Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Übung05: Informationstechnik (IT)

Tobias Schwalb & Timo Sandmann & Stephan Werner

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Vererbung & Polymorphie, Strings und STL

Inhalt: Übung05

Karlsruher Institut für Technologie

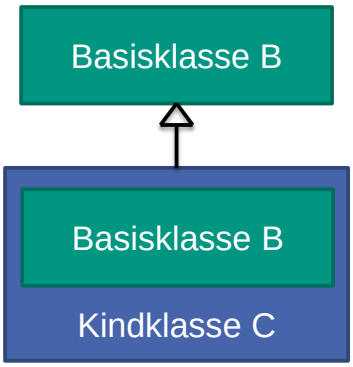
1. Besprechung Übungsaufgaben 4.01-4.04
2. Zulassungsaufgabe
3. Vererbung / Polymorphie
4. Strings
5. STL

Vererbung

- Vererbung ermöglicht eine Reduktion des Quellcodes und die Erstellung einer logischen Hierarchie im Code
- Eine Basisklasse vererbt der abgeleiteten Klasse (Kindklasse) alle ihre **`public`** und **`protected`** Methoden und Attribute
  - Kindklasse hat die ganze Funktionalität der Basisklasse
  - Kann zusätzlich um weitere Attribute und Methoden ergänzt werden

- Beispiel:

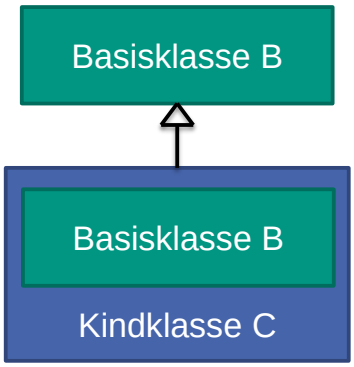
```
class C : public B {
  private:
    /* Deklaration der zusätzlichen privaten
       Datenelemente und Elementfunktionen */
  public:
    /* Deklaration der zusätzlichen öffentlichen
       Datenelemente und Elementfunktionen */
};
```



Vererbung - Zugriffsrechte

- Alle **`public`** Elemente der Basisklasse B sind auch in der abgeleiteten Kindklasse C öffentlich

- Methoden Kindklasse C können allerdings nicht auf die **`private`** Elemente der Basisklasse B zugreifen (nur über dessen öffentliche Methoden)



- Beispiel:

```
class Kfz {
  private:
    long nr;
    string hersteller;

  public:
    long getNr();
    //...
};
```

```
class Pkw : public Kfz {
  private:
    string pkwTyp;
    bool schiebe;

  public:
    void display( void );
    //...
};
```

```
void Pkw::display( void ) {
  cout << "Hersteller: "
       << hersteller << endl;
```
Privates Attribut der Basisklasse
```
  cout << "Kfz-Nummer: "
       << getNr() << endl;
```
Öffentliche Methode der Basisklasse
```
  cout << "Typ: " << pkwTyp;
}
```
Privates Attribut der Kindklasse

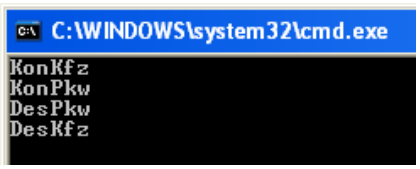
Vererbung - Redefinition

- In der Kindklasse können Methoden und Attribute redefiniert / überladen werden
  - Gleicher Namen, wie Methoden bzw. Attribute der Basisklasse
  - Elemente der Basisklasse werden verdeckt, sind aber verfügbar
- Der Zugriff auf Basisklassen-Methoden erfolgt über den Bereichsoperator `::`
- Nützliches Werkzeug in Bezug auf Polymorphie (ab Folie 9)

```
class Kfz {
  public:
    void display( void );
    //...
};

class Pkw : public Kfz {
  public:
    void display( void );
    //...
};

int main() {
  Pkw fahrzeug;

  fahrzeug.display();

  fahrzeug.Kfz::display();
  return 0;
}
```


