Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 10

10. Ausgewählte Komponenten der Standardbibliothek

ANSI/ISO-C++-Standardbibliothek - Allgemeines

• Grundsätzliches

◇ Mit dem **ANSI/ISO-Standard** für C++ ist neben der eigentlichen Sprache auch eine dazugehörige **Standard-Bibliothek** genormt worden.

◇ Die C++-Standardbibliothek verwendet eine Reihe neuerer erst **relativ spät in die Sprache übernommener Sprachmittel** (z.B. Namespaces, Exception Handling, Templates, Datentyp `bool`, neue Typwandlungs-Operatoren). Daher ist sie u. U. noch nicht in allen vorhandenen C++-Systemen vollständig realisiert.

◇ Die Standardbibliothek wurde nicht von Grund auf neu entwickelt, sondern fasst zahlreiche schon vorher weitgehend unabhängig voneinander entstandene Komponenten und Ansätze zusammen, die soweit wie möglich aneinander angepasst, sowie um zahlreiche Vorschläge und Mechanismen ergänzt und erweitert wurden.
Zusätzlich wurde auch die **ANSI-C-Standardbibliothek übernommen**.
Diese ist somit - z. T. geringfügig angepasst - Bestandteil der ANSI-C++- Standardbibliothek.
Die **C++-Standardbibliothek** stellt daher **kein homoges Gebilde** dar, sondern besteht aus **verschiedenen Teilen**, die weitgehend unabhängig voneinander sind, die aber doch in einigen Details miteinander verbunden sind.

◇ Alle in der C++-Standardbibliothek definierten **globalen Namen** - auch die in der integrierten ANSI-C-Standardbibliothek definierten - sind im **Namensbereich `std`** definiert.

• Konventionen für Header-Dateien :

◇ Header-Dateien werden in der `#include`-Anweisung **ohne Namens-Extension** angegeben, z. B.

```
#include <iostream>
```

◇ Auch die **Header-Dateien der C-Standardbibliothek** werden **ohne Extension** angegeben, allerdings wird ihnen zur Unterscheidung **ein c vorangestellt**, z.B.

```
#include <cstdlib>        // C-Header-Datei stdlib.h
```

◇ Die Angabe der Header-Dateinamen ohne Extension bedeutet nicht, dass Header-Dateien generell keine Namens-Extension mehr besitzen.
Vielmehr ist die Umsetzung des in der `#include`-Anweisung verwendeten Dateinamens in den tatsächlich im jeweiligen C++-System verwendeten Dateinamen implementierungsabhängig.
Z.B. kann

```
#include <iostream>
```

vom Preprozessor umgesetzt werden in

```
namespace std
{
    #include <iostream.h>
}
```

◇ Aus **Kompatibilitätsgründen** können für die **C-Header-Dateien** auch die **"normalen" Namen mit Extension** verwendet werden.
Dies gilt in der Praxis - wenn auch nicht im Standard festgelegt - ebenfalls für C++-spezifische Header-Dateien, die bereits vor der Definition der Standardbibliothek verwendet wurden, z. B.

```
#include <iostream.h>
```

Werden **Header-Datei-Namen mit Extension** verwendet, so befinden sich die definierten globalen Namen im **globalen Namensbereich** (nicht `std`)

ANSI/ISO-C++-Standardbibliothek - Überblick

- **Bestandteile der Standardbibliothek** :

  ◇ **Sprachunterstützungs-Bibliothek (*Language support library*)**

  Funktionen und Typen, die zur Realisierung bestimmter Sprachelemente benötigt werden, wie z.B.
  Funktionen, die während der Ausführung von C++-Programmen implizit aufgerufen werden, Definition zugehöriger verwendeter Typen, Festlegung implementierungsabhängiger Eigenschaften der Standardtypen
  (u.a. Klassen-Template **`numeric_limits`**)

  ◇ **Diagnose-Bibliothek (*Diagnostics library*)**

  Komponenten zum Entdecken und Melden von Fehler-Bedingungen,
  u.a. Hierarchie von Fehlerklassen (***exception classes***), Macro **`assert`**, Fehlernummern-Macros

  ◇ **Utility-Bibliothek (*General utility library*)**

  Ergänzende Hilfsmittel, u.a. Datentypen für Wertepaare (Klassen-Template **`pair`**) und "sichere" Pointer (Klassen-Template **`auto_ptr`**), Default-Allokator-Klasse **`allocator`** (Allokator-Objekte repräsentieren Speichermodelle)

  ◇ **String-Bibliothek (*Strings library*)**

  Klassen-Template **`basic_string`** und konkrete Realisierungen (Template-Klassen) **`string`** und **`wstring`**

  ◇ ***Localization library***

  Komponenten zur portablen Unterstützung nationaler Eigenheiten
  (z.B. Zeichendarstellung und Zeichensatz, Datums-, Währungs-, Zahlformate), u.a. Klasse **`locale`**

  ◇ **STL -** ***Standard Template Library***

  Bibliothek zur Verwaltung und Bearbeitung von Mengen beliebigen Typs.
  Sie besteht aus 3 Teilen :

  ▻ ***Containers library***
  Container-Klassen (Objekte dieser Klassen speichern und verwalten andere Objekte)

  ▻ ***Iterators library***
  Iterator-Klassen (Iteratoren dienen zum Zugriff zu einzelnen Elementen einer Objektmenge, insbesondere zu Elementen von Containern)

  ▻ ***Algorithms library***
  "freie" Funktionen zur Realisierung von Algorithmen, mit denen Mengen und Elemente in Mengen bearbeitet werden können.

  In diesem Teil der Bibliothek sind auch Komponenten enthalten, die nicht direkt auf Objekt-Mengen bezogen sind, z. B.
  - Klassen-Template **`bitset`** zur Verwaltung und Bearbeitung von Bit-Feldern fester aber beliebig wählbarer Größe (*Containers library*)
  - "freie" Funktions-Templates zur Realisierung häufig benötigter Hilfsfunktionen, wie z. B. `max()`, `min()`, `swap()` (*Algorithms library*)

  ◇ **Numerische Bibliothek (*Numerics library*)**

  Klassen-Templates für komplexe Zahlen (**`complex`**) und numerische Felder, wie z.B. Vektoren und Matrizen (**`valarray`**), ...
  Erweiterung der numerischen Funktionen der C-Standard-Bibliothek durch Überladen (freie Funktionen)

  ◇ **Stream-I/O-Bibliothek (*Input/Output library*)**

  Stream-Klassen (einschließlich File- u. String-Streams), Standard-Stream-Objekte, Stream-Buffer-Klassen, Standard-Manipulatoren

ANSI/ISO-C++-Standardbibliothek – Überblick Header-Dateien

Language support library	
Typen	**<cstddef>**
Implementierungsabhängige Eigenschaften	**<limits>, <climits>, <cfloat>**
Programm-Start u. Beendigung	**<cstdlib>**
Dynamische Speicherverwaltung	**<new>**
Typidentifikation	**<typeinfo>**
Ausnahmebehandlung	**<exception>**
weitere Laufzeitunterstützung	**<cstdarg>, <csetjmp>, <ctime>, <csignal>, <cstdlib>**
Diagnostics library	
Exception-Klassen	**<stdexcept>**
Assertions	**<cassert>**
Fehlernummern	**<cerrno>**
General utilities library	
Utility-Komponenten	**<utility>**
Funktions-Objekte	**<functional>**
Speicher	**<memory>**
Datum und Zeit	**<ctime>**
Strings library	
Zeicheneigenschaften (*char traits*)	**<string>**
String-Klassen	**<string>**
NUL-terminierte Zeichenfolgen	**<cctype>, <cwctype>, cstring>, <cwchar>, <cstdlib>**
Localization library	
Locales-Komponenten u. Kategorien	**<locale>**
C-Bibliotheks-Locales	**<clocale>**
Containers library	
Folgen (sequences)	**<deque>, <list>, <queue>, <stack>, <vector>**
Assoziative Container	**<map>, <set>**
Bitset	**<bitset>**
Iterators library	
Iteratoren	**<iterator>**
Algorithms library	
Diverse Algorithmen	**<algorithm>**
C-Bibliotheks-Algorithmen	**<cstdlib>**
Numerics library	
Komplexe Zahlen	**<complex>**
Numerische Felder	**<valarray>**
Numerische Operationen	**<numeric>**
Numerische C-Bibliothek	**<cmath>, <cstdlib>**
Input/Output library	
Forwärts-Deklarationen	**<iosfwd>**
Standard-Stream-Objekte	**<iostream>**
I/O-Stream-Basisklassen	**<ios>**
Stream-Buffer	**<streambuf>**
Formatierung und Manipulatoren	**<istream>, <ostream>, <iomanip>**
String-Streams	**<sstream>, <cstdlib>**
File-Streams	**<fstream>, <cstdio>, <cwchar>**

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (1)

• Grundsätzliches

◇ Die ANSI-C++-Standardbibliothek stellt auch einige Exception-Klassen zur Verfügung.
Diese bilden eine **Klassen-Hierarchie**, die von der Klasse **`exception`** abgeleitet ist.

◇ Einige dieser Exception-Klassen gehören zur **Sprachunterstützungsbibliothek** (*Language Support Library*).
Sie werden von **Sprachkonstrukten** verwendet (z.B. von `new, dynamic_cast, typeid`)

◇ Die übrigen Exception-Klassen werden von der **Standardbibibliothek selbst** benutzt. Sie sind – mit einer Ausnahme – in der **Diagnosebibliothek** (*Diagnostics Library*) definiert.
Objekte dieser Exception-Klassen können auch im **Anwender-Code** geworfen werden.
Darüber hinaus lassen sie sich auch als **Basisklassen** für **selbstdefinierte Exception-Klassen** einsetzen.

• Die Exception-Basisklasse `exception`

◇ Bestandteil der Sprachunterstützungsbibliothek.

◇ Sie ist die Basisklasse für alle Objekte, die als Exceptions von einigen Sprachausdrücken sowie von Bestandteilen der Standardbibliothek geworfen werden können.

◇ Ihre **öffentliche Schnittstelle** ist in der **Headerdatei `<exception>`** wie folgt **definiert :**

```
class exception
{
  public :
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
  private :
    // ...
};
```

◇ Die virtuelle **Methode** `const char* what() const` dient zur Ermittlung einer Information über die **Ursache** einer Exception. Der von ihr für die Klasse `exception` zurückgelieferte C-String ist **implementierungsabhängig**.
Häufig wird in den von `exception` abgeleiteten Bibliotheks-Klassen diese Methode jeweils – ebenfalls implementierungsabhängig – überschrieben.
Auch in selbst definierten Exception-Klassen, die von einer der Bibliotheks-Exception-Klassen abgeleitet sind, kann sie – und muss sie gegebenenfalls – in geeigneter Weise überschrieben werden.
Bei den Exception-Klassen, deren Instanzen in Sprachausdrücken geworfen werden können, ist die zurückgelieferte Information häufig direkt in der Methode `what()` implementiert, bei den von Komponenten der Standardbibliothek verwendeten Klassen lässt sich diese Information bei der Exception-Objekt-Erzeugung dem Konstruktor übergeben.

