

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 9: Elementare Datenstrukturen

M.Sc. Roman Kalkreuth

Lehrstuhl für Algorithm Engineering (LS11)

Fakultät für Informatik

Inhalt

- Definition: Abstrakter Datentyp (ADT)
- ADT Stapel
- ADT Schlange
- ADT Liste
- ADT Binärer Suchbaum
- ADT Graph
- Exkurse:
 - Einfache Dateibehandlung
 - C++-Strings

Definition:

Abstrakter Datentyp (ADT) ist ein Tripel (T, F, A) , wobei

- T eine nicht-leere Menge von **Datenobjekten**,
- F eine Menge von **Operationen**,
- A eine nicht-leere Menge von **Axiomen**,
die die Bedeutung der Operationen erklären.

Abstrakt?

- Datenobjekte brauchen keine konkrete Darstellung (Verallgemeinerung).
- Die **Wirkung** der Operationen wird beschrieben,
nicht deren algorithmische Ausprägung.

→ „**WAS, nicht WIE!**“

Beispiel: ADT bool**F: Operationen**

true : → bool
false : → bool
not : bool → bool
and : bool x bool → bool
or : bool x bool → bool

Festlegung, **welche Methoden** es gibt

A: Axiome

not(false) = true
not(true) = false
and(false, false) = false
and(false, true) = false
and(true, false) = false
and(true, true) = true
or(x, y) = not(and(not(x), not(y)))

Festlegung, **was** die Methoden bewirken

Eigenschaften

- Wenn man einen ADT kennt, dann kann man ihn **überall verwenden**.
- Implementierung der Funktionen für Benutzer nicht von Bedeutung.
- **Trennung von Spezifikation und Implementierung**
- Ermöglicht späteren Austausch der Implementierung, ohne dass sich der Ablauf anderer Programme, die ihn benutzen, ändert!

Nur Operationen geben Zugriff auf Daten.

→ Stichwort: **Information Hiding**

Lineare Datenstrukturen: Keller bzw. **Stapel** (*engl. stack*)

create : → Stapel
 push : Stapel x T → Stapel
 pop : Stapel → Stapel
 top : Stapel → T
 empty : Stapel → bool

empty(create) = true
 empty(push(k, x)) = false
 pop(push(k, x)) = k
 top(push(k, x)) = x

Aufräumen:
 Kiste in den Keller,
oben auf Haufen.
 Etwas aus Keller holen:
 Zuerst **oberste** Kiste,
 weil oben auf Haufen.



LIFO:
 Last in, first out.

Klassendefinition: (Version 1)

```
template<typename T>
class Stapel {
public:
    Stapel();           // Konstruktor
    void push(T &x);    // Element auf den Stapel legen
    void pop();        // oberstes Element entfernen
    T top();           // oberstes Element ansehen
    bool empty();     // Stapel leer?
private:
    static unsigned int const maxSize = 100;
    int sz;           // Stapelzeiger
    T data[maxSize]; // Speichervorrat für Nutzdaten
};
```

Alternative: anonymer enum („the enum trick“)

```
enum { maxSize = 100 };
```

Implementierung: (Version 1)

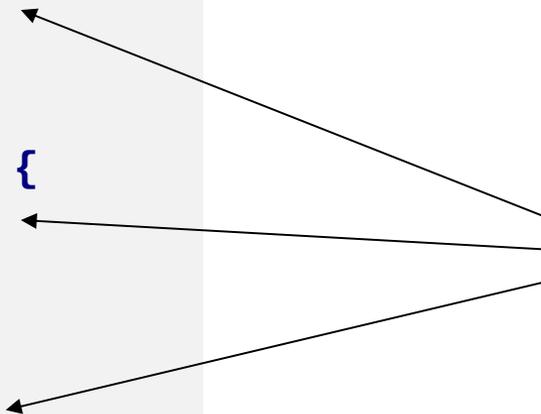
```
template<typename T>
Stapel<T>::Stapel() {
    sz = -1;
}
```

```
template<typename T>
void Stapel<T>::push(T &x) {
    data[++sz] = x;
}
template<typename T>
void Stapel<T>::pop() {
    sz--;
}
template<typename T>
T Stapel<T>::top() {
    return data[sz];
}
template<typename T>
bool Stapel<T>::empty() {
    return (sz == -1);
}
```

Idee:

unzulässiger Arrayindex -1
kennzeichnet leeren Stapel

Problem:
Arraygrenzen!



Wann können Probleme auftreten?

