Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 5

5. Funktions- und Klassen-Templates

5.1. Generische Funktionen (Funktions-Templates)

5.2. Generische Klassen (Klassen-Templates)

Generische Funktionen (Funktions-Templates) in C++ (1)

• **Allgemeines** :

◇ Generische Funktionen sind **Funktions-Schablonen**, die ein Muster für eine ganze **Gruppe von Funktionen**, die den **prinzipiell gleichen Algorithmus** für jeweils **unterschiedliche Parametertypen** ausführen, definieren.

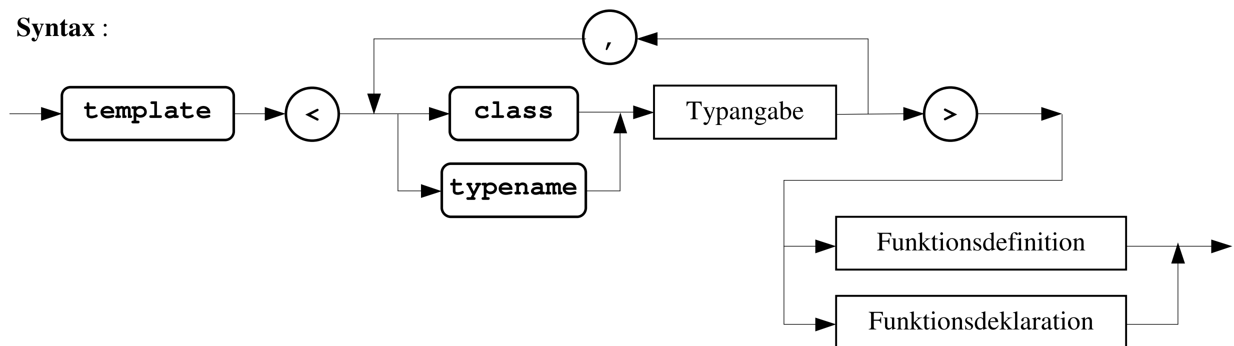
◇ Die Definition einer generischen Funktion (Funktions-Template, ***function template***) erzeugt noch keinen Code. Passender – den Typen der jeweiligen aktuellen Parameter entsprechender – **Code** wird **vom Compiler** erst **bei der erstmaligen Verwendung der generischen Funktion** mit diesen Parameter-Typen erzeugt (**Instantiierung** des Templates → **instantiierte Funktion**, **Template-Funktion**, ***template function***).

◇ Im Prinzip stellt eine generische Funktion eine Funktion dar, die sich **automatisch selbst überladen** kann. Im Unterschied zu allgemeinen überladenen Funktionen, bei denen jede Funktion prinzipiell einen anderen Algorithmus realisieren und eine unterschiedliche Anzahl von Parametern haben kann, haben alle Versionen einer generischen Funktion die gleiche Anzahl von Parametern und realisieren den gleichen Algorithmus - nur eben mit unterschiedlichen Parametertypen.

◇ In der Vereinbarung einer generischen Funktion werden die **Parametertypen**, die sich **ändern** können, als (Template-) **Parameter** festgelegt.

• **Vereinbarung einer generischen Funktion** :

◇ **Syntax** :



◇ Typangabe ist ein **formaler Typ-Parameter** (**Template-Parameter**). Jeder Typ-Parameter kann innerhalb der **Parameterliste der Funktionsvereinbarung** als Platzhalter für eine **aktuelle Typangabe** verwendet werden (Festlegung des Typs eines Funktionsarguments).

◇ Ein Template-Parameter kann auch verwendet werden
- ▻ zur **Festlegung** des **Rückgabetyps** der Funktion
- ▻ zur **Festlegung** des Typs **lokaler Variabler** der Funktion

◇ **Beispiele** :

```
template <class T> void swap(T& a, T&b)
{
  T h=a;
  a=b;
  b=h;
}

template <typename TYPE1, typename TYPE2>
void myfunc(TYPE1 x, TYPE2 y)
{
  cout << x << "   " << y << '\n';
}

template <class T1, class T2>
T2 convert(T1 w)
{
  return (T2)w;
}
```

