Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 15: Templates (Parametrisierte Klassen und Funktionen)

- Parametrisierte Klassen
- Parametrisierte Funktionen
- Spezialisierung von Funktionen

Inhalt

Motivation (1)

- Es ist ein allgemeines Prinzip guten Programmierstils, daß man jede Entscheidung, jeden Algorithmus, jede Idee nur an einer Stelle im Programm (einmal) aufschreiben sollte.
  - ◇ deswegen z.B. Konstanten für Arraygrößen etc.
- Programmieren durch Kopieren und Modifizieren (Copy&Paste Programmierung) ist meist schlechter Stil: Mehrere, fast identische Programmteile.
  - ◇ deswegen z.B. Prozeduren/Funktionen

Motivation (2)

- Warum ist Copy&Paste Programmierung schlecht?
  - ◇ Mehr Tippaufwand,
  - ◇ mehr zu lesen und zu verstehen für neue Teammitglieder,

    Man muß jetzt ja auch erst einmal verstehen, daß es bis auf die kleinen Änderungen der gleiche Programmcode ist.
  - ◇ Denkstrukturen bei der Programm-Entwicklung entsprechen nicht Struktur des Programmtexts,
  - ◇ Änderungen werden aufwendig und führen leicht zu Inkonsistenzen (manche Kopien werden bei der Änderung vergessen).

Motivation (3)

- Prozeduren (Funktionen) bieten die Möglichkeit, ähnliche Folgen von Statements durch eine einmal definierte Prozedur mit Parametern zu ersetzen.

  Man abstrahiert also von den kleinen Unterschieden in diesen Folgen von Statements und erkennt, daß es — davon abgesehen — eigentlich die gleiche Statementfolge ist.

- Auf höherer Ebene, bei den Klassen, stellt sich das gleiche Problem:
  - ◇ Z.B. ist eine Klasse für Listen von ganzen Zahlen (`int`-Werten) sehr ähnlich zu einer Klasse für Listen von `Date`-Objekten.

Motivation (4)

- Man könnte natürlich in der Definition der Listen-Klasse einen `union`-Typ für die Elemente benutzen (z.B. `int` und `Date`).
- Jetzt ist aber der Programmierer dafür verantwortlich, daß er z.B. nicht in einem Listenelement einen `int`-Wert speichert, und den Wert in diesem Listenelement später als `Date` behandelt.

  Es werden auch gemischte Listen aus `int` und `Date` Elementen möglich, die vermutlich nicht beabsichtigt sind.

- Der Compiler hilft so nicht bei der Typprüfung!

Motivation (5)

- Weitere Varianten problematischer "Lösungen" (schalten Typprüfung durch den Compiler aus):
  - ◇ Untypisierter Pointer-Typ `void*` (falls die in der Liste zu speichernden Dinge alle Pointer sind).
  - ◇ Gemeinsame Oberklasse für alle möglichen Elementtypen (falls alle zu speichernden Dinge Objekte sind), in den Listenelementen dann Zeiger auf diese Oberklasse.

    Dies entspricht im wesentlichen dem `union`-Ansatz. Bei polymorphen Klassen (mit virtuellen Funktionen) kann man wenigstens zur Laufzeit eine Typprüfung mit dem `dynamic_cast` durchführen.

Motivation (6)

- Übrigens verwendet die Programmiersprache Java die Lösung mit einer höchsten Oberklasse "`Object`".
  - ◇ Generische Listen haben Elemente dieses Typs.
  - ◇ Java ist in dieser Hinsicht nicht typsicher.
- Allgemein gilt: Fehler, die schon durch den Compiler gefunden werden können, werden
  - ◇ sicher gefunden (nicht zu übersehen),
  - ◇ leichter lokalisiert,
  - ◇ und sind damit meist leichter zu beseitigen

  als Fehler, die nur durch Testen zu finden sind.

Templates (1)

- Templates erlauben (u.a.) die Definition von Datentypen/Klassen mit Parametern.
- Man kann damit einen Typ (eine Klasse) `list<T>` definieren, für einen beliebigen Elementtyp `T`.
- `T` ist hier also ein Typ-Parameter.
- Man bekommt aus einem Template (engl. für Schablone) durch Instanziierung eine konkrete Klasse für einen konkreten Elementtyp, z.B. `list<int>`.

