Templates

Bedeutung von Templates
in C++

Templates

- Motivation
  - In Standard C nur Makros
  - C++ Templates Übersicht
- Funktions-Templates
  - Definition & Verwendung
  - Spezialisierung
- Klassen Templates
  - Definition & Verwendung
  - Sonderfälle
- Anwendungsfälle
- Anhang
  - Einbindung von Templates

Motivation: Warum Vorlagen / Schablonen ?

- Bisher sind allen Funktionen und Klassen für einen bestimmten Datentyp definiert
  - Für jeden neuen Datentyp muss die Funktion / Klasse neu definiert werden.
  - Neue Codierung und Wartung für seit langen getesteten Code
- Auch Überladen von Funktionen hilft nur wenig. Die Funktion heißt zwar gleich, muss aber dennoch codiert werden.

```
// Berechnet Mittelwert
int mittel (int const *x , int n)
{
   int summe = x[0];
   for (int i=1;i<n;i++)
        summe += x[i];
   return summe/n;
}
```

Für `int a[] = { 1,3,5,7,8}`
lautet der Aufruf: `mittel(a,5)`

```
float mittel(float const *x ,int n);

… für jeden Datentyp …
```

Ohne C(++) : Nur bedingt durch Precompiler

- Achtung – keine echte Funktion nur Inline-coding
- Es wird an jeder Stelle ersetzt
- Keine Typüberprüfung
- Typische Fehlerquellen:
  - max(i++,n++);
    wird zu:
  - ((i++>n++)?i++:n++)
- Nur für sehr kleine "Debug" Tools sinnvoll
  - Heute eher nicht mehr verwenden

```
// MAX (A,B)
#define max(A,B) ((A>B)?A:B)
int main()
{
    int i=1,n=5, k;
    k=max(i,n);
    cout << i << n <<k<<endl;
}
```

```
wird zu  (Compiler Option /E )
int main()
{
    int i=1,n=5, k;
    k=((i>n)?i:n);
    cout << i << n <<k<<endl;
}
```

C++ Templates

- Für Funktionen
  - parametrisierte Funktionen / Funktions-templates
  - Generische Funktion
  - die Datentypen sind quasi "Variablen" (Parameter) für die Funktionsdefinition.

template <class Para> Para func (Para A) {…}

Schlüsselwort template

<class ….> (oder auch typename)

leitet Definition der Parameter ein

- Für Klassen
  - Parametrisierte Klassen / Klassen-templates
  - Generische Klassen

template <class Para> class classname {… Verwendung von Para…}

Funktions-Templates

- Es können 1-n Parameter definiert werden.
  - template <class P1, class P2> P1 funktion (P2 variable) {…}

- Der Geltungsbereich der Parameter ist die Template Deklaration
  - template <class P1, class P2> P1 funktion (P1 v1, P2 v2)
    {
      P1 x1 = v1* v2;
      return x1;
    }

- Aufruf/Verwendung des Funktions-Templates durch:
  - Name der Funktion und den in der Funktionsschnittstelle definierten Parametern
    - float x,a; int b;
      x = funktion (a,b); // x und a vom Typ P1 == float, b vom Typ int

