§5 Templates – Allgemeines

Leitideen: *Templates dienen zur Spezifikation datentypabhängiger Klassen und Funktionen. Erzeugt werden nur diejenigen Klassen und Funktionen, die tatsächlich benötigt werden (Templateauswertung).*

Voreinstellungen für Templateparameter und die Ermittlung von Templateparametern aus Funktionsargumenten erleichtern den Umgang mit Templates.

Es gibt Mechanismen zur Sonderbehandlung von Templates mit speziellen Typparametern. (Templatespezialisierung).

Templates in Klassen (member templates) erlauben die Konstruktion passender Unterstützungsklassen, z.B. Iteratoren für Behälter.

Der Einsatz dieser Sprachelemente wird exemplarisch an einzelnen Klassen- und Funktionstemplates der STL dargestellt.

§5 Templates – Themenübersicht

- Templatevereinbarung I,II
- Templateauswertung
- Templatespezialisierung
- Beispiel: Konjugierte Gradienten
- Beispiel: Ringe
- Templates in Klassen
- Funktionsobjekte in der STL - Allgemeines
- Funktionsobjekte in der STL - Beispiel map I,II
- Funktionstemplates in der STL - Beispiele I,II,III,IV
- Mengen und Mengenoperationen in der STL
- Stromklassen als STL-Templates I,II,III
- Unicode (UCS-4) und UTF-8
- Stringklassen als STL-Templates

Templatevereinbarung

Begriff: Templates („Vorlagen“) definieren Familien von Klassen oder Funktionen

- Zwei Arten: Klassentemplates, Funktionstemplates
- Sowohl Templatedefinitionen als auch Templatedeklarat.
- Zwei Arten von Template-Parametern:

Typparameter	beliebiger Datentyp (nicht nur Klassen)
Nichttypparameter	Konstanten (ganzzahlig)
“	(Zeiger auf externe Funktionen oder externe Objekte)

- Typparametern wird `class` oder `typename` vorangestellt (*kein* Bedeutungsunterschied!)
- Nichttypparameter: Normale Deklarationssyntax
- Voreinstellungen für Template-Parameter möglich
  Bsp.: Klassentemplatedeklaration für vector

```
template <class T, class Allocator =
                        allocator<T>> class vector;
```

Templatevereinbarung - Fortsetzung

- Innerhalb einer Klassentemplatevereinbarung: Komp.funktionen sind automat. Funktionstemplates.
- Gilt *nicht* für `friend`-Funktionen.
- Innerhalb einer Klassentemplatevereinbarung können bei der Klassenangabe die Template-Parameter weggelassen werden.

Weitere Anmerkungen zu den array-Beispielen

Ziel: C-Vektoren mit (eingeschränkter) Behälterfunktionalität

- `array` enthalten in C++11/14 (Header `<array>`)
- Standardnamen wie `iterator` per typedef
- `instance` - Name des internen C-Vektors
- Hier verwendete Namen vereinfacht ggü. Implementierung von Boost, dort z.B. `_M_instance` statt `instance`
- Funktionstemplate `two_norm` als Anwendungsbeispiel Besonderheit: `typename` in for-Schleife

Templateauswertung

Begriff: Einsetzen konkreter Templateargumente führt zur Erzeugung der entsprechenden Klasse bzw. Fkt. (template instantiation)

- Erzeugung nur im Bedarfsfall
- Bei Nichttypparametern: Genaue Übereinstimmung des Parameterdatentyps mit Argumenttyp erforderlich
  Jedoch Integererweiterung (und andere triviale Typumw.)
  (Sonst `array<float,2ul>` statt `array<float,2>`)
- Funktionen: Ermittlung der Templatearg. über Fkt.arg.
  Bsp.: `two_norm(x)` statt `two_norm<float,2>(x)`

Anmerkungen zum myarray-Beispiel

- Zweck: Herausfinden, welche Klassen und Funktionen tatsächlich erzeugt werden
- Vorgehensweise:
  1. Konstruktor gibt Typinf. aus (`<typeinfo>`) $\rightarrow$ Klassen `myarray` erbt Komp. von `array`, Konstr. wird hinzugefügt
  2. Objektcodedatei mit nm analysieren (Linux) $\rightarrow$ Funktionen

