Inhaltsverzeichnis

1. Templates in C++

☑ **Inhalt:**
☐ Aufbau und Funktionalität der Templates kennen lernen.

☑ **Ziel:**
☐ STL in eigenen Programmen/Problemlösungen einsetzen können.

☑ **Voraussetzungen:**
☐ C, C++ Kenntnisse

1.1. Was sind Templates

☑ Quelle: aus Wikipedia, der freien Enzyklopädie

Templates oder Schablonen sind „Programmgerüste“, die eine vom Datentyp **unabhängige** Programmierung ermöglichen.

Auch die C++-Standardbibliothek stellt viele nützliche Komponenten in Form eines Template-Frameworks – der **Standard-Template-Library** (STL)- zur Verfügung.

Es gibt zwei Arten von Templates in C++:
☑ **Funktionstemplates** und
☑ **Klassentemplates**.

1.2. Funktionstemplates

Ein **Funktionstemplate** (auch *Templatefunktion* genannt) verhält sich wie eine Funktion, die in

der Lage ist, Argumente verschiedener Typen entgegenzunehmen, und/oder verschiedene Rückgabetypen zu liefern.

Bisher konnten wir durch die Methode des Überladens von Funktionen folg. erreichen:

```
short Max(short x, short y){
     if (x < y)
          return y;
     else
          return x;
}

long Max(long x, long y){
     if (x < y)
          return y;
     else
          return x;
}

float Max(float x, float y){
     if (x < y)
          return y;
     else
          return x;
}
```

Nun kann man mittels eines Templates diese Variante wesentlich eleganter lösen:

```
template <typename T>
T Max(T x, T y){
  if (x < y)
    return y;
  else
    return x;
}
```

Diese Schablone kann genauso aufgerufen werden wie eine Funktion:

```
cout << Max<int>(3, 7);   // gibt 7 aus
```

Das Ganze funktioniert sowohl für `int` also auch für `std::string` oder irgendeinen anderen Typ, für den der Vergleich `x<y` eine wohldefinierte Operation darstellt.

☑ Hinweis: Bei selbstdefinierten Klassen/Typen
macht man von **Operator-Überladung** Gebrauch, um die Bedeutung von < für den Typ festzulegen und dadurch die Verwendung von `Max()` für den betreffenden Typ zu ermöglichen.

Das Beispiel für sich genommen mag nicht besonders nützlich erscheinen, im Zusammenspiel mit der C++-Standardbibliothek erschließt sich aber eine enorme Funktionalität für einen

neuen Typ ganz einfach dadurch, dass man ein paar Operatoren definiert.

Allein schon durch die **Definition von <** wird ein Typ in die Lage versetzt, mit den Standardalgorithmen **std::sort() und std::binary_search()** zusammenzuarbeiten, sowie mit Datenstrukturen wie **Mengen, Stapeln, assoziativen Feldern**, usw.

Anmerkung:

Die C++-Standardbibliothek enthält bereits das Funktionstemplate `std::max(x, y)`. Es gibt entweder `x` oder `y` zurück, und zwar abhängig davon, welches der beiden Argumente größer ist.

☑ **Hinweis: template in die Header-Datei**
Arbeiten Sie mit getrennten Dateien für Deklarationen der Funktionen (Header-Dateien) und deren Definitionen (Quellcode-Dateien), so müssen Sie

das Funktions-Template immer mit in die **Header**-Datei aufnehmen!

Die endgültige Funktion wird ja erst beim Aufruf einer durch das Funktions-Template vorgegebenen Funktion erstellt, und dazu benötigt der Compiler den Code der Funktion.

1.2.1. Spezialisierung von Template Funktionen

☑ **Das Problem: Die Adressen werden verglichen**
Was passiert nun, wenn Sie die Funktion *Max(...)* mit zwei **C-Strings** (*char*-Zeigern) aufrufen, um die Strings miteinander zu vergleichen? Da der Compiler beim Aufruf der Funktion die formalen Datentypen durch die tatsächlichen ersetzt, generiert er Ihnen eine Funktion, die nicht die Strings sondern **nur deren Adressen** im Speicher vergleicht! Was also tun?