Templates (2)

- Man kann das gleiche Template mehrfach mit verschiedenen Typen für den Parameter instanziieren:
  - ◇ Z.B. `list<int>` und `list<Date>`.
- Damit hat man erreicht, daß
  - ◇ man den allgemeinen Code für Listen nur einmal aufgeschrieben hat (im Template),
  - ◇ aber trotzdem die volle Typsicherheit gegeben ist: Aus Sicht des Compilers sind `list<int>` und `list<Date>` ganz unterschiedliche Typen.

    Es gibt dazwischen keine automatische Typumwandlung. Es gibt keinen gemeinsamen Obertyp (das Template selbst ist kein Typ).

Templates (3)

- Eine typische Implementierung von Templates ist, daß der Compiler für jede Instanziierung eines Templates den Programmcode intern dupliziert und dabei die konkreten Werte für die Parameter einsetzt.

  Man muß dazu natürlich intern Klassen- und Funktionsnamen (Bezeichner) erlauben, die "<" und ">" enthalten (oder die durch Instanziierung entstehenden Namen irgendwie anders codieren). Natürlich wird der Compiler bei der Definition eines Templates möglichst viel prüfen (Syntaxfehler etc.), und nicht die ganze Prüfung auf die Instanziierung verschieben (die Fehlermeldungen wären dann schwerer zu verstehen). Er wird sich dann auch eine schon vorverarbeitete interne Darstellung des Programmtextes im Template merken, und nicht einfach die vom Programmierer eingegebene Zeichenfolge.

Templates (4)

- Allgemein geht es nicht wesentlich besser, als für jede Instanziierung intern eigenen Programmcode zu erzeugen, weil die Speichergrößen der eingesetzten Datentypen unterschiedlich sein könnten:
  - ◇ Dann wären auch die Offsets zum Zugriff auf die Komponenten (Attribute) eines Objektes in den verschiedenen Instanziierungen eines Templates unterschiedlich.
- All das spielt sich aber intern ab: Der Programmierer schreibt das Template nur ein Mal auf.

Templates (5)

- Mit Templates bekommt man die Möglichkeit, eigene Typkonstruktoren zu definieren:
  - ◇ Z.B. ist der Typkonstruktor "Array der Größe `N` mit Elementtyp `T`" in die Sprache eingebaut.
  - ◇ Neben den Arrays gibt es aber noch viele weitere "Containertypen": Listen, Stacks, Queues, Mengen, Multimengen, Dictionaries, . . .
- Templates können nicht nur Typ-Parameter haben (für den Elementtyp `T`), sondern auch konstante Werte, insbesondere ganze Zahlen (für die Größe `N`).

Templates (6)

- Templates erlauben generische Programmierung in C++, also die Definition von Funktionen, die auf Werten unterschiedlicher Typen arbeiten.
- Diese so allgemeinen Funktionsdefinitionen erlauben dann eine bessere Wiederverwendbarkeit des Programmcodes.
- Die Standard-C++ Bibliothek macht viel Gebrauch von Templates.

  Früher gab es die STL ("Standard Template Library") von HP und später SGI. Sie ist größtenteils in der Standard C++ Bibliothek aufgegangen.

Templates (7)

- "Independent concepts should be independently represented and should be combined only when needed. Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies. Either way, you get a less flexible set of components out of which to compose systems."

  [Stroustrup, The C++ Programming Language, 2000, Seite 327.]

Inhalt

Beispiel: Listen (1)

- Definition des Templates für einfache Listenknoten:

```
template<class T> class List {
    private:
        T elem;
        List* next;
    public:
        List(T e) { elem = e; next = 0; }
        void set_next(List* l) { next = l; }
        T get_elem() { return elem; }
        List* get_next() { return next; }
};
```


Beispiel: Listen (2)

- Syntaktisch besteht ein Klassentemplate aus einer gewöhnlichen Klassendeklaration, der das Schlüsselwort “template” und dann in spitzen Klammern die Parameter vorangestellt sind.

  Als spitze Klammern werden das “größer”- und das “kleiner”-Zeichen verwendet.

- Typparameter werden in der Parameterliste durch das Schlüsselwort “class” gekennzeichnet.

  Man kann dafür aber später beliebige Typen einsetzen, nicht nur Klassen. Insofern passt das Schlüsselwort nicht ganz. Inzwischen kann man alternativ das Schlüsselwort “typename” verwenden (kein semantischer Unterschied, eventuell Probleme bei älteren Compilern).