Template-Spezialisierung

Zweck: Sonderbehandlung für einzelne Datentypen

- Ersetzen von Templateparametern durch zulässige Argumente, dazu gehören auch Templateausdrücke:
  Bsp.: `vector<T,A>` → `vector<bool,A>`
  `iterator_traits<T>` → `iterator_traits<T*>`
- Syntax: Templatevereinbarung, bei der *Templatename* durch *Templatename*`<...>` ersetzt wird
  (z.B. Ersetzen von `vector` durch `vector<bool,A>` bzw. `iterator_traits` durch `iterator_traits<T*>`)
- Spezialisierung von Funktionstemplates analog
  Bsp.: `swap<T>` → `swap<vector<T,A>>`
- Unterschied: Templateausdrücke nach dem Funktionsnamen überflüssig, wenn aus Argumenttypen erschließbar
- Falls alle Template-Parameter durch feste Typen oder konstante Ausdrücke ersetzt werden, beginnt die Templatevereinbarung mit `template<>`

Verfahren der konjugierten Gradienten

Aufgabenstellung

Sei $A \in \mathbb{R}^{n \times n}$ symmetrisch, positiv definit, $b \in \mathbb{R}^n$.
Bestimme Lösung $x \in \mathbb{R}^n$ von $Ax = b$.

Verfahren

$x_0 \in \mathbb{R}^n$, $r_0 = b - Ax_0$, $p_0 = r_0$, $\rho_0 = r_0^\top r_0$

$$\left.\begin{aligned} q_k &= Ap_k \\ \alpha_k &= \frac{\rho_k}{p_k^\top q_k} \\ x_{k+1} &= x_k + \alpha_k p_k \\ r_{k+1} &= r_k - \alpha_k q_k \\ \rho_{k+1} &= r_{k+1}^\top r_{k+1} \\ \beta_k &= \frac{\rho_{k+1}}{\rho_k} \\ p_{k+1} &= r_{k+1} + \beta_k p_k \end{aligned}\right\} \quad k = 0, 1, \ldots, n-1 \text{ , falls } \rho_k \neq 0$$

Falls $r_k = 0$ (d.h. ρ_k=0), gilt $x = x_k$.
Das ist spätestens der Fall, wenn $k = n$ (bei exakter Rechn.!).

Anmerkungen zum Ring-Beispiel

- Ring $(R, +, \cdot)$, d.h. $(R, +)$ Gruppe mit neutralem Element 0, $(R, \cdot)$ Halbgruppe
- Nullteiler: $a \neq 0$, $b \neq 0$, $a \cdot b = 0$
  a linker, b rechter Nullteiler
  Ziel: Bestimmung aller linken Nullteiler von R
- Ring Klasse, Ringelemente Objekte
- Operationen: Addition `+`, Multiplikation `*`, Ausgabe (`<<`)
- Null: Statische Komponentenfunktion `null()`
- Durchwandern von R mit Cursoren (statt Iteratoren):
  ```
  for (r=R::start(); r!=R::stop(); ++r)
  ```
  Präfixoperator `++` und stat. Komp.fkt. `start()`, `stop()`
  Vergleiche `==` und `!=`

Anmerkungen zum Ring-Beispiel - Fortsetzung

Zwei Ausprägungen

- $\mathbb{Z}_n = \{0, \ldots, n-1\}$, Addition und Multiplikation modulo n
  `null()`, `start()`: liefern 0
  `stop()`: liefert n (keine Modulorechnung)
- Produktring $R \times S = \{(r, s) : \ r \in R, s \in S\}$
  Addition und Multiplikation komponentenweise
  `null()`: liefert `(R::null(),S::null())`
  `start()`: liefert `(R::start(),S::start())`
  `stop()`: liefert `(R::stop(),S::stop())`
  Modifikation bei `++`: r variiert langsam, s schnell in (r, s)
  Falls `S::stop()` erreicht, Zurücksetzen auf `S::start()` und Erhöhen von `r`

Templates in Klassen (member templates)

- Template-Parameter: explizite (eigene) + implizite (vom umschließenden Klassentemplate)

Bsp.: Template-Deklarationen in `<vector>`

```
template <class T, class A> class vector {
  public:
    template <class It>
      vector(It anf, It end, const A&=A());
    // Konstruktortemplatedef., daher folg. Vereinb. mögl.:
    // vector<T> a(anf,end);
              :
    template <class It> void insert
      (iterator pos, It anf, It end)
    // Funktionstemplatedefinition, ermöglicht:
    // a.insert(pos,anf,end)
              :
}
```

