C++ Standardbibliothek

1

- Zu jedem Compiler gibt es eine Bibliothek mit nützlichen Klassen und Routinen.
- Die C++ Standardbibliothek (standard library) stellt ein erweiterbares Rahmenwerk mit folgenden Komponenten zur Verfügung:
  - Diagnose
  - Strings
  - Container
  - Algorithmen
  - komplexe Zahlen und numerische Algorithmen
  - Anpassung an nationale Zeichensätze
  - Ein-/Ausgabe und vieles mehr.
- Sie basiert zum Teil auf der Standard Template Library (STL), die von Hewlett-Packard (HP) entwickelt wurde.
- Im Rahmen der Vorlesung werden wir uns zunächst mit Containern und Iteratoren und gewissen Algorithmen beschäftigen.
- Wir werden nach und nach die in der Standardbibliothek vorhandenen Algorithmen diskutieren.

Breymann_Folien

Container

2

- Eine Behälterklasse (container) ist eine Datenstruktur zur Speicherung einer Anzahl von Objekten.
- In C gibt es zwei Arten von eingebauten Containern:
  - C-Felder enthalten gleichartige Objekte:

```
int  a[500];
```

  - Strukturen fassen logisch zusammengehörige Daten zusammen

```
struct Student {
                    string              name;
                    Geschlecht          geschlecht;
                    unsigned short      semesterzahl;
                    Studienfach         studienfach;
                    unsigned long       matrikelnummer;
                    unsigned short      uebungsnummer;
                    string              name_des_bremsers;
                    vector<int>         resultate_der_uebungen;
                    float               note;
            };
```

Breymann_Folien

Container

- In der C++ Standardbibliothek gibt es komplexere, aber komfortablere Behälterklassen:

Header-Name:

<vector>
<list>
<queue>
<stack>
<deque>
<map>
<set>
<bitset>

- Bevor wir uns die Funktionalität der wichtigsten Behälterklassen anschauen, betrachten wir kurz „eigene“ Implementierungen der Klasse „list“ und „queue“ und diskutieren das so genannte Iterator-Konzept.

Eine Beispielklasse für Listen

- Auf den folgenden Seiten stellen wir die Deklaration und einen Teil der Definition (Implementierung) einer Beispielklasse für Listen vor:

```
#ifndef list_t
#define list_t
#include<cstddef>                       // wegen NULL

template <class T>
class  List {
   public:
     List();                            // Defaultkonstruktor
     List(const List&);                 // Kopierkonstruktor
     ~List( );                          // Destruktor
     List& operator=(const List&);      // Zuweisungsoperator
```

Eine Beispielklasse für Listen


```
bool empty( ) const { return number == 0;}
int  size()       const { return number;}

// am Anfang bzw. am Ende einfügen
void push_front( const T&);
void push_back( const T&);

// am Anfang und Ende löschen
void pop_front();
void pop_back();

// am Anfang bzw. Ende lesen
T& front();
const T& front() const;
T& back();
const T& back() const;

// Anwenden von Funktionen f ( ) auf alle Elemente
void apply( void (*f)(T&));
```


Eine Beispielklasse für Listen


```
private:
   class ListElement  {                          // Klassendeklaration in der
              friend class List<T>;              // Klassendeklaration:
              T data;                            // „nested classes“
              ListElement *next, *prev;
              ListElement(const T& dat)
              :  data(dat), next(NULL), prev(NULL) { }
     };       // Ende der Deklaration von ListElement

   // Zeiger auf den Anfang und das Ende der Liste
   ListElement  *start, *end;

   // Anzahl der Listenelemente
   int  number;

};  //  Ende der Deklaration von List
```

Eine Beispielklasse für Listen


```
private:
   class ListElement  {                          // Klassendeklaration in der
            friend class List<T>;                // Klassendeklaration:
            T data;                              // „nested classes“
            ListElement *next, *prev;
            ListElement(const T& dat)
            :  data(dat), next(NULL), prev(NULL) { }
    };
```

- Die Struktur eines Listenelements wird durch die geschachtelte Klasse „ListElement“ beschrieben.
- Damit die Funktionen der Klasse „List“ auf die Daten der Klasse „ListElement“ zugreifen können, wird „List“ innerhalb von „ListElement“ als „friend“-Klasse deklariert.