Beispiel: Listen (3)

- In der Klassendeklaration kann der Typparameter T wie ein normaler Typ verwendet werden.
- Der Gültigkeitsbereich des Bezeichners T erstreckt sich bis zum Ende der Deklaration.
- Falls man in der Klassendeklaration den Rumpf für eine Funktion nicht angegeben hat, muß man die Definition über ein weiteres Template nachholen:

```
template<class T>
    void List<T>::set_next(List* l)
        { next = l; }
```


Beispiel: Listen (4)

- Im Gültigkeitsbereich List<T> ist die explizite Angabe des Parameters “<T>” überflüssig (aber erlaubt).
- Man könnte also überall, wo im obigen Beispiel nur “List” steht, auch “List<T>” schreiben.

  Einzige Ausnahme ist die Definition der Template-Klasse selbst. In template<class T> class List{ darf man nur “List” schreiben. Dort definiert man ja gerade das Template. Bei allen Benutzungen (auch innerhalb der Klassendefinition) darf man List<T> schreiben.

- In der Festlegung des Namensraums (links vom ::) bei der nachträglichen Definition einer Funktion ist der Parameter nötig (man muß List<T> schreiben).

Template-Instanziierung (1)

- Man instanziiert das Template, indem man einen konkreten Typ für den Typ-Parameter einsetzt:

```
int main()
{   List<int>* tmp = new List<int>(3);
    List<int>* first = tmp;
    List<int>* last = tmp;
    tmp = new List<int>(5); // Element anhängen
    last->set_next(tmp);
    last = tmp;
    for(List<int>*p = first; p; p=p->get_next())
        cout << p->get_elem() << "\n";
    return 0;
}
```


Template-Instanziierung (2)

- Da es etwas mühsam ist, den Parameter immer explizit anzugeben, kann man mit `typedef` einen Namen für diese Instanz der Templateklasse einführen:

```
typedef List<int> intlist;
```

  `typedef` erzeugt in C++ keinen neuen Typ. Der Compiler behandelt `intlist` und `List<int>` anschließend als den gleichen Typ.

- Da man häufig Zeiger benötigt, wäre auch Folgendes sinnvoll (man verwende in einem Projekt/Team einheitliche Namenskonventionen):

```
typedef List<int> intlist_c;
typedef intlist_c *intlist_t;
```


Template-Instanziierung (3)

- In der Template-Deklaration können für Werte des Typ-Parameters zunächst beliebige Operatoren, Methoden, Funktionen benutzt werden.
- Bei der Instanziierung kann es dann zu einem Fehler kommen, wenn ein Typ eingesetzt wird, der diese Funktionen etc. nicht hat.
- Es ist eine Schwäche von C++, daß man Anforderungen an Template-Parameter in der Template-Deklaration nicht explizit angeben kann.

Template-Instanziierung (4)

- Man kann sich den Template-Mechanismus also wie eine Art Macro vorstellen:
  - ◇ Bei der Template-Definition wird im wesentlichen nur der Text abgespeichert.
  - ◇ Bei der Instanziierung werden die Parameter ausgefüllt und erst dann findet die eigentliche Compilierung und Prüfung statt.
- Der Compiler achtet aber darauf, daß mindestens innerhalb eines Übersetzerlaufes ein Template für einen Parameterwert nur einmal instanziiert wird.

Template-Instanziierung (5)

- Ein guter Compiler wird bei der Instanziierung eines Templates nur die tatsächlich benötigten Methoden erzeugen.
- Noch eine syntaktische Feinheit:
  - ◇ Für einen Template-Parameter kann man auch einen Typ einsetzen, der selbst durch Instanziierung eines Templates entsteht.
  - ◇ Falls man dabei am Ende die Zeichenfolge "> >" erhält, ist das Leerzeichen nötig (>> wäre ja der Shift-Operator).

Inhalt

Beispiel: Stack (1)

- Ein Stack ist eine Datenstruktur, die nach dem LIFO-Prinzip arbeitet: "Last in, First out".
- Z.B. wird ein Stack intern für die Prozeduraufrufe verwendet: Die zuletzt aufgerufene Prozedur wird zuerst beendet. Hier hat man also einen Stack von Rücksprungadressen.

  Tatsächlich ist es komplizierter, weil dort auch die Parameter und lokalen Variablen abgelegt werden.
- Stacks werden auch zur Syntaxanalyse bei Klammerstrukturen verwendet.

