32 String

Dieses Kapitel behandelt die folgenden Themen:

- Konstruktor und Basismethoden
- Anhängen, Einfügen, Ersetzen
- Suchen und Finden
- Numerische Umwandlungen

Die C++-Stringklasse (Header `<string>`) kann in den meisten Fällen die C-Strings ersetzen. Sie ist nicht nur verständlicher in der Anwendung für Neulinge, sondern auch erheblich komfortabler. Der in der Standardbibliothek definierte Typ `basic_string` ist ein Template und ermöglicht das Arbeiten mit verschiedenen Arten von Zeichen, also auch mit »wide characters« (Typ `wchar_t`). `wstring` ist die Spezialisierung für `wchar_t`. Der Typ `string` ist die Spezialisierung von `basic_string` für den Datentyp `char`:

```
typedef basic_string<char> string;
```

Die im Folgenden beschriebene Schnittstelle der Klasse `basic_string` bezieht sich der Kürze wegen nur auf die Spezialisierung `string`. Die von `string` bereitgestellten öffentlichen Datentypen korrespondieren mit denen der Tabelle 28.2 auf Seite 765. Der Typ `string::size_type` wird im Folgenden kurz `size_type` genannt. Zunächst folgen die Konstruktoren sowie diejenigen Methoden, die in ähnlicher Form auch in anderen Containern der C++-Standardbibliothek auftreten.

- `string()` ist der Standardkonstruktor. Er erzeugt einen leeren String.
- `string(const string& s, size_type pos = 0, size_type n = string::npos)`
  Der Kopierkonstruktor erzeugt einen String, wobei `s` ab Position `pos` bis zum Ende kopiert wird. Dabei gilt die Einschränkung, dass maximal `n` Zeichen kopiert werden. `string::npos` ist eine –1 konvertiert in den `unsigned`-Typ `size_type`, also die größtmögliche `unsigned`-Zahl.
- `string(const char* s, size_type n)`
  Bei der Erzeugung des Strings werden `n` Zeichen aus dem bei `s` beginnenden Array kopiert.
- `string(const char* s)` erzeugt String aus dem C-String `s`.
- `string(size_type n, char c)` erzeugt String mit `n` Kopien von `c`.
- `template<InputIterator> string(InputIterator a, InputIterator b)`
  Falls `InputIterator` ein integraler Typ ist, entspricht dieser Konstruktor dem vorhergehenden (`string(size_type n, char c)`), wobei `a` und `b` in die entsprechenden Typen `size_type` und `char` umgewandelt werden. Andernfalls wird der String aus den Zeichen im Intervall `[a,b)` gebildet.
- `~string()` Destruktor
- `const_iterator begin() const` und
  `iterator begin()`
  geben den Anfang des Strings zurück.
- `const_iterator end() const` und
  `iterator end()`
  geben die Position *nach* dem letzten Zeichen zurück.
- `const_iterator rbegin() const` und
  `iterator rbegin()`
  geben einen Iterator zurück, der auf das letzte Zeichen zeigt.
- `const_iterator rend() const` und
  `iterator rend()`
  geben einen Iterator zurück, der auf die Position vor dem Anfang zeigt.
- `size_type size() const`
  gibt die aktuelle Größe des Strings zurück (Anzahl der Zeichen).
- `size_type length() const` ist dasselbe wie `size()`.
- `resize(size_type n, char c = '\0')`
  Der String wird durch eine auf `n` Zeichen verkürzte Kopie ersetzt, falls $n \leq$ `size()` ist. Andernfalls wird der String durch eine auf `n` Zeichen vergrößerte Kopie ersetzt, wobei die restlichen Elemente mit `c` initialisiert werden.
- `void reserve(size_type n = 0)`
  Speicherplatz reservieren, sodass der verfügbare Platz (Kapazität) größer als der aktuell benötigte ist. Zweck: Vermeiden von Speicherbeschaffungsoperationen während der Benutzung des Strings.
- `void shrink_to_fit()`
  Wenn keine weiteren Operationen mit dem String mehr zu erwarten sind, reduziert der Aufruf dieser Funktion den Speicherplatz auf das Notwendige.

