktionsaufruf zu übersetzen.

Template-Instantiierung

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
swap (n1, n2);
…
swap (r1, r2);

swap (n1, r2); // ???
```

```
void swap (double& a, double& b)
{
  double c = a;
  a = b;
  b = c;
}
```

Fehlermeldung: keine passende Funktion gefunden, implizite Konversion greift bei Template-Funktionen nicht.

```
swap.cpp: In function 'int main()':
swap.cpp:22: error: no matching function for call to 'swap(int&, double&)'
```

Templates: Anforderungen

- Eine Template-Funktion stellt implizit **Anforderungen** an den Typ T

```
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
…
const int m1 = 1;
const int m2 = 1;
swap (m1, m2); // ???
```

Hier: Werte vom Typ T müssen zuweisbar sein (Standard-Jargon: "T is assignable")

```
swap.cpp: In function 'void swap(T&, T&)
[with T = const int]':
swap.cpp:26:   instantiated from here
swap.cpp:7: error: assignment of read-only
reference 'a'
swap.cpp:8: error: assignment of read-only
reference 'b'
```

Templates: Anforderungen

- Eine Template-Funktion stellt implizit **Anforderungen** an den Typ T
- Diese Anforderungen sollten dokumentiert sein:

```
// PRE: T is assignable
// POST: values of a and b are interchanged
template <typename T>
void swap (T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
```

Generisches Sortieren

- Zur Erinnerung: minimum-sort

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
void minimum_sort (int* first, int* last)
{
  for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p; // pointer to current minimum
    int* q = p;     // pointer to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

Mit dieser Funktion können nur `int`'s sortiert werden, die in einem Feld gespeichert sind.

Generisches Sortieren

- Generisches minimum-sort:

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
template <typename It>
void minimum_sort (It first, It last)
{
  for (It p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    It p_min = p; // pointer to current minimum
    It q = p;     // pointer to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

It ist Platzhalter für einen **Iteratortyp**

Anforderungen:
- Vergleichsoperatoren == und !=
- Dereferenzierungsoperator *
- Präinkrement-Operator ++

Standard-Jargon: "Forward Iterator"

Generisches Sortieren

- Generisches minimum-sort:

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
template <typename It>
void minimum_sort (It first, It last)
{
  for (It p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    It p_min = p; // pointer to current minimum
    It q = p;     // pointer to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

It ist Platzhalter für einen **Iteratortyp**

Anforderungen an den "value type" von `It`:
- assignable
- comparable

Generisches Sortieren

- Generisches minimum-sort:

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
template <typename It>
void minimum_sort (It first, It last)
{
  for (It p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    It p_min = p; // pointer to current minimum
    It q = p;     // pointer to current element
    while (++q != last)
      if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap (p, p_min);
  }
}
```

It ist Platzhalter für einen **Iteratortyp**

Erlaubt die Sortierung aller Container, die
- einen Forward Iterator anbieten
- zuweisbare und vergleichbare Daten enthalten

Generische Ausgabe

```
// PRE: [first, last) is a valid range
// POST: the elements of the range are written to std::cout
template <typename It>
void print(It first, It last)
{
  for (It j = first; j != last; ++j)
    std::cout << *j << " ";
  std::cout << std::endl;
}
```

`It` ist ein "forward iterator" (hier reicht sogar ein "input iterator", der jedes Element nur einmal anschauen darf)

Sortieren von ganzen Zahlen in Feldern

```
// sort array of numbers
int numbers[] = {2, 4, 0, -1, 5, 7, 0, -3, 10, 3};
minimum_sort (numbers, numbers+10);
print (numbers, numbers+10);
```

Instantiierung für T = `int*`

Ausgabe:

```
-3 -1 0 0 2 3 4 5 7 10
```

Sortieren von Strings in Feldern

```
// sort array of strings
std::string months[] = {
   "January",
   "February",
   "March",
   "April",
   "May",
   "June",
   "July",
   "August",
   "September",
   "October",
   "November",
   "December"
};
minimum_sort (months, months+12);
print (months, months+12);
```

Instantiierung für T = `std::string*`

Sortieren von Strings in Feldern

```
minimum_sort (months, months+12);
print (months, months+12);
```

Ausgabe:

```
April August December February January July June March May November
October September
```

`operator<` für Strings, der in `minimum_sort` beim Vergleich `*q < *p_min` benutzt wird, vergleicht lexikographisch wie im Wörterbuch.