Vererbung - Konstruktoren & Destruktoren

- Beim Anlegen bzw. Zerstören einer abgeleiteten Klasse wird auch der Konstruktor bzw. Destruktor der Basisklasse aufgerufen
- Dabei gilt die Reihenfolge:
  1. Konstruktor der Basisklasse
  2. Konstruktor der Kindklasse
  3. Verwendung der Klasse
  4. Destruktor der Kindklasse
  5. Destruktor der Basisklasse



- Beispiel:

```
class Kfz {
  public:
    Kfz() { cout << "KonKfz" << endl; }
    ~Kfz() { cout << "DesKfz" << endl; }
};

class Pkw : public Kfz {
  public:
    Pkw() { cout << "KonPkw" << endl; }
    ~Pkw() { cout << "DesPkw" << endl; }
};

int main() {
  Pkw fahrzeug;
  return 0;
}
```

Protected - Bezeichner

- Attribute und Methoden einer Basisklasse sollen einerseits vor dem Zugriff von außen geschützt sein – aber von der Kindklasse aus zugreifbar
- Verwendung der **`protected`**-Deklaration
  - Methoden von abgeleiteten Klassen können auf **`protected`** Elemente der Basisklasse zugreifen
  - **`protected`** Elemente sind allerdings von außerhalb der Klasse nicht zugreifbar

- Beispiel:

```
class Kfz {
  protected:
    bool tunen;
    //...
};

class Pkw : public Kfz {
  public:
    void display( void );
    //...
};

void Pkw::display( void ) {
  cout << tunen << endl;
}

int main() {
  Pkw fahrzeug;
  fahrzeug.display();
  //fahrzeug.tunen = true;
  return 0;
}
```

OK - Protected Attribut aus Methode der Kindklasse zugreifbar

Fehler - Protected Attribut von außen nicht zugreifbar

Typumwandlung / Konvertierung

- Jedem Objekt einer Basisklasse kann ein Objekt einer Kindklasse zugewiesen werden → implizite Typumwandlung
  - Inhalt der entsprechenden Attribute wird kopiert
  - Bsp: **`Pkw fahrzeug;  Kfz auto;  auto = fahrzeug;`**
- Zeiger auf die Basisklasse kann auch eine Kindklasse referenzieren
  - Zeiger kann nur auf öffentliche Elemente der Basisklasse zugreifen
  - Bsp: **`Kfz* kfzPtr = &fahrzeug;`**
- Umkehrung nur durch explizite Typumwandlung möglich
  - Programmierer muss auf den korrekten Objekttyp achten
  - Bsp: **`( (Pkw*) kfzPtr )->display();`**

OK – Nur Klammern beachten

```
Kfz* zweiterKfzPtr = &auto;
( (Pkw*) zweiterKfzPtr )->display();
```

Laufzeitfehler → Programmabsturz

Polymorphie - Problemstellung

- Problemstellung Beispiel:

```
class Kfz {
  public:
    void hupen( void );
    ~Kfz() {}
    //...
};

class Pkw : public Kfz {
  public:
    void hupen( void );
    //...
};

class Lkw : public Kfz {
  public:
    void hupen( void );
    //...
};

void Kfz::hupen( void ) {
  cout << "hup" << endl;
}

void Pkw::hupen( void ) {
  cout << "Tuuto" << endl;
}

void Lkw::hupen( void ) {
  cout << "Troeoet" << endl;
}

int main() {
  Kfz* autoPark[3];
  autoPark[0] = new Pkw();
  autoPark[1] = new Lkw();
  autoPark[2] = new Lkw();

  for( int i = 0; i < 3; i++ ) {
    autoPark[i]->hupen();
  }
  for( int i = 0; i < 3; i++ ) {
    delete autoPark[i];
  }
  return 0;
```

Wie können die Fahrzeuge richtig hupen ?