Beispiel: Stack (2)

- Ein Stack hat die zentralen Operationen:
  - ◇ `push(x)`: Einen Wert auf den Stack legen.
  - ◇ `pop()`: Den obersten Wert vom Stack nehmen.
  - ◇ `empty()`: Test, ob der Stack leer ist.
- Man kann bei einem Stapel immer nur an das oberste Element herankommen.

  Deutsch heißen Stacks "Kellerspeicher": Bei einem sehr vollen und unaufgeräumten Keller muß man sich von vorne nach hinten vorarbeiten.

Beispiel: Stack (3)

- Der Datentyp der Elemente ist wieder egal, also ein Template-Parameter.
- Man kann Stacks mit verketteten Listen implementieren, bei denen man nur vorne einfügt bzw. herausnimmt.
- Hier sollen Stacks aber mit einem Array implementiert werden.
- Dann braucht man noch einen zweiten Parameter für die Maximalgröße.

Beispiel: Stack (4)

- Definition des Templates für Stacks:

```
template<class T, int Max> class Stack {
    private:
        T elems[Max];
        int num_elems;
    public:
        Stack() { num_elems = 0; }
        void push(T elem)
            { if(num_elems < Max)
                elems[num_elems++] = elem; }
        T pop();
        bool empty() { return num_elems == 0; }
};
```


Beispiel: Stack (5)

- Definition der Methode `pop()`:

```
template<class T, int Max> T Stack<T,Max>::pop()
{
    if(num_elems > 0) // Stack ist nicht leer
        return elems[--num_elems];
    else
        return 0;
};
```

- Diese Lösung hat den Nachteil, daß sie nur für Parametertypen `T` funktioniert, in die sich die Zahl `0` umwandeln läßt (numerische Typen und Zeiger).

  Versucht man z.B. die Instanziierung `Stack<Date,10>`, bekommt man einen Fehler (erst bei der Instanziierung, `Stack<int,10>` wäre ja ok).

Beispiel: Stack (6)

- Es wäre ohnehin besser, den falschen Aufruf nicht durchgehen zu lassen:

```
template<class T, int Max> T Stack<T,Max>::pop()
{
    if(num_elems > 0) // Stack ist nicht leer
        return elems[--num_elems];
    else {
        cerr << "Fehler: pop bei leerem Stack\n";
        exit(1);
        return elems[0];
    }
};
```

(Erläuterungen siehe nächste Seite . . .)

Beispiel: Stack (7)

- `cerr` ist die Standard-Fehler-Ausgabe.

  Normalerweise landet sie wie `cout` auf dem Bildschirm. Aber wenn der Benutzer die Standard-Ausgabe auf eine Datei umgelenkt hat, wird die Fehlermeldung dennoch auf dem Bildschirm angezeigt (falls er/sie `cerr` nicht auch umgelenkt hat). `cerr` ist auch ungepuffert.

- Ein Aufruf von `exit(`n`)` beendet das Programm.

  Für den Parameter n wählt man den Wert 0, wenn alles in Ordnung ist, und kleine positive Zahlen für unterschiedliche Fehler.

- Die `return`-Anweisung wird nie ausgeführt, aber weil der Compiler das eventuell nicht weiß, muß man einen Wert vom richtigen Typ angeben.
- Noch besser: Exceptions, siehe Kapitel 16.

Template-Parameter

- Aktuelle Parameter von Templates können sein:
  - ◇ Datentypen (formaler Parameter mit `class`),
  - ◇ Konstante Ausdrücke (z.B. für formale Parameter vom Typ `int`),

    Der Compiler instanziiert das Template einmal für jeden verschiedenen Parameter-Wert. Er muß den Parameter-Wert daher schon zur Compilezeit kennen. Nur so kann er auch die Typ-Gleichheit prüfen: Verschiedene Parameter-Werte geben verschiedene Typen. Konstante Strings sind als Template-Argumente nicht möglich. Auch die Typen `float` und `double` sind ausgeschlossen.

  - ◇ Adressen von Objekten / Funktionen (global).

    String-Konstanten werden abgelehnt, weil `static`.
    Bei Bedarf `char s[] = "..."` (global). Dann geht `s`.

