Informatikpraktikum CE2

Übung 1: Einfach verkettete Liste

Inhaltsverzeichnis

1 Anforderungen an die Liste

Da die Hauptaufgabe der Liste im speichern von wenigen Listenelementen bestehen wird, ist der Aufbau der Liste relativ simpel gehalten. Es werden keine Optimierungen für schnellen Listenzugriff implementiert. Die Datenstrukturen und Funktionen sind auf eine einfach verkettete Liste ausgelegt.

Die Listenklasse hat in diesem Fall den Namen `CList`. Die einzelen Listenelemente werden ebenfalls in Klassenform gespeichert, wobei die Klassendefinition hier in der Listenklasse selbst eingeschlossen ist, um das Prinzip der Kaspelung zu halten. Der Name der Listenelement - Hilfsklasse ist `CList::CElement`.

1.1 Beschreibung der notwendigen Funktionen

Es werden fünf Gruppen von Funktionen vorgesehen:

- Konstruktion / Destruktion
- Datenzugriffsfunktionen
- Hilfsfunktionen zur Statusabfrage
- private Hilfsfunktionen
- Debug - Funktionen

1.1.1 Konstruktion / Destruktion

Konstruktor:

```
template <class T>
CList<T>::CList();
```

Der Konstruktor initialisiert lediglich die Member-Variable auf das erste Listenelement auf NULL, um den Korrekten interen Zustand der Liste herzustellen.

Destruktor:

```
template <class T>
CList<T>::~Clist();
```

Der Destruktor der Liste zerstört sämtliche erzeugten Listenelment - Objekte. Die *direkt* darin gespeicherten Daten gehen dadurch verloren. Werden Pointer in der Liste gespeichert, werden die darin angegebenen Speicherbereiche *nicht* freigegeben.

1.1.2 Datenzugriffsfunktionen

Es werden drei Datenzugriffsfunktionen vorgesehen. Die beiden lesenden Funktionen nutzen Referenzparameter um die gesicherten Daten zurückzuliefern.

Element sichern

```
template <class T>
void CList<T>::AddData (T pData);
```

Diese Zugriffsfunktion fügt das Datum pData am Ende der Liste ein. Weitere Fehlerkontrollen oder -werte sind hier nicht notwendig.

Parameter:

- *T pData:* Zu speicherndes Datum

Element lesen

```
template <class T>
virtual int CList::<T>::GetData (int iPos, T & pResult);
```

GetData liefert im übergebenen Referenzparameter ein gespeicherte Datum zurück.

Parameter:

- *int iPos:* Index für das gesuchte Element. Die Indizierung beginnt bei 1.
- *T & pResult:* Rückgabeparameter für das gesuchte Datum.

Rückgabewerte:

- *0:* Das zurückgelieferte Datum is ungültig, da der Index nicht innerhalb des gültigen Bereiches lag.
- *1:* Funktionsaufruf erfolgreich.

Element lesen und entfernen

```
template <class T>
virtual int CList<T>::RemoveData (int iPos, T & pResult);
```

RemoveData liefert, wie GetData, ein gespeichertes Datum zurück. Einziger Unterschied zu GetData liegt darin, das das zurückgelieferte Listenelement gleichzeitig aus der Liste entfernt wird.

Parameter:

- *int iPos:* Index für das gesuchte Element. Die Indizierung beginnt bei 1.
- *T & pResult:* Rückgabeparameter für das gesuchte Datum.

Rückgabewerte:

- *0:* Das zurückgelieferte Datum is ungültig, da der Index nicht innerhalb des gültigen Bereiches lag.
- *1:* Funktionsaufruf erfolgreich.

1.1.3 Hilfsfunktionen zur Statusabfrage

Status: Liste leer

```
template <class T>
int CList<T>::IsEmpty();
```

Mit Hilfe von IsEmpty kann festgestellt werden, ob bereits Elemente in der Liste gespeichert sind.