Polymorphie

- Normalerweise werden Methoden während des Kompilierens entsprechend dem Typ des Zeigers ausgewählt
  - Im Beispiel war **`autoPark`** vom Typ **`Kfz*`**, daher wurde die Methode **`hupen()`** von der Klasse **`Kfz`** verwendet
  - Andere Methode durch Typcast möglich – ABER: Verschiedene Kindklassen im Array abgelegt → Wie richtige Typ/Methode auswählen?!
- Methoden anhand des Typs des Objekts zur Laufzeit automatisch auswählen mit dem Schlüsselwort **`virtual`**
  - Methode muss in der Basisklasse als **`virtual`** deklariert werden
  - Methode kann (muss aber nicht) in den Kindklassen redefiniert werden
  - Entscheidung geschieht zur Laufzeit (Late Binding)
  - Besonders wichtig auch bei Destruktoren
  - Virtuelle Methoden kosten etwas mehr Speicherplatz und sind etwas langsamer – Vorteile überwiegen allerdings

Polymorphie - virtual

- Problemstellung Beispiel:

```
class Kfz {
  public:
    virtual void hupen( void );
    virtual ~Kfz() {}
    //...
};

class Pkw : public Kfz {
  public:
    void hupen( void );
    //...
};

class Lkw : public Kfz {
  public:
    void hupen( void );
    //...
};

void Kfz::hupen( void ) {
  cout << "hup" << endl;
}
```

Einzige Änderungen

```
void Pkw::hupen( void ) {
  cout << "Tuuto" << endl;
}

void Lkw::hupen( void ) {
  cout << "Troeoet" << endl;
}

int main() {
  Kfz* autoPark[3];
  autoPark[0] = new Pkw();
  autoPark[1] = new Lkw();
  autoPark[2] = new Lkw();

  for( int i = 0; i < 3; i++ ) {
    autoPark[i]->hupen();
  }

  for( int i = 0; i < 3; i++ ) {
    delete autoPark[i];
  }
  return 0;
}
```

```
C:\WINDOWS\
Tuuto
Troeoet
Troeoet
```


Zwischenübung01: Vererbung & Polymorphie

Bitte zuordnen?

```
//Anweisungsblock A:
irgendeinePflanze = meinBaumBernd;
irgendeinePflanze->vergammel();
```
1

```
//Anweisungsblock A:
irgendeinePflanze = narzisse;
((Blume*)irgendeinePflanze)->vergammel();
```
2

```
//Anweisungsblock A:
meinBaumBernd->bringeSamen();
```
3

```
//Anweisungsblock A:
einfachePflanze->bringeSamen();
```
4

```
C:\WINDOWS\system32\cmd.exe
Ich bin eine grundlegende Pflanze.
```

```
C:\WINDOWS\system32\cmd.exe
oh nein, meine Sch÷nheit ist dahin!
```

```
C:\WINDOWS\system32\cmd.exe
oh ich zerfalle zu Humus
```

```
C:\WINDOWS\system32\cmd.exe
ich habe Samen geworfen.
```

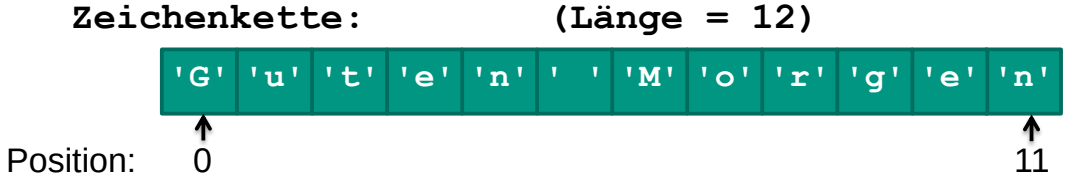
```
class Pflanze {
  public:
    Pflanze(){}
    virtual ~Pflanze(){}
    void vergammel();
    virtual int bringeSamen();
};
class Blume : public Pflanze {
  public:
    void vergammel();
    int bringeSamen();
};
class Baum : public Pflanze {
  public:
    int bringeSamen();
};

void Pflanze::vergammel() {
  cout << "oh ich zerfalle zu Humus" << endl;
}
int Pflanze::bringeSamen() {
  cout << "Ich bin eine grundlegende Pflanze." <<endl;
  return 1;
}
int Blume::bringeSamen() {
  cout << "Ich habe geblüht. Viele Samen." << endl;
  return 400;
}
void Blume::vergammel() {
  cout << "oh nein, meine Schönheit ist dahin!" << endl;
}
int Baum::bringeSamen() {
  cout << "ich habe Samen geworfen." << endl;
  return 5000;
}
```

```
int main() {
  Pflanze* einfachePflanze = new Pflanze();
  Baum* meinBaumBernd = new Baum();
  Blume* narzisse = new Blume();
  Pflanze* irgendeinePflanze;
  //Anweisungsblock A
  return 0;
}
```