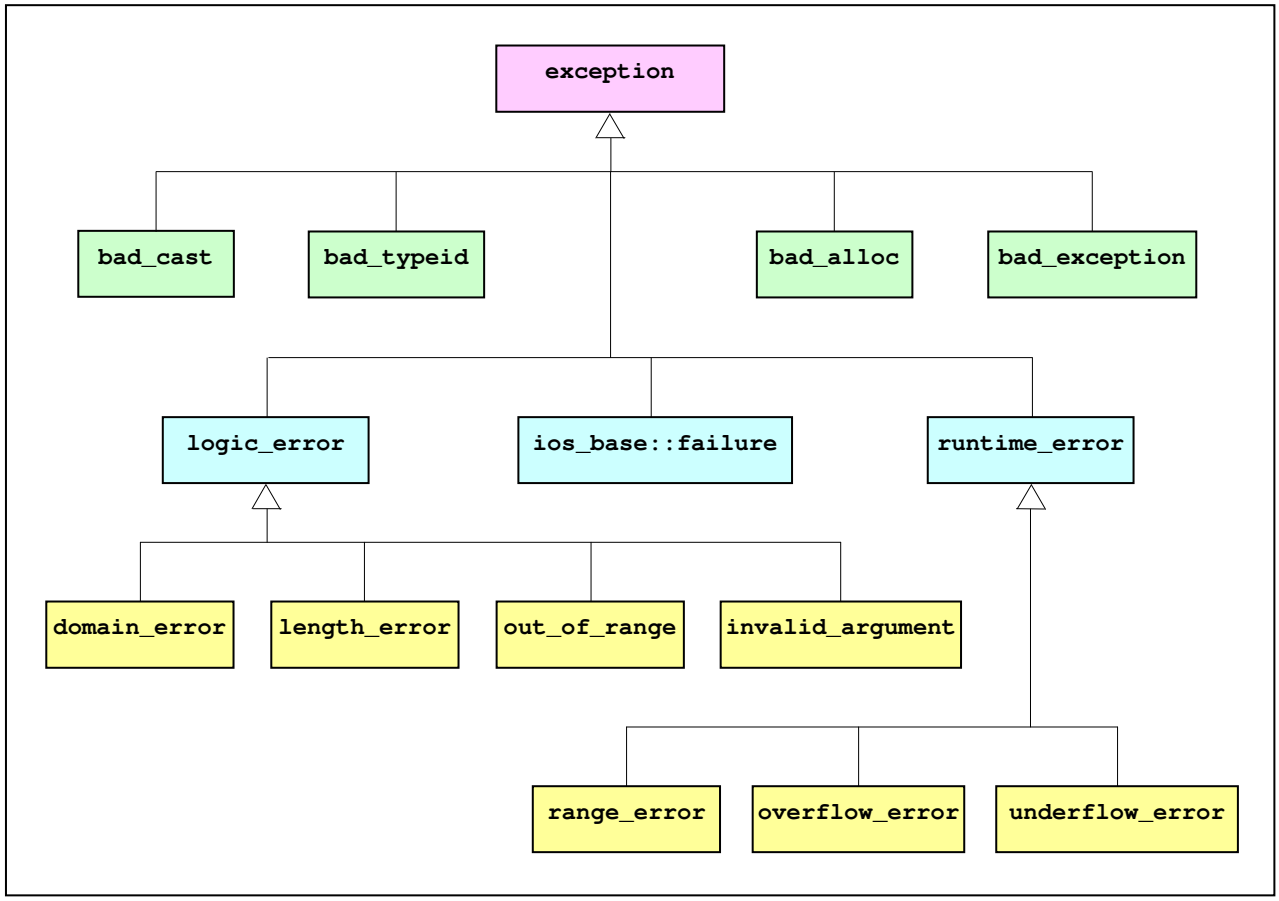
→ portable Auswertung der Exception-Information :

```
try
{
  // ...
}
catch (const exception& ex)
{
  cerr << endl << ex.what() << endl;
  // ...
}
```

◇ Die **Exception-Spezifikation `throw()`** der verschiedenen Memberfunktionen der Klasse `exception` legen fest, dass diese **selbst keine Exceptions werfen** dürfen.

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (2)

- **Hierarchie der Bibliotheks-Exception-Klassen**



- **Die Exception-Klassen zur Sprachunterstützung**

Klassenname	Headerdatei	geworfen von ... bei
bad_cast	<typeinfo>	Operator **dynamic_cast** bei ungültigem Referenz-Cast
bad_typeid	<typeinfo>	Operator **typeid** bei dereferenziertem NULL-Pointer im Operandenausdruck
bad_alloc	<new>	Operator **new** bei fehlgeschlagener Allokation (falls entsprechend implementiert)
bad_exception	<exception>	unter gewissen Umständen durch die Funktion `unexpected()`, wenn durch eine Funktion eine **nicht vorgesehene Exception** (Exception-Spec) geworfen wird

◇ Die **öffentliche Schnittstelle** für alle Exception-Klassen dieser Gruppe entspricht der Schnittstelle von `exception` :

- ▻ Konstruktor ohne Parameter
- ▻ Copy-Konstruktor
- ▻ Zuweisungsoperatorfunktion
- ▻ virtueller Destruktor
- ▻ virtuelle Memberfunktion `const char* what() const throw()` zur Ermittlung von Information über die Exception-Ursache

In einigen Implementierungen existiert darüber hinaus zusätzlich ein **Konstruktor**, dem eine **genauere Information** über die **Exception-Ursache** (bei Visual-C++ z. B. als C-String) übergeben werden kann.

ANSI/ISO-C++-Standardbibliothek : Standard-Exception-Klassen (3)

- **Die in der Standardbibliothek eingesetzten Exception-Klassen**

  ◇ Aus dieser Gruppe sind **drei Klassen direkt** von der Klasse **`exception`** **abgeleitet**, zwei in der Diagnosebibliothek und eine in der IO-Bibliothek. Von den Klassen der Diagnosebibliothek sind weitere abgeleitete Klassen definiert.

  ◇ Klasse **`logic_error`**

  ▻ definiert in der Headerdatei **`<stdexcept>`** (Diagnosebibibliothek)

  ▻ dient als **Basisklasse** für **weitere Exceptionklassen**, die den Auftritt **logischer Fehler** charakterisieren.
  Unter logischen Fehlern werden solche Fehler verstanden, die in der Programmlogik liegen und damit prinzipiell schon **vor dem Programmstart** oder durch das Überprüfen von Funktionsparametern gefunden werden können.

  Die folgenden von dieser Klasse **abgeleiteteten Klassen** sind ebenfalls in der Headerdatei **`<stdexcept>`** definiert:

  ▻ Klasse **`domain_error`**
  Objekte dieser Klasse werden beim Auftritt von Domänenfehlern (Wertebereichsfehlern) geworfen.

  ▻ Klasse **`length_error`**
  Objekte dieser Klasse werden geworfen, wenn versucht wird, ein Objekt zu erzeugen, dessen Größe die maximal zulässige Größe überschreiten würde (z. B. durch einige Memberfunktionen der Bibliotheksklasse `string`).

  ▻ Klasse **`out_of_range`**
  Objekte dieser Klasse können geworfen werden, wenn ein Funktionsparameter außerhalb des zulässigen Bereichs liegt (vor allem bei Positionsangaben, z. B. durch einige Memberfunktionen der Bibliotheksklasse `string` zur Auswahl von Stringelementen)

  ▻ Klasse **`invalid_argument`**
  Objekte dieser Klasse können geworfen werden, wenn einer Funktion ein Parameter mit einem ungültigen Wert übergeben wird (z.B. durch den Konstruktor des Klassen-Templates `bitset<>,` wenn ihm ein Stringparameter übergeben wird, der andere Zeichen als `'0'` und `'1'` enthält)

  ◇ Klasse **`runtime_error`**

  ▻ ebenfalls definiert in der Headerdatei **`<stdexcept>`** (Diagnosebibliothek)

  ▻ dient als Basisklasse für weitere Exceptionklassen, die den Auftritt von **Laufzeitfehlern** kennzeichnen.
  Laufzeitfehler sind Fehler, die prinzipiell **erst zur Laufzeit** erkannt werden können (z.B. Divison durch `0`).

  Die folgenden von dieser Klasse **abgeleiteten Klassen** sind ebenfalls in der Headerdatei **`<stdexcept>`** definiert :

  ▻ Klasse **`range_error`**
  Objekte dieser Klasse kennzeichnen Bereichsfehler, die bei arithmetischen Berechnungen auftreten können.

  ▻ Klasse **`overflow_error`**
  Objekte dieser Klasse können geworfen werden, wenn ein arithmetischer Überlauf auftritt
  (z.B. in einer Memberfunktion des Klassen-Templates `bitset<>,` mit der ein Bitset in einen `unsigned long` Wert umgewandelt werden soll, wenn das Bitset nicht als `unsigned long` dargestellt werden kann)

  ▻ Klasse **`underflow_error`**
  Objekte dieser Klasse können geworfen werden, wenn ein arithmetischer Unterlauf auftritt

  ◇ Klasse **`ios_base::failure`**

  ▻ definiert in der Headerdatei **`<ios>`** innerhalb der Klasse `ios_base` (IO-Bibliothek)
  Die Headerdatei `<ios>` wird durch die Headerdatei `<iostream>` eingebunden.

  ▻ Objekte dieser Klasse (und davon abgeleiteter Klassen) werden zur Kennzeichnung von Fehlersituationen in der IO-Bibliothek geworfen.

  ◇ Für alle Klassen dieser Gruppe ist ein **Konstruktor** definiert, dem ein **`string`-Objekt**, das die **Fehlerursache** kennzeichnet, als Parameter zu übergeben ist.
  Beispiel : **`explicit logic_error(const string& what_arg);`**

  In realen Implementierungen sind i.a. zusätzlich definiert :

  ▻ virtueller Destruktor

  ▻ virtuelle Memberfunktion `const char*` **`what()`** `const throw()`

  Für die Klasse `ios_base::failure` sind diese Memberfunktionen auch in der ANSI-Norm vorgesehen.

Standard-Exception-Klassen - einfaches Demonstrationsprogramm (1)

- **Modul mit exception-werfender Funktion (C++-Quelldatei `testbibexcept_func.cpp`)**

```
#include <iostream>
#include <fstream>
#include <exception>
#include <stdexcept>
#include <typeinfo>
#include <new>
using namespace std;


class Base
{ public : virtual ~Base() {}; /* ... */ };

class Derived : public Base
{ public : virtual ~Derived() {}; /* ... */ };

class Der2 : public Base
{ public : virtual ~Der2() {}; /* ... */ };


void func1(int i)
{ switch(i)
  { case 0 : cout << "\ncase 0 : ";
             throw(exception());
             break;
    case 1 : cout << "\ncase 1 : ";
             throw(bad_exception());
             break;
    case 2 : cout << "\ncase 2 : ";
             { Base* bp=new Der2;
               dynamic_cast<Derived&>(*bp);    // throw(bad_cast());
             }
             break;
    case 3 : cout << "\ncase 3 : ";
             { Base* bp=NULL;
               typeid(*bp);                   // throw(bad_typeid());
             }
             break;
    case 4 : cout << "\ncase 4 : ";
             new char[0x7fffffff];            // throw(bad_alloc());
             break;
    case 5 : cout << "\ncase 5 : ";
             { ifstream istrm;
               istrm.exceptions(ios::eofbit | ios::failbit | ios::badbit);
               istrm.setstate(ios::eofbit);   // throw(ios_base::failure());
             }
             break;
    case 6 : cout << "\ncase 6 : ";
             throw(ios_base::failure("mein IO-Fehler"));
             break;
    case 7 : cout << "\ncase 7 : ";
             throw(logic_error("Exception wg. logischem Fehler"));
             break;
    case 8 : cout << "\ncase 8 : ";
             throw(runtime_error("Exception wg. Laufzeit-Fehler"));
             break;
    case 9 : cout << "\ncase 9 : ";
             throw(range_error("Exception wg. Range Error"));
             break;
    default: cout << "\ndefault: ";
             throw("Keine Bibliotheks-Exception");
             break;
  }
}
```

Standard-Exception-Klassen - einfaches Demonstrationsprogramm (2)

- **Modul mit exception-fangender `main()`-Funktion (C++-Quelldatei `testbibexcept_m.cpp`)**

```
#include <iostream>
#include <exception>
#include <new.h>       // Visual-C++-spezifisch, nur fuer _set_new_handler(_PNH)
using namespace std;

#define MAX_NR 10

extern void func1(int);

int mynewhdlr(size_t)
{ throw(bad_alloc());
}


int main(void)
{ _set_new_handler(mynewhdlr);      // Visual-C++-spezifische Funktion
  for (int i=0; i<=MAX_NR; i++)
  { try
    { func1(i);
      cout << "func(" << i << ") normal beendet" << endl;
    }
    catch(const exception& e)
    { cerr << e.what() << endl;
    }
    catch(const char* cpe)
    { cerr << cpe << endl;
    }
  }
  cout << "\nWeiter gehts !\n";
  return 0;
}
```