Funktionsobjekte in der STL - Allgemeines

- Einsatz in Behälterklassen (z.B. map) und in Algorithmen (z.B. sort)
- *Denn:* Funktorklassen leicht als Templateparameter verwendbar
- Objekte einer Funktorklasse dienen z.B. zur Definition einer Ordnung
- Definition von Funktorklassen kaum aufwendiger als Definition entsprechender Funktionen
- Häufig Benutzung des vom Standardkonstruktor bereitgestellten Funktionsobjekts
  (Syntax etwas gewöhnungsbedürftig!)
- Wegen Inlining Funktionsobjekte oft effizienter bzgl. Laufzeit als Funktionen, allerdings bei Behälterklassen Gefahr der Code-Aufblähung (code bloat)
- STL-Algorithmen mit Funktionsparameter akzeptieren sowohl Funktionen als auch Funktionsobjekte*(später!)*

Funktionsobjekte in der STL - Beispiel map

`map<I,T> a`

- `a` aufsteigend geordnet entspr. $<$ von `I`
  (Z.B. beim Durchlauf von [`a.begin(),a.end()`
- Nachteil: Weiterer Datentyp `I` erforderlich bei anderer Ordnung (lästig v.a. bei eingebauten Datentypen wie `string`)

`map<I,T,Comp> a`

- `a` aufsteigend geordnet entspr. $<$ def. durch `Comp()`
  (Standard-Funktionsobjekt der Funktorklasse `Comp`)
- Voreinstellung für `Comp`: `less<I>`

STL-Funktorklassen für Vergleiche

- *Zweck:* Übergabe bereits def. Vergleichsop. in Templates
  Bsp.: `map<I,T,greater<I>>`
- Bereitgestellt in `<functional>`: `less greater less_equal greater_equal equal_to not_equal_to`

Funktionsobjekte in der STL - Beispiel map II

`map<I,T,Comp> a(comp)`

- `a` aufsteigend geordnet entspr. $<$ def. durch `comp` (Funktionsobjekt `comp` aus Funktorklasse `Comp` als Konstruktorargument)
  Vorteil: Vermeidet Code-Aufblähung, falls verschiedene `comp` aus der gleichen Funktorklasse `Comp`
- Sogar: Einsetzen von Funktionen möglich (mit Hilfe von Funktionszeigertypen *ohne* Definition einer Funktorklasse!)

Abschließende Bemerkung

- Unterschied zwischen Funktionsobjekten und Funktorklassen genau beachten!
  Bsp: `less<int>()` Fktobj. `less<int>` Funktorklasse
- Behälterklassen: oft Funktorklasse als zusätzliches Templateargument
- Funktionen bzw. Funktionstemplates in `<algorithm>`: häufig Funktionsobjekt als zusätzliches Argument, z.B. `sort(a.begin(),a.end(),greater<double>())`

Funktionstemplates in der STL - Beispiele

Zweck: Erläuterung der Funktionsweise - kein systematischer Durchgang durch die STL

max_element

- `max_element(anf,end)` Liefert Pos. des größten El. Ordnung durch $<$ festgelegt
  `max_element(anf,end, cmp)` Liefert Pos. des größten El. Ordn. durch Fkt.obj. `cmp`
- *Bsp.:* `*max_element(a.begin(),a.end())` Max.wert der Elemente im Iteratorbereich
- Iteratorbereich: [*anf*, *end*[halboffenes Intervall (obwohl in STL-Quellen oft als `first` und `last` bezeichnet)
- Leerer Container: Dennoch Rückgabewert! (STL: `a.begin()==a.end()`, falls `a` leerer Container)

Funktionstemplates in der STL - Beispiele II

copy

- `copy(anf1,end1,anf2)`
  Kopiert Elemente im Iteratorbereich $[anf1, end1[$
  auf Elemente im Iteratorbereich ab *anf2*.
  Liefert Position *nach* letztem kopierten Element.
- Erforderlich: Hinreichend großer Zielbereich oder
  Verwendung eines *einfügenden* Iterators
- Beispiele für einfügende Iteratoren:

```
vector<double> a(n),b;
copy(a.begin(),a.end(),back_inserter(b));
list<double> c;
copy(a.begin(),a.end(),front_inserter(c));
copy(c.begin(),c.end(),
     ostream_iterator<double>(cout," "));
```

