Programmieren 3 (C++)

Zusammenfassung v1.3

Kälin Thomas, Abteilung I
SS 06

1. Grundlagen

1.1. Dateien

1.1.1. Sourcefiles (*.cpp)

- Diese Dateien enthalten implementierenden Quellcode und werden auch „Sourcefiles“ genannt.
- Beispiele: Code von Methoden, Definition und Initialisierung von statischen Objekten, Code der Methoden von Klassen

1.1.2. Headerfiles (*.h)

- Die Headerfiles kann man mit Schnittstellen vergleichen. Diese werden mittels #include in Sourcefiles eingebunden. Es enthält für gewöhnlich nur Deklarationen, keinen implementierenden Code.
- Beispiele für den Inhalt eines Headerfiles: Deklaration von Funktionen, Deklaration von statischen Objekten, Definition von Klassen (oder allg. Typen), Definition von Globalen Konstanten.
- Ein Headerfile muss so gestaltet sein, dass es unabhängig von einem einbindenden Sourcefile in sich komplett ist.
- Nachteil: Bei unvorsichtigem Vorgehen kann sehr leicht die ODR (One-Definition-Rule) verletzt werden.

1.1.3. Include

```
#include <iostream>      //Headerfiles der Standardbibliothek
#include <whatEver.h>    //Headerfiles von anderen Bibliotheken
#include „drugs.h“       //Eigene Headerfiles. („“ -> lokales Verzeichnis)
```

- Regeln: Nur wirklich benötigte Headerfiles einbinden. Header nach Möglichkeit in Sourcefiles einbinden. #include innerhalb von Namespaces verwenden.

1.1.4. Beispiel *.cpp & *.h

```
Beispiel.h                //Schnittstelle
#ifndef HELPERS_H_
#define HELPERS_H_
#include <string>
const std::string strBeispiel = „Test!“;
std::string getMessage();
#endif /*HELPERS_H_*/

Beispiel.cpp              //Implementierender Code
#include „Beispiel.h“
std::string getMessage() {
 return strBeispiel;
}

Main.cpp
#include „Beispiel.h“
#include <iostream>
int main() {
 std::cout << getMessage();
}
```

1.2. Definition vs. Deklaration

1.2.1. Deklaration

- Eine Deklaration ist eine Anweisung, die einen Namen in ein Programm einführt. Sie legt ausserdem einen Typ für diesen Namen fest. Jeder Name, der in einem Programm verwendet wird, muss zuvor deklariert worden sein.
- Es darf mehrere Deklarationen für einen Namen geben, sofern diese im Typ übereinstimmen

1.2.2. Definition

- Eine Definition spezifiziert eine Entität, auf die sich ein Name bezieht. Eine Entität kann z.B. folgendes sein: Menge an Speicherplatz, Code einer Funktion, konstanter Wert, neuer Typ (Klasse).
- Es muss immer genau eine Definition für einen Namen geben (One Definition Rule).

- Jede Definition ist auch immer eine Deklaration

1.2.3. Beispiele

```
char c;                          //Dek & Def
int count = 1;                   //Dek & Def
const double pi = 3.14159;       //Dek & Def
extern int fehler_nummer;        //Dek (Bekanntmachung des Namens!)
struct Datum {int t, m, j; };    //Dek & Def
double sqrt(double);             //Dek
typedef complex<short> Punkt;    //Dek & Def
namespace NS { int a; }          //Dek & Def
```

1.3. Initialisierung vs. Zuweisung

1.3.1. Initialisierung

Die Instantierung findet bei der Objektgenerierung statt.

1.3.2. Zuweisung

Eine Zuweisung kann zu jedem Zeitpunkt stattfinden, das Objekt besteht bereits

```
int a = 15;        //Definition und Initialisierung
int b;             //Nur Definition
b = 16;            //Zuweisung
```

1.4. Gültigkeit

- Globale und innerhalb eines Namespace definierte Objekte / Variablen werden mit dem passenden 0-Wert automatisch initialisiert
- Lokale Variablen / Objekte werden nicht standardmässig initialisiert

1.5. Namespace

- Namespaces ermöglichen es, die Gültigkeitsbereiche von Namen einzuschränken, bzw. voneinander abzugrenzen.
- Alle Elemente der Standardbibliothek sind innerhalb des Namespaces „std“ definiert. Auf diese Elemente wird daher wie folgt zugegriffen:

```
std::AnyName; //std = Name des Namespaces, :: = Scope Operator
```

1.5.1. Verwendung vereinfachen

- Mittels „using namespace std“ wird dem Compiler mitgeteilt, dass er für unbekannte Wörter in diesem Bereich auch im Namespace „std“ gesucht werden soll.

```
using namespace std;
```

1.5.2. Anonyme Namespaces

- Anonyme Namespaces ermöglichen es, die Gültigkeit von globalen Elementen auf die Kompiliereinheit einzuschränken. In diesen anonymen Namespaces steckt immer auch eine „using directive“. Elemente des AS können in der Kompiliereinheit ohne weitere Qualifizierung verwendet werden.

```
namespace {
 long counter;
} //Implizit ein using namespace; ausgeführt!
```

1.6. Include-Guard

- Ein Include-Guard stellt sicher, dass ein Headerfile beim Kompiliervorgang nur einmal eingebunden wird

```
#ifndef FILENAME_H_
#define FILENAME_H_
//…
#endif /*FILENAME_H_*/
```

1.7. Main-Return

Die main()-Funktion gibt bei fehlerfreier Ausführung den Wert 0 zurück. Dies ist eine alte C-Erblast.

2. Klassen

2.1. Beispiel einer Klasse

```
//Headerfile, Deklaration

class clsBeispiel {
private:                  //Label (Sichtbarkeit: Private)
 int intInteger;          //Instanzvariable
 static int intStatic;    //Klassenvariable

public:                   //Label (Sichtbarkeit: Public)
 clsBeispiel(int intV);   //Konstruktor mit Parameter
 ~clsBeispiel();          //Dekonstruktor, k.Rückgabewert, Immer Public!
 int getInteger();
};                        //Achtung: Semikolon am Ende der Klasse!

//Sourcefile, Definition
#include „headerfile"
clsBeispiel::clsBeispiel(int intV) : intInteger(intV),intStatic(0) {
}

int clsBeispiel::getInteger() {
 return this->intInteger;      //this ist ein Zeiger
}

//Mainfile
#include „headerfile"
clsBeispiel refClass(5);
```

2.2. Sichtbarkeiten

Die folgenden Schlüsselwörter dürfen mehrere Male in beliebiger Reihenfolge vorkommen. Standard (ohne Angabe) ist „private" drin.

- Public: Die Elemente sind für alle Sichtbar.
- Protected: Die Elemente sind für die eigene und alle abgeleiteten Klassen sichtbar.
- Private: Die Elemente sind nur innerhalb der Klasse sichtbar

2.2.1. Empfehlungen

- **public:** für alle Methoden, die die öffentliche Schnittstelle einer Klasse bilden. Set() / Get().
- **protected:** für die Methoden, die von der Klasse selbst und abgeleiteten Klassen benutzt werden dürfen
- **private:** für die Methoden, die nur von der Klasse selbst benutzt werden dürfen. Empfohlen auch für alle Attribute. Private Methoden werden nicht vererbt.

2.3. Konstruktor

Immer wenn ein Objekt zur Laufzeit instantiert wird, läuft ein Konstruktor ab. Das System wählt automatisch den passenden Konstruktor aus. Ohne Implementierung eines Konstruktors stellt das System automatisch einen Standard-Konstruktor zur Verfügung. Ziel des Konstruktors ist es, das Objekt in einen definierten Anfangszustand zu bringen.

Tritt inerhalb eines Konstruktor eine Excexption auf, wir das Objekt NICHT erstellt. In einem solchen Fall wird auch der Destruktor für das Objekt nie aufgerufen. Daher sollte man beim Resservieren von Ressourcen im Konstruktor vorsichtig sein, mögliche Probleme mit try-catch behandeln.

2.3.1. Init-Methode

Der Konstruktor selber führt nur wenige Initialisierungen aus. Die Hauptarbeit der Initialisierung wird von einer speziellen Init-Methode ausgeführ, die vom Benutzer selber aufgerufen werden muss. Dies bedingt, dass bei jeder öffentlichen Methode zuerst der Objekt-Zustand geprüft werden muss. Fazit: Zu teuer, Exceptions im Konstruktor sind der bessere Weg!

2.4. Destruktor

Immer wenn ein Objekt zur Laufzeit ungültig wird, läuft der Destruktor der entsprechenden Klasse ab. Es kann nur einen Destruktor pro Klasse geben. Auch hier stellt das System einen Destruktor zur Verfügung, wenn keiner Implementiert wird. Der Destruktor hat die Aufgabe, allfällige reservierte Ressourcen (meistens Memory) freizugeben.

2.5. Initialisierungsliste

- Die Datenelemente werden initialisiert, bevor der Code des Konstruktors läuft. Die Reihenfolge der Abarbeitung wird durch die Reihenfolge der Datenelemente in der Klassendefinition bestimmt.
- Zwingend für: Konstante Attribute, Attribute vom Typ Referenz, Initialisierung der Daten der Basisklasse bei Vererbung

```
clsBeispiel::clsBeispiel(intParam=5)
 :intInteger(intParam),dlbDouble(3.1415) {}
```

2.6. Struct vs Class

Der einzige Unterschied zwischen „struct“ und „class“ ist, dass bei ersterem der Default-Sichtbarkeitsbereich „public“ ist und bei zweitem (class) „private“.

2.7. Klassen mit einstelligem Konstruktor (explicit)

Klassen mit einstelligem Konstruktor können automatisch Konverter sein. Der Compiler wird dann automatisch probieren, diesen anzuwenden.

```
class Person {                          |     int a = ‚a‘;
 public:                                |     String s = a;
       Person(std::string s);           |     s++;
       void operator++();               |
};                                      |
```

Um das zu vermeiden, muss der Konstruktor mit dem Keyword „explicit“ markiert werden.

```
explicit Person(std::string s);
```

2.8. Vererbung

```
class Baer : public Zootier {…};
class Panda : public Baer, public Pflanzenfresser {…};
```

- Objekte der abgeleiteten Klassen enthalten automatisch ein anonymes Objekt der Basisklasse. Dieses wird initialisiert, bevor der Code-Block des Konstruktors der Subklasse abgearbeitet wird. Deshalb sollten Objekte der Basisklasse mit der Initialisierungsliste initialisiert werden. Am bequemsten natürlich mittels den Konstruktoraufrufen. Einzelne, geerbte Elemente können jedoch NICHT über die Initialisierungsliste aufgerufen werden.
- Attribute & Methoden werden vererbt, ausser: Konstruktoren, Destruktoren, Zuweisungsoperatoren.
- Alle Zustände, die ein objekt einer Basisklasse annehmen kann, muss auch ein Objekt der abgeleiteten Klasse übernehmen können. Das heisst, dass Methoden und Attribute in Unterklassen keine neue Bedeutung erhalten dürfen.