- `size_type capacity() const`
  gibt die Größe des dem String zugewiesenen Speichers zurück. Der Wert ist größer oder gleich dem Argument von `reserve()`, falls `reserve()` vorher aufgerufen wurde.
- `void clear()` löscht den Inhalt des Strings; entspricht `erase(begin(), end())`.
- `bool empty() const` gibt `size() == 0` bzw. `begin() == end()` zurück.
- `const_reference operator[](size_type n) const` und
  `reference operator[](size_type n)`
  geben eine Referenz auf das n-te Zeichen zurück.
- `const_reference at(size_type n) const` und
  `reference at(size_type n)`
  geben eine Referenz auf das n-te Zeichen zurück, wobei die Gültigkeit des Arguments geprüft wird. Es wird eine `out_of_range`-Exception geworfen, falls `n` $\geq$ `size()` ist.
- `size_type copy(char* z, size_type n, size_type pos = 0) const`
  überschreibt das Array `z` ab `pos` mit den Zeichen des Strings, aber maximal `n`. Das Stringendezeichen wird nicht kopiert. Es wird vorausgesetzt, das `z` auf einen Bereich mit genug Platz verweist. Die Methode gibt die Anzahl der kopierten Zeichen zurück.
- `void swap(const string& s)` vertauscht den Inhalt der beiden Strings.
- `const char* c_str()` und
  `const char* data()`
  geben einen Zeiger `z` auf das erste der intern gespeicherten Zeichen zurück. Für alle weiteren Positionen `i` im Bereich `[0, size()]` gilt entsprechend `p + i = &operator[](i)`.
- `const char& front() const` und
  `char& front()`
  geben eine Referenz auf das erste Zeichen zurück, d.h. `operator[](0)`.
- `const char& back() const` und
  `char& back()`
  geben eine Referenz auf das letzte Zeichen zurück, d.h. `operator[](size()-1)`.

Interessanter sind die Methoden, die in anderen Containern nicht vertreten und speziell zur Bearbeitung von Zeichenketten geeignet sind, zum Beispiel Finden eines Substrings. Die wichtigsten sind im Folgenden aufgeführt.

Zuweisen und Anhängen

- `string& append(const string& s)`
  `string& append(const char* s)`
  `string& operator+=(const string& s)`
  `string& operator+=(const char* s)`
  `string& operator+=(char c)`
  verlängern den String um den C-String oder String `s` bzw. das Zeichen `c`.
- `string& append(const string& s, size_type pos, size_type n)`
  Von der Position `pos` des Strings `s` bis zum Ende wird alles an den String angehängt, aber nicht mehr als `n` Zeichen.
- `string& append(const char* s, size_type n)` verlängert den String um `string(s,n)`.
- `string& append(size_type n, char c)` verlängert den String um `string(n,c)`.

- `void push_back(char c)` bewirkt dasselbe wie `append(1, c)`.
- `string& assign(const string& s)`
  `string& assign(const char* s)`
  `string& operator=(const string& s)`
  `string& operator=(const char* s)`
  `string& operator=(char c)`
  weisen dem String den C-String oder String `s` bzw. das Zeichen `c` zu.
- `string& assign(const string& s, size_type pos, size_type n)`
  Dem String wird der String `s` von der Position `pos` des Strings `s` an bis zum Ende zugewiesen, aber nicht mehr als `n` Zeichen. Die vorher beschriebene Funktion `assign(s)` entspricht `assign(s, 0, string::npos)`.