Zum einen können Templates für bestimmte Datentypen spezialisiert werden. Um für einen bestimmten Datentyp ein spezielles Funktions-Template zu erstellen, wird zunächst die *template*-Anweisung angegeben, jetzt jedoch mit einer leeren spitzen Klammer. Der Datentyp für den dieses Funktions-Template verwendet werden soll, wird dann nach dem Funktionsnamen in spitzen Klammer angegeben.

```
template<> char* Max<char*>(char* p1, char* p2)
```

Hier eine Zusammenfassung:

```
// Allgemeine Templatedefinition
template <typename T>
T Max(T x, T y){
  if (x < y)
    return y;
  else
    return x;
}

//Spezielles Funktions-Template
template<>
char* Max<char*>(char* x, char* y){
    if (strcmp(x,y) > 0)
        return x;
```

```
    else
        return y;
}


//Aufruf des speziellen Funktions-Template
char*  pMax;
pMax = Max<char*>(pName1, pName2);
```

1.2.2. Aufgabe: templateMax.cpp (u)

Vervollständigen Sie folgendes Programm:

```
// Allgemeine Templatedefinition
??????????????


//Spezielles Funktions-Template
???????????????

int main(){
        char s1[]="aaa";
        char s2[]="bbb";

        cout << Max(0,3) << endl;
        cout << Max(4.67,5.67) << endl;
        cout << Max<int>(0,3) << endl;
        cout << Max<char*>(s1, s2);

        cout << endl;
        return 0;
}
```

Anmerkung: Wenn die Parameter eindeutig zugeordnet werden können, kann die Type-Angabe entfallen (Max(0,3) entspricht demnach Max<int>(0, 3))

1.2.3. Beispiel: Sortieren verschiedener Arrays (templateSort.cpp)

Das folg. Programm zeigt die Anwendung von Funktions-Templates zum Sortieren von Arrays.

☑ Das **Funktions-Template Sort(...)**
zum Sortieren erhält als Parameter einen Zeiger auf den Beginn des zu sortierenden Arrays sowie die Anzahl der Arrayelemente.

Innerhalb von *Sort(...)* wird ein weiteres
☑ **Funktions-Template Swap(...)**
aufgerufen, das zwei beliebige Datenelemente vertauscht.

☑ Im Hauptprogramm wird dann ein *long*- und ein *double*-Array definiert und mit beliebigen Werten gefüllt.

```
    long    longArray[] = {1,-10,-2,20};
    double  doubleArray[] = {1.1f, 0.9f, -1.2f, 5.5f};
```

Beide Arrays werden dann zur Kontrolle zunächst im unsortierten Zustand ausgegeben, dann durch Aufruf des Funktions-Template *Sort(...)* sortiert und zum Schluss im sortierten Zustand nochmals ausgegeben.
Eine mögliche Programmausgabe:

Unsortierte long:
1, -10, -2, 20,
Sortierte long:
-10, -2, 1, 20,

Unsortiert double:
1.1, 0.9, -1.2, 5.5,
Sortierte double:
-1.2, 0.9, 1.1, 5.5,

☑ Hier die Lösung: templateSort.cpp (b)

```
// a.hofmann
// templateSort.cpp

#include <iostream>
#include <string>

using std::cout;
using std::endl;

// Funktionstemplate zum Tauschen von Werten beliebigen Datentyps
// ===============================================================
template <typename T>
void Swap(T& val1, T& val2){
    T temp(val1);
    val1 = val2;
    val2 = temp;
}

// Funktionstemplate zum Sortieren von Zahlen
// beliebigen Datentyps innerhalb eines Feldes
// ===========================================
// pValues ist der Zeiger auf den Beginn des Datenfeldes
// und noOfValues enthaelt die Anzahl der Daten
template <typename T>
void Sort(T pValues, int noOfValues){
    bool changed;       // Tauschflag

    // Tauschschleife
    do
    {
        // Tauschflag loeschen
        changed = false;
        // Alle Elemente vergleichen
```

```
        for (int index=0; index<noOfValues-1; index++)
        {

            // Falls getauscht werden muss
            if (pValues[index]>pValues[index+1])
            {
                // Werte tauschen
                Swap(pValues[index],pValues[index+1]);
                // Tauschflag setzen
                changed = true;
            }
        }
    } while (changed);
    // Schleife so lange durchlaufen, bis nicht mehr getauscht wurde
}


//
// HAUPTPROGRAMM
// =============
int main(){
    int     index;
    long    longArray[] = {1,-10,-2,20};
    double  doubleArray[] = {1.1f, 0.9f, -1.2f, 5.5f};


    cout<< "\n\nunsortiert long: "<<endl;
    for (int i=0; i < 4; i++)
        cout << longArray[i] << ", " ;

    cout<< "\n\nunsortiert double: "<<endl;
    for (int i=0; i < 4; i++)
        cout << doubleArray[i] << ", " ;

    Sort<long*>(longArray, 4);
    Sort<double*>(doubleArray, 4);

    cout<< "\n\nsortiert long: "<<endl;
    for (int i=0; i < 4; i++)
        cout << longArray[i] << ", " ;

    cout<< "\n\nsortiert double: "<<endl;
    for (int i=0; i < 4; i++)
        cout << doubleArray[i] << ", " ;

    cout << endl<<endl;
    return 0;
}
```