2.8.1. Mehrfachvererbung

Mehrfachvererbung ermöglicht das elegante Abbilden von Strukturen der realen Welt. Sie kann jedoch auch zu Problemen führen:

- **Namenskonflikte**: Falls zwei oder mehrere Subklassen identische Funktionsnamen / Signaturen besitzen.
- **Speicherverschwendung**: Falls zwei Elternklassen dieselbe Basisklasse verwenden, existieren in der Subklasse zwei anonyme Objekte der Basisklasse. Lösung: virtuelle Basisklasse.

2.8.2. Up-Casting

```
Eisbaer Lars;
Baer EinBaer;
EinBaer = Lars;    //Informationsverlust, aber erlaubt
```

2.8.3. Private/Public Vererbungen

- Public: Alle Elemente der Basisklasse behalten ihre Sichtbarkeit auch in der abgeleiteten Klasse.
- Private: Alle public und protected Elemente der Basisklasse werden private Mitglieder der abgeleiteten Klasse.

2.8.4. Methoden überschreiben

Methoden der Basisklasse dürfen in Subklassen bewusst mit exakt derselben Signatur überschrieben werden.

2.9. Virtuelle Methoden

Das Schlüsselwort „virtual“ teilt dem Compiler mit, dass eine Methode als virtuelle Funktion behandelt werden soll. Dies bedeutet, dass die Methode erst zur Laufzeit gebunden wird. Innerhalb einer Vererbungshierarchie entscheidet das System zur Laufzeit anhand des Datentyps, welche Implementierung einer überschriebenen Methode verwendet wird.

Dieser Mechanismus funktioniert nur, wenn Methoden eines Objekts über Zeiger oder Referenzen aufgerufen werden!

```
class Tier {
 public:
       void gibLaut() { cout << "---"; }
       virtual void bewege() { cout << "---"; }
};

class Vogel : public Tier {
 public:
       void gibLaut() { cout << "Zwitscher"; }
       virtual void bewege() { cout << "fliegen"; }
};

Tier * pTier = NULL;
Vogel * pVogel = new Vogel;
pTier = pVogel;
pTier->gibLaut();        //--- => Datentyp des Zeigers bestimmt Methode
pTier->bewege();         //fliegen => Datentyp des Objekts bestimmt Methode
```

- Aufruf direkt über das Objekt: Es gilt Implementierung der Klasse, zu welcher das Objekt gehört.
- Aufruf über Zeiger oder Referenz: Es gilt Implementierung der Klasse des Objekts, auf das der Zeiger zeigt, bzw. die Referenz sich bezieht.

2.9.1. Regeln

- Nicht virtuelle Methoden einer Basisklasse sollten in Subklassen nicht überschrieben werden.
- Soll eine Methode in Subklassen überschrieben werden, so macht es Sinn, die entsprechenden Methoden bereits in der Basisklasse als virtuell zu kennzeichnen.
- Wenn in einer Klasse virtuelle Methoden vorkommen, muss auch der Destruktor virtuell deklariert sein.

2.10. Abstrakte Klassen / Pure Virtual

Abstrakte Klassen können NIE direkt als Objekt instantiert werden. C++ kennt kein Schlüsselwort, um abstrakte Klassen direkt kennzeichnen zu können. Eine Klasse wird allerdings als abstrakt betrachtet, wenn in der Klasse mindestens eine REIN VIRTUELLE METHODE enthalten ist.

```
virtual void foo() = 0;
```

Rein virtuelle Methoden haben keine Implementierung in der Basisklasse. Man wird also gezwungen, die Methoden in Subklassen zu implementieren.

2.11. Interfaces

Es existiert in C++ kein Schlüsselwort für Interfaces. Dies kann jedoch auch anders bewerkstelligt werden, indem einfach alle Methoden als „rein virtuell“ deklariert werden.

2.12. PIMPL Idiom

Das Klassenkonzept in C++ hat auch Nachteile. So müssen beispielsweise alle Benutzer einer Klasse ihren Code neu kompilieren, falls etwas an der Klassendefinition geändert wird. Als Lösung für dieses Problem realisieren wir so etwas wie einen Wrapper.

- Die Wrapperklasse verwaltet die versteckte Klasse über eine Pointervariable.
- Die Wrapperklasse bietet alle public-Methoden der versteckten Klasse
- Die versteckte Klasse (Implementierung) leistet die eigentliche Arbeit

Wrapper.cpp
```
class Implementierung;   //Vorwärtsdeklaration
class Wrapper {
 public:
       Wrapper() { pImpl = new Implementierung; }
       ~Wrapper() { delete pImpl; }
       void foo() { pImpl->foo(); }
 private:
       Implementierung * pImpl;
};
```
Implementierung.cpp
```
class Implementierung {
 public:
       void foo() { cout << „Beispiel" << endl; }
};
```

2.12.1. Anwendungen des PIMPL-Idioms

- In grossen Projekten kann die Compile-Time reduziert werden, indem haeufig geaenderte Teile ueber eine Wrapper-Klasse verwendet werden.
- Wrapper-Klassen koennen eingesetzt werden, um die genaue Implementation z.B. in einer Class-Library zu verstecken. (Information Hiding)
- Wrapper-Klassen koennen verwendet werden, um Schnittstellen-Informationen/Verwendung anzupassen/zu kontrollieren.

2.12.2. Variante 2

- Die versteckte Klasse wird direkt im .cpp-File der Wrapperklasse definiert und implementiert. Dadurch bleibt Sie auf jeden Fall verborgen, denn so kann auch nicht zufällig das Headerfile der versteckten Klasse eingebunden werden.

3. Funktionen

In C++ gibt es im Gegensatz zu Java Funktionen, welche unabhängig von Klassen sind.

3.1. Deklaration / Definition

Funktionen können benutzt werden, sobald Sie deklariert sind. Diese Funktion muss natürlich an einem anderen Ort irgendwann auch noch definiert werden.

```
int incVal(int);               //Deklaration, Parametername optional
int decVal(int intVal);        //Deklaration, Parametername gesetzt

int incVal(int intVal) { return ++intVal; }      //Definition
int decVal(int intVal) { return --intVal; }      //Definition
```

3.2. Funktionen überladen

Funktionen mit gleichen Funktionsnamen aber unterschiedlicher Parameterliste können definiert werden. Der Rückgabetyp spielt bei der Unterscheidung, wie auch in Java, jedoch keine Rolle, da dieser ja ignoriert werden kann.

```
int    incVal(int);
int    incVal(double);
double incVal(double);   //Wird nicht funktionieren!
```

3.3. Fallstrick mit Funktionen

Alles, was irgendwie als Funktionsdeklaration interpretiert werden kann, wird vom Compiler als Funktion angesehen! Klammern zeichnen Funktionen aus!

```
/*
* Compiler vermutet eine Funktion v die einen Vector<int> zurück gibt und
* als Parameter zwei Funktionen übernimmt
*/
vector<int> v(istream_iterator<int>(cin),istream_iterator<int>());
```

3.4. Funktionen als Parameter

In C++ können Funktionen auch als Parameter von Funktionen verwendet werden. Die Funktion wird dabei zum Objekt, das per Wert angegeben wird.

Nachfolgende Funktion übernimmt einen Integer-Wert und eine Funktion (*func), welche einen int-Wert zurückliefert und einen int-Wert als Parameter verlangt.

```
void doSomething(int intVal, int (*func)(int)) {
 func(i);          //Rufe die übergebene Funktion auf.
 (*func)(i);       //Alternative schreibweise
}

int funcName(int intParam) {
 //Funktion macht irgendetwas
}

doSomething(10, funcName);     //Aufrufen der Funktion, übergebe funcName
```

4. Datentypen

4.1. Allgemeines

Einige Beispiele für erlaubte Zuweisungen.

```
char blank = 32.0;            char a = ‚A‘;                int okt = 0100;
int hex = 0x100               unsigned int u = 10U;       long l = 10L;
unsigned long ul = 10UL;
```

4.2. Ganze Zahlen

```
char, signed char, unsigned char, short int, unsigned short int, bool
⇨ werden beim Rechnen autom. nach INT gewandelt.
```

Bei unsigned werden die darstellbaren Bits übernommen. Bei signed bleibt der Wert erhalten, passt der Ausgangswert nicht in den Zieltyp, ist das Ergebnis implmentierungsdefiniert.

4.3. Gleitkommazahlen

Floats werden automatisch nach double gewandelt. Am Besten nur mit double arbeiten, da i.d.R. schneller und Speicherbedarf nur bei sehr grossen Datenmengen relevant ist.

Beim Rechnen werden die Operanden immer auf den höchsten, verwendeten Operator promoviert. Beispiel: Float + Double -> Beide nach Double gewandelt!

4.4. Konvertierung zwischen „Ganze Zahlen“ <> „Gleitkommazahlen“

Die Konvertierung erfolgt automatisch. Kommazahlen werden abgeschnitten (GKZ > GZ). Falls die GKZ zu gross ist, ist das Verhalten undefiniert.

4.5. Bool

- bool ist kompatibel mit int, ist als ein Ganzzahlwert. Wir können sogar mit bool rechnen.
  - Promotion nach int, true=1, false=0
- 0 ist falsch, alles andere ist wahr. Folge: Man kann sogar fast alles als Bedingung verwenden.

4.6. Strings

- C++ kennt keinen eigenen Datentyp für Strings. Deshalb bietet die Standardbibliothek einen String-Typ zur Verfügung. Ein „String“ ist jedoch nicht einfach eine Klasse, sondern ein angewandtes Template!

```
typedef basic.string<…> strings;
```

4.6.1. Verwenden von Strings

```
#include <string>
std::string strBeispiel1 = „Hallo Welt!“;
std::string strBeispiel2 = strBeispiel1 + „ Ich bin der Thomas! ;-)“;
if (strBeispiel1 == strBeispiel2) { // Vergleichen von Strings }
cout << strBeispiel.length() << endl; //11
cout << strBeispiel[0] << endl; //H
```

4.6.2. substr()

- Mit der substr()-Funktion können Teile eines gegebenen Strings ausgeschnitten werden.

```
std::string strHello = "Hello World 1";
std::cout << strHello.substr(0,2) << std::endl;
//std::string(„Hello World 1“).substr(6,2)
```

4.7. Typkonstruktionen

```
int &x;                          //x ist Referenz auf int
int const &x;                    //x ist Referenz auf const int
char * x;                        //x ist Zeiger auf char
char const * x;                  //x ist Zeiger auf const char
char const * const x;            //x ist const Zeiger auf const char
double (*x)();                   //x ist Funktionszeiger auf Funktion
                                 //mit Rückgabewert double, 0-Parameter!
```

4.8. Explizite Konvertierung

4.8.1. Klassischer Cast

```
//(Typ) Ausdruck;
int intBsp = 10;
double dblBsp1 = (double)intBsp;     //Cast nach double
double dblBsp2 = double(intBsp);     //Defaultkonstruktor!
```

In moderenem C++ sollte der Cast-Operator nicht mehr verwendet werden.

