

Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

- ▶ Klassen und Vererbung I
- ▶ IO-Streams

Tafelproblem der Woche: Operatoren innerhalb oder außerhalb von Klassen implementieren?

Fehlerbehandlung in C

- ▶ Es gibt verschiedene Möglichkeiten Fehler zu finden
 - Verwenden von `assert ()`
 - Ausgabe von Informationen während das Programm läuft (logging)
 - Benutzung eines Debuggers
- ▶ Diese Möglichkeiten sind komplementär
- ▶ Fehlerbehandlung in C:
 - C-Standard-Lib / Unix-Funktionen haben Rückgabewerte
 - `= 0` heißt „alles OK“
 - `< 0` heißt „Fehler“
 - Für die Windows API gilt das gleiche (`HRESULT`)
- ▶ Nicht sehr Informativ
- ▶ Das Propagieren von Fehlern ist ebenfalls nicht vollständig umgesetzt:
 - Die umschließende Funktion muss den Fehler korrekt weitergeben
 - Die umschließende Funktion muss angemessen reagieren

C++ Exceptions (1)

- ▶ Exceptions stellen ein Mechanismus zur Fehlerbehandlung zur Programmlaufzeit dar
- ▶ Programmcode, dass einen Fehler entdeckt, aber nicht weiß, wie darauf reagiert werden soll, wirft eine Exception
- ▶ Programmcode, der weiß, wie auf solche Fehler reagiert werden kann, fängt die Exception ab
- ▶ Kann der Aufrufer der Funktion sein, muss es aber nicht
- ▶ Komplementär zu anderen Fehlerbearbeitungsmechanismen, z.B., Assertions
- ▶ Eine Exception ist ein Wert, der einen Fehler beschreibt
- ▶ Kann ein Objekt oder gar eine einfache Primitive sein
- ▶ Normalerweise wird eine spezielle Klasse (oder eine Menge von Klassen) für Exceptions genutzt

C++ Exceptions (2)

- ▶ Die C++-Standard-Bibliothek stellt Exception-Klassen zur Verfügung
- ▶ Oft aber auch: Eigenes Exception Handling
- ▶ Werfen einer Exception:

```
if (index >= size)
    throw invalid_argument("Index too large!");
```

- ▶ Auffangen einer Exception mittels try-catch-Block:

```
SparseVector sv(10);
try {
    // This code should blow up.
    sv.setElem(318, 6);
}
catch (invalid_argument) {
    // It did blow up!
    cout<< "Oops..." << endl;
}
```

C++ Exceptions (3)

- ▶ Code innerhalb eines `try`-Blocks kann eine Exception werfen
- ▶ `catch`-blocks behandeln die geworfenen Exceptions
- ▶ Spezifizierung welche Exception abgefangen wird, steht am Anfang des `catch`-Blocks
- ▶ Namensgebung der Exception ist möglich:

```
SparseVector sv(10);  
try {  
    // This code should blow up.  
    sv.setElem(318, 6);  
}  
catch (invalid_argument &ia) {  
    // It did blow up!  
    cout << ia.what() << endl;  
}
```

- ▶ Details über die Fehler können so weiter gegeben werden
- ▶ Alle C++-Standard-Exceptions haben eine `what ()`-Methode

C++ Exceptions (4)

- Das Abfangen mehrerer Exceptions ist möglich:

```
try {  
    sv.setElem(318, 6);  
}  
catch (invalid_argument &ia) { // Invalid args.  
    cout << ia.what() << endl;  
}  
catch (bad_alloc &ba) { // Out of memory.  
    cout << "Ran out of memory." << endl;  
}  
catch (exception &e) { // Something else???  
    cout << "Caught another standard exception: "  
        << e.what() << endl;  
}
```

- Reihenfolge spielt eine Rolle!
- Der erste matchende catch-Block wird genommen

C++ Exceptions (5)

- ▶ Der catch-Block kann auch Exceptions auslösen:

```
catch (ProcessingError &pe) {  
    if (!recover())  
        throw CatastrophicError ("Couldn't recover!");  
}
```