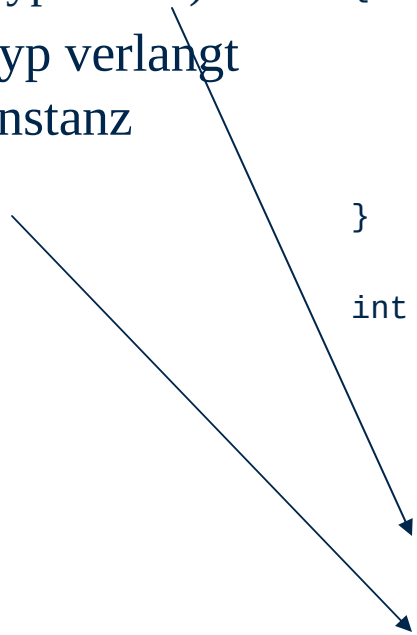
Funktions-Templates - Definition

- Funktion mittel als Template
  - int mittel (int const *x , int n)
- Typ int mittel (int const *x,…) wird ersetzt durch:
  - template <class **P**>
  - Hierbei ist der feste Typ int durch den Parameter **P** ersetzt
- Der Parameter P wird in der Deklaration der "Funktion" (mehrfach) verwendet
- Die Funktion selbst kann noch weitere Parameter (hier int n) verwenden

```
// mittel als Template


template <class P> P
        mittel(P const *x, int n)
{
    P summe = x[0];
    for (int i=1;i<n;i++)
          summe += x[i];
    return summe/n;
}

int main () {
    int a[] = { 1,4,2,5,7};
    int b[] = { 1,4,2,5,7,3,4,5,6};
    double c[] = {4.2, 2.0, 5.6, 7.3};
    cout << mittel(a,5) <<endl;
    cout << mittel(b,9) << endl;
    cout << mittel(c,4) << endl;
    return 0;
}
```

Funktions-Templates - Verwendung

- Die Funktion wird beim ersten Auftreten mit dem erforderlichen Typ vom Compiler erzeugt
  - Instanziert (hier Typ a = int)
- Erst wenn ein neuer Typ verlangt wird, wird eine neue Instanz erzeugt. (hier double)



```
// mittel als Template


template <class P> P
       mittel(P const *x, int n)
{
    P summe = x[0];
    for (int i=1;i<n;i++)
          summe = summe + x[i];
    return summe/n;
}

int main () {
    int v = 5;
    int a[] = { 1,4,2,5,7};
    int b[] = { 1,4,2,5,7,3,4,5,6};
    double c[] = {4.2, 2.0, 5.6, 7.3};
    cout << mittel(a,v) <<endl;
    cout << mittel(b,9) << endl;
    cout << mittel(c,4) << endl;
    return 0;
}
```

Funktions-Templates - Verwendung

- Auch eigene Klassen (Typen) können verwendet werden.
  - Achtung alle Operatoren müssen definiert sein
  - Hier verwendet mittel "+" und /

```
main ()
{
  RaumKo x(1.0,2.0),y(2.0,2.0),z(3.0,5.0);
  RaumKo w[] = {x,y,z}; //Array von RaumKo
  x.print();
  y.print();
  z.print();
  RaumKo m = mittel(w,3);
  m.print();
  return 0;
}
```

```
class RaumKo {
      double x,y;
  public:
      RaumKo (double a=0.0, double b=0.0)
          { x=a, y=b;}
      void print()
          {cout << " x,y : " << x
                << "," << y << endl;}
      RaumKo operator+ ( RaumKo Op2)
      {   RaumKo temp;
          temp.x = Op2.x + x;
          temp.y = Op2.y + y;
          return temp;  }
      RaumKo operator / (int div) const
      {   RaumKo temp = *this;
          temp.x /= div;
          temp.y /= div;
          return temp;  }
};

template<class Param> Param mittel (Param
    const *x , int n) { ... s.o.  }
```

Funktions-Templates - Zusammenfassung

- Definition "generischer" Funktionen mit:
  - einem oder mehreren "generischen" Parametern
  - zusätzlichen "normalen" Parametern
  - Vorteil: "Gleicher" Code gilt für alle Datentypen

- Spezialisierung
  - Hinweis: Auch Templates können spezialisiert werden
    Syntax: template <> **float** mittel<**float**> (const **float** *x, int n)
    - bei häufigen Ausnahmen ist die Verwendung von Templates fraglich

- Überladen von Funktions-Templates
  - Wie bei "normalen" Funktionen
    - template <class T> void func (T p1)
    - template <class T, class U> void func (T p1, U p2)

