

Objektorientierte Programmierung mit C++

Version 1.2

Eine Einführung für C ProgrammiererInnen

1-1

Kursinhalt

- Teil 1 - Einführung
- Teil 2 - C++ Grundlagen
- Teil 3 - Klassen und Objekte
- Teil 4 - Template Klassen
- Teil 5 - Beziehungen zwischen Klassen
- Teil 6 - Exceptions
- Teil 7 - Standardbibliothek

1-2

Literatur

- **The C++ Programming Language (3rd Edition)**
Bjarne Stroustrup
ISBN 0-201-88954-4 (Addison Wesley 1997, 10th Printing)
- **C++ Der Einstieg für den C-Programmierer (mit Diskette)**
Sharam Hekmatpour
(Carl Hanser Verlag München Wien)
- **Programmieren in C++**
Einführung in den Sprachstandard C++ Version 3.0
Martin Schader, Stefan Kuhlins
(Springer Verlag)
- **Die C++ Standardbibliothek (mit CD-ROM)**
Nicolai Josuttis
ISBN: 3-8273-1023-7 (Addison-Wesley 1996)
- **Effective C++**
50 Specific Ways to Improve Your Programs and Designs
Scott Meyers
ISBN 0-201-56364-9 (Addison-Wesley 1992)

WWW-Ressourcen

- | | |
|---------------------------------|--|
| • ISO/IEC C++ Working Group | anubis.dkuug.dk/jtc1/sc22/wg21 |
| • Bjarne Stroustrup's Home Page | www.research.att.com/~bs |
| • C++ Libraries | www.trumphurst.com/cpplib |
| • C and C++ Style Guides | www.cs.umd.edu/users/cml/cstyle |
| • C/C++ Users Group | www.hal9k.com/cug |
| • Links zu Objects & Components | www.cetus-links.org |
| • C++ FAQ-Lite | www.parashift.com/c++-faq-lite |

Teil 1

Einführung

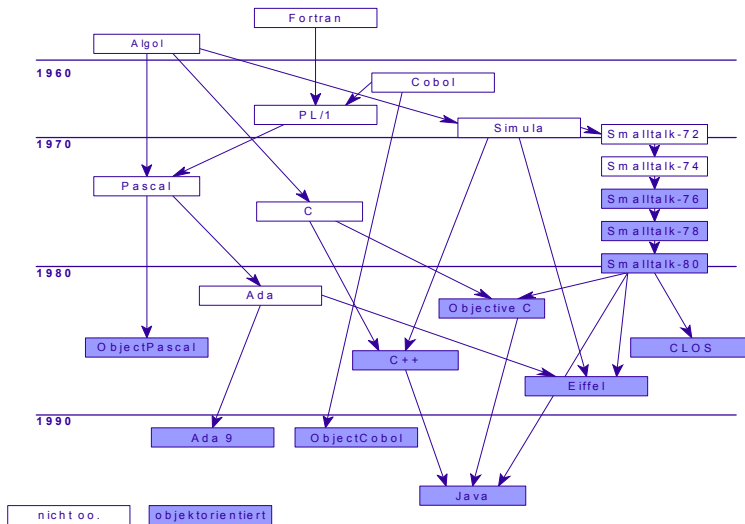
1-5

Teil 1 - Inhalt

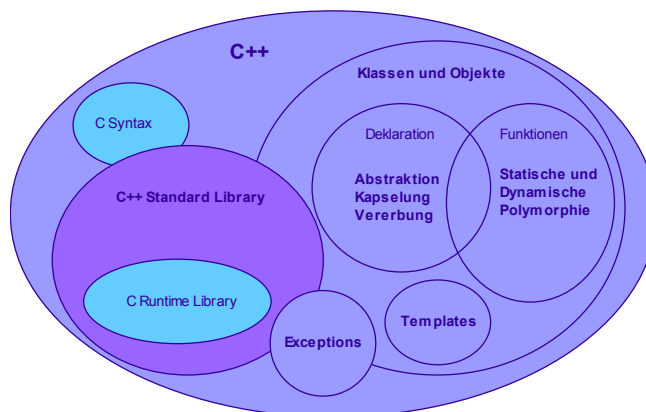
- Objektorientierte Konzepte (vgl. OOSE)
- Objektorientierte Konzepte von C++

1-6

Objektorientierte Programmiersprachen



Objektorientierte Konzepte von C++ I



Objektorientierte Konzepte von C++

II

- **Abstraktion**
 - Bildung von **Klassen** als benutzerdefinierter abstrakter Datentyp
 - Eine Klasse setzt sich zusammen aus:
 - **Data Members** (Attribute, Variablen, Properties)
 - **Function Members** (Operationen, Funktionen, Methoden)
 - **Instanziierung** erlaubt die Erzeugung von (Klassen-) **Objekten**
- **Kapselung**
 - Angabe der **Zugriffsrechte** für Funktionen und Daten
- **Modularisierung**
 - Aufteilung von Interface und Implementation einer Klasse auf separate Files
- **Typenprüfung**
 - Strikte Typenprüfung erweitert für Klassen (inklusive parametrisierte Klassen)



Skript: 1.5

1-9

Objektorientierte Konzepte von C++

III

- **Hierarchie**
 - **Klassenhierarchie** zur Beschreibung hierarchischer Beziehungen zwischen Klassen
 - **Vererbung** der Eigenschaften einer Basisklasse (Funktionen und Daten) an die davon abgeleiteten Klassen
 - **Aggregation** erlaubt die Verschachtelung von Klassen (lokale Klassen)
- **Polymorphismus**
 - **Statische (signaturgebundene) Polymorphie**: Unterscheidung von Funktionen mit gleichem Namen aber unterschiedlicher Signatur (Function Overloading)
 - **Dynamische (vererbungsgebundene) Polymorphie**: Unterscheidung von Funktionen mit gleichem Namen und Signatur aber in voneinander abgeleiteten Klassen (Function Overriding)
- **Generizität**
 - Generische Datentypen durch **Template-Klassen** implementierbar



Skript: 1.5

1-10

- **Exception Handling**
 - Mechanismus zur Behandlung von erwarteten und unerwarteten Laufzeitfehlern
- **Persistenz**
 - Durch C++ nicht direkt unterstützt (Verwendung einer Klassen-Bibliothek oder OODB)
- **Gleichzeitigkeit**
 - Durch C++ nicht direkt unterstützt (Verwendung einer Klassen-Bibliothek)

Teil 2

C++ Grundlagen

2-1

Teil 2 - Inhalt

- **Geschichte von C++**
- **Syntaktische Grundlagen von C++**
 - Lexikalische Konventionen
 - Datentypen
 - Definition und Deklaration von Variablen
 - Geltungsbereich und Lebensdauer
 - Ausdrücke und Typenumwandlung
 - Anweisungen
 - Funktionen
 - Modularisierung

2-2

Geschichte von C++

I



- 1979 Beginn der Entwicklung durch Bjarne Stroustrup für AT&T internen Gebrauch ("C with Classes")
- 1983 Erster Gebrauch von C++ ausserhalb von AT&T
- 1985 Erstes kommerzielles C++ Entwicklungsprodukt

C++

2-3

Geschichte von C++

II

- 1989 Bildung eines Komitees zur Standardisierung von C++ innerhalb der ANSI (American National Standards Institute)
- 1991 Definition des "C++ Standards" durch B. Stroustrup in "The C++ Programming Language, 2nd Edition"
- 1991 Zusammenführung des ANSI-Komitees und Gremien anderer Länder unter der ISO (International Standards Organization)
- 1995 Veröffentlichung des "Draft C++ Standard"
- 1996 Praktischer Abschluss der Standardisierung
- 1998 Internationaler ANSI/ISO C++ Standard wird formal als ISO/IEC 14882:1998(E) akzeptiert (ca. 770 Seiten)

C++

2-4

Syntaktische Grundlagen von C++

- ANSI C definiert ein Subset von C++
- C++ Syntax Definition als Erweiterung der C Syntax
- Beispiel - “Hello World” C++ Version:

```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
    string str;

    cout << "Please Enter Your Name: ";
    cin >> str;
    cout << "Hello, " << str << "!" << endl;
    return 0;
}
```



Lexikalische Konventionen

- Die erste Phase der C++ Compilation besteht in der Verarbeitung durch den Präprozessor (analog zu C) und liefert folgende Tokens:
 - Trennzeichen (Whitespace): space, tab, formfeed, newline (Windows vs. Unix!)
 - Kommentare (Comments): `/* multiline comment */` und `// end of line comment`
 - Bezeichner (Identifier): case-sensitive Variablennamen mit “beliebiger” Länge
 - Schlüsselwörter (Keywords): C erweitert um neue C++ Keywords (vgl. Kap. 2.1.4)
 - Literale (Literals): integer, float, character, string, boolean Konstanten
 - Operatoren (Operators): C Operatoren erweitert um `new` und `delete` sowie dem Scope Operator (`::`)



Datentypen

- Vordefinierte Datentypen
- Abgeleitete Datentypen
- Benutzerdefinierte Datentypen



Vordefinierte Datentypen (Built-In Types)

- Vordefinierte Datentypen sind integraler Bestandteil der Sprachdefinition und dienen als Basis für alle anderen Typen.
- C++ kennt folgende vordefinierten Datentypen:
 - Ganzzahlige Datentypen: `char`, `short`, `int`, `long` (vgl. `<limits.h>`)
jeweils mit `signed` oder `unsigned` Präfix
 - Gleitkomma-Datentypen: `float`, `double`, `long double` (vgl. `<float.h>`)
 - Void Datentyp: `void` oder `void*` (Pointer auf unbekannten Typ)
 - Multibyte Zeichen Datentyp: `wchar_t` (wide char type)
 - Boolescher Datentyp: `bool` Datentyp mit Wertebereich `true` oder `false`
- Der Wertebereich von Ganzzahl- und Gleitkomma-Datentypen ist Compiler und Plattform abhängig!



Boolscher Datentyp

- Mit ANSI/ISO C++ eingeführter, vordefinierter logischer Datentyp.
- Ein Boolscher Datentyp wird durch das Keyword `bool` definiert und kann die Literale `true` oder `false` als Werte annehmen.
- Datentyp `bool` ist mit `int` kompatibel (implizite Typkonversion):
 - `int(true)` – entspricht gemäss Definition 1
 - `int(false)` – entspricht gemäss Definition 0
 - `bool(i)` – ist `false` für `i=0` und `true` für alle `i!=0`
 - Null-Pointer werden implizit zu `false`, alle anderen zu `true` konvertiert

- Beispiele:

```
bool b1 = (a == b);    // b1 ist true falls a gleich b ist
bool b2 = 7;           // b2 wird mit true initialisiert
int i = true;          // i wird mit 1 initialisiert
```



Abgeleitete Datentypen

- Aus den vordefinierten Basisdatentypen lassen sich mittels Operatoren die folgenden weiteren Datentypen ableiten:
 - Feld (Array): `type array_name[number];`
 - Zeichenkette (String): `char string_name[number];` (char-Array mit `'\0'` Terminierung)
 - Zeiger (Pointer) : `type* pointer_name;` oder `type *pointer_name;`
 - Referenz (Reference): `type& reference_name;`



Referenz

I

- Eine Referenz beschreibt einen alternativen Namen (Alias) für eine bestehende Variable, Konstante oder Funktion.
- Referenzen müssen bei ihrer Definition initialisiert werden.
- Eine Referenz wird implizit immer als konstant definiert. Der referenzierte Name einer Referenz kann deshalb nach der Initialisierung nicht mehr geändert werden.
- Referenzen können als konstante Pointer verstanden werden, die bei jeder Verwendung automatisch dereferenziert werden.



Skript: 2.2.2.3

2-11

Referenz

II

- Beispiele - Definition und Verwendung von Referenzen:

```
int x;          // Definition der Variablen x
int* p;         // Definition eines Pointer p auf int
int i = 1;      // Definition und Initialisierung der Variablen i
int& r = i;     // Definition einer Referenz r auf i
int& r1;        // Compiler Fehler: "references must be initialized"

...

x = r;          // x = 1

r = 2;          // i = 2, r = 2, x bleibt 1 da x keine Referenz ist
i++;           // i = 3, r = 3
r++;           // i = 4, r = 4

p = &r;         // Zuweisung der Adresse von r an p (&r == &i)
(*p)++;        // i = 5, r = 5
```



Skript: 2.2.2.3 (Demo-2-02 - References)

2-12

- **Beispiel - Referenzen als Funktionsparameter:**

```
// Implementation mit Pointer
void Increment( int* iValue ) { (*iValue)++; }

void MyFunction() {
    int iLocalValue = 1;
    Increment( &iLocalValue );      // iLocalValue = 2
}

// Implementation mit Referenz
void Increment( int& iValue ) { iValue++; }

void MyFunction() {
    int iLocalValue = 1;
    Increment( iLocalValue );      // iLocalValue = 2
}
```



Benutzerdefinierte Datentypen

- Mit benutzerdefinierte Datentypen können aus vordefinierten Datentypen neue abstrakte Datentypen definiert werden
- C++ kennt folgende benutzerdefinierten Datentypen:

- Eigener Typ: `typedef unsigned char BYTE;`
- Aufzählung: `enum Color {red, green, blue};`
- Union: `union Value {int ival; double dval; char* cval};`
- Struktur: `struct Date {int day; int month; int year};`
- Klasse: `class Point {int xCoord; int yCoord};`



Klassen

I

- Klassen bilden den zentralen Datentyp für die objektorientierte Programmierung.
- Klassen ermöglichen die Abstraktion, Kapselung und Vererbung von Informationen und Operationen sowie die Instanzierung (Erzeugung) von Objekten.
- Eine Klassen-Deklaration spezifiziert nicht nur die Daten, sondern auch die darauf anwendbaren Funktionen.
- Klassen werden im Source-Code nur als Datentyp deklariert und existieren damit nur zur Kompilationszeit. Eine Klasse benötigt deshalb zur Laufzeit kein Memory (im Gegensatz zu einem instanziierten Objekt dieser Klasse).



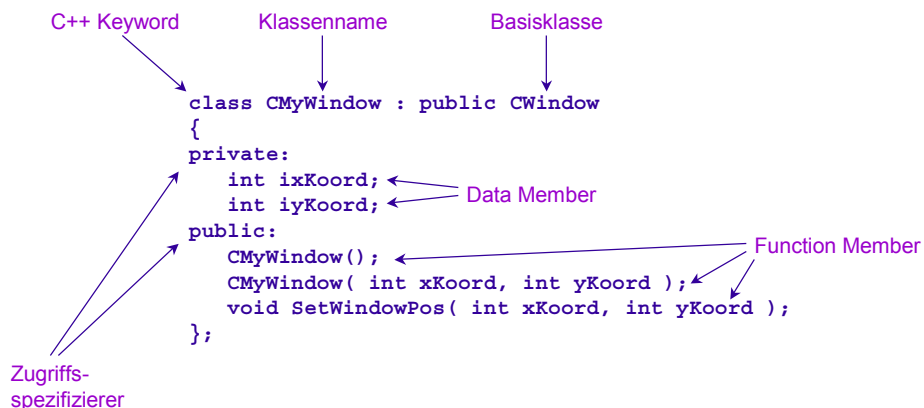
Skript: 2.2.3.3

2-15

Klassen

II

- Beispiel - Deklaration einer abgeleiteten Klasse:



Skript: 2.2.3.3

2-16

Deklaration und Definition von Variablen

- Jede Variable ist vor ihrer Verwendung genau einmal zu definieren.
 - Notation: `[storage class] type variable_name [ = literal];`
- In C++ muss die Definition einer Variable nicht am Anfang eines Blockes stehen, sondern kann an einer beliebigen Stelle erfolgen.
 - Beispiel: `for(int i=0; i<max; i++) { cout << i; ... }`
- (Lokale) Variablen sollten bei der Definition initialisiert werden.
- Bei Definition einer Variablen mit `const` kann diese nachträglich nicht verändert werden (Symbolische Konstante) und muss deshalb zwingend initialisiert werden.
 - Beispiel: `const double PI = 3.14159;`
- Definition symbolischer Konstanten mittels `#define` Anweisung ist veraltet und schwierig im Unterhalt (z.B. Debugging).



Spezifikation von Speicherklassen



- C++ unterscheidet verschiedene Speicherklassen (storage class):
 - Automatischer Speicher (Automatic Memory, Stack)
 - Statischer Speicher (Static Memory, Data Section)
 - Dynamischer Speicher (Dynamic Memory, Free Store, Heap)
- Durch die explizite Angabe der Speicherklasse kann der Geltungsbereich bzw. die Lebensdauer einer Variablen beeinflusst werden:
 - `auto` Default-Speicherklasse für lokale Variablen (Automatic Memory)
 - `register` Hinweis zur Performance-Optimierung für den Compiler (CPU Register)
 - `static` Verlängert Lebensdauer lokaler Variablen auf die Programmlebensdauer. Default-Speicherklasse für globale Variablen (Static Memory).
 - `extern` Variablen werden nur deklariert (ohne Speicher-Allokation). Die Definition erfolgt in einem anderen, durch den Linker referenzierten, Modul.



Spezifikation von Speicherklassen

II

	Definition ausserhalb eines Blockes	Definition innerhalb eines Blockes (mit static)	Definition innerhalb eines Blockes (ohne static)
Geltungsbereich	(File) global	lokal	lokal
Lebensdauer	gesamte Laufzeit des Programms	gesamte Laufzeit des Programms	Laufzeit des Blockes
Speicherkategorie	static	static	auto
Initialisierung	implizit zu 0	implizit zu 0	keine
Speicherung	Static Memory	Static Memory	Stack



Skript: 2.7.1

2-19

Nutzung von Dynamischem Speicher

I

- Dynamische Variablen werden für Daten benötigt deren Grösse zur Kompilationszeit nicht bekannt ist.
- Dynamische Variablen werden in C++ mittels `new` Operator im Dynamic Memory alloziert und mittels `delete` Operator gelöscht.
- Die Lebensdauer einer dynamische Variablen endet mit deren Freigabe (`delete`) oder bei Programmende.
- Falls `new` bei der Erzeugung einer neuen Variablen nicht genügend Speicher allozieren kann, wird ein Null-Pointer zurückgegeben.
- In einem C++ Programm müssen für Klassen immer die Operatoren `new` und `delete` verwendet werden (Konstruktor/Destruktor Aufrufe)!



Skript: 2.3.2

2-20

Nutzung von Dynamischem Speicher

II

- Beispiele - Allokation von dynamischem Speicher mit new:

```
// Einfache Beispiele
char* str = malloc(20);      // in C   - alloziert 20 Byte
char* ch  = new char;        // in C++ - alloziert ein char
char* str = new char[20];    // in C++ - alloziert Array von 20 char

// Beispiel mit komplexerer Datenstruktur
struct Node {int item; Node* left; Node* right;};
...

// Allokation in C
struct Node* node = (struct Node*) malloc(sizeof(struct Node));

// Allokation in C++
Node* node = new Node;
Object* objects = new Object[100];
```



Nutzung von Dynamischem Speicher

III

- Beispiele - Freigabe von dynamischem Speicher mit delete:

```
free(Object);      // in C
delete Object;     // in C++ für das Löschen einfacher Variablen
delete[] objects;  // in C++ für das Löschen von Arrays
```

- Anwendung von delete auf einen Null-Pointer hat keine Wirkung.
- Falls new ein Array alloziert, muss beim dazugehörigen delete ebenfalls die Array Notation (delete[]) verwendet werden (damit die dazugehörigen Destruktoren aufgerufen werden).
- Werden mit new allozierte Speicherbereiche nicht explizit wieder freigegeben, können Memory Leaks entstehen.