Rückgabewerte:

- *0:* Die Liste ist leer.
- *1:* Die Liste enthält mindestens ein Element.

1.1.4 Private Hilfsfunktionen

Suchen eines bestimmten Listenelementes

```
private:
template <class T>
virtual CList<T>::CElement * CList<T>::GetElement (int iPos);
```

Die private Hilfsfunktion GetElement sucht auf der Basis des Indexes *iPos* ein Element der Liste.

Parameter:

- *int iPos:* Index für das gesuchte Element. Die Indizierung beginnt bei 1.

Rückgabewerte:

- *NULL:* Das gesuchte Element wurde nicht gefunden. Der Index ist ausserhalb des gültigen Bereiches.
- *!(NULL):* Der zurückgelieferte Wert ist ein gültiger `CElement` Zeiger, der auf das gesuchte Element zeigt.

Hilfsklasse CList::CElement

Diese private Hilfsklasse wird zum speichern der einzelnen Listenelemente verwendet. Die Funktionen in dieser Klasse beschränken sich auf Zugriffsfunktionen auf die einzelnen Datenelemente und werden im nächsten Abschnitt näher erläutert. Ich will hier an dieser Stelle lediglich auf den Konstruktor dieser Hilfsklasse eingehen:

Konstruktor der Hilfsklasse CElement

```
template <class T>
CList<T>::CElement::CElement (CElement * pNext, T pData);
```

Dieser Konstruktor initialisiert die beiden Member-Variablen der Hilfsklase `CElement`.

Parameter:

- *CElement * pNext:* Initialisiert den Zeiger auf das nächste Datenelement.
- *T pData:* Initialisiert das Datum innerhalb des Listenelements.

1.1.5 Debug - Funktionen

Augeblicklich existiert lediglich eine einzige Debug - Funktion zur Ausgabe des aktuellen Listeninhaltes. Diese Funktionen werden nur dann genutzt werden, wenn während des kompilierens das Define `_DEBUG` gesetzt ist.

Ausgeben des aktuellen Listeninhaltes

```
#ifdef _DEBUG
template <class T>
void CList<T>::Dump();
#endif //_DEBUG
```

Die Dump Funktion von jedem vorhandenen Listenelement die Speicheradresse aus und übergibt das dereferenzierte Datum dem IOStream cout. Aus diesem Grunde kann diese Funktion nur dann angewendet werden, wenn die Liste für Pointer genutzt wird und für diese Pointer der Stream - Operator “¡¡”definiert ist.

1.2 Beschreibung der notwendigen Datenstrukturen

Für die Listenklasse benötigt zwei Datenstrukturen, die im folgenden näher erläutert werden:

- Daten zur Verwaltung der Liste selbst
- Datenstruktur zum speichern der Daten

1.2.1 Verwaltung der Liste

Zeiger auf Listenanfang

```
template <class T>
class CList<T>
{...
private:
CElement * m_pFirst;
...}
```

Dieser Zeiger verweist auf das erste Listenelement. Ist die Liste leer, so hat dieser Zeiger den Wert NULL.

Zugriffsfunktionen: Für `m_pFirst` sind keine Zugriffsfunktionen definiert, da eine Änderung dieses Zeigers nur von der Liste korrekt durchgeführt werden kann.

1.2.2 Hilfsklasse zum speichern der Daten

Die Listenelemente besitzen zwei Member-Variablen, die benötigt werden um die Anforderungen der Aufgabe zu erfüllen:

Zeiger auf nächstes Listenelement

```
template <class T>
class CList<T>::CElement
{...
private:
CElement * m_pNext;
...}
```

Dieser Zeiger verweist auf das nächste Listenelement. Ist er NULL, so ist as Ende der Liste erreicht.

Zugriffsfunktionen:

- `CElement * GetNext()`: Lesen der Variable
- `void SetNext (CElement *)`: Setzen der Variable

Specherbereich für Datum

```
template <class T>
class CList<T>::CElement
{...
private:
T m_pData;
...}
```

In dieser Variable wird das zu sichernde Datum gespeichert.