Funktionstemplates in der STL - Beispiele III

unique_copy, unique

- `unique_copy(anf1,end1,anf2)`
  Kopiert Elemente im Iteratorbereich [*anf*1, *end*1[
  auf Elemente im Iteratorbereich ab *anf2* und entfernt dabei aufeinanderfolgende Duplikate.
  Liefert Position *nach* letztem kopierten Element.
- Erforderlich: Hinreichend großer Zielbereich oder Verwendung eines *einfügenden* Iterators
- `unique(anf1,end1)`
  Entfernt aufeinanderfolgende Duplikate im Iteratorbereich [*anf*1, *end*1[durch Überschreiben, verkürzt aber den Iteratorbereich *nicht*.
  Liefert Position des ersten überschüssigen Elements.

```
vector<double> a(n)
pos=unique(a.begin(),a.end());
a.erase(pos,a.end());
```

  Andere löschende Algorithmen (`remove`, `remove_if`) verkürzen den Iteratorbereich ebenfalls nicht!.

Funktionstemplates in der STL - Beispiele IV

Häufig benutzte STL-Funktionstemplates

- `min max swap reverse sort sort_stable`
- Bei `min` und `max` Typgleichheit beider Argumente notw.
- Bei Listen ist die Komp.fkt. `reverse` evtl. schneller und daher vorzuziehen. Gilt auch für andere STL-Funktionen.
- `sort` (Quicksort) ist im Durchschnitt schneller als `stable_sort`, aber in Ausnahmefällen viel schlechter ($O(n^2)$ statt $O(n \ln n)$).
- `stable_sort` hat Zeitkomplexität $O(n(\ln n)^2)$ und verändert Reihenfolge äquivalenter Elemente nicht.

Algorithmen für sortierte Bereiche

- `lower_bound upper_bound binary_search includes`
- `merge set_union set_intersection set_difference set_symmetric_difference`
- Oft Zeitkomplexität $O(\ln n)$ anstelle von $O(n)$
- Undefiniertes Verhalten, falls Iteratorbereich *nicht* sortiert

Mengen

Ausprägungen

set assoziative Datenstruktur vergleichbar zu `map`

bitset Bitvektor zur Teilmengenbeschreibung von $\{0, \ldots, n-1\}$

Datentyp set

- Speicherung als balancierter Binärbaum wie bei `map`, daher schneller Suchzugriff auf Elemente ($O(\ln n)$)
- Kein Indexoperator `[]`, Suchen mit `find`
- Elemente nicht modifizierbar, ggf. löschen und neu einfügen
- bidirektionale Iteratoren wie bei `map`, `*pos` liefert Element, nicht Index/Wertepaar wie bei `map`
- Mengenoperationen (Durchschnitt etc.) auch für andere Behältertypen in STL (`<algorithm>`) vorhanden
- Einsatz des Mengentyps z.B. bei der Überprüfung, ob Elemente bereits zuvor betrachtet wurden

Bitsets

Vereinbarung, Eigenschaften

- `bitset<n> a` vereinbart Bitvektor *a* aus *n* Bits (*n* fest!)
- Stellt Bitoperationen entsprechend denen für ganze Zahlen bereit
- Indexoperator zum Lesen oder Setzen einzelner Bits
- Überladene Ein/Ausgabeoperatoren
- Umwandlungen in `unsigned long` und umgekehrt (falls Bitzahlen gleich)
- Umwandlungen in `string` (höchstes Bit $\hat{=}$ Komp. 0 des Strings)

Beispiele

1. Bitmuster von ganzen und Gleitpunktzahlen (Konversion per Zeiger/Referenz)
2. Sieb des Eratosthenes, Zweiquadratesatz von Fermat

Stromklassen als STL-Templates I

Allgemeines

- Stromklassen für Standard-, Datei- und Stringausgabe sind Templates mit einem Typparameter (`char`, `wchar_t`), teilweise abgeleitet von `basic_ostream<charT>`

`basic_ostream<char>`	`ostream`
`basic_ofstream<char>`	`ofstream`
`basic_ostringstream<char>`	`ostringstream`
`basic_ostream<wchar_t>`	`wostream`
`basic_ofstream<wchar_t>`	`wofstream`
`basic_ostringstream<wchar_t>`	`wostringstream`