Generische Funktionen (Funktions-Templates) in C++ (2)

• Aufruf von Template-Funktionen

◇ Eine Template-Funktion wird **bei** ihrem **erstmaligen Aufruf generiert**. (→ **Implizite Instantiierung**)
Für die **Angabe des Funktionsnamens** existieren hierfür zwei Möglichkeiten :
▻ Verwendung nur des Template-Namens als Funktionsnamen.
▻ Ergänzung des Template-Namens um die Angabe aktueller Template-Parameter

◇ Bei **Aufruf** einer Template-Funktion **allein** mit dem **Template-Namen** müssen die zu verwendenden aktuellen Template-Parameter aus den aktuellen Funktions-Parametern (Funktions-Argumenten) ermittelbar sein.
Das bedeutet,
- sämtliche Template-Parameter müssen zur Festlegung des Typs von Funktions-Parametern dienen
- die Typen der aktuellen Funktions-Parameter müssen – ohne Anwendung impliziter Typkonvertierungen – eine **eindeutige Template-Instantiierung** ermöglichen.

◇ Bei **Aufruf** einer Template-Funktion mit dem um eine **aktuelle Parameterliste ergänzten Template-Namen** legt die Parameterliste die aktuellen Template-Parameter fest.
In diesem Fall können gegebenenfalls implizite Typkonvertierungen der aktuellen Funktions-Parameter in die aktuellen Template-Parameter stattfinden.
Dienen Template-Parameter allein zur Festlegung des Rückgabetyps der Funktion und/oder allein zur Festlegung des Typs von lokalen Funktions-Variablen muß diese Form des Aufrufs verwendet werden.

• Templates bei mehreren Programm-Modulen

◇ Bei der Übersetzung eines Template-Funktions-Aufrufs (d.h. bei der Erzeugung einer Template-Funktion) muss dem Compiler der Code des Funktions-Templates vorliegen.

◇ Soll dasselbe Funktions-Template in mehreren Modulen verwendet werden, ist es zweckmässig den **Template-Code** (also die Template-Definition) in eine **Headerdatei** aufzunehmen, die dann von den verwendenden Modulen eingebunden werden muss.

◇ Eine Template-Funktion, die nicht als `inline` definiert worden ist, wird wie eine normale globale Funktion behandelt. Sie lässt sich daher aus verschiedenen Modulen aufrufen. Andererseits darf sie in einem Programm nur einmal definiert sein.
Der o.a. Mechanismus bewirkt aber zunächst, dass eine Template-Funktion in jedem Modul, in dem sie aufgerufen wird, auch erzeugt (d.h. definiert) wird.
Der Linker ist dafür verantwortlich, dass – bis auf eine – alle weiteren Instanzen einer derartigen Funktion wieder entfernt werden.

• Überladen generischer Funktionen :

◇ Ein Funktions-Template läßt sich auch **explizit überladen**.

◇ Das Überladen kann erfolgen mit
- einem **weiteren Funktions-Template**, das eine **unterschiedliche Funktions-Parameter-Liste** besitzt.
- einer **Nicht-Template-Funktion**, die eine von dem Funktions-Template **abweichende Parameter-Liste** besitzt.
- einer **Nicht-Template-Funktion**, deren **Parameter-Liste** mit der Parameter-Liste einer auch aus dem Template instantiierbaren Template-Funktion **übereinstimmt**.
  In diesem Fall "überschreibt" die explizit überladende Funktion die spezielle – ihren Parametertypen entsprechende – Template-Funktion (Instanz des Funktions-Templates).
  → Definition einer **expliziten Spezialisierung** des Funktions-Templates.

◇ ANSI-C++ sieht zur **Kennzeichnung** einer **expliziten Template-Spezialisierung** den Vorsatz

```
template<>
```

vor, der der Funktions-Vereinbarung vorangestellt werden kann.