Zugriffsfunktionen:

- `T GetData()`: Lesen der Variable
- `void SetData(T data)`: Setzen der Variable

1.3 Festlegung der Funktions - Algorithmen

Die Listenklasse benötigt in der aktuellen Implementierung lediglich einen Algorithmus zum Suchen der benötigten Daten. Dieser ist in der Funktion `CList::GetElement(int iPos)` realisiert. Es wird das einfachste mögliche System genutzt: Die Daten werden linear durchsucht, bis das gesuchte Element gefunden wird oder das Ende der Liste erreicht ist.

2 Beschreibung der Klassenstruktur

Der genaue Aufbau der Klasse `CList` dürfte aus dem Listing `CList.h` und der Funktionsbeschreibung aus Kapitel 1 ausreichend hervorgehen. Ich möchte trotzdem auf einige wesentliche Dinge der Klassenstruktur eingehen.

2.1 Schnittstellenfunktionen

2.1.1 Zugriffsfunktionen

Alle lesenden Zugriffsfunktionen fordern die Angabe eines Indexes, wodurch mehrere Objekte die gleiche Liste simultan nutzen können. Zugriffsfunktionen im Stile von MoveNext oder MovePrevious würden hier die Verwendbarkeit einschränken, da Zugriffsnummern eingeführt werden müßten.

2.1.2 Virtuelle, öffentliche Funktionen

Um sicherzustellen, daß die Liste als Basisklasse für zukünftige Erweiterungen verwendbar ist, wurden mehrere Member - Funktionen als `virtual` deklariert:

```
template <class T>
class CList<T>
{...
public:
virtual int RemoveData (int iPos, T & pResult);
virtual int AddData (T pData);
...}
```

Es ist nicht notwendig, die Zugriffsfunktion `CList<T>::GetData` virtuell zu implementieren, da sie, wie auch `CList<T>::AddData` und `CList<T>::RemoveData`, auf eine interne Hilfsfunktion zurückgreift um das angeforderte Listenelement zu deklarieren. Dadurch wird in `CList<T>::GetData` nur noch das Listenelement angefordert und danach das gespeicherte Datum zurückgeliefert.

2.1.3 Sonstige Funktionen

int IsEmpty()
Je nach Anwendungsgebiet erschien es mir sinnvoll, eine Möglichkeit zur Abfrage des `m_pFirst` - Zeigers hinzuzufügen. Dadurch erhält die Anwendung die Möglichkeit, auf eine leere Liste zu prüfen, beispielsweise um bei Ausgabe - Funktionen einer komplizierte Behandlung der ersten Ausgabe aus dem Weg zu gehen.

void Dump()
Zusätzlich existiert zur Fehlersuche eine Debugfunktion, die den aktuellen Listeninhalt auf `cout` ausgiebt. Bedingung hierfür ist, das die Liste mit `_DEBUG` kompiliert wird. Es entsteht folgende Bilschirmausgabe:

```
*** DEBUG - Ausgabe Objekt CList in 0x804a4a8
  Element 1 at 0x804a4c8 contains 1
  Element 2 at 0x804a4e8 contains 4
  Element 3 at 0x804a508 contains 9
  Element 4 at 0x804a528 contains 16
  Element 5 at 0x804a548 contains 25
*** DEBUG Ausgabe abgeschlossen
```

Wie aus hierraus hervorgeht, werden ebenfalls die einzelnen gesicherten Daten ausgegeben. Dies ist allerdings nur möglich, wenn Zeiger gespeichert werden, und für die durch die Zeiger definierten Objekte der Stream - Ausgabeoperator << definiert ist.

Wird eine dieser Bedingungen nicht erfüllt, sollte auf die Verwendung dieser Funktion verzichtet werden.