Geltungsbereich und Lebensdauer

- Der Programmbereich innerhalb dem eine Variablendefinition gültig ist, wird als Geltungsbereich (Scope) bezeichnet.
- Meist bestimmt der Scope auch die Lebensdauer einer Variablen.
- Beim Geltungsbereich von Variablen wird unterschieden zwischen:
 - Global Scope: Für (globale) Variablen, definiert ausserhalb eines Blockes
 - Local Scope: Für (lokale) Variablen, definiert innerhalb eines Blockes
- Der Geltungsbereich einer globalen Variablen beginnt am Definitionspunkt und erstreckt sich bis zum Ende des C++ - Files.
- Der Geltungsbereich einer lokalen Variablen beginnt am Definitionspunkt und erstreckt sich bis zum Ende des Blockes welcher die Variablen-Definition umgibt.



Skript: 2.7

2-23

Scope (Resolution) Operator

- Der Scope Operator (::) wird für die explizite Angabe des Scopes eines Identifiers verwendet (Problem der Namensverdeckung).
- Beispiel - Qualifizierung mittels Global Scope Operator:

```
int n = 0;                                     // globale Definition

int main(void)
{
    cout << "n=" << n << endl;                 // Ausgabe: n=0

    int n = 1;                                 // lokale Definition 1
    cout << "n=" << n << endl;                 // Ausgabe: n=1
    {
        int n = 2;                             // lokale Definition 2
        cout << "n=" << n << endl;           // Ausgabe: n=2
    }
    cout << "n=" << n << endl;                 // Ausgabe: n=1
    cout << "n=" << ::n << endl;             // Ausgabe: n=0 (Global)

    return 0;
}
```



(Demo-2-04 - Scope Operator)

2-24

Namensbereich (Namespace)

I

- Um Namenskonflikte zwischen verschiedenen Modulen oder Libraries zu vermeiden, können in C++ Namensbereiche verwendet werden.
- Ein Namensbereich unterteilt den globalen Namensraum und definiert einen Geltungsbereich (Scope) für einen Identifier.
 - Beispiel:

```
namespace A{void f(void);};  
namespace B{void f(void);};
```
- Die Qualifizierung des Namensbereichs erfolgt mit Hilfe des Scope-Operators.
 - Beispiel:

```
A::f(); B::f();
```
- Mit dem Keyword `using` wird der (default) Namespace angegeben.
 - Beispiel:

```
using namespace A;
```



Skript: 7.1.1 (Demo-2-05 - Namespace)

2-25

Namensbereich (Namespace)

II

- Alle C++ Standard Library Funktionen (inkl. C Runtime Library) sind im Namespace `std` deklariert.
- Header Files der C++ Standard Library werden ohne Filename Extension eingebunden.
 - Beispiel:

```
#include <iostream>
```
- Dateinamen von C Header Files werden mit einem vorangestellten “c” übernommen.
 - Beispiel:

```
#include <cstdlib>
```
- Aus Kompatibilitätsgründen sind die “normalen” C Header Files ebenfalls vorhanden, allerdings sind diese dann nicht im Standard Namespace (`std`) definiert.



Skript: 7.1.1 (Demo: <cstdlib> und <stdlib.h>)

2-26

Ausdrücke und Typenkonversion

- Ein Ausdruck ist eine Verknüpfung von Literalen, Operatoren und Identifiern. Mehrere Ausdrücke werden durch Strichpunkt getrennt.
 - Beispiel: `a = 15.5 * (b + CalcValue(c));`
- Die Auflösung von Ausdrücken erfolgt unter Berücksichtigung der Prioritäten und Auswertungsrichtung von Operatoren (vgl. Kap. 2.4.2).
- Mit Klammern wird die Auswertungsreihenfolge explizit festgelegt.
- Werden in einem Ausdruck verschiedene Datentypen verwendet, ist eine implizite (durch den Compiler) oder explizite (durch den Programmierer) Typenkonversion notwendig.



Konversion vordefinierter Datentypen

- Die Konversion vordefinierter Datentypen erfolgt implizit durch den Compiler gemäss einer arithmetischen Standardkonversion.
- Falls möglich erfolgt die Konversion ohne Informationsverlust, indem der “kleinere” Typ dem “grösseren” Typ angepasst wird.
Sichere (verlustfreie) implizite Typenkonversionen sind:
 - float --> double --> long double
 - unsigned char --> unsigned short int --> unsigned int --> unsigned long int
 - signed char --> short int --> int --> long int
 - bool --> int --> long int
- Beispiele für Typenkonversionen die u.U. verlustbehaftet sind:
 - signed --> unsigned (für negative Werte)
 - unsigned long int --> long int



Explizite Typenkonversion (Casting)

- Mit einer expliziten Typenkonversion kann eine Typenumwandlung gezielt durchgeführt oder eine implizite Standardkonversion den spezifischen Bedürfnissen angepasst werden.
- **Konversion mittels C Cast-Notation:**
 - Beispiel: `double d = (int) (12 * 1.2345);`
- **Die Konversion mittels Funktionalem-Casting ermöglicht die Typenkonversion mit einem oder mehreren Argumenten (für Klassen entspricht dies dem Aufruf eines Konstruktors).**
 - Beispiel: `double d = int(12 * 1.2345);`
- **Die Konversion mittels C++ Cast Operatoren wird hauptsächlich für Klassen verwendet. (vgl. Teil 5)**



Anweisungen

- **Selektionen**
 - if-else statement: Auswahl zwischen zwei Alternativen
 - switch-case statement: Auswahl zwischen mehreren Alternativen
- **Iterationen**
 - while statement: Schleife mit Prüfung einer Anfangsbedingung
 - do-while statement: Schleife mit mindestens einer Durchführung
 - for statement: Schleife mit einer genau definierten Anzahl Iterationen
- **Sprunganweisungen**
 - break statement: Zum Verlassen einer switch oder Iterations-Anweisung
 - continue statement: Springt ans Iterationsende und überprüft die Abbruchbedingung
 - return statement: Zum Rücksprung aus einer Funktion evtl. mit Rückgabewert
 - goto statement: Zum direkten und unbedingten Sprung auf ein Programm-Label



Funktionen

- Eine Funktion ist ein in sich abgeschlossenes Teilprogramm dem Parameter übergeben werden können und das einen Rückgabewert zurückgeben kann.
- Neben Keywords definiert der C++ Standard eine Anzahl von Standardklassen und -funktionen, welche den Funktionsumfang und die Anwendbarkeit von C++ erheblich vergrößern. (Diese sind in der C++ Standard-Klassenbibliothek zusammengefasst.)
- Funktionen können rekursiv verschachtelt werden.



Aufbau und Verwendung von Funktionen

- Bevor eine Funktion verwendet werden kann muss diese im Kontext definiert oder zumindest deklariert werden. Die Predeklaration erfolgt meist in einem Header-File.
 - Beispiel: `ShowMessage(char*, int, int);`
- Ablauf eines Funktionsaufrufs:
 - Beim Aufruf der Funktion wird ein Stack (Automatic Memory) erzeugt in welchem die Funktionsargumente sowie lokale Variablen abgelegt werden.
 - Der Funktionsname wird als Einsprungpunkt für die Code Ausführung verwendet.
 - Beim Verlassen der Funktion wird ein allfälliger Rückgabewert mit `return` zurückgegeben und der Stack (inklusive lokaler Variablen) gelöscht.



Syntax von Funktionen

- **Syntax einer Funktionsdefinition:**

```
[Spezifizierer][Rückgabotyp] Identifier ([Typ Arg1], ...) {  
    Statements;  
    [return Rückgabewert;]  
}
```

- **Spezifizierer:**

- `inline` Compiler-Hinweis den Funktionscode direkt einzufügen (nicht aufzurufen).
- `virtual` Erlaubt Member-Funktionen in einer abgeleiteten Klasse neu zu definieren.

- **Rückgabotyp:**

- Als Rückgabotyp können mit Ausnahme von Array und Funktionen beliebige Datentypen verwendet werden. Die Werterückgabe erfolgt mit dem `return` statement.

- **Argumente:**

- Die Übergabe der Argumente erfolgt durch "call by value" oder "call by reference".



Funktionsüberladung (Overloading)



- In C++ werden Funktionen aufgrund ihres Namens sowie der Anzahl und der Typen ihrer Argumente unterschieden (Signatur).
- Die Deklaration von Funktionen mit dem selben Namen aber unterschiedlicher Signatur wird als Funktionsüberladung bezeichnet (Statische oder signaturgebundene Polymorphie).
- Die eindeutige Funktionsunterscheidung für den Linker erfolgt durch die Generierung eindeutiger Symbole, welche neben dem Funktionsnamen auch die Signatur berücksichtigen (Name Mangling)
- Dieses Name Mangling erfolgt statisch zur Kompilationszeit (Static Binding) und ist Compiler und Plattform abhängig!



Funktionsüberladung (Overloading)

II

- Für die Funktionsüberladung gelten folgende Regeln:
 - Funktionen die sich nur durch den Rückgabotyp unterscheiden, sind nicht überladbar.
 - `Type` und `const Type` sind als Typen von Argumenten nicht unterschiedlich genug.
 - `Type` und `Type&` sind als Typen von Argumenten nicht unterschiedlich genug.
 - `Type*` und `Type []` sind als Typen von Argumenten nicht unterschiedlich genug.
 - Argument-Typen die sich nur durch ein `typedef` unterscheiden, sind nicht überladbar.
 - Kombinationen von obigen Regeln führen ebenfalls zu ungültigen Überladungen.



Skript: 2.6.1.8

2-35

Funktionsüberladung (Overloading)

III

- Beispiel:

```
// typedef Deklaration
typedef char* PSTR;

// Gültige Funktionsüberladung
void print( char* str, int width );
void print( char* str );
void print( int value );
void print( double value );

// Ungültige Funktionsüberladung - Rückgabotyp
int print( char* str );

// Ungültige Funktionsüberladung - typedef
void print( PSTR str );

// Ungültige Funktionsüberladung - int&
void print( int& value );
```



Skript: 2.6.1.8 (Demo-2-06 - Function Overloading / Name Mangling)

2-36

Standardargumente (Default Arguments) I

- Die Spezifikation von Default-Argumenten in C++ erlaubt die optionale Angabe von Parametern beim Aufruf einer Funktion.
 - Beispiel: `ShowMessage(char* text, int pos_x = 0, int pos_y = 0);`
- Einschränkung: Hat ein Argument einen Default-Wert, müssen alle nachfolgend aufgeführten Argumente ebenfalls mit Default-Wert deklariert werden.
- Wirkungsvoller Mechanismus für die Implementation von nachträglichen Code-Erweiterungen oder Anpassungen.
- Durch die Verwendung überladener Funktionen und/oder durch implizite Typenkonversion können Mehrdeutigkeiten entstehen (Compile- oder Runtime-Fehler)!



Standardargumente (Default Arguments) II

- **Beispiel:**

```
void ShowNum(double d, int prec=6, bool fix=true) {
    cout.precision(prec);
    if (fix) {
        cout << fixed;
    } else {
        cout << scientific;
    }
    cout << "Value = " << d << endl;
}

...

// Verwendung
double num = 12345.123456789012345;
...
ShowNum(num);           // Value = 12345.123457
ShowNum(num, 2);        // Value = 12345.12
ShowNum(num, 10, false); // Value = 1.2345123457e+004
ShowNum(num, false);    // Value = 12345 (!!!)
```



Pointer auf Funktionen

- Ein Pointer auf eine Funktion zeigt auf die Startadresse (Entry Point) dieser Funktion im Code-Teil des Programms.
- Funktions-Pointer werden verwendet um Funktionen dynamisch (zur Laufzeit) in den Code zu integrieren (z.B. Windows DLL).
 - Beispiel:

```
// Deklaration des Funktionspointers
typedef int (*LibFunc)(char *);

...
char *pString = "Test";
LibFunc fpFunc;
fpFunc = GetProcAddress("TheLibFunction");
...
// Aufruf der Funktion
iReturn = (*fpFunc)(pString);
```
- C Applikationen mit dynamischen Funktions-Pointern, können in C++ oft mit polymorphen-Funktionen realisieren werden.



Modularisierung

- Zur besseren Trennung und Modularisierung werden in C++ Interface und Implementierung einer Klasse physisch getrennt.
- Die Deklaration einer Klasse (Interface) wird in einem Header-File mit File Extension .h (.hpp oder .hxx) gespeichert.
- Die Implementation der Klasse erfolgt in einem Code-File mit File Extension .cpp (.cxx).
- Eine weitergehende Modularisierung kann durch die Verwendung von statischen und dynamische Libraries oder durch eigenständig lauffähige Software-Komponenten erreicht werden.



Einbindung von C Code in C++

I

- Aufgrund des C++ Name Mangling Mechanismus kann der Linker die Namen extern definierter C Symbole nicht auflösen.
- Mit der `extern "C"` Directive kann das Name Mangling des C++ Compilers explizit ausgeschaltet werden.
- Beispiel zur Einbindung einer einzelnen C-Funktion in C++

```
extern "C" int MyOldCFunction(char*);
```

- Beispiel zur Einbindung ganzer C-Header Files in C++

```
extern "C" {  
    #include "MyOldCLib.h"  
}
```



Skript: 2.8.2

2-41

Einbindung von C Code in C++

II

- C-Header Files zur gleichzeitigen Verwendung in C und C++

```
#ifdef __cplusplus  
    extern "C" {  
#endif  
  
    int MyOldCFunction1(char*);  
    int MyOldCFunction2(char*);  
  
#ifdef __cplusplus  
    }  
#endif
```



Skript: 2.8.2

2-42

Einsprungpunkt in ein C++ Programm

I

- **Startpunkt eines C++ Programm bildet die Funktion main():**
 - main() muss in einem Programm genau einmal definiert werden.
 - main() darf programmintern nicht als Funktion aufgerufen werden.
 - main() darf nicht überladen werden.
 - main() darf nicht static oder inline definiert werden.
 - Die Adresse von main() kann programmintern nicht ermittelt werden.
- **Vor dem Aufruf von main(), werden in C++ die Konstruktoren von globalen und static Objekten aufgerufen (Static Initialization).**
- **Analog dazu werden in C++ nach dem Verlassen von main(), die Destruktoren dieser Objekte aufgerufen (Static Destruction).**



Skript: 2.8.3

2-43

Einsprungpunkt in ein C++ Programm

II

- **Der C++ Standard definiert zwei Implementation von main():**
 - `int main(void)`
 - `int main(int argc, char *argv[])`
- **Weitere (implementationsabhängige) Parameter können vom Compiler-Hersteller hinzugefügt werden.**
 - Beispiel (MS): `int main(int argc, char *argv[], char *envp[])`
- **Der Rückgabewert von main() wird definitionsgemäss als Error-Code interpretiert und ans Betriebssystem zurückgegeben. (Rückgabewert = 0 entspricht fehlerfreiem Programmdurchlauf)**
- **Erfolgt in main() keine explizite return Anweisung, wird in C++ implizit ein "return 0;" ausgeführt.**



Skript: 2.8.3

2-44

Teil 3

Klassen und Objekte

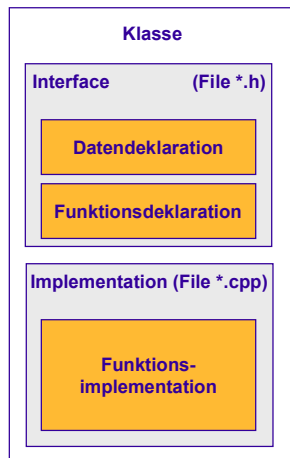
3-1

Teil 3 - Inhalt

- Was ist eine Klasse
- Erzeugung und Verwendung von Instanzen
- Zugriffsrechte und Geltungsbereich
- inline Funktionen
- const Funktionen
- Spezielle Funktions-Member
- static Daten- und Funktions-Member
- Überladen von Funktions-Member
- Operatoren für Klassen
- friend Funktionen und Klassen

Was ist eine Klasse

I



- Eine Klasse wird zur Deklaration eines benutzerdefinierten, abstrakten Datentyps verwendet, welcher neben Daten auch die auf diese Daten anwendbaren Funktionen umfasst.
- In C++ spricht man von Daten- bzw. Funktions-Member oder zusammenfassend von Klassen-Member.
- Eine Klasse wird im Code nur deklariert und besitzt somit zur Laufzeit keinen physischen Speicher.

C++

Skript: 3.1

3-3

Was ist eine Klasse

II

- **Beispiel - Interface (Deklaration) einer Klasse (Header File *.h):**

```
class CSWS_Point
{
public:
    int iX_Coord;           // X Koordinate des Punktes
    int iY_Coord;           // Y Koordinate des Punktes

    void Set(int x, int y); // Deklaration der Funktion Set
};
```

- **Beispiel - Implementation einer Klasse (Source File *.cpp):**

```
void CSWS_Point::Set(int x, int y)
{
    iX_Coord = x;
    iY_Coord = y;
}
```

C++

Skript: 3.2 und 3.3

3-4

Erzeugung von Instanzen

I

- Aus einer Klassendeklaration können Objekte (Instanzen) dieser Klasse statisch oder dynamisch definiert (instanziert) werden.
- Bei der Instanziierung von Objekten wird vom Compiler Memory bereitgestellt, wobei jede Instanz eine eigene Kopie der Daten-Member dieser Klasse enthält. Der Code der Funktions-Member dieser Klasse wird von allen Instanzen gemeinsam verwendet.
- Beispiel - Statische Instanziierung:

```
// Definition zweier Objekte der Klasse CSWS_Point  
CSWS_Point P1, P2;
```



Skript: 3.4

3-5

Erzeugung von Instanzen

II

- Beispiel - Dynamische Instanziierung:

```
// Definition zweier Pointer auf Objekte der Klasse CSWS_Point  
CSWS_Point *pP1, *pP2;  
  
// dynamische Erzeugung der Objekte  
pP1 = new CSWS_Point;  
pP2 = new CSWS_Point;  
  
// Verwendung der dynamisch angelegten Objekte  
...  
  
// Löschen der Objekte  
delete pP1;  
delete pP2;
```



Skript: 3.4

3-6

Verwendung von Instanzen

I

- **Beispiel - Verwendung statischer Objekte:**

```
CSWS_Point P1, P2;           // Objekte der Klasse CSWS_Point
int ix;                       // Hilfswert
...

// Ansprechen der Objekt Daten-Member
P1.iX_Coord = 25;
P1.iY_Coord = P1.iX_Coord;
ix = P1.CSWS_Point::iX_Coord; // explizite Schreibweise
...

// Ansprechen der Objekt Funktions-Member
P2.Set(100, 100);           // normale Schreibweise
P2.CSWS_Point::Set(100, 100); // explizite Schreibweise
...
```



Skript: 3.5.1

3-7

Verwendung von Instanzen

II

- **Beispiel - Verwendung dynamischer Objekte:**

```
// Definition zweier Pointer auf Objekte der Klasse CSWS_Point
CSWS_Point *pP1, *pP2;

// dynamische Erzeugung der Objekte
pP1 = new CSWS_Point;
pP2 = new CSWS_Point;

// Ansprechen der Member dynamischer Objekte
pP1->iX_Coord = 10;
(*pP1).iY_Coord = 20;
pP1->Set(10, 20);
pP2->CSWS_Point::Set(200, 150); // explizite Schreibweise

// Löschen der Objekte
delete pP1;
delete pP2;
```



Skript: 3.5.2

3-8

this Pointer - Implizites Funktions-Argument

- Zur eindeutigen Identifizierung der Objektdaten wird jeder Member-Funktion implizit ein zusätzliches **this** Argument übergeben.
- Dieses Argument entspricht einem **const-Pointer** auf das aufrufende Objekt und kann mit dem Keyword **this** explizit referenziert werden.