Eine Beispielklasse für Listen


```
template<class T>                                // Defaultkonstruktor
List<T>::List()
: end(NULL), start(NULL), number(0) {  }

template<class T>                                // Kopierkonstruktor
List<T>::List(const List& L)
 : end(NULL), start(NULL), number(0) {
            ListElement *temp = L.end;
            while (temp) {
                     push_front(temp->data);
                     temp = temp->prev;
            }
}

template<class T> List<T>::~List()   {       // Übungsaufgabe   }
```

Eine Beispielklasse für Listen


```
template<class T>
List<T>::push_front(const T& dat) {
  ListElement *temp = new ListElement(dat);
  // der Kopierkonstruktor setzt temp->prev = NULL
  temp->next = start;
  if (!start)  end = temp;
  else start->prev = temp;
  start = temp;
  number++;
}

//  Funktion f auf alle Elemente der Liste anwenden
template<class T>
List<T>::apply(void(*f) (T&))  {
  ListElement *temp = start;
  while (temp) {
            f(temp->data);          // wende f auf das aktuelle Listenelement an
            temp = temp->next;
  }
}
```

Breymann_Folien

Eine Beispielklasse für eine Warteschlange

- Queues zeichnen sich durch folgende Eigenschaften aus:
  - Elemente können nur am Anfang eingefügt werden und
  - nur am Ende entnommen werden (FIFO: first in – first out)
- Zur Implementierung einer Queue können wir die Listenklasse verwenden.
- Das Listenobjekt L wird privat angelegt und die Elementfunktionen der Klasse Queue rufen die öffentlichen Elementfunktionen des Objekts L auf.
- Die Klasse Queue delegiert somit Aufgaben an die Klasse List (Prinzip: Delegation).

Breymann_Folien

Eine Beispielklasse für eine Warteschlange

11

```
#include “list.t“

template<class T>
class Queue {
        public:
                bool        empty( )   const     { return L.empty(); }
                int         size( )    const     { return L.size(); }

                void        push( const T& x)    { L.push_back(x); }   // am Ende einfügen

                void        pop()                { L.pop_front(); }    // am Anfang entnehmen

                T&          front()              { return L.front(); } // am Anfang bzw.
                const T&    front()    const     { return L.front(); } // am Ende lesen
                T&          back()               { return L.back(); }
                const T&    back()     const     { return L.back(); }


                void apply(void(*f)(T&) )        { L.apply(f); }       // f() anwenden

        private:
                List<T>  L;
};
```

Breymann_Folien

Eine Beispielklasse für eine Warteschlange

12

- Man beachte:
  - Durch die Delegation enthält jedes Queue-Objekt ein Objekt L vom Typ „List“.
  - Da wir einen Kopierkonstruktor für die Klasse List definiert haben, müssen wir für Queue keinen Kopierkonstruktor implementieren, da der vom Compiler zur Verfügung gestellte Kopierkonstruktor diese Aufgabe schon perfekt erledigt.
  - Der Default-Kopierkonstruktur ruft nämlich den Kopierkonstruktor der Klasse List für das einzige Datenelement L der Klasse Queue auf.
  - Analoge Aussagen gelten für den Zuweisungsoperator und den Destruktor der Klasse Queue.
  - Die Implementierungen einiger Elementfunktionen wird in den Übungen vorgestellt.

Breymann_Folien

Iteratoren

13

- Ein Iterator ist kein Typ und kein Objekt, sondern ein Konzept, ein Name, der auf eine Menge von Klassen und Typen zutrifft, die bestimmten Anforderungen entsprechen.
- Iteratoren dienen dazu, auf Elemente eines Containers zuzugreifen.
- Wir betrachten zunächst einen einfachen Iterator „MyIterator“ vom Typ „int*“ (Zeiger auf int), bevor wir uns dann eine Klasse „Iterator“ für Listen anschauen.
- Wir verwenden hierbei zum ersten Mal das Schlüsselwort „typedef“, das es dem Nutzer erlaubt neue Namen für komplexe Deklarationen einzuführen:

```
typedef  float  real;

int main( ) {
    real Zahl = 1.756;      // Zahl ist vom Typ float
}
```

- Wenn wir dasselbe Programm mit double Zahlen rechnen lassen wollen, brauchen wir nur die typedef-Zeile zu ändern:

```
typedef double real;
```

Breymann_Folien

Zeiger als Iterator

14

```
// include‘s und namespace lassen wir aus Platzgründen weg
typedef   int* MyIterator;                          // neuer Name MyIterator
typedef void (*Function) (int&);                    // Function

void print( int &x) {                                          // global
        cout << x;
}

void apply( MyIterator start, MyIterator end, Function f) {    // global
        while (start != end)  f(*start++);
}

int main( ) {
        const int number = 10;
        int array[number];
        for (size_t  i=0; i < number; i++)  array[i] = i * i;          // initialisieren

        MyIterator  start = array, end = array + number;
        apply(start, end, print);
}
```