2.2 Private Hilfsfunktionen

Derzeit existiert eine einzige private Hilfsfunktion:

```
template <class T>
class CList<T>
{...
private: virtual CElement * GetElement (int iPos);
...}
```

Diese Funktion ist das Kernstück der lesenden Zugriffsfunktionen. Sie sucht jeweils das zu einem Index `iPos` passende CElement Objekt und liefert es an die aufrufende Funktion zurück. Sollte die Klasse erweitert werden ist hier ein einfacher Ansatzpunkt für alle Zugriffsfunktionen, die einen Index dereferenzieren müssen.

2.3 Private Hilfsklassen

Die Klasse `CList` enthält eine Hilfsklasse, die als private deklariert wurde, um eine weitere Klasse zum speichern der Daten zur Verfügung zu stellen. Diese Klasse wurde einerseits intern implementiert, um die Möglichkeit eines Namenskonflikte im globalen Namensraum zu verhindern, und andererseits ist der Aufbau der Listenelemente nur für die Listenklasse selbst von Bedeutung.

3 Anmerkungen des Autors

Im Anhang befindet sich der komplette Quellcode der Listenklasse. Vorschläge oder Verbesserungswünsche nehme ich gerne entgegen.

Anhang C stellt eine Beispielanwendung dar, die mit der Liste arbeitet. Sie muß mit dem Define _DEBUG kompiliert werden, da die Debug-Funktionen genutzt werden.

Die komplette Listenklasse ist ebenfalls im Internet unter `http://www.foo.fh-furtwangen.de/~nehmer/clist.html` erhältlich.

A CList.h

```
//////////////////////////////////////////////////////////////////////
//
//  LISTENKLASSE "CList"
//
//  Datei:      CList.h
//  Zweck:      Einfach vorwaerts verkettete Liste zum speichern von
//              Pointern auf durch ein Template festgelegte Objekte.
//  Autor:      Torben Nehmer <Torben.Nehmer@gmx.net>
//  Version:    1.30
//  Datum:      15.10.1998
//
//  Implementierung in:
//              CList.cpp
//
//////////////////////////////////////////////////////////////////////

#if !defined(INCLUDE_CLIST_HEADER)
#define INCLUDE_CLIST_HEADER

template <class T>
class CList
{
private:
//////////////////////////////////////////////////////////////////////
//  Hilfsklasse fuer Datenelemente
//
    class CElement
    {
    private:
      T m_pData;
       // Pointer auf Daten
      CElement * m_pNext;
       // Pointer auf naechstes Element
    public:
      CElement ( CElement * pNext, T pData);
       // Standardkonstruktor

    //////////////////////////////////////////////////////////////////
    //  Datenzugriffsfunktionen

      CElement * GetNext () const;
       // Liefert Pointer auf naechstes Element
      void SetNext ( CElement * pNext );
       // Setzt Pointer auf naechstes Element
      T GetData() const;
       // Liefert gespeicherte Daten
      void SetData (T data);
       // Setzt Daten
    };

       // Hilfsklasse zum speichern der Objektpointer. Definition als
       // Klasse, da Implementierung durch Konstruktor vereinfacht wird.

//////////////////////////////////////////////////////////////////////
//  Private Hilfsfunktionen
//
    virtual CElement * GetElement (int iPos) const;
       // Liefert Pointer auf n-tes Element oder NULL Pointer wenn
```

```
       // Element nicht vorhanden.

//////////////////////////////////////////////////////////////////
//  Datenelemente
//
    CElement * m_pFirst;
        // Zeiger auf erstes Datenelement, wird vom Konstruktor
        // auf NULL initialisiert.


public:

//////////////////////////////////////////////////////////////////
//  Kunstruktion / Destuktion
//
    CList();
        // Standardkonstruktor
    virtual ~CList();
        // Destruktor

//////////////////////////////////////////////////////////////////
//  Zugriffsoperatoren
//
    int GetData (int iPos, T & pResult) const;
        // Liefert n-ten Datenpointer
    virtual int RemoveData (int iPos, T & pResult);
        // Liefert n-ten Datenpointer und loescht Element

        // GetData UND RemoveData geben 0 zurueck, wenn
        // das gesuchte Element nicht existiert.

    virtual void AddData (T pData);
        // Fuegt ein neues Element am Ende der Liste ein.


//////////////////////////////////////////////////////////////////
// Statusabfragen
//
    int IsEmpty () const;
        // TRUE, wenn Liste leer.

//////////////////////////////////////////////////////////////////
//  Debug Funktionen
//  Gehen von 'T *' und definierten 'cout << ...' aus!
//
#ifdef _DEBUG
    void Dump () const;
        // Gibt Pointeradressen und Zeigerinhalte (als T *) auf
        // cout aus.

#endif // _DEBUG


};

//////////////////////////////////////////////////////////////////
//  Include der Implementierung, sonst möglicher Fehler wg. Template
#include "CList.cpp"

#endif // INCLUDE_CLIST_HEADER
```