– Beispiel:

```
// P1.Set(X,Y) --> Set(CSWS_Point* const this, int x, int y);

void CSWS_Point::Set(int x, int y)
{
    this->iX_Coord = x; // identisch mit: iX_Coord = x
    this->iY_Coord = y; // identisch mit: iY_Coord = y
    func( this );      // Funktionsaufruf: func(CSWS_Point*)
}
```

- Der **this-Pointer** hat innerhalb der Member-Funktion einen konstanten Wert und kann nicht modifiziert werden.



Zugriffsrechte und Geltungsbereich

I

- Das OO-Konzept der Kapselung wird in C++ mit der Spezifikation der folgenden Klassen-Zugriffsrechte realisiert:
 - **public:** Auf public Klassen-Member kann aus jeder Funktion eines Programms zugegriffen werden. Verwendet für Member die den Zugang zu Daten und Funktionen der Klasse ermöglichen sollen.
 - **protected:** Auf protected Klassen-Member kann aus den eigenen Member-Funktionen und aus den Funktionen vererbter Klassen zugegriffen werden. Von Aussen (globaler Scope oder bei nicht vererbten Klassen) besteht jedoch keine Zugriffsmöglichkeit.
 - **private:** Klassen-Member mit private Zugriffsspezifizierer können nur von Member- und Friend-Funktionen der eigenen Klasse angesprochen werden (vollständige Kapselung gegen aussen).
- Der Zugriffsschutz wird durch den Compiler geprüft und erfolgt damit statisch bereits während dem kompilieren.



Zugriffsrechte und Geltungsbereich

II

- Das standard Zugriffsrecht für Klassen die mittels `class` definiert werden ist `private`, für Klassen die mittels `struct` definiert werden gilt `public` Zugriff.
- Innerhalb einer Klassendeklaration können beliebig viele Zugriffsspezifizierer in beliebiger Reihenfolge definiert werden. Ein definierter Zugriff gilt für alle folgenden Member bis zum nächsten Zugriffsspezifizierer oder bis zum Ende der Klassendeklaration.
- Die Namen von Klassen-Member haben den Scope Klasse, weshalb der selbe Name für Daten- oder Funktions-Member in mehreren Klassen definiert werden kann.

– Beispiel: `CSWS_Point P1; CSWS_Line L1;`
...
`P1.Set(150, 100);` // Aufruf von `CSWS_Point::Set()`
`L1.Set(15, 25);` // Aufruf von `CSWS_Line::Set()`



Skript: 3.6

3-11

Verschachtelte Klassendeklaration

I

- Die Deklaration einer lokal (innerhalb einer Klasse) verwendeten Klasse kann innerhalb der Deklaration der umgebenden Klasse erfolgen (Nested Class Declaration).

• Beispiel:

```
class CSWS_Line
{
    // verschachtelte Klassendeklaration
    class CSWS_Point {
        int iX;
        int iY;
    public:
        void Set(int x, int y);
    };

private:
    CSWS_Point P1, P2;
    ...
};
```



3-12

Verschachtelte Klassendeklaration

II

- Der verschachtelte (eingebettete) Klassenname besitzt nur lokalen Scope innerhalb der umgebenden Klasse.
- Die Implementation einer Member-Funktion einer verschachtelten Klasse muss deshalb mit dem Namen der umgebenden Klasse qualifiziert werden.
- **Beispiel:** `void CSWS_Line::CSWS_Point::Set(int,int) {...};`
- Member-Funktionen einer verschachtelten Klasse haben keine besonderen Zugriffsrechte auf Klassen-Member der umgebenden Klasse und umgekehrt.

C++

3-13

forward Deklaration

- Eine Forward-Deklaration deklariert den Namen einer Klasse, sodass Pointer und Referenzen auf die Klasse erzeugt werden können, bevor die Klasse vollständig deklariert ist.
- **Beispiel - Header-File Deklaration:**

```
class CSWS_Point;

class CSWS_Line {
private:
    CSWS_Point* pP1;
    CSWS_Point* pP2;
    ...
};
```

C++

3-14

inline Funktionen

- Die Definition einer `inline` Funktion ist ein Hinweis für den Compiler keinen Funktionsaufruf durchzuführen, sondern direkt den Code der Funktion an den entsprechenden Stellen im Programm einzusetzen.

- **Beispiel – Klassendeklaration mit Impliziter inline-Definition:**

```
class CSWS_Point{
private:
    int iX_Coord, iY_Coord; // X und Y Koordinate des Punktes
public:
    // Implizite inline Implementation in der Klassendeklaration
    void Set(int x, int y) {iX_Coord = x; iY_Coord = y;}
};
```

- **Beispiel - Explizite inline-Definition einer Funktion:**

```
inline void CSWS_Point::Set(int x, int y){
    iX_Coord = x;
    iY_Coord = y;
}
```



const Funktionen



- Member-Funktionen welche die Daten-Member eines Objektes nicht verändern, können als `const` deklariert werden.
- Der `const` Spezifizierer bildet einen Teil der Signatur einer Member-Funktion und wird deshalb beim Überladen einer Funktion ebenfalls berücksichtigt.
- Innerhalb einer `const` Member-Funktion können nur Member-Funktionen verwendet werden, die ebenfalls mit `const` deklariert sind.
- Auf ein mit `const` definiertes Objekt kann nur mit `const` Member-Funktionen zugegriffen werden.



- **Beispiel:**

```
// Deklaration einer const Member-Funktion
class CSWS_Line {
public:
    void List(void) const;
    int GetLength(void);
    ...
};

// Implementation einer const Member-Funktion
void CSWS_Line::List( void ) const {
    cout << "P1 at x:" << P1.x << " y:" << P1.y << endl;
    ...
}

// Verwendung einer const Member-Funktion
const CSWS_Line L1(1,1,5,5); // const Objektdefinition
L1.List();                  // Ok, const Funktionsaufruf
int i = L1.GetLength();     // Compiler Error: non-const Function
```

Spezielle Funktions-Member

- Bei der Deklaration einer Klasse werden vom Compiler automatisch die folgenden vier Member-Funktionen erzeugt:
 - Default-Konstruktor
 - Default-Copy-Konstruktor
 - Default-Zuweisungsoperator
 - Default-Destruktor

Konstruktor

I

- C++ verwendet für die Initialisierung eines Objektes spezielle Member-Funktionen, die sogenannten Konstruktoren.
 - Beispiel:

```
// Konstruktor Funktionsdeklaration (Klassenname)
CSWS_Point(int x, int y);
```
- Der Konstruktor wird beim instanzieren eines Objektes automatisch durch den Compiler aufgerufen, unabhängig davon ob die Instanzierung statisch oder dynamisch erfolgt.
- Ein Konstruktor der ohne Argumente aufgerufen werden kann, wird als Default-Konstruktor (Standard-Konstruktor) bezeichnet.
 - Allgemeine Schreibweise:

```
T::T()
```
 - Beispiel:

```
CSWS_Point::CSWS_Point();
```



Skript: 3.12.1

3-19

Konstruktor

II

- Regeln zur Implementation von Konstruktoren:
 - Der Funktionsname eines Konstruktors ist identisch mit dem Klassennamen.
 - Eine Klasse kann mehrere Konstruktoren besitzen. Diese müssen aber in Bezug auf den Typ bzw. die Anzahl der Argumente verschieden sein (vgl. Funktionsüberladung).
 - Ein Konstruktor darf keinen Rückgabetyt besitzen (auch nicht void).
 - Ein Konstruktor kann nicht static oder virtual deklariert werden.
 - Falls in der Klassendeklaration kein Konstruktor spezifiziert wird, erzeugt der Compiler automatisch einen Default-Konstruktor.
 - Wird für eine Klasse ein Konstruktor definiert, wird der Default-Konstruktor vom Compiler nicht automatisch erzeugt. Ein allenfalls notwendiger Default-Konstruktor muss in diesem Fall manuell implementiert werden.
- Für global definierte Objekte wird der Konstruktor automatisch aufgerufen und zwar vor dem Start des Programms (main).



Skript: 3.12.1

3-20

- **Beispiel - Konstruktordeklaration:**

```
class CSWS_Point {  
    int iX_Coord, iY_Coord;    // X,Y Koordinaten des Punktes  
public:  
    CSWS_Point(void);          // Default-Konstruktor  
    CSWS_Point(int x, int y);  // Konstruktor  
    void Set(int x, int y);    // Deklaration der Funktion Set()  
};
```

- **Beispiel - Konstruktorimplementation:**

```
CSWS_Point::CSWS_Point(void)  
{  
    iX_Coord = 0; iY_Coord = 0;  
}  
  
CSWS_Point::CSWS_Point(int x, int y)  
{  
    iX_Coord = x; iY_Coord = y;  
}
```



- **Beispiel - “statischer” Konstruktoraufruf:**

```
CSWS_Point P1;          // impliziter Aufruf von CSWS_Point()  
CSWS_Point P2();        // Fehler: Funktionsdeklaration !!  
CSWS_Point P3(100, 50); // Aufruf von CSWS_Point(int, int)
```

- **Beispiel - “dynamischer” Konstruktoraufruf:**

```
CSWS_Point* pP1 = new CSWS_Point;          // impliziter Aufruf  
CSWS_Point* pP2 = new CSWS_Point();        // expliziter Aufruf  
CSWS_Point* pP3 = new CSWS_Point(100, 50); // expliziter Aufruf
```



Konstruktor Initialisiererliste

- Die Initialisierung von Daten-Member eines Objektes kann im Konstruktor auch mit Hilfe einer Initialisiererliste erfolgen.

– Beispiel:

```
CSWS_Point::CSWS_Point(int x, int y)
:iX_Coord(x), iY_Coord(y)
{
    ...
}
```

- Diese Art der Initialisierung ist effizienter als eine Zuweisung im Rumpf des Konstruktors, bei const- Daten-Membren ist es sogar die einzig mögliche.



3-23

Copy-Konstruktor

I

- Ein Copy-Konstruktor ist ein Konstruktor um Objekte zu kopieren.
- Implizit wird der Copy-Konstruktor aufgerufen, wenn ein Objekt einer Funktion "by value" übergeben wird, oder wenn ein Objekt bei der Instanzierung durch eine Objektzuweisung initialisiert wird.

– Beispiele:

```
CSWS_Point P1(5,10);           // normaler Konstruktor
CSWS_Point P2(P1);             // expliziter Copy-Konstruktor
CSWS_Point P3 = P1;             // impliziter Copy-Konstruktor
CSWS_Point P4 = CSWS_Point(P1); // impliziter Copy-Konstruktor
CSWS_Point P5 = CSWS_Point(5,10); // normaler Konstruktor
```

- Ein Copy-Konstruktor wird minimal mit einer Referenz auf das zu kopierende Objekt deklariert.
 - Allgemeine Schreibweise: `T::T(const T&)`
 - Beispiel: `CSWS_Point::CSWS_Point(const CSWS_Point&)`



Skript: 3.12.2

3-24

Copy-Konstruktor

II

- **Regeln zur Implementation von Copy-Konstruktoren:**
 - Wird ein Copy-Konstruktor nicht explizit deklariert, erzeugt der Compiler automatisch einen Default-Copy-Konstruktor.
 - Der Default-Copy-Konstruktor kopiert Daten-Member elementweise.
 - Bei der Variablen-Definition mit einer Zuweisung wird implizit ein Copy-Konstruktor aufgerufen.
 - Werden innerhalb einer Klasse Daten-Member dynamisch alloziert, muss die explizite Implementation eines Copy-Konstruktors geprüft werden.
 - Wird für eine Klasse ein spezifischer Zuweisungsoperator implementiert, muss die explizite Implementation eines Copy-Konstruktors geprüft werden.



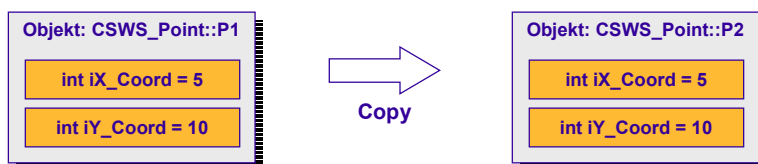
Skript: 3.12.2

3-25

Copy-Konstruktor

III

- Für Objekte mit Membern “by value” kopiert der Default-Copy-Konstruktor die Daten-Member elementweise (Shallow Copy):



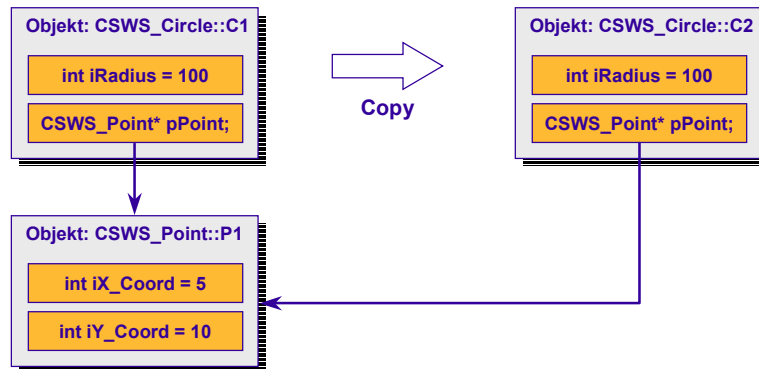
Skript: 3.12.2.1

3-26

Copy-Konstruktor

IV

- Für Objekte mit Membern “by reference” kopiert der Default-Copy-Konstruktor auch Pointer Daten-Member elementweise:



C++

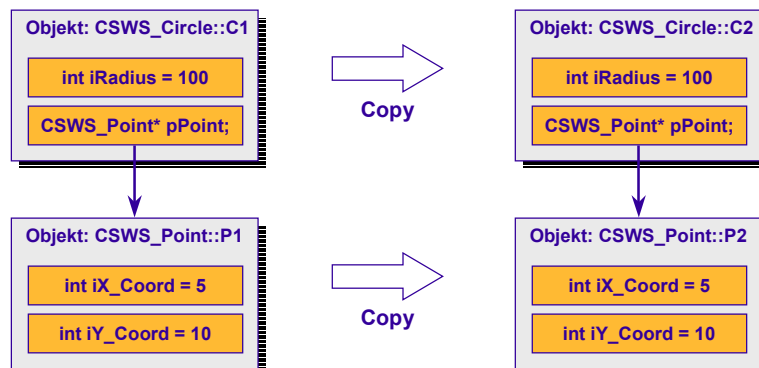
Skript: 3.12.2.2

3-27

Copy-Konstruktor

V

- Für Objekte mit Membern “by reference” muss meist ein spezieller Copy-Konstruktor manuell implementiert werden (Deep Copy):



C++

Skript: 3.12.2.3 (Demo-3-01 - CopyConstructor)

3-28

Zuweisungsoperator

I

- Die Zuweisung an Objekte einer Klasse wird mit dem Zuweisungsoperator definiert.
 - Allgemeine Schreibweise: `T& T::operator=(const T&)`
 - Beispiel: `CSWS_Point& CSWS_Point::operator=(const CSWS_Point&)`
- Bei Definition mit Zuweisung wird implizit ein Copy-Konstruktor aufgerufen, nur bei vorgängiger Definition und anschliessender Zuweisung (Initialisierung), wird der Zuweisungsoperator benutzt.
 - Beispiele:

```
CSWS_Point P1(5,10); // normaler Konstruktor
CSWS_Point P2(P1);   // expliziter Copy-Konstruktor
CSWS_Point P3 = P1;  // impliziter Copy-Konstruktor
CSWS_Point P4;       // normaler Default-Konstruktor
P4 = P1;              // Zuweisungsoperator
```
- Beim Zuweisungsoperator sind grundsätzlich die gleichen Regeln zu berücksichtigen wie bei einem Copy-Konstruktor.



Skript: 3.12.3

3-29

Zuweisungsoperator

II

- Regeln für die Implementation eines Zuweisungsoperators:
 - Wird ein Zuweisungsoperator nicht explizit deklariert, erzeugt der Compiler automatisch einen Default-Zuweisungsoperator.
 - Beim Default-Zuweisungsoperator erfolgt die Zuweisung elementweise.
 - Werden innerhalb einer Klasse Daten-Member dynamisch alloziert, muss die explizite Implementation eines Zuweisungsoperators geprüft werden.
 - Wird für eine Klasse ein spezifischer Copy-Konstruktor implementiert, muss die explizite Implementation eines Zuweisungsoperators geprüft werden.