Sortieren von ganzen Zahlen in Listen

```
// sort list of numbers
std::list<int> l;
for (int i=0; i<10; ++i)
  l.push_front(i);
minimum_sort (l.begin(), l.end());
print (l.begin(), l.end());
```

Instantiierung für T = `std::list<int>::iterator`

„Zeiger" auf das erste und hinter das letzte Element der Liste

Ausgabe:

```
0 1 2 3 4 5 6 7 8 9
```

Template-Klassen

- Nicht nur viele Funktionen, sondern auch viele Klassen braucht man für ganz verschiedene Typen
- Beispiel: Treaps mit Zahlen (Benutzer-Nummern), Treaps mit Strings (Wörter-buch)

Generischer TreapNode: `TreapNode<T>`

```
template <typename T>
class TreapNode
{
public:
  // constructor
  TreapNode(T key, int priority, TreapNode<T>* left = 0, TreapNode<T>* right = 0);

  // Getters
  const T& get_key() const;
  int get_priority() const;
  const TreapNode<T>* get_left() const;
  const TreapNode<T>* get_right() const;

private:
  T key_;
  int priority_;
  TreapNode<T>* left_;
  TreapNode<T>* right_;
  friend class Treap<T>;
};
```

Name des generischen Typs

Sonst genau gleich wie der alte TreapNode für `int`, ausser hier:
`const T&` anstatt `T`

Grund: `get_key()` ist dann auch effizient für „schwere" Typen mit aufwändigem Copy-Konstruktor

Generischer Treap

```
template <typename T>
class Treap {
public:
  // default constructor:
  // POST: *this is an empty tree
  Treap();

  // copy constructor
  // POST *this is a copy of other
  Treap(const Treap<T>& other);
  …
private:
  TreapNode<T>* root_;
  …
}
```

Sortieren mit Treaps

- Ausgabe der Schlüssel in „in-order"-Reihenfolge:
  - Schlüssel im linken Teilbaum, „in-order"
  - Wurzel
  - Schlüssel im rechten Teilbaum, „in-order"
- Das gibt die Schlüssel in sortierter Reihenfolge aus!

Sortieren mit Treaps

```
template <typename T>
void Treap<T>::print(std::ostream& o) const
{
  print(o, root_);
}
template <typename T>
void Treap<T>::print(std::ostream& o,
                     const TreapNode<T>* current) const
{
  if (current != 0) {
    print(o, current->get_left());
    o << current->get_key() << " ";
    print(o, current->get_right());
  }
}
```

Öffentliche Mitgliedfunktion für „in-order"-Ausgabe

Sortieren mit Treaps

```
template <typename T>
void Treap<T>::print(std::ostream& o) const
{
  print(o, root_);
}
template <typename T>
void Treap<T>::print(std::ostream& o,
                     const TreapNode<T>* current) const
{
  if (current != 0) {
    print(o, current->get_left());
    o << current->get_key() << " ";
    print(o, current->get_right());
  }
}
```

Private Mitgliedfunktion für „in-order"-Ausgabe eines Teilbaums

Sortieren mit Treaps - Zahlen

```
// sort some numbers
int numbers[] = {2, 4, 0, -1, 5, 7, 0, -3, 10, 3};
Treap<int> tn;
for (int i=0; i<10; ++i)
  tn.insert (numbers[i]);
tn.print(std::cout); std::cout << std::endl;
```

Instantiierung für T = `int`

Ausgabe:

```
0 1 2 3 4 5 6 7 8 9
```

Sortieren mit Treaps - Strings

```
// sort some strings
std::string months[] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};
Treap<std::string> ts;
for (int i=0; i<12; ++i)
  ts.insert (months[i]);
ts.print(std::cout); std::cout << std::endl;
```

