```
    bool empty(); // liefert { true   falls Stack leer
                             { false  sonst
    /* Alle Operationen haben Laufzeit O(1). Der Platzbedarf ist
    proportional zur Anzahl der Elemente in Stack.*/
}
push: int_stack ×int→int_stack
pop: int_stack×int_stack⇝int_stack×int
top: int_stack⇝int
```

4.05.09

1.1.2 Bemerkungen zur Klassendeklaration in C++ (*int_stack*)

Konstruktor *int_stack()*, realisiert *create* →*int_stack*

- hat gleichen Namen wie die Klasse
  - Kann auch Argumente haben, z.B. *int_stack(int sz)* ←maximale Größe *(create:int→int_stack)*
  - kein Resultat
  - wird automatisch aufgerufen bei der Deklaration einer Variable des Datentyps *int_stack S;* mit Argumenten *int_stack*

  Parameterübergabe an Konstruktor auch mit =

Beispiel

```
string x("hallo");
string x = "hallo";
```

Destruktor *~int_stack();*

- ~ + Klassenname, keine Argumente
  - automatisch aufgerufen, wenn Gültigkeitsbereich (Lebensdauer, Scope) einer Varibale des Datentyps endet (am Ende des Blocks und der Variable deklariert wurde).

```
{ int_stack S,
    ...
    s.push(17);
    ...
} //←Aufruf von ~ int_stack() für Variable S
```

  - wird hauptsächlich verwendet, um belegten Speicherplatz freizugeben.

1. Bei den "normalen" Methoden (*push, pop*, ...) erscheint das Objekt (s), auf dem die Operation ausgeführt wird, nicht in der Parameterliste. Es wird implizit übergeben (ist aber bei der Implementierung über den sog. "*this*"-Pointer verfügbar).
   Zum Aufruf der Operation verwendet man die Memberschreibweise

   ```
   S.push(17) // (statt push(S,17) in sprache "C")
   ```

   Operationen heißen auch Memberfunktionen oder Methoden.

2. Die Operationen "*top*" und "*empty*" verändern ihr Objekt nicht.
   Dies kann man mit dem Stichwort *const* hinter der Parameterliste anzeigen.

   ```
   int top() const;
   bool empty() const;
   ```

   Der Compiler nutzt diese Hinweise, um *const*-Regeln zu überprüfen. Insbesondere dürfen auf Konstanten des Datentyps nur *const*-Operationen aufgerufen werden (später mehr hierzu).

3. Parameter
   Normale Parameterübergabe by value (nicht Referenz →JAVA)

   ```
   S.push(x) // Kopie von x wird auf stack S gepusht.
   ```

   (→Kontrolle der Art der Parameterübergabe, später)

1.2 Implementierung von Datentypen

1.2.1 Implementierung von Stacks

(im Gegensatz zur Spezifikation)

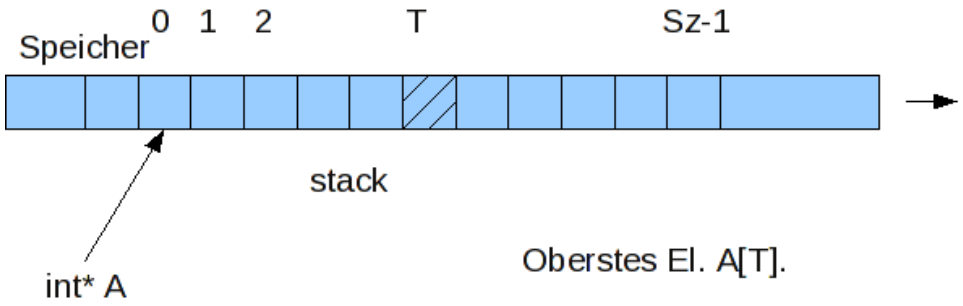
Array-Implementierung. Datenstruktur besteht aus

1. Feld ganzer Zahlen
   a) Verweis auf oberstes Element
   b) Größe des Feldes

   ```
   class int_stack {
       ...
   private: int* A; // Feld (Pointer auf 1. El)
            int T;  // Top-Index
            int sz; //Größe des Feldes
   ```

   Pointer, Felder, Referenzen in C/C++ bitte anschauen.