Bei **pop**, falls Stapel leer ist:

→ Stapelzeiger wird -2, anschließendes **push** versucht auf **data[-1]** zu schreiben

Bei **top**, falls Stapel leer ist:

→ es wird undefinierter Wert von **data[-1]** zurückgegeben

Bei **push**, falls Stapel voll ist:

→ es wird versucht auf **data[maxSize]** zu schreiben (erlaubt: 0 bis maxSize – 1)

⇒ diese Fälle müssen abgefangen werden, **Fehlermeldung**

```
void error(char const *info) {  
    cerr << info << endl;  
    exit(1);  
}
```

gibt Fehlermeldung **info** aus und bricht das Programm durch **exit(1)** sofort ab und liefert den Wert des Arguments (hier: 1) an das Betriebssystem zurück

Klassendefinition: (Version 2; Ergänzungen in **rot**)

```
template<typename T>
class Stapel {
public:
    Stapel();           // Konstruktor
    void push(T &x);   // Element auf den Stapel legen
    void pop();        // oberstes Element entfernen
    T top();           // oberstes Element ansehen
    bool empty();      // Stapel leer?
    bool full();       // Stapel voll?
private:
    static unsigned int const maxSize = 100;
    int sz;            // Stapelzeiger
    T data[maxSize];  // Speichervorrat für Nutzdaten
    void error(char const *info); // Fehlermeldung + Abbruch
};
```

Implementierung: (Version 2, Änderungen und Zusätze in **rot**)

```
template<typename T>
Stapel<T>::Stapel() {
    sz = -1;
}
template<typename T>
void Stapel<T>::push(T &x) {
    if (full()) error("voll");
    data[++sz] = x;
}
template<typename T>
void Stapel<T>::pop() {
    if (empty()) error("leer");
    sz--;
}
```

```
template<typename T>
T Stapel<T>::top() {
    if (empty()) error("leer");
    return data[sz];
}
template<typename T>
bool Stapel<T>::empty() {
    return (sz == -1);
}
template<typename T>
bool Stapel<T>::full() {
    return (sz == maxSize - 1);
}
```

```
template<typename T>
void Stapel<T>::error(char const * info) {
    std::cerr << info << std::endl;
    exit(1);
}
```

← **private Methode:**
kann nur innerhalb der
Klasse aufgerufen werden

Erster Test ...

```
#include <iostream>
#include "Stapel.h"
using namespace std;

int main() {
    Stapel<int> s;
    for (int i = 0; i < 100; i++) s.push(i);
    cout << s.top() << endl;
    for (int i = 0; i < 90; i++) s.pop();
    while (!s.empty()) {
        cout << s.top() << endl;
        s.pop();
    }
    return 0;
}
```

Ausgabe: 99
9
8
7
6
5
4
3
2
1
0

Lineare Datenstrukturen: **Schlange** (engl. *queue*)

create : \rightarrow Schlange
enq : Schlange \times T \rightarrow Schlange
deq : Schlange \rightarrow Schlange
front : Schlange \rightarrow T
empty : Schlange \rightarrow bool

empty(create) = true
empty(enq(s, x)) = false
deq(enq(s, x)) = empty(s) ? s : enq(deq(s), x)
front(enq(s, x)) = empty(s) ? x : front(s)

FIFO:

First in, first out.

Schlange an der Supermarktkasse:

Wenn Einkauf fertig, dann **hinten** anstellen. Der nächste Kunde an der Kasse steht ganz **vorne** in der Schlange.

Eingehende Aufträge werden „geparkt“, und dann nach und nach in der Reihenfolge des Eingangs abgearbeitet.

Klassendefinition: (Version 1; schon mit Fehlerbehandlung)

```
template<typename T>
class Schlange {
public:
    Schlange();           // Konstruktor
    void enq(T &x);      // Element anhängen
    void deq();          // erstes Element entfernen
    T front();           // erstes Element ansehen
    bool empty();        // Schlange leer?
    bool full();         // Schlange voll?
private:
    static unsigned int const maxSize = 100;
    int ez;              // Endezeiger
    T data[maxSize];     // Array für Nutzdaten
    void error(char const *info); // Fehlermeldung
};
```

Implementierung: (Version 1; Fehler bei Arraygrenzen werden abgefangen)

```
template<typename T>
Schlange<T>::Schlange() : ez(-1) {
}
template<typename T>
void Schlange<T>::enq(T &x) {
    if (full()) error("voll");
    data[++ez] = x;
}
template<typename T>
void Schlange<T>::deq() {
    if (empty()) error("leer");
    for (int i = 0; i < ez; i++)
        data[i] = data[i+1];
    ez--;
}
```

```
template<typename T>
T Schlange<T>::front() {
    if (empty()) error("leer");
    return data[0];
}
template<typename T>
bool Schlange<T>::empty() {
    return (ez == -1);
}
template<typename T>
bool Schlange<T>::full() {
    return (ez == maxSize - 1);
}
```

```
template<typename T>
void Schlange<T>::error(char const *info) {
    std::cerr << info << std::endl;
    exit(1);
}
```

← **private Methode:**
kann nur innerhalb der
Klasse aufgerufen werden

Erster Test ...

```
#include <iostream>
#include "Schlange.h"
using namespace std;

int main() {
    Schlange<int> s;

    for (int i = 0; i < 100; i++) s.enq(i);
    cout << s.front() << endl;
    for (int i = 0; i < 90; i++) s.deq();
    while (!s.empty()) {
        cout << s.front() << endl;
        s.deq();
    }
    return 0;
}
```

Ausgabe: 0
90
91
92
93
94
95
96
97
98
99

Benutzer des (abstrakten) Datentyps **Schlange** wird feststellen, dass

1. fast alle Operationen schnell sind, aber
2. die Operation **deq** vergleichsweise langsam ist.