Templates und Default-Werte

- Für Parameter von Templates kann man wie bei Funktions-Parametern Default-Werte angeben.
- Wenn der Stack z.B. normalerweise ein Array von 100 Elementen verwendet, aber man dem Benutzer des Templates die Möglichkeit lassen will, eine andere Größe zu wählen, schreibt man:

```
template<class T, int Max = 100> class Stack {
```

- Nun kann man z.B. `Stack<int>` schreiben.
- Default-Werte kann man auch für Typ-Parameter eines Templates angeben (z.B. `class T = int`).

Templates und Vererbung (1)

- Für Template-Klassen kann man auch Vererbung verwenden, d.h. eine Template-Klasse kann sowohl von einer normalen Klasse erben, als auch von einer Template-Klasse.

  Das zusätzliche Problem ist hier, daß der Compiler nun ggf. implizit auch Oberklassen instanziieren muß. Wenn man in der Unterklasse ein von der Oberklasse geerbtes Attribut ansprechen will, muß man eventuell `this->` davor schreiben (und/oder mit `::` explizit die Klasse angeben).

- Z.B. könnte man die Verwaltung des Stackpointers `num_elems` in eine Klasse `BasicStack` (gewöhnliche Klasse ohne Parameter) auslagern.

Templates und Vererbung (2)

- Eine Abtrennung der vom Parameter unabhängigen Teile könnte sich lohnen, wenn
  - ◇ man ein Template mehrfach für verschiedene Parameterwerte instanziiert, und
  - ◇ und der vom Parameter unabhängige Programmcode nicht ganz einfach/kurz ist.

  Beim Stack gilt das nicht, aber es ist hier nur ein Syntax-Beispiel.
- Dann könnte man schreiben:

```
template<class T, int Max>
        class Stack: public BasicStack {
```


Templates und Vererbung (3)

- Bsp.: Man will eine Templateklasse ExtStack<T,Max> definieren, in der Stack<T,Max> um zusätzliche Operationen erweitert ist (z.B. int size()).

  D.h. ExtStack<T,Max> ist eine Unterklasse von Stack<T,Max>.
- Das geht so:

```
template<class T, int Max>
        class ExtStack: public Stack<T,Max> {
```

- Auch ExtStack hat die Parameter T und Max, aber die sind bei der Template-Definition vorangestellt.

  Bei der Oberklasse muß man die Parameter dagegen explizit angeben (sonst Oberklasse ohne Parameter wie BasicStack).

Templates und Vererbung (4)

- Zwischen verschiedenen Instanziierungen des gleichen Templates besteht keine Subtyp-Beziehung.
- Beispiel:
  - ◇ Student ist eine Unterklasse von Person.
  - ◇ Dennoch ist List<Student> nicht eine Unterklasse von List<Person>.
- Wenn das auch zunächst vernünftig aussieht, geht es technisch nicht: Student-Objekte sind größer als Person-Objekte, der next-Zeiger der Listen-Objekte steht also an unterschiedlicher Position.

Parameter bündeln (1)

- Man kann auch mehrere Parameter bündeln, und in einer Klasse verpacken, die man extra zu diesem Zweck definiert.
- Wenn die Template-Klasse C z.B. von einem Typ T und Funktionen f, g und h (jeweils von T nach T) abhängig ist, kann man diese Funktionen in einer Klasse X definieren, z.B. (für T = int)

```
class Xint {
    static int f(int n) { ...}
    static int g(int n) { ...}
    static int h(int n) { ...}
}
```

Parameter bündeln (2)

- Dann wählt man als Parameter von C nur T und X, und kann im Template z.B. X::f(...) benutzen.
- Zum Beispiel kann man jetzt C<int,Xint> schreiben.

  Mit der direkten Lösung hätte das Template vier Parameter, und man hätte C<int,f,g,h> schreiben müssen.
- Wenn man will, kann man auch den Typ T innerhalb der Parameter-Klasse X mit typedef definieren (und ihn dann in C mit typename X::T ansprechen).

  typename ist nötig, weil der Compiler bei der Syntaxanalyse des Templates nicht weiß, daß bei der später für X eingesetzten Klasse T ein Typ ist (es gibt syntaktisch mehrdeutige Fälle).

Parameter bündeln (3)

- Natürlich hätte man auch sonst für C<T,f,g,h> mit typedef einen Typnamen einführen können, so daß der Tippaufwand bei Variablendeklarationen kein Argument ist. Aber:
  - ◇ Auf diese Weise macht man sehr deutlich, daß f, g und h (und ggf. T) zusammengehören.
  - ◇ Wenn man diese Parameter an weitere Templates übergeben muß, die in C benutzt werden, wird das Programm etwas kürzer und klarer.