4.8.2. Static_Cast

```
//static_cast<Typ>(Ausdruck)
double d = static_cast<double>(5);
```

Undefiniertes Verhalten oder ungewöhnliche Werte sind weiterhin möglich, falls Wert von „Ausdruck“ nicht in Wertbereich von „Typ“ passt!

4.8.3. Reinterpret_Cast

```
//reinterpret_cast<Typ>(Ausdruck)
```

Führt keine Typkonvertierung durch, sondern interpretiert das Bitmuster des „Ausdrucks“ als „Typ“.

4.8.4. Dynamic_Cast

```
//dynamic_cast<Typ>(Referenz/Pointer/Ausdruck)
```

Dient zum „downcasten“ von Objektzeigern entlang einer Klassenhierarchie. Funktioniert nur, wenn Klassen „virtual“ Member-Funktionen haben.

4.8.5. Const_Cast

```
//const_cast<Typ>(const Referenz/const Ausdruck)
```

Dient dazu, bei const-Referenzen und const-Pointern die „const“ zu entfernen. Ist nicht zu empfehlen!

4.9. POD – Plain Old Data

POD-Typen sind alle einfachen vordefinierten Datentypen und Zeiger. Zusätzlich Klassen, die keinen Konstruktor, keine virtuellen Memberfunktionen und alle Membervariablen public haben.

```
struct POD {
 char const *s;
 int x;
 void print();
};
void POD::print() { cout << s << „ = “ << x << endl; }

POD xy1 = {„xy1“,1};
POD xy2 = {„xy2“};       //Restliche Elemente mit Default-Werten
xy1.print();             //xy1 = 1
xy2.print();             //xy2 = 0;
```

5. Arrays

C++ erlaubt wie C und andere Sprachen Arrays. Im Gegensatz zu Java wird der Array-Zugriff bei C++ jedoch nicht überprüft. Deshalb sollte bei Applikationen immer std::vector verwendet werden.

5.1. Typische Probleme

- Out-of-Bounds Zugriff
- Kein dynamisches Wachstum (fixe Grösse)
- Grösse des Arrays geht bei Übergabe als Funktionsparameter „verloren“. Deshalb muss immer ein expliziter „Längenparameter“ zusätzlich zum Array übergeben werden.

5.2. Parameter der main-Funktion()

```
int main(int argc, char* argv[]) {}
//argv[0] = Programmname, argv[1] = 1. Argument, argv[2] = 2.Argument, …
```

5.3. Codebeispiele

```
int a[10] = {1, 1, 2, 3, 5, 8,};           //Rest mit 0 gefüllt
double d[] = {0.0, 1.1, 2.2, 3.3};         //Fix 4 Elemente
char const s[] = „Hello World“;            //Implizit 0-Byte als Endemarker!
ostream_iterator<int> out(cout,“, “);
copy(a,a + sizeof(a)/sizeof(a[0]),out);    //40Byte / 4 Byte = 10 Elemente
cout << s << endl;                         //Ausgabe als „String“ möglich
```

5.4. boost::array

Boost bietet eine Klasse für fixe Arrays. Ausserdem enthält sie, soweit möglich, die „üblichen“ Container-Methoden: begin(), end(), rbegin(), rend(), empty(), size().

```
include <boost/array.hpp>
boost::array<int,4> a = {1,2,3};     //Array von int, 4 Elemente
std::cout << a.at(0) << std::endl;   //Liefert 1
std::cout << a.at(5) << std::endl;   //Wirft range_error-Exception
```

6. Ein- / Ausgabe

6.1. std::cout

```
std::cout << „Ein wunderbarer Text“ << std::endl;
std::cout << ‚A’ + ‚B’ + 3.1415 << std::endl;
```

6.2. std::cerr / std::clog

```
std::cerr << „Fehlermeldungen bitte auf cerr!“ << std::endl;
std::clog << „Auch Fehlermeldunge, aber gepuffert!“ << std::endl;
```

6.3. std::cin

```
int intInteger;
double dblDouble;
std::cin >> intInteger;
std::cin >> intInteger >> dblDouble; //Zwei Werte einlesen
```

6.3.1. std::cin.get()

```
char c;
while(cin.get(c)) {
 ++chars;
}
```

6.4. std::getline()

```
std::string strBeispiel;
std::getline(std::cin,strBeispiel); //Liest ganze Zeile ein
```

6.5. std::endl

- Das Verhalten des Rechners im Zusammenhang mit Spezialzeichen hängt vom OS ab
- std::endl ist unabhängig vom Betriebssystem, ausserdem wird immer auch ein flush() ausgelöst

```
std::cout << „Es folgt ein Zeilenumbruch“ << std::endl;
```

6.6. Eingabe mittels Istream_Iterator

```
istream_iterator<int> itEof;          //Spezialobjekt, markiert das Ende
istream_iterator<int> itInp(cin);     //Einlesen von cin
vector<int> v(itInp,itEof);           //Vector füllen
```

6.7. Ausgabe mittels Ostream_Iterator & Copy

```
copy(menge.begin(),menge.end(),ostream_iterator<int>(cout,“, “));
```

6.8. String -> Integer

```
char chrZeichen = '0';
int intZeichen = 0;
for (unsigned int i=0; i < strNumber.length(); ++i) {
 chrZeichen = strNumber[i];
 intZeichen = atoi(&chrZeichen);
}
```

6.9. Zeichen einlesen mit Tokenizer (Boost)

```
#include "boost/tokenizer.hpp"
using namespace boost;
getline(cin,strInput);
//\x0D ist Zeilenumbruch, \x20 ist Leerzeichen
char_separator<char> chrSeperator("\x0D\x20");  //Trennzeichen
tokenizer<char_separator<char>> refTok(strInput,chrSeperator);

for(tokenizer<char_separator<char>>::iterator it=refTok.begin();
it != refTok.end(); ++it) {
 cout << *it; //Gib Werte aus
}
```

6.10.Lexical Cast (Boost)

```
#include "boost/lexical_cast.hpp"
try { cout << boost::lexical_cast<double>(strString) << " "; }
catch (bad_lexical_cast &) { cout << „Fehler beim Casten!";}
```

7. Input/Output-Streams

7.1. File-Streams

7.1.1. Konstruktoren

```
ofstream(const char* p, openmode m=out)    //Ausgabe nach File
ifstream(const char* p, openmode m=in)     //Lesen von File
```

7.1.2. Hilfsfunktionen

```
bool is_open()
void open(const char* p, openmode m=out/in)      //Impli. v. Konstruktor
void close()                                     //Impli. v. Destruktor
```

7.1.3. Stream-Zustände

Stream-Zustände können entweder über Methoden der Basisklasse, oder aber direkt über Status-Flags abgerufen werden.

```
//Methoden der Basisklasse
bool good()        //Alles ok
bool eof()         //Dateiende erreicht
bool fail()        //Fehler, Stream aber noch intakt
bool bad()         //Fehler, Stream nicht länger verwendbar
//Abfrage direkt über Flags
iostate rdstate()              //Stream zustand abfragen
void clear(iostate f=goodbit)  //Zustand setzen
void setstate(iostate f)       //Zusätzliche Zustände hinzufügen (ODER)
```

7.1.4. Openmode

Die einzelnen Modi können mit dem Oder-Operator (|) kombiniert werden.

```
ios_base::app      //An geöffnete Datei anhängen
ios_base::ate      //Zum Ende der Datei springen (beim Öffnen)
ios_base::binary   //Binärmode anstatt Textmode
ios_base::in       //Nur zum lesen öffnen
ios_base::out      //Nur zum Schreiben öffnen
ios_base::trunc    //Datei auf Länge 0 setzten
```

7.1.5. Beispiel Input / Output

```
#include <fstream>
#include <iostream>
#include <string>

int main() {
 std::ofstream theOutFile(„out.txt");
 if (!theOutFile) {
       //Fehlerausgabe
 } else {
       theOutFile << „Text ins File schreiben." << std::endl;
 }

 std::ifstream theInFile(„in.txt");
 std::string strGelesen;
 theInFile >> strGelesen;
}
```

7.2. String-Stream

Auch Strings können als Eingabe-/Ausgabe-Stream verwendet werden.

7.2.1. Beispiel

```
#include <sstream>
#include <iostream>
#include <string>

int main() {
 std::ostringstream theOutString;
 theOutString << „Irgendein Text..“ << std::endl;
 std::istringstream theInString(„Was auch immer.“);
 std::string strWort;
 while (theInString >> strWort) {
       std::cout << strWort << std::endl; //Leerzeichen trennt Wörter!
 }
}
```

7.3. Formatierung von Streams

Die Ein-/Ausgabe-Formatierung von Streams kann über Flags der Basisklasse „ios_base“ angepasst werden.

7.3.1. Formatzustände (ios_base::fmtflags)

Flag	Bedeutung	Flag	Bedeutung
skipws	Whitespace bei Inp überlesen	showbase	Präfix O (oct) bzw 0x (hex)
left	Linksbündig - Füllzeichen	showpoint	Nachkommastellen forcierern
right	Füllzeichen - rechtsbündig	showpos	explizit positives Vorzeichen
internal	Vorzeichen - Füllz. - Wert	uppercase	'E' und 'X' anstelle von 'e', 'x'
boolalpha	true/false symbolisch	adjustfield	Flags für Feldausrichtung
dec	dezimal	basefield	Flags für ganzahlige Darst.
hex	hexadezimal	floatfield	Flags für Gleitkommadarst.
oct	oktal	unitbuf	automatische Flushen
scientific	Gleitkomma Notation		
fixed	Festkomma Notation		

7.3.2. Flags Funktionen

```
fmtflags flags();              //Flags lesen
fmtflags flags(fmtflags f);    //Flags setzen
fmtflags setf(fmtflags f);     //Flag hinzufügen
void unsetf(fmtflags mask);    //Flags löschen
```

7.3.3. Beispiele

```
cout.setf(ios_base::hex,ios_base::basefield);    //Hex-Anzeige, Persistent
cout.setf(ios_base::fixed,ios_base::floatfield); //Persistent
```

7.4. Manipulatoren

Sogenannte Manipulatoren ermöglichen es, die Formatierung direkt als Bestandteil des Outputs vornehmen zu können, ohne die Flags modifizieren zu müssen.