B CList.cpp

```
////////////////////////////////////////////////////////////////////////
//
//  LISTENKLASSE "CList" - Implementierung
//
//  Datei:      CList.cpp
//  Autor:      Torben Nehmer <Torben.Nehmer@gmx.net>
//
////////////////////////////////////////////////////////////////////////

#include <stdio.h>

////////////////////////////////////////////////////////////////////////
// Konstruktion / Destruktion
//

template <class T>
CList<T>::CList()
{
  // Standardkonstruktor initialisiert m_pFirst auf leere Liste
    m_pFirst = NULL;
}

template <class T>
CList<T>::~CList()
{
  // Destruktor entfernd noch vorhandende Speicherobjekte. Es werden
  // jedoch NICHT die gespeicherten Objekte geloescht.
    if (!IsEmpty())
    {
        CElement * pTemp = m_pFirst;
        CElement * pDel;
        while (pTemp)
        {
            pDel = pTemp;
            pTemp = pTemp->GetNext();
            delete pDel;
        }
    }
}

////////////////////////////////////////////////////////////////////////
//  Zugriffs - Funktionen
//

template <class T>
void CList<T>::AddData(T pData)
{
  // Fuegt einen neuen Listeneintrag am Ende der Liste ein.
    if (IsEmpty())
        m_pFirst = new CElement(NULL, pData);
    else
    {
        CElement * pLast = m_pFirst;
        while (pLast->GetNext())
            pLast = pLast->GetNext();
        pLast->SetNext(new CElement(NULL, pData));
    }
}
```

```
template <class T>
int CList<T>::GetData (int iPos, T & pResult) const
{
  // Liefert den Datensatz an der Stelle iPos zurueck, wobei die
  // Indizierung mit 1 beginnt.
  // Der gespeicherte Zeiger wird in pResult zurueckgeliefert.
  // Die Funktion selbst liefert TRUE (nonzero), falls der Aufruf
  // erfolgreich war.
    CElement * pElement = GetElement (iPos);
    if (pElement)
      {
pResult = pElement->GetData();
return 1;
      }
    else
      return 0;
}

template <class T>
int CList<T>::RemoveData (int iPos, T & pResult)
{
  // Liefert den Datensatz an der Stelle iPos zurueck, wobei die
  // Indizierung mit 1 beginnt. Der betroffene Eintrag wird aus
  // der Liste entfernt.
  // Der gespeicherte Zeiger wird in pResult zurueckgeliefert.
  // Die Funktion selbst liefert TRUE (nonzero), falls der Aufruf
  // erfolgreich war.
    CElement * pDel = GetElement (iPos);
    if (!pDel)
        return 0;

    if (iPos == 1)
        m_pFirst = pDel->GetNext();
    else
    {
        CElement * pPrev = GetElement (iPos - 1);
        pPrev->SetNext(pDel->GetNext());
    }
    pResult = pDel->GetData();
    delete pDel;
    return 1;
}

//////////////////////////////////////////////////////////////////////
//  Statusabfrage - Funktionen
//

template <class T>
inline int CList<T>::IsEmpty () const
{
  // Liefert TRUE (nonzero), wenn KEINE Elemente in der Liste
  // vorhanden sind.
    return (!m_pFirst) ? 1 : 0;
}

//////////////////////////////////////////////////////////////////////
//  Private Hilfsfunktionen
//
template <class T>
```

```
CList<T>::CElement * CList<T>::GetElement(int iPos) const
{
  // Private Hilfsfunktion, die versucht, den Listeneintrag an der