1.2.4. Beispiel: Sortieren von Objekt-Arrays (templateSortAdresse.cpp)

1.2.5. Aufgabe: templateSortAdresse.cpp (u)

Mit Hilfe der im vorherigen Beispiel aufgeführten Funktions-Templates *Swap(...)* und *Sort(...)* soll nun ein

☑ ARRAY aus Objekten sortiert werden.

```
Address *pAddress = new Address[SIZE];
```

☑ Die Adressdaten Name und Wohnort sind als *string*-Objekte innerhalb der Klasse Adresse abgelegt.

☑ Die Adressenliste selbst wird durch ein ObjektArray vom Typ *Address* implementiert.
☑ Für die Ausgabe der Adressdaten wird der überladene Operator << verwendet.

☑ Im Hauptprogramm wird eine Adressenliste für vier Einträge dynamisch erstellt und mit Adressdaten belegt. Zur Kontrolle werden die Adressdaten ausgeben.

TODO:
Ihre Aufgabe ist es nun, diese Adressenliste mit Hilfe der beiden Funktions-Templates *Sort(...)* und *Swap(...)* alphabethisch nach dem Namen zu sortieren.
So einfach diese Übung am Anfang auch scheinen mag, hier steckt die Schwierigkeit im Detail.

Hier ein paar Hinweise zur Lösung:

☑ An den beiden Funktions-Templates sind keinerlei Änderungen notwendig.

☑ Die Klasse *Address* benötigt **zwei zusätzliche überladene Operatoren**. Welche das sind, das sollen Sie selbst herausfinden.
Als kleiner Tipp: Sehen Sie sich die Funktions-Templates einmal genauer an, welche Operatoren dort verwendet werden.

☑ Zusätzlich müssen Sie der Klasse *Address* noch einen weiteren, ganz **bestimmten Konstruktor hinzufügen**. Dieser wird vom Funktions-Template *Swap(...)* benötigt.

Verwenden Sie folg. Programmfragment:

```
// N.N
// templateSortAdresse.cpp
// Array mit Addressobjekten sortieren
//
#include <iostream>
#include <string>
using std::cout;
using std::endl;

// Funktionstemplate zum Tauschen von Werten beliebigen Datentyps
// ===============================================================
template <typename T>
void Swap(T& val1, T& val2){
```

```
    T temp(val1);
    val1 = val2;
    val2 = temp;
}

// Funktionstemplate zum Sortieren von Zahlen
// beliebigen Datentyps innerhalb eines Feldes
// ===========================================
// pValues ist der Zeiger auf den Beginn des Datenfeldes
// und noOfValues enthaelt die Anzahl der Daten
template <typename T>
void Sort(T pValues, int noOfValues){
    bool changed;       // Tauschflag

    // Tauschschleife
    do
    {
        // Tauschflag loeschen
        changed = false;
        // Alle Elemente vergleichen
        for (int index=0; index<noOfValues-1; index++)
        {
            // Falls getauscht werden muss
            if (pValues[index]>pValues[index+1])
            {
                // Werte tauschen
                Swap(pValues[index],pValues[index+1]);
                // Tauschflag setzen
                changed = true;
            }
        }
    } while (changed);
    // Schleife so lange durchlaufen, bis nicht mehr getauscht wurde
}

// Definition der Klasse fuer die Adressdaten
class Address{
    std::string name;           // Name
    std::string location;       // Ort
public:
    Address()             // ctor, hat nichts zu tun
    {}
    void SetData(const char* const pN, const char* const pL);
    friend std::ostream& operator << (std::ostream& os, const Address&
obj2);
};

// Definition der Memberfunktionen
// Setzen der Objektdaten
void Address::SetData(const char* const pN, const char* const pL){
    name = pN;
    location = pL;
}
```

```
// Ueberladener Operator << fuer Ausgabe
std::ostream& operator << (std::ostream& os, const Address& obj2){
    os << "Name: " << obj2.name;
    os << "  Ort: " << obj2.location << endl;
    return os;
}

//
// HAUPTPROGRAMM
// =============
int main()
{
    int     index;

    // Objektfeld fuer Adressdaten anlegen
    const int SIZE = 4;
    Address *pAddress = new Address[SIZE];

    // Objektfeld mit Daten belegen
    pAddress[0].SetData("Karl Maier","AStadt");
    pAddress[1].SetData("Agathe Mueller","XDorf");
    pAddress[2].SetData("Xaver Lehmann","CHausen");
    pAddress[3].SetData("Berta Schmitt","FStadt");

    // unsortiertes Objektfeld ausgeben
    cout << "Unsortierte Adressen:\n";
    for (index=0; index<SIZE; index++)
        cout << pAddress[index] << endl;

    // Hier fuer die Uebung die Adressen sortieren und

    Sort<Address*>(pAddress, SIZE);

    // erneut ausgeben
    // ????????????????????????????????????????????


    delete [] pAddress;
    return 0;
}
```