Strings - Allgemein

- Allgemein
  - Einbinden der Bibliothek **<string>**
  - Zur Darstellung und Verarbeitung von Zeichenketten in C++
  - Speicherplatz wird automatisch reserviert und angepasst
- Initialisierung / Zuweisung
  - Ohne Initialisierung ist der String leer (kein zufälliger Inhalt)
  - String aus der Konsole einlesen mit **cin**
    - Operator **>>** immer nur ein Wort (ohne führende Leerzeichen)
    - **getline()** einlesen einer kompletten Zeile (mit Leerzeichen)

Beispiel:
```
string meldung = "Guten Morgen";
```



Strings und C-Strings

- C-Strings:
  - Char-Arrays zur Speicherung von Zeichenfolgen
  - Gespeichert wird: Zeichenfolge + String-Endezeichen **'\0'**
  - Spezielle Funktionen **strlen(), strcpy(), strcmp()**

Beispiel:
```
char name[] = "Evi"; //oder
char name[] = { 'E', 'v', 'i', '\0' };
```

- Umwandelung von Strings in C-Strings (für spezielle Funktionen):
  - Anhand der speziellen Methode **c_str()** der Klasse **string**
  - Rückgabewert dieser Methode ist vom Typ **const char***

Beispiel:
```
string name = "ADAM";
double x = atof( name.c_str() );
```

Strings verketten und vergleichen

- Strings verketten mit Operator **+**
  - Mit String-Variablen, String-Konstanten, einzelnen Zeichen
  - Ein Operand muss vom Typ **`string`** sein
  - Anhängen mit dem Operator **+=**

Beispiel:
```
string erg, s1 = "Kaffee", s2 = "ta";
erg = s1 + s2 + "ss" + 'e';  //erg = "Kaffeetasse"
```

- Strings vergleichen mit Operatoren **< <= == != > >=**
  - Vergleich erfolgt zeichenweise anhand der ASCII-Tabelle
  - Liefert ein Ergebnis von Typ **`bool`** (**`true`** oder **`false`**)

Beispiel:
```
string s3 = "A", s4 = "a", s5 = "Aal";
s3 > s4;    s3 < s5;     s3 + s4 + 'l' == s5;
 False        True                    True
```


Methoden für Strings(1)

- Benutzen von Methoden mit dem Punkt-Operator

- Einfügen mit Methode **`insert()`**
  - Zeichenkette an eine bestimmte Position im String einfügen

Beispiel:
```
erg.insert( 6, "maschinencode", 0, 9 );
```
Startposition — Startposition des einzufügenden Strings (optional) — String zum Einfügen — Anzahl der einzufügenden Zeichen (optional)

```
//erg == "Kaffeetasse"        //erg == "Kaffeemaschinentasse"
```

- Löschen mit Methode **`erase()`**
  - Löschen einer Anzahl an Zeichen in einem String

Beispiel:
```
erg.erase( 9, 9 );
```
Startposition — Anzahl der zu löschenden Zeichen (optional)

```
//erg == "Kaffeemaschinentasse"        //erg == "Kaffeemasse"
```

Methoden für Strings(2)

- Suchen mit Methode **`find()`**
  - Suchen einer Zeichenfolge in einem String, Rückgabe der Position
  - Gibt **`-1`** bzw. **`string::npos`** zurück, falls die Zeichenfolge nicht gefunden wurde