Inhalt

Funktions-Templates (1)

- Templates sind nicht nur für Klassen möglich, sondern auch für Funktionen:

```
template<class T> T maximum(T n, T m)
    { if(n > m) return n; else return m; }
```

- Während man bei der Verwendung von Klassen-Templates Werte für die Parameter explizit angeben muß, bestimmt der Compiler bei Funktionstemplates wie max den Parameter T aus dem Aufruf.

  Soweit als möglich. Es kann auch sein, daß man Werte für Parameter explizit angeben muß, zumindest für einen Teil der Parameter.

Funktions-Templates (2)

- Z.B. ist bei folgendem Aufruf klar, daß `T` = `int` ist:

```
int i = ...; cout << maximum(i, 0);
```

Ziffernfolgen sind Literale (Datentyp-Konstanten) vom Typ `int`, sofern Sie in diesen Typ passen und nicht mit einem speziellen Suffix gekennzeichnet sind. Daher haben beide Argumente wirklich den Typ `int` (genauer ist `i` ein Lvalue, und hat formal den Typ `int&`, aber von einem Lvalue kann direkt zum Wert übergegangen werden).

- Für die automatische Bestimmung des Template-Parameters ist Voraussetzung, daß die Argumenttypen genau passen. Eine zusätzliche Typanpassung des Argumentes ist nicht möglich.

Außer triviale Anpassungen, z.B. `const int` → `int` und `int&` → `int`.

Funktions-Templates (3)

- Folgender Aufruf ist z.B. nicht möglich:

```
short int s = ...; cout << maximum(s, 0);
```

- Die beiden Argumente haben hier unterschiedlichen Typ (`short int` und `int`).
- Normalerweise wäre die Umwandlung von `short int` in `int` überhaupt kein Problem, wenn die Funktion z.B. mit `int`-Parametern deklariert wäre:

```
int maximum(int n, int m)
    { if(n > m) return n; else return m; }
```

- So würde obiger Funktionsaufruf funktionieren.

Funktions-Templates (4)

- Ein Template erzeugt eine unendliche Menge von überladenen Funktionsvarianten, auch `T` = `short int`:

```
short int maximum(short int n, short int m)
    { if(n > m) return n; else return m; }
```

- Da für die Parameter-Übergabe die gleichen Regeln wie für Zuweisungen gelten, ist auch die Umwandlung `int` → `short int` möglich, und die auszuwählende Funktionsvariante nicht eindeutig.

Da es um verschiedene Argumente geht, spielt keine Rolle, daß nach der Prioritätenliste für die Auswahl überladener Funktionen die werterhaltende Umwandlung `short int` → `int` vorzuziehen wäre.

Funktions-Templates (5)

- Eine Lösung ist, die Mehrdeutigkeit zu vermeiden, indem man den Typ des einen Arguments explizit in den des anderen Argumentes umwandelt, z.B.

```
maximum(static_cast<int>(s), 0);
```

- Eine alternative Lösung ist, den Typ-Parameter des Templates explizit anzugeben:

```
maximum<int>(s, 0);
```

- Nur bei Argumenten mit Typ-Parameter muß man auf die automatischen Typ-Anpassungen verzichten, bei anderen Argumenten geht es dagegen.

Funktions-Templates (6)

- Funktions-Templates können auch Parameter haben, die nicht aus den Argumenten des Funktionsaufrufs zu bestimmen sind.

  Wie Klassen-Templates, also nur Konstanten.

- Z.B.: Ergebnistyp wird nicht aus Kontext erschlossen (entspricht wieder überladenen Funktionen).
- Wenn man die unbedingt anzugebenen Parameter in der Parameterliste vorn anordnet, brauchen nur diese angegeben zu werden.

  Das funktioniert so, als wenn es einen impliziten Default-Wert für die aus den Argumenten zu erschließenden Parameter geben würde.

Inhalt

Exkurs: C-Präprozessor (1)

- Eines der Ziele bei der Entwicklung von C++ war, in C viel verwenden Präprozessor durch bessere (vor allem sicherere) Alternativen überflüssig zu machen (bis auf `#include`, `#if`, `#ifdef`).
- In C wurde der Präprozessor zur Ersetzung von Makros verwendet, bevor der eigentliche Compiler den Programmtext zu sehen bekam.