Klassen-Templates

- Ein Klassen-template funktioniert analog zu einer template-Funktion mit dem Unterschied, dass Klassen und nicht Funktionen generiert werden. Eine parametrisierte Klasse definiert, wie individuelle Klassen erzeugt werden können.
  - Synonyme: Klassenschablone, Klassen-template, generische Klasse
  - Definition: template <class T,...> class Xname { ...};
- Es sind mehrere Parameter erlaubt, darunter auch gewöhnliche Parameterdefinitionen:
  - template <class T, int n, class U> class Xname { ...};
- Da Templates zur Compilierungszeit instanziert werden, müssen Werte gewöhnlicher Parameter konstant sein.
  - Instanziierung: Xname<int> objekt;

Klassen-Template: Einfacher Stack

- Mit einem stack können Werte auf den Stapel abgelegt werden (push) und vom Stapel abgeholt werden (pop). Ein stack ist eine lineare Datenstruktur, auf die nur an einem Ende zugegriffen werden kann. Mit einer stack-Klasse wird die Implementierung der Datenstruktur des Stapels verborgen und der Zugriff nur über public-Elementfunktionen erlaubt.
  - push ()
  - pop()
  - isEmpty()
  - isFull()

```
template <class T, int size> class Stack
{
  private:
    T data[size];
    int top;

  public:
    Stack() : top(-1) {}

    void push(const T X)
     {
      assert(!isFull());
      data[++top]=X;
    }

    T pop()
    {
      assert(!isEmpty());
      return data[top--];
     }

    int isEmpty() const
    {return top==-1;}

    int isFull() const
    {return top==size-1;}
};
```

Klassen-Template: Einfacher Stack - Aufruf

- der normale Parameter size muss eine Konstante sein
  - Dadurch wird die private Datenstruktur zum Zeitpunkt der Instanziierung initialisiert
- der generische Parameter bestimmt den Datentyp
- In diesem Beispiel ist die Größe des Stack zum Compile Zeitpunkt festgelegt

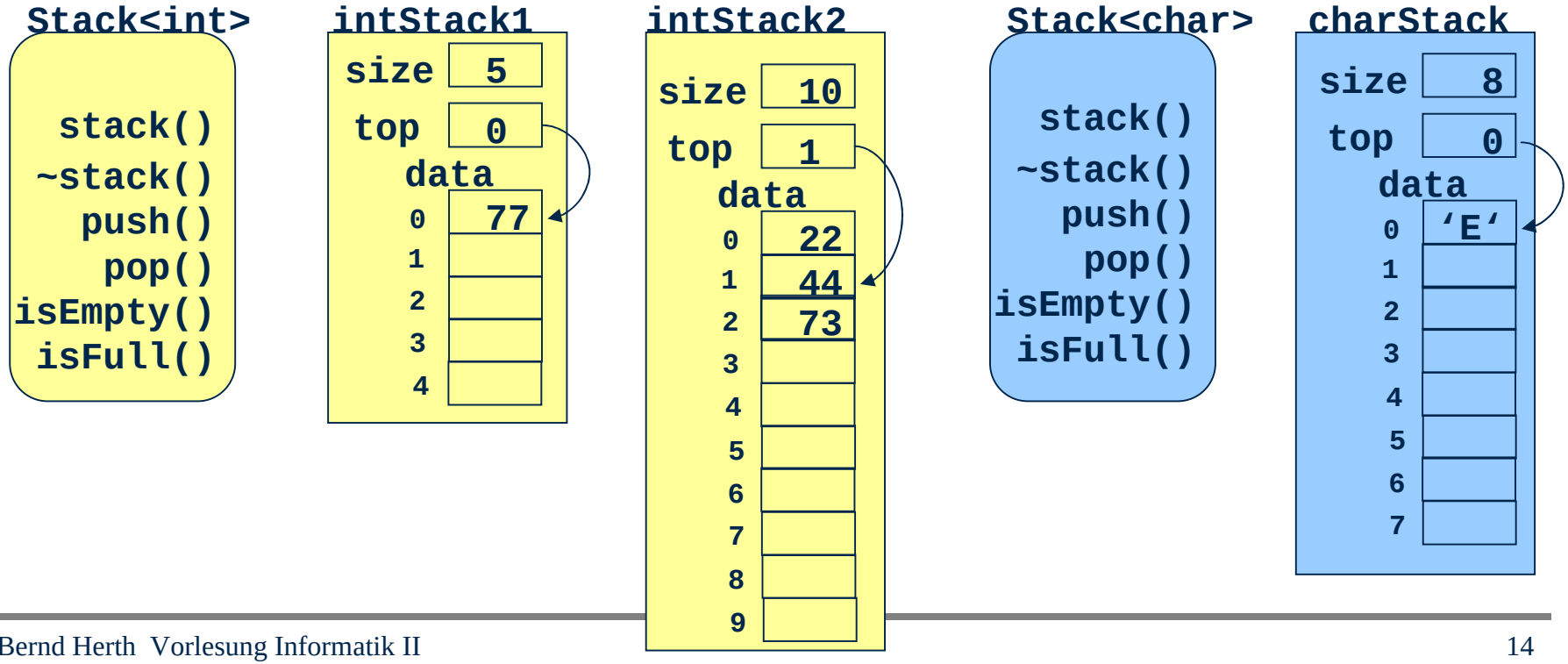
```
template <class T, int size> class Stack
{
  private:
    T data[size];
    int top;
……….

int main ()
{
    Stack<int, 5> S1;
    S1.push(5);
    S1.push(15);
    cout << S1.pop() <<endl<<
    S1.pop()<<endl;
    if (S1.isEmpty())
        cout << "Stack ist leer";
    else
        cout << "Stack ist voll";
    return 0;
 }
```

```
Ausgabe
15
5
Stack ist leer
```