Zu suchender String — Startposition (optional)

Beispiel:
```
int pos = erg.find( 'a', 3 );
//erg == "Kaffeemasse"                    //pos == 7
```

- Ersetzen mit Methode **`replace()`**
  - Ersetzen eines Teilstrings durch einen anderen String

Startposition — Startposition des einzufügenden Strings (optional)

Beispiel:
```
erg.replace( 6, 5, "maschinencode", 0, 8 );
```
Länge zum Ersetzen — Anzahl der einzufügenden Zeichen (optional)

```
//erg == "Kaffeemasse"          //erg == "Kaffeemaschine"
```


Methoden für Strings(3)

- Länge mit Methode **`length()`**
  - Kein Argument, Anzahl der Zeichen als Rückgabewert

Beispiel:
```
int strlang = erg.length();
//erg == "Kaffeemaschine"                 //strlang == 14
```

- Teilstring mit Methode **`substr()`**
  - Rückgabe eines Teilstrings eines anderen Strings

Startposition des Strings

Beispiel:
```
string teilA = erg.substr( 5, 4 );
```
Länge des Strings (optional)

```
//erg == "Kaffeemaschine"          //teilA == "emas"
```

Zugriff auf Zeichen in Strings

- Zugriff mit dem Index-Operator `[]` oder Methode `at()`
- Zeichen wird anhand seines Index / Position identifiziert
  - Erstes Zeichen: `0` Letztes Zeichen: `s.length()-1`

Beispiel:

```
string s = "Tee";
char c = s[0];     //c == 'T'
//s[0] == 'T'  s[1] == 'e'  s[2] == 'e'
```

- Index muss ein Integer-Ausdruck sein
  - Beim Index-Operator erfolgt keine Fehlermeldung beim Kompilieren beim Überschreiten des Bereichs → Absturz / Fehlverhalten zur Laufzeit
  - Methode `at()` führt eine Bereichsprüfung durch → Auslösung einer Exception → kann abgefangen werden

Beispiel:

```
s.at( 3 ) = 'x'; //Auslösung einer Exception
```


Umwandlung eines Strings in eine Zahl mit `atof()`

- Die Funktion `atof()`
  - Konvertiert die übergebene Zeichenfolge (C-String) bis zum ersten Leerzeichen im String bzw. dem ersten unzulässigen Zeichen des Strings in einen `float`-Wert
  - Gibt 0 zurück, bei fehlerhafter Umwandlung
    - Nicht zu unterscheiden von `atof( "0" );`

Beispiel:

```
string text = "12.5 3";
double x = atof( text.c_str() );
//x == 12.5
```

Umwandlung eines Strings in eine Zahl mit `strtod()`

- Die Funktion **`strtod()`**
  - Konvertiert die übergebene Zeichenfolge (C-String) bis zum ersten Leerzeichen im String bzw. dem ersten unzulässigen Zeichen des Strings in einen **`double`**-Wert
  - Gibt 0 zurück, bei fehlerhafter Umwandlung
  - Bekommt zusätzlich ein **`char**`** übergeben zum Speichern der Adresse, wo eine Umwandlung nicht mehr möglich war

Beispiel:

```
char* rest;
string text = "12.5ab";
double x = strtod( text.c_str(), &rest );

//x == 12.5, rest == "ab"
```


Umwandlung eines Strings in eine Zahl mit `stringstream`

- Einbinden der Bibliothek **`<sstream>`**
- Variable vom Typ **`stringstream`** wird durch den **`<<`** Streamausgabeoperator gefüllt
- Jede Art von Variable kann mit dem **`>>`** Streameingabeoperator gefüllt werden
  - Mehrere Zahlen können mit einem Leerzeichen getrennt werden
  - Falls der String unzulässige Zeichen (keine Zahlen) beinhaltet wird beim ersten unzulässigen Zeichen abgebrochen
  - Methode **`fail()`** der Klasse **`stringstream`** gibt beim fehlerhaften Umwandeln einer Variablen **`true`** zurück