Instantiierung für T = `std::string`

Sortieren mit Treaps - Strings

```
Treap<std::string> ts;
for (int i=0; i<12; ++i)
  ts.insert (months[i]);
ts.print(std::cout); std::cout << std::endl;
```

Ausgabe:

```
April August December February January July June March May November
October September
```

Ganze Zahlen als Template-Parameter

```
#include<iostream>

// POST: returns n % m
template <int m>
int mod (int n)
{
  return n % m;
}

int main ()
{
  std::cout << mod<7>(10) << std::endl; // 3

  return 0;
}
```

Nerd-Alarm: Fibonacci-Zahlen zur Kompilierungsszeit

- Ziel: Definition einer Template-Klasse **`fib<n>`** mit einem konstanten Datenmitglied, dessen Wert die n-te Fibonacci-Zahl ist.

```
struct A {
  static const int value = 5;
};
int main ()
{
  std::cout << A::value << std::endl; // 5
}
```

Daten-Mitglied, das der Klasse selbst gehört und nicht den Objekten der Klassse.

Nerd-Alarm: Fibonacci-Zahlen zur Kompilierungsszeit

```
// general template
template <int n>
struct fib {
  static const unsigned int value =
    fib<n-1>::value + fib<n-2>::value;
};
// specialized templates, n = 0, 1
template <>
struct fib<0> {
  static const unsigned int value = 0;
};
template <>
struct fib<1> {
  static const unsigned int value = 1;
};
```

`fib<n>` ist eine andere Klasse als `fib<n-1>` und `fib<n-2>`

Template-Spezialisierung: eigene Klassendefinitionen für die Fälle `n=0` und `n=1`

Nerd-Alarm: Fibonacci-Zahlen zur Kompilierungsszeit

```
int main() {
  std::cout << fib<10>::value << std::endl; // 55

  return 0;
}
```

`Fib<10>` ist eine Klasse, die vom Compiler erstellt wurde; insbesondere hat er dabei das Datenmitglied value berechnet. Zur Laufzeit des Programms passiert gar nichts mehr!

Nerd-Alarm: Ackermann-Zahlen zur Kompilierungsszeit

$$A(m,n) = \begin{cases} n+1, & \text{falls } m = 0 \\ A(m-1,1), & \text{falls } m > 0,\ n = 0 \\ A(m-1, A(m,n-1)), & \text{falls } m > 0,\ n > 0 \end{cases}$$

- ist *berechenbar*, aber *nicht primitiv rekursiv* (man dachte Anfang des 20. Jahrhunderts, dass es diese Kombination gar nicht gibt)
- wächst *extrem* schnell

Nerd-Alarm: Ackermann-Zahlen zur Kompilierungsszeit

m \ n	0	1	2	3	...	*n*
0	1	2	3	4	...	*n+1*
1	2	3	4	5	...	*n+2*
2	3	5	7	9	...	*2n+3*
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{6553}}-3$	...	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3$

Berechnung: Dieser Wert ging gerade noch...

Nerd-Alarm: Ackermann-Zahlen zur Kompilierungsszeit

```
//general template
template <int m, int n>
struct ackermann {
  static const unsigned int value =
    ackermann<m-1, ackermann<m,n-1>::value>::value;
};

// specialized template, m = n = 0
template <>
struct ackermann<0,0> {
  static const unsigned int value = 1;
};

// specialized templates, m = 0
template <int n>
struct ackermann<0,n> {
  static const unsigned int value = n+1;
};

// specialized templates, n = 0
template <int m>
struct ackermann<m,0> {
  static const unsigned int value =
    ackermann<m-1,1>::value;
};
```

Versuchen wir den Wert, der gerade noch ging...

```
int main() {
  std::cout << ackermann<4,1>::value << std::endl;

  return 0;
}
```

Fazit: Compiler braucht „ewig".