Abbildung 2: Array-Implementierung



INVARIANTE der Datenstruktur

1. A ist Feld von sz Elementen.
2. $-1 \leq T \leq \text{sz} - 1$
3. Inhalt des Stacks ist dann $A[0], A[1], ..., A[T]$, wobei $A[T]$ das oberste Element (top).
4. Der Stack ist leer genau dann wenn $T = -1$

Anlegen eines Feldes

```
int* A = new int[100];
A[7]⟺*(A+7)←Zugriff über Pointer
```

```
Freigabe: delete[] A;
```

1.2.2 Implementierung der Operationen (Methoden)

Die in der Klasse deklarierten Methoden werden nun definiert.

Syntax

```
Resultatstyp, Klassenname :: Operation(Parameterliste) const
{
    Rumpf
}
```

```
Konstruktor
```

```
int_stack::int_stack()
{ //initialisiere Datenstruktur
```

```
        sz = 100 //am Anfang Feld der Größe 100 (INVARIANTE)
        A= new int[sz];
        T = -1; //leerer Stack

    }
```

Destruktor

```
    int_stack ::~int_stack() // pop-Operation
    { delete[]A; }
```

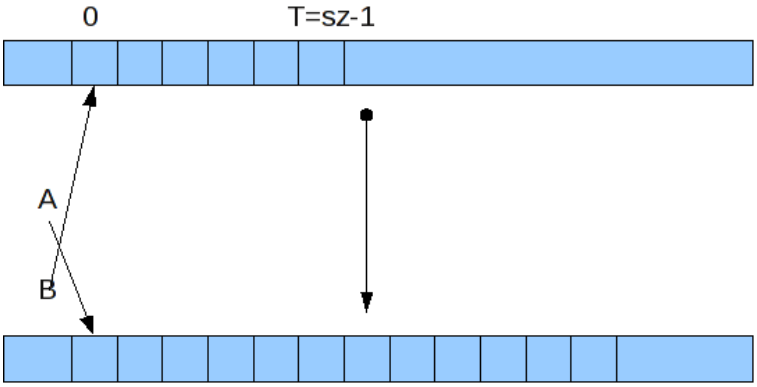
push-Operation: verdoppelt Feld, falls es zu klein ist

***void**: int_ stack::push(int x)*

```
    void: int_stack::push(int x)
    {_
        if (T+1 == sz)
        { // verdopple Feld
            int* B = A;
            sz *= 2;
            A = new int [sz];
            for (int i = 0; i <= T; i++)
                A[j] = B[j];
            delete[] B;
        }
        A[++T]=x;

    }
```

Abbildung 3: Verdopple Feld



***pop*-Operation:** nur auf nicht-leeren Stack (Precondition)

```
int int_stack::pop()
{
    if (T == -1)
    {
        EXCEPTION (''pop auf leerem Stack'');
    }
    return A[T--];
}
```

```
bool int_stack::empty() const
```

```
bool int_stack::empty() const
{
    return T == -1;
}
```

11.05.09

Dateien *int_stack.h* →Deklaration der Klasse (Spezifikation)
int_stack.cpp →Implementierung der Operatoren (Methoden)

Anwendungsprogramm

```
#include <int_stack.h>
...
int_stack S;
S.push(17);
...
```

C++ inline Funktionen Bei kleinen Funktionen z.B. *s.empty()* ist Aufwand für Aufruf und Rückkehr oft viel größer als die tatsächlichen Kosten der Funktion.

Dann ist es viel effizienter den Rumpf an der Aufrufstelle (als Programmtext) einzukopieren, statt einen wirklichen Funktionsaufruf auszuführen. (vgl. mit Makro-Konzept in C).