Generische Funktionen (Funktions-Templates) in C++ (3)

- **Demonstrationsprogramm `genfunc1`**

```
// ------------------------------------------------------------------
// Programm genfunc1     (C++-Quelldatei genfunc1_m.cpp)
// Beispiel zu generischen Funktionen und zu zusätzlicher explizit
// überladender Funktion (explizite Spezialisierung)
// ------------------------------------------------------------------

#include <iostream>
using namespace std;


template <class TY> TY max(TY a, TY b)
{
  return a>b ? a : b;
}


template <typename TYPE1, typename TYPE2>
TYPE2 convert(TYPE1 w)
{
  return (TYPE2)w;
}


template<>                          // explizite Spezialisierung :
char *max(char *a, char *b)         // "überschreibt" generische Version
{                                   // von char *max(char *, char *)
  int i=0;
  while (a[i]==b[i] && a[i]!=0)
    i++;
  return a[i]>b[i] ? a : b;
}


int main(void)
{
  cout << endl << "int-max        : " << max<int>(7, -2);
  cout << endl << "double-max     : " << max(3.14, 27.9);
  cout << endl << "char-max       : " << max('a', 'z');
  cout << endl << "string-max     : " << max("Hausdach", "Haus");
  cout << endl << "long-max       : " << max(-256000L, 4L);
  cout << endl << "long-max       : " << max<long>(-256000L, 4);
  cout << endl << "convert (d->i) : " << convert<double, int>(45.14);
  cout << endl;
  return 0;
}
```

- **Ausgabe des Programms**

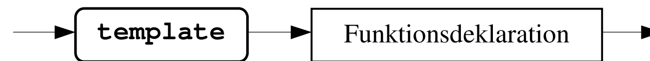
```
int-max        : 7
double-max     : 27.9
char-max       : z
string-max     : Hausdach
long-max       : 4
long-max       : 4
convert (d->i) : 45
```

Generische Funktionen (Funktions-Templates) in C++ (4)

• Explizite Instantiierung eines Funktions-Templates

◇ ANSI-C++ sieht auch eine explizite Instantiierung eines Funktions-Templates vor.
Hierbei wird der Code einer Template-Funktion erzeugt, ohne dass die Funktion aufgerufen wird.

◇ **Syntax** :



Beispiel :

```
template <class TY>            // Template-Definition
TY max(TY a, TY b) { return a>b ? a : b;}
template int max(int, int);    // Template-Instantiierung
```

◇ Eine explizite Template-Instantiierung sollte zweckmässigerweise in einer `cpp`-Datei und nicht in einer Headerdatei stehen. Diese `cpp`-Datei kann dann auch die Template-Definition enthalten.

◇ Eine explizit instantiierte Funktion kann **in** anderen **Übersetzungseinheiten verwendet** (aufgerufen) werden, **ohne** dass sie **erneut generiert** werden muss.
Für ihre Verwendung ist lediglich eine – häufig in eine Headerdatei gestellte – **Extern-Deklaration** erforderlich :
- entweder der jeweiligen Template-Funktion
- bzw. (als gemeinsame Deklaration für alle Template-Funktionen) des Funktions-Templates

• Beispiel zur expliziten Instantiierung eines Funktions-Templates

```
// C++-Headerdatei templmax2.h
// Nur Deklaration des Funktions-Templates max<>

template <class TY> TY max(TY a, TY b);
```

```
//C++-Quelldatei templmax.cpp
//Explizite Instantiierung eines Funktions-Templates

#include "templmax2.h"     // Einbinden der Headerdatei hier nicht erforderlich

template <class TY> TY max(TY a, TY b)
{ return a>b ? a : b;}

template int max(int, int);
template char max(char, char);
template double max(double, double);
template long max(long, long);
```

```
// Programm genfunc2     (C++-Quelldatei genfunc2_m.cpp)
// Beispiel zur Verwendung der expliziten Instantiierung eines Funktions-Templates

#include <iostream>
using namespace std;

#include "templmax2.h"

int main(void)
{ cout << endl << "int-max    : " << max(7, -2);
  cout << endl << "double-max : " << max(3.14, 27.9);
  cout << "char-max   : " << max('a', 'z');
  // cout << endl << "string-max : " << max("Hausdach", "Haus"); // keine
  //                                                           // Instantiierung!
  cout << endl << "long-max   : " << max<long>(-256000L, 4);
  cout << endl;
  return 0;
}
```