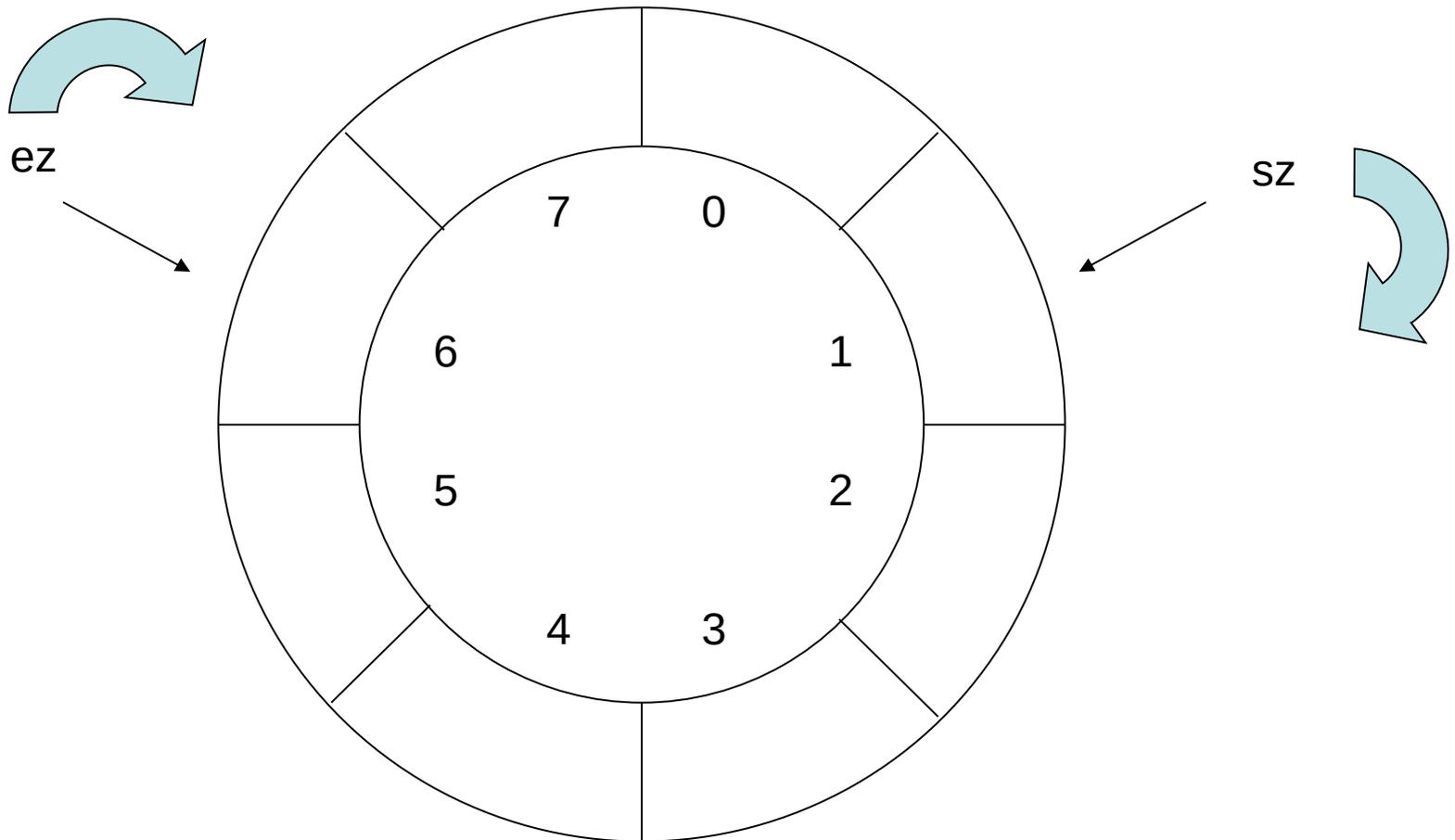
Laufzeit / Effizienz der Operation **deq**

```
template<typename T>
void Schlange<T>::deq() {
    if (empty()) error("leer");
    for (int i = 0; i < ez; i++)
        data[i] = data[i+1];
    ez--;
}
```

ez = Anzahl Elemente in Schlange
Insgesamt **ez** Datenverschiebungen

Worst case: (**maxSize** – 1) mal

Idee: Array zum Kreis machen; zusätzlich Anfang/Start markieren (sz)



Implementierung: (Version 2; mit Ringspeicher)

```
template<typename T>
class Schlange {
public:
    Schlange();
    void enq(T &x);
    void deq();
    T front();
    bool empty();
    bool full();
private:
    static unsigned int const maxSize = 100;
    int ez;           // Endezeiger
    int sz;           // Startzeiger
    T data[maxSize];
    void error(char const *info);
};
```

Implementierung: (Version 2; mit Ringspeicher)

```
template<typename T> Schlange<T>::Schlange() {
    sz = 0;
    ez = -1;
}
template<typename T> T Schlange<T>::front() {
    if (empty()) error("leer");
    return data[sz];
}
template<typename T> bool Schlange<T>::empty() {
    return (ez == -1);
}
template<typename T> bool Schlange<T>::full() {
    if (empty()) return false;
    return ((ez + 1) % maxSize) == sz;
}
```

Implementierung: (Version 2; mit Ringspeicher)

```
template<typename T>
void Schlange<T>::enq(T &x) {
    if (full()) error("full");
    ez = (ez + 1) % maxSize;
    data[ez] = x;
}
```

Laufzeit:

unabhängig von Größe
der Schlange

```
template<typename T>
void Schlange<T>::deq() {
    if (empty()) error("leer");
    if (sz == ez) { sz = 0; ez = -1; }
    else sz = (sz + 1) % maxSize;
}
```

Laufzeit:

unabhängig von Größe
der Schlange

Unbefriedigend bei der Implementierung:

Maximale festgelegte Größe des Stapels bzw. der Schlange

→ Liegt an der unterliegenden Datenstruktur Array:

Array ist **statisch**, d.h.