⇒Die Definition der Inline-Funktion muss in *int_stack.h* (HeaderDatei) stehen!

2 Möglichkeiten (in h-Datei)

Nach Klasse

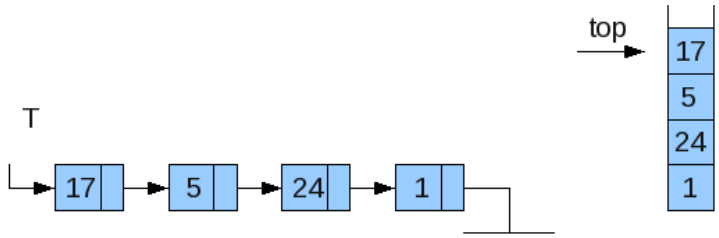
```
inline bool int_stack::empty() const
    { return T == -1; }
```

Schreibe den Rumpf direkt hinter die Deklaration der Methode in der Klasse

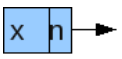
```
class int_stack {
    ...
        public
        (inline) bool empty() const {return T == -1; };
```

1.2.3 Implementierung 2: Listen

Abbildung 4: Liste




```
class int_stack { //←int_stack.h
    class stack_elem;
    {
        int vol;
        stack_elem* netx;
        public
        stack_elem(int x, stack_elem* n)
        { val = x; next = n; } //Konstanter
        stack_elem(int x, stack_elem* n): val(x),next(n){} //alternativ
        //besser
    };
```



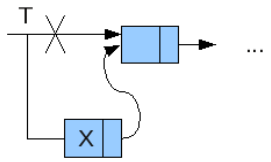

```
    int_stack.cpp
    //Konstruktor
```


```
        int_stack::int_stack ()
        {T = NULL;}
        bool int_stack::empty() const
        { return T ==s NULL; }
        void int_stack:: push (int x)
        { //vorne einfüge
            T= new stack_elem (x,T);}
```

```
//Destruktor: muss alle Listenelemente mit delete freigeben
```

→Vollständige Implementierung: Übung.

Abbildung 5: das Einfügen



Laufzeit

Listenimplementierung: O(1) pro Operation worst-case!

Array: O(1) amortisiert (O(n) worst-case!)

In der Praxis ist die Array-Implementierung meistens schneller, weil sie die Daten kompakt (lokal) im Speicher ablegt →sehr gutes Cache-Verhalten! (später mehr dazu)

Korrektheit unserer Implementierung Zunächst müssen wir definieren, was Korrektheit bedeutet.

Eigentlich haben wir jeweils 2 Datentypen.

1. den abstrakten Datentyp *stack* (durch die Deklaration der Klasse in *stack.h*)
2. den konkreten Datentyp durch die Definition der privaten Daten und der Methoden (*stack.cpp*).

Wir müssen zeigen, dass diese beiden Typen äquivalent sind.

1.2.4 Array-Implementierung

abstrakter Zustand Folge von *int*'s.

konkreter Zustand Inhalt der Variablen A, sz, T

In der Implementierung garantieren wir, dass nicht alle Kombinationen von T, sz, A möglich sind, sondern nur die mit

1. A ist Feld der Länge sz (siehe INVARIANTE)
2. $-1 \leq T < sz$

→gültige korrekte Zustände Z

Um die Korrektheit zu zeigen

1. definiere eine Abbildung F von den korrekten Zuständen Z in die Folge der abstrakten Zustände S mit $\overbrace{A, sz, T}^{Z} \xrightarrow{F} \begin{cases} \text{Folge A[0], ... , A[T]} & \text{falls} T \geq 0 \\ \text{leere Folge} & \text{sonst} \end{cases}$
2. Jede Operation *op* des abstrakten Datentyps wird realisiert durch Methode f_{op} des konkreten Typs

   s ist abstsrakter Zustand, d.h. $s \in S$.

