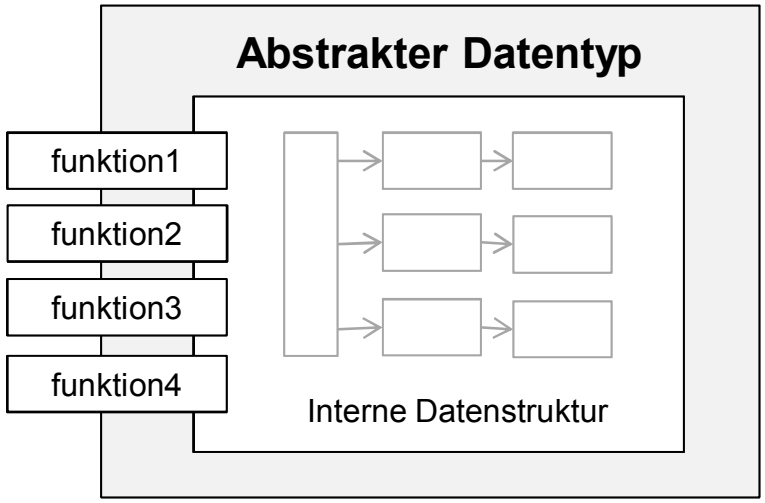
Kapitel 16

Abstrakte Datentypen

Die Trennung von WAS und WIE

Die Container des letzten Abschnitts bestanden jeweils aus einer Datenstruktur zusammen mit einer Reihe von Zugriffsfunktionen. Das Anwendungsprogramm griff ausschließlich über diese Schnittstelle auf den Container zu und musste nicht wissen, wie der Container implementiert war. Idealerweise sollte das Anwendungsprogramm nicht wissen, **WIE** der Container arbeitet und der Container sollte nicht wissen, **WAS** er verwaltet. Wenn man diese Trennung gedanklich vollzieht, kommt man zum Begriff des abstrakten Datentyps.

> Ein **Abstrakter Datentyp** ist eine Datenstruktur zusammen mit einer Reihe von Operationen, die auf dieser Datenstruktur arbeiten.
>
> Der abstrakte Datentyp verbirgt nach außen seine Implementierung und wird ausschließlich über die Schnittstelle seiner Operatoren bedient.



Der Lebenszyklus abstrakter Datentypen

Abstrakte Datentypen haben in der Regel einen **Konstruktor**, der die Aufgabe hat, den Datentyp vor seiner ersten Verwendung in einen konsistenten Anfangszustand zu bringen.

Da der Datentyp nach der Konstruktion nur über die zulässige Schnittstelle verändert werden kann, kann er durch einen Anwender nicht in einen inkonsistenten Zustand gebracht werden.

In der Regel hat ein abstrakter Datentyp auch einen **Destruktor**, der, wenn der Datentyp nicht mehr benötigt wird, dafür sogt, dass er sauber wieder beseitigt wird.

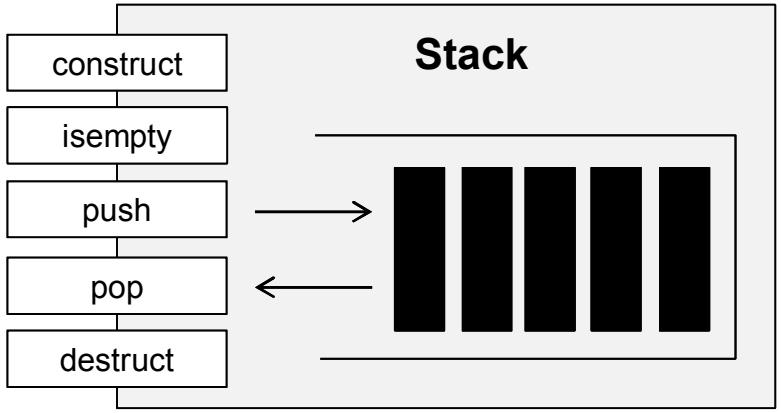
Die Programmiersprache C unterstützt das Konzept der abstrakten Datentypen nicht. Trotzdem ist es sehr sinnvoll, beim Programmdesign immer das Konzept des abstrakten Datentypen als Entwurfsmuster zu verwenden.

Das Thema der abstrakten Datentypen erhält erst in C++ mit Klassen und Templates einen befriedigenden Abschluss.