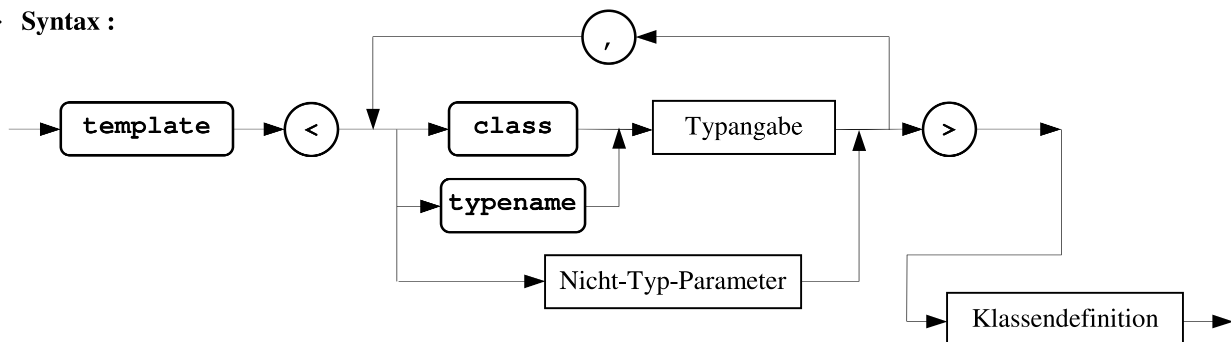
Generische Klassen (Klassen-Templates) in C++ (1)

• Allgemeines :

Generische Klassen (Klassen-Templates, *class templates*) sind **Schablonen**, die ein **Definitions-Muster** für eine ganze **Gruppe von Klassen**, die **prinzipiell gleich aufgebaut** sind und deren **Member-Funktionen** den jeweils **prinzipiell gleichen Algorithmus** realisieren, festlegen.
Die einzelnen nach diesem Muster konstruierbaren Klassen unterscheiden sich lediglich im **Typ einer oder mehrerer Komponenten** (und in den dadurch gegebenen Auswirkungen auf die Member-Funktionen).
Die **Typen**, in denen sich die **einzelnen Klassen unterscheiden**, werden bei der **Definition** einer generischen Klasse als **formale Parameter** festgelegt. (Generische Klassen werden daher auch als **parameterisierte Typen** bezeichnet).

• Definition einer generischen Klasse :

◇ **Syntax :**



◇ **Beispiel :**

```
template <class T> class Stack  // Generische Klasse "Stack für Typ T"
{                               // (Klassen-Template)
  public:
    Stack (int);           // Konstruktor
    void push(T);          // Ablage eines Elements auf Stack
    T pop(void);           // Rückholen eines Elements vom Stack
  private:
    int size;              // Groesse des Stacks
    T *stck;               // Pointer auf Speicherbereich des Stacks
    int tos;               // Index des Top des Stacks (Stackpointer)
};
```

• Template-Klassen :

◇ Die **Definition einer generischen Klasse** generiert noch **keine konkrete Klasse** (und erzeugt noch keinen Code für Member-Funktionen). Dies erfolgt erst bei der **erstmaligen Verwendung** ihres Namens (des **Klassen-Template-Namens**) zusammen **mit aktuellen Parametern**
→ **implizite** (z.B. in einer Objekt-Vereinbarung) oder **explizite Instantiierung** des Klassen-Templates.
Jeder **unterschiedliche** Satz **aktueller Parameter** definiert eine **neue Klasse**
(→ instantiierte Klasse, **Template-Klasse**, *template class*).

◇ Der **Klassen-Template-Name** gefolgt von **in spitzen Klammern eingeschlossenen aktuellen Parametern** (Parameter-Zusatz) stellt den **Namen einer konkreten Klasse** dar und kann genauso wie jeder andere Klassenname verwendet werden (→ **Template-Klassen-Name**).