Breymann_Folien

Zeiger als Iterator

15

```
void apply( MyIterator start, MyIterator end, Function f) {  // global
        while (start != end)  f(*start++);
}
```

- Das „Durchwandern“ einer Datenstruktur oder eines Containers in einer bestimmten Reihenfolge ist für viele Anwendungen sinnvoll, zum Beispiel, um auf jedes Element eine bestimmte Operation anzuwenden.
- Die obige Funktion „apply“ ähnelt der Funktion „for_each()“ aus der C++ Bibliothek:

```
template<class Iterator, class Function>
Function for_each(Iterator start, Iterator end, Function f) {
    while (start != end) f(*start++);
    return f;
}
```

- Die obige Funktion zeigt, welche Operationen (Operatoren) für einen Iterator benötigt werden:
  - Dereferenzieroperator *
  - Referenzieroperator &
  - Vergleichsoperatoren !=, ==
  - Inkrementoperatoren ++, -- und einige andere

Breymann_Folien

Zeiger als Iterator

16

- Beim trivialen Iterator „MyIterator“ sind diese Operatoren natürlich vorhanden und implementiert.
- Betrachten wir jedoch komplexe Datenstrukturen wie Listen oder (später) Bäume, so muss der Iterator einiges über den internen Aufbau der Datenstruktur wissen, da z.B. der operator++() nicht die nächste Speicheradresse, sondern einen Iterator auf das nächste Element liefern soll.
- Zunächst schauen wir uns noch ein weiteres Anwendungs-beispiel für einen Iterator für Listen an, bevor wir die Deklaration und die Definition des Iterators studieren werden.
- Im Beispielprogramm ist „ListIter“ der Iterator, der mit einer Liste L verknüpft wir.
- Sein Typ ist innerhalb der Klasse „List“ als geschachtelte Klasse definiert.

Breymann_Folien

Iterator für Listen


```
// include‘s und namespace lassen wir aus Platzgründen weg

void print( int& x) { cout << x; }

int main( ) {
        int  i;
        List<int> L;
        for ( i=0; i < 10; i++)  L.push_front(i*i);          // initialisieren

        List<int>::Iterator  ListIter;
        ListIter = L.begin();
        while ( ListIter != L.end()) {
                if (*ListIter == 36) {                       // 36 löschen, falls vorhanden
                        cout << *ListIter << “ wird gelöscht\n“;
                        L.erase(ListIter);
                        cout << *ListIter << “an aktueller Position\n“;
                        break;
                }
                else ++ListIter;
        }
        for_each( L.begin(), L.end(), print);                //  #include<algorithm>
}
```


Iterator für Listen


```
// die Implementierung der Klasse ListIter in List könnte wie folgt aussehen:

template<class T>
class List {
        public:
           // den Anfang unserer Listenklasse übernehmen wir an dieser Stelle
           // bis zum Ende von public, wo wir die folgenden Zeilen einfügen:
           class Iterator {
                   public:
                      friend class List<T>;

                      //  zunächst die Konstruktoren
                      Iterator(ListElement*  Init = NULL}
                      :  current(Init) {
                      }

                      Iterator(const List& L) {  // setze Iterator auf Anfang der Liste L
                        current = L.begin();
                      }

                      //  auf der nächsten Seite geht es weiter ✍
```

Iterator für Listen


```
// nun folgen die Dereferenzierungen
const T& operator*( ) const {
        return current->data;
}

T&  operator*( )  {
        return current->data;
}

// die Inkrementoperatoren
Iterator& operator++( ) {                          // präfix
        if (current) current = current->next;
        return *this;
}

Iterator& operator++(int) {                        // postfix
        Iterator temp = *this;
        ++*this;
        return temp;
}
```


Iterator für Listen


```
// Vergleichsoperatoren
bool operator==( const Iterator& x) const {
        return current == x.current;
}

bool operator!=(const Iterator& x )  {
        return current != x.current;
}

private:
  // das einzige Datenelement der Klasse Iterator
  ListElement* current;

};  // Ende der Klasse Iterator

// Es fehlen noch die neuen Methoden begin(), end() und
// erase()  der Klasse List: Siehe nächste Seite ✍
```