Der Stack als abstrakter Datentyp

Wir wollen einen Stack implementieren, der einen ihm unbekannten Datentyp verwaltet, von dem er nur die Größe (in Bytes) kennt.

Neben Konstruktor und Destruktor gibt es die Operationen `push` und `pop` und eine Funktion `isempty`, die testet, ob der Stack leer ist.



Operation	Eingehende Parameter	Ausgehende Parameter	Beschreibung
construct	Stackgröße und Elementgröße	Stack	Erzeuge einen leeren Stack der gewünschten Stackgröße für Elemente der gewünschten Elementgröße.
isempty	Stack	0 oder 1	Teste, ob der Stack leer ist.
push	Stack und Element	OK oder OVERFLOW	Lege ein Element auf den Stack.
pop	Stack	EMPTY oder OK und sofern OK, das oberste Element vom Stack	Hole ein Element vom Stack.
destruct	Stack		Beseitige den mit construct erzeugten Stack.

Die Schnittstelle des Stacks

```
# define OK 1
# define OVERFLOW -1
# define EMPTY 0

struct stack
    {
    char *stck;
    int ssize;
    int esize;
    int pos;
    };

struct stack *stack_construct( int ssiz, int esiz);
void stack_destruct( struct stack *s);
int stack_isempty( struct stack *s);
int stack_push( struct stack *s, void *v);
int stack_pop( struct stack *s, void *v);
```

Bei der Konstruktion wird festgelegt, wie viele Elemente maximal auf dem Stack liegen können (`ssiz`) und wie groß die einzelnen Elemente (`esiz`) sind.

Der Stack kennt an der Schnittstelle nur die Größe der zu verwaltenden Datenpakete und erhält daher einen unspezifizierten Zeiger (`void *`), wenn er die Daten auf den Stack legen oder vom Stack nehmen soll.

Die Implementierung des Stacks

Konstruktor

```
struct stack *stack_construct( int ssiz, int esiz)
    {
    struct stack *s;
    s = (struct stack *)malloc( sizeof( struct stack));
    s->stck = (char *)malloc( ssiz*esiz);
    s->ssize = ssiz;
    s->esize = esiz;
    s->pos = 0;
    return s;
    }
```

Destruktor

```
void stack_destruct( struct stack *s)
    {
    free( s->stck);
    free( s);
    }
```

push

```
int stack_push( struct stack *s, void *v)
    {
    if( s->pos >= s->ssize)
        return OVERFLOW;
    memcpy( s->stck + s->pos*s->esize, v, s->esize);
    s->pos++;
    return OK;
    }
```

isempty

```
int stack_isempty( struct stack *s)
    {
    return s->pos == 0;
    }
```

Vergleich auf Gleichheit, keine Zuweisung.

pop

```
int stack_pop( struct stack *s, void *v)
    {
    if( !s->pos)
        return EMPTY;
    s->pos--;
    memcpy( v, s->stck + s->pos*s->esize, s->esize);
    return OK;
    }
```

Die Funktion `memcpy(dst,src,size)` kopiert eine gewisse Anzahl (`size`) Bytes von einer Quelladresse (`src`) zu einer Zieladresse (`dst`).

Die Verwendung des Stacks

```
struct test
    {
    int i1;
    int i2;
    };

void main()
    {
    struct stack *mystack;
    struct test t;
    int i;

    srand( 12345);
    mystack = stack_construct( 100, sizeof( struct test));
    for( i = 0; i < 5; i++)
        {
        t.i1 = rand( )%1000;
        t.i2 = rand()%1000;
        printf( "(%3d, %3d) ", t.i1, t.i2);
        stack_push( mystack, &t);
        }
    printf( "\n");
    while( !stack_isempty( mystack))
        {
        stack_pop( mystack, &t);
        printf( "(%3d, %3d) ", t.i1, t.i2);
        }
    printf( "\n");
    stack_destruct( mystack);
    }
```