Skript: 3.12.3

3-30

Zuweisungsoperator

III

- Eine Referenz als Rückgabewert des Zuweisungsoperators erlaubt die Verkettung der Zuweisungsoperation in einem Ausdruck:

- Beispiel - einfacher Aufruf (auch mit void Rückgabotyp möglich):

```
y = z;           // impliziter Aufruf  
y.operator=(z);  // expliziter Aufruf
```
- Beispiel - verketteter Aufruf (nur mit Referenzrückgabotyp möglich):

```
x = y = z;           // impliziter Aufruf  
x.operator=(y.operator=(z)); // expliziter Aufruf
```



Destruktor

I

- Analog dem automatischen Konstruktoraufruf beim Erstellen eines Objektes, wird beim Löschen des Objektes durch den Compiler implizit der Destruktor aufgerufen.
- Der Destruktor kann verwendet werden um beim Löschen eines Objektes dynamisch allozierten Speicher freizugeben oder um geöffnete Files zu schliessen etc.
- Der Funktionsname des Destruktors ist identisch mit dem Klassennamen mit einem vorangestellten Tilde-Zeichen “~” (auch als Komplement des Konstruktors bezeichnet).
 - Allgemeine Schreibweise: `T::~~T()`
 - Beispiel: `CSWS_Point::~~CSWS_Point();`



- **Regeln für die Implementation eines Destruktors:**
 - Ein Destruktor kann keinen Funktionswert zurückgegeben (nicht einmal void) und besitzt keine Argumente, weshalb ein Destruktor nicht überladen werden kann.
 - Da der Rumpf eines Destruktors ausgeführt wird bevor die Destrukturen der eingebetteten Daten-Member aufgerufen werden, können Klassen-Member innerhalb des Destruktors normal verwendet werden.
 - Wird für eine Klasse kein expliziter Destruktor deklariert, erzeugt der Compiler einen Default-Destruktor, welcher die Daten-Member elementweise löscht.
 - In Vererbungshierarchien sind Destrukturen als virtual zu deklarieren.
- **Für global definierte Objekte wird der Destruktor automatisch aufgerufen und zwar nach Beenden des Programms.**



- **Static Daten-Member sind nicht an ein bestimmtes Objekt einer Klasse gebunden, sondern speichern ihren Wert einmal für alle Objekte einer bestimmten Klasse.**
- **Die Deklaration eines static Daten-Members innerhalb einer Klassen-deklaration ist noch keine Definition! Diese muss an separater Stelle des Programms implementiert werden.**
- **Static Daten-Member können ohne die Instanzierung eines Objektes (unter Berücksichtigung der Zugriffsrechte) angesprochen werden.**



- **Beispiel - Static Daten-Member Deklaration (Header-File *.h):**

```
class CSWS_Point {  
    int iX_Coord, iY_Coord; // X,Y Koordinatendes Punktes  
public:  
    static bool fVisible;    // Flag für Sichtbarkeit aller Punkte  
public:  
    void Set(int, int);  
    void SetVisibility(bool);  
};
```

- **Beispiel - Static Daten-Member Definition (Source-File *.cpp):**

```
bool CSWS_Point::fVisible = true; // Definition und Initialisierung
```

- **Beispiel - Zugriff auf einen Static Daten-Member:**

```
cout << "Visibility = " << CSWS_Point::fVisible;
```



static Member-Funktionen

- Eine static Member-Funktions ist eine in der Klasse deklarierte Funktion die unabhängig von einem Objekt verwendet werden kann, um auf static Daten-Member zuzugreifen.
- Eine static Member-Funktion besitzt somit keinen this-Pointer, weshalb innerhalb einer solchen Funktion nicht auf non-static Klassen-Member zugegriffen werden kann.
- Eine static Member-Funktion darf weder virtuell deklariert, noch darf der selbe Name für ein static und ein non-static Klassen-Member verwendet werden.
- **Beispiel - Static Data-Member als Objektzähler**



Effective C++

- C++ wird oft zu unrecht als "langsam" bezeichnet, weil Code ineffizient implementiert wird. Zu beachten sind deshalb die folgenden Regeln:
- Weil Konstruktoren, Copy-Konstruktoren und Zuweisungs-Operatoren sehr oft aufgerufen werden, sollten diese möglichst effizient implementiert werden.
- Für einen Funktionsaufruf mit Objekt-Parametern sollte nach Möglichkeit ein "Call By Reference" Aufruf verwendet werden.
- Die Verwendung einer Initialisiererliste für Objekt-Member ist effizienter als eine Zuweisung im Konstruktor.



Überladen von Funktions-Member

- Analog wie normale Funktionen können auch Konstruktoren und andere Funktions-Member einer Klasse überladen werden (statische oder signaturgebundene Polymorphie).
- Für überladene Member-Funktionen gelten die gleichen Regeln bezüglich der Eindeutigkeit der Signatur wie für normale Funktionen.
- Jede Deklaration einer überladenen Member-Funktion wird als eigenständige Funktion mit Klassen-Scope implementiert.
- Die Vererbung von überladenen Member-Funktionen ist ebenfalls möglich.



Operatoren für Klassen

I

- Neben Funktionen können für Klassen auch Operatoren deklariert und überladen werden (Operator Overloading).
- Eine Deklaration des “Funktionsnamens” für Operatoren besteht aus dem Keyword `operator` und einem entsprechenden Operatorzeichen.
 - Beispiel: `bool CSWS_Line::operator>=(const CSWS_Line &aLine);`
- Das verwendete Operatorzeichen kann beliebig ausgewählt werden, sollte aber möglichst dem üblichen Gebrauch entsprechen.



Skript: 3.10

3-39

Operatoren für Klassen

II

- Für die Definition von Operatoren gelten folgende Regeln:
 - Es ist nicht möglich neue Operatorzeichen zu definieren oder bestehende Operatorzeichen zu erweitern (z.B. ist `**` für die Potenzierung ist nicht zulässig).
 - Die Priorität für die Auflösung von Ausdrücken ist ebenfalls festgelegt und kann nicht verändert werden (Sprachstandard).
 - Die für vordefinierte Datentypen verwendeten Operatoren werden nicht automatisch auf abstrakte Datentypen (Klassen) übertragen.
 - Die Operatoren für vordefinierte Datentypen sind festgelegt und können weder umdefiniert noch erweitert werden.
 - Die Operatoren `new` und `delete` können ebenfalls überladen werden. (vgl. Skript)
 - Die folgenden Operatoren können nicht überladen werden:
 - Operator `sizeof()` (Grösse)
 - Operator `.` (Elementzugriff)
 - Operator `::` (Scope-Operator)
 - Operator `?:` (Konditional-Operator)



Skript: 3.10

3-40

Operatoren für Klassen

III

- **Beispiel - Operator Deklaration:**

```
CSWS_Line operator+(CSWS_Line& addLine);
```

- **Beispiel - Operator Implementation:**

```
CSWS_Line CSWS_Line::operator+(CSWS_Line& addLine)
{
    CSWS_Line tmpLine;          // define temporary local object

    // calculate components of temporary line
    tmpLine.P1 = P1;
    tmpLine.P2 = P2 + addLine.P2 - addLine.P1;
    return tmpLine;             // implicit call of copy constructor
}
```

- **Beispiel - Operator Aufruf:**

```
L3 = L1 + L2 ;                // normale Schreibweise
L3 = L1.operator+(L2);        // explizite Schreibweise
```



Skript: 3.10 (Demo-3-05 - Operators)

3-41

friend Funktionen

I

- Eine friend Deklaration erlaubt Non-Member Funktionen den Zugriff auf geschützte (protected, private) Daten-Member einer Klasse.
- Als Friend-Funktionen können sowohl globale Funktionen als auch Member-Funktionen einer anderen Klasse verwendet werden.
- Eine Friend-Deklaration wird durch Zugriffsspezifizierer (public, protected, private) nicht beeinflusst und kann an beliebiger Stelle innerhalb der Klassendeklaration erfolgen.
- Ist eine Friend-Funktion kein Klassen-Member, ist der implizite this-Pointer undefiniert und kann nicht benutzt werden. (Objekte müssen dann als Argumente übergeben werden.)
- Friend Beziehungen werden weder vererbt noch sind sie transitiv.



Skript: 3.13

3-42

friend Funktionen



- Friend-Funktionen widersprechen dem Prinzip der Kapselung und werden vor allem aus Effizienzgründen eingesetzt.
- Beispiel - Deklaration von Friend-Funktionen:

```
class CSWS_Point;  
  
CSWS_Line {  
    CSWS_Point* pP1; CSWS_Point* pP2;  
public:  
    void CSWS_Line::List(void);  
    ...  
};  
  
class CSWS_Point {  
    friend void CSWS_Line::List(void);  
private:  
    int x; int y;  
public:  
    CSWS_Point(void);  
    ...  
};
```



Skript: 3.13

3-43

friend Klassen

- Neben einzelnen Funktionen können auch ganze Klassen als friend deklariert werden.
- Beispiel - Deklaration von CSWS_Line als friend-Klasse:

```
class CSWS_Point {  
    friend class CSWS_Line;  
private:  
    int x;  
    int y;  
public:  
    CSWS_Point(void);  
    ...  
};
```



Skript: 3.13

3-44

Teil 4

C++ Templates

4-1

Teil 4 - Inhalt

- Template Funktionen
- Template Klassen

Templates

- C++ - Templates (Vorlage, Schablone) ermöglichen die generische (allgemeingültige) Programmierung von Funktionen und Klassen die mit einem oder mehreren Datentypen parametrisiert werden können.
- Der typenspezifische Code für Funktionen und Klassen wird vom Compiler für konkret definierte Datentypen automatisch, entsprechend der Template-Deklaration, generiert (instanziert).
- Templates werden hauptsächlich für die Realisierung von typensicheren, allgemeinen Hilfsfunktionen (z.B. min, max) und für Container-Klassen (z.B. Stacks, verkettete Listen) verwendet.
- Richtig eingesetzt, sind Templates zeitsparend und fehler-reduzierend weil die Implementation von redundantem Code vermieden werden kann.



Template Funktionen



- Die Deklaration von Template Funktionen und Klassen erfolgt mit dem Keyword `template` sowie einer Typenparameterliste.

- Beispiel - Generische max() Funktion:

```
// Deklaration der Template-Funktion max()
template <class T> T max(T, T);

// Implementation der Template-Funktion max()
template <class T> T max(T a, T b)
{
    return ( a > b ? a : b );
}

// Verwendung der Template-Funktion max()
int i1=10, i2=20, i3=0;
double d1=10.1, d2=20.2, d3=0.0;
i3 = max(i1,i2);    // generiert: int max(int,int);
d3 = max(d1,d2);    // generiert: double max(double, double);
```



Template Funktionen



- **Regeln für die Implementation von Template-Funktionen:**

- Template-Funktionen können inline implementiert werden.
- Beim Aufruf einer Template-Funktionen erfolgt keine implizite Typenumwandlung der Funktionsparameter durch den Compiler.
- Die Implementation einer Template-Funktion kann intern beliebige lokale Variablen des Template-Datentyps besitzen. Lokale Variablen können zudem statisch deklariert werden, wobei für jeden Datentyp für den die Funktion generiert wird, eine separate statische Variable definiert wird.
- Eine Template-Funktion kann durch mehrere Datentypen parametrisiert werden:

```
template <class T1, class T2>
void func(T1 a, T2 b, int size) {
    cout << "Argument 1: " << a << endl;
    cout << "Argument 2: " << b << endl;
    cout << "Argument 3: " << size << endl;
}
```



Skript: 5.1

4-5

Template Funktionen



- Durch Überladen einer Template-Funktion kann die Funktion für beliebige Kombinationen von Typenparametern verwendet oder die Implementation für spezifische Datentypen optimiert werden.

- **Beispiel - Standard max() Template-Funktion für C-Strings:**

```
char *s1 = "StringA", *s2 = "StringB";
char* maxstr = max(s1, s2); // Vergleicht nur String-Adressen!
```

- **Beispiel - Überladen der max() Template-Funktion für C-Strings:**

```
char* max(char* a, char* b) {
    return ((strcmp(a, b) > 0) ? a : b);
}
```



Skript: 5.1 (Demo-4-01 - TemplateFunctions)

4-6

Template Klassen

I

- Eine Template-Klasse ist eine generische Klasse, die mit einem oder mehreren Datentypen parametrisiert ist.
- Ein Objekt einer Template-Klasse wird vom Compiler erzeugt (instanziert), indem die Typenparameter durch die in der Definition spezifizierten Datentypen ersetzt werden.
 - Beispiel: `Stack<int> IntStack; // generiert Stack vom Typ int`
- Damit der Compiler die typenspezifischen Klassen instanzieren kann, muss der Template-Code im Header-File implementiert werden (Widerspruch zum Konzept der Kapselung).



Skript: 5.2

4-7

Template Klassen

II

- Regeln für die Implementation von Template-Klassen:
 - Klassen können grundsätzlich als Template (allgemeingültig) implementiert werden.
 - Für jeden unterschiedlichen Typparameter eines Templates wird durch den Compiler implizit Code generiert.
 - Innerhalb einer Template-Klasse kann der Datentyp des Template-Parameters wie jeder andere Datentyp zur Deklaration von Klassen-Membnern verwendet werden.
 - Bei der Instanzierung einer Template-Klasse müssen die verwendeten Datentypen explizit deklariert werden.
 - Template-Klassen können in Vererbungen verwendet werden.
 - In Template-Klassen ist das Überladen von Member-Funktionen erlaubt.
 - Inline-Implementationen von Member-Funktionen sind zulässig.
 - Template-Klassen können verschachtelt werden, d.h. eine Template-Klasse kann als Typenparameter einer anderen Template-Klasse verwendet werden.
 - Statische Variablen können verwendet werden, wobei separate statische Member pro instanzierter Klasse angelegt werden.



Skript: 5.2

4-8

Template Klassen - Beispiel

I

- **Beispiel - Deklaration einer generischen Stack-Klasse:**

```
template <class T>
class Stack {
private:
    int _iSize;
    int _iCount;
    T* _Elements;
public:
    ~Stack(void); // destructor
    Stack(int size = 10); // constructor
    Stack(const Stack<T>&); // copy constructor
    Stack<T>& operator=(const Stack<T>&); // assignment operator

    void Push(T); // push element on stack
    T Pop(void); // pop element from stack
};
```



Skript: 5.2.1

4-9

Template Klassen - Beispiel

II

- **Beispiel - Implementation einer generischen Stack-Klasse:**

```
// stack constructor
template <class T> Stack<T>::Stack(int size)
: _iSize(size), _iCount(0), _Elements(new T[_iSize])
{}

// stack destruktur
template <class T> Stack<T>::~~Stack() {
    delete[] _Elements;
}

// push stack element
template <class T> void Stack<T>::Push(T x) {
    _Elements[_iCount++] = x;
}

// pop stack element
template <class T> T Stack<T>::Pop(void) {
    return _Elements[--_iCount];
}
```



Skript: 5.2.1

4-10

- Beispiel - Verwendung einer generischen Stack-Klasse:

```
// Statische Definition
Stack<int> IntStack(20);           // stack of integers
Stack<double> DoubleStack;        // stack of doubles
Stack<CSWS_Point> PointStack;     // stack of CSWS_Points

// Dynamische Definition
Stack<int>* pIntStack = new Stack<int>; // pointer to int stack

// Typedef Definition
typedef Stack<int> IStack;
IStack aIntStack[10];             // array of int stacks

// Verschachtelte Definition (Leerzeichen beachten!!)
Stack< Stack<int> > IntStackStack; // stack of int stacks
Stack< Stack<CSWS_Point> > PointStackStack; // stack of point stacks
```

Teil 5

Klassenbeziehungen

5-1

Teil 5 - Inhalt

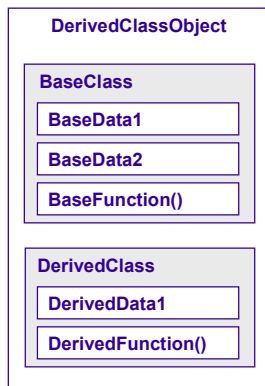
- Vererbungsbeziehung
- Assoziationsbeziehung
- Aggregationsbeziehung
- Kompositionsbeziehung
- Abhängigkeitsbeziehung

Klassenbeziehungen

- Eine Klasse wird in einer realen Applikation nicht für sich isoliert verwendet, sondern meist bestehen vielfältige Beziehungen zu anderen Klassen.
- Die Modellierung eines Systems durch miteinander interagierende Objekte, erfordert eine entsprechende Modellierung der Beziehungen zwischen Klassen.
- C++ kennt verschiedene Möglichkeiten Klassenbeziehungen abzubilden, wobei die Vererbungsbeziehung die mächtigste darstellt und für die objektorientierte Programmierung unbedingt erforderlich ist.



Vererbungsbeziehung



- Eine Vererbungsbeziehung (Inheritance-Relation) beschreibt die hierarchische Abhängigkeit einer Unterklasse von einer Basisklasse.
- Eine Vererbungsbeziehung wird zur Abbildung eines Generalisierungs- bzw. Spezialisierungs-Verhältnisses zwischen zwei Klassen verwendet.



Vererbungsbeziehung



- Bei der Vererbung werden die Daten- und Funktions-Member der Basisklasse (unter Berücksichtigung der Zugriffsrechte) an die abgeleitete Klasse übertragen.
- Eine Ausnahme bilden die Konstruktoren, der Destruktor und der Zuweisungsoperator, die nicht an die abgeleitete Klasse vererbt werden. Die Default-Implementation dieser Member-Funktionen wird für die abgeleitete Klasse implizit vom Compiler erzeugt.
- Für die Default-Implementation dieser Member-Funktionen gelten die gleichen Regeln (vgl. Copy-Konstruktor, Zuweisungsoperator) wie bei einer nicht vererbten Klasse.



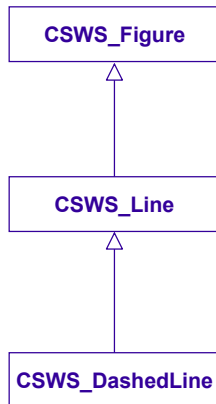
Vererbungsbeziehung



- Die Vererbung bewirkt eine Verschachtelung des Geltungsbereichs von Klassen, wobei der Scope einer abgeleiteten Klasse vom Scope der Basisklasse(n) umgeben wird.
- Innerhalb der Implementation einer abgeleiteten Member-Funktion kann deshalb auf alle Klassen-Member der Basisklasse(n) genauso zugegriffen werden als seien diese in der abgeleiteten Klasse definiert worden (Zugriffsrechte berücksichtigen).
- Durch mehrfaches Ableiten über mehrere Stufen können Klassen-hierarchien aufgebaut werden (Bsp. MFC-Klassenbibliothek).
- Man unterscheidet zwischen direkter und indirekter Basisklasse, je nachdem ob eine vererbte Klasse unmittelbar oder von einer weiter oben in der Vererbungshierarchie deklarierten Klasse abgeleitet wird.



Vererbungsbeziehung - Beispiel



```
class CSWS_Figure {
protected:
    int iColor;
public:
    CSWS_Figure(void);
};

class CSWS_Line : public CSWS_Figure {
protected:
    CSWS_Point P1, P2;
public:
    CSWS_Line(void);
    ...
};

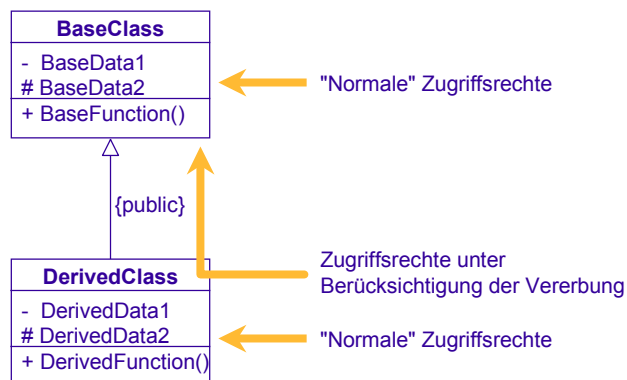
class CSWS_DashedLine : public CSWS_Line {
public:
    CSWS_DashedLine(void);
    ...
};
```



Vererbung von Zugriffsrechten

I

- Für die Zugriffsrechte auf Member einer abgeleiteten Klasse muss berücksichtigt werden wo diese Klassen-Member definiert wurden:



Vererbung von Zugriffsrechten

II

- **Regeln für die Vererbung von Zugriffsrechten:**

- **public:** Bei einer **public-Vererbung** einer Klasse werden public Member der Basisklasse zu public Members der abgeleiteten Klasse und protected Member der Basisklasse zu protected Members der abgeleiteten Klasse.
- **protected:** Bei einer **protected-Vererbung** einer Klasse werden public und protected Member der Basisklasse zu protected Members der abgeleiteten Klasse.
- **private:** Bei einer **private-Vererbung** einer Klasse werden public und protected Member der Basisklasse zu private Members der abgeleiteten Klasse.
- Unabhängig von der Vererbung der Zugriffsrechte sind private Klassen-Member der Basisklasse in abgeleiteten Klassen nicht zugänglich (Ausnahme: friend deklarierte Klassen der Basisklasse).
- Ohne explizite Angabe des Zugriffsspezifizierers wird für eine mit class deklarierte abgeleitete Klasse die private-Vererbung als Default verwendet. Für eine mit struct deklarierte abgeleitete Klasse wird als Default public-Vererbung benutzt.