- Analog für Eingabestromklassen (`basic_istream`) und für Ein/Ausgabestromklassen (`basic_stream`)
- Vordefinierte Stromobjekte:

`cin`	`istream`
`cout, cerr`	`ostream`
`wcin`	`wistream`
`wcout, wcerr`	`wostream`

Stromklassen als STL-Templates II

Allgemeines - Fortsetzung

- Nicht vordefinierte Stromobjekte durch Variablenvereinb. mit geeigneten Konstruktorargumenten, z.B.
  `ifstream ein("datei.bin",ios::binary)`
  oder ohne gefolgt von Komp.fkt. (z.B. `open`)
- Klasse `basic_ostream<charT>` ist abgeleitet von `basic_ios<charT>` (und diese von `ios_base`)
  Beinhaltet Strom- und Formatzustand (inkl. Flags)

`basic_ios<char>`	`ios`
`basic_ios<wchar_t>`	`wios`

- Strom- und Formatflags sind geerbt von `ios_base`, daher statt `ios::` bzw. `wios::` auch `ios_base::` möglich.
- „Lokalisierung“ von Stromklassen über Locale-Klassen im C++-Standard vorgesehen (Zeichensatzeigenschaften, Zahlenschreibweise, Datumsschreibweise etc.) – *nur kurz an Hand von Beispielen behandelt (→ Inf.bl.25)*

Stromklassen als STL-Templates III

Interne und externe Darstellung von Zeichensätzen

- Programmintern: Verwendung von Zeichensätzen fester Länge, Unterscheidung zwischen *narrow chars* (`char`) und *wide chars* (`wchar_t`)
  Vorsicht: Größe von `wchar_t` betriebssystemabhängig
- Externe Darstellung:
  Bei *narrow chars* interne = externe Darstellung
  Bei *wide chars* interne $\neq$ externe Darstellung möglich, ebenso Einsatz von Multibyte-Zeichen, d.h ein Zeichen vom Datentyp `wchar_t` wird extern (z.B bei Ein/Ausgabe) durch ein bzw. mehrere Bytes dargestellt.
- Linux unterstützt UCS-4 (intern) und UTF-8 (extern).

Unicode (UCS-4) und UTF-8

- Unicode: maximal 2^{31} Zeichen
  Zeichen $\rightarrow$ Zahl $\in \{0, 1, \ldots 2^{31} - 1\}$
  ASCII $\hat{=}$ 0 . . . 127, ISO-Latin-1 $\hat{=}$ 128 . . . 255
- Interne Darstellung: Typ `wchar_t` (Linux: `int`)
- Externe Darstellung: Bytefolge variabler Länge (Multibytes)
  Verwendung u.a. in Textdateien und Dateinamen
  Häufig für Multibytes: Zeichencodierung UTF-8

Unicode	*UTF-8*
0–7F	0bbbbbbb
80–7FF	110bbbbb 10bbbbbb
800–FFFF	1110bbbb 10bbbbbb 10bbbbbb
10000–1FFFFF	11110bbb 10bbbbbb 10bbbbbb 10bbbbbb
:	:

Beispiel: ä: $228 = (E4)_{16} = (11100100)_2$ (Unicode)
$\rightarrow$ $(11000011\ 10100100)_2 = (C3A4)_{16}$ (UTF-8)

Stringklassen als STL-Templates

Literale für Datentyp wchar_t

- Schreibweise in einfachen Apostrophen wie bei Literalen für `char`, jedoch mit vorangestelltem `L`, Ersatzdarstellungen `L'\u`*xxxx*`'` bzw. `L'\U`*xxxxxxxx*`'` möglich (*x* Hexadezimalziffer)

  Bsp.: `L'A'  L'\u0041'  L'ä'  L'\u00e4'`

C-Zeichenkettenliterale für Datentyp wchar_t

- Schreibweise in doppelten Apostrophen wie bei Literalen für `char`, jedoch mit vorangestelltem `L`, Ersatzdarstellungen wie bei Zeichenliteralen für `wchar_t`

  Bsp.: `L"ABC"  L"äöü"` usw.

C++-Strings für Datentyp wchar_t

- Template-Stringklasse `basic_string` mit Zeichentyp parametrisiert

```
basic_string<char>     string
basic_string<wchar_t>  wstring
```