Struktur, die auf den Stack soll

Zeiger auf den abstrakten Datentyp

Konstruktion eines Stacks für 100 Datenstrukturen der entsprechenden Größe

Push von Zufallsdaten

Test auf leeren Stack

Pop der Testdaten

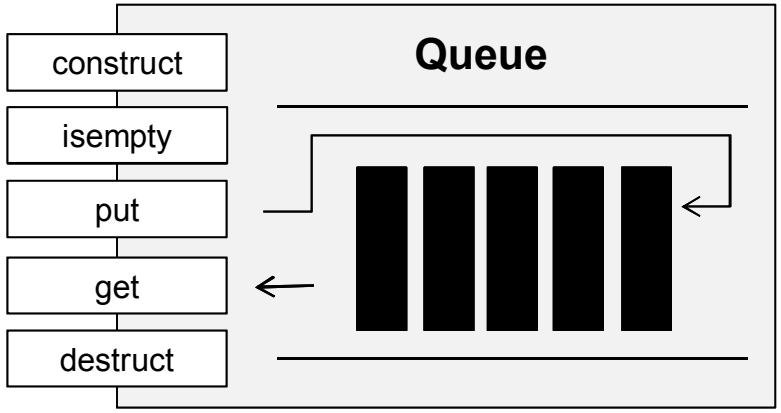
Beseitigung des Stacks

```
(584, 164) (795, 125) (828, 405) (477, 413) ( 72, 404)
( 72, 404) (477, 413) (828, 405) (795, 125) (584, 164)
```

Die Queue als abstrakter Datentyp

Auch die Queue soll einen ihr unbekannten Datentyp verwalten, von dem sie nur die Größe (in Bytes) kennt.

Neben Konstruktor und Destruktor gibt es die Operationen `put` und `get` und eine Funktion `isempty`, die testet, ob die Queue leer ist.



Operation	Eingehende Parameter	Ausgehende Parameter	Beschreibung
construct	Queuegröße und Elementgröße	Queue	Erzeuge eine leere Queue der gewünschten Queuegröße für Elemente der gewünschten Elementgröße.
isempty	Queue	0 oder 1	Teste, ob die Queue leer ist.
put	Queue und Element	OK oder OVERFLOW	Lege ein Element in die Queue.
get	Queue	EMPTY oder OK und sofern OK, das nächste Element aus der Queue	Hole ein Element aus der Queue.
destruct	Queue		Beseitige die mit construct erzeugte Queue.

Die Schnittstelle der Queue

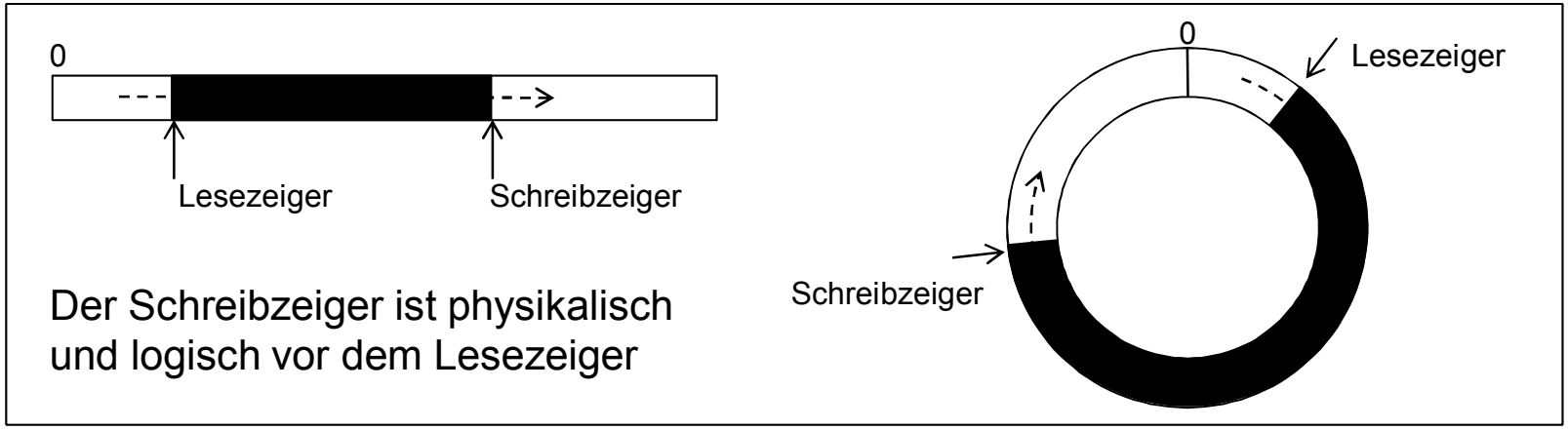
```
# define OK 1
# define OVERFLOW -1
# define EMPTY 0

struct queue
    {
    char *que;
    int qsize;
    int esize;
    int first;
    int anz;
    };

struct queue *queue_construct( int qsiz, int esiz);
void queue_destruct( struct queue *q);
int queue_isempty( struct queue *q);
int queue_put( struct queue *q, void *v);
int queue_get( struct queue *q, void *v);
```

Bei der Konstruktion wird festgelegt, wie viele Elemente maximal in der Queue liegen können (`ssiz`) und wie groß die einzelnen Elemente (`esiz`) sind.