Einfügen

- `string& insert(size_type pos, const char* s)`
  `string& insert(size_type pos, const string& s)`
  C-String bzw. String `s` vor der Stelle `pos` einfügen (das heißt am Anfang, falls `pos` gleich 0).
- `string& insert(size_type pos1, const string& s,`
  `size_type pos2, size_type n)`
  Vor die Stelle `pos1` wird der String `s` eingefügt, wobei an der Position `pos2` von `s` begonnen wird und insgesamt nicht mehr als `n` Zeichen kopiert werden.
- `string& insert(size_type pos, const char* s, size_type n)`
  wirkt wie `insert(pos, string(s,n))`.
- `string& insert(size_type pos, size_type n, char c)`
  wirkt wie `insert(pos, string(n,c))`.
- `template<InputIterator>`
  `void insert(iterator p,InputIterator first, InputIterator last)`
  `p` ist ein Iterator des Strings selbst, `first` und `last` sind Iteratoren eines anderen Strings oder Containers. Die Funktion bewirkt das Einfügen der Zeichen im Bereich `[first, last)` vor der Stelle `p`.
- `iterator insert(iterator p, char c)`
  `p` ist ein Iterator des Strings selbst. `c` wird vor der Stelle `p` eingefügt.
- `iterator insert(iterator p, size_type n, char c)`
  `p` ist ein Iterator des Strings selbst. `n` Kopien von `c` werden vor der Stelle `p` eingefügt.

Löschen und Ersetzen

- `iterator erase(iterator p)`
  Zeichen an der Stelle `p` löschen. Zurückgegeben wird die Position direkt vorher, sofern sie existiert (andernfalls `end()`).
- `iterator erase(iterator p, iterator q)`
  Zeichen im Bereich `p` bis ausschließlich `q` löschen.
- `string& erase(size_type pos = 0, size_type n = string::npos)`
  löscht alle Zeichen ab der Stelle `pos`, aber nicht mehr als `n` Zeichen.

- `string& replace(size_type pos1, size_type n1, const string& s, size_type pos2, size_type n2)`
  Alle Zeichen des Strings ab der Stelle `pos1`, aber maximal `n1` Zeichen, werden entfernt. An dieser Stelle werden alle Zeichen des Strings `s` ab der Stelle `pos2`, aber maximal `n2` Zeichen, eingefügt.
- `string& replace(size_type pos, size_type n, const string& s)`
  `string& replace(size_type pos, size_type n, const char* s)`
  Alle Zeichen des Strings ab der Stelle `pos`, aber maximal `n` Zeichen, werden entfernt. An dieser Stelle werden alle Zeichen des Strings bzw. C-Strings `s` eingefügt.
- `string& replace(size_type pos, size_type n1, const char* s, size_type n2)`
  wirkt wie `replace(pos, n1, string(s, n2))`.
- `string& replace(size_type pos, size_type n1, size_type n2, char c)`
  wirkt wie `replace(pos, n1, string(n2, c))`.
- `string& replace(iterator p, iterator q, const string& s)`
  `string& replace(iterator p, iterator q, const char* s)`
  Bereich zwischen `p` und ausschließlich `q` durch `s` ersetzen.
- `string& replace(iterator p, iterator q, const char* s, size_type n)`
  wirkt wie `replace(p, q, string(s, n))`.
- `string& replace(iterator p, iterator q, size_type n, char c)`
  wirkt wie `replace(p, q, string(n, c))`.
- `template<class InputIterator>`
  `string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2)`
  wirkt wie `replace(i1, i2, string(j1, j2))`.