1.3. Klassentemplates

Templates sind nicht nur auf Funktionen beschränkt, sondern können auch für Klassen eingesetzt werden.

Ein **Klassentemplate** wendet das gleiche Prinzip auf Klassen an. Klassentemplates werden oft zur Erstellung von generischen **Containern** verwendet.

Beispielsweise verfügt die C++-Standardbibliothek über einen Container, der eine verkettete Liste implementiert. Um eine verkettete Liste von `int` zu erstellen, schreibt man einfach

```
#include <list>
...
list<int> aList;
```

Eine verkettete Liste von Objekten des Datentypes `string` wird zu

```
#include <list>
#include <string>

...
list<string> aList;
```

Mit `list` ist ein Satz von Standardfunktionen definiert, die immer verfügbar sind, egal was man als Argumenttyp in den spitzen Klammern angibt. (siehe nächstes Arbeitsblatt: STL)

Wir wollen nun die Klasse CStack als Klassentemplate entwickeln.

Beachten Sie, dass die STL bereits über eine derartige Klasse verfügt. Das hier vorgestellte Beispiel soll als Demobeispiel dienen.

```
template <typename T>
class CStack{
    T *pData;
    ....
  public:
    CStack(int size){
        pData = new T[size];
        ....
    }
    bool Push(const T& val);
    bool Pop(T& val);
};
```

Achtung:

Beachten Sie besonders im Zusammenhang mit Klassen-Templates bei den Parametern von Memberfunktionen, dass Sie entweder mit Zeigern oder Referenzen arbeiten sollten. Vermeiden Sie nach Möglichkeit die direkte Übergabe eines Werts. Bei der Parameterübergabe eines Objekts werden sonst unter Umständen relativ viel kopiert.

1.3.1. Definition von Memberfunktionen innerhalb der Klasse

Werden Memberfunktionen innerhalb der Klasse definiert, so kann die Definition der Memberfunktion wie gewohnt erfolgen.

Aber denken Sie auch daran, dass in der Regel innerhalb der Klasse definierte Memberfunktionen als *inline*-Memberfunktionen betrachtet werden. Und dies kann unter Umständen den Code Ihres Programms beträchtlich vergrößern!

```
template <typename T>
class CStack{
    T *pData;
    ....
  public:
    CStack(int size){
        ....
    }

    bool Push(const T& val){
        pData[sIndex++] = val;
        ....
    }

    bool Pop(T& val){
        val = pData[--sIndex];
        ....
    }
};
```

Und noch ein Hinweis: Soll die dargestellte Klasse *Stack* auch Objekte verarbeiten können, so sollten Sie für die abzulegenden Klassen auch den **Zuweisungsoperator '='** definieren, da in den Memberfunktion *Push(...)* und *Pop(...)* Objektzuweisung statt finden!

1.3.2. Definition von Memberfunktionen außerhalb der Klasse

Werden die Memberfunktionen außerhalb der Klasse definiert, so ist eine auf den ersten Blick etwas verwirrende Definition erforderlich. Die allgemeine Syntax zur Definition einer Memberfunktion eines Klassen-Templates außerhalb der Klasse lautet:

```
template <typename T>
RETURNTYP CLASS<T>::METHODENNAME(....)
```

T ist wieder der formale Datentyp und *CLASS* der Name des Klassen-Templates.