- **Ausgabe des Programms**

```
case 0 : Unknown exception

case 1 : bad exception

case 2 : Bad dynamic_cast!

case 3 : Attempted a typeid of NULL pointer!

case 4 : bad allocation

case 5 : ios::eofbit set

case 6 : mein IO-Fehler

case 7 : Exception wg. logischem Fehler

case 8 : Exception wg. Laufzeit-Fehler

case 9 : Exception wg. Range Error

default: Keine Bibliotheks-Exception

Weiter gehts !
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (1)

- **Eigenschaften**

  ◇ Das in der **Utility-Bibliothek** enthaltene Klassen-Template **`auto_ptr`** ist die Standard-Bibliotheks-Implementierung von **Smart-Pointern**.
  Smart-Pointer sind Objekte, die auf andere Objekte verweisen und die wie Pointer verwendet werden können, gegenüber diesen aber zusätzliche Eigenschaften besitzen.

  ◇ `auto_ptr`-Objekte enthalten einen **Pointer** auf das durch sie jeweils **referierte ("verwaltete") Objekt**.
  Das **referierte Objekt** muss **dynamisch** (mit `new`) **alloziert** worden sein.
  `auto_ptr`-Objekte dagegen werden meist statisch alloziert.
  Beim **Zerstören** eines **`auto_ptr`-Objekts** (z.B. beim Verlassen seines Gültigkeitsbereichs) wird das von ihm referierte Objekt **automatisch zerstört** (mit `delete`).

  ◇ Für `auto_ptr`-Objekte gilt eine **strikte Besitzsemantik** (*semantics of strict ownership*) : Ein von `auto_ptr`-Objekten verwaltetes Objekt darf immer nur von **einem einzigen** `auto_ptr`-Objekt referiert werden. Dieses referierende Objekt "**besitzt**" das referierte Objekt.
  Dies erfordert eine **andere Kopiersemantik** als die, die für "normale" Pointer gilt : Beim Kopieren zweier `auto_ptr`-Objekte (Copy-Konstruktor, Zuweisungsoperator) geht der "Besitz" des referierten Objekts vom Quell-`auto_ptr` zum Ziel-`auto_ptr` über. Der Quell-`auto_ptr` zeigt danach auf nichts (enthält den `NULL`-Pointer).
  → zerstörendes Kopieren.
  Wegen dieser Änderung des Quell-`auto_ptr`-Objekts darf es sich bei diesem nicht um ein konstantes Objekt handeln.
  Wird ein Objekt von mehr als einem `auto_ptr`-Objekt "besessen", ergibt sich ein undefiniertes Verhalten.

  ◇ Copy-Konstruktor und Zuweisungsoperator sind so überladen, dass auch das **Kopieren von `auto_ptr`-Objekten** ermöglicht wird, deren **referierter Objekt-Typ nicht identisch** ist. Es muss lediglich eine implizite Konvertierung zwischen den normalen Pointern auf die beteiligten Typen möglich sein.
  Damit wird die **Polymorphie zwischen Pointern** (Pointer auf Basisklasse – Pointer auf abgeleitete Klasse) auch **auf `auto_ptr`-Objekte übertragen**.

  ◇ **Weitere Funktionalitäten**, die von Smart-Pointern sinnvoll ausgeführt werden können, sind **nicht implementiert**.

- **Schnittstelle**

  ◇ Das Klasen-Template `auto_ptr` ist in der Headerdatei **`<memory>`** definiert.

  ◇ In der ANSI/ISO-Norm ist folgende **Schnittstelle** festgelegt :

```
template <class X> class auto_ptr
{   template <class Y> struct auto_ptr_ref { /* ... */ }; // Hilfsklasse
    // ... private Komponenten, z.B. X* ptr
  public :
    typedef X element_type;
    explicit auto_ptr(X* p = 0) throw();                  // Konstruktor
    ~auto_ptr() throw();

    auto_ptr(auto_ptr&) throw();                          // Copy-Konstruktor
    template <class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();               // Zuweisungsoperator
    template <class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();

    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();          // Rückgabe des Pointers auf ref. Objekt
    X* release() throw();            // Freigabe des referierten Objekts
    void reset(X* p = 0) throw();    // Änderung des referierten Objekts

    auto_ptr(auto_ptr_ref<X>) throw();    // Konstruktor zur Typkonvertierung
    template <class Y> operator auto_ptr_ref<Y>() throw();
    template <class Y> operator auto_ptr<Y>() throw();    // Typkonvertierung
};
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (2)

- **Kurzbeschreibung der Memberfunktionen, 1. Teil**

Anmerkungen :

- ▻ Alle Memberfunktionen von `auto_ptr` werfen keine Exceptions (Exception-Deklaration `throw()`)
- ▻ Der Datentyp `X` ist formaler Template-Parameter

◇ **`auto_ptr(X* p = 0)`** (Konstruktor)

- ▻ Initialisiert ein neues `auto_ptr`-Objekt mit dem Pointer `p` (Default `p = NULL`-Pointer)
  → das `auto_ptr`-Objekt besitzt das durch `p` referierte Objekt
- ▻ `p` muss auf ein dynamisch erzeugtes Objekt zeigen (oder der `NULL`-Pointer sein)

◇ **`~auto_ptr()`** (Destruktor)

- ▻ Falls das aktuelle `auto_ptr`-Objekt ein Objekt referiert, wird dieses mittels `delete` zerstört

---

◇ **`auto_ptr(auto_ptr& ap)`** (Copy-Konstruktor)

- ▻ Initialisiert ein neues `auto_ptr`-Objekt mit dem bisher von `ap` gehaltenen Pointer.
  → das bisher von `ap` besessene Objekt geht in den Besitz des neuen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr

◇ **`template <class Y> auto_ptr(auto_ptr<Y>& ap)`** (überladener Copy-Konstruktor)

- ▻ Initialisiert ein neues `auto_ptr`-Objekt mit dem bisher in `ap` enthaltenen Pointer.
  → das bisher von `ap` besessene Objekt geht in den Besitz des neuen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr
- ▻ Ermöglicht das Kopieren von `auto_ptr`-Objekten, die Objekte unterschiedlichen Typs referieren.
- ▻ `Y*` muss implizit in `X*` konvertierbar sein.

◇ **`auto_ptr& operator=(auto_ptr& ap)`** (Zuweisungsoperator)

- ▻ Falls das aktuelle `auto_ptr`-Objekt ein Objekt referiert,wird dieses mittels `delete` zerstört
- ▻ Weist dem aktuellen `auto_ptr`-Objekt den bisher von `ap` gehaltenen Pointer zu
  → das bisher von `ap` besessene Objekt geht in den Besitz des aktuellen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr

◇ **`template <class Y> auto_ptr& operator=(auto_ptr<Y>& ap)`** (überladener Zuweisungsoperator)

- ▻ Falls das aktuelle `auto_ptr`-Objekt ein Objekt referiert, wird dieses mittels `delete` zerstört
- ▻ Weist dem aktuellen `auto_ptr`-Objekt den bisher von `ap` gehaltenen Pointer zu
  → das bisher von `ap` besessene Objekt geht in den Besitz des aktuellen `auto_ptr`-Objektes über
- ▻ `ap` verweist danach auf kein Objekt mehr
- ▻ Ermöglicht das Kopieren von `auto_ptr`-Objekten, die Objekte unterschiedlichen Typs referieren.
- ▻ `Y*` muss implizit in `X*` konvertierbar sein.

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (3)

- **Kurzbeschreibung der Memberfunktionen, 2. Teil**

Anmerkungen :

- ▻ Alle Memberfunktionen von `auto_ptr` werfen keine Exceptions (Exception-Deklaration `throw()`)
- ▻ Der Datentyp `X` ist formaler Template-Parameter

◇ **`X& operator*() const`** (Dereferenzierungs-Operator)

- ▻ Gibt eine Referenz auf das vom aktuellen `auto_ptr`-Objekt besessene Objekt zurück
- ▻ Erfordert, dass das aktuelle `auto_ptr`-Objekt auch tatsächlich ein Objekt referiert

---

◇ **`X* operator->() const`** (Delegations-Operator)

- ▻ Gibt den vom aktuellen `auto_ptr`-Objekt gehaltenen Pointer zurück
  (Pointer auf besessenes Objekt bzw `NULL`-Pointer)

◇ **`X* get() const`**

- ▻ Gibt den vom aktuellen `auto_ptr`-Objekt gehaltenen Pointer zurück
  (Pointer auf besessenes Objekt bzw `NULL`-Pointer)

◇ **`X* release()`**

- ▻ Gibt den Besitz am bisher referierten Objekt frei
  → der bisher gehaltene Pointer wird durch den `NULL`-Pointer ersetzt.
- ▻ Das bisher referierte Objekt wird nicht zerstört.
- ▻ Gibt den vom aktuellen `auto_ptr`-Objekt bisher gehaltenen Pointer zurück

---

◇ **`void reset(X* p = 0)`**

- ▻ Falls der vom aktuellen `auto_ptr`-Objekt bisher gehaltene Pointer `!= p` ist, wird `p` zum neuen gehaltenen Pointer
  → das aktuelle `auto_ptr`-Objekt ändert seinen Besitz vom bisher referierten Objekt in das durch `p` referierte Objekt
- ▻ Das vom bisher gehaltenen Pointer referierte Objekt (das bisher besessene Objekt) wird zerstört
- ▻ `p` muß auf ein dynamisch erzeugtes Objekt zeigen (oder der `NULL`-Pointer sein)

◇ **`template <class Y> operator auto_ptr<Y>()`** (Typkonvertierung)

- ▻ Erzeugt ein `auto_ptr`-Objekt, das das vom aktuellen `auto_ptr`-Objekt bisher besessene Objekt referiert.
- ▻ Das aktuelle `auto_ptr`-Objekt gibt das bisher besessene Objekt frei
  → der bisher gehaltene Pointer wird durch den `NULL`-Pointer ersetzt.
- ▻ Das aktuelle und das neu erzeugte `auto_ptr`-Objekt können Objekte unterschiedlichen Typs (`X` bzw `Y`) referieren.
  `X*` muss aber in `Y*` konvertierbar sein. (Übertragung der Pointer-Polymorphie auf `auto_ptr`-Objekte)

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `auto_ptr` (4)

- **Abweichende Implementierungen**

  ◇ In einigen Systemen (z.B. auch **Visual-C++**) ist das Konzept der **strikten Besitzsemantik abweichend** vom ANSI/ISO-Standard **implementiert** :

  ▻ bei der **Aufgabe** (`release()`) bzw. **Übergabe** (Copy-Konstruktor, Zuweisung) des **Besitzes** an einem referierten Objekt wird der gehaltene **Pointer beibehalten** und nicht durch den `NULL`-Pointer ersetzt.
  → Das bisherige besitzende Objekt zeigt weiterhin auf das Objekt, das ihm nicht mehr gehört

  ▻ In einer weiteren Datenkomponente ist vermerkt, ob durch den gehaltenen Pointer ein besessenes Objekt referiert wird.
  Dadurch wird sichergestellt, dass ein referiertes Objekt tatsächlich immer nur von einem einzigen `auto_ptr`-Objekt zerstört werden kann.