Klassen-Template: Einfacher Stack - Zustandsanalyse

- Zustand der 2 Stack-Klassen und der 3 Stack Objekte nach dem letzten Befehl in Main.

```
int main ()
{
    Stack<int, 5> intStack1;
   Stack<int,10> intStack2;
   Stack<char,8> charStack;
    intStack1.push(77);
    intStack2.push(22);
    intStack2.push(44);
   charStack.push('E');
    intStack2.push(73);
   intStack2.pop();
```



Klassen-Template: Zusatz

- Im Unterschied zu Funktions-Templates können bei Klassen auch generische Parameter Default-Werte besitzen
  - `template <class T, int size> class Stack`
  - `template <class T=int, int size=100> class Stack`

```
int main ()
{
    Stack<int, 5> S1;   // Alle Werte angegeben
    Stack<> S2;         // Typ int, size 100
    Stack<double> S3;   // Typ double size 100
```

- Wie bei einem Funktions-Template kann auch bei einer Klasse eine Spezialisierung erfolgen
  - `template <class T> class Array`
  - `template <> class Array<float>`

  - Auch hier: Nur Sinnvoll wenn nicht zu häufig Ausnahmen

Typische Anwendungsfälle

- Umfangreiche getestete Algorithmen
  - Sortierung
- Bessere Implementierung von
  - Array, Vektoren,
  - Stack, Listen,…
  - mit dynamischer Speicherverwaltung
  - Vorkehrungen für Bereichsüberschreitung…

Anhang

- Sonderfälle beim Einbinden von Templates
  - Dateistruktur

Einbinden von Templates

- Die Template-Deklaration muss dem Compiler zur Compile-zeit zur Verfügung stehen.
- Bei großen Projekten kann es passieren, dass der Compiler bei jeder Datei (die ein Template benötigt) eine T-Funktion instanziiert. Dies führt zu langen Laufzeiten und großem Code.
- Geschicktes Aufsplitten der Template header Datei in:

1. meinTemplate.h // enthält template Deklaration mit Prototyp
2. meinTemplate.cpp // enthält umfangreiches Coding der Methoden

zusätzlich kann man das Instanziieren erzwingen.

Z.B. in einer kleinen Datei

3. Instanzen.cpp

- enthält: template class meinTyp<int>; // explizite Instanz.
  template class meinTyp<double>; // explizite Instanz.

Einbinden von Templates