Größe wird **zur Übersetzungszeit festgelegt**

und ist während der Laufzeit des Programms **nicht veränderbar**.

Schön wären **dynamische** Datenstrukturen, d.h.

Größe wird **zur Übersetzungszeit nicht festgelegt**

und ist während der Laufzeit des Programms **veränderbar**.

⇒ **Dynamischer Speicher!** (Stichwort: **new / delete**)

Lineare Datenstrukturen: **Schlange** (*engl. queue*)

create : \rightarrow Schlange
enq : Schlange \times T \rightarrow Schlange
deq : Schlange \rightarrow Schlange
front : Schlange \rightarrow T
empty : Schlange \rightarrow bool

create : erzeugt leere Schlange

enq : hängt Element ans Ende der Schlange

deq : entfernt Kopf der Schlange

front : gibt im Kopf der Schlange gespeichertes Element zurück

empty : prüft, ob Schlange leer ist

→ Implementierung mit statischem Speicher ersetzen durch dynamischen Speicher

Bauplan:

Datentyp *Variable = **new** Datentyp; (Erzeugen)
delete Variable; (Löschen)

Bauplan für Arrays:

Datentyp *Variable = **new** Datentyp[Anzahl]; (Erzeugen)
delete[] Variable; (Löschen)

Achtung:

Dynamisch erzeugte Objekte müssen auch wieder gelöscht werden,
keine automatische Speicherbereinigung!

Vorüberlegungen für ADT Schlange mit dynamischem Speicher:

Wir können bei der Realisierung der Schlange statt statischem (Array) nun **dynamischen Speicher** verwenden ...

Ansatz: `new int[oldsize+1]` ... bringt uns das weiter?

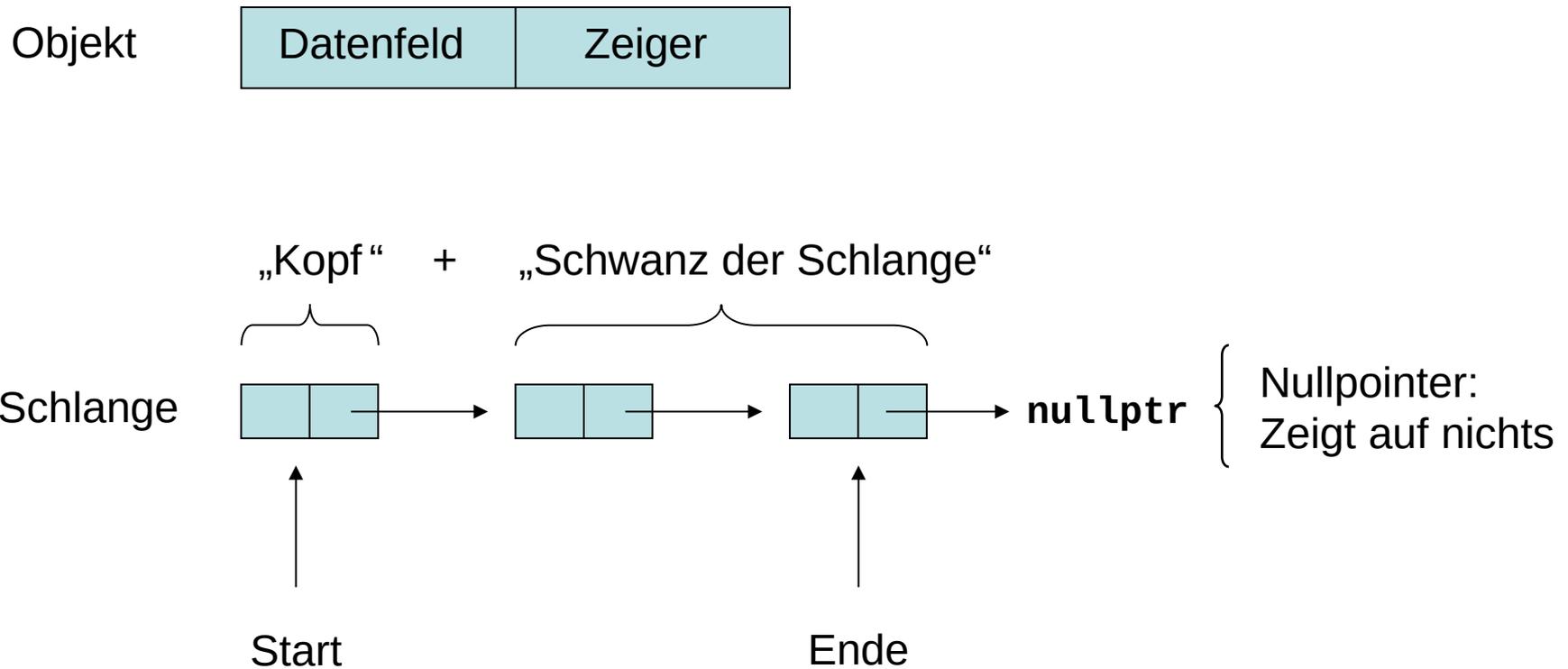
→ Größe kann zwar zur Laufzeit angegeben werden, ist aber dann fixiert!