- ▶ Die neue Exception wird aus dem gesamten try-catch-Block herauspropagiert
- ▶ Um die Exception erneut zu werfen, kann man einfach throw sagen (*rethrow*):

```
catch (ProcessingError&pe) {  
    numErrors++;  
    throw; // Rethrows exactly what we caught.  
}
```

- ▶ Funktioniert nur innerhalb des catch-Blocks

C++ Exceptions (6)

- ▶ Um alles abzufangen gibt es „...“:

```
try {  
    doSomethingRisky();  
}  
catch (...) {                // Catches ANYTHING!  
    cout << "Hmm, caught something..." << endl;  
}
```

- ▶ Für gewöhnlich keine gute Idee
- ▶ Tipps:
 - Wenn eine Exception in einer Funktion nicht abgefangen wird, dann terminiert diese Funktion
 - Lokale Variablen werden via Destructor aufgeräumt
 - Wenn die aufrufende Funktion keine Exceptions behandelt, wird diese ebenfalls beendet, u.s.w. ... („stack unwinding“)
 - Wenn `main()` keine exceptions behandelt, dann wird das Programm beendet.

Einführung in die Programmiersprache C++

... FÜR FORTGESCHRITTENE ...

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

C++ Templates (1)

- ▶ Oft müssen Funktionen mehrfach für verschiedene Datentypen implementiert werden müssen
- ▶ In vielen Fällen möchte man Datenstrukturen / Algorithmen unabhängig von den verwendeten Datentypen implementieren
- ▶ Datenstrukturen: Listen, Bäume, Matrizen, ...
- ▶ Algorithmen: Sortieren, Permutieren, Traversieren, ...
- ▶ Forderung: Code für einen Typ sollte für compatible Typen wiederverwendbar sein
- ▶ Bessere Übersicht und Wartung
- ▶ Beispiel:

```
struct node {  
    int value;  
    node *next;    // Pointer to next node in list  
};
```

Warum gerade ints ?



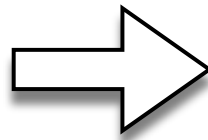
C++ Templates (2)

- ▶ C++ bietet die Möglichkeit, mit Hilfe von *Templates* (Schablonen, Vorlagen) eine parametrisierte Familien von Klassen und Funktionen zu definieren
- ▶ *Funktions-Templates* legen die Anweisungen einer Funktion fest, wobei anstellen eines konkreten Typs ein Parameter T gesetzt wird
- ▶ *Klassen-Templates* legen die Definition einer Klasse fest, wobei anstelle eines Types ein Parameter T eingesetzt wird
- ▶ Vorteile:
 - Ein Template muss nur einmal codiert werden
 - Einheitliche Lösung für gleichartige Probleme
 - Alle Typen, die die verwendeten Operationen unterstützen können die von Templates-Funktionen / -klassen zur Verfügung gestellten Funktionalität nutzen

C++ Templates (3)

- ▶ Beispiel: swap ()-Funktion
- ▶ Verhalten ist für jeden Typ gleich (elementar und andere!)
- ▶ Naiver Ansatz: schreibe für jeden Typ eine eigene überladene Funktion
- ▶ Besser: Definiere eine Schablone für diese Funktion
- ▶ Idee:

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

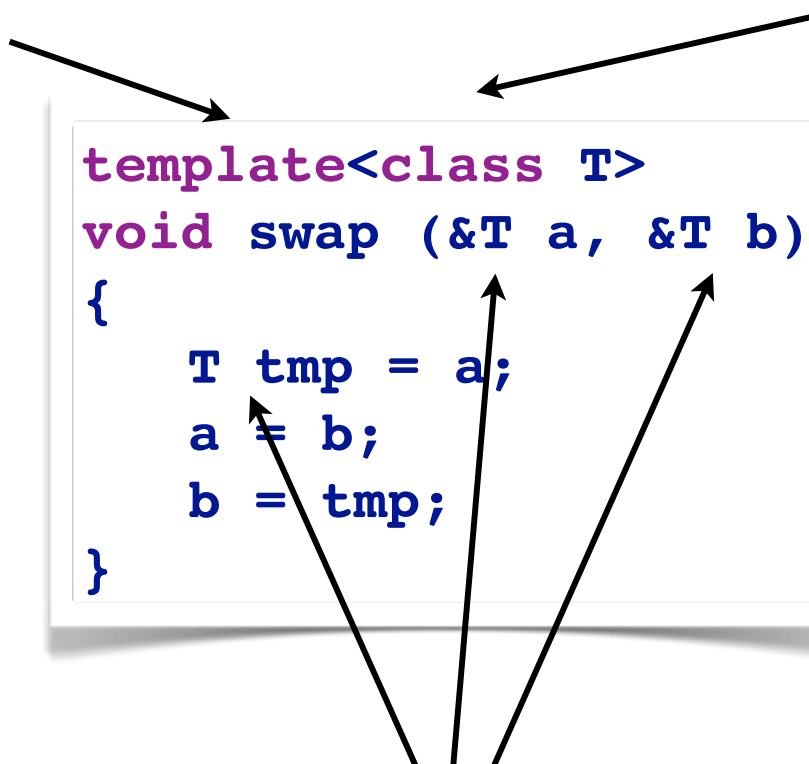


```
void swap(T &a, T &b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

C++ Templates (3)

Die `template<...>`-Zeile sagt dem Compiler: „*Die folgende Deklaration oder Definition ist als Schablone zu verwenden.*“

Ein Template erwartet als Argumente Platzhalter für Typen: *Templateparameter*



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

Die Platzhalter innerhalb des Templates werden später beim Compilieren durch spezifische Typen ersetzt.

C++ Templates (4)

- ▶ Templates sollen den Aufwand für Entwickler reduzieren:
 - Templates werden vom Compiler nach Bedarf in eine normale Funktion / Klasse umgewandelt
 - wird `swap()` für `int`-Werte verwendet, wird eine eigene Funktion / Klasse generiert
 - wird `swap()` für `Fruit`-Werte verwendet, wird eine eigene Funktion / Klasse generiert
 - Beim Kompilieren findet auch die Typprüfung statt
- ▶ Problem hierbei?
- ▶ Der Compiler muss die Templates bereits kennen, um passenden Code generieren zu können
- ▶ Daher muss der gesamte Template-Code inkludiert werden!

C++ Templates (4)

► Weiteres Beispiel

```
template<typename T>
class Point {
    T x_coord, y_coord;
public:
    Point() : x_coord(0), y_coord(0) {}
    Point(T x, T y) : x_coord(x), y_coord(y) {}
    T getX() { return x_coord; }
    void setX(T x) { x_coord = x; }
    ...
};
```

► Mit welchen Datentypen kann man Point instanzieren?

- T muss die Initialisierung mit 0 unterstützen!
- T muss einen Copy-Konstruktor besitzen!
- T muss einen Zuweisungs-Operator haben!

C++ Templates (5)

- Erweiterung der Point-Klasse:

```
template<typename T>
class Point {
    ...
    T distanceTo(Point<T> &other) {
        T dx = x_coord-other.getX();
        T dy = y_coord-other.getY();
        return (T)sqrt((double)(dx * dx + dy * dy));
    }
};
```

- T muss jetzt Addition, Subtraktion und Multiplikation unterstützen
- „Casting“ von double nach T und von T nach double
- Sehr viele Bedingungen für die Datentypen im Template!

C++ Templates (6)

- ▶ Ein großes Problem mit Templates ist, dass man nicht explizit spezifizieren muss, welche Eigenschaften / Operationen die Klasse T unterstützen muss.
- ▶ Wenn notwendige Operationen nicht von T unterstützt werden, sieht man nur eine Menge kryptische Fehler (beim Übersetzen)
- ▶ Dokumentation ist wichtig
- ▶ Falls Ihr STL benutzt, werdet Ihr Euch daran gewöhnen...

C++ Templates (7)

- ▶ Die Umwandlung eines Templates in eine übersetzbare Einheit nennt man *Template-Instanzierung*
- ▶ Das Template (Funktion oder Klasse) wird für einen bestimmten Typ (den Template-Parameter) *spezialisiert*
- ▶ Syntax für explizite Spezialisierung