  ◇ Weiterhin **fehlen** in einigen Implementierungen (z. B. auch in **Visual-C++**)

  ▻ die innere Hilfsklasse `template <class Y> struct auto_ptr_ref`

  ▻ sowie die Memberfunktionen :

  - `X* reset()`
  - `auto_ptr(auto_ptr_ref<X>)`
  - `template <class Y> operator auto_ptr_ref<Y>()`
  - `template <class Y> operator auto_ptr<Y>()`
  - `template <class Y> auto_ptr(auto_ptr<Y>&)`
  - `template <class Y> auto_ptr& operator=(auto_ptr<Y>&)`

    ⇒ Übertragung der zwischen Pointern bestehenden Polymorphie auf `auto_ptr`-Objekte ist nicht implementiert

Einfaches Demonstrationsprogramm 1 zum Klassen-Template `auto_ptr`

- **C++-Quelldatei `demobibap1_m.cpp`**

```
// Demonstrationsprogramm 1 zum Bibliotheks-Klassen-Template auto_ptr

#include <memory>
#include <iostream>
#include <exception>
using namespace std;


class Integer
{ public :
    Integer(int i = 0) : m_iDat(i)
      { cout << "\nKonstruktor Integer(" << m_iDat << ")\n"; }
    ~Integer() { cout << "\nDestruktor Integer(" << m_iDat << ")\n"; }
    void set(int i) { m_iDat=i; }
    int get() const { return m_iDat; }
  private :
    int m_iDat;
};

ostream& operator<<(ostream& out, const Integer& iv)
{ out << iv.get() ; return out; }


void func1()
{  try
  { auto_ptr<Integer> api(new Integer(25));
    Integer * pi = new Integer(4);
    cout << "\nWert von *api in func1 : " << *api;
    cout << "\nWert von  *pi in func1 : " << *pi << endl;
    throw(*new exception);
    delete pi;
  }
  catch (const exception& e)
  { cerr << endl << e.what() << endl;
  }
}


int main(void)
{ func1();
  cout << "\nmain() beendet\n";
  return 0;
}
```

- **Ausgabe des Programms `demobibap1`**

```
Konstruktor Integer(25)

Konstruktor Integer(4)

Wert von *api in func1 : 25
Wert von  *pi in func1 : 4

Destruktor Integer(25)

Unknown exception

main() beendet
```

Einfaches Demonstrationsprogramm 2 zum Klassen-Template `auto_ptr` (1)

- **C++-Quelldatei `demobibap2_m.cpp`**

```
// Demonstrationsprogramm 2 zum Bibliotheks-Klassen-Template auto_ptr

#include <memory>
#include <iostream>
using namespace std;


class Integer
{ public :
    Integer(int i = 0) : m_iDat(i)
      { cout << "\nKonstruktor Integer(" << m_iDat << ")\n"; }
    ~Integer() { cout << "\nDestruktor Integer(" << m_iDat << ")\n"; }
    void set(int i) { m_iDat=i; }
    int get() const { return m_iDat; }
  private :
    int m_iDat;
};


ostream& operator<<(ostream& out, const Integer& iv)
{ out << iv.get() ; return out; }


template <class T>
void refObjOut(ostream& out, const auto_ptr<T>& ap)
{
  if (ap.get()==NULL)
    out << "NULL";
  else
    out << *ap;
}


void func2(void)
{
  auto_ptr<Integer> api1(new Integer(2003));
  auto_ptr<Integer> api2(api1);
  auto_ptr<Integer> api3(new Integer(6));
  auto_ptr<Integer> api4(new Integer(2));

  cout << "\nWert von api1 in func2 : "; refObjOut(cout, api1);
  cout << "\nWert von api2 in func2 : "; refObjOut(cout, api2);
  cout << "\nWert von api3 in func2 : "; refObjOut(cout, api3);
  cout << "\nWert von api4 in func2 : "; refObjOut(cout, api4); cout << endl;
  api3=api2;
  cout << "\nnach Zuweisung api3=api2 :";
  cout << "\nWert von api2 in func2 : "; refObjOut(cout, api2);
  cout << "\nWert von api3 in func2 : "; refObjOut(cout, api3); cout << endl;
  api3.release();
  cout << "\nnach api3.release() :";
  cout << "\nWert von api2 in func2 : "; refObjOut(cout, api2);
  cout << "\nWert von api3 in func2 : "; refObjOut(cout, api3); cout << endl;
}

int main(void)
{
  func2();
  return 0;
}
```

Einfaches Demonstrationsprogramm 2 zum Klassen-Template `auto_ptr` (2)

- **Ausgabe des Programms `demobibap2` (übersetzt mit Visual-C++ 6.0)**

```
Konstruktor Integer(2003)

Konstruktor Integer(6)

Konstruktor Integer(2)

Wert von api1 in func2 : 2003
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 6
Wert von api4 in func2 : 2

Destruktor Integer(6)

nach Zuweisung api3=api2 :
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 2003

nach api3.release() :
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 2003

Destruktor Integer(2)
```

- **Ausgabe des Programms `demobibap2` (übersetzt mit Borland-C++ 5.5)**

```
Konstruktor Integer(2003)

Konstruktor Integer(6)

Konstruktor Integer(2)

Wert von api1 in func2 : NULL
Wert von api2 in func2 : 2003
Wert von api3 in func2 : 6
Wert von api4 in func2 : 2

Destruktor Integer(6)

nach Zuweisung api3=api2 :
Wert von api2 in func2 : NULL
Wert von api3 in func2 : 2003

nach api3.release() :
Wert von api2 in func2 : NULL
Wert von api3 in func2 : NULL

Destruktor Integer(2)
```

Einfaches Demonstrationsprogramm 3 zum Klassen-Template `auto_ptr` (1)

- **C++-Quelldatei `demobibap3_m.cpp`**

```
// Demonstrationsprogramm 3 zum Bibliotheks-Klassen-Template auto_ptr
// hier : Demo für Kopieren von auto_ptr-Objekten für unterschiedliche Typen der
//        referierten Objekte
//        sowie Aufruf der Memberfunktion reset()
//        läuft nicht unter Visual-C++

#include <memory>
#include <iostream>
using namespace std;


class Zahl
{ public :
    virtual ~Zahl() { cout << "Destruktor Zahl\n"; }
    virtual const char* className() const { return "Zahl"; }
    virtual void output(ostream&) const = 0;
  protected :
    Zahl() { }
};


class Integer : public Zahl
{ public :
    Integer(int i = 0) : m_iDat(i)
      { cout << "\nKonstruktor Integer(" << m_iDat << ")\n"; }
    ~Integer() { cout << "\nDestruktor Integer(" << m_iDat << ")\n"; }
    void output(ostream& out) const { out << m_iDat; }
    const char* className() const { return "Integer"; }
    void set(int i) { m_iDat=i; }
    int get() const { return m_iDat; }
  private :
    int m_iDat;
};

ostream& operator<<(ostream& out, const Zahl& iv)
{ iv.output(out) ; return out; }


template <class T>
void refObjOut(ostream& out, const auto_ptr<T>& ap)
{
  if (ap.get()==NULL)
    out << "NULL";
  else
    out << *ap << " (ref. Objekt : " << ap->className() << ')';
}


int main(void)
{
  auto_ptr<Zahl> apz;
  auto_ptr<Integer> api(new Integer(2003));
  apz=api;
  cout << "\nWert von api : "; refObjOut(cout, api); cout << endl;
  cout << "\nWert von apz : "; refObjOut(cout, apz); cout << endl;
  apz.reset(new Integer(30));
  cout << "\nWert von apz : "; refObjOut(cout, apz); cout << endl;
  return 0;
}
```

Einfaches Demonstrationsprogramm 3 zum Klassen-Template `auto_ptr` (2)

- **Ausgabe des Programms `demobibap3` (übersetzt mit Borland-C++ 5.5)**

```
Konstruktor Integer(2003)

Wert von api : NULL

Wert von apz : 2003 (ref. Objekt : Integer)

Konstruktor Integer(30)

Destruktor Integer(2003)
Destruktor Zahl

Wert von apz : 30 (ref. Objekt : Integer)

Destruktor Integer(30)
Destruktor Zahl
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `pair` (1)

- **Definition des Klassen-Templates `pair`**

  ◇ An einigen Stellen der Standardbibliothek werden **Wertepaare** verwendet.
  Beispielsweise verwalten die in der STL definierten Container-Klassen **`Map`** und **`Multimap`** Mengen, deren Elemente Wertepaare (Schlüssel/Wert) sind.

  ◇ Zur Unterstützung der Arbeit mit **Wertepaaren** ist in der **Utility-Bibliothek** als generischer Datentyp das – als **`struct`** realisierte – **Klassen-Template `pair`** definiert.

  ◇ Die **Definition** befindet sich in der Headerdatei **`<utility>`** :

```
template <class T1, class T2>
struct pair
{
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;                          // erster Wert des Paares
  T2 second;                         // zweiter Wert des Paares
  pair();                            // Default-Konstruktor
  pair(const T1& x, const T2& y);    // Konstruktor mit Initialisierungswerten
  template <class U, class V>
    pair(const pair<U, V>& p);       // Copy-Konstruktor
};
```

  ◇ Die **beiden Werte** eines Wertepaares sind durch zwei entsprechende **Datenkomponenten** realisiert.
  Diese sind **öffentlich** zugänglich (`public` ist Default bei `struct`), zu ihnen kann also direkt zugegriffen werden.

  ◇ Drei Konstruktoren bilden die einzigen Memberfunktionen.

- **Beschreibung der Konstruktoren**

  ◇ **`pair();`** (Default-Konstruktor)
  - ▻ Initialisierung der beiden Datenkomponenten durch Aufruf des Default-Konstruktors ihres jeweiligen Typs
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
pair() : first(T1()), second(T2()) {}
```

  ◇ **`pair(const T1& x, const T2& y);`** (Konstruktor mit Initialisierungswerten)
  - ▻ Initialisierung der beiden Datenkomponenten mit den übergebenen Parametern `x` und `y`
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
pair(const T1& x, const T2& y) : first(x), second(y) {}
```

  ◇ **`template <class U, class V> pair(const pair<U, V>& p);`** (Copy-Konstruktor)
  - ▻ Initialisierung der beiden Datenkomponenten mit den entsprechenden Datenkomponenten des Parameters `p`, wobei gegebenenfalls implizite Typkonvertierungen durchgeführt werden
  - ▻ typische Implementierung innerhalb der Klassendefinition :