Falls maximale Größe erreicht, könnte man

1. größeres Array anlegen
2. Arraywerte ins größere Array **kopieren** und
3. kleineres Array löschen.

ineffizient!

Vorüberlegungen für ADT Schlange mit dynamischem Speicher:



Klassendefinition: (Version 3; mit dynamischem Speicher)

```
template<typename T>
class Schlange {
public:
    Schlange();                // Konstruktor
    void enq(T &x);
    void deq();
    T front();
    bool empty();
    void clear();             // löscht alle Einträge
    ~Schlange();             // Destruktor
private:
    struct Objekt {           // interner Datentyp
        Objekt *tail;        // Zeiger auf Schlangenschwanz
        T data;              // Datenfeld
    } *sz, *ez;             // Zeiger auf Start + Ende
    void error(char const *info); // für Fehlermeldungen
};
```

Implementierung: (Version 3)

```
template<typename T>
Schlange<T>::Schlange() {
    ez = sz = nullptr;
}
template<typename T>
T Schlange<T>::front() {
    if (empty()) error("leer");
    return sz->data;
}
template<typename T>
bool Schlange<T>::empty() {
    return (ez == nullptr);
}
template<typename T>
void Schlange<T>::clear() {
    while (!empty()) deq();
}
```

nullptr ist der Nullzeiger!

```
template<typename T>
Schlange<T>::~~Schlange() {
    clear();
}
template<typename T>
void Schlange<T>::error(
    char const *info){
    cerr << info << endl;
    exit(1);
}
```

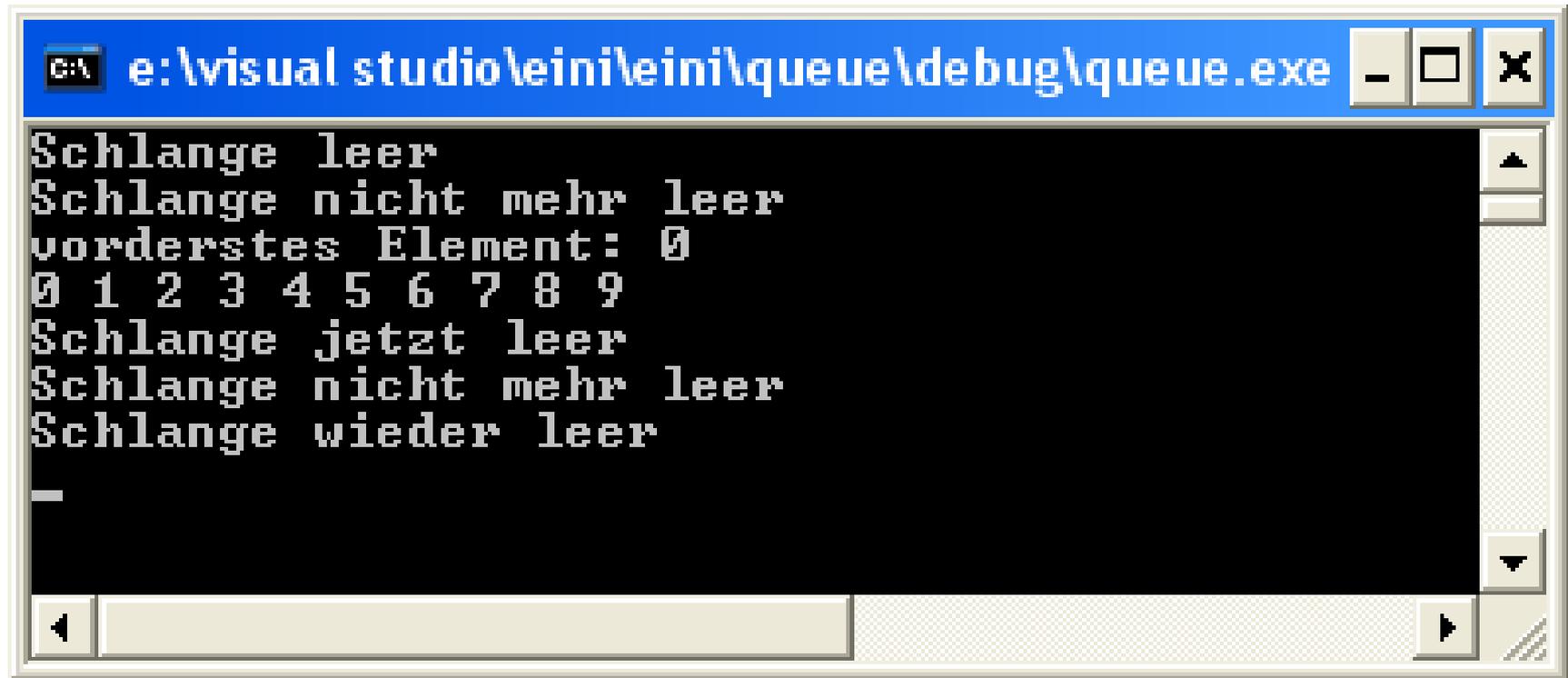
Implementierung: (Version 3)

```
template<typename T>
void Schlange<T>::enq(T &x) {
    Objekt *obj = new Objekt; // neues Objekt anlegen
    obj->data = x;             // Nutzdaten speichern
    obj->tail = nullptr;
    if (empty()) sz = obj;    // falls leer nach vorne,
    else ez->tail = obj;      // sonst hinten anhängen
    ez = obj;                 // Endezeiger aktualisieren
}
template<typename T>
void Schlange<T>::deq() {
    if (empty()) error("leer");
    Objekt *obj = sz;        // Zeiger auf Kopf retten
    sz = sz->tail;           // Start auf 2. Element
    if (sz == nullptr) ez = nullptr; // Schlange leer!
    delete obj;              // ehemaliges 1. Element
                              // löschen
}
```