Suchen und Finden

- `size_type find(const string& s, size_type pos = 0) const`
  gibt die Position zurück, an der der Substring `s` gefunden wird, andernfalls wird `string::npos` zurückgegeben. Gesucht wird ab `pos`. `find()` findet die erste Position bei mehrfachem Vorkommen von `s`.
- `size_type find(const char* s, size_type pos, size_type n) const`
  gibt `find(string(s, n), pos)` zurück.
- `size_type find(const char* s, size_type pos = 0) const`
  gibt `find(string(s), pos)` zurück.
- `size_type find(char c, size_type pos = 0) const`
  gibt `find(string(1,c), pos)` zurück.
- `size_type rfind(const string& s, size_type pos = string::npos) const`
  Gibt die Position zurück, an der der Substring `s` gefunden wird, andernfalls wird `string::npos` zurückgegeben. Gesucht wird ab `pos`. `rfind()` findet die letzte Position bei mehrfachem Vorkommen von `s`.
- `size_type rfind(const char* s,size_type pos,size_type n) const`
  gibt `rfind(string(s, n), pos)` zurück.
- `size_type rfind(const char* s, size_type pos = string::npos) const`
  gibt `rfind(string(s), pos)` zurück.

- `size_type rfind(char c, size_type pos = string::npos) const`
  gibt `rfind(string(1,c), pos)` zurück.
- `size_type find_first_of(const string& s, size_type pos=0) const` und
  `size_type find_first_of(const char* s, size_type pos = 0) const`
  geben die erste Position zurück, an der ein Zeichen gefunden wird, das auch im String bzw. C-String `s` vorhanden ist, andernfalls wird `string::npos` zurückgegeben. Gesucht wird ab `pos`.
- `size_type find_first_of(const char* s, size_type pos, size_type n) const`
  gibt `find_first_of(string(s,n), pos)` zurück.
- `size_type find_first_of(char c, size_type pos = 0) const`
  gibt `find_first_of(string(1,c), pos)` zurück.
- `size_type find_last_of(const string& s, size_type pos = string::npos) const`
  `size_type find_last_of(const char* s, size_type pos = string::npos) const`
  `size_type find_last_of(const char* s, size_type pos, size_type n) const`
  `size_type find_last_of(char c, size_type pos = string::npos) const`
  Diese Funktionen entsprechen `find_first_of()` mit dem Unterschied, dass jeweils die letzte gefundene Position zurückgegeben wird.
- `size_type find_first_not_of(const string& s, size_type pos = 0) const`
  `size_type find_first_not_of(const char* s, size_type pos = 0) const`
  `size_type find_first_not_of(const char* s, size_type pos, size_type n) const`
  `size_type find_first_not_of(char c, size_type pos = 0) const`
  Diese Funktionen entsprechen `find_first_of()` mit dem Unterschied, dass jeweils die erste Position zurückgegeben wird, an der ein Zeichen steht, das *nicht* in `s` vorkommt bzw. das nicht `c` entspricht. Wenn so eine Position nicht gefunden wird, wird `string::npos` zurückgegeben.
- `size_type find_last_not_of(const string& s, size_type pos = string::npos) const`
  `size_type find_last_not_of(const char* s, size_type pos = string::npos) const`
  `size_type find_last_not_of(const char* s, size_type pos, size_type n) const`
  `size_type find_last_not_of(char c, size_type pos=string::npos) const`
  Diese Funktionen entsprechen `find_first_not_of()` mit dem Unterschied, dass jeweils die letzte gefundene Position zurückgegeben wird.

Substrings und Vergleiche

- `string substr(size_type pos = 0, size_type n = string::npos) const`
  Gibt den Substring zurück, der ab `pos` beginnt. Die Anzahl der Zeichen im Substring wird durch das Ende des Strings bestimmt, kann aber nicht größer als `n` werden.
- `int compare(const string& s) const`
  vergleicht zeichenweise die Strings `*this` und `s`. Es wird 0 zurückgegeben, wenn keinerlei Unterschied festgestellt wird. Falls das erste unterschiedliche Zeichen von `*this` kleiner als das entsprechende in `s` ist, wird eine negative Zahl zurückgegeben, andernfalls eine positive. Falls bei unterschiedlicher Länge bis zum Ende eines der Strings keine verschiedenen Zeichen gefunden werden, wird eine negative Zahl zurückgegeben, falls `size() < s.size()` ist, andernfalls eine positive Zahl.