Das nachfolgende Bespiel zeigt die Definitionen der Memberfunktionen *Push(...)* und *Pop(...)* der vorherigen Klasse *Stack*. Beachten Sie, dass zwischen dem Returntyp der Memberfunktion und dem Namen der Memberfunktion der Klassenname steht, gefolgt vom formalen Datentyp in spitzen Klammern.

```
template <typename T>
bool CStack<T>::Push(const T& val)
{....}

template <typename T>
bool CStack<T>::Pop(T& val)
{....}
```

1.3.3. Definition von Objekten

```
// Objektdefinition + Template-Instanziierung
CStack<long> longStack(10);
CStack<char> charStack(50);
```

1.3.4. Aufgabe: templateStack.cpp (u)

Bringen Sie folg. Programmfragment zum Laufen:

```
//????????????????????????????????

int main(){

   CStack<int> iStack(100);
   CStack<string> sStack(100);

   for (int i=0; i < 100; i++)
      iStack.Push(i);

   for (int i=0; i < 100; i++){
      int wert;
      iStack.Pop(wert);
      cout << wert << ",";
   }

   cout << endl<<endl;

   return 0;
}
```

1.3.5. +Aufgabe: boundVector.cpp (t)

Lernziele:
☑ template
☑ vererbung, protected
☑ operator overloading

Datei:
☑ boundVector.cpp

Aufgabe1:
Erstellen sie die Klasse CVector als TEMPLATE mit folg. Aufbau:

```
// ------------------ Basis - Klasse
template <typename TYPE>
class CVector{
   pri.....:
      int size;
   pro.....:
      TYPE * v;
   ..........:
      CVector(){
```

```
            size=100;
            .............;
        }
        CVector(int len){
            size=len;
            .............;
        }
        ~CVector(){
            delete [] v;
        }
};
```

Aufgabe2:
Erstellen sie die Klasse BVector als Unterklasse der Klasse CVector (s.o.).
BVector enthalte die zusätzlichen Attribute/Member

```
  int l; // lower bound
  int r; // upper bound


// ------------------- Unter - Klasse
template <typename TYPE>
class BVector : public ..... {
    .........:
        int l; int r;
    .........:
        BVector(int left, int right): ............(right-left+1){
            l= left;
            r= right;
        }
        ....... operator[](int i){
            return ............::v[i-l];
        }
       ......... ostream& operator<<(ostream& os, const bvector<T>& other){
            for (int i=0; i< other.r-other.l+1; i++)
                        os <<other.v[i]<<' ';
            return os;
        }
};
```

Dadurch kann man Vektoren auf folg. Weise verwenden:

```
  BVector aVector(-10,10);  // einen vektor mit dem Index-Bereich -10 bis
                                  // +10 definieren

  aVector[-7]= 17;
```

Aufgabe3:
Erstellen sie das Testprogramm, um die Funktionsweise dieses Vektors ausgiebig zu testen.

```
// ------------------- Test - Programm
int main(int argc, char *argv[])
{
  BVector<int> aVector(-10,10);
```

```
 for (int i=-10; i<11; i++)
   aVector[i]=0;

 aVector[-7]=9;

 cout << aVector  << endl;

 return 0;
}
```

Fragen:
☑ auf welche Weise kann eine Unterklasse direkt auf die Attribute der Oberklasse zugreifen?
Nenne Nach/Vorteile dieser Methode!

1.3.6. +Aufgabe: MyStack.c (m)

Schreiben Sie folg. Programm derart um, dass es sich um ein Klassen-Template handelt.

```
class IntStack {
public:
     IntStack() { s=0; size=0; topindex=-1; }
     ~IntStack() { if (s!=0) delete [] s; }

     void push(int e);
     int top() const { /* Fehler-Check */
          return s[topindex];
     }
     int pop() { /* Fehler-Check */
          return s[topindex--];
     }
     bool isEmpty() const { return (topindex==1); }
private:
     int* s;
     int size;
     int topindex;
};

inline void IntStack::push(int e){
     if (size <= topindex+1){ // Platz auf dem Stack schaffen
          int* news = new int[size=size*2+1];
          for (int i = 0; i <= topindex; i++)
               news[i] = s[i];
          if (s != 0) delete [] s;
          s = news;
     }
     s[++topindex] = e;
}
```

1.4. Ausblick

Im nächsten Kapitel wollen wir uns mit der STL beschäftigen.