◇ **Syntax :** **`klassen-template-name<akt_par_liste>`**

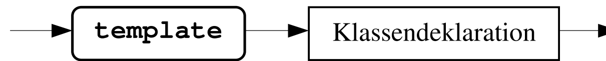
◇ **Beispiele :**

```
// implizite Template-Instantiierungen
Stack<int>    is(20);       // Stack fuer int-Werte
Stack<char*>  cps(30);      // Stack fuer char-Pointer
Stack<double> ds(50);       // Stack fuer double-Werte
```

Generische Klassen (Klassen-Templates) in C++ (2)

- **Explizite Instantiierung eines Klassen-Templates**

  ◇ **Syntax** :



  Beispiel : `template class Stack<int>;` // Stack fuer int-Werte

- **Verwendung des Klassen-Template-Namens**

  ◇ **Außerhalb** der **Definition der generischen Klasse** darf der **Klassen-Template-Name nicht allein** sondern nur **zusammen mit** einem **Parameter-Zusatz** verwendet werden (gegebenenfalls mit den formalen Parametern, s. Definition der Member-Funktionen generischer Klassen).

  ◇ **Innerhalb** der Klassen-Definition **darf** - muß aber nicht - bei **Eindeutigkeit** der **Parameter-Zusatz** mit den formalen Parametern **auch weggelassen** werden.
  Lediglich bei **Konstruktor**- und **Destruktor**-Namen **darf** der Parameter-Zusatz **nicht** angegeben werden.

  ◇ **Beispiel :**

```
template <class T> class List      // Generische Klasse : "Element für
{                                  // lineare Liste von Werten des Typs T"
  public :
    List(T);                       // Konstruktor
    ~List(void);                   // Destruktor
    void add(List<T> *node);       // oder :  ...(List *node);
    List<T> *getNext(void) const;  // oder :  List *get...
    T getData(void)const;
  private :
    T data;
    List<T> *next;                 // oder :  List *next;
};
```

- **Explizite Spezialisierung einer generischen Klasse**

  ◇ Eine **generische Klasse** kann durch die **explizite Definition** einer **konkreten Klasse** für einen speziellen aktuellen Parametersatz "**überladen**" werden. Kennzeichnung nach ANSI-C++ : Vorsatz `template<>`
  Sofern die explizite Definition vor Anwendung der konkreten Klasse erfolgt, hat sie **Vorrang** vor einer Instantiierung der generischen Klasse für den speziellen Parametersatz.

  ◇ Innerhalb einer derartigen Klassendefinition müssen **Konstruktornamen** - **nicht** jedoch **Destruktornamen** - um einen **Parameter-Zusatz** (mit dem speziellen aktuellen Parametersatz) **ergänzt** werden.

  ◇ Für eine explizit definierte konkrete Klasse müssen auch **alle Member-Funktionen** – auch wenn sie im Namen und Aufbau denen der generischen Klasse entsprechen – **gesondert definiert** werden.

  ◇ **Beispiel :**

```
template <class T> class List    // Definition generische Klasse
{ /* ... */ };

template<>                       // Kennzeichnung einer
class List<char*>                // explizite Spezialisierung
{ public:                        // "Überladen" der generischen Klasse
    List<char *>(char *);
    ~List();
    void add(List<char*> *node);
    List<char*> *getNext(void) const;
    char *getData(void) const;
  private:
    char *data;
    List<char*> *next;
};
```

Generische Klassen (Klassen-Templates) in C++ (3)

• Member-Funktionen generischer Klassen

◇ Für **jede** aus einer generischen Klasse erzeugten **konkreten Klasse** wird ein **eigener Satz Member-Funktionen** angelegt.

◇ Jede Member-Funktion einer generischen Klasse ist **implizit** eine **generische Funktion** (Funktions-Template) mit der **gleichen Typ-Parameterliste** wie die **generische Klasse**.
Bei ihrer **Definition außerhalb der Klassendefinition** muß sie daher **als Funktions-Template** (Beginn ebenfalls mit **`template < ... >`**) formuliert werden.