```
template<class U, class V> pair(const pair<U, V> &p)
: first(p.first), second(p.second) {}
```

ANSI/ISO-C++-Standardbibliothek : Klassen-Template `pair` (2)

- **Zusätzlich definierte freie Funktionen :**

  ◇ Die Utility-Bibliothek enthält zusätzlich **7 freie Funktionen** zur Verwendung mit dem Klassen-Template `pair`. Typischerweise sind diese Funktionen in der Headerdatei `<utility>` als **`inline`-Funktionen** definiert. 6 dieser Funktionen sind **Vergleichsfunktionen,** die 7. dient zur **Erzeugung eines `pair`-Objektes**.

◇
```
template <class T1, class T2>
  inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return x.first == y.first && x.second == y.second; }
```
  ▻ Test zweier `pair`-Objekte auf Gleichheit

  ▻ Zwei `pair`-Objekte sind genau dann gleich, wenn ihre beiden entsprechenden Komponenten jeweils gleich sind

---

◇
```
template <class T1, class T2>
  inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return x.first < y.first || (!(y.first < x.first) && x.second < y.second); }
```
  ▻ Test zweier `pair`-Objekte auf "kleiner als"

  ▻ Für den Vergleich wird primär die erste Komponente überprüft. Wenn diese für beide Objekte gleich ist, wird das Vergleichsergebnis durch Vergleich der zweiten Komponente ermittelt.

◇
```
template <class T1, class T2>
  inline bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(x == y); }
```
  ▻ Test zweier `pair`-Objekte auf Ungleichheit

◇
```
template <class T1, class T2>
  inline bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return y < x; }
```
  ▻ Test zweier `pair`-Objekte auf "größer als"

◇
```
template <class T1, class T2>
  inline bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(x < y); }
```
  ▻ Test zweier `pair`-Objekte auf "größer gleich"

◇
```
template <class T1, class T2>
  inline bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y)
  { return !(y < x); }
```
  ▻ Test zweier `pair`-Objekte auf "kleiner gleich"

◇
```
template <class T1, class T2>
  inline pair<T1, T2> make_pair(const T1& x, const T2& y)
  { return pair<T1, T2>(x, y); }
```
  ▻ Erzeugung eines `pair`-Objektes aus zwei Komponenten ohne explizite Angabe der Komponententypen

  Beispiel : `return` **`make_pair(5, 3.728);`** statt `return pair<int, double>(5, 3.728);`

ANSI/ISO-C++-Standardbibliothek : String-Bibliothek Überblick (1)

- **String-Bibliothek**
  - ◇ Zur ANSI-C++-Standardbibliothek gehörende Teilbibliothek für die **String-Unterstützung**.
  - ◇ Sie stellt ein **Klassen-Template** zur Verfügung, dessen Instanzen, die **String-Klassen**, eine sehr **komfortable** und **sichere** (Speicherverwaltung und Bereichskontrolle) Bearbeitung von Strings ermöglichen.
    Wesentlicher **Template-Parameter** ist der **Datentyp** zur Darstellung von **Zeichen**.
  - ◇ Strings werden also – im Unterschied zu C-Strings – als **Objekte** dargestellt und verwendet.
    Die in den Objekten enthaltene Zeichenfolge ist nicht mit einem speziellen Endzeichen (z.B. `'\0'`–Character) abgeschlossen. Vielmehr kann sie jedes Zeichen des verwendeten Zeichen-Datentyps enthalten.
    Der für die Ablage der Zeichenfolge benötigte Speicherplatz wird dem jeweiligen Bedarf entsprechend durch die Memberfunktionen des Klassen-Templates alloziert bzw freigegeben.
  - ◇ Alle für die Anwendung der String-Bibliothek notwendigen Vereinbarungen sind in der Headerdatei `<string>` enthalten. Sie liegen alle im Namensraum **`std.`**

- **Klassen-Template `basic_string`**
  - ◇ Zentrales Klassen-Template für die String-Klassen, das deren Verhalten und Fähigkeiten definiert.
  - ◇ Die aus dem Template instantiierbaren String-Klassen sind durch drei **Template-Parameter** bestimmt :
    - ▻ **`charT`** : **Datentyp** zur Darstellung der **Zeichen** (Zeichentyp)
    - ▻ **`traits`** : **Klasse** zur Beschreibung von **Eigenschaften** (Merkmalen) des **Zeichentyps** (*character traits*).
      Als **Default** ist hierfür die Instanz des ebenfalls mit dem Zeichen-Datentyp `charT` parameterisierten Klassen-Templates `char_traits`, das ist **`char_traits<charT>`**, festgelegt.
      Das Klassen-Template `char_traits` ist auch in der Header-Datei `<string>` definiert.
    - ▻ **`Allocator`** : **Allokator-Klasse**, die das Speichermodell beschreibt, das die zu verwendende dynamische Speicherverwaltung festlegt.
      Als **Default** ist die Instanz des ebenfalls mit dem Zeichen-Datentyp `charT` parametrisierten Klassen-Templates `allocator`, das ist **`allocator<charT>`**, vorgesehen.
      Das Klassen-Template `allocator` bestimmt die im System eingesetzten Standard-Allokator-Klassen, die `new` und `delete` zur Speicherallokation verwenden. Seine Definition befindet sich in der Headerdatei `<memory>`.
  - ◇ **Definition** :

```
template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
  class basic_string { /* ... */ };
```

- **String-Klassen `string` und `wstring`**
  - ◇ Standardmäßig sind in `<string>` zwei String-Klassen als **Instanzen** von `basic_string` vordefiniert.
  - ◇ Klasse **`string`** für den Datentyp **`char`** ("normale" Ein-Byte-Zeichen) unter Verwendung der Defaults für die anderen Parameter :

```
typedef basic_string<char> string;
```

  - ◇ Klasse **`wstring`** für den Datentyp **`wchar_t`** (Mehr-Byte-Zeichen, z.B. Unicode-Zeichen) unter Verwendung der Defaults für die anderen Parameter :