Iterator für Listen


```
// begin()
Iterator begin() const {
        return Iterator(start);
 }

// end()
Iterator end() const {
        return Iterator();
 }


void erase(Iterator& pos) {
        if (pos.current == start) {
                pop_front();
                pos.current = start;   // neuer Anfang
        }
        else if (pos.current == end) {
                pop_back();
                pos.current = end;   //  neues Ende
        }                                //  weiter auf der nächsten Seite
```


Iterator für Listen


```
    else  {  // zwischen zwei Elementen
            pos.current->next->prev = pos.current->prev;
            pos.current->prev->next = pos.current->next;
            ListElement *temp = pos.current;
            pos.current = pos.current->next;
            delete temp;
            --number;
    }   // Ende else
  } //  Ende  erase()
private:    //  hier folgt der private Rest von List wie
            //  bereits beschrieben

} // Ende der Klasse List
```

Container

- Sei X der Datentyp eines Beispiel-Containers, z.B.,
  X = vector<T> size_type = int, long, size_t, ... usw.
- Jeder Container stellt einen öffentlichen Satz von Methoden zur Verfügung:

Rückgabetyp Methode	Bedeutung
X( )	Standardkonstruktor für leeren C.
X(const X&)	Kopierkonstruktor
~X( )	Destruktor
iterator begin()	Anfang des Containers
const_iterator begin( )	Anfang des Containers
iterator end()	Position nach Ende des C.
const_iterator end( )	Position nach Ende des C.
size_type size( )	Aktuelle Größe des C.
size_type max_size( )	Maximal mögliche Größe des C.
bool empty( )	size == 0 ?
void swap(X&)	Vertauschen mit Argumentcont

Container

Rückgabetyp Methode	Bedeutung
X& operator=(const X&)	Zuweisungsoperator =
bool operator==(const X&)	Vergleichsoperator ==
bool operator!=(const X&)	Vergleichsoperator !=
bool operator<(const X&)	Vergleichsoperator <
bool operator>(const X&)	Vergleichsoperator >
bool operator<=(const X&)	Vergleichsoperator <=
bool operator>=(const X&)	Vergleichsoperator >=

Container: Bitset

- Der Header <bitset> definiert eine Template-Klasse und zugehörige Funktionen zur Darstellung und Bearbeitung von Bitfolgen fester Größe.
- Die Deklaration der Klasse ist

```
template<size_t N> class bitset;
```

- Eine ausführliche Beschreibung dieses Containers finden Sie im Breymann-Buch auf den Seiten 485-488.

Container: Deque

- Der Name Deque ist eine Abkürzung für „double ended queue“, also eine Warteschlange, die das Hinzufügen und Entnehmen sowohl am Anfang als auch am Ende erlaubt.
- Die Deklaration der Klasse ist

```
template<class T> class deque;
```

- Eine ausführliche Beschreibung dieses Containers finden Sie im Breymann-Buch ab Seite 489.
- Die interessantesten „zusätzlichen“ Funktionen sind:

Rückgabetyp Methode	Bedeutung
reverse_iterator rbegin() const_reverse_iterator rbegin()	Reverser Iterator, der beim letzten Element beginnt.
reverse_iterator rend() const_reverse_iterator rend()	Fiktive Position vor dem ersten Element
T& front() const T& front( ) const	Referenz auf erstes Element Referenz auf erstes Element
T& back( ) const T& back( ) const	Referenz auf das letzte Element Referenz ...

Container: Deque

Rückgabetyp Methode	Beschreibung
T& operator[ ](size_type n)	Referenz auf das n-te Element
T& at(size_type n)	Referenz auf das n-te Element
void push_front(const T& t)	Fügt t am Anfang ein
void push_back(const T& t)	Fügt t am Ende ein
void pop_front()	Löscht das erste Element
void pop_back()	Löscht das letzte Element
iterator insert(iterator p,const T& t)	Fügt eine Kopie von t vor p ein
iterator erase(iterator q)	Löscht das Element, auf das q zeigt, und zeigt dann auf das Element nach q
iterator erase(iterator s, iterator e)	Löscht den Bereich [s,e)
void clear( )	Löscht alle Elemente
void resize(size_type n, T t = T())	Dequegröße ändern