- Bei Manipulatoren ohne Parameter werden keine Klammern verwendet
- Manipulatoren ohne Parameter sind in <iostream> definiert
- Manipulatoren mit Parameter sind in <iomanip> definiert

7.4.1. Standard-Manipulatoren

Manipulator	Bedeutung	Manipulator	Bedeutung
`boolalpha`	true/false symbolisch	`dec`	Dezimal
`noboolalpha`		`hex`	Hexadezimal
`showbase`	Ausgabepräfix O bzw 0x	`oct`	Oktal
`noshowbase`		`fixed`	Festkomma
`showpoint`	Nachkommastellen forciert	`scientific`	Wissenschaftlich
`noshowpoint`		`endl`	End-Of-Line ('\n') + flush()
`showpos`	Explizites Vorzeichen	`ends`	End-Of-String ('\0') + flush()
`noshowpos`		`flush`	flush
`skipws`	Whitespaces überlesen	`ws`	Whitespaces überlesen
`noskipws`		`resetioflags (ios_base::fmtflags f)`	Flags löschen
`uppercase`	X und E statt x und e	`setiosflags (ios_base::fmtflags f)`	Flags setzen
`nouppercase`		`setbase(int b)`	Integer zur Basis b
`internal`	Interne Ausrichtung	`setfill(int c)`	c als Füllzeichen
`left`	Linksbündig	`setprecision(int n)`	Genauigkeit n Stellen
`right`	Rechtsbündig	`setw(int n)`	Ausgabebreite n Zeichen

7.4.2. Beispiele

```
#include <iostream>
#include <iomanip>

int main() {
 double dblOutput = 1234.56789;
 std::cout << std::setw(10);         //Nächste Ausgabe 20 Zeichen breit
 std::cout << std::left;             //Linksbündig (Default: rechts)
 std::cout << std::setfill('_');     //Mit _ Füllen
 std::cout << dblOutput;             //1234.56___
 std::cout << std::fixed;            //Feste Nachkomma-Zahl
 std::cout << std::setprecision(4); //Stellen nach dem Komma
 std::cout << dblOutput;             //1234.5678
}
```

8. Vector

Vector ist der Standardcontainer für die Sammlung bei gleichartigen Elementen.

8.1. Codebeispiele

8.1.1. Allgemeines

```
#include <vector>
vector<int> V(10);       //Vector der Grösse 10 vom Typ int
vector.size();           //Anzahl Elemente (10)
V[0] = 15;               //Erstes Element auf den Wert 15 setzen
V.push_back(11);         //Neues Element anhängen, Wert 11
V.pop_back();            //Letztes Element löschen
V.clear();               //Alle Elemente löschen
```

8.1.2. Füllen eines Vectors mit Istream_Iterator

```
istream_iterator<int> itEof;         //Spezialobjekt, markiert das Ende
istream_iterator<int> itInp(cin);    //Einlesen von cin
vector<int> v(itInp,itEof);          //Vector füllen
```

8.1.3. Füllen eines Vectors mit Boost

```
#include <boost/assign.hpp>
using namespace boost::assign;
vector<int> v;
v += 31, 41, 59; //Vector mit 3 Elementen gefüllt
vector<int> v2 = list_of(31)(41)(59);
```

8.1.4. Füllen eines mehrdimensionalen Vektors mit Boost

```
#include <boost/assign.hpp>
using namespace boost::assign;
vector< vector<string> > vMulti;
vMulti += list_of("1")("2"), list_of("3")("4");
```

8.2. Grundfunktionen

8.2.1. size()

Anzahl Elemente des Vectors. Rückgabewert ist ein unsigned int.

8.2.2. push_back(value)

Element am Ende des Vectors anfügen. Die Grösse des Vektors wird automatisch angepasst.

8.2.3. pop_back()

Letztes Element wird gelöscht. Grösse wird automatisch angepasst.

8.2.4. clear()

Alle Elemente des Vectors werden gelöscht.

8.2.5. begin()

Gibt einen Iterator zurück, der auf den Anfang des Containers zeigt.

8.2.6. end()

Gibt einen Iterator zurück, der auf das erste Element HINTER dem Container zeigt.

9. Container

Neben Vector gibt es auch noch anderen Typen von Containern als vorgefertigte Standardklassen. Diese sollen nachfolgend betrachtet werden.

9.1. Set / Multiset

Standardklasse für Mengen von Elementen. Jedes Element kann typischerweise nur einmal vorkommen. Ausserdem ist ein Set automatisch sortiert. Ausserdem muss für die Elemente, welche im Set gespeichert werden sollen, der „<“-Operator definiert sein.

Der einzige Unterschied von einem Multiset zu einem Set ist, dass Ersteres (Multiset) Duplikate erlaubt.

9.1.1. Codebeispiel

```
#include <set>
istream_iterator<int> itEof;
istream_iterator<int> itInp(cin);
set<int> menge(itInp,itEof);         //Einlesen der Werte
copy(menge.begin(),menge.end(),ostream_iterator<int>(cout,", "));
if (menge.count(42)) {                    //Ergibt True, wenn 42 gefunden
 cout << „42 wurde eingegeben!“ << endl;
}
```

9.1.2. Spezielle Methoden

```
s.find(Element);         //Suche Element im Set, gibt einen Iterator zurück
s.insert(Element);       //Füge ein Element zum Set hinzu, kein push_back()
s.erase(Element);        //Entferne Element aus dem Set
```

9.2. Map / Multimap

Definiert eine Nachschlagetabelle für so genannte Key-Value Paare, auch bekannt als assoziative Arrays oder Dictionary. Natürlich darf jeder Key nur einmal vorkommen. Ausserdem ist die Map nach Keys sortiert. Dadurch ergibt sich ein effizienter Suchzugriff auf Values, wenn nach dem Key gesucht wird (O(log(n))!).

Bei der Multimap darf, wie auch schon beim Multiset, der Wert des Keys mehrmals vorkommen.

9.2.1. Iteratoren für Map

Da jedes Element einer Map aus einem Paar (Key / Value) besteht, gibt es dafür eine vordefinierte Datenstruktur namens „pair“.

```
//std::pair<Key,Value>
pair<int,string> refPair(42,"Thomas");
cout << refPair.first << " " << refPair.second;
```

Der Iterator einer map (map::iterator) verweist immer auf diese Paare. Nachfolgend ein Beispiel:

```
map<string,int>::iterator it = m.begin();
(*it).first;       //Zugriff auf Key, Alternative: it->first
(*it).second       //Zugriff auf Value, Alternative: it->second
```

9.2.2. Map füllen (klassisch)

```
map<string,int> m;
m[„Frodo“] = 21;
m[„Bilbo“] = 123;
m[„Gandalf“] = 502;
```

9.2.3. Map füllen (Boost!)

```
#include <map>
using namespace boost::assign;
map<string,double> preise;
insert(preise) („Aepfel“,2.95)(„Birnen“,3.95);
string strObst;
while (cin >> strObst) {
 cout << strObst << „ kostet „ << preise[strObst] << endl; }
```

9.2.4. Map Ausgeben per Schleife

```
map<char,int>::iterator itRun;
for (itRun = refMap.begin(); itRun != refMap.end(); ++itRun) {
 cout << itRun->first << "\t" << itRun->second << endl;
 //cout << (*itRun).first << "\t" << (*itRun).second << endl;
}
```

9.2.5. Mehrdimensionale Map

```
map<int, map<int,string> > refMap;
map<int, string> refMapInner;
refMapInner[0] = „Hallo“;
refMapInner[1] = „Welt“;
refMap[0] = refMapInner;
cout << refMap[0][0] << “ “ << refMap[0][1]; //Hallo Welt
```

10. Iteratoren

Iteratoren werden insbesondere dazu benutzt, um Containerelemente anzusprechen. Sie fungieren als Bindeglied zwischen Containern und Algorithmen. Das Ende eines Bereichs gehört nicht mehr dazu, siehe auch Beispiele bei den Algorithmen!

10.1. Beispiel: Elemente eines Vectors iterieren

```
//Klassisch
vector<int>::iterator it = v.begin();
while(it != v.end()) {
 cout << *it << endl;
 ++it;
}

//Modern
#include <algorithm>
#include <iostream>
copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\n"));
```

10.2. Operationen von Iteratoren

```
*it    //Zugriff auf das aktuelle Element, auf das it verweist
++it   //Zum nächsten Element springen (Inkrementieren)

//Noch einige Beispiele, Elemente des Vectors: 3,1,4,1,5
vector<int> iterator it;
*v.begin();                          //3
it = v.begin(); ++it; ++it; *it;     //4
it = v.begin(); it = it+3; *it;      //1
it = v.begin(); it[3];               //1, *(it+3)
it = v.begin(); *++it = 7;           //Zweites Element wird 7
```

10.3. Reverse Iteratoren

```
vector<int>::reverse_iterator it = v.rbegin(); //Wir fangen hinten an
```

10.4. Spezialiteratoren

Die Standardbibliothek definiert spezielle Iteratoren für die Nutzung der Algorithmen für I/O-Streams.

10.4.1. ostream_iterator

```
ostream_iterator<Typ>(ZielStream,Trennzeichen) //Beispiel: 6.7 oder 10.6.2
```

10.4.2. istream_iterator

Dient zum Einlesen von Elementen vom Typ <Typ>. Das Ende wird mittels eines Spezialobjektes definiert, das keinen Stream als Parameter übernimmt. Achtung: Bei diesem Iterator werden Whitespaces überlesen (Siehe auch:istreambuf_iterator).

```
istream_iterator<int> itEof;         //Spezialobjekt, markiert das Ende
istream_iterator<int> itInp(cin);    //Iterator auf cin
vector<int> v(itInp,itEof);          //Vector füllen (cin)
```

10.4.3. istreambuf_iterator

Funktioniert eigentlich genau gleich wie der normale istream_iterator, mit der Ausnahme, dass auch Whitespaces gelesen werden können.

```
istreambuf_iterator<char> itEof;          //Markiere Ende des Inputs
istreambuf_iterator<char> itInp(cin);     //Iterator auf cin
vector<char> v(itInp,itEof);              //Vector füllen (cin)
```

10.5. Iteratortypen

```
struct input_iterator_tag {};
Einmal iterieren, nur ein Zugriff (*it) auf jedes Element erlaubt. Beispiel
ist der „istream_iterator".
```

```
struct output_iterator_tag {};
Einmal iterieren, nur ein Zugriff (*it) auf jedes Element erlaubt. Beispiel
ist der „ostream_iterator".

struct forward_iterator_tag : public input_iterator_tag {};
In eine Richtung iterieren, aber mehrfacher Zugriff auf jedes Element
erlaubt.

struct bidirectional_iterator_tag : public_forward_iterator_tag {};
Erlaubt es, in beide Richtungen zu iterieren (++ / --), mehrfacher Zugriff
auf jedes Element erlaubt.

struct random_access_iterator_tag : public_bidrectional_iterator_tag {};
Erlaubt es, in beide Richtungen zu iterieren (++ / --). Zusätzlich
Indexzugriff ([1]) möglich. Mehrfacher Zugriff auf jedes Element möglich.
```

10.6.Eigene Iteratoren

Praktisch alle Standardalgorithmen arbeiten mit Ranges, die durch Iteratoren definiert werden. Aus diesem Grund kann man auch eigene Iteratoren schreiben. Zur Definition sollte man dabei auf das Basisklassentemplate std::iterator<tag, value_type> zurückgreifen.

10.6.1. Benötigte Operatoren

Je nach Iteratorkategorie müssen verschiedene Operatoren überladen werden.

```
OutputIterator
operator*()                          //Für linke Seite der Zuweisung
operator++()                         //Prefixoperator
operator++(int)                      //Postfixoperator

InputIterator
operator*()                          //Für Wertzugriff
operator++()                         //Prefixoperator
operator++(int)                      //Postfixoperator
operator==(InpIt const&) const       //Gleichsoperator
operator!=(InpIt const&) const       //Ungleichoperator

ForwardIterator
Wie InputIterator + OutputIterator (einige Restriktionen weniger)

BidirectionalIterator
Wie InputIterator + OutputIterator, zusätzlich jedoch:
operator-()

RandomAccessIterator
Wie BidirectionalIterator, zusätlich jedoch:
operator[](distance_type)
„Pointerarithmetik"
```

10.6.2. Beispiel eines InputIterators

```
#include <iterator>

class SimpleIterator : public std::iterator<std::input_iterator_tag, int> {
 private:
 int intCounter;
 public:
 SimpleIterator(int i) : counter(i) {}
 int operator*() const { return counter; }
 SimpleIterator& operator++() { ++counter; return *this; }  //Prefix
 SimpleIterator operator++(int) {                           //Postfix
       SimpleIterator result(*this);
       ++counter;
       return result;
 }
```

```
 bool operator==(SimpleIterator const &other) const {
       return this->counter == other.counter;
 }
 bool operator!=(SimpleIterator const &other) const {
       return !(*this == other);
 }
};

//Aufruf
SimpleIterator start(1);
SimpleIterator end(11);        //Geht bis 10!
ostream_iterator<int> output(cout, " ");
copy(start, end, output);      //1 2 3 4 5 6 7 8 9 10
transform(start, end, output, bind1st(multiplies<int>(), 3)); //3 6 9 …
transform(start, end, start, output, multiplies<int>()); //1 4 9 16 …
```

11. Algorithmen

11.1. Grundlegendes

- Algorithmen sind vorgefertigte Methoden, welche auf Container angewendet werden und viel benötigte Probleme erledigen.
- Algorithmen werden verwendet, um Operationen über alle Elemente eines Containers durchzuführen.
- Sie verlangen #include <algorithm>.
- Algorithmen verwenden als Parameter meist Iteratoren oder Funktionsobjekte.
- Funktionsobjekte sind im Headerfile <functional> definiert.

11.2. Eigenschaften

- Algorithmen sind mit Hilfe von Templates implementiert
- Algorithmen führen weder bei Eingabe noch bei Ausgabe Bereichsüberprüfungen durch. Bereichsfehler müssen deshalb auf andere Art verhindert werden

11.3. Gruppierungen

11.3.1. Nichtmodifizierende Algorithmen

```
for_each()          find()              find_if()           find_first_of()
adjacent_find()     count()             count_if()          mismatch()
equal()             search()            find_end()          search_n()
```

11.3.2. Modifizierende Algorithmen

```
transform()         copy()              copy_backwards()    swap()
iter_swap()         swap_ranges()       replace()           replace_if()
replace_copy()      replace_copy_if()   fill()              fill_n()
generate()          generate_n()        remove()            remove_if()
remove_copy()       remove_copy_if()    unique()            unique_copy()
reverse()           reverse_copy()      rotate()            rotate_copy()
random_shuffle()
```

11.3.3. Sortierende Algorithmen

```
sort()              stable_sort()       partial_sort()      partial_sort_copy()
nth_element()       lower_bound()       upper_bound()       equal_range()
binary_search()     merge()             inplace_merge()     partition()
stable_partition()
```

11.3.4. Mengenalgorithmen

```
includes()          set_union()         set_difference()    set_intersection()
```

11.3.5. Heap-Operationen

```
make_heap()         push_heap()         pop_heap()          sort_heap()
```

11.3.6. Permutationen

```
next_permutation()                      prev_permutation()
```

11.3.7. Minimum und Maximum

```
min()               max()               min_element()       max_element()
lexicographical_compare()
```

11.4. Copy

Als Ziel einer Kopieraktion kann alles, was durch einen Output-Iterator beschrieben wird, angegeben werden. Dazu gehören insbesondere der ostream_iterator und der back_inserter.

```
//Kopiere einen Bereich von Anfang bis Ende nach Ausgabe
//copy(Anfang,Ende,Ausgabe)
copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, ", ") );
copy(v.begin(), v.end(), std::back_inserter(v2) );
```

11.4.1. Copy_If

Aus unerklärlichen gründen fehlt der copy_if()-Algorithmus in der STL.

11.5.Find

Elemente suchen, liefert bei einem Treffer einen Iterator auf das gefundene Element, ansonsten einen Iterator auf das Ende des Bereichs.

```
//find(Anfang,Ende,Suchterm)
if (find(v.begin(),v.end(),42) != v.end()) {
 cout << „Die Zahl 42 wurde gefunden!“;
}
```

11.6.Sort

Sortiert Elemente im Bereich [Anfang,Ende)

```
//sort(Anfang,Ende)
sort(v.begin(),v.end());
```

11.6.1. Eigene Sortfunktion

```
bool compare(int intL, int intR) { return intL > intR; }
sort(v.begin(),v.end(),compare);
```

11.7.Count

Zählt, wie oft „Wert“ im Bereich [Anfang,Ende) vorkommt.

```
//count(Anfang,Ende,Wert)
count(v.begin(),v.end(),0); //Zähle alle 0en im Vektor
```

11.8.Distance

Zählt die Elemente, die zwischen Anfang und Ende liegen und liefert einen Integer zurück.

```
//distance(Anfang,Ende)
distance(v.begin(),v.end()) == v.size() //Muss natürlich identisch sein!
```

11.9.Accumulate

Summiert Werte im Bereich [Anfang,Ende) auf. Dritter Parameter ist der Startwert für die Summe.

```
//accumulate(Anfang,Ende,Startwert)
#include <numeric>
accumulate(v.begin(),v.end(),0)
```

11.10. For_Each

Dieser Algorithmus ruft für jedes Element im Bereich [Anfang,Ende) die Funktion im 3.Parameter auf. Liefert ausserdem diese Funktion / Functor am Ende als Rückgabewert.

```
//Funktion for_each(Anfang,Ende,Funktion)
void print(int i) { cout << i << " "; }          //Funktion festlegen
Function for_each(v.begin(),v.end(),print);      //Rufe „print()“ auf.
```

11.11. Generate_n

Erzeugt eine bestimmte Anzahl (Size) von Elementen mittels Funktion.

```
//generate_n(OutputIterator,Size,Funktion);
generate_n(back_inserter(v),10,rand);      //Erzeuge 10 Zufallszahlen
```

11.12. Generate

Erzeuge neue Elemente im Bereich [Anfang,Ende) mittels Funktion im 3.Parameter.

```
//generate(Anfang,Ende,Funktion);
generate(v.begin(),v.end(),rand);    //Erzeuge neue Zufallszahlen
```

12. Try.. Catch

C++ kennt wie Java den Try-Catch Mechanismus. In C++ kann jedes Objekt (bzw. auch jede Variable) als Fehlerobjekt geworfen und gefangen werden.

12.1. Vordefinierte Exceptions

bad_alloc	new	<new>
bad_cast	dynamic_cast	<typeinfo>
bad_typeid	typeid	<typeinfo>
bad_exception	Ausnahmespez.	<exception>
out_of_range	at(), Bitset[]	<stdexcept>
invalid_argument	Bitset Konstruktor	<stdexcept>
overflow_error	Bitset	<stdexcept>

12.2. Exceptions ankündigen

- Jede Methode, bei der nichts Genaueres angegeben wird, kann grundsätzlich jede Art von Exception werfen!
- Es muss in der Signatur einer Methode deshalb explizit angegeben werden, dass keine Exception geworfen wird.

```
int getInt();                   //Kann alles werfen
int getInt() throw();           //Wirft keine Exceptions
int getInt() throw(ex1,ex2);    //Wirft ex1 o. ex2 (oder Ableitung davon)
```

12.3. Werfen, Fangen und Weiterwerfen

- Exceptions werden als Referenz gefangen

```
try {
 throw bad_cast(„Fehlerbeschreibung.“);
} catch (bad_cast& ex) {
 cout << ex.what() << endl;    //Fehlerbeschreibung ausgeben
} catch (exception& ex) {
 //unerwartete Fälle
 throw;                        //Weiter werfen der Exception
} catch (…) {
 //noch unerwartetere Fälle
}
```

12.4. Vorsicht

- Exceptions stören den normalen Ablauf.
- Automatisch angelegte Objekte innerhalb des aktuellen Blocks werden vom System beim Werfen einer Exception automatisch wieder freigegeben. Explizit reservierte Ressourcen bleiben aber reserviert.

13. Referenzen

13.1. Grundlegendes

- Eine Referenz ist ein alternativer Name für ein Objekt oder eine Variable.
- In C++ werden Referenzen verwendet, um bestehende Objekte / Variablen mit anderen Namen ansprechen zu können.
- Die Referenz ist nur ein Zweitname (Alias).
- Nicht initialisierte Referenzen sind NICHT erlaubt.
  - Dies führt dazu, dass eine Referenz immer initialisiert werden muss.

13.2. Beispiel

```
void f() {
 int   i = 1;
 int&  r1 = i;     //Initialisierung der Referenz, r1 zeigt auf i
 int&  r2;         //FALSCH, KEINE NULL REFRENZEN ERLAUBT!
}
```

13.3. Call by..

```
void fct(int arg1, int& arg2, const int& arg3)
```

13.3.1. Value (arg1)

Das Objekt wird kopiert (Vorsicht: Kopierkonstruktor), Änderungen ohne Wirkung nach aussen. Argumente mit einfachen Datentypen werden normalerweise so übergeben.

13.3.2. Reference (arg2)

Das Objekt hat nur einen anderen Namen, Änderungen sind nach aussen wirksam.

13.3.3. Const Reference (arg3)

Das Objekt hat nur einen anderen Namen, ist aber durch const geschützt. Keine Änderung zugelassen! Objekte als Argumente werden normalerweise so übergeben.

13.4. Return by..

```
int    fct(int i)  { int a=i; return a; }
int&   fct(int& i) { i=5; return i; }
```

13.4.1. Value (Beispiel 1)

Das Rückgabeobjekt wird kopiert, gefahrlos.

13.4.2. Reference (Beispiel 2)

Die Funktion gibt nur einen Namen bekannt, mit dem auf das Objekt zugegriffen werden kann. Achtung: Nur als Argument übergebene Argumente sollten als Referenzen zurückgeben werden. Andernfalls zeigt die Referenz nach der Funktion auf einen ungültigen Bereich!

13.5. Const

```
int getValue() const;    //Ändert keine Werte im Objekt, liest nur!
```

Bei konstanten Objekten (Parameterübergabe) dürfen nur diejenigen Methoden verwendet werden, die KEINE Objektdaten ändern. Solche Methoden müssen mit „const“ gekennzeichnet werden.

14. Pointer

Da Referenzen nicht auf „null“ zeigen können, brauchen wir noch ein anderes Sprachelement: Zeiger.

14.1. Syntax

14.1.1. Zeiger auf einfache Datentypen

```
void foo(int * intPtr) {
 cout << *intPtr << endl;
}
//Aufruf
int i = 42;
foo(&i);
```

14.1.2. Zeiger auf Objekte

```
void foo(SimpleClass * ptrClass) {
 if (ptrClass) {                    //Pointer degenerieren zu bool!
       (*ptrClass).print();
       ptrClass->print();           //Äquivalent zu (*ptrClass)
 }
}
```

14.1.3. This-Zeiger

```
void SimpleClass::print() {
 cout << this->argument << endl;    //this ist auch ein Pointer
}
```

14.2. Gefahren

- Pointer können 0 sein. Vor Derefenzierung unbedingt prüfen!
- Pointer können auf verschwundene Objekte zeigen.

14.3. Kanonische Klassen

Immer dann, wenn man Pointer in einer Klasse verwaltet oder selbst „this“ als Pointer irgendwo einträgt, muss man folgende Memberfunktionen implementieren, bzw. private deklarieren, um ihren automatischen Aufruf zu verhindern:

- Destruktor: Implementieren
- Kopierkonstruktor: Als „private“ deklarieren oder implementieren
- Zuweisungsoperator: Als „private“ deklarieren oder implementieren

```
~Klasse();                                  //Destruktor
private:
Klasse(Klasse const &);                     //Kopierkonstruktor
Klasse& operator=(Klasse const &);          //Zuweisungsoperator
```

14.3.1. boost::noncopyable

Bei der Verwendung von boost::noncopyable sind Kopieren und Kopier-Zuweisungen nicht erlaubt. Dabei muss die zu schützende Klasse einfach von boost::noncopyable erben.

```
#include <boost/utility.hpp>
class Beispiel : private boost::noncopyable {
 …
};
//Verhinder beispielsweise folgendes:
myClass a;
myClass b;
b = a;              //Fehler, Zuweisungsoperator nicht erlaubt
myClass c(a);       //Fehler, Kopierkonstruktor nicht erlaubt
```

14.4. Dynamischer Speicher

Standardmässig legt C++ Variablen auf dem Stack oder dem globalen Speicher an und entsorgt (beim Verlassen des Blocks / Programms) diese auch automatisch wieder. Bei der Erstellung von dynamischen Elementen auf dem Heap müssen diese aber auch wieder selber aufgeräumt werden!

14.4.1. Syntax

```
T * refHeap = new T;     //new erzeugt Pointervariablen
delete objekt;           //Ruft automatisch Destruktor auf
```

14.4.2. Beispiel

```
Person * ptrAdam = new Person(„Adam“);     //Erzeugen
ptrAdam->print();
delete ptrAdam;                            //Löschen
delete ptrAdam;                            //Undefiniertes Verhalten!
```

14.5.Pointer in Containern

Im Gegensatz zu den Java-Collections sind C++ Container value-basiert. Entfernt man ein Pointer-Element aus einem Container, wird aber das zugrunde liegende Objekt nicht mittels delete gelöscht!

14.5.1. boost::shared_ptr

Diese Boost-Klasse erlaubt sicheres Verwenden von mit new angelegten Objekten in Containern. Diese werden automatisch aufgeräumt, sobald nirgends mehr auf sie referenziert wird.

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/shared_ptr.hpp>

using namespace std;

struct Drug {
 Drug(string strN, int intP) : intPrice(intP),strName(strN) {}
 ~Drug() { cout << "Sold out " << strName << endl; }
 int intPrice;
 string strName;
};

typedef boost::shared_ptr<Drug> ptrDrug;

void print(ptrDrug const & refPtr) {
 cout << refPtr->strName << ": " << refPtr->intPrice << endl;
}

int main() {
 vector<ptrDrug> refVec;
 ptrDrug pDrug1( new Drug("Cocaine",100) );
 refVec.push_back(pDrug1);
 {
       //pDrug2 wird nach dem Block ungültig sein
       ptrDrug pDrug2( new Drug("Ecstasy",10) );
       refVec.push_back(pDrug2);
 }
 vector<ptrDrug>::iterator refIter;
 for (refIter=refVec.begin();refIter!=refVec.end();refIter++) {
       print(*refIter);  //Zugriff auf Iterator-Element
       cout << "Referenzierungen: " << (*refIter).use_count() << endl;
 }
 refVec.clear();
}

/*     Output:
*      Cocain: 100
*      Referenzierungen: 2
*      Ecstasy: 10
*      Referenzierungen: 1
*      Sold out Ecstasy
*      Sold out Cocain
*/
```

- Erklärung: Bei refVec.clear() sind alle Referenzierungen auf pDrug2 weg. Das Heap-Objekt wird also automatisch aufgeräumt. Auf pDrug1 existiert jedoch noch eine Referenzierung. Dieses Heap-Objekt wird erst am Programmende aufgeräumt.
- use_count() gibt die Anzahl Referenzierungen auf den aktuellen shared_ptr zurück
- reset() erlaubt es eine Referenzierung zu löschen. (pDrug1.reset()).

14.5.2. RAII-Pattern

Um Ressource-Leaks zu vermeiden, sollte möglichst jedes mit „new“ angelegte Objekt in einem shared_ptr gekapselt werden. Dadurch fallen natürlich jegliche delete-Anweisungen weg, da die dynamischen Objekte in den shared_ptr automatisch gelöscht werden, falls keine Referenzierungen mehr vorhanden sind.

14.6.STD-Pointer

14.6.1. std::auto_ptr<T>

- Kann nicht in Containern gespeichert werden.
- Kopiert man auto_ptr, so besitzt die Kopie den ursprünglichen Pointer und der kopiert auto_ptr ist leer (0-Pointer).

```
#include <boost/auto_ptr.hpp>
boost::auto_ptr<int> zuerstLeer;
boost::auto_ptr<int> pint(new int);
*pint = 42;
zuerstLeer = pint; //pint ist nun 0, zuerstLeer enthält den Zeiger!
```

14.7.Boost-Pointer

Neben dem boost::shared_ptr<T> existieren auch andere Pointer-Typen.

14.7.1. boost:scoped_ptr<T>

- Kann nur mit new initialisiert werden
- Kopien sind nicht erlaubt.
- Beim Verlassen des Scopes ({…}) wird der gespeicherte Pointer mittels delete gelöscht

```
#include <boost/scoped_ptr.hpp>
{
 boost::scoped_ptr<int> pint(new int);
 *pint = 42;
}
//Speicher wird hier wieder freigegeben.
```

14.7.2. boost::weak_ptr<T>

- Verwendet intern einen shared_ptr. Zur Benutzung muss man deshalb zu jedem weak_ptr zuerst einen shared_ptr erzeugen.

14.8.Tabelle Pointer-Typen

	Memory-Leaks?	Dangling-Pointers?	0-Pointer?
T*	Leaks können vorkommen.	Kann passieren	Kann 0 sein, abfragbar
T&	Im Normalfall nicht möglich.	Nicht möglich.	Kann nicht 0 sein.
auto_ptr	Keine Zyklen möglich.	Nicht möglich.	Kann 0 sein, abfragbar
shared_ptr	Ohne Zyklen keine Leaks.	Nicht möglich.	Kann 0 sein, abfragbar
scoped_ptr	Keine Zyklen möglich.	Nicht möglich.	Kann 0 sein, abfragbar
weak_ptr	(irrelevant, läuft mit shared_ptr)	Nicht möglich. (Optimal!)	Kann 0 sein, abfragbar

15. Enum

15.1. Beispiel

```
//Definition
enum enuFarben {BLACK=0,RED,GREEN,BLUE=3};
enum enuFarben {BLACK=0,RED=1,GREEN=2,BLUE=3};   //Identisch mit Zeile 1

//Funktionsheader mit einem Enum als Parameter
fct(enuFarben enuFarbe);

//Aufruf der Funktion mit einem Enum als Parameter
fct(enuFarben.BLACK);
fct(Klassenname.BLACK);  //Falls Enum in einer Klasse ist
```

16. Operatoren überladen

C++ erlaubt das Überladen der eingebauten Operatoren. Es gibt nur einige wenige Ausnahmen, bei denen Operatoren NICHT überladen werden können.

Diese Ausnahmen sind: | **.** | **.*** | **?:** | **::** | **size_of()** | **type_of()**

Achtung: Ausserdem können die Operatoren für eingebauten Typen nicht neu definiert werden!

16.1. Implementierung

Eigentlich ist ein selbst definierter Operator nichts anderes als eine normale Funktion mit einer etwas anderen Syntax.

```
a + b  ->    operator+(a,b)    |     z = y ->    z.operator=(y)
a += b ->    a.operator+=(b)   |
```

16.1.1. Als Member-Funktion

Bei einer Memberfunktion wird „this“ implizit als erstes Argument verwendet! Ausserdem sollte die Funktion wie immer als const deklariert werden, wenn das Objekt nicht verändert wird.

Eine Memberfunktion sollte immer dann verwendet werden, wenn direkter Zugriff auf die interne Repräsentation einer Datenstruktur notwendig ist.

```
//Bei allen Member-Funktionen ist this implizit erstes Argument!
Type& operator+=(const Type& rRight);      //Const-Ref ist Normalfall
Type  operator-=(const Type& rRight);      //Neues Objekt zurückgeben
Type  operator*=(Type rRight);             //Param als Value (Kopie)
Type& operator/=(Type rRight);             //Refrenz möglich, da this 1.Par
```

16.1.2. Als globale (freie) Funktion

Alle Argumente müssen angegeben werden! Dabei muss mindestens ein Parameter mit einem selbstdefinerten Typen vorhanden sein.

Eine freie Funktion sollte dann verwendet werden, wenn kein direkter Zugriff auf die interne Struktur benötigt wird oder wenn eines der beiden Argumente kein selbst definierter Typ ist.

```
//Bei freie Funktionen müssen beide Seiten des Operators übergeben werden!
//Es werden keine Referenzen zurückgegeben, immer neues Objekt!
Type operator+(const Type& rLeft, const Type& rRight);
Type operator-(Type rLeft, Type rRight);              //Param als Kopie
Type operator*(Type rLeft, const Type& rRight);
```

16.1.3. I/O Operatoren als freie Funktionen

I/O-Operatoren sollten immer als freie Funktion implementiert werden. Möglichkeit über „friend“ ist schlechtes Design! Bei den I/O-Operatoren tritt ein Sonderfall auf: Sie geben eine Referenz zurück, da I/O-Streams nicht kopiert werden können.

```
//Output, Zahl als const-Ref da Objekt nur gelesen wird
ostream& operator<<(ostream& osOut, const Type& refObj);
//Input, Zahl als Ref, da Objekt geändert wird
istream& operator>>(istream& isIn, Type& refObj);
```

16.1.4. Vergleichsoperator

Ein Vergleichsoperator muss bool zurückliefern und ausserdem als „const“ deklariert sein. Vergleiche ändern schliesslich nichts in der Datenstruktur, sondern vergleichen nur.

```
bool operator<(const Type &refObj) const;
bool operator>(const Type &refObj) const;
```

16.1.5. Spezialfälle ++ und --

Die Post- und Prefix-Varianten werden durch virtuelles zusätzliches Argument beim Postfix unterschieden. Anzumerken ist hierbei auch, dass die Rückgabewerte fakultativ sind, also auch weggelassen werden können.

```
//als Memberfunktionen
klasse& operator++();    //Prefix
```

```
klasse  operator++(int); //Postfix

//Als freie Funktion
klasse& operator++(klasse& refKlasse);          //Prefix
klasse  operator++(klasse& refKlasse,int);      //Postfix
```

16.1.6. Cast-Operator

```
//operator typ() const;
operator double() const;
```

16.2.Beispiel

```
//Rational.h ----------------------------------------------------------------
#include <iosfwd>        //Macht iostream bekannt.
class Rational {
 public:
       void print(ostream& osOut) const;
       Rational& operator+=(const Rational& rRight);
       Rational operator+(const Rational& refRight);
       bool operator<(const Rational& refNumber) const;
 private:
       long lngZaehler;
       long lngNenner;
};
ostream& operator<<(ostream& osOut, const Rational& refZahl);

//Rational.cpp --------------------------------------------------------------
#include „Rational.h"
#include <iostream>      //Bindet iostream komplett ein

Rational& Rational::operator+=(const Rational& rRight) {
 //this ist implizit erster Parameter!
 lngZaehler = (lngZaehler*rRight.lngNenner)+(rRight.lngZaehler*lngNenner);
 lngNenner *= rRight.lngNenner;
 return *this;           //Bei Zuweisung aktuelles Objekt liefern!
}

Rational Rational::operator+(const Rational& refRight) {
 //this ist implizit erster Parameter!
 Rational refResult(refRight); //Hilfsvariable für Rückgabewert
 return refResult += *this;
}

bool Rational::operator<(const Rational& refNum) const {
 //this ist implizit erster Parameter
 return (lngZaehler/lngNenner) < (refNum.lngZaehler/refNum.lngNenner);
}

void Rational::print(ostream& osOut) const {
 //this ist implizit erster Parameter
 osOut << lngZaehler << „/" << lngNenner << endl;
}

ostream& operator<<(ostream& osOut, const Rational& refZahl) {
 refZahl.print(osOut);
 return osOut;
}
```

16.3.Faustregeln

- Member-Funktionen sollten „so const wie möglich" sein.
- Parameter als const-ref oder per value (Vorsicht bei grossen Datentypen!), sofern möglich. Nur dann per Referenz, wenn Parameterobjekt geändert werden muss (Istream!).
- Return per value, in Ausnahmefällen (Zuweisungsoperatoren) anders.

16.4.Operatoren mit Boost überladen

Wir haben beim Operator Overloading gelernt, dass man beispielsweise für die Addition nur den operator+= ausprogrammieren muss und den operator+ auf seine Implementierung abstützen kann. Boost bietet für diesen schematischen Code einen Automatismus.

- Man muss nur noch operator+= implementieren und operator+ wird automatisch erzeugt. Dasselbe gilt natürlich auch für die meisten anderen Operatoren.
- Man erbt von einem Template, dem man die eigene Klasse als Parameter mitgibt. Hierbei reicht die private Vererbung.

16.4.1. Möglichkeiten der Vererbung

```
class Example : boost::addable<Example> {};            // += erzeugt +
class Example : boost::subtractable<Example> {};       // -= erzeugt -
class Example : boost::multipliable<Example> {};       // *= erzeugt *
class Example : boost::dividable<Example> {};          // /= erzeugt /
class Example : boost::arithmetic<Example> {};         // Alle 4 zusammen

class Example : boost::less_than_comparable<Example>{}; // < erz. <= > >=
class Example : boost::equivalent<Example> {};           // < erz. == !=
class Example : boost::equality_comparable<Example>{};   // == erz. !=
```

16.4.2. Beispiel

```
#include <boost/operators.hpp>
class Beispiel : boost::arithmetic<Beispiel> {
 Beispiel& operator+= (const Beispiel&);
};
…
cout << bsp1 + bsp2 << endl;         //Funktioniert, da + automatisch impl.
```

16.4.3. Beispiel 2

```
#include <boost/operators.hpp>
class Beispiel : boost::addable<Beispiel>, boost::dividable<Beispiel> {…};

//Nachfolgendes ist äquivalent
class Beispiel : boost::addable<Rational, boost::dividable<Rational> > {…};
```

17. Funktoren – Überladen von ()

Funktoren sind Klassen, die im Wesentlichen nur den operator() = Call-Operator implementiert haben. Häufig werden Sie als struct{} implementiert, können jedoch auch als class erstellt werden.

Hierdurch können Objekte wie Funktionen benutzt werden, auch „Funktoren" genannt. Solche Funktoren-Klassen kapseln üblicherweise Algorithmen und sind meistens zustandslos, haben also keinen privaten Bereich.

Ein Vorteil von Funktoren gegenüber Funktionen / Zeigern ist dass sie als Objekte einen Zustand haben können, man kann sich also Dinge merken.

17.1.0-stelliger Funktor

0-Stellige Funktoren machen nur dann Sinn, wenn diese einen Seiteneffekt haben. Normalerweise besitzen Sie deshalb ein „Gedächtnis" (Zählvariable, ...).

```
Type operator()()

struct NullStellig {
 typedef int result_type;
 result_type operator()() const { return 42; }
};
int i = NullStellig()(); //1.Klammernpaar: Neues Objekt
                         //2.Klammernpaar: Funktor-Aufruf
```

17.2.1-stelliger Funktor

```
Type operator()(ParamType)
std::unary_function<ParamType,ReturnType> //unary_function -> std::bind1st

struct EinStellig : public std::unary_function<int,long> {
 long operator()(int n) const { return n*n; }
};
```

17.2.1. Prädikate (_if-Suffix)

Einige Algorithmen verwenden so genannte Prädikate. Ein Prädikat ist eine Funktion mit Rückgabetype bool. Gewöhnlicherweise enden diese Algorithmen auf „_if". Die normalen Algorithmen benutzt Gleichheit mit einem Wert, der als Parameter mitgegeben wird. Bei den IF-Varianten hingegen wird ein unäre Funktion oder ein unärer Funktor als Parameterübergeben.

17.2.2. Beispiel: find_if()

```
bool isEven(intNumber) { return 0 == (intNumber % 2); }      //Funktion

struct isNotEven : public std::unary_function<int,bool> {    //Funktor
 bool operator()(int n) const { return (n % 2 == 0) ? false : true; }
};

vector<int>::iterator found = find_if(v.begin(),v.end(), isEven);
vector<int>::iterator found = find_if(v.begin(),v.end(), isNotEven());
```

17.2.3. For_Each-Beispiel

```
struct mean : public std::unary_function<int,void> {
 int intCount, intSum;
 mean() : intCount(0), intSum(0) {}
 void operator()(int intI) { ++intCount; intSum += intI; }
 int meanv() { return intSum / intCount; }
};

vector<int> v = {1,3,5};
//For_Each liefert den Functor als Ergebnis zurück!
int intMeanV = for_each(v.begin(),v.end(),mean()).meanv();
//intMeanV enthält 3 -> 1+3+5 / (1+1+1) = 3
```

17.3.2-stelliger Funktor

```
Type operator()(ParamType1, ParamType2)
std::binary_function<ParamType1,ParamType2,ReturnType>
```

17.3.1. Beispiel: transform()

```
struct multi : public std::binary_function<int,int,long> {
 long operator()(int n1, int n2) const { return n1*n2; }
};
ostream_iterator<int> out(cout,", ");
transform(v1.begin(),v1.end(),v2.begin(),out,multi());
```

17.3.2. Beispiel: Set sortieren / Vergleichsfunktor

Einem Set kann bei der Erzeugung als zweiter Parameter ein Funktor übergeben werden.

```
struct intcompare:public std::binary_function<int,int,bool> {
 bool operator()(int l, int r) { return l > r; }
};
set<int,intcompare> s;
```

17.4.Vordefinierte Funktoren

```
//Binäre arithmetische und logische Funktoren
plus(),  minus(),  divides(),  multiplies(),  modulus(),  logical_and(),
logical_or()
//Unäre arithmetische und logische Funktoren
negate(), logical_not()
//Binäre Vergleichsprädikate
less(),
 less_equal(), equal_to(), greater_equal(), greater(), not_equal()
```

17.5.Binder

Binder erlauben es, Argumente auf fixe Werte zu setzen. Die STL bietet dafür zwar grundlegende Möglichkeiten, es wird jedoch empfohlen, für diese Zwecke die Boost-Lib zu verwenden.

17.5.1. Boost::Bind

Der Header <boost/bind.hpp> erlaubt es, Argumente auf fixe Werte festzusetzen. Solche kombinierten Funktoren können dann den Algorithmen als Parameter mitgegeben werden.

17.5.2. Beispiel: find_if()

```
vector<int>::iterator itFound =
 find_if(v.begin(),v.end(),boost::bind(less<int>(),_1,42);
```

In diesem Beispiel wird der Funktor less() (Vergleichsfunktor) verwendet. Als 1.Parameter für less wird „_1“ gesetzt, welches dem Element aus dem Vektor entspricht. Der 2.Wert für den Vergleich wird fix auf 42 gebunden.

18. Templates

Templates erlauben generisches Programmieren, mein braucht Code nur einmal zu schreiben und spart sich ineffiziente Copy-Paste-Arbeit. Dennoch kann die Template-Programmierung aus folgenden Gründen ziemlich schwierig sein:

- Syntax mit spitzen Klammern relativ schwer lesbar
- Bisher waren Compiler eher schlecht dabei, Fehler bei Template Benutzung / Definition zu melden
- Viele Compiler hatten (und haben noch) Schwierigkeiten bei der Instanziierung. Das „export"-Keyword wird meist noch nicht unterstützt, deshalb müssen die Definitionen im Header-File stehen.
- Teilweise existieren unnötigte Einschränkungen aus historischen Gründen.

18.1. Funktionstemplate

- Funktionstemplates definieren eine Familie von Funktionen mit unterschiedlichen Argument-Typen.
- Template-Funktionen können überladen werden
- Die meisten Compiler können Template-Funktionen nur korrekt instaiieren, wenn sie „inline" definiert sind. „Inline" wird benötigt um One Definition Rule bei im Header implementierten Funktionen einzuhalten, für „kurze" Funktionen wird der Compiler angewiesen, die Funktionsimplementierung effizient an der Aufrufstelle „einzufügen".

18.1.1. Beispiel

```
//MyMin.h
template <typename T> inline T const & min(T const &a, T const &b) {
 return (a < b) ? a : b;
}

//Main.cpp
int main() {
 cout << min(42,88) << endl;              //42

 string s1(„Hallo"),s2(„Welt");
 cout << min(s1,s2) << endl;              //Hallo

 cout << min<String>(„Hallo", „Welt");    //Fehler ohne <String>!
}
```

18.1.2. Class statt typename

Anstelle des Keywords „typename" kann auch „class" verwendet werden. Es wird aber empfohlen, heute immer „typename" zu verwenden.

18.1.3. Concept / Anforderungen

Templates sollten immer mit mgölichst minimalen Anforderungen an die Template-Paramater gestaltet werden. In obigem Beispiel sind folgende Anforderungen gestellt:

- <-Operator muss definiert sein und boolean zurückliefern

18.1.4. Type-Deduktion anhand Parameter

Der C++ Compiler ermittelt automatisch anhand der Parametertypen die richtige Version der Template-Funktion. Bei Mehrdeutigkeiten muss man den gewünschten Parameter-Typ explizit angeben.

```
min<double>(3.14, 42);
```

18.2. Klassentemplate

Statt einzelnen Funktionen kann man auch ganze Klassen als Template definieren und so gestalten, dass sie für unterschiedliche Template-Parameter benutzt werden können.

- Im Gegensatz zu Funktionstemplates können die Template-Parameter nicht automatisch hergeleitet werden, sondern müssen immer explizit angegeben werden.

```
std::vector<int>
```

- Dafür sind bei Klassentemplates auch Default-Paramater möglich

```
template <T=int> class Example { … };
```

18.2.1. Beispiel

```
template <typename T> class Sack {
 public:
       Sack() {};
       void putInto(T const &item) { … };
       T getOut() { … };
 private:
       typedef std::vector <T> SackType;
       typedef typename SackType::size_type size_type;
       SackType theSack;
};
```

18.2.2. Typename

Bei Elementen innerhalb eines Templates, die von Template-Parametern direkt oder indirekt abhängen, weiss der Compiler ggf. nicht, was ein Name bedeuted. Typename dient dazu, dem Compiler mitzuteilen, dass ein vom Template-Parameter abhängiger Name ein Datentyp und nichts anderes ist.

18.2.3. Methoden ausserhalb Klassentemplates

Man kann die Methoden auch ausserhalb des Klassentemplates (jedoch immernoch im Header-File!) definieren. Dabei muss allerdings vor der Definition das Keyword „template“ du die Parameterdeklaration wiederholt werden.

```
//Headerfile
template <typename T> T Sack<T>::getOut() { … };
```

18.2.4. Export-Keyword

Sofern das Keyword „export“ unterstützt wird, kann man Template-Definitionen auch in einer separaten Übersetzungseinheit (Sourcefile) realisieren.

```
//Headerfile
template <typename T> class Sack {
 export T getOut();
};

//Sourcefile
export template <typename T> T Sack<T>::getOut() { … };
```

18.2.5. Funktionstemplates in Klassentemplates

Memberfunktionen eines Klassentemplates können Funktionstemplates sein. Die Methode ist dann sowohl von den Klassentemplate-Parametern, als auch von den Funktionstemplate-Parametern abhängig.

18.2.6. Werte als Template Parameter

Auch Werte sind als Templateparameter zulässig, solange es Ganzzahlkonstanten (true, false, 1, 2, ...) oder Pointer auf Objekte mit „external linkage“ sind.

18.2.7. Partitielle Spezialisierung

```
template <typename T>
class Heap <T *>
```

18.2.8. Explizite Spezialisierung

```
Template <>
class Heap <char *>
```

18.2.9. Explizite Instantiierung von Templates

Will man die Menge der möglichten Template-Instanzen bewusst begrenzen, so kann man Templates auch explizit im Header instaziieren. Damit verliert man aber jegliche automatische Instanziierung! Im folgendenen Beispiel sind nur noch Säcke von „Geschenken“ möglich.

```
//Headerfile
template class Sack<Geschenke>;
```

19. Traits

- Traits sind ein Konzept, um zur Compilezeit Zusatzinformationen zu Datentypen für die Nutzung in Template-Instanziierungen bereitzustellen.
- Meist sind traits bestimmte per Konvention definierte Klassen, die nur Typdefinitionen beinhalten und so kontextabhängige Datentypen auf vordefinierte, in templates verwendete Namen abbilden.
- Jeder STL Container definiert bestimmte Datentypen zur Nutzung anhand seiner konkreten Template Parameter und anhand von Implementierungsdetails. Beispiele hierfür sind: iterator, reverse_iterator, value_type und size_type.
- Traits definieren Typen, die dazu genutzt werden
  - Die effizientesten Algorithmus-Implementierungen zu nutzen (iterator_traits)
  - bind1st und bind2nd mit Funktoren zu nutzen (std::unary_function, std::binary_function)
  - Template-Meta-Programmierzung zu ermöglichen (z.B. result_type)

19.1. Grundlegendes Beispiel

```
template <typename T> struct mytraits {
 typedef T* pointer_type;
 typedef T& reference_type;
}
…
mytraits<int>::pointer_type pi;
```

20. Verschiedenes

20.1. Ausnahmefestigkeit

Eine Operation auf einem Objekt heisst ausnahmefest, wenn das Objekt in einem gültigen Zustand bleibt, auch wenn die Operation durch eine Ausnahme abgebrochen worden ist.

20.1.1. Invariante

- Elementwerte (und Objekte, auf die von Elementen verwiesen wird) werden als Zustand oder Wert eines Objektes bezeichnet
- Die Eigenschaft, die einen Zustand eines Objektes wohldefiniert (gültig) macht, wird Invariante genannt.
- Während Ausführung von Elementfunktionen wir die Invariante eines Objektes für gewöhnlich nicht eingehalten.
- Öffentliche Methoden sollten nur aufgerufen werden können, solange die Invariante gilt

20.1.2. Implementierungstechniken

- Try-Catch
- Anwenden von Ressourcenmanagement (RAII)
- „Gib nie Informationen aus der Hand, bevor du nicht einen Ersatz speichern kannst."
- „Bringe ein Objekt immer in einen gültigen Zustand, bevor Du eine Ausnahme wirst oder weiterwirfst."

20.2. Zusicherungen

- Grundlegende Zusicherung: Für alle Operationen eines Objektes gilt: die Invariante wird eingehalten, es gehen keine Ressourcen verloren.
- Strenge Zusicherung: Zusätzlich zur grundlegenden Zusicherung: Eine Operation wird entweder erfolgreich durchgeführt oder sie hat keinen Effekt
- Keine Ausnahme Zusicherung: Zusätzlich zur grundlegenden Zusicherung: Eine Operation sichert zu, dass sie keine Ausnahmen wirft.

20.2.1. Container der STL

Die Container der STL bieten eine grundlegende Zusicherung, solange sich der Anwender Code wie erwartet verhält:

- Benutzerdefinierte Operationen, die von den Containern benötigt werden, dürfen Container-Elemente nicht in undefinierten Zustand zurücklassen
- Destruktoren dürfen keine Exceptions werfen

20.2.2. Zusicherung für Container-Operationen

	vector	deque	list	map
clear()	keine *	keine *	keine	keine
erase()	keine *	keine *	keine	keine
1 Element einfügen	streng *	streng *	streng	streng
n Elemente einfügen	streng *	streng *	streng	grundlegend
merge()			keine *	
push_back()	streng	streng	streng	
push_front()		streng	streng	
pop_back()	keine	keine	keine	
pop_front()		keine	keine	
remove()			keine *	
remove_if()			keine *	
reverse()			keine	
splice()			keine	
swap()	keine	keine	keine	keine *
unique()			keine *	

20.3.BOOST_STATIC_ASSERT

Um portablen Code zu schreiben, macht man oft Annahmen über Eigenschaften der Plattform oder der Compile-Zeit-Verwendung. Es wäre jedoch gut, wenn man Bedingungen zur Laufzeit prüfen könnte. Hier kommt BOOST_STATIC_ASSERT ins Spiel.

```
#include <boost/static_assert.hpp>
…
BOOST_STATIC_ASSERT(Compile-Time-Expression)
//Compile-Time-Expression = Etwas, das bool liefert
```

20.3.1. Beispiel

```
#include <boost/static_assert.hpp>
#include <boost/type_traits.hpp>

//Überprüfung, ob 32bit-Maschine
BOOST_STATIC_ASSERT( sizeof(int) == 4 );

//Bereichüberprüfung (0..4)
BOOST_STATIC_ASSERT( i>0 && i<4);

//Templateparameter Ganzzahl? Braucht boost/type_traits.hpp
BOOST_STATIC_ASSERT( boost::is_integral<T>::value );
```

20.4.Boost::Function

Mit dem Klassentemplate boost::function können Funktionsobjekte gespeichert werden.

```
#include <boost/function.hpp>
…
bool foo(int i) {
 return i < 10;
}
…
boost::function<bool (int)> f;       //Func. liefert bool, 1.Parameter int
f = foo;
…
bool b;
b = foo(42);       //foo liefert für 42 false
b = f(42);         //ist identisch mit foo(42)!
```

20.5.Boost::Regex

Boost bietet mit boost::regex die Unterstütung von Regulären Ausdrücken in C++ Programmen. Im Gegensatz zu den vielen anderen boost-Bibliotheken, die rein als Header-Dateien realisiert sind, besteht boost::regex zusätzlich aus einer Bibliothek. Diese muss dem Linker angegeben werden.

20.5.1. Beispiel (regex_match)

```
#include <boost/regex.hpp>
boost::regex rationaleZahlen(„[0-9]+\\s*/\\s*[0-9]+\\s*“);
if (boost::regex_match(„42 / 42“, rationaleZahlen) {
 cout << „Übereinstimmung gefunden!“;
}
```

20.5.2. Beispiel (regex_search)

```
#include <boost/regex.hpp>
string strText = „Das ist ein wundervoller Text.“;
boost::regex pattern(„(ist)|(ein)“);
boost::smatch matches;
if (boost::regex_search(strText, matches, pattern)) {
 if (matches[0].matched) {           //Gesamter Ausdruck
       cout << matches[0].str();     //Gefundenes Pattern
       cout << matches.position(0);  //Position im Text
       cout << matches.prefix();     //Teil vor Match
       cout << matches.suffix();     //Teil nach Match
 }
```

```
 if (matches[1].matched) {           //1.Teilausdruck (ist)
       cout << matches[1].str();
       cout << matches.position(1);
 }
}
```