Beispiel:

```
stringstream strstr;
strstr << "12 64.3 49 99 200";
double wert1 = 0, wert2 = 0;
strstr >> wert1;
strstr >> wert2;
cout << strstr.fail() << endl;

//wert1 == 12, wert2 == 64.3
//fail() == false
```

```
stringstream strstr;
strstr << "12 a 64";
double wert1 = 0, wert2 = 0;
strstr >> wert1;
strstr >> wert2;
cout << strstr.fail() << endl;

//wert1 == 12, wert2 == 0
//fail() == true
```

Methoden für Strings

- Benutzen von Methoden mit dem Punkt-Operator
  - **`insert()`:** Zeichenkette an eine Position im String einfügen
  - **`erase()`:** Löschen einer Anzahl an Zeichen in einem String
  - **`find()`:** Suchen einer Zeichenfolge, Rückgabe derPosition
  - **`replace()`:** Ersetzen eines Teilstrings durch einen anderen String
  - **`length()`:** Anzahl der Zeichen als Rückgabewert
  - **`substr()`:** Extraktion eines Teilstrings aus einem String
  - **`at()`:** Zugriff auf ein einzelnes Zeichen in einem String
  - **`c_str()`:** Umwandeln eines Strings in einen C-String

- Nützliche Methoden in Zusammenhang mit Strings
  - **`atoi()`:** Umwandlung eines C-Strings in eine Integer-Zahl
  - **`atof()`:** Umwandlung eines C-Strings in eine Float-Zahl
  - **`strtod()`:** Umwandlung eines C-Strings in eine Double-Zahl

Zwischenübung02: Strings

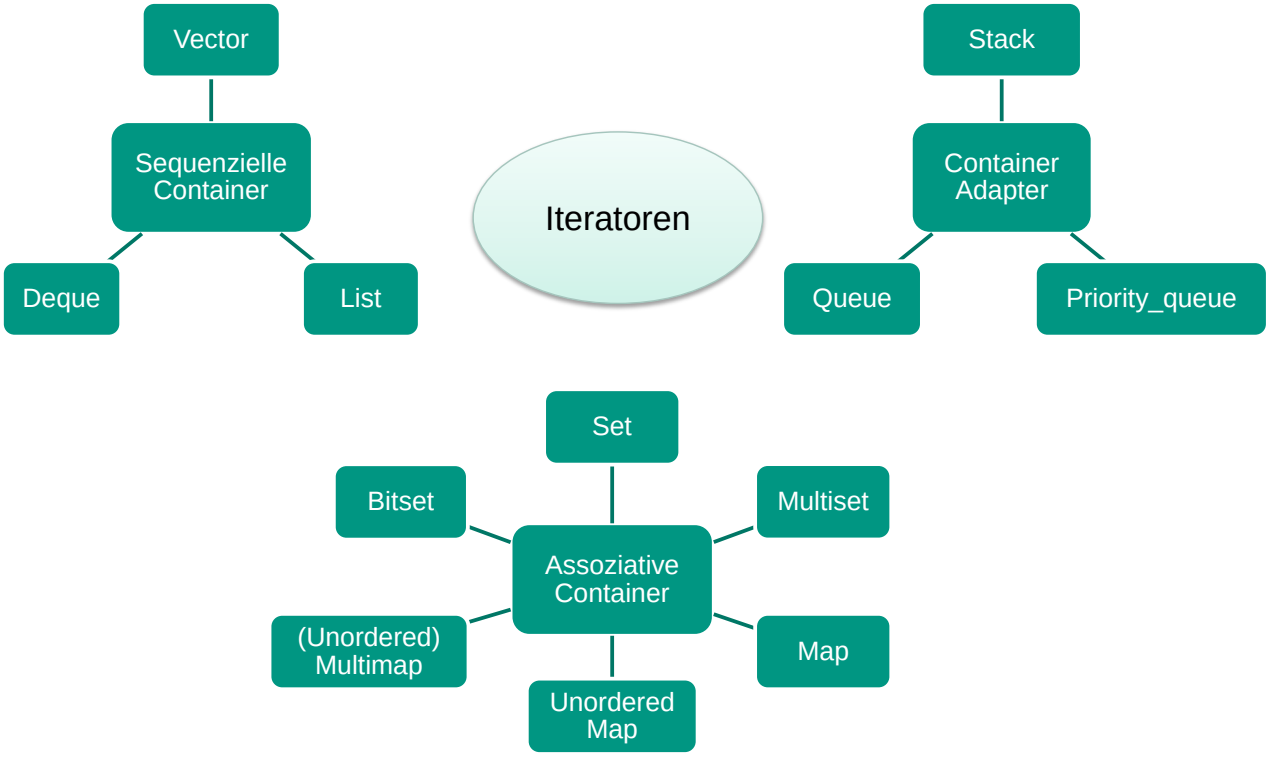
Welche Zeichenfolge speichert der String **`s`** *nach folgenden Anweisungen?*