  Der C-Präprozessor ist ein vom Compiler getrenntes Programm, das auch für andere Sprachen außer C verwendet werden kann. Es versteht die C-Syntax nicht, sondern führt rein textuelle Ersetzungen durch. Das kann manchmal zu überraschenden Fehlern führen.

Exkurs: C-Präprozessor (2)

- Folgendes war ein klassisches Beispiel:

```
#define MAX(N,M) ((N)>(M)?(N):(M))
```

  Die Klammern sind zur Sicherheit nötig, damit es auch funktioniert, wenn für `N` und `M` Ausdrücke eingesetzt werden, die eine geringere Priorität als > haben.

- Anschließend ersetzt der Präprozessor z.B. `MAX(i,0)` im Programmtext durch

```
((i)>(0)?(i):(0))
```

- Der Präprozessor war sehr nützlich, aber man konnte damit auch ganz unleserliche Programme schreiben, oder schwierig zu findene Fehler verursachen.

Exkurs: C-Präprozessor (3)

- Beispiel für Probleme mit Makros: Da es nur eine textuelle Ersetzung ist, erhöht `MAX(i++,0)` die Variable `i` zweimal, wenn `i` größer als `0` ist.
- Ein erster Schritt zur Ablösung der Makros waren `inline`-Funktionen:

```
inline int MAX(int n, int m)
    { return (n>m) ? n : m; }
```

- Vorher war es ein Vorteil von Makros gegenüber klassischen Funktionen, daß der Overhead für den Funktionsaufruf vermieden wurde.

Exkurs: C-Präprozessor (4)

- Nun mußte man allerdings die Maximums-Funktion mehrfach definieren (für jeden benötigten Typ getrennt).
- Mit Funktions-Templates kann man nun (fast) alle Vorteile des `MAX`-Makros haben, ohne die potentiellen Probleme dafür in Kauf nehmen zu müssen.

  Das `MAX`-Makro hätte allerdings auch bei unterschiedlichen Typen für die Argumente funktioniert.

- Insbesondere ist dafür auch die automatische Bestimmung des Template-Parameters wichtig.

Inhalt

1. Einführung, Motivation, Anwendungen
2. Beispiel: Listen. Instanziierung von Templates
3. Beispiel: Stack. Fortgeschrittene Konzepte
4. Funktions-Templates
5. Exkurs: C-Präprozessor
6. Für Effizienz-Fans: Template-Spezialisierung

Template-Spezialisierung (1)

- Man kann für ein Template zusätzlich zur allgemeinen Definition auch Definitionen für spezielle Parameterwerte angeben (um Ausnahmen zu behandeln, oder häufige Spezialfälle effizienter zu implementieren).
- Wenn man z.B. für `List<int>` eine andere Implementierung angeben will, definiert man zunächst wie gehabt das allgemeine Template:

```
template<class T> class List { ...};
```

Template-Spezialisierung (2)

- Dann definiert man den Spezialfall:

```
template<> class List<int> { ... };
```

- Dieser Spezialfall hat selbst keine Parameter mehr, aber er implementiert `List<int>`.

  Nach dem Schlüsselwort `template` listet man die echten Parameter auf, für die bei der Instanziierung dieses Spezialfalls noch Werte gefunden werden müssen. Nach dem Namen des Templates gibt man die Parameterliste bezogen auf den allgemeinen Fall an.

- Der Compiler wählt für eine konkrete Instanziierung immer die speziellste Implementierung.

Template-Spezialisierung (3)

- Auch partielle Spezialisierungen sind möglich, z.B. passt Folgendes auf Zeigertypen:

```
template<class T> class List<T*> { ... };
```

- Wenn eine Variable z.B. mit

```
List<Student*> L;
```

  deklariert wird, so wird diese Implementierung verwendet, instanziiert für `T = Student`.

  Es sei denn, es gibt eine noch spezielle Template-Definition `template<> class List<Student*> { ... };`

Template-Spezialisierung (4)

- Man kann diesen Mechanismus z.B. verwenden, um eine gemeinsame Implementierung für `List<T>` für alle Zeigertypen `T` zu definieren.
- Ansonsten könnte, wenn man das gleiche Template mit vielen unterschiedlichen Typen instanziiert, das entstehende Objektprogramm sehr groß werden.
- Da Zeiger intern alle gleich groß sind, kann man z.B. eine Implementierung für `void*` definieren, und die dann mittels `private`-Vererbung im Template-Spezialfall für allgemeine Zeigertypen verwenden.