```
typedef basic_string<wchar_t> wstring;
```

ANSI/ISO-C++-Standardbibliothek : String-Bibliothek Überblick (2)

• Überblick über die Operationen mit Strings

◇ Die String-Bibliothek von C++ erlaubt es, dass mit Strings – im Gegensatz zu C-Strings – wie mit elementaren Datentypen gearbeitet werden kann.
De facto steht mit der Bibliotheks-Klasse `string` (bzw. `wstring` bzw. jeder anderen Klassen-Instanz von `basic_string<>`) ein quasi-fundamentaler Datentyp zur Verfügung.

◇ Das Klassen-Template **`basic_string<>`** definiert

- ▻ grundlegende Fähigkeiten zum **Anlegen** (Initialisieren) und **Kopieren** von Strings
- ▻ zahlreiche Methoden zum **Manipulieren** und **Bearbeiten** von Strings, wie Konkatenieren, Einfügen, Löschen, Suchen, Ersetzen, Vergleichen und Teilstringbildung.
  Die meisten dieser Methoden sind mehrfach so überladen, dass sich die entsprechenden Operationen – wo sinnvoll – auch im Zusammenhang mit C-Strings und Einzelzeichen durchführen lassen.
  Für einige der Bearbeitungsoperationen existieren neben Memberfunktionen mit einem geeigneten Namen auch entsprechende – z. T. freie – Operatorfunktionen (s. unten).
- ▻ Methoden zum **Zugriff** zu den **einzelnen Zeichen** eines Strings (String-Elementen)
- ▻ Methoden zum **Umwandeln** von Strings in **C-Strings**
- ▻ Methoden zum Bereitstellen von **String-Iteratoren**
- ▻ Methoden zur **Ermittlung** und **Änderung** der **Stringlänge** und **Stringkapazität**.

◇ Weiterhin sind in der Stringbibliothek zahlreiche **freie Operatorfunktionen** für Strings definiert.
Diese ermöglichen

- ▻ die Stream-**Ein-** und **Ausgabe** von Strings
- ▻ die **Konkatenation** von Strings, C-Strings und Einzelzeichen an Strings
  sowie die Konkatenation von Strings an Strings, C-Strings und Einzelzeichen
- ▻ den **Vergleich** von Strings mit Strings, Strings mit C-Strings und C-Strings mit Strings

◇ In vielen – aber nicht allen – Funktionen, in denen die Möglichkeit zu einem Speicherzugriffsfehler besteht, können **Exceptions** geworfen werden :

- ▻ Exception der Klasse **`out_of_range`** beim Zugriff zu einer unzulässigen String-Position
- ▻ Exception der Klasse **`length_error`** beim Versuch einen String über die maximal mögliche Länge hinaus zu verlängern

Beide Exception-Klassen sind in der Standard-Bibliothek definiert. Sie sind von der Exception-Klasse `logic_error` abgeleitet. Diese wiederum ist von der Exception-Basis-Klasse `exception` abgeleitet.

• Anmerkungen zur Beschreibung der String-Funktionen

◇ Zur Vereinfachung und Erhöhung der Übersichtlichkeit werden die einzelnen String-Funktionen im folgenden nicht in der originalen Template-Form sondern in einer für die – am häufigsten verwendete – **Klassen-Instanz `string` spezialisierten Darstellung** angegeben :

- ▻ statt dem Klassen-Template `basic_string` wird `string` eingesetzt
- ▻ statt dem Template-Parameter `charT` wird der Zeichen-Datentyp `char` eingesetzt

◇ **Beispiele** :

▻ statt : `basic_string& append(const charT* s);` (Memberfunktion)
wird formuliert : **`string& append(const char* s);`**

▻ statt :
```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is,
               (basic_string<charT, traits, Allocator>& str);
```
wird formuliert : **`istream& operator>>(istream& is, string& str);`**

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (1)

- **Datentypen als Klassenelemente**

  ◇ Innerhalb der Klasse `string` sind mehrere Datentypen – meist indirekt – definiert. Es sind `public`-Komponenten . Sie stehen damit zur Verwendung außerhalb der Klasse `string` zur Verfügung.

  ◇ Einer dieser Datentypen ist : **`size_type`**
  Es handelt sich um einen vorzeichenlosen ganzzahligen Datentyp, der zur Angabe von String-Positionen, Anzahl-, Längen- und Größenangaben verwendet wird.
  Dieser wird durch die **Allokator-Klasse** festgelegt. Für ISO-konforme Container muss dieser dem Typ **`std::size_t`** entsprechen.

- **`public`-Datenkomponente der Klasse `string`**

  ◇ **`static const size_type npos = -1;`**

  Definition einer Konstanten, die – je nach Anwendung – als (**Default-**)**Parameterwert** für "**alle Zeichen**" ("bis zum Stringende") oder zur Kennzeichnung einer **ungültigen Position** verwendet wird.
  Da der Typ der Konstante (**`string::size_type`**) als vorzeichenloser ganzzahliger Datentyp definiert ist, entspricht der Wert **`-1`** dem **größtmöglichen Wert** dieses Typs.

Achtung : Außerhalb der Qualifikation ist für **`size_type`** und **`npos`** der Klassennamen `string::` mitanzugeben!

- **Konstruktoren und Destruktor der Klasse `string`**

`string();`	Erzeugung eines Leer-Strings (Länge `0`)
`string(const string& str, size_type pos=0, size_type n= npos);`	Erzeugung eines Strings, der mit `n` Zeichen ab der Position `pos` des Strings `str` initialisiert wird. Für die Default-Parameterwerte: Copy-Konstruktor
`string(const char* s);`	Erzeugung eines Strings, der mit dem C-String `s` initialisiert wird
`string(const char* s, size_type n);`	Erzeugung eines Strings, der mit den ersten `n` (höchstens aber allen) Zeichen des C-Strings `s` initialisiert wird
`string(size_type n, char c);`	Erzeugung eines Strings der Länge `n`. Alle Komponenten werden mit dem Zeichen `c` initialisiert
`~string();`	Destruktor

- **Memberfunktionen zur Ermittlung der Länge und Kapazität**

`size_type size() const;`	Ermittlung der aktuellen Stringlänge (Anzahl der Zeichen im String)
`size_type length() const;`	Ermittlung der aktuellen Stringlänge (Anzahl der Zeichen im String)
`size_type capacity() const;`	Ermittlung der maximal möglichen Stringlänge ohne Neuallokation
`size_type max_size() const;`	Ermittlung der maximalen Größe, die ein String haben kann
`bool empty() const;`	Ermittlung, ob String leer ist, wenn ja Rückgabe von `true`, sonst `false`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (2)

- **Memberfunktionen zur Änderung der Länge und Kapazität**

`void resize(size_type n, char c);`	Falls `n<=max_size()` : Änderung der Stringlänge auf den Wert `n` bei Stringverlängerung : Auffüllen mit dem Zeichen `c` Falls `n>max_size()` : Werfen der Exception **`length_error`**
`void resize(size_type n);`	Falls `n<=max_size()` : Änderung der Stringlänge auf den Wert `n` bei Stringverlängerung : die zusätzlichen Zeichen bleiben undefiniert Falls `n>max_size()` : Werfen der Exception **`length_error`**
`void clear();`	Löschen aller Zeichen im String, neue Stringlänge = `0`
`void reserve(size_type res=0);`	Falls `res>capacity()` : Vergrößerung der Kapazität Falls `res<=max_size()` : neue Kapazität `>= res` Falls `res>max_size()` : Werfen der Exception **`length_error`** Falls `res<=capacity()` : keine Wirkung

- **Memberfunktionen zur Zuweisung**

Rückgabewert für alle Zuweisungsfunktionen : **Referenz** auf das **aktuelle** String-Objekt (`*this`)	
`string& operator=(const string& str);`	Zuweisung des Strings `str`
`string& operator=(const char* s);`	Zuweisung des C-Strings `s`
`string& operator=(char c);`	Zuweisung des Einzelzeichens `c` (neue Stringlänge `= 1`)
`string& assign(const string& str);`	Zuweisung des Strings `str`
`string& assign(const string& str` **`size_type pos, size_type n);`**	Falls `pos<str.size()` : Zuweisung von `n` (höchstens aber allen) Zeichen ab der Position `pos` aus dem String `str` Falls `pos==str.size()` : keine Wirkung Falls `pos>str.size()` : Werfen der Exception **`out_of_range`**
`string& assign(const char* s);`	Zuweisung des C-Strings `s`
`string& assign(const char* s, size_type n);`	Zuweisung der ersten `n` (höchstens aber alle) Zeichen des C-Strings `s`
`string& assign(size_type n, char c);`	Zuweisung eines Strings der Länge `n`, bei dem alle Zeichen gleich dem Zeichen `c` sind

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (3)

- **Memberfunktionen zum Zugriff zu Einzel-Zeichen im String (String-Elementen)**

`const char& operator[](size_type pos) const;` `char& operator[](size_type pos);`	Falls `pos<size()` : Rückgabe des Zeichens (bzw Referenz auf das Zeichen) an der Position (Index) `pos` Falls `pos>=size()` : undefiniert
`const char& at(size_type pos) const;` `char& at(size_type pos);`	Falls `pos<size()` : Rückgabe des Zeichens (bzw Referenz auf das Zeichen) an der Position (Index) `pos` Falls `pos>=size()` : Werfen der Exception **`out_of_range`**

- **Memberfunktionen zur Konvertierung von Strings in `char`-Arrays bzw C-Strings**

`size_type copy(char* s, size_type n, size_type pos = 0) const;`	Kopieren von `n` Zeichen ab der Position `pos` (höchstens jedoch bis zum Stringende) in das durch `s` referierte `char`-Array. Es wird kein `'\0'`-Character angehängt Rückgabewert : Anzahl kopierter Zeichen
`const char* data() const;`	Rückgabe eines Pointers auf ein `char`-Array, dessen Elemente mit den Zeichen des aktuellen Strings übereinstimmen. Das `char`-Array ist nicht mit dem `'\0'`-Character abgeschlossen
`const char* c_str() const;`	Rückgabe eines Pointers auf einen C-String (Abschluss mit `'\0'`-Character), dessen Zeichen mit den Zeichen des aktuellen Strings übereinstimmen.

- **Memberfunktionen zum Vergleich von Strings bzw. Teilstrings**

Rückgabewert für **alle Vergleichsfunktionen** : `<0`, wenn akt. (Teil-) String `< str` (bzw `s`) `0`, wenn akt. (Teil-) String `== str` (bzw `s`) `>0`, wenn akt. (Teil-) String `> str` (bzw `s`)	
`int compare(const string& str) const;`	Vergleich aktueller String mit dem String `str`
`int compare(size_type pos, size_type n, const string& str) const;`	Vergleich `n` Zeichen des aktuellen Strings ab Position `pos` mit dem String `str`
`int compare(size_type pos1, size_type n1, const string& str, size_type pos2, size_type n2) const;`	Vergleich `n1` Zeichen des aktuellen Strings ab Position `pos1` mit `n2` Zeichen des Strings `str` ab Position `pos2`
`int compare(const char* s) const;`	Vergleich aktueller String mit dem C-String `s`
`int compare(size_type pos1, size_type n1, const char* s, size_type n2=npos) const;`	Vergleich `n1` Zeichen des aktuellen Strings ab Position `pos1` mit `n2` Zeichen (default : bis String-Ende) des C-Strings `s`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (4)

• Memberfunktionen zum Anhängen (Konkatenation)

Für **alle Funktionen zum Anhängen** gilt :	**Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge übersteigen würde, wird die Exception **`length_error`** geworfen
`string& operator+=(const string& str);`	Anhängen des Strings `str` an den aktuellen String
`string& operator+=(const char* s);`	Anhängen des C-Strings `s` an den aktuellen String
`string& operator+=(char c);`	Anhängen des Zeichens `c` an den aktuellen String
`string& append(const string& str);`	Anhängen des Strings `str` an den aktuellen String
`string& append(const string& str,` **`size_type n,`** **`size_type pos);`**	Falls `pos<=str.size()` : Anhängen von `n` Zeichen des Strings `str` ab der Position `pos` (aber maximal bis zum Stringende) an den aktuellen String. Falls `pos>str.size()` : Werfen der Exception **`out_of_range`**
`string& append(const char* s);`	Anhängen des C-Strings `s` an den aktuellen String
`string& append(const char* s,` **`size_type n);`**	Anhängen der ersten `n` (höchstens aber allen) Zeichen des C-Strings `s` an den aktuellen String
`string& append(size_type n, char c);`	Anhängen von `n`-mal das Zeichen `c` an den aktuellen String

• Memberfunktionen zum Einfügen

Für **alle Einfügefunktionen** gilt :	**Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls eine Positionsangabe größer als die jeweilige Stringlänge ist, wird die Exception **`out_of_range`** geworfen Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge (`max_size()`) übersteigen würde, wird die Exception **`length_error`** geworfen
`string& insert(size_type pos,` **`const string& str);`**	Einfügen des Strings `str` in den aktuellen String ab der Position `pos`
`string& insert(size_type pos1,` **`const string& str,`** **`size_type pos2, size_type n);`**	Einfügen von maximal `n` Zeichen ab der Position `pos2` aus dem String `str` in den aktuellen String ab der Position `pos1`
`string& insert(size_type pos, const char* s);`	Einfügen des C-Strings `s` in den aktuellen String ab der Position `pos`
`string& insert(size_type pos, const char* s,` **`size_type n);`**	Einfügen der ersten `n` (maximal allen) Zeichen des C-Strings `s` in den aktuellen String ab Pos. `pos`
`string& insert(size_type pos,` **`size_type n, char c);`**	Einfügen von `n`-mal das Zeichen `c` in den aktuellen String ab der Position `pos`

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (5)

- **Memberfunktionen zum Ersetzen und Löschen**

`void swap(string& str);`	Vertauschen des Inhalts des aktuellen Strings mit dem Inhalt des Strings `str`
Für **alle** folgenden **Funktionen** gilt : **Rückgabewert** ist **Referenz** auf das **aktuelle** String-Objekt (`*this`) Falls eine Positionsangabe größer als die jeweilige Stringlänge ist, wird die Exception **`out_of_range`** geworfen Für die **Ersetzungs**-Funktionen gilt : Die Anzahl der ersetzenden Zeichen kann kleiner, größer oder gleich der Anzahl der ersetzten Zeichen sein Falls die neue Länge des Strings den maximal möglichen Wert für die Stringlänge (`max_size()`) übersteigen würde, wird die Exception **`length_error`** geworfen	
`string& replace(size_type pos, size_type n, const string& str);`	Ersetzen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` durch alle Zeichen des Strings `str`
`string& replace(size_type pos1,size_type n1, const string& str, size_type pos2, size_type n2);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos1` durch max. `n2` Zeichen des Strings `str` ab der Position `pos2`
`string& replace(size_type pos, size_type n, const char* s);`	Ersetzen von max. `n` Zeichen des aktuellen Strings ab der Position `pos` durch alle Zeichen des C-Strings `s`
`string& replace(size_type pos, size_type n1, const char* s, size_type n2);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos` durch max. `n2` Zeichen des `C`–Strings `s`
`string& replace(size_type pos, size_type n1, size_type n2, char c);`	Ersetzen von max. `n1` Zeichen des aktuellen Strings ab der Position `pos` durch `n2`–mal das Zeichen `c`
`string& erase(size_type pos = 0, size_type n = npos);`	Löschen von max. `n` Zeichen des aktuellen Strings ab der Position `pos`. Für Default-Parameterwerte : Löschen aller Zeichen

- **Memberfunktion zur Ermittlung eines Teilstrings**

`string substr(size_type pos = 0, size_type n = npos);`	Bildung und Rückgabe eines neuen Strings (`string`–Objekt), der aus max. `n` Zeichen des aktuellen Strings ab der Position `pos` besteht. Für Default-Parameterwerte : Rückgabe einer Kopie des aktuellen Strings. Falls `pos>size()` ist, Werfen der Exception **`out_of_range`**

- **Memberfunktion zur Ermittlung und Verwendung von Iteratoren**

  ◇ Es existieren eine Reihe von Memberfunktionen zur Ermittlung von Iteratoren für das aktuelle `string`–Objekt

  ◇ Darüber hinaus existieren für viele der o. a. String-Bearbeitungsfunktionen Versionen, die Iteratoren statt Positions-(Index-) Angaben als Parameter verwenden

ANSI/ISO-C++-Standardbibliothek : Klasse `string` (6)

- **Memberfunktionen zum Suchen**

  ◇ Es existieren insgesamt **24 Memberfunktionen** zum Suchen in Strings

  ◇ Es kann nach **Teilstrings**, **C-Teilstrings**, **Einzelzeichen** oder **Zeichen aus einer Zeichenmenge** gesucht werden.
  Es kann auch nach dem Auftritt von Zeichen gesucht werden, die in einer **Zeichenmenge nicht enthalten** sind

  ◇ Es ist eine Suche nach dem **ersten** oder dem **letzten Auftritt** des Suchparameters möglich

  ◇ Alle Suchfunktionen sind `const`-Funktionen, d. h. sie ändern das aktuelle Objekt nicht

  ◇ Alle Suchfunktionen geben eine Wert vom Typ `string::size_type` zurück.
  Bei **Such-Erfolg** handelt es sich um die **Position** (Index) des (Beginns des) gefundenen Auftritts.
  Bei **Misserfolg** (Suchparameter ist im aktuellen Objekt nicht vorhanden) wird **`npos`** als Kenzeichnung einer **ungültigen Position** zurückgegeben.

  ◇ Es existieren die folgenden Funktionsgruppen :

▻ **`size_type find(...) const`**	Suchen nach **erstem** Auftritt
▻ **`size_type rfind(...) const`**	Suchen nach **letztem** Auftritt
▻ **`size_type find_first_of(...) const`**	Suchen nach **erstem** Auftritt eines Zeichens aus einer **Zeichenmenge**
▻ **`size_type find_last_of(...) const`**	Suchen nach **letztem** Auftritt eines Zeichens aus einer **Zeichenmenge**
▻ **`size_type find_first_not_of(...) const`**	Suchen nach **erstem** Auftritt eines Zeichens, das in einer **Zeichenmenge nicht** enthalten ist
▻ **`size_type find_last_not_of(...) const`**	Suchen nach **letztem** Auftritt eines Zeichens, das in einer **Zeichenmenge nicht** enthalten ist

  ◇ **Einige ausgewählte Suchfunktionen** :

`size_type find(const string& str, size_type pos = 0) const;`	Suchen nach erstem Auftritt des Strings `str` ab der Position `pos` im aktuellen String
`size_type find(const char* s, size_type pos = 0) const;`	Suchen nach erstem Auftritt des C-Strings `s` ab der Position `pos` im aktuellen String
`size_type find(char c, size_type pos = 0) const;`	Suchen nach erstem Auftritt des Zeichens `c` ab der Position `pos` im aktuellen String
`size_type find(const char* s, size_type pos, size_type n) const;`	Suchen nach erstem Auftritt der ersten `n` Zeichen des C-Strings `s` ab der Position `pos` im aktuellen String
`size_type rfind(const string& str, size_type pos = npos) const;`	Suchen nach letztem Auftritt des Strings `str` ab der Position `pos` im aktuellen String
`size_type find_first_of(const string& str, size_type pos = 0) const;`	Suchen nach erstem Auftritt eines der im String `str` enthaltenen Zeichen ab der Position `pos` im aktuellen String
`size_type find_last_not_of(const string& str, size_type pos = npos) const;`	Suchen nach letztem Auftritt eines der im String `str` nicht enthaltenen Zeichen ab der Position `pos` im aktuellen String

ANSI/ISO-C++-Standardbibliothek : Freie Stringbearbeitungsfunktionen (1)

• Freie Operatorfunktionen zur String-Konkatenation

Für **alle Funktionen** gilt : Sie erzeugen einen **neuen String** (`string`–Objekt), den sie als **Funktionswert** zurückgeben
In ihrem Verhalten werden sie auf den Aufruf der Memberfunktion `append()` zurückgeführt
Das bedeutet, dass die Exception **`length_error`** geworfen wird, falls die Länge des neu zu erzeugenden Strings den maximal möglichen Wert für die Stringlänge übersteigt.

`string operator+(const string& str1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem String `str2` besteht
`string operator+(const char* s1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des umgewandelten C-Strings `s1` mit dem String `str2` besteht
`string operator+(const string& str1, const char* s2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem umgewandelten C-String `s2` besteht
`string operator+(char c1, const string& str2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des umgewandelten Zeichens `c1` mit dem String `str2` besteht
`string operator+(const string& str1, char c2);`	Erzeugung eines neuen Strings, der aus der Konkatenation des Strings `str1` mit dem umgewandelten Zeichen `c2` besteht

• Freie Operatorfunktionen zum String-Vergleich

Für die **folgenden Funktionen** gilt : Das Symbol ***`op`*** steht für einen der Vergleichsoperatoren **`== != < <= > >=`**
Rückgabewert : `true`, falls der Vergleich erfüllt ist,
`false`, falls der Vergleich nicht erfüllt ist

`bool operator op(const string& str1, const string& str2);`	Vergleich des Strings `str1` mit dem String `str2`
`bool operator op(const char* s1, const string& str2);`	Vergleich des C-Strings `s1` mit dem String `str2`
`bool operator op(const string& str1, const char* s2);`	Vergleich des Strings `str1` mit dem C-String `s2`

• Freie Funktion zum Vertauschen zweier Strings

`void swap(string& str1, string& str2);`	Vertauschen des Inhalts Strings `str1` mit dem Inhalt des Strings `str2` Wirkungsgleich mit : **`str1.swap(str2);`**

ANSI/ISO-C++-Standardbibliothek : Freie Stringbearbeitungsfunktionen (2)

- **Freie Operatorfunktionen zur Stream-Ein-/Ausgabe**

istream& operator>>(istream& is, string& str);	Einlesen der nächsten Zeichen aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Falls eine Feldbreite mit `is.width(...)` explizit gesetzt worden ist, werden maximal `is.width()` Zeichen eingelesen, andernfalls werden maximal `str.max_size()` Zeichen eingelesen. Das Einlesen wird früher beendet, wenn das nächste einzulesende Zeichen ein *White-Space-Character* ist oder das Dateiende erreicht ist Rückgabewert : `is`
ostream& operator<<(ostream& os, string& str);	Ausgabe des Strings `str` in den durch `os` referierten Ausgabestream Rückgabewert : `os`

- **Freie Funktionen zum Einlesen von Textzeilen aus Eingabe-Streams**

istream& getline(istream& is, string& str, char delim);	Einlesen einer mit dem Zeichen `delim` abgeschlossenen Zeichenfolge aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Das Zeichen `delim` wird aus dem Stream entfernt aber nicht mit im String abgelegt. Das Einlesen wird vor Auftritt des Zeichens `delim` beendet, wenn bereits `str.max_size()` Zeichen gelesen wurden oder das Dateiende erreicht ist
istream& getline(istream& is, string& str);	Einlesen einer mit dem Zeichen `'\n'` abgeschlossenen Zeichenfolge aus dem durch `is` referierten Eingabestream und Ablage dieser im String `str`. Das Zeichen `'\n'` wird aus dem Stream entfernt aber nicht mit im String abgelegt. Das Einlesen wird vor Auftritt des Zeichens `'\n'` beendet, wenn bereits `str.max_size()` Zeichen gelesen wurden oder das Dateiende erreicht ist Der Aufruf dieser Funktion entspricht dem Aufruf von `getline(is, str, '\n');`

Demo-Programm 1 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (1)

- **C++-Quelldatei `stringdem_1.cpp` (`main()`-Funktion des Programms `stringdem1`)**