- `int compare(size_type pos, size_type n, const string& s) const`
  gibt `string(*this,pos,n).compare(s)` zurück. Das heißt: Es werden nur die Zeichen ab Position `pos` in `*this` berücksichtigt, aber maximal `n`.
- `int compare(size_type pos1, size_type n1, const string& s,`
  `size_type pos2, size_type n2) const`
  gibt `string(*this,pos1,n1).compare(string(s,pos2,n2))` zurück, Das heißt, es werden nur die Zeichen ab Position `pos1` in `*this` berücksichtigt, aber maximal `n1`. In `s` werden nur die Zeichen ab Position `pos2` berücksichtigt, aber maximal `n2`.

Numerische Umwandlungen

Die Funktionen zur Umwandlung eines Strings in eine Zahl bzw. umgekehrt sind keine Methoden der Klasse `string`, sondern im Namespace `std` [ISOC++].

```
int stoi(const string& str, size_t *idx = 0, int base = 10)
long stol(const string& str, size_t *idx = 0, int base = 10)
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10)
long long stoll(const string& str, size_t *idx = 0, int base = 10)
unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10)
float stof(const string& str, size_t *idx = 0)
double stod(const string& str, size_t *idx = 0)
long double stold(const string& str, size_t *idx = 0)
```

Ein Anwendungsbeispiel finden Sie auf Seite 625. Die Funktionen benutzen intern die C-Funktionen `strtol()`, `strtod()` usw., deren Wirkungsweise auf Seite 625 beschrieben wird. `base` ist die gewünschte Zahlenbasis. Falls der Zeiger `idx` ungleich 0 ist, wird der Index des ersten nicht umgewandelten Zeichens in `*idx` abgelegt. Eine Alternative zu diesen Funktionen ist `boost::lexical_cast<T>(arg)` (Seite 627). Zur Umwandlung von Zahlen in einen String gibt es die Funktionen

```
string to_string(X)
```

wobei `X` für einen der Zahl-Typen `int`, `float`, `double` usw. steht, einschließlich der `unsigned` und `long`-Varianten.

Binäre Operatoren

Darüber hinaus gibt es einige Funktionen, die mit Strings arbeiten, aber *keine* Elementfunktionen sind:

- `string operator+(const string&, const string&)`
  `string operator+(const string&, const char*)`
  `string operator+(const char*, const string&)`
  Diese Operatoren verketten zwei Strings (bzw. einen String und einen C-String oder umgekehrt) und geben das Ergebnis zurück.
- `string operator+(const string&, char)`
  `string operator+(char, const string&)`
  Diese Operatoren verketten einen String mit einem Zeichen und geben das Ergebnis zurück.
- `bool operator==(X, Y)`
  `bool operator!=(X, Y)`

```
bool operator<=(X, Y)
bool operator>=(X, Y)
bool operator<(X, Y)
bool operator>(X, Y)
```

sind die relationalen Operatoren zum Vergleichen von Strings. `X` und `Y` stehen hier für jeweils einen der Typen `const string&` oder `const char*`. Es sind drei Kombinationen für `X` und `Y` möglich:

```
const string&, const string&
const char*, const string&
const string&, const char*
```

- `istream& operator>>(istream&, string&)`
  Dieser Operator erlaubt das Einlesen von Strings auf bequeme Weise. Die üblichen Eigenschaften des `>>`-Operators werden beibehalten (vgl. Seite 94).
- `ostream& operator<<(ostream&, string&)` Ausgabeoperator für Strings.
- `istream& getline(istream& is, string& s, char ende ='\n')`
  Liest Zeichen für Zeichen aus der Eingabe `is` in den String `s`, bis das Zeichen `ende` gelesen wird. `ende` wird zwar gelesen, aber *nicht* an den String angehängt (vergleiche `getline()` auf Seite 382).