Testprogramm!

```
int main() {
    Schlange<int> s;
    if (s.empty()) cout << "Schlange leer" << endl;
    for (int i = 0; i < 10; i++) s.enq(i);
    if (!s.empty()) cout << "Schlange nicht mehr leer" << endl;
    cout << "vorderstes Element: " << s.front() << endl;
    while (!s.empty()) {
        cout << s.front() << " ";
        s.deq();
    }
    cout << endl;
    if (s.empty()) cout << "Schlange jetzt leer" << endl;

    for (int i = 0; i < 100; i++) s.enq(i);
    if (!s.empty()) cout << "Schlange nicht mehr leer" << endl;
    s.clear();
    if (s.empty()) cout << "Schlange wieder leer" << endl;
    return 0;
}
```



```
e:\visual studio\ini\ini\queue\debug\queue.exe  
Schlange leer  
Schlange nicht mehr leer  
vorderstes Element: 0  
0 1 2 3 4 5 6 7 8 9  
Schlange jetzt leer  
Schlange nicht mehr leer  
Schlange wieder leer  
-
```

Kopieren von Klassenobjekten

```
template<typename T>
class Schlange {
    T data[100];
    int sz, ez;
};
```

→

```
Schlange<int> s1;
for (int i=0;i<10;i++)
    s1.enq(i);
Schlange<int> s2 = s1;
```

statischer Speicher:
 byteweises
 Speicherabbild
 ⇒ **OK**

```
template<typename T>
class Schlange {
    struct Objekt {
        Objekt *tail;
        T data;
    } *sz, *ez;
};
```

→

```
Schlange<int> s1;
for (int i=0;i<10;i++)
    s1.enq(i);
Schlange<int> s2 = s1;
```

dynam. Speicher:
 byteweises
 Speicherabbild
 ⇒ **Problem!**

Es werden nur die **Inhalte der Zeiger** kopiert!

Bei Verwendung von dynamischem Speicher muss auch dieser kopiert werden.

⇒ In C++ kann das durch den **Kopierkonstruktor** realisiert werden.

Kopierkonstruktor (*copy constructor*)

Wird für eine Klasse **kein Kopierkonstruktor** implementiert, dann erzeugt ihn der Compiler **automatisch**.

Achtung:

Es wird dann ein **byteweises Speicherabbild** des Objektes geliefert.

⇒ „flache Kopie“ (engl. *shallow copy*)

Problem:

- Konstruktor fordert dynamischen Speicher an → nur Kopie des Zeigers
- Konstruktor öffnet exklusive Datei (o.a. Ressource) → nicht teilbar! Crash!

⇒ dann „tiefe Kopie“ (engl. *deep copy*) nötig

⇒ man **muss** Kopierkonstruktor (und Destruktor) implementieren!

```

#include <iostream>
#include "Stapel.h"      // statischer Speicher
#include "Schlange.h"   // dynamischer Speicher
using namespace std;
int main() {
    Stapel<int> stack1;
    Schlange<int> queue1;
    for (int i = 0; i < 10; i++) {
        stack1.push(i);
        queue1.enq(i);
    }
    Stapel<int> stack2 = stack1;
    Schlange<int> queue2 = queue1;
    while (!stack1.empty()) stack1.pop();
    while (!queue1.empty()) queue1.deq();
    while (!stack2.empty()) {
        cout << stack2.top() << endl;
        stack2.pop();
    }
    while (!queue2.empty()) {
        cout << queue2.front() << endl;
        queue2.deq();
    }
    return 0;
}

```

Programmiertes Unheil:

Stapel1 / Schlange1 mit Daten belegen.

Stapel1 / Schlange1 kopieren

Stapel1 / Schlange1 löschen

kopierten Stapel ausgeben

kopierte Schlange ausgeben ...



crash!