Skript: 4.2.2

5-9

Vererbung von Zugriffsrechten

III

- **Zusammenfassung der Vererbung von Zugriffsrechten:**

Vererbung	Basisklasse		Abgeleitete Klasse
public	public	→	public
	protected	→	protected
	private	-- ✗ --	no access
protected	public	→	protected
	protected	→	protected
	private	-- ✗ --	no access
private	public	→	private
	protected	→	private
	private	-- ✗ --	no access



Skript: 4.2.2

5-10

Typenkonversion abgeleiteter Klassen

- Zwischen einer public abgeleiteten Klasse und ihrer Basisklasse sind die folgenden impliziten Typenkonversionen definiert:
 - Ein **Objekt einer abgeleiteten Klasse** wird implizit in ein Objekt der Basisklasse konvertiert.
 - Eine **Referenz auf ein Objekt einer abgeleiteten Klasse** wird implizit in eine Referenz auf ein Objekt der Basisklasse konvertiert.
 - Ein **Pointer auf ein Objekt einer abgeleiteten Klasse** wird implizit in einen Pointer auf ein Objekt der Basisklasse konvertiert.
- Durch diese implizite Typenkonversion können bei Initialisierungen, Parameterübergaben und Zuweisungen anstelle von Objekten der Basisklasse auch Objekte der abgeleiteten Klasse stehen.
- Dadurch ist eine public abgeleitete Klasse vollständig typenkompatibel zu ihrer Basisklasse, dies gilt aber nicht für private oder protected abgeleitete Klassen!



5-11

C++ Cast Operatoren

I

- Der Operator `static_cast<T>()` wird verwendet, wenn eine implizite Konversion vom Typ des Ausdrucks in Typ (T) möglich ist.
 - Beispiel:

```
int n = 3, m = 2;
double x = static_cast<double>(n)/m;
```
- Mittels `const_cast<T>()` Operator kann die `const`-Deklaration eines Pointers aufgehoben werden.
 - Beispiel:

```
const int* i = new int(0); int n = 10;
*(const_cast<int*>(i)) = n;
```
- Bei der Umwandlung mittels `reinterpret_cast<T>()` wird jegliche Typenprüfung ausgeschaltet. (Verwendung zum Beispiel für `void*` Pointer Konvertierung oder Treiber-Entwicklung)
 - Beispiel:

```
char* s = "Hello world!";
cout << reinterpret_cast<double*>(s);
```



Skript: 2.4.4

5-12

- Mit dem Operator `dynamic_cast<T>()` wird dynamisch geprüft, ob der Typ eines polymorphen Klassen-Pointers in eine abgeleitete Klasse konvertiert werden kann. Ist die Umwandlung nicht möglich, wird ein Nullzeiger zurückgegeben.

– Beispiel: `class A {...}; class B : public A {...};`

```
void f(A* a)
{
    B* b = dynamic_cast<B*>(a); // "Downcast"
    ...
}
```

- Eine explizite Typenkonversion in C++ ist oft ein Hinweis auf ein schlechtes Applikations-Design!

Runtime Type Information (RTTI)

- Runtime Type Information (RTTI) erlaubt zur Laufzeit Informationen über (dynamische) Datentypen (v.a. Klassen) zu erhalten.
- Der `typeid()` Operator liefert zu einem Ausdruck oder einem Typnamen ein `type_info` - Objekt zurück, das die (dynamischen) Typ Information des Arguments repräsentiert.
- Zwei `type_info` - Objekte können miteinander auf Gleichheit bzw. Ungleichheit geprüft werden. Mit der Funktion `name()` kann der Name des Typs abgefragt werden.

– Beispiel: `class A {...}; class B : public A {...};`

```
void ShowDiagnostics(A* a)
{
    if (typeid(*a) != typeid(A))        // Typvergleich
        cout << typeid(*a).name();    // "class B"
}
```

Vererbung und Member-Funktionen

- Konstruktoren und Destruktoren
- Member-Verdeckung (Hidding)
- Funktionsüberladung (Function Overloading)
- Funktionsüberschreibung (Function Overriding)
- Virtuelle Funktionen (Virtual Functions)
- Abstrakte oder rein-virtuelle Funktionen (Pure Virtual Functions)



Konstruktor- und Destruktor-Verkettung

- Bei der Instanzierung von Objekten einer abgeleiteten Klasse werden zuerst die Daten-Member der Basisklasse(n) initialisiert. Dies geschieht in der logischen Reihenfolge der Klassenhierarchie durch impliziten Aufruf des Default-Konstruktors der Basisklasse(n).
- Soll bzw. kann für eine Basisklasse nicht der Default-Konstruktor aufgerufen werden, kann bzw. muss der zu verwendende Konstruktor explizit in der Initialisiererliste des Konstruktors der abgeleiteten Klasse angegeben werden.
 - Beispiel: `DerivedClass::DerivedClass(int x) : BaseClass(x) {...};`
- Beim Löschen eines Objektes einer abgeleiteten Klasse wird in umgekehrter Reihenfolge zuerst der Destruktor der abgeleiteten Klasse ausgeführt und danach implizit der Destruktor der Basisklasse(n) aufgerufen.



Klassen-Member Verdeckung (Hidding)

- Wird in einer abgeleiteten Klasse ein Klassen-Member mit dem gleichen Namen wie in der Basisklasse deklariert, so verdeckt der Member der abgeleiteten Klasse denjenigen der Basisklasse. (Dies gilt auch wenn die Typen der Member nicht übereinstimmen!)
- Auf verdeckte Member der Basisklasse kann aber durch Qualifizierung mit dem Namen der Basisklasse trotzdem zugegriffen werden (Scope-Operator).
- Beispiel - Daten-Member Verdeckung:

```
void CSWS_DashedLine:SetDashColor(int iColor) {  
    int point_color = CSWS_Figure::iColor;  
    int dash_color  = iColor;  
    ...  
}
```



Polymorphismus

- Unter Polymorphismus (“Vielgestaltigkeit”) versteht man die Eigenschaft, dass sich eine Funktion (in unterschiedlichen Klassen) unterschiedlich verhalten kann.
- Bei der statischen Polymorphie ermittelt der Compiler zur Compile-Zeit welche Funktion zur Laufzeit aufgerufen werden muss (static binding). In C++ realisiert durch Funktionsüberladung und statische Funktionsüberschreibung.
- Bei der dynamischen Polymorphie wird erst zur Laufzeit ermittelt welcher Funktions-Code ausgeführt werden muss (dynamic oder late binding). In C++ realisiert durch dynamische Funktionsüberschreibung (virtuelle Funktionen).



Funktionsüberladung (Function Overloading)

- Für das Überladen von Member-Funktionen in Vererbungsstrukturen gelten die gleichen Regeln wie für das statische Überladen von Funktionen innerhalb einer Klasse, d.h. überladene Funktionen müssen durch ihre unterschiedliche Signatur unterscheidbar sein.
- Vererbte Member-Funktionen können innerhalb der Klassenhierarchie auch in einer abgeleiteten Klasse überladen werden.



Funktionsüberschreibung (Overriding)

- Von Funktionsüberschreibung (Function Overriding) spricht man, wenn eine Member-Funktion einer Basisklasse in einer abgeleiteten Klasse durch eine Funktion mit gleicher Signatur überschrieben wird.
- Bei der statischen Überschreibung wird die aufzurufende Funktion zur Compile-Zeit ermittelt (static binding). Für "normal" deklarierte Funktionen ist dies der Standard.
- Bei der dynamischen Überschreibung wird die konkret aufzurufende Funktion dynamisch zur Laufzeit-Zeit ermittelt (dynamic binding). Dies wird für Funktionen verwendet, die mit `virtual` deklariert sind.



Polymorphe Variablen

- **Pointer- und Referenz- Variablen können zu verschiedenen Zeiten auf Objekte unterschiedlicher Klassen zeigen. Sie heissen deshalb auch polymorphe Variablen.**
 - Beispiel: `CSWS_Figure* pF1 = new CSWS_Line(15, 15, 25, 25);`
- **Bei polymorphen Variablen unterscheidet man zwischen dem statischen und dem dynamischen Typ:**
 - Der statische Typ ist der Typ, mit der die Variable deklariert wurde.
 - Der dynamische Typ ist der Typ des Objekts, auf das die Variable zur Laufzeit zeigt.
- **Standardmässig werden die Member-Funktionen des statischen Typs einer polymorphen Variablen aufgerufen, auch wenn der dynamische Typ Member-Funktionen mit gleicher Signatur besitzt (static binding).**



5-21

Virtuelle Funktionen

I

- **Wird eine Member-Funktion einer Klasse als `virtual` spezifiziert, so kann diese in abgeleiteten Klassen dynamisch überschrieben werden.**
 - Beispiel:

```
class CSWS_Figure {
public:
    virtual void List(void);
    ...
};
```
- **Damit wird erreicht, dass bei einer polymorphen Variablen der dynamische Typ entscheidet welche Member-Funktion ausgeführt wird, wenn in der abgeleiteten Klasse eine Funktion mit gleicher Signatur existiert (dynamic binding).**



Skript: 4.2.4.3

5-22

Virtuelle Funktionen



- Die überschreibende Member-Funktion in der abgeleiteten Klasse ist implizit wieder virtual, sie kann aber auch explizit als virtual spezifiziert werden.
- In der Implementation einer überschreibenden Member-Funktion kann die virtuelle Funktion der Basisklasse durch die Verwendung des Scope-Operators aufgerufen werden.

```
– Beispiel: Line::List()  
{  
    Figure::List(); // output figure color  
    cout << "P1 (" << x << ", " << y << ")" << endl;  
    cout << "P2 (" << x << ", " << y << ")" << endl;  
}
```



Virtuelle Funktionen



- Eine Klassen mit virtuellen Funktionen wird auch als polymorphe Klassen bezeichnet.
- Die Implementation einer polymorphe Klassen erfolgt durch eine vom Compiler automatisch erzeugte “virtual function table” die intern mit einem “virtual function pointer” (__vfptr) referenziert wird.
- Konstruktoren und static Member-Funktionen können nicht virtuell deklariert werden, da sie nicht für existierende Objekte aufgerufen werden.
- Der Destruktor einer Klasse sollte immer dann virtuell deklariert werden, wenn von einer Basisklasse weitere Klassen abgeleitet werden, für die ein eigener Destruktor implementiert ist.



Abstrakte oder rein virtuelle Funktionen

- Wird eine virtuelle Funktion mit "0 initialisiert", spricht man von einer abstrakten oder rein virtuelle Funktion (Pure Virtual Function).
 - Beispiel:

```
class CSWS_Figure {  
public:  
    virtual void List(void) = 0;  
    ...  
};
```
- Weil rein virtuelle Funktionen in der Klasse in der sie deklariert werden nicht implementiert werden müssen, können sie für die Interface-Spezifikation verwendet werden.
- Ausnahme: Ein rein virtueller Destruktor muss eine Implementation besitzen weil dieser infolge der Destruktor-Verkettung automatisch aufgerufen wird (Linker Error).



Abstrakte Klassen

- Eine Klasse, die mindestens eine rein virtuelle Funktion enthält, wird als abstrakte Klasse bezeichnet.
- Abstrakte Klasse sind nicht instanzierbar und können deshalb nur als Interface-Klassen verwendet werden.
- Eine abstrakte Klasse wird als Basisklasse einer anderen Klasse verwendet, in welcher die rein virtuellen Funktionen implementiert werden. Die abgeleitete Klasse ist damit nicht mehr abstrakt und kann instanziiert werden.
- Die Deklaration von Pointern und Referenzen auf abstrakte Klassen ist möglich.



Mehrfachvererbung

- In C++ können Klassen auch von mehreren direkten Basisklassen abgeleitet werden. Die abgeleitete Klasse erbt dann die Klassen-Member von jeder dieser Basisklassen.
- Bei Mehrfachvererbung können leicht Mehrdeutigkeiten entstehen. So ist der Name eines Klassen-Members bereits mehrdeutig, wenn es in den Basisklassen mehrere Deklarationen dieses Namens gibt (auch bei unterschiedlichen Typen).
- Solche Mehrdeutigkeiten können aufgelöst werden, indem der entsprechende Name vollständig qualifiziert wird.

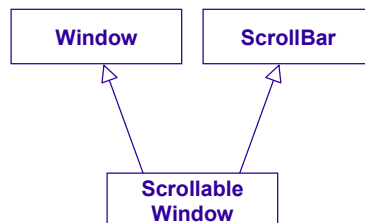


Mehrfachvererbung - Beispiel



```
// Deklaration der Basisklassen
class Window {
public:
    Window(void);
    Move(int x, int y);
    ...
};
class ScrollBar {
public:
    ScrollBar(void);
    Move(int x, int y);
    ...
};

// Deklaration der mehrfach abgeleiteten Klasse
class ScrollableWindow : public Window, public ScrollBar {
public:
    ScrollableWindow(void);
    ...
};
```



Mehrfachvererbung - Beispiel

II

```
// ScrollableWindow Definition
ScrollableWindow ScrollWnd;

// Verschieben des Fensters
ScrollWnd.Move(10, 10);          // Compiler-Error: Mehrdeutigkeit
```

- **Auflösung der Mehrdeutigkeit:**

```
// expliziter Funktionsaufruf mit Scope-Operator
ScrollWnd.Window::Move(10, 10);
ScrollWnd.ScrollBar::Move(10, 10);

// Definition von Move() in der abgeleiteten Klasse
void ScrollableWindow::Move(int x, int y) {
    Window::Move(x, y);
    ScrollBar::Move(x, y);
}
```



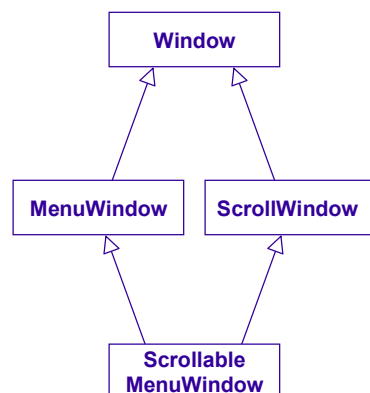
Skript: 4.2.3

5-29

Virtuelle Basisklassen

I

- Bei der Mehrfachvererbung kann es vorkommen, dass dieselbe Klasse mehrfach als indirekte Basisklasse auftritt.
- In diesem Fall ist die Basisklasse entsprechend mehrfach als Teilobjekt in der indirekt abgeleiteten Klasse enthalten.



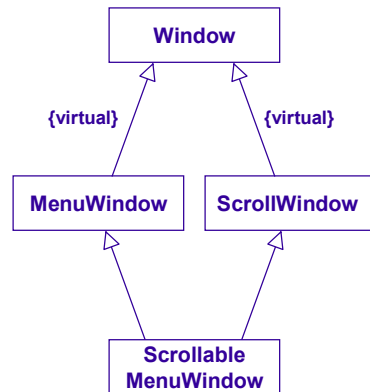
Skript: 4.2.7

5-30

- Mit der `virtual` Deklaration in allen direkten Ableitungen der Basisklasse kann erreicht werden, dass die Basisklasse nur einmal als Teilobjekt in indirekt abgeleiteten Klassen enthalten ist.

- **Beispiel:**

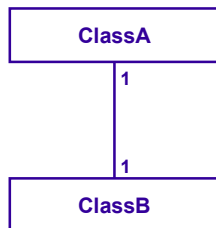
```
class ScrollWindow:  
    virtual public Window {  
        ...  
    };  
class MenuWindow:  
    virtual public Window {  
        ...  
    };
```



Assoziationsbeziehung

- Eine Assoziation wird zur Modellierung einer gleichberechtigten strukturellen Beziehung zwischen zwei Klassen verwendet.
- Weil zur Darstellung einer Assoziationsbeziehung (im Gegensatz zur Vererbung) in C++ keine spezielle Syntax existiert, wird eine Assoziation mit Pointern implementiert.
- Die zur Implementation einer Assoziationsbeziehung notwendigen Pointer-Attribute werden in einem UML Klassendiagramm normalerweise nicht dargestellt.

Assoziationsbeziehung - Implementation I



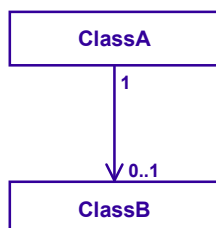
- **Deklaration einer bidirektionalen Assoziation:**

```
class ClassB; // forward declaration of ClassB
class ClassA {
    ClassB* _pClassB;
public:
    void SetClassB(ClassB* pB) {_pClassB = pB;};
    ...
};
class ClassB {
    ClassA* _pClassA;
public:
    void SetClassA(ClassA* pA) {_pClassA = pA;};
    ...
};
```
- **Initialisierung einer bidirektionalen Assoziation:**

```
ClassA A; ClassB B;
A.SetClassB(&B);
B.SetClassA(&A);
```



Assoziationsbeziehung - Implementation II



- **Deklaration einer gerichteten Assoziation:**

```
class ClassB {...};
class ClassA {
    ClassB* _pClassB;
public:
    ClassA(void): _pClassB(0) {...};
    ClassA(ClassB* pB): _pClassB(pB) {...};
    void SetClassB(ClassB* pB) {_pClassB = pB;};
    ...
};
```
- **Initialisierung einer gerichteten Assoziation:**

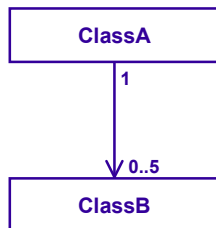
```
ClassB B;

// Variante 1: Multiplizität 0..1
ClassA A1;
A1.SetClassB(&B);

// Variante 2: Multiplizität 1 (exakt)
ClassA A2(&B);
```



Assoziationsbeziehung - Implementation III



- **Assoziation mit definierter Multiplizität n:**

```

class ClassB {...};
class ClassA {
    ClassB* _aClassB[5]; // Beispiel: n = 5
public:
    ClassA(void) {
        for (int i=0; i<5; i++) _aClassB[i] = 0;
    };
    void SetClassB(ClassB* pB, int index) {
        _aClassB[index] = pB;
    };
    ...
};
  
```

- **Initialisierung einer Assoziation mit Multiplizität n:**

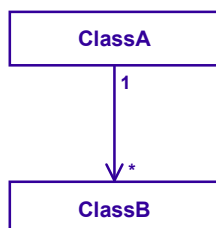
```

ClassB B[5]; // Beispiel: n = 5
ClassA A;

for (int i=0; i<5; i++) A.SetClassB(&B[i], i);
  
```



Assoziationsbeziehung - Implementation IV



- **Gerichtete Assoziation mit * Multiplizität:**

```

class ClassB {...};
class ClassA {
    List<ClassB*> _ClassBList;
public:
    ClassA(void) {...};
    void AddClassB(ClassB* pB) {...};
    ...
};
  
```

- **Initialisierung einer Assoziation mit * Multiplizität:**

```

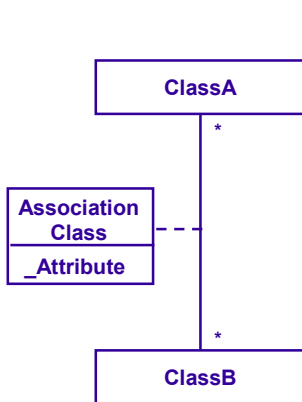
ClassB B0, B1, B2, ...;
ClassA A;
  
```

```

A.AddClassB(&B0);
A.AddClassB(&B1);
A.AddClassB(&B2);
...
  