```
// C++-Quelldatei stringdem1_m.cpp
// Demonstrationsprogramm stringdem1 zur Bibliotheks-Klasse string

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


void stringInfo(ostream& out, const string& s)
{ out << s;
  out << "\nsize     : " << s.size();
  out << "\ncapacity : " << s.capacity();
  out << "\nmax_size : " << s.max_size();
}


int main()
{
  cout << "\nWert von npos : " << hex << string::npos << dec << endl;

  string s1;         // Leer-String
  cout << "\ns1 Leerstring";
  stringInfo(cout, s1);
  cout << endl;

  char* cp="Hallo !";
  string s2(cp);
  cout << "\ns2 aus C-String : ";
  stringInfo(cout, s2);
  cout << endl;

  string s3(s2);
  cout << "\ns3 mit Copy-Konstruktor : ";
  stringInfo(cout, s3);
  cout << endl;

  string s4(3, 'Z');
  cout << "\ns4 aus Einzelzeichen : ";
  stringInfo(cout, s4);
  cout << endl;

  string::size_type nLen=3;
  s2.resize(nLen);
  cout << "\ns2 nach resize(" << nLen << ") : ";
  stringInfo(cout, s2);
  cout << endl;

  nLen=10;
  s2.resize(nLen, 'o');
  cout << "\ns2 nach resize(" << nLen << ") mit Fuellzeichen : ";
  stringInfo(cout, s2);
  cout << endl;

  nLen=80;
  s2.reserve(nLen);
  cout << "\ns2 nach reserve(" << nLen << ") : ";
  stringInfo(cout, s2);
  cout << endl;

  return 0;
}
```

Demo-Programm 1 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (2)

- **Ausgabe des Programms `stringdem1`**

```
Wert von npos : ffffffff

s1 Leerstring
size     : 0
capacity : 0
max_size : 4294967293

s2 aus C-String : Hallo !
size     : 7
capacity : 31
max_size : 4294967293

s3 mit Copy-Konstruktor : Hallo !
size     : 7
capacity : 31
max_size : 4294967293

s4 aus Einzelzeichen : ZZZ
size     : 3
capacity : 31
max_size : 4294967293

s2 nach resize(3) : Hal
size     : 3
capacity : 31
max_size : 4294967293

s2 nach resize(10) mit Fuellzeichen : Haloooooo
size     : 10
capacity : 31
max_size : 4294967293

s2 nach reserve(80) : Haloooooo
size     : 10
capacity : 95
max_size : 4294967293
```

Demo-Programm 2 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string`

- **C++-Quelldatei `stringdem2_m.cpp` (`main()`-Funktion des Programms `stringdem2`)**

```
// C++-Quelldatei stringdem2_m.cpp
// Demonstrationsprogramm stringdem2 zur Bibliotheks-Klasse string

#include <string>
#include <iostream>
#include <exception>
#include <iomanip>
using namespace std;

int main()
{ string s1;
  string s2("Blaer bricht das Voelkerrecht");
  string s3, s4;
  try
  { s1="Busch";
    s4=s1;
    s1.append(" Monkey");
    cout << endl << s1.c_str() << endl;    // s1.c_str() statt s1 nur zu Demo-Zweck
    s3.assign(s2, 5, string::npos);
    cout << endl << s3 << endl;
    s4+=s3;
    cout << endl << s4 << endl;
    cout << endl << endl;
    for (unsigned i=6; i<20; i++)
      cout << hex << setw(2) << setfill('0') << (s1[i]&0xff) << ' ';
    cout << dec << setfill(' ') << endl;
    for (i=6; i<20; i++)
      cout << hex << setw(2) << setfill('0') << (s1.at(i)&0xff) << ' ';
    cout << dec << setfill(' ') << endl;
  }
  catch (const exception& e)
  { cerr << "\nexception : " << e.what() << endl;
  }
  try
  { cout << "\nreserve(" << hex << string::npos << dec << ") : ";
    s4.reserve(string::npos);
  }
  catch (const exception& e)
  { cerr << "\nexception : " << e.what() << endl;
  }
  return 0;
}
```

- **Ausgabe des Programms `stringdem2`**

```
Busch Monkey

 bricht das Voelkerrecht

Busch bricht das Voelkerrecht


4d 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
4d 6f 6e 6b 65 79
exception : invalid string position

reserve(ffffffff) :
exception : string too long
```

Demo-Programm 3 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (1)

- **C++-Quelldatei `stringdem_3.cpp` (`main()`-Funktion des Programms `stringdem3`)**

```
// C++-Quelldatei stringdem3_m.cpp
// Demonstrationsprogramm stringdem3 zur Bibliotheks-Klasse string

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

int main(int argc, char** argv)
{
  int iRet=0;
  string dname;
  if (argc>=2)
    dname=argv[1];
  else
  { cout << "\nName der Textdatei ? ";
    cin >> dname;
  }

  ifstream idat(dname.c_str());
  if (!idat)
  {
    cout << "\nDatei \"" << dname << "\" kann nicht geoeffnet werden !\n";
    iRet=1;
  }
  else
  {
    string text;
    string zeile;
    string grzeil;

    while (getline(idat, zeile))
    {
      if (zeile>grzeil)
        grzeil=zeile;
      text+=zeile;
      text+='\n';
    }
    cout << "\nInhalt der eingelesenen Textdatei :\n";
    cout << endl << text << endl;
    cout << "Gesamte Textlaenge : " << text.length() << endl;
    cout << "\n\"Groesste\" Zeile :\n";
    cout << grzeil << endl;

    string wort;
    cout << "\nSuchen nach ? ";
    cin >> wort;
    string::size_type pos=text.find(wort);
    // Haeufig trifft man auch folgendes an:
    //   size_t pos=text.find(wort);
    while(pos!=string::npos)
    {
      cout << "enthalten ab Pos. : " << pos << endl;
      text.insert(pos, 1, '*');
      pos+=1+wort.length();
      text.insert(pos++, 1, '*');
      pos=text.find(wort, pos+1);
    }
    cout << "\nMarkierte Textdatei :\n";
    cout << endl << text << endl;
  }
  return iRet;
}
```

Demo-Programm 3 zur ANSI/ISO-C++-Standardbibliotheks-Klasse `string` (2)

- **Ausgabe des Programms `stringdem3` (Beispiel-Aufruf : `stringdem3 hausverbot.txt`)**

```
Inhalt der eingelesenen Textdatei :

Hausverbot fuer Bush und Blair in Geburtskirche
US-Praesident George Bush und der britische Premierminister Tony Blair haben
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.
Dies betonte der palaestinensische Archimandrit Attalah Hanna vom
griechisch-orthodoxen Patriarchat in Jerusalem bei "Islam-online".
Auch US-Verteidigungsminister Donald Rumsfeld und der britische
Aussenminister Jack Straw duerften das Gotteshaus niemals mehr betreten.
Gleichzeitig sprach der Archimandrit, ranghoechstes Mitglied des
griechisch-orthodoxen Patriarchats, auch von einer "Exkommunikation".
Bush und Blair haetten sich selbst aus der Kirchengemeinschaft
ausgeschlossen, weil sie die Warnungen aller christlichen Kirchen vor
einem Irak-Krieg missachtet haetten, sagte der Geistliche.
2. April 2003 (SZ)

Gesamte Textlaenge : 817

"Groesste" Zeile :
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.