  // Stelle iPos zurueckzuliefern. Existiert ein solches Element,
  // wird der CElement Pointer darauf zurueckgegeben, ansonsten der
  // NULL Pointer.
    if (!m_pFirst || iPos < 1)
        return NULL;

    CElement * pTemp = m_pFirst;

    for (int i = 1; i < iPos; i++)
    {
        if (!pTemp->GetNext())
            return NULL;
        pTemp = pTemp->GetNext();
    }

    return pTemp;
}

//////////////////////////////////////////////////////////////////////
//  Debug Funktionen
//


#ifdef _DEBUG

#include <iostream.h>

template <class T>
void CList<T>::Dump() const
{
  // DEBUG Funktion, mit deren Hilfe sich der Inhalt des Objektes
  // MIT Speicheradressen nach COUT ausgegeben werden kann. Die
  // gespeicherten Objekte muessen deshalb ueber einen definierten
  // Stream Output Operator verfuegen.
    CElement * pTemp;
    cout << "*** DEBUG - Ausgabe Objekt CList in " << this << endl;

    if (!IsEmpty())
    {
        pTemp = m_pFirst;
        int i = 1;
        while (pTemp)
        {
            cout << "  Element " << i++ << " at " << pTemp << " contains "
                << *((T) pTemp->GetData()) << endl;
            pTemp = pTemp->GetNext();
        }
        cout << "*** DEBUG Ausgabe abgeschlossen\n";
    }
    else
        cout << "Keine Elemente vorhanden!\n*** DEBUG "
            << "Ausgabe abgeschlossen!\n";
}

#endif _DEBUG

//////////////////////////////////////////////////////////////////////
```

```
//  Implementierung Hilfsklasse CList::CElement
//  und deren Zugriffsfunktionen
//

template <class T>
CList<T>::CElement::CElement ( CElement * pNext, T pData)
  : m_pNext(pNext), m_pData(pData)
{
}

template <class T>
CList<T>::CElement * CList<T>::CElement::GetNext () const
{
  return m_pNext;
}

template <class T>
void CList<T>::CElement::SetNext ( CElement * pNext )
{
  m_pNext = pNext;
}

template <class T>
T CList<T>::CElement::GetData() const
{
  return m_pData;
}

template <class T>
void CList<T>::CElement::SetData (T data)
{
  m_pData = data;
}
```

C Beispiel: main.cpp

```
//////////////////////////////////////////////////////////////////////
//
//  LISTENKLASSE "CList" - Beispielanwendung
//
//  Datei:      main.cpp
//  Autor:      Torben Nehmer <Torben.Nehmer@gmx.net>
//
//////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <iostream.h>
#include "CList.h"

int main()
{
  // Liste erstellen
    CList<int *> * pList = new CList<int *>();

    cout << "\nCreating list...\n\n";

    int * pInt;
    int iCount;

  // Liste füllen
    for (iCount = 1; iCount <= 5; iCount++)
    {
        pInt = new int;
        *pInt = iCount * iCount;
        pList->AddData (pInt);
    }

    cout << "\nList filled...\n\n";

    pList->Dump();

    cout << "\nRemoving from list...\n\n";

  // Liste leeren
    iCount=1;
    while (pList->RemoveData(1,pInt))
    {
       cout << "Executing loop " << iCount++ << " - Result of GetData: " <<
         *(pInt) << endl;
       delete pInt;
    }

    cout << "\nNew list contents...\nPress almost any key to continue\n\n";

    pList->Dump();



    return 0;
}
```