```
double d1 = 4.0;  
double d2 = 2.0;  
swap<double>(d1, d2);
```

- ▶ Explizit bedeutet, der Programmierer spezifiziert ausdrücklich, mit welchem Typ die Template-Parameter zu ersetzen sind
- ▶ Es geht auch anders...

C++ Templates (8)

- ▶ Templates können implizit spezialisiert werden
- ▶ Typparameter tauchen an verschiedenen Stellen eines Templates auf:
 - Funktionsargumente
 - Rückgabewerte
 - lokale, statische oder Member-Variablen
- ▶ Der Compiler kann Templates automatisch zuweisen, wenn alle Parameter eindeutig aus dem Programmkontext abgeleitet werden können:

```
double d1 = 4.0;  
double d2 = 2.0;  
swap(d1, d2);           // Ok, T has to be a double!
```

- ▶ d1 und d2 sind eindeutig als doubles bekannt

C++ Templates (9)

- ▶ Anderes Beispiel: Minimum von zwei Werten

```
template<class T>
T min(T a, T b) {
    return ((a < b) ? a : b);
}
```

- ▶ Beispiel:

```
double d1 = 4.0, d2 = 2.0;
double m = min(d1, d2);
```

- ▶ Anderes Beispiel:

```
double d1 = 4.0;
float f1 = 10.0f;
double m = min(d1, f1);
```

- ▶ Das geht nicht!!!
- ▶ Der Compiler meldet:

```
% No matching function call to min(double&, float&)
```

C++ Templates (10)

```
template<class T>
    T min(T a, T b) {
        return ((a < b) ? a : b);
    }
```

► Entweder:

Implizite Typkonvertierung von <f1>

```
double m = min<double>(d1, f1);
```

► oder:

Explizite Typkonvertierung von <f1>

```
double m = min(d1, double(f1));
```

C++ Templates (11)

► Template-Klassen

```
template <class T>
class Stack {
private:

    T* inhalt;           // Datenbereich des Stacks
    int index, size;    // Aktueller Index, Grösse des Stacks
public:

    Stack(int s): index(-1), size(s) {
        inhalt = new T[size]; // Stack als Array implementiert
    }

    void push(T item) { // Ein Element auf den Stack "pushen"
        if (index < (size - 1)) {
            inhalt[++index] = item;
        }
    }
};
```

C++ Templates (12)

- ▶ Bei der Allokierung eines Stacks muss explizit der Typ angegeben werden (<int>, <Rect>, ...)

```
int main() {  
    // Stack für 100 int  
    Stack<int> intStack(100);  
  
    // Stack für 250 double  
    Stack<double> doubleStack(250);  
  
    // Stack für 50 rect  
    Stack<rect> rectStack(50);  
  
    intStack.push(7);  
    doubleStack.push(3.14);  
    rectStack.push(rect(2, 5));  
}
```


C++ Templates (13)

- ▶ Auch hier gilt: Die entsprechende Klasse wird bei Bedarf vom Compiler generiert
- ▶ Trennung von Definition und Deklaration:

```
template <class T>
class Stack {
private:
    T* inhalt;
    int index;
    int size;
public:
    Stack(int s);
    void push(T item);
};
```

stack.hh

```
#include "stack.h"
using namespace std;

// Achtung: nicht Stack::Stack(int s)!
template <class T>
Stack<T>::Stack(int s):index(-1),size(s)
{
    inhalt = new T[size];
}

template <class T>
void Stack<T>::push(T item) {
    if(index<size) {inhalt[++index]=item;}
}
```

stack.cpp

C++ Templates (14)

► Benutzung des Stacks

```
#include "stack.cpp"
using namespace std;
int main() {
    Stack<int> intStack(100);
    Stack<double> doubleStack(250);
    intStack.push(7);
    doubleStack.push(3.14);
}
```

- Man beachte: `stack.cpp`, nicht `stack.h`!
- Bei Templates muss die komplette Implementierung eingebunden werden
- Die reine Deklaration reicht nicht

C++ Templates (15)

- ▶ Die Angabe des Datentyps bei Gebrauch eines Templates legt fest für was für welche Instanzen es gebraucht werden kann