Beispiel :

```
template <class T> class List
{ public:
    List(T);                          // Konstruktor
    void add(List<T> *node);          // Element zur Liste hinzufügen
    T getData(void)const;             // Rückgabe Komponente data
  // ...
};

template <class T> List<T>::List(T d)  // generischer Konstruktor
{ data=d; next=NULL; }

template <class T> void List<T>::add(List<T> *node)
{ node->next=this; next=NULL; }

template <class T> T List<T>::getData(void) const
{ return data; }
```

◇ Für **spezielle aktuelle Parameter** - also für eine spezielle konkrete Klasse (spezielle Instanz des Klassen-Templates) – kann eine **Member-Funktion** auch **überladen** werden.
Bei einer expliziten Spezialisierung müssen alle Memberfunktionen gesondert definiert, d.h. überladen werden.

Beispiel :

```
List<char *>::List(char *s)      // Konstruktor für konkrete Klasse
{ data=new char[strlen(s)+1];    // Überladen des generischen Konstruktors
  strcpy(data, s);
  next=NULL;
}
```

◇ **Explizite Funktions-Templates** als Member-Funktionen sind auch **möglich**.

• Befreundete Funktionen generischer Klassen

◇ Sind **nicht implizit** generische Funktionen.

◇ Eine befreundete Funktion kann **von Template-Parametern unabhängig** und damit für alle aus dem Klassen-Template erzeugbaren **konkreten Klassen dieselbe** Freund-Funktion sein.

◇ Eine befreundete Funktion kann aber auch **von Template-Parametern abhängen** und damit für **jede konkrete Klasse unterschiedlich** sein. In diesem Fall sollte sie **explizit als generische Funktion** definiert werden.

Demonstrationsbeispiel zu generischen Klassen in C++ (1)

```
// ----------------------------------------------------------------------
// Header-Datei  StackTempl.h
// ----------------------------------------------------------------------
// Definition eines Klassen-Templates Stack
// ("einfacher Stack für unterschiedliche Objekte")
// ----------------------------------------------------------------------

#ifndef STACK_TEMPL_H
#define STACK_TEMPL_H

#define DEF_SIZE 10

template <class T> class Stack   // Generische Klasse "Stack für ..."
{                                // (Klassen-Template)
  public:
    Stack (int=DEF_SIZE);        // Konstruktor
    ~Stack();                    // Destruktor
    void push(T);                // Ablage eines Elements auf Stack
    T pop(void);                 // Rückholen eines Elements vom Stack
  private:
    int size;                    // Größe (Tiefe) des Stacks
    T* stck;                     // Pointer auf Speicherbereich des Stacks
    int tos;                     // Index des Top of Stack (Stackpointer)
};

#endif
```

```
// ----------------------------------------------------------------------
// C++-Quell-Datei  StackTempl.cpp
// Implementierung des Klassen-Templates Stack
// ----------------------------------------------------------------------

#include "StackTempl.h"

#include <iostream>
#include <cstdlib>
using namespace std;

template <class T> Stack<T>::Stack(int s)
{ stck=new T[size=s];
  tos=0;
}

template <class T> Stack<T>::~Stack()
{ delete [] stck;
  stck=NULL;
  size=tos=0;
}

template <class T> void Stack<T>::push(T c)
{ if (tos==size)
    cout << "Stack ist voll !\n";
  else
  { stck[tos]=c;
    tos++;
  }
}

template <class T> T Stack<T>::pop(void)
{ if (tos==0)
  { cout << "Stack ist leer !\n";
   return (T)0;
  }
  else
  { tos--;
    return stck[tos];
  }
}
```

Demonstrationsbeispiel zu generischen Klassen in C++ (2)

```
// --------------------------------------------------------------------------
// C++-Quell-Datei templstack_m.cpp
// --------------------------------------------------------------------------
// Programm templstack
// --------------------------------------------------------------------------
// Einfaches Demonstrationsprogramm zu Klassen-Templates in C++
// --------------------------------------------------------------------------
// Verwendung des Klassen-Templates Stack
// --------------------------------------------------------------------------

#include "StackTempl.h"
#include "StackTempl.cpp"   // nur bei impliziter Template-Instantiierung

#include <iostream>
using namespace std;

int main(void)
{
  Stack<char> cs1, cs2(20);
  int i;

  cs1.push('A');
  cs2.push('b');
  cs1.push('C');
  cs2.push('D');
  for (i=1; i<3; i++)
    cout << '\n' << cs1.pop();
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << cs2.pop();

  Stack<double> ds1(30), ds2;
  ds1.push(3.75);
  ds2.push(-4.7);
  ds1.push(2.3);
  ds2.push(5.83);
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << ds1.pop();
  cout << '\n';
  for (i=1; i<3; i++)
    cout << '\n' << ds2.pop();
  cout << '\n';
  return 0;
}
```