```



Assoziationsbeziehung - Implementation V



- Deklaration einer Assoziationsklasse:

```
class ClassB {...};
class ClassA {...};

class AssociationClass {
    ClassA* _pClassA;
    ClassB* _pClassB;
    int    _Attribute;
public:
    AssociationClass(ClassA* pA, ClassB* pB)
        : _pClassA(pA), _pClassB(pB) {...};
    void SetAttribute(int i) {_Attribute = i;};
    ...
};
```

- Initialisierung einer Assoziationsklasse:

```
ClassA A; ClassB B;
AssociationClass AB(&A, &B);
AB.SetAttribute(5);
```

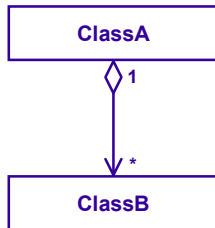


Aggregationsbeziehung

- Eine Aggregation ist eine spezielle Assoziationsbeziehung die verwendet wird um eine strukturelle Ganzes-Teil Beziehung (Whole-Part Relation) zwischen zwei Klassen zu modellieren.
- Die Lebensdauer der Teile eines Aggregates sind hierbei unabhängig von der Lebensdauer des Aggregates selber.
- Die Navigierbarkeit und spezifische Multiplizität einer Aggregation wird analog wie für eine Assoziationsbeziehung implementiert.
- Die zur Implementation der Aggregationsbeziehung notwendigen Pointer-Attribute werden in einem UML Klassendiagramm normalerweise nicht dargestellt.



Aggregationsbeziehung - Implementation



- **Deklaration einer (gerichteten) Aggregation:**

```
class ClassB {...};
class ClassA {
    List<ClassB*> _ClassBList;
public:
    ClassA(void) {...};
    void AddClassB(ClassB* pB) {...};
    ...
};
```

- **Initialisierung einer (gerichteten) Aggregation:**

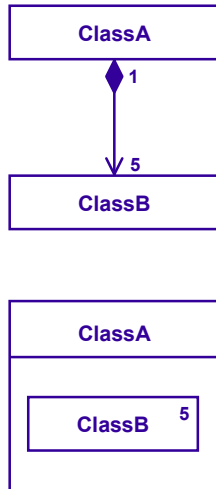
```
ClassB B0, B1, B2, ...;
ClassA A;

A.AddClassB(&B0);
A.AddClassB(&B1);
A.AddClassB(&B2);
...
```

Kompositionsbeziehung

- Eine Komposition ist eine spezielle Assoziationsbeziehung die verwendet wird um eine existenzabhängige, strukturelle Ganzes-Teil Beziehung zwischen zwei Klassen zu modellieren.
- Durch die Existenzabhängigkeit wird die maximal Lebensdauer der Teile eines Aggregates beschränkt auf die Lebensdauer des Aggregates.
- Die Navigierbarkeit und spezifische Multiplizität einer Komposition wird analog wie für eine Assoziationsbeziehung implementiert.
- Die zur Implementation der Kompositionsbeziehung notwendigen Pointer-Attribute werden in einem UML Klassendiagramm normalerweise nicht dargestellt.

Kompositionsbeziehung - Implementation I



- Deklaration einer Komposition - Variante 1:

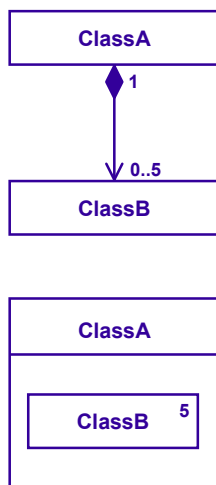
```

class ClassA {
    class ClassB {...};
    ClassB _aClassB[5]; // Beispiel: n = 5
public:
    ClassA(void) {...};
    ~ClassA(void) {...};

    void OperationA(void)
    {
        for (int i=0; i<5; i++)
        {
            _aClassB[i].OperationB();
        }
    };
    ...
};
  
```



Kompositionsbeziehung - Implementation II



- Deklaration einer Komposition - Variante 2:

```

class ClassA {
    class ClassB {...};
    ClassB* _pClassB[5]; // Beispiel: n = 5
    int _Index;
public:
    ClassA(void): _Index(0) {
        for (int i=0; i<5; i++) _pClassB[i] = 0;
    };

    void AddClassB(void) {
        _pClassB[_Index++] = new ClassB;
    };

    ~ClassA(void) {
        for (int i=0; i<_Index; i++) {
            delete _pClassB[i];
        }
    };
};
  
```

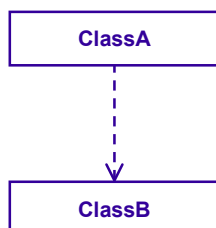


Abhängigkeitsbeziehung

- Eine Abhängigkeit (Dependency oder Using Relation) beschreibt eine allgemeine, d.h nicht strukturelle, gerichtete Beziehung zwischen zwei Klassen.
- Eine Abhängigkeitsbeziehung wird oft verwendet, wenn eine Klassenbibliothek oder eine allgemeine Utility-Klasse in einer Applikation eingesetzt wird.

Abhängigkeitsbeziehung - Implementation I

- Abhängigkeitsbeziehung (Signatur):

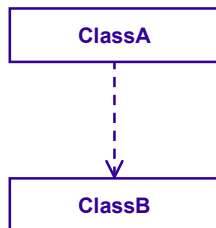


```
class ClassB {
public:
    ClassB(void);
    void OperationB(void);
    ...
};

class ClassA {
public:
    ClassA(void);
    void OperationA(ClassB* pB) {
        pB->OperationB();
        ...
    };
    ...
};
```

Abhängigkeitsbeziehung - Implementation II

- Abhängigkeitsbeziehung (lokales Objekt):



```
class ClassB {
public:
    ClassB(void);
    void OperationB(void);
    ...
};

class ClassA {
public:
    ClassA(void);
    void OperationA(void) {
        ClassB B;           // local B object
        B.OperationB();
        ...
    };
    ...
};
```

Teil 6

C++ Exceptions

6-1

Teil 6 - Inhalt

- Fehlerbehandlung
- Exception - Konzept
- C++ Exception Syntax

Fehlerbehandlung

- Das prinzipielle Problem bei der Fehlerbehandlung besteht darin, dass dort wo ein Fehler entdeckt wird meist nicht genügend Informationen vorhanden sind um den Fehler zu behandeln.
- Die Idee der Ausnahmebehandlung besteht darin, dass eine Funktion, die einen auftretenden Fehler nicht behandeln kann, eine Ausnahme (exception) wirft um einer übergeordneten Funktion die angemessene Fehlerbehandlung zu überlassen.
- Exceptions sind kein Mittel zur Kontrolle von Ausnahmesituationen die von aussen gesteuert werden (Interrupts, Signals, Messages ...).
- Jeder Fehler der bereits vom Compiler entdeckt werden kann, muss zur Laufzeit nicht mit Exceptions behandelt werden oder aufwendig mit dem Debugger gesucht werden!



Exception - Konzept

- Mit dem ANSI C++ Standard wurde ein allgemeiner Mechanismus zur getrennten Erkennung und Behandlung von Laufzeitfehlern in C++ integriert.
- Der Mechanismus beruht auf einem Ausdruck, der im Fehlerfall eine Ausnahme (exception) wirft (throw), die von einem entsprechenden Exception-Handler aufgefangen (catch) und behandelt wird.
- Diese Methode des Exception-Handlings erlaubt eine klare Trennung des eigentlichen Applikations-Codes und des Codes zur Behandlung von Fehlern.



C++ Exception Syntax

I

- Zum Exception-Handling sind in C++ folgende Keywords definiert:
 - `try` definiert den Gültigkeitsbereich, innerhalb dessen Fehler mit `catch` abgefangen und behandelt werden.
 - `throw` "wirft" eine entsprechende Ausnahme, wenn ein Fehler auftritt.
 - `catch` dient dazu eine Ausnahme "einzufangen" und den Fehler zu behandeln.
- Beim Eintreten einer Applikations- oder Runtime-Exception (z.B. ein File- oder Memory Allokierungsfehler) wird ein Exception-Objekt erzeugt und anschliessend alle übergeordneten Blöcke geordnet verlassen, bis die Ausnahme behandelt werden kann.
- Die Unterscheidung der auftretenden Fehler im `catch` Statement wird über den Typ des geworfenen Ausdrucks vorgenommen. Dabei können auch benutzerdefinierte Typen (z.B. Fehlerklassen) verwendet werden.



Skript: 6.3

6-5

C++ Exception Syntax

II

- Einfaches Beispiel mit Integer-Typ als Exception:

Deklaration: `void copy(const char* src, const char* dest)`

```
{
    ifstream in(src);
    if (!in.good()) throw 0;

    ofstream out(dest);
    if (!out.good()) throw 1;

    char c;
    while (in.get(c)) out.put(c);
}
```

...

Main:

```
try {
    copy(file0, file1);
}
catch(int i) {
    cerr << "Fehler beim Oeffnen des "
        << (i == 0) ? "Quellfiles" : "Zielfiles";
}
```



Skript: 6.3

6-6

Definition von Fehlerklassen

- Fehlerklassen sind normale Klassen und unterstehen deshalb den normalen Regeln von C++ für die Implementation von Klassen.
- Für verschiedene Fehlerkategorien können verschiedene hierarchisch gegliederte (vererbte) Fehlerklassen definiert werden.
- Beispiel - "Stack Range Error" für Stack Template-Klasse:

```
class CErrorStackRange
{
public:
    CErrorStackRange (int err) : err_code(err) {};
    int GetErrorCode(void) {return err_code;};

private:
    int err_code;
};
```



Auslösen einer Exception (throw)

- Nach dem Erkennen eines Applikationsfehlers wird eine Exception gezielt mit dem "throw" Statement ausgelöst.
- Beispiel - Pop() Funktion für Stack Template-Klasse:

```
template <class T>
T Stack<T>::Pop()
{
    // check for available stack element
    if ( numElems == 0 )
    {
        // "Stack Underflow Range Error", throw error code 1010
        throw CErrorStackRange(1010);
    }

    // pop element from stack
    return elements[ --numElems ];
}
```



Behandeln einer Exception (catch)

- Das Abfangen und die Behandlung einer Exception erfolgt durch eine Liste von “catch” Statements.
- Beispiel - Fehlerbehandlung für Stack Template-Klasse:

```
...
for (int i=0; i<intStack.GetElementNum(); i++) {
    try {
        cout << "Stack Element: " << intStack.Pop() << endl;
    }
    catch ( CErrorStackRange& err ) {
        cout << "Stack Range Error (Error Code: "
            << err.GetErrorCode() << ")" << endl;
    }
    catch (...) {
        cout << "Undefined Stack Error" << endl;
        throw; // re-throw exception for upper level handling
    }
}
```



Behandlung nicht aufgefangener Exceptions

- Nicht abgefangene Exceptions führen zu einem aussergewöhnlichen Programmabbruch. Zu diesem Zweck wird die Funktion `terminate()` aufgerufen, die wiederum die Funktion `abort()` aufruft. Dadurch wird das Programm sofort beendet, ohne z.B. die Destruktoren statisch erzeugter Objekte aufzurufen.
- Um allfällige Fehler einer Software so zu vermeiden (nicht richtig geschlossene Files, DB's etc.) kann die `terminate()` Funktion mit `set_terminate()` durch eine eigene Terminate-Funktion ersetzt werden.
- Alternativ dazu kann in der Funktion `main()` auch ein Top-Level `catch(...)` Handler implementiert werden.



Exception-Spezifikation für Funktionen

- Da die Exceptions einer Funktion einen wesentlichen Bestandteil der Interface-Definition darstellen, kann die Funktionsdeklaration mit einer Liste der geworfenen Exceptions ergänzt werden.

– Beispiele: `// Funktion kann beliebige Exceptions werfen`
`void f();`

`// Funktion wirft bestimmte Exceptions`
`void f() throw ( Exception1, Exception2 );`

`// Funktion wirft keine Exceptions`
`void f() throw ();`

Teil 7

C++ Standardbibliothek

7-1

Teil 7 - Inhalt

- Allgemeine Konzepte
- Strings
- I/O Streams
- File I/O Streams
- Utilities
- Standard Template Library (STL)

Allgemeine Konzepte

- **Pre-ANSI Standard Implementationen von C++ Compiler** beinhalteten nur die von ANSI C übernommene C Library
- **Nachteil: unterschiedliche Implementation von allgemeinen Hilfsklassen durch verschiedene C++ Compiler Hersteller (z.B. String-Klassen)**
- **Die C++ Standardbibliothek bildet einen integralen Bestandteil des ANSI Standards, wodurch jeder C++ Standard konforme Compiler diese Library unterstützen muss.**



Aufgabenbereiche der Standard Library

- **Die Standardbibliothek kann in verschiedene Aufgabenbereiche unterteilt werden:**
 - Language Support: dynamische Speicherverwaltung, RTTI
 - Diagnostics: Exceptions, Assertions, Fehlercodes
 - Strings: String-Verarbeitung
 - I/O-Streams: Ein- und Ausgabe-Streams, Formatierung, Manipulatoren
 - Numerics: komplexe Zahlen und numerische Arrays
 - General Utilities: allgemeine verwendbare Klassen und Funktionen
 - Localization: Lokalisierte Darstellung von Daten (Datum, Zeit, Währung, etc.)
 - Containers (STL): Objektverwaltung (vector, deque, list, set, map)
 - Iterators (STL): Iteratoren für Container
 - Algorithms (STL): Algorithmen für Container (Suchen, Sortieren, etc.)
- **Die C Library ist ebenfalls Bestandteil der C++ Standardbibliothek**



Standard Library Namespace “std”

- Alle C++ Standard Library Funktionen (inkl. C Runtime Library) sind im Namespace `std` deklariert.
- Header Files der C++ Standard Library werden ohne Filename Extension eingebunden.
 - Beispiel: `#include <iostream>`
- Dateinamen von C Header Files werden mit einem vorangestellten “c” übernommen.
 - Beispiel: `#include <cstdlib>`
- Aus Kompatibilitätsgründen sind die “normalen” C Header Files ebenfalls vorhanden, allerdings sind diese dann nicht im Standard Namespace (`std`) definiert.
 - Beispiel: `#include <stdlib.h>`



Standard Library Exceptions



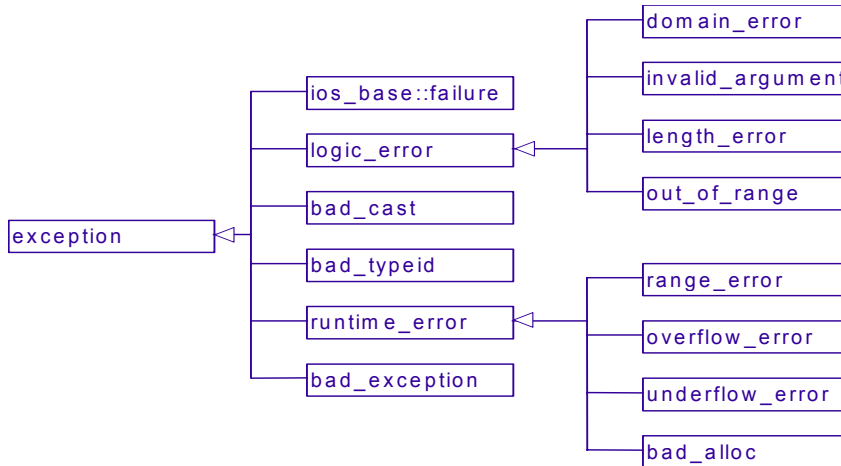
- Exceptions können durch das C++ Runtime System oder durch Funktionen der Standardbibliothek ausgelöst werden.
- Aus Kompatibilitätsgründen und durch die Heterogenität der Standard Library, werden Exceptions uneinheitlich verwendet.
- Verwendung der Basisklasse `exception` für alle vom System erzeugten Exceptions (Header File `<exception>`).
- Von dieser Basisklasse werden für verschiedene Fehler-Kategorien weitere Standard-Exceptions abgeleitet (Header File `<stdexcept>`).
- Vordefinierte Exceptions können durch eigene Exception-Klassen erweitert werden.



Standard Library Exceptions

II

- Klassenhierarchie der Standard Library Exceptions:



Skript: 7.1.6

7-7

Standard Library Strings

I

- Die Standard Library definiert im Header File `<string>` mehrere String-Klassen:
 - `basic_string` ist die String-Template-Klasse, welche die ganze String-Funktionalität unabhängig von einem speziellen Zeichentyp implementiert.
 - `string` ist die konkret instanzierte Template-String-Klasse für Strings mit Zeichen vom Typ `char`. (Deklaration: `typedef basic_string<char> string;`)
 - `wstring` ist eine weitere konkret instanzierte Template-String-Klasse für Strings mit Zeichen vom Typ `wchar_t`. Diese Strings dienen der Internationalisierung. (Deklaration: `typedef basic_string<wchar_t> wstring;`)
- Die so definierten String-Klassen können wie vordefinierte Datentypen verwendet werden, wobei die Buffer-Grösse falls notwendig automatisch erhöht wird.



Skript: 7.4

7-8

Standard Library Strings

II

- **Konstruktion, Kopieren und Löschen von Strings:**

- `string s` erzeugt einen Leer-String
- `string s(s1)` erzeugt einen String als Kopie eines andern Strings
- `string s(10, ' ')` erzeugt einen String mit 10 Leerzeichen
- `string s("hallo")` erzeugt einen C++ String aus einem C-String
- ...
- `~string()` Destruktor, löscht den String und gibt den Speicher frei

- **Nicht modifizierende String-Funktionen:**

- `size()` liefert die Anzahl Zeichen im String
- `length()` liefert die Anzahl Zeichen im String (äquivalent zu `size()`)
- `empty()` testet ob der String leer ist
- `max_size()` liefert die max. Anzahl Zeichen die der String enthalten kann
- `reserve(anz)` reserviert String Speicherplatz für mindestens Anzahl Zeichen
- `capacity()` liefert die Anzahl Zeichen, die ohne Reallozierung vom String aufgenommen werden können



Skript: 7.4.1 und 7.4.2 (Demo-7-01 - C++ Strings)

7-9

Standard Library Strings

III

- **Modifizierende String-Funktionen:**

- `operator+=( )` anhängen eines Strings am Ende eines bestehenden Strings
- `operator+( )` verketteten von Strings
- `operator=( )` zuweisen von Strings
- `assign( )` zuweisen von Strings (versch. Varianten)
- `insert( )` einfügen von Strings (versch. Varianten)
- `replace( )` ersetzen von Strings (versch. Varianten)
- `erase( )` löschen von Zeichen (versch. Varianten)
- `resize(anz, ' ')` setzen der Länge eines Strings (versch. Varianten)
- ...