a)
```
string s( "WOW!" );
int pos = s.find( "!" );
if( pos != string::npos ) {
  s.insert( pos, ", " + s, 0, 5 );
}
```

b)
```
string s = "winter sports";
s.erase( 0, 7 );
s.erase( 5, 1 );
```

c)
```
string s( "Super!!!" );
int n = s.rfind( "!!" );
s.replace( n - 1, 3, "novatni", 0, 4 );
```

Standard Template Library (STL)



STL – Container

- Speichert eine Sammlung von Objekten in einer Datenstruktur
- Sequenzielle Container
  - Vector
    - Speichert die Elemente in einer strikt linearen Reihenfolge
    - Elemente liegen kontinuierlich hintereinander im Speicher (wie bei Arrays)
    - Indizierter Zugriff auf die Elemente, effizientes Hinzufügen und Löschen von Elementen am **Ende**
  - Deque
    - **D**ouble-**e**nded **que**ue
    - Speichert die Elemente in einer strikt linearen Reihenfolge
    - Indizierter Zugriff auf die Elemente, effizientes Hinzufügen und Löschen von Elementen an **Anfang** und **Ende**
  - List
    - Doppelt verkettete Liste
    - Speichert die Elemente in einer linearen Reihenfolge
    - Effizientes Hinzufügen, Löschen und verschieben von Elementen an allen Positionen

STL – Container Adapter

- Container Adapter bieten ein einheitliches Interface für verschiedenartige Datenstrukturen
  - **`push(elem)`** - Fügt das Element **`elem`** in die Datenstruktur ein
  - **`top()`** - Liefert das erste Element des Containers
  - **`pop()`** - Löscht das erste Element des Containers
- Durch das Interface wird von der internen Implementierung abstrahiert
- Verwenden intern die sequenziellen Container
- Verfügbare Klassen in der STL:
  - Stack
    - LIFO Prinzip (last-in first-out)
    - Verwendet standardmäßig **`deque`**, auch **`vector`** oder **`list`** möglich
  - Queue
    - FIFO Prinzip (first-in first-out)
    - Verwendet standardmäßig **`deque`**, auch **`list`** möglich
  - Priority_queue
    - Implementiert eine Prioritätswarteschlange, **`top()`** liefert das größte Element
    - Verwendet standardmäßig **`vector`**, auch **`deque`** möglich

STL – Assoziative Container

- Assoziative Container
  - Set
    - Speichert eindeutige Elemente in sortierter Reihenfolge
    - Effizientes Suchen von beliebigen Elementen
  - Multiset
    - Wie **`set`**, aber Duplikate möglich
  - Map
    - Speichert Paar aus eindeutigem Schlüssel und Wert
    - Sortierung der Elemente (Paare) nach den Schlüsseln
    - Effizientes Suchen von beliebigen Elementen
  - Unordered_Map
    - Wie **`map`**, jedoch ohne Sortierung, dadurch effizienteres Einfügen und Suchen
  - (Unordered) Multimap
    - Wie **`map`** / **`unordered_map`**, aber Duplikate möglich
  - Bitset
    - Ähnlich einem Array, speichert jedoch Bits und belegt daher weniger Speicherplatz