Die Queue kennt im weiteren nur die Größe der zu verwaltenden Datenpakete und erhält daher einen unspezifizierten Zeiger (`void *`), wenn sie die Daten in die Queue legen oder aus der Queue nehmen soll.

Exkurs Ringpuffer (Sebastian Vettel kann hinter Fernando Alonso herfahren und trotzdem in Führung liegen)

Ein Ringpuffer ist ein Array, der gedanklich zu einem Ring geschlossen ist, sodass man, wenn man hinten herausläuft, vorn wieder hineinkommt. In einem Ringpuffer kann man eine Queue mit Schreib und Lesezeiger anlegen, die nicht aus dem zugrundeliegenden Array hinausläuft. Man muss nur darauf achten, dass der Schreibzeiger den Lesezeiger nicht überrundet.



Die Zeigerbewegungen können mit einfachen Modulo-Operationen implementiert werden:

```
zeiger = (zeiger + offset)%pufferlaenge
```

Die Implementierung der Queue

```
struct queue *queue_construct( int qsiz, int esiz)
    {
    struct queue *q;
    q = (struct queue *)malloc( sizeof( struct queue));
    q->que = (char *)malloc( qsiz*esiz);
    q->qsize = qsiz;
    q->esize = esiz;
    q->first = 0;
    q->anz = 0;
    return q;
    }
```

Konstruktor

```
void queue_destruct( struct queue *q)
    {
    free( q->que);
    free( q);
    }
```

Destruktor

```
int queue_put( struct queue *q, void *v)
    {
    if( q->anz >= q->qsize)
        return OVERFLOW;
    memcpy( q->que + ((q->first+q->anz)%q->qsize)*q->esize, v, q->esize);
    q->anz++;
    return OK;
    }
```

put

```
int queue_isempty( struct queue *q)
    {
    return q->anz == 0;
    }
```

isempty

```
int queue_get( struct queue *q, void *v)
    {
    if( !q->anz)
        return EMPTY;
    memcpy( v, q->que + q->first*q->esize, q->esize);
    q->first = (q->first+1)%q->qsize;
    q->anz--;
    return OK;
    }
```

get

Die Queue wird als Ringpuffer implementiert. Der Schreibzeiger läuft dem Lesezeiger immer um `q->anz` Elemente logisch voraus, wobei im Array Modulo `q->qsize` gerechnet wird. Der Schreibzeiger kann sich physikalisch hinter dem Lesezeiger befinden, überholt ihn aber nicht, da immer `q->anz < q->qsize` ist.

Die Verwendung der Queue

```
struct test
    {
    int i1;
    int i2;
    };

void main()
    {
    struct queue *myqueue;
    int i;
    struct test t;

    srand( 12345);

    myqueue = queue_construct( 100, sizeof( struct test));
    for( i = 0; i < 5; i++)
        {
        t.i1 = rand( )%1000;
        t.i2 = rand( )%1000;
        printf( "(%3d, %3d) ", t.i1, t.i2);
        queue_put( myqueue, &t);
        }
    printf( "\n");
    while( !queue_isempty( myqueue))
        {
        queue_get( myqueue, &t);
        printf( "(%3d, %3d) ", t.i1, t.i2);
        }
    printf( "\n");
    queue_destruct( myqueue);
    }
```

Struktur, die in die Queue soll

Zeiger auf den abstrakten Datentyp

Konstruktion einer Queue für 100 Datenstrukturen der entsprechenden Größe

Put von Zufallsdaten

Test auf leere Queue

Get der Testdaten

Beseitigung der Queue

```
(584, 164) (795, 125) (828, 405) (477, 413) ( 72, 404)
(584, 164) (795, 125) (828, 405) (477, 413) ( 72, 404)
```