```
Stack<Person> personStack(100);  
// Nur für Typ Person verwendbar
```

- ▶ Auch Unterklassen von Person können auf diesem Stack abgelegt werden
- ▶ Dabei werden die Objekte aber in Person konvertiert!
- ▶ Spezifische Funktionalität geht verloren (kein Polymorphismus)
- ▶ Lösung: Zeiger auf die Basisklasse speichern

```
Stack<Person*> personStack(100);  
// Für alle verwendbar
```

- ▶ Polymorphismus funktioniert
- ▶ Zusätzliche Information geht nicht verloren
- ▶ Wenn man weiss worauf man casten muss....

C++ Templates (16)

► Konstante Elementartyp-Ausdrücke

```
template <class T, int N>
class Buffer {
    T v[N];
public:
    void clear () {
        for (int i=0; i<N; ++i) v[i] = T(0);
    }
};
```

- Ermöglichen evtl. Optimierungen zur Compilezeit (loop unrolling)
- Erlauben flexible Speichernutzung ohne dynamischen Speicher
- Parameter müssen zur Compilezeit auswertbar und konstant sein!

```
void f (int i) {
    Buffer<char, i> buf; //Fehler
}
```

C++ Templates (17)

- ▶ Templates von Templates...
- ▶ Templates können als Parameter für andere Template dienen:

```
complex<float> c1, c2;  
swap<complex<float> >(c1, c2);
```

- ▶ Anm: `complex<T>` ist eine Klasse der C++-Standard-Lib

Achtung!

Bei geschachtelten Templates muss man aufeinanderfolgende '>' durch Leerzeichen trennen, da sie sonst als Shift-Operator ('>>') fehlinterpretiert werden

C++ Templates (18)

- ▶ Verschachtelte Templates können unübersichtlich werden:

```
vector<map<int, complex<float> > > m;
```

- ▶ Mit typedef kann man Aliase festlegen:

```
typedef vector<map<int, complex<float> > > CMapVector;  
CMapVector m;
```

- ▶ typedefs definieren keine neuen Typen
- ▶ Daher sind folgende Spezialisierungen gleich:

```
typedef unsigned int UInt;  
Buffer<UInt, 20> buf1;  
Buffer<unsigned int, 20> buf2;
```

- ▶ Konstante Ausdrücke (zur Compile-Zeit) führen zu gleichen Template-Typen:

```
Buffer<int, 5*4> buf1;  
Buffer<int, 20> buf2;
```


C++ Templates (19)

- ▶ Ein Template existiert nicht als Typ oder Objekt
- ▶ Erst bei Instanzierung eines Templates entsteht ein neuer Typ
- ▶ Der Typ ist durch die Template Parameter definiert
- ▶ Dies hat Konsequenzen für den Umgang mit Template-Code:
 - Bestimmte Fehler können vom Compiler erst sehr spät erkannt werden
 - Der Compiler weiss nicht im Voraus, für welche Typen Spezialisierungen generiert werden sollen
 - Fehler, die durch Templateparameter entstehen, können nicht vor der ersten Benutzung erkannt werden!!!

C++ Templates (20)

- Wo / Warum beschwert sich der Compiler?

```
#include <complex>
using namespace std;
/* ... */
complex<double> a, b;
complex<double> c = min<complex<double> >(a, b);
```



```
template <class T>
T min (T a, T b) {
    return ((a<b) ? a : b);
}
```

In function `T min(T, T) [with T = std::complex<double>]`:
...
no match for `std::complex<double>& < std::complex<double>&`
operator

Templates - Code-Organisation (1)

► Ohne Templates:

- Interfaces und Implementierungen werden in separaten Dateien untergebracht
- Der Linker stellt die Verbindung mit dem dazugehörenden Objektcode her

► Mit Templates:

- Der Code wird erst bei Bedarf (durch Bindung an spezifische Template-Parameter) generiert
- Kann nur der vom Template generierte Code übersetzt und gelinkt werden

► ***Wo soll man die Implementierung unterbringen?***

Templates - Code-Organisation (2)

- ▶ **Variante 1:** *Erzwinge die Template-Instanziierung für eine feste Menge von Typen*

- ▶ .hh-Datei:

```
// Declaration
template <class T>
T min (T a, T b);
```

- ▶ .cc-Datei:

```
#include "myLib.h"
template <class T> // Implementation
T min (T a, T b); {
    return ((a<b) ? a : b);
}
template float min<float> (float, float);
```

- ▶ Ein Template ohne <> erzwingt die explizite Instanziierung des Templates
- ▶ Egal ob es irgendwo in dieser Ausprägung verwendet wird

Templates - Code Organisation (3)

- ▶ **Variante 2a:** *Behandle Code wie inline-Funktionen*

- ▶ .hh-Datei:

```
//Declaration and implementation
template <class T>
T min (T a, T b) {
    return ((a<b) ? a : b);
}
```

- ▶ .cc-Datei:

```
// empty
```

- ▶ Verwendung:

```
#include "myLib.h"
int x = min<int>(10, 100);
```

- ▶ Verlegt die Implementierung in die Header-Datei
- ▶ Legt die Implementierung offen
- ▶ Jeder, der das Template verwendet, muss den Code neu übersetzen

Templates - Code-Organisation (4)

- ▶ **Variante 2b:** *Wie 2a, aber packe die Implementation in separaten Header:*

- ▶ .hh-Datei:

```
//Declaration
template <class T>
T min (T a, T b);
```

- ▶ .icc-Datei

```
template <class T> // Implementation
T min (T a, T b); {
    return ((a<b) ? a : b);
}
```

- ▶ Gebrauch:

```
#include "mylib.h"
#include "mylib.icc"
int x = min<int>(10, 100);
```

Template-Spezialisierung (1)

- Manchmal kann es sinnvoll sein für bestimmte Templateargumente alternative Implementierungen vorzugeben

```
template <class T, int Size>
void MyVector::multiply (int d) {
    for (int i=0; i<Size; ++i) {
        data[i] *= T(d);
    }
}
```

Standard (fallback) Implementation

```
template <>
void MyVector<double,2>::multiply (double d) {
    data[0] *= d;
    data[1] *= d;
}
```

Optimierte Implementation

Template-Spezialisierung (2)

- ▶ Die Spezialisierung kann nachträglich hinzugefügt werden
- ▶ Bestehender Code muss dazu nicht abgeändert werden

```
MyVector<int, 10> a;  
a.multiply(42.0);           // Nutzt Standard Version  
MyVector<double, 2> b;      // Nutzt automatisch die  
b.multiply(42.0);           // spezialisierte Version
```

- ▶ Es sind auch partielle Spezialisierungen möglich

```
template <class T>  
void MyVector<T,2>::multiply (double d) { ... }
```

Templateklassen - Fallstricke (1)

► Beispiel:

```
template<typename T>
class List
{
public:
    struct node
    {
        node(T t) : data(t);
        T data;
        node* next;
    };
};
```

► Jetzt wollen wir in der Implementierung eine node-Variable anlegen:

```
List<int>::node my_node(5);
```

► Geht so nicht!

Templateklassen - Fallstricke (2)

- ▶ Node hängt vom Templateparameter T ab
- ▶ C++ geht in solchen Fällen standardmäßig davon aus, dass `List::node` eine Methode oder Variable ist.
- ▶ Daher muss man extra sagen, dass es ein Typ ist:
- ▶ `typename List::node my_node` funktioniert.
- ▶ Weiteres Beispiel:

```
template<typename T>
class A
{
public:
    T t;
    int i;
};
```

```
template<typename T>
class B : public A<T>
{
public:
    B()
    {
        m = 5;
        i = 6;
    }
};
```


Templateklassen - Fallstricke (3)

- ▶ Die Member der Oberklasse werden nicht gefunden
- ▶ Bei Templates müssen die Member der Oberklasse immer direkt mit der aktuellen Instanz verknüpft werden.
- ▶ Lösung:

```
template<typename T>
class B : public A<T>
{
public:
    B()
    {
        this->m = 5;
        this->i = 6;
    }
};
```