Suchen nach ? und
enthalten ab Pos. : 21
enthalten ab Pos. : 76
enthalten ab Pos. : 379
enthalten ab Pos. : 615

Markierte Textdatei :

Hausverbot fuer Bush *und* Blair in Geburtskirche
US-Praesident George Bush *und* der britische Premierminister Tony Blair haben
lebenslanges Hausverbot fuer die Geburtskirche in Bethlehem erhalten.
Dies betonte der palaestinensische Archimandrit Attalah Hanna vom
griechisch-orthodoxen Patriarchat in Jerusalem bei "Islam-online".
Auch US-Verteidigungsminister Donald Rumsfeld *und* der britische
Aussenminister Jack Straw duerften das Gotteshaus niemals mehr betreten.
Gleichzeitig sprach der Archimandrit, ranghoechstes Mitglied des
griechisch-orthodoxen Patriarchats, auch von einer "Exkommunikation".
Bush *und* Blair haetten sich selbst aus der Kirchengemeinschaft
ausgeschlossen, weil sie die Warnungen aller christlichen Kirchen vor
einem Irak-Krieg missachtet haetten, sagte der Geistliche.
2. April 2003 (SZ)
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (1)

• Allgemeines

◇ **Funktionsobjekte** sind Objekte, für die der Funktionsaufruf-Operator **`operator()`** definiert ist.
Derartige Objekte lassen sich **wie Funktionen verwenden**.
Gegenüber normalen Funktionen können sie einen inneren Zustand (Datenkomponenten) besitzen.
Dadurch lassen sie sich flexibler und effizienter als diese verwenden.

◇ In der **Utility-Bibliothek** sind zahlreiche Klassen für Funktionsobjekte, die diverse Standard-Operationen implementieren, definiert.
Um eine von den Operanden-Typen der jeweiligen Operation unabhängige Formulierung zu ermöglichen, sind sie als **Klassen-Templates** (mittels `struct`) implementiert.
In erster Linie sind sie für die Verwendung mit den Algorithmen der STL vorgesehen.
Sie lassen sich aber auch unabhängig davon einsetzen.

◇ U.a. stellt die Bibliothek für sämtliche in der Sprache enthaltenen **arithmetischen**, **Vergleichs-** und **logischen Operatoren** (Standard-Operationen) entsprechende Funktionsobjekt-Klassen zur Verfügung.

◇ Ihre Definitionen (und Implementierungen) sind in der **Headerdatei `<functional>`** enthalten.

• Basisklassen für Funktionsobjekte

◇ Es ist je eine Basisklasse für "**einstellige Funktionsobjekte**" und für "**zweistellige Funktionsobjekte**" definiert.
Für "einstellige Funktionsobjekte" wird die Operatorfunktion `operator()` mit einem Parameter aufgerufen,
für "zweistellige Funktionsobjekte" analog mit zwei Parametern.

◇ Durch die Basisklassen werden im wesentlichen nur standardisierte **Namen** für die **Parameter-** und **Rückgabe-Typen** definiert.
Diese Typnamen werden für die Definition der – ebenfalls in der Utility-Bibliothek implementierten – **Binder-Klassen** benötigt.

◇ **Basisklasse für "einstellige Funktionsobjekte"**

```
template <class Arg, class Result>
  struct unary_function
  {
    typedef Arg    argument_type;
    typedef Result result_type;
  }
```

◇ **Basisklasse für "zweistellige Funktionsobjekte"**

```
template <class Arg1, class Arg2, class Result>
  struct binary_function
  {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  }
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (2)

- **Funktionsobjekt-Klassen für arithmetische Operationen**

  ◇ **Prinzipielle Definition** der Klassen-Templates :

```
template <class T>
  struct plus : binary_function<T, T, T>
  {
    T operator()(const T& x, const T& y) const
    { return (x + y);
    }
  }
```

  ◇ **Überblick**

Funktionsobjekt-Klasse	Parameter von `operator()`	Ergebnis von `operator()`
plus<T>	(const T& **x**, const T& **y**)	**x + y**
minus<T>	(const T& **x**, const T& **y**)	**x - y**
multiplies<T>	(const T& **x**, const T& **y**)	**x * y**
divides<T>	(const T& **x**, const T& **y**)	**x / y**
modulus<T>	(const T& **x**, const T& **y**)	**x % y**
negate<T>	(const T& **x**)	**- x**

- **Funktionsobjekt-Klassen für Vergleichs- und logische Operationen**

  ◇ Die Operatorfunktion `operator()` dieser Klassen liefern einen Funktionswert vom Typ `bool`. Funktionsobjekte mit dieser Eigenschaft werden **Prädikate** genannt.

  ◇ **Prinzipielle Definition** der Klassen-Templates :

```
template <class T>
  struct equal_to : binary_function<T, T, bool>
  {
    bool operator()(const T& x, const T& y) const
    { return (x == y);
    }
  }
```

  ◇ **Überblick**

Funktionsobjekt-Klasse	Parameter von `operator()`	Ergebnis von `operator()`
equal_to<T>	(const T& **x**, const T& **y**)	**x == y**
not_equal_to<T>	(const T& **x**, const T& **y**)	**x != y**
greater<T>	(const T& **x**, const T& **y**)	**x > y**
less<T>	(const T& **x**, const T& **y**)	**x < y**
greater_equal<T>	(const T& **x**, const T& **y**)	**x >= y**
less_equal<T>	(const T& **x**, const T& **y**)	**x <= y**
logical_and<T>	(const T& **x**, const T& **y**)	**x && y**
logical_or<T>	(const T& **x**, const T& **y**)	**x || y**
logical_not<T>	(const T& **x**)	**! x**

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (3)

- **Einfaches Demo-Programm**

```
// C++-Quelldatei bibfodem1_m.cpp
// Modul mit main()-Funktion fuer das Programm bibfuncobjdem1
// Einfaches Demo-Programm zu Funktionsobjekten der Standard-Bibliothek

#include <functional>
#include <iostream>
using namespace std;


void testfunc();

int main(void)
{
  plus<int> iadd;        // Definition des Funktionsobjekts iadd
  int i1, i2, i3, i4;
  i1=5;
  i2=6;
  i3=10;
  i4=iadd(i1, i2);       // Aufruf operator() für Objekt iadd

  bool gleich;
  gleich=equal_to<int>()(i4, i3); // Erzeugung eines namenlosen
                                  // temporären Funktionsobjektes
                                  // Aufruf operator() fuer dieses Objekt
  cout << boolalpha ;
  cout << "\nUngleichheit von (" << i1 << "+" << i2 << ") und ";
  cout << i3 << " : " << logical_not<bool>()(gleich) << endl;
  testfunc();
  return 0;
}

template<class T, class Func>
   T binop(const T& a, const T& b, Func f)
   { return f(a, b);
   }

double quadsum(double a, double b)
{ return a*a + b*b; }

void testfunc()
{
  cout << "\nAddition   : "
       << binop(3.5, 2.7, plus<double>());     // Uebergabe Funktionsobjekt
  cout << "\nDivision   : "
       << binop(37, 7, divides<int>());        // Uebergabe Funktionsobjekt
  cout << "\nQuad-Summe : "
       << binop(3.2, 4.3, quadsum) << endl;    // Uebergabe Funktionspointer
}
```

Ausgabe des Programms :

```
Ungleichheit von (5+6) und 10 : true

Addition   : 6.2
Division   : 5
Quad-Summe : 28.73
```

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (4)

• Funktions-Adapter

◇ In der Utility-Bibliothek sind – ebenfalls in der Headerdatei **`<functional>`** – auch so genannte **Funktions-Adapter** definiert.
Dies sind Klassen (genauer : Klassen-Templates), die es auf einfache Art und Weise ermöglichen, neue Funktions-Objekte durch Modifikation anderer Funktionsobjekte bzw. durch Anpassung von Funktions-Pointern zu erzeugen.

◇ Es gibt vier **Gruppen von Funktions-Adaptern** :

- ▻ **Binder** (*binders*)
  Sie erzeugen aus einem zweistelligen Funktionsobjekt ein einstelliges Funktionsobjekt, indem ein Argument der Operatorfunktion `operator()` an einen bestimmten festen Wert gebunden wird.
- ▻ **Memberfunktionsadapter** (*adapters for pointers to members*)
  Sie ermöglichen es, dass eine Memberfunktion wie ein Funktionsobjekt als Argument eines Algorithmus verwendet werden kann.
- ▻ **Funktionszeigeradapter** (*adapters for pointers to functions*)
  Sie ermöglichen es, dass eine freie (oder statische Member-) Funktion wie ein Funktionsobjekt als Argument eines Algorithmus verwendet werden kann.
- ▻ **Negierer** (*negators*)
  Sie invertieren ein Prädikat.

◇ Funktions-Adapter sind auch miteinander **kombinierbar**.

◇ **Funktions-Adapter** sind als **Klassen-Templates** definiert.
Template-Parameter ist die Klasse des umzuwandelnden Funktionsobjektes.
Dem Konstruktor wird – gegebenenfalls zusammen mit weiteren Parametern – das umzuwandelnde Funktions-Objekt als Parameter übergeben. Er speichert dieses in einer Datenkomponente.
Die Operator-Funktion `operator()` führt die modifizierte Operation unter Verwendung des gespeicherten Funktionsobjekts aus.
Zu jedem Adapter gehört eine geeignete **Umwandlungsfunktion** (definiert als Funktions-Template), die aus einem als Argument übergebenen Funktionsobjekt ein Funktionsobjekt vom Adapter-Typ erzeugt.
Die jeweilige Umwandlungsfunktion stellt die **Anwendungsschnittstelle** eines Funktions-Adapters dar.

• Überblick über die Funktions-Adapter

Funktionsdapter-Klasse	Umwandlungsfunktion	Gruppe
`binder2nd`	`bind2nd()`	Binder
`binder1st`	`bind1st()`	
`mem_fun_t`	`mem_fun()`	Memberfunktionsadapter
`const_mem_fun_t`	`mem_fun()`	
`mem_fun1_t`	`mem_fun()`	
`const_mem_fun1_t`	`mem_fun()`	
`mem_fun_ref_t`	`mem_fun_ref()`	
`const_mem_fun_ref_t`	`mem_fun_ref()`	
`mem_fun1_ref_t`	`mem_fun_ref()`	
`const_mem_fun1_ref_t`	`mem_fun_ref()`	
`pointer_to_unary_function`	`ptr_fun()`	Funktionszeigeradapter
`pointer_to_binary_function`	`ptr_fun()`	
`unary_negate`	`not1()`	Negierer
`binary_negate`	`not2()`	

ANSI/ISO-C++-Standardbibliothek : Funktionsobjekte (5)

- **Beispiel für einen Binder : Umwandlungsfunktion `bind2nd` (Adapter `binder2nd`)**

  ◇ Erzeugt **aus** einem **zweistelligen Funktionsobjekt** und einem Wert `y` ein **einstelliges Funktionsobjekt**, in dem das zweite Argument (des zweistelligen Funktionsobjekts) an den Wert `y` gebunden wird

  ◇ **Definition des Klassen-Templates `binder2nd`**

```
template<class Operation>
  class binder2nd
    : public unary_function<typename Operation::first_argument_type,
                            typename Operation::result_type>
{ protected :
    Operation op;
    typename Operation::second_argument_type value;
  public :
    binder2nd(const Operation& binop,
              const typename Operation::second_argument_type& y)
      : op(binop), value(y) { }
    typename Operation::result_type  operator()
      (const typename Operation::first_argument_type& x) const
    { return op(x, value); }
};
```

  ◇ **Definition der Umwandlungsfunktion `bind2nd`**

```
template<class Operation, class T>
  binder2nd<Operation> bind2nd(const Operation& binop, const T& y)
  { return
      binder2nd<Operation>(binop, typename Operation::second_argument_type(y));
  }
```

  ◇ **Beispiel :**

  `less<int>()`
  ist ein – namenloses – **zweistelliges Funktionsobjekt** (expliziter Konstruktoraufruf) für den Vergleich zweier `int`-Werte.
  Seiner Operatorfunktion `operator()(a, b)` müssen die beiden zu vergleichenden Werte `a und b` als Parameter übergeben werden..

  mittels
  `bind2nd(less<int>(), 9)`
  wird hieraus ein **einstelliges Funktionsobjekt**, das einen `int`-Wert mit dem Wert `9` vergleicht.
  Der Operatorfunktion `operator()(a)` dieses Objekts ist nur der Wert `a` als einziger Parameter zu übergeben.