Breymann_Folien

Container: Liste

Rückgabetyp Methode	Beschreibung
list(size_type n, const T& t)	Erzeugt Liste mit n Kopien von t
void assign(size_type n, const T& t = T())	Liste löschen und anschließend n Kopien von t einfügen
void remove(const T& t)	Alle Elemente entfernen, die gleich t sind.
void reverse( )	Reihenfolge umkehren
void sort( )	Sortiert die Liste mit dem Vergleichsoperator < von T
template<class Compare> void sort(Compare cmp)	Sortiert mit dem Sortierkriterium des Compare-Objekts cmp S.342
void unique()	Entfernt gleiche aufeinander-Folgende Objekte bis auf das erste
void merge(list& L)	Mischt sortierte Listen <
void splice(iterator p, list& x)	Fügt x vor p ein

Breymann_Folien

Funktionsobjekte

- Funktoren sind Objekte, die sich wie Funktionen verhalten, aber alle Eigenschaften von Objekten haben.
- Diese Technik wird in den Klassen der Algorithmen und Klassen der C++-Standardbibliothek häufig eingesetzt.
- Bei Funktoren wird der Funktionsoperator () mit der Operatorfunktion operator()() überladen.
- Das Objekt kann dann wie eine Funktion aufgerufen werden.
- Als Beispiel definieren eine Klasse „less_Point2D“ für Vergleiche von Punkten.

Funktionsobjekte


```
// Klassendeklaration von less_Point2D
class less_Point2D {
    public:
      bool operator( ) (const Point2D&  a, const Point2D& b) {
                if (a.X() < b.X())  return true;
                else if (a.X() > b.X()) return false;
                else if (a.Y() < b.Y()) return true;
                      else return false;
       }
};

//  mögliches Hauptprogramm mit Anwendung von less ohne Header
int main() {
    list<Point2D> L;
    //  L wird irgend wie mit Punkten gefüllt.
    less_Point2D  cmp;
    L.sort(cmp);
}
```

Container: Map

- Der Header <map> definiert die Klasse map<Key, T>, die Paare von Schlüsseln und zugehörigen Daten speichert.
- Hierbei ist der Schlüssel eindeutig (es gibt keine zwei Datensätze mit dem gleichen Schlüssel).
- Die Deklaration der Klasse ist

```
template<class Key,           // Schlüssel
         class T,             // Daten
         class Compare = less<Key> >  // Standardvergleich
class map;
```

- Eine ausführliche Beschreibung dieses Containers finden Sie im Breymann-Buch ab Seite 494.
- Mit der map-Klasse lässt sich leicht ein Wörterbuch realisieren:

```
map<string, string> Woerterbuch;
```

Breymann_Folien

Weitere Container

- Queues <queue> und Stacks <stacks> wurden in der Vorlesung bereits ausführlich diskutiert.
- Diesen beiden Klassen stellen natürlich nur einen Teil der Funktionalität von Deque zur Verfügung.
- Eine Priority-Queue ist eine prioritätsgesteuerte Warteschlange. Priority-Queues werden wir später in der Vorlesung behandeln.
- Die Klasse set<Key, T> entspricht der Klasse map, nur dass Schlüssel und Daten zusammenfallen, d.h., es werden nur Schlüssel gespeichert.
- Die Klasse <vector> haben wir bereits ausführlich diskutiert und verwendet.

Breymann_Folien

Algorithmen

33

- Alle im Header <algorithm> vorhandenen Algorithmen sind unabhängig von der speziellen Implementierung der Container.
- Sie kennen nur Iteratoren, über die sie auf die Datenstrukturen in den Containern zugegriffen werden kann.
- Die Iteratoren müssen nur wenigen Kriterien genügen.
- Aus Zeitgründen verzichten wir hier auf eine Diskussion der vorhandenen Algorithmen (siehe Breymann-Skript Seite 430).

Breymann_Folien

ADTs und ihre Implementierungen

34

- Abstrakte Datentypen (ADT) kapseln Daten und Funktionen.
- Eine Klasse ist ein ADT, der in einer Programmiersprache formuliert ist.
- Eine ADT wird ausschließlich über die öffentliche Schnittstelle spezifiziert.
- Die STL erlaubt für manche ADTs verschiedene Implementierungen.

Breymann_Folien

ADTs und ihre Implementierungen


```
template< T, class Containertyp  =  deque<T>>
class stack {
    public:
            bool empty() const {
                    return c.empty();
            }
            // und weitere öffentliche Methoden
    private:
            Containertyp  c;
};
```

- Mögliche Implementierungen von „stack“:

```
stack<int> intStack1;
stack<int,  list<int>>  intStack2;
stack<int,  vector<int>>  intStack3;
```

Breymann_Folien