- **Ausgabe des Programms :**

```

C
A

D
b

2.3
3.75

5.83
-4.7
```

Generische Klassen (Klassen-Templates) in C++ (4)

• Andere Template-Parameter (Nicht-Typ-Parameter)

◇ Klassen-Templates können auch **Parameter** besitzen, die **keine Typen** sind, sondern normalen Funktionsparametern entsprechen.
Als entsprechende aktuelle Parameter sind nur zulässig :
- konstante Ganzzahl- oder Aufzählungstyp-Ausdrücke (bei Nicht-Referenz-Parametern)
- Adressen von global zugreifbaren Objekten und Funktionen (bei Pointer-Parametern)
- Referenzen auf Objekte, die global (Speicherklasse `extern`) oder als `static`-Komponente definiert wurden (bei Referenz-Parametern)

Sinnvoll sind solche Nicht-Typ-Parameter, wenn über sie bestimmte Eigenschaften einer konkreten Klasse oder ihrer Komponenten festgelegt werden können (z.B. die Größe eines Arrays o.ä.)

◇ **Beispiel :**

```
template <class T, int i> class Stack
{ public:
    Stack (void);
    void push(T);
    T pop(void);
  private:
    int size;
    T stck[i];  // Größe des Stacks durch Template-Parameter bestimmt
    int tos;
};

template <class T, int i> Stack<T,i>::Stack(void)
{
  size=i;
  tos=0;
}

// ...

int main(void)
{
  Stack<char,20> cs1;   // cs1 und cs2 sind Objekte
  Stack<char,50> cs2;   // unterschiedlicher Klassen
  Stack<char,2*25> cs3; // cs2 u. cs3 sind Objekte der gleichen Klasse
  Stack<double,30> ds;
  // ...
}
```

◇ **Zwei** aus **einem Klassen-Template** erzeugte **konkrete Klassen** (Template-Klassen) sind nur **dann gleich**, wenn sie in den **aktuellen Typ-Parametern übereinstimmen** und ihre aktuellen **Nicht-Typ-Parameter gleiche Werte** haben (s. obiges Beispiel).

• `typedef`-Namen für Template-Klassen

◇ Häufig eingeführt, um die wiederholte Auflistung von Template-Parametern zu vermeiden.

◇ **Beispiel :**

```
typedef Stack<char,20>   CharStack20;
typedef Stack<char,50>   CharStack50;
typedef Stack<double,30> DoubleStack30;
// ...
CharStack20 cs1;
CharStack50 cs2;
DoubleStack30 ds;
```

Generische Klassen (Klassen-Templates) in C++ (5)

• Generische Klassen mit statischen Komponenten

◇ Wenn eine generische Klasse statische Komponenten besitzt, werden für **jede** von ihr erzeugte **Template-Klasse eigene Vertreter** dieser **statischen Komponenten** angelegt.

◇ Die **Definition** (Speicherplatz-Reservierung) für die **statischen Datenkomponenten** kann
- für alle Template-Klassen mittels einer **Template-Vereinbarung** gemeinsam formuliert werden oder
- für jede Template-Klasse gesondert erfolgen (z.B. zur getrennten Angabe von Initialisierungswerten)

◇ **Beispiel** :

```
template <class T> class X
{ //  ...
  public :
    static T s;
  //  ...
};

template <class T> T X<T>::s;     // Template-Vereinbarung für stat. Datenkomp.

double X<double>::s = 5.26;

int main(void)
{
  X<int> xi;         // X<int> besitzt statische. Komponente s vom Typ int
  X<char*> xcp;      // X<char*> besitzt statische Komponente s vom Typ char*

  X<int>::s=12;
  X<char*>::s="Hallo !";

  cout << endl << "s von X<int>    : " << X<int>::s;
  cout << endl << "s von X<char*>  : " << X<char*>::s;
  cout << endl << "s von X<double> : " << X<double>::s;
  cout << endl;

  return 0;
}
```

• Klassen-Templates als Template-Parameter

◇ Eine generische Klasse (oder eine generische Funktion) kann auch **Parameter** besitzen, die selbst **Klassen-Templates** sind (→ **Verschachteln von Templates**).