- **String-Funktionen zum suchen:**

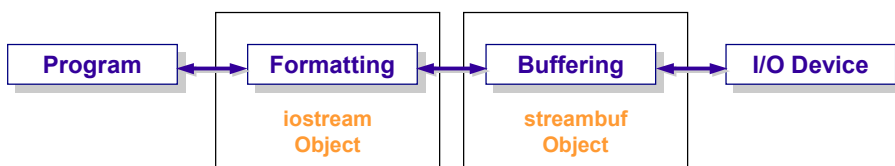
- `find( )` sucht ersten Teil-String (versch. Varianten)
- `rfind( )` sucht letzten Teil-String (versch. Varianten)
- `find_first_of( )` sucht erstes Zeichen einer Menge (versch. Varianten)
- `find_last_of( )` sucht letztes Zeichen einer Menge (versch. Varianten)
- ...



Skript: 7.4.3 und 7.4.4

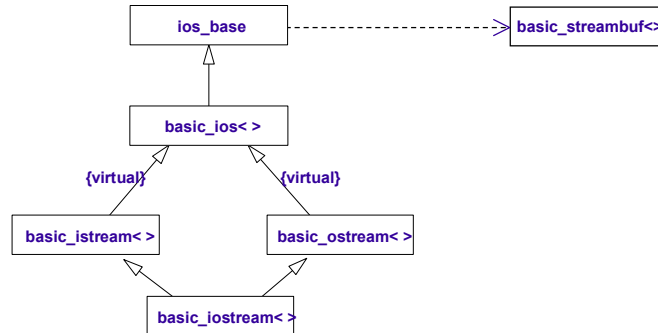
7-10

- **String Vergleichsoperatoren:**
 - `==, !=, <, >, <=, >=` Standard Vergleichsoperatoren
 - `compare()` testet zwei Strings auf Gleichheit (versch. Varianten)
- **String-Funktionen für den direkten Zeichenzugriff:**
 - `at(index)` Zeichenzugriff an der Stelle index (prüft Indexgrenzen)
 - `operator[index]` Zeichenzugriff an der Stelle index (prüft Indexgrenzen nicht)
 - `c_str()` konvertiert ein String-Objekt in einen C-String (Null terminiertes Char Array)
- **String-Funktionen für Ein- und Ausgabe:**
 - `operator<<()` ausgeben eines Strings auf dem Output-Stream
 - `operator>>()` einlesen einer Zeichenfolge vom Input-Stream bis zum ersten Trennzeichen (Whitespace).
 - `getline()` einlesen einer Zeichenfolge vom Input-Stream bis zum ersten Newline-Zeichen (versch. Varianten)



- Die Ein- und Ausgabe erfolgt in C++ mittels sogenannter Streams. Dies gilt nicht nur für Tastatur/Bildschirm sondern auch für Files.
- Ein Stream kann als “Datenstrom” verstanden werden von dem Daten gelesen oder hinein geschrieben werden können.
- Streams sind als Klassen implementiert, wobei man zwischen Input Streams (istream) für die Eingabe und Output Streams (ostream) für die Ausgabe unterscheidet.

- Die Klassen zur Implementation von Streams sind in folgender Klassenhierarchie organisiert:



- ios_base**
Basisklasse welche die allgemeinen Eigenschaften aller Stream-Klassen unabhängig von einem speziellen Zeichentyp definiert (z.B. Status- und Format-Flags).
- basic_ios<T>**
 - Diese von der Basisklasse abgeleitete Template-Klasse definiert die allgemeinen Eigenschaften der Stream-Klassen die vom speziellen Zeichentyp abhängig sind (z.B. Definition des zum Zeichentyp gehörenden Datenpuffers).
 - `ios` - ist die konkret instanzierte Template-Klasse für den Zeichentyp `char`.
(Deklaration: `typedef basic_ios<char> ios;`)

- **basic_streambuf<>**
 - Template-Klasse zur Implementation der gepufferten Lese- und Schreiboperationen auf einen spezifischen "Datenkanal".
 - **streambuf** - ist die konkret instanzierte Template-Klasse für den Zeichentyp char. (Deklaration: `typedef basic_streambuf<char> streambuf;`)
- **basic_istream<> bzw. basic_ostream<>**
 - Template-Klassen für die Implementation der Input- bzw. Output- spezifischen Stream-Funktionen.
 - **istream** bzw. **ostream** - sind die konkret instanziierten Template-Klassen für den Zeichentyp char. (Deklaration: `typedef basic_istream<char> istream;` bzw. `typedef basic_ostream<char> ostream;`)
- **basic_iostream<>**
 - **iostream** ist die char Klassen-Instanz für die kombinierte Stream Ein- und Ausgabe. (Deklaration: `typedef basic_iostream<char> iostream;`)



- **Für die Standard Ein- und Ausgabe Streams gibt es die folgenden vordefinierten globalen Objekte:**
 - **cin** - Stream-Objekt für den Standard-Eingabekanal, entspricht der C-Variable stdin. (Definition: `istream cin;`)
 - **cout** - Stream-Objekt für den Standard-Ausgabekanal, entspricht der C-Variable stdout. (Definition: `ostream cout;`)
 - **cerr** - Stream-Objekt des Standard-Ausgabekanals für Fehler, entspricht stderr. Die Ausgaben auf cerr werden nicht gepuffert. (Definition: `ostream cerr;`)
 - **clog** - Analog cerr, die Ausgaben erfolgen jedoch gepuffert. In C ist dafür keine entsprechende Variable definiert. (Definition: `ostream clog;`)
- **Durch include des Header Files <iostream> werden alle notwendigen Stream-Klassen und die vordefinierten Stream-Objekte in eigene Programme eingebunden.**



I/O Streams: Funktionen

- **Standard Stream-Operatoren:**

Die Operatoren >> und << sind für alle vordefinierten Datentypen überladen und bilden die Standardoperationen für die Stream Ein- und Ausgabe. Für die Ein- und Ausgabe eigener Datentypen werden mit Vorteil ebenfalls diese Operatoren überladen.

- **Stream-Funktionen für die Eingabe:**

- `get()` liefert das nächste gelesene Zeichen oder EOF zurück (versch. Varianten).
- `getline()` liest eine ganze Zeile bis zum EOF oder '\n' Zeichen.
- `read()` liest eine maximale Anzahl Zeichen in einen C-String.
- `peek()` liefert nächstes Zeichen aus dem Stream oder EOF, ohne es auszulesen. (Beim nächsten Lesen wäre das Zeichen also immer noch vorhanden.)
- `gcount()` liefert die Anzahl gelesener Zeichen des letzten Lesebefehls.
- ...

- **Stream-Funktionen für die Ausgabe:**

- `put()` gibt das übergebene Zeichen als nächstes Zeichen aus.
- `write()` schreibt eine spezifizierte Anzahl Zeichen eines C-Strings.
- `flush()` leert den Ausgabepuffer, indem alle Zeichen ausgegeben werden.



I/O Streams: Stream State

- **Zum Erkennen einer Fehlersituation verfügen Streams über einen Zustand der durch Fehlerzustands-Flags beschrieben wird:**

- `ios::goodbit` alles in Ordnung (keine Fehler-Bit gesetzt)
- `ios::eofbit` end of file - das Ende der Daten ist erreicht
- `ios::failbit` Fehler - der letzte Vorgang wurde nicht korrekt abgeschlossen (z.B. ungültiges Zeichen für int Typ eingelesen)
- `ios::badbit` fataler Fehler - Zustand des Streams ist nicht definiert (z.B. Disk voll)
- **Ist eines der drei Fehler-Flags gesetzt, wird der entsprechende Stream für die weitere Ein- oder Ausgabe gesperrt. Zurücksetzen erfolgt mit der Funktion `clear()`.**

- **Funktionen zum Setzen und Abfragen des Stream-Zustandes:**

- `good()` true wenn alles in Ordnung ist (`ios::goodbit` gesetzt)
- `eof()` true bei end of file (`ios::eofbit` gesetzt)
- `fail()` true bei Fehler (`ios::failbit` oder `ios::badbit` gesetzt)
- `bad()` true bei fatalem Fehler (`ios::badbit` gesetzt)
- `rdstate()` liefert die Kombination der gesetzten Zustands-Flags
- `clear()` löscht alle gesetzten *Fehlerzustands-Flags*
- `exceptions()` aktiviert Exception-Mechanismus bei Änderungen des Zustands



I/O Streams: Manipulatoren

- Manipulatoren sind spezielle Stream Funktionen, deren Ausgabe eine Manipulation am Stream durchführt. Damit kann z.B. das Ausgabeformat entsprechend eingestellt werden.
- Im Header File `<iostream>` sind die folgenden häufig verwendeten Manipulatoren definiert:
 - `endl` Newline ausgeben und Ausgabepuffer leeren
 - `flush` Ausgabepuffer leeren
 - `ws` Whitespace bei der Eingabe ignorieren
- Eigene Stream-Manipulatoren können ebenfalls definiert werden.



7-19

I/O Streams: Formatdefinitionen

I

- Zur Definition des Ein- bzw. Ausgabeformates werden in der Stream-Klasse `ios_base` verschiedene Bitfelder für Format-Flags definiert.
- Zum Setzen oder Abfragen der Stream Format-Flags können prinzipiell zwei Möglichkeiten verwendet werden:
 - Direkter Format-Flag Zugriff mit den Funktionen `setf()`, `unsetf()` und `flags()`.
 - Zugriff mittels den allg. Manipulatoren `setiosflags()` bzw. `resetiosflags()` oder mit speziell definierten Manipulatoren für bestimmte Format-Flags. Diese Format-Flag Manipulatoren werden durch das Header File `<iomanip>` eingebunden.
- Beispiele (alle Varianten sind äquivalent):

```
// Setzen der Flags "ios::showbase" und "ios::hex"
cout.setf(ios::showbase | ios::hex); cout << 255 << endl;
cout << setiosflags(ios::showbase | ios::hex) << 255 << endl;
cout << showbase << hex << 255 << endl;

cout.flags(0); // alle Format-Flags zurücksetzen
```



Skript: 7.5.6

7-20

- **Definition der Feldbreite, Füllzeichen und Ausrichtung:**

- `width()` Funktion zum Setzen der Feldbreite
- `fill()` Funktion zum Setzen des Füllzeichens (Default: ' ')
- `ios::left` Format-Flag für linksbündig
- `ios::right` Format-Flag für rechtsbündig
- `ios::internal` Format-Flag für Vorzeichen linksbündig und Wert rechtsbündig

- **Vorzeichen und Grossbuchstaben numerischer Werte:**

- `ios::showpos` Format-Flag um positives Vorzeichen auszugeben
- `ios::uppercase` Format-Flag für grosse Buchstaben in numerischen Werten

- **Definition des Zahlensystems:**

- `ios::oct` Format-Flag für octale Ein- und Ausgabe
- `ios::dec` Format-Flag für dezimale Ein- und Ausgabe (default)
- `ios::hex` Format-Flag für hexadezimale Ein- und Ausgabe
- `ios::showbase` Format-Flag um das Zahlensystem zu kennzeichnen



- **Definition der Gleitkommanotation:**

- `ios::fixed` Format-Flag für Dezimaldarstellung von Gleitkommazahlen
- `ios::scientific` Format-Flag für Exponentialdarstellung von Gleitkommazahlen
- `precision()` Funktion zum Setzen oder Abfragen der Genauigkeit von Gleitkommazahlen, wobei die Werte entsprechend gerundet werden.

- **Allgemeine Format-Flags:**

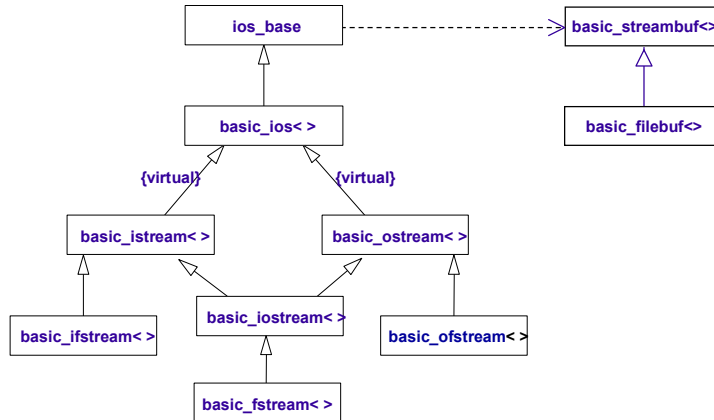
- `ios::skipws` Format-Flag um führende Whitespaces zu überlesen
- `ios::unitbuf` Format-Flag um den Ausgabebuffer nach jeder Ausgabeoperationen zu leeren



File I/O Streams

I

- Streams können auch verwendet werden um Files zu Lesen und zu Schreiben. Zu diesem Zweck wird die Stream-Klassenhierarchie um spezifische File-Stream-Klassen erweitert:



C++

Skript: 7.5.7

7-23

File I/O Streams

II

- `basic_ifstream<>`**
 - Abgeleitete Template-Klasse für Input File Stream spezifische Funktionen.
 - `ifstream` - ist die konkret instanzierte Template-Klasse für den Zeichentyp `char`. (Deklaration: `typedef basic_ifstream<char> ifstream;`)
- `basic_ofstream<>`**
 - Abgeleitete Template-Klasse für Output File Stream spezifische Funktionen.
 - `ofstream` - ist die konkret instanzierte Template-Klasse für den Zeichentyp `char`. (Deklaration: `typedef basic_ofstream<char> ofstream;`)
- `basic_fstream<>`**
 - `fstream` - `char` Klassen-Instanz für die kombinierte Ein- und Ausgabe auf File-Streams. (Deklaration: `typedef basic_fstream<char> fstream;`)
- `basic_filebuf<>`**
 - Abgeleitete Template-Klasse für File Buffer spezifische Funktionen.
 - `filebuf` - ist die konkret instanzierte Template-Klasse für den Zeichentyp `char`. (Deklaration: `typedef basic_filebuf<char> filebuf;`)

C++

Skript: 7.5.7

7-24

- Durch Ableitung der File-Stream-Klassen von “normalen” Streams können die Funktionen und Operatoren für die eigentliche Ein- und Ausgabe von Streams (<<, >>, get(), getline(), put() etc.) auch zum Lesen und Schreiben von Files verwendet werden.
- Das Gleiche gilt für die von den “normalen” Stream-Klassen bekannten Fehlerzustands-Flags und Formatdefinitionen, welche sämtliche auch für die File Ein- und Ausgabe verwendbar sind.
- Durch include des Header Files <fstream> werden alle notwendigen File-Stream-Klassen in eigene Programme eingebunden.
- Leider fehlt der Zugriff auf allgemeine File Informationen wie z.B. File-Grösse oder File-Attribute. Für diese Informationen müssen (plattform-abhängige) System-Calls verwendet werden.



File I/O Streams: Files öffnen

- Ein Vorteil bei der Verwendung von File Streams besteht darin, dass Files nicht explizit geöffnet werden müssen, sondern dies im Konstruktor des File Stream Objektes “automatisch” erfolgt.
 - Beispiel: `ofstream txtfile("myfile.txt");`
- Zur genauen Spezifikation einer File-Operation sind die folgenden File-Flags definiert:
 - `ios::in` Lesen (Default bei ifstream)
 - `ios::out` Schreiben (Default bei ofstream)
 - `ios::app` Anhängen (append)
 - `ios::ate` am File Ende positionieren (at end)
 - `ios::trunc` alten File Inhalt löschen (truncate)
 - `ios::binary` File im binären Modus öffnen
 - Beispiel: `ofstream txtfile("myfile.txt", ios::app | ios::binary);`



File I/O Streams: Files schliessen

- Analog zum Öffnen erfolgt das Schliessen eines Files im Destruktor des File Stream Objektes. Ein File muss deshalb nicht explizit geschlossen werden, sondern dies wird “automatisch” beim Löschen des Objektes durchgeführt.

- **Beispiel:**

```
// statisches Erzeugen eines File Streams
ifstream infile("input.txt");

// dynamisches Erzeugen eines File Streams
ofstream* p_outfile = new ofstream("output.txt");

...
// explizites Löschen des dynamisch erzeugten File Streams
delete p_outfile;

// Beim Verlassen des Scoops wird der statisch erzeugte File Stream
// automatisch gelöscht und das dazugehörige File geschlossen.
```



File I/O Streams: Positionierung im File

- Für die freie Positionierung des aktuellen File-Pointers sind folgende Funktionen und File-Flags definiert:

- `seekg()` Funktion zum Setzen einer Lese-Position (`seek get`)
- `tellg()` Funktion welche die aktuelle Lese-Position liefert (`tell get`)
- `seekp()` Funktion zum Setzen einer Schreib-Position (`seek put`)
- `tellp()` Funktion welche die aktuelle Schreib-Position liefert (`tell put`)
- `ios::beg` File-Flag für eine Position relativ zum Dateianfang (`begin`)
- `ios::cur` File-Flag für eine Position relativ zur aktuellen Position (`current`)
- `ios::end` File-Flag für eine Position relativ zum Dateiende (`end`)

- **Beispiel: Positionierung des File-Pointers in einem File**

```
ios::pos_type pos = file.tellg(); // aktuelle Position merken
file.seekp(pos); // absolute Positionierung
file.seekp(10, ios::beg); // rel. Position nach File-Anfang
file.seekp(20, ios::cur); // rel. Position von aktueller Position
file.seekp(-10, ios::end); // rel. Position 10 Zeichen vor File-Ende
```



File I/O Streams: Standardkanäle umleiten

- Durch die Zuweisung der Buffer von Stream-Objekten können die Standard Ein- und Ausgabe-Kanäle in Files umgelenkt werden:

```
– Beispiel: void main()
{
    cout << "Zeile vor Umlenkung" << endl;    // screen

    // merken des stream buffers und anlegen eines files
    stringstream strm_buffer = cout.rdbuf();
    ofstream redirect_file("redirect.txt");

    // Umlenkung durchführen
    cout.rdbuf(redirect_file.rdbuf());
    cout << "Zeile während Umlenkung" << endl; // file

    // Umlenkung zurückstellen
    cout.rdbuf(strm_buffer);
    cout << "Zeile nach Umlenkung" << endl;    // screen
}
```



Standard Library Utilities



- Im Header File <utility> der Standard Library werden neben allgemein verwendbaren Funktionen der Datentyp `pair` deklariert:

- Datentyp `pair`

– Die Standardbibliothek verwendet an einigen Stellen Wertepaare (z.B. Map, Multimap), für welche ein entsprechender Template Datentyp `pair` deklariert wird:

```
template<class T, class U> struct pair {
    T first;                // first data member
    U second;               // second data member
    pair();                 // default constructor
    pair(const T& x, const U& y); // init constructor
};
```



Standard Library Utilities

II

- Im Header File `<algorithm>` der Standard Library werden die folgenden allgemein verwendbaren Hilfsfunktionen definiert:
- **Hilfsfunktionen**
 - Die `min()` und `max()` Template-Funktionen können verwendet werden um das Minimum bzw. Maximum zweier Werte eines beliebigen Datentyps zu ermitteln (Voraussetzung: Definition eines entsprechenden Vergleichsoperators `<`).
 - Die `swap()` Template-Funktion kann zum Vertauschen zweier Werte eines beliebigen Datentyps verwendet werden.