Kopierkonstruktor (*copy constructor*)

```
template<typename T>
class Schlange {
public:
    Schlange(); // Konstruktor
    Schlange(const Schlange<T>& s);
    ~Schlange(); // Destruktor
};
```

Kann wie eine Zuweisung
interpretiert werden

← **Kopierkonstruktor**

```
template<typename T>
Schlange<T>::Schlange(const Schlange<T>& s){
    ez = nullptr;
    Objekt *ptr = s.sz;
    while (ptr != nullptr) {
        enq(ptr->data);
        ptr = ptr->tail;
    }
}
```

Entstehendes
Objekt wird mit einem
bestehenden Objekt
initialisiert

Kopierkonstruktor (*copy constructor*)

Bauplan:

ObjektTyp (**const** ObjektTyp & bezeichner);

→ Kopierkonstruktor liefert / soll liefern byteweises Speicherabbild des **Objektes**

Wird **automatisch** aufgerufen, wenn:

1. ein neues Objekt erzeugt und mit einem bestehenden initialisiert wird;
2. ein Objekt per Wertübergabe als Parameter an eine Funktion übergeben wird;
3. ein Objekt mit **return** als Wert zurückgegeben wird.

```
Punkt a(1.2, 3.4); // neu
Punkt b(a);       // Kopie: direkter Aufruf des Kopierkonstruktors
Punkt c = b;     // Kopie: bewirkt Aufruf des Kopierkonstruktors
b = a;          // Zuweisung, keine Kopie → gleiche Problematik!
```

Wenn für eine Klasse der **Zuweisungsoperator** nicht überschrieben wird, dann macht das der Compiler **automatisch**.

Vorsicht:

Speicher des Objektes wird **byteweise** überschrieben.

Problem:

z.B. wenn Objekt dynamischen Speicher verwendet

⇒ gleiche Problematik wie beim Kopierkonstruktor

Merke:

Wenn die Implementierung eines **Kopierkonstruktors** nötig ist, dann höchstwahrscheinlich auch **Destruktor** und überschriebene **Zuweisung**. (Das ist die sogenannte „*Rule of Three*“.)

Überladen von Operatoren

- Operator ist eine Verknüpfungsvorschrift
- Kann man auffassen als Name einer Funktion:

Bsp: Addition $a + b$ interpretieren als $+(a, b)$

- in C++ als: $c = \text{operator+}(a, b)$


Zweck:

eine Klasse mit Funktionalität ausstatten,
die vergleichbar mit elementarem Datentyp ist

insbesondere bei
Zuweisung und **Gleichheit**

Vorteil:

Quellcode wird übersichtlicher

Überladen von Operatoren

Welche?

+	^	==	+=	^=	!=	<<	()
-	&	>	-=	&=	&&	<<=	new
*		>=	*=	=		>>	delete
/	~	<	/=	++	->	>>=	=
%	!	<=	%=	--	->*	[]	



Wie?

Objektyp& operator (const ObjektTyp& bezeichner)

Objektyp operator (const ObjektTyp& bezeichner)

Überladen von Operatoren: Zuweisung

```
template<typename T>
Schlange<T>& Schlange<T>::operator= (const Schlange<T>& s) {
    if (this == &s) return *this; // falls Selbstzuweisung
    clear(); // Speicher freigeben
    Objekt *ptr = s.sz;
    while (ptr != nullptr) {
        enq(ptr->data);
        ptr = ptr->tail;
    }
    return *this;
}
```



this ist ein Zeiger auf das Objekt selbst.

Bei der Zuweisung wird ja keine neue Instanz erzeugt;
tatsächlich wird eine vorhandene Instanz verändert.

Deshalb ist Rückgabewert eine **Referenz auf sich selbst!**

Überladen von Operatoren: Test auf Gleichheit

```
template<typename T>
bool Schlange<T>::operator==(const Schlange<T>& s) {
    if (this == &s) return true;           // Selbstvergleich?
    Objekt *ptr1 = sz;                    // this->sz
    Objekt *ptr2 = s.sz;
    while (ptr1 != nullptr && ptr2 != nullptr) {
        if (ptr1->data != ptr2->data) return false;
        ptr1 = ptr1->tail;
        ptr2 = ptr2->tail;
    }
    return (ptr1 == ptr2);
}
```

Zwei Schlangen sind gleich genau dann, wenn sie

1. gleich viele Elemente haben und
2. die Inhalte in gleicher Reihenfolge paarweise gleich sind.

Unterschied zwischen Kopierkonstruktor und Zuweisung

Kopierkonstruktor:

Initialisierung einer **neu** deklarierten Variable von **existierender** Variable

Zuweisung:

- wirkt zwar wie Kopierkonstruktor (flache Kopie bzw. tiefe Kopie), überschreibt jedoch Speicher der **existierenden** Variable mit dem Speicher der zuweisenden, **existierenden** Variable
- zusätzlich ggf. Aufräumen: Freigabe dynamischer Speicher
- außerdem: Rückgabe einer Referenz auf sich selbst

Automatisch erzeugte Methoden erzwingen (C++11)

- Schlüsselwort **default** ist Anweisung an Compiler, die Standardimplementierung zu erzeugen
- z.B. für den parameterlosen Standard-Konstruktor

```
class Punkt {  
private:  
    double x, y;  
public:  
    Punkt() = default;  
};
```

Der Compiler soll seine Standardimplementierung verwenden

```
Punkt(double ax, double ay) : x(ax), y(ay){}  
};
```

Automatisch erzeugte Methoden *verhindern* (C++11)