◇ **Beispiel :** `Stack<Stack<int> > istackstack;        // Stack von int-Stacks`

◇ **Anmerkung** :Die beiden spitzen Klammern '**>**' müssen durch mindestens ein Leerzeichen getrennt werden, damit sie nicht als Operator '**>>**' interpretiert werden.

• Ort von Template-Definitionen

Klassen- (und Funktions-)Templates dürfen **nur** auf der **globalen Ebene** oder **innerhalb** einer **Klasse** bzw eines **Klassen-Templates** (***member templates***) definiert werden.
→ Klassen-Templates können also auch **innere Klassen**, **nicht** jedoch **lokale Klassen**, beschreiben.

Generische Klassen (Klassen-Templates) in C++ (6)

• Default-Argumente von Templates

◇ Für Template-Parameter (sowohl von Klassen-Templates als auch Funktions-Templates) können auch **Default-Werte** festgelegt werden.
Dies gilt sowohl für Typ-Parameter als auch für Nicht-Typ-Parameter.

◇ Die Festlegung von Default-Parametern erfolgt
- entweder in der **Template-Definition**
- oder in einer **Template-Deklaration** in einem Modul

Beispiel :
```
template <class T =double, int n = 256>
  class Array
  { /* ... */ };
```

◇ In der **gleichen Übersetzungseinheit** (Modul) darf für ein- und denselben Template-Parameter **nur eine Default-Festlegung** angegeben werden.

◇ Default-Werte für Template-Parameter müssen – wie bei Funktions-Parametern – immer **am Ende der Parameterliste** angegeben werden.
Wird für einen Parameter ein Default-Wert festgelegt, so müssen für alle folgenden Parameter ebenfalls Default-Werte angegeben werden.

Beispiel :
```
template <class T1 = int, class T2> class B;      // Fehler !
```

◇ Bei der **Instantiierung** eines Templates mit Default-Parametern können dann die **entsprechenden aktuellen Parameter weggelassen** werden.

Beispiel :
```
Array<char> buffer;     // Typ : Array<char, 256>
```

◇ Existieren für alle Template-Parameter Default-Festlegungen und werden auch alle verwendet, kann bei der Angabe der Template-Klasse eine **leere Parameterliste** angegeben werden.

Beispiel :
```
Array<> polvec;         // Typ : Array<double, 256>
```

Achtung : Die leere Parameterliste (öffnende und schliessende spitze Klammern) muss angegeben werden.

• Anwendung von Klassen-Templates

◇ Die Hauptanwendung von Klassen-Templates liegt in der Implementierung von **Container-Klassen**.
Container-Klassen sind Klassen, deren Objekte zur **Verwaltung anderer Objekte** dienen. Die verwalteten Objekte sind entweder als Komponenten enthalten oder werden über enthaltene Pointer-Komponenten referiert.
Typische Beispiele hierfür sind Stacks, Listen, assoziative Arrays, Mengen u.ä.

◇ Container-Klassen besitzen die **besondere Eigenschaft**, daß der **Typ der Objekte**, die sie verwalten, für die Definition der Klasse von **untergeordnetem Interesse** ist.
Im **Vordergrund** stehen die **Operationen**, die mit den Objekten - unabhängig von ihrem Typ - auszuführen sind.
Mit entsprechend definierten Klassen-Templates lassen sich dann gleichartige Container-Klassen für "Verwaltungs"-Objekte unterschiedlichsten Typs realisieren.

◇ In der **ANSI-C++-Standardbibliothek** sind mehrere derartige Klassen-Templates enthalten :
u.a. **deque**, **list**, **vector, stack** (→ **STL – Standard Template Library**)