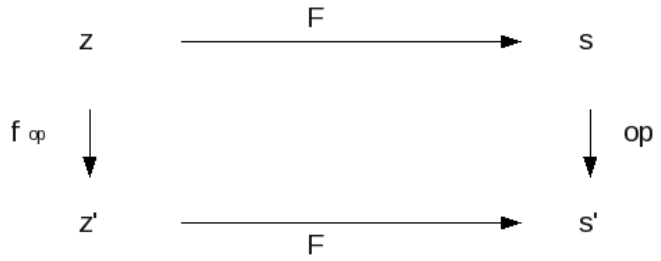
   Beispiel $\mathsf{push}(s, x) \rightarrow s'$

   $\mathrm{f}_{\mathrm{push}} \overbrace{A, sz, T}^{Z}, x \rightarrow A', sz', T'$

   (z ist konkreter gültiger Zustand $z = A, sz, T$)
3. Zeige, dass für jede Operation *op* und gültigen konkreten Zustand z gilt.

   $F\left(f_{op}(s)\right) = op\left(F(z)\right)$

Abbildung 6: kommutatives Diagramm



4. Zeige, dass die Konstruktoren

   a) gültige konkrete Zustände erzeugen

   b) die Create-Operation realisieren

beim Stack F (A,sz,T) = A[0], ...,A[T]

18.05.09

Korrektheit der int_stack Implementierung →Übung

dazu:

1. zeige F(für alle Operationen
2. zeige, dass Konstruktoren gültige konkrete zustände erzeugen

Zur Vervollständigung der Datentypenparameter (Resultatsrückgabe) und Wertzuweisung in C++ grundsätzlich 2 Arten:

Übergabe "by value". Es wird eine Kopie des Objektes an die Funktion übergeben.

Syntax: F(T x) z.B. F (int i)

```
void int_stack::push (int x)
```

Aufruf F(y), dann wird im Rumpf von F der formale Parameter x mit einer Kopie des aktuellen Parameters initialisiert. Änderungen des Wertes vom formalen Parameter x haben einen Einfluss auf den Wert von y.

Übergabe "by reference" Syntax: F(T & X) z.B. F(int & i)

Es wird das Objekt selbst übergeben (Referenz).

Aufruf F(y).

Im Rumpf bezeichnet der formale Parameter y dasselbe Objekt im aktuellem Parameter x (keinefalls eine Kopie) d.h. Änderung vom y ändert auch x.

Beispiel

```
void cmp_and_swap(int & a, int & b)
{
    if(a>b)
    {
        int tmp = a;
        a = b;
        b = tmp;
    }
}
```

Aufruf

```
int x = 17; int y = 7;
cmp_and_swap(x,y)
```

Variante "const reference" Es wird das Objekt selbst übergeben, aber im Rumpf der Funktion wie eine Konstante behandelt (Wert darf nicht verändert werden)

1. nur const-Methoden
2. keine Wertzuweisung
3. mit Übergabe by Value oder const

Syntax: F (const T & x)

Anwendung Rumpf ändert den Wert von x nicht

- by value Übergabe zu teuer (Kopie)

Rückgabe von Resultaten

1. by value

```
T F( ... )
{
    T x;
    ...
    return x;
}
```

Aufruf `y = F(...) Kopie der lokalen Variable x im Rumpf`

2. by reference

```
T & F (...)
```

Beispiel

```
int max (int & a, int & b)
{ return (a > b) ? a:b; }
x = 17, y = 7;
max (x, y) = 13;
//variable (L-value)
```

Array-Klasse

```
class int_array
{
    int & elm (int i ) { ... }
    A.elem(i) = 17;
```

```
}
```

später: A [i] →Überladen von Operatoren.

3. const int & max (const int & a, const int & b)

Definition der “by value” Übergabe für Klassen-Form

→copy Konstruktor

Wenn nichts