- Schlüsselwort **delete** verhindert die Erzeugung von Methoden
- z.B. für Klassen, deren Instanzen nicht kopiert werden können

```
class Punkt {  
private:  
    double x, y;  
public:  
    Punkt(const Punkt& p) = delete;
```

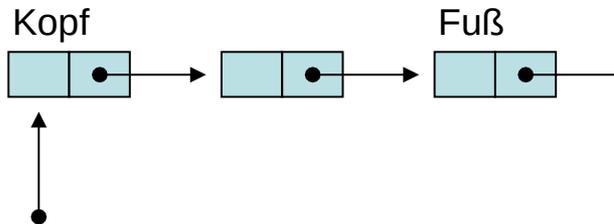
Der Compiler soll keinen Kopierkonstruktor erzeugen

```
    Punkt& operator=(const Punkt& p) = delete;
```

Der Compiler soll keinen Zuweisungsoperator erzeugen

```
};
```

ADT Liste (1. Version)



Liste wird nur durch **einen Zeiger** auf ihren Listenkopf repräsentiert

Operationen:

create : → Liste

empty : Liste → bool

append : T x Liste → Liste

prepend : T x Liste → Liste

clear : → Liste

is_elem : T x Liste → bool

hängt am Ende an
vor Kopf einfügen

ist Element enthalten?

ADT Liste (1. Version)

```
template<typename T>
class Liste {
public:
    Liste(); // Konstruktor
    Liste(const Liste<T>& liste); // Kopierkonstruktor
    void append(const T& x); // hängt hinten an
    void prepend(const T& x); // fügt vorne ein
    bool empty(); // Liste leer?
    bool is_elem(const T& x); // ist Element x in Liste?
    void clear(); // Liste leeren
    ~Liste(); // Destruktor
private:
    struct Objekt { // privater Datentyp
        T data; // Nutzdaten
        Objekt *next; // Zeiger auf nächstes
    };
    Objekt *sz; // Startzeiger auf
    Listenkopf
    void clear(Objekt *obj); // Hilfsmethode zum Leeren
};
```

ADT Liste (1. Version)

```
template<typename T>
Liste<T>::Liste() {
    sz = nullptr;
}
template<typename T>
void Liste<T>::clear(Objekt *obj) {
    if (obj == nullptr) return;
    clear(obj->next);
    delete obj;
}
template<typename T>
void Liste<T>::clear() {
    clear(sz);
    sz = nullptr;
}
template<typename T>
Liste<T>::~~Liste() {
    clear();
}
```

Laufzeit:

unabhängig von Listenlänge

rekursives Löschen von
„hinten“ nach „vorne“**Laufzeit:**

proportional zur Listenlänge



ADT Liste (1. Version)

```
template<typename T>
bool Liste<T>::empty() {
    return (sz == nullptr);
}
template<typename T>
bool Liste<T>::is_elem(const T& x) {
    Objekt *ptr = sz;
    while (ptr != nullptr) {
        if (ptr->data == x) return true;
        ptr = ptr->next;
    }
    return false;
}
template<typename T>
void Liste<T>::prepend(const T& x){
    Objekt *obj = new Objekt;
    obj->data = x;
    obj->next = sz;
    sz = obj;
}
```

Laufzeit:

unabhängig von Listenlänge

iterativer Durchlauf von
„vorne“ nach „hinten“

Laufzeit:

proportional zur Listenlänge

Laufzeit:

unabhängig von Listenlänge

ADT Liste (1. Version)

```

template<typename T>
void Liste<T>::append(const T& x) {
    Objekt *obj = new Objekt;
    obj->data = x;
    obj->next = nullptr;
    if (empty()) sz = obj;
    else {
        Objekt *ptr = sz;
        while (ptr->next != nullptr)
            ptr = ptr->next;
        ptr->next = obj;
    }
}

```

} neuen Eintrag erzeugen

Liste leer? → Kopf = neuer Eintrag

} iterativer Durchlauf von
„vorne“ nach „hinten“

Laufzeit:

proportional zur Listenlänge

```

template<typename T>
Liste<T>::Liste(const Liste<T>& liste) : sz(nullptr) {
    for (Objekt *ptr = liste.sz; ptr != nullptr; ptr = ptr->next)
        append(ptr->data);
}

```

Laufzeit: quadratisch proportional zur Listenlänge!

ADT Liste (1. Version)

Zusammenfassung:

1. Laufzeit von **clear** proportional zur Listenlänge
 - kann nicht verbessert werden, weil ja jedes Element gelöscht werden muss
 - unproblematisch, weil nur selten aufgerufen
2. Laufzeit des **Kopierkonstruktors** quadratisch proportional zur Listenlänge
 - kann nur verbessert werden, wenn **append** verbessert werden kann
 - bestenfalls Laufzeit proportional zur Listenlänge: muss alle Elemente kopieren!
3. Laufzeit von **is_elem** proportional zur Listenlänge
 - kann bei dieser **Datenstruktur** nicht verbessert werden
 - später verbessert durch ADT BinärerSuchbaum
4. Laufzeit von **append** proportional zur Listenlänge
 - kann durch Veränderung der **Implementierung** verbessert werden
 - zusätzlicher Zeiger auf das Ende der Liste