STL - Iteratoren

- Iteratoren bieten einen einheitlichen Zugriff auf die Elemente der verschiedenen Container-Typen
- Einfachstes Beispiel: Zeiger beim Array-Zugriff
  - Iterieren über die Arrayelemente mit dem ++-Operator
  - Zugriff auf die Elemente über Dereferenzierung (* oder -> Operator)
- STL-Container stellen passende Iterator-Klassen bereit, um auf ihre Elemente zugreifen zu können
- Grundsätzliche Iterator-Kategorien
  - Input-Iterator: lesender Zugriff für einen Durchlauf
  - Output-Iterator: schreibender Zugriff für einen Durchlauf
  - Forward-Iterator: sequenzieller Zugriff lesend und schreiben in eine Richtung
  - Bidirectional-Iterator: wie Forward-Iterator jedoch in beide Richtungen
  - Random Access-Iterator: wahlfreier Zugriff, z.B. gewöhnliche Zeiger

STL – Iteratoren Beispiel

```
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<int> data(10);
  vector<int>::iterator it( data.begin() );

  for( int i = 0; it != data.end(); ++it, ++i ) {
    *it = 100 - i;
  }
  for( it = data.begin(); it != data.end(); ++it ) {
    cout << *it << " ";
  }
  cout << endl;

  sort( data.begin(), data.end() );

  for( it = data.begin(); it != data.end(); ++it ) {
    cout << *it << " ";
  }
  cout << endl;
  return 0;
}
```

```
C:\Windows\system32\cmd.exe
100 99 98 97 96 95 94 93 92 91
91 92 93 94 95 96 97 98 99 100
```

STL – list

- Anlegen
  - `#include <list>` - Einbinden der Bibliothek
  - `list<T> list_name;` - Legt die Liste `list_name` vom Typ `T` an
- Kapazitätsfunktionen
  - `bool empty() const;` - Ist die Liste leer?
  - `size_t size() const;` - Liefert die Anzahl der Elemente in der Liste
  - `size_t max_size() const;` - Maximal mögliche Anzahl an Elementen
  - `void resize(size_t sz);` - Vergrößert / verkleinert die Liste auf sz Elemente
- Elementzugriff
  - `T& front();` - Liefert eine Referenz auf das erste Element
  - `T& back();` - Liefert eine Referenz auf das letzte Element

STL – list

- Modifikationen
  - `void push_front( const T& x );` - Fügt eine Kopie von `x` als neues Element am Anfang ein
  - `void push_back( const T& x );` - Fügt eine Kopie von `x` als neues Element am Ende ein
  - `void pop_front();` - Löscht das erste Element
  - `void pop_back();` - Löscht das letzte Element
  - `iterator insert( iterator position, const T& x );` - Fügt eine Kopie von `x` als neues Element an Position `position` ein. Gibt einen Iterator auf das eingefügte Element zurück.
  - `iterator erase( iterator position );` - Löscht das Element an Position `position`. Gibt einen Iterator auf das nächste Element zurück.
  - `void remove( const T& value );` - Löscht das Element mit Wert `value` aus der Liste
  - `void clear();` - Löscht alle Elemente aus der Liste
  - `void sort();` - Sortiert die Liste aufsteigend

Ausblick

- Kompendium: Kapitel 8 – Objektorientierung
- Tutorium: Aufgabe 13 & 17

- Wie kann ich eine große Dateimenge in mein Programm einlesen?
- Wie funktioniert die Implementierung einer verketteten Liste?
- …

Vielen Dank für Ihre Aufmerksamkeit

Tobias Schwalb & Timo Sandmann & Stephan Werner
Karlsruher Institut für Technology (KIT) – ITIV
tobias.schwalb@kit.edu
timo.sandmann@kit.edu
stephan.werner@kit.edu