Skript: 7.2.2

7-31

Standard Library Utilities

III

- In den Header Files `<cstddef>` und `<cstdlib>` werden, wie in den entsprechenden C Header Files, häufig verwendete Konstanten, Typen und Makros definiert:
- **`<cstddef>`**
 - `NULL` Zeigerwert "nicht definiert" (In C++ ist dieser Wert: `int 0`)
 - `size_t` Datentyp für Grössenangaben
 - `ptrdiff_t` Datentyp für Zeiger-Abstände
 - `offsetof()` Offset einer Komponente in einer Struktur oder einer Union
- **`<cstdlib>`**
 - `exit(int status)` Programm beenden
 - `abort()` Programm abbrechen
 - `atexit(void (*func)())` Funktion welche durch `exit()` aufgerufen werden soll



Skript: 7.2.2.3

7-32

- Im Header File `<limits>` werden die (plattformabhängigen) numerischen Limiten der verschiedenen Datentypen durch die Template Klasse `numeric_limits` zur Verfügung gestellt:

- Deklaration von `numeric_limits`

```
template<class T> class numeric_limits {  
public:  
    static T max() throw();  
    static T min() throw();  
    static const bool is_integer;  
    static const bool is_modulo;  
    static const bool is_signed;  
    ...  
};
```

- Beispiel:

```
cout << "max(long): " << numeric_limits<long>::max() << endl;
```



Weitere Standard Library Klassen

- Utility-Klasse für Auto-Pointer
- String-Stream-Klassen für die Ein- und Ausgabe von C++ Strings
- Bibliothek für die Verarbeitung von komplexen Zahlen `<complex>` und für numerische Arrays `<valarray>` zur Vektor und Matrizen-Berechnung
- Template-Klasse für die Verarbeitung von Bitsets
- Klassen für die Internationalisierung



Standard Template Library (STL)

- Die Standard Template Library bildet einen zentralen Bestandteil der C++ Standard Library (entwickelt von der Firma HP).
- Das Design der STL entspricht nicht dem typisch objektorientierten Lösungsansatz, weil Daten und die dazugehörigen Funktionen absichtlich voneinander getrennt implementiert sind.
- Der Vorteil besteht in der Entkoppelung von Operationen und Daten, sodass diese beliebig miteinander kombiniert werden können, was die Library sehr universell einsetzbar macht.
- Durch die Implementation mittels Template-Klassen kann die STL mit beliebigen (auch eigenen) Datentypen verwendet werden und die Performance bleibt trotzdem sehr gut.



Standard Template Library WWW Ressourcen

- Online Reference Manuals und weitere Ressourcen:
 - www.cs.rpi.edu/projects/STL/htdocs/stl.html (STL Online Reference)
 - www.sgi.com/tech/stl/index.html (SGI STL Programmer's Guide)
 - www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html (STL Newbie Guide)
 - www.mathcs.sjsu.edu/faculty/horstman/safestl.html ("Safe" STL Implementation)
 - www.cs.brown.edu/people/jak/proglang/cpp/stitut/tut.html (STL Tutorial)
 - www.infosys.tuwien.ac.at/Research/Component/tutorial/prwmain.htm (STL Tutorial)



STL - Allgemeines Konzept

- **Die STL setzt sich aus den folgenden Bestandteilen zusammen:**
 - **Container** werden verwendet um Mengen von Daten eines bestimmten Datentyps zu organisieren und zu verwalten.
 - **Iteratoren** werden verwendet um eine Menge von Daten zu durchlaufen (iterieren).
 - **Algorithmen** werden verwendet um Mengen als Ganzes oder nur bestimmte Daten in diesen Mengen zu bearbeiten.
- **Die Funktionsweise dieser STL Bestandteile besteht im wesentlichen in der Verwaltung von Daten durch Container, wobei deren Bearbeitung durch Algorithmen erfolgt und die Iteratoren als Bindeglied zwischen Containern und Algorithmen eingesetzt werden.**



STL - Container

- **Container werden verwendet um Mengen von Objekten oder Werte vordefinierter Datentypen zu verwalten, wobei je nach Anwendung verschiedene Container verwendet werden können:**
 - Bei **sequentiellen Containern** besitzt jedes Datenelement eine bestimmte Position, welche durch den Ort des Einfügens definiert wird. Die STL kennt `vector`, `deque` und `list` als sequentielle Container.
 - **Assoziative Container** sortieren ihre Datenelemente automatisch anhand einer Vergleichsfunktion, die von aussen definiert werden kann. Da diese Container als Binärbäume implementierten sind, ist insbesondere die Suche von Elementen sehr schnell. Die STL kennt `set` bzw. `multiset` und `map` bzw. `multimap` als assoziative Container.
- **Die Container Klassen können auch rekursiv verwendet werden (z.B. Liste von Vektoren).**



STL - vector Container

- Der `vector` Container ist als dynamisches Array implementiert. Auf die einzelnen Datenelemente kann mit Hilfe des überladenen Index-Operators direkt zugegriffen werden (random access).



- Das Zeitverhalten des Vektors ist für die Mutation von Daten am Ende optimal, für das Einfügen und Löschen von Elementen am Anfang und im Bereich der Mitte jedoch sehr langsam.
- Der Speicherplatz eines Vektors wird dynamisch erweitert, indem eine Reallozierung durchgeführt wird. Dabei werden die Elemente mittels Copy-Konstruktor in den neuen Speicher kopiert und die alten Elemente mit dem Destruktor gelöscht (Zeitverhalten!).



STL - vector Container Beispiel

```
#include <iostream>
#include <vector>
...
// int vector definieren
vector<int> iMenge;

// int vector füllen
for ( int iElement = 10; iElement <= 100; iElement += 10 ) {
    iMenge.push_back( iElement );
}

// int vector auslesen
for ( int iCounter = 0; iCounter < iMenge.size(); iCounter++ ) {
    cout << iCounter << ". Element: "<< iMenge[ iCounter ] << endl;
}

// Ausgabe: 0. Element: 10
//           1. Element: 20
//           2. Element: 30
...
```



STL - deque Container

- Der deque Container ("double ended queue") ist als dynamisches Array implementiert, welches in beide Richtungen wachsen kann. Der Zugriff auf die einzelnen Datenelemente erfolgt mittels dem Index-Operators (random access).



- Das Zeitverhalten ist hier für Änderungen am Anfang und am Ende optimal, hingegen ist die Deque für Einschieben und Löschen von Datenelementen im Bereich der Mitte nicht ideal. Durch die aufwendigere Speicherorganisation ist dieser Container etwas langsamer als der Vektor.



STL - deque Container Beispiel

```
#include <iostream>
#include <deque>
...
// int deque definiern
deque<int> iDeque;

//int deque füllen
for ( int iCounter = 1; iCounter <= 10; iCounter++ ) {
    iDeque.push_back( iCounter );
    iDeque.push_front( iCounter );
}

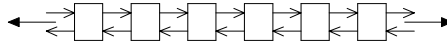
// int deque auslesen
for ( iCounter = 0; iCounter < iDeque.size(); iCounter++ ) {
    cout << iDeque[ iCounter ] << " ";
}
cout << endl;

// Ausgabe: 10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10
```



STL - list Container

- Der `list` Container ist als doppelt verkettete Liste implementiert. Der direkte Zugriff auf Datenelemente ist damit nicht möglich, weshalb eine sequentielle Suche mit linearem Zeitverhalten notwendig ist. (Die Liste besitzt dem entsprechend keinen Index-Operator.)



- Das Zeitverhalten der Liste ist in Bezug auf Einfügen von Elementen am Anfang und am Ende sehr gut. Das Einfügen und Löschen von Datenelementen im Bereich der Mitte der Liste ist sehr schnell, da aufgrund der Pointer-Verknüpfung nur sehr wenige Elemente bearbeitet werden müssen.



STL - list Container Beispiel

```
#include <iostream>
#include <list>
...
// int liste definieren
list<int> iListe;

// int liste füllen
for ( int iCounter = 1; iCounter <=10; iCounter++ ) {
    iListe.push_back( iCounter );
}

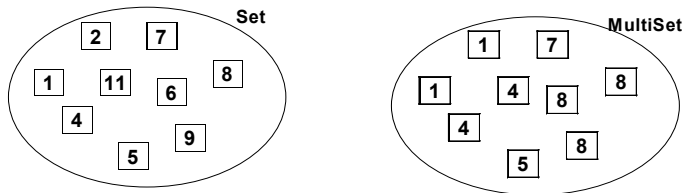
// int liste auslesen
while ( ! iListe.empty() ) {
    cout << iListe.front() << ' ';
    iListe.pop_front();
}
cout << endl;

// Ausgabe:   1 2 3 4 5 6 7 8 9 10
```



STL - set und multiset Container

- Ein `set` ist ein assoziativer Container in dem die Datenelemente jeweils nur einmal vorkommen dürfen und die automatisch nach ihrem Wert sortiert werden. In einem `multiset` können die Elemente auch mehrfach vorkommen.

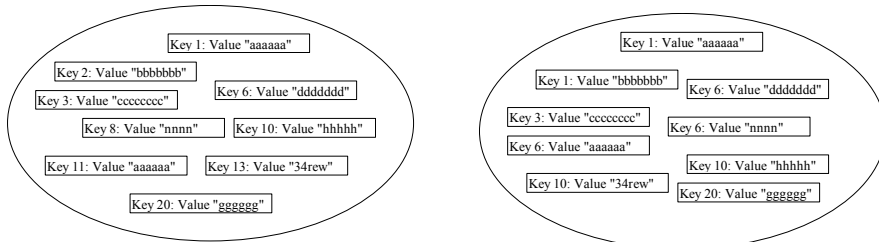


- Aufgrund der Sortierung und der Implementation als binäre Baumstruktur erfolgt vor allem das Suchen von Elementen sehr schnell. Das Zeitverhalten für Einfügen und Löschen von Datenelementen ist ähnlich wie bei einer Liste.



STL - map und multimap Container

- `map` bzw. `multimap` Container verwalten Schlüssel-Wert-Paare als Datenelemente. Die Elemente werden nach dem Schlüssel sortiert, wobei in einem `Map` jeder Schlüssel nur einmal, bei einem `Multimap` mehrmals vorkommen kann.



- Da `Map` und `Multimap` Container als binäre Baumstruktur implementiert sind, ist das Zeitverhalten analog wie für `Set` Container.



STL - Container Adapter

- Mit Container-Adaptoren werden die Standard Container an spezifische Bedürfnisse angepasst und das Interface entsprechend vereinfacht. Die STL definiert die folgenden Adapter-Klassen:
 - **Queue** wird zur Implementation einer FIFO-Queue ("First In First Out") verwendet, wobei intern ein Deque-Container verwendet wird.
 - **Priority-Queue** definiert eine Queue deren Datenelemente nach ihrer Priorität wieder ausgelesen werden. Für die Implementation wird intern ein Vector-Container verwendet.
 - **Stack** wird zur Implementation eines LIFO-Speichers ("Last In First Out") verwendet. Für die interne Implementation wird intern ein Deque-Container verwendet.



7-47

STL - Container Eigenschaften

I

- **Gemeinsame Eigenschaften aller Container-Klassen (Auszug):**
 - `ContTyp c` Erzeugt einen leeren Container ohne Datenelemente
 - `ContTyp c1(c2)` Erzeugt einen Container c1 als Kopie des Containers c2
 - `~ContTyp` Destruktor, löscht alle Datenelemente und gibt den Speicher frei
 - `c.size()` liefert die aktuelle Anzahl Datenelemente im Container
 - `c.max_size()` liefert die maximal mögliche Anzahl Datenelemente
 - `c.empty()` prüft ob der Container leer ist
 - `c1 ==` bzw. `!= c2` prüft ob c1 gleich bzw. ungleich c2 ist
 - `c1 <` bzw. `> c2` vergleicht ob c1 kleiner bzw. grösser c2 ist
 - `c1 <=` bzw. `>= c2` vergleicht ob c1 kleiner od. gleich bzw. grösser od. gleich c2 ist
 - `c1 = c2` weist c1 die Datenelemente von c2 zu
 - `c.begin()` liefert einen Iterator für das erste Datenelement
 - `c.end()` liefert einen Iterator für Position hinter dem letzten Datenelement
 - `c.insert(pos,e)` fügt an der Position (pos) das Datenelement (e) in c ein
 - `c.erase(pos)` löscht das Datenelement an der Position (pos)
 - `c.clear()` löscht alle Datenelemente (leert den Container)



Skript: 7.3.2

7-48

- **Anforderungen damit ein (eigener) Datentyp als Datenelement eines Containers verwendet werden kann:**
 - Da Container intern Kopien der Datenelemente anlegen und auch Kopien zurückgegeben werden, muss jeder Datentyp einen **Copy-Konstruktor** zur Verfügung stellen.
 - Da Container und Algorithmen Zuweisungen verwenden, muss jeder Datentyp einen **Zuweisungsoperator** bereitstellen.
 - Da beim Löschen der Container auch die internen Kopien der Datenelemente gelöscht werden, muss ein entsprechend implementierter **Destruktor** aufgerufen werden können.
 - Für einige Funktionen sequentieller Container muss zusätzlich ein **Default-Konstruktor** implementiert sein.
 - Bei assoziativen Containern muss für die automatische Sortierung der Datenelemente grundsätzlich der **Operator <** definiert sein.
 - Für zahlreiche Funktionen (hauptsächlich zum Suchen von Datenelementen) muss der **Vergleichsoperator ==** definiert sein.



- **Da die STL Container die Datenelemente default-mässig “by value” verwalten, d.h. eine lokale Kopie der Elemente erstellen, ist dies bei Verwendung der Container mit Pointer-Elementen besonders zu berücksichtigen:**
 - Vergleichsoperationen werden mit Pointer-Adressen durchgeführt, was meist nicht der gewünschten Funktionalität entspricht (--> eigene Vergleichsfunktion implementieren).
 - Externe Verweise auf Datenelemente können ungültig werden, weil bei verschiedenen Operationen die Elemente intern verschoben oder umkopiert werden.
- **Bei Verwendung der STL Container mit “by value” Datenelementen können ebenfalls Probleme entstehen, wenn z.B. Objekte einer abstrakten Basisklasse verwaltet werden sollen (Compiler-Error).**
- **Im weiteren ist zu beachten, dass die Funktionen und Operatoren der STL keine Fehlerprüfungen durchführen und somit keine Exceptions ausgelöst werden! (vgl. “Safe” STL Implementation)**



STL - Iteratoren

I

- Iteratoren sind Objekte, die verwendet werden um Container zu durchlaufen (iterieren). Sie ermöglichen eine allgemeingültige Verbindung zwischen Algorithmen und Containern herzustellen.
- Jede Container-Klasse definiert eine Container spezifische Iterator-Klasse mit einem einheitlichen Interface, das Algorithmen erlaubt verschiedene Container entsprechend einheitlich zu verwenden. (vgl. Iterator Design Pattern)
- Folgende Operatoren bilden die Grundfunktionalität aller Iteratoren:
 - `iterator::operator*()` liefert das Element an der aktuellen Position des Iterators
 - `iterator::operator++()` setzt den Iterator zum nächsten Datenelement
 - `iterator::operator==()` prüft ob zwei Iteratoren auf das gleiche Datenelement zeigen
 - `iterator::operator!=()` prüft ob zwei Iteratoren auf versch. Datenelemente zeigen



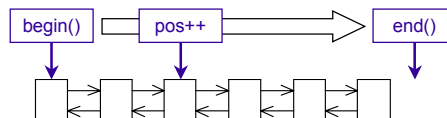
Skript: 7.3.3

7-51

STL - Iteratoren

II

- Iteratoren werden für die Schleifensteuerungen bei der iterativen Bearbeitung von Container-Elementen verwendet. Dabei wird ein Iterator-Objekt als Pointer auf das aktuelle Datenelement benutzt.



- Iterator-Objekte für Container-Beginn und -Ende können für jeden Container mittels folgenden Funktionen erzeugt werden:
 - `c.begin()` liefert einen Iterator für das erste Element
 - `c.end()` liefert einen Iterator für die Position hinter dem letzten Element
- Für Iteratoren müssen keine spezifischen Header-Files eingebunden werden, weil diese innerhalb der jeweiligen Container definiert sind.



Skript: 7.3.3

7-52

```
#include <set>

// int set definieren
set<int> iSet;

// set füllen
for ( int iCounter = 1; iCounter <= 10; iCounter++ ) {
    iSet.insert( iCounter );          // set 1. mal füllen
}
// set füllen - mit teilweise gleichen Werten
for ( iCounter = 6; iCounter <= 15; iCounter++ ) {
    iSet.insert( iCounter );          // set 2. mal füllen
}

// set mit iterator auslesen
set<int>::iterator pos;
for ( pos = iSet.begin(); pos != iSet.end(); pos++ ) {
    cout << *pos << ' ';
}
cout << endl;

// Ausgabe: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```



- Neben den Grundfunktionen können Iteratoren weitere Funktionen besitzen. Aufgrund dieser spezifischen Eigenschaften werden Iteratoren in verschiedene Kategorien eingeteilt:
 - **Forward Iteratoren** - Diese Iteratoren können nur vorwärts (operator++) iteriert werden.
 - **Bidirectional Iteratoren** - Diese Iteratoren können vorwärts (operator++) und rückwärts (operator--) iteriert werden.
 - **Random Access Iteratoren** - Diese Iteratoren können ebenfalls vorwärts und rückwärts iteriert werden, haben aber zusätzlich weitere Funktionen implementiert um z.B. direkt auf einzelne Datenelemente zugreifen zu können (operator[]). Die Container-Klassen vector und deque erzeugen Iteratoren dieser Kategorie.



- STL-Algorithmen bieten eine grosse Anzahl von Funktionen zum Finden, Vertauschen, Sortieren, Kopieren und Modifizieren von Containern.
- Die Algorithmen sind keine Funktions-Member von Container-Klassen, sondern globale Funktionen die mit Hilfe von Iteratoren gesteuert werden und damit Zugriff auf die Elemente der Container erhalten.
- Alle Algorithmen können auch nur auf einen bestimmten Bereich (range) von Datenelementen angewendet werden.
- Viele Algorithmen können durch die Übergabe einer benutzer-definierten Funktion für spezifische Aufgaben spezialisiert werden.



- Alle STL Algorithmen sind im Header File `<algorithm>` definiert.
- **Nicht modifizierende Algorithmen**
 - `find()` sucht ein bestimmtes Datenelement
 - `for_each()` ruft für jedes Datenelement eine readonly Operation auf
 - `count()` zählt Datenelemente die eine bestimmte Bedingung erfüllen.
- **Modifizierende Algorithmen**
 - `copy()` kopiert einen Bereich
 - `transform()` modifiziert jedes Element eines Bereichs
- **Löschende Algorithmen**
 - `remove()` löscht bestimmte Elemente
 - `unique()` löscht aufeinanderfolgende Duplikate



- **Mutierende Algorithmen**

- `reverse()` kehrt die Reihenfolge von Elementen um
- `random_shuffle()` bringt die Reihenfolge der Elemente durcheinander

- **Sortierende Algorithmen**

- `sort()` sortiert Elemente
- `partial_sort()` sortiert die ersten n Elemente

- **Algorithmen für sortierte Bereiche**

- `includes()` prüft ob alle Elemente einer Teilmenge enthalten sind
- `merge()` fasst die Elemente zweier Bereiche zusammen

- **Numerische Algorithmen**

- `accumulate()` verknüpft Elemente
- `partial_sum()` verknüpft ein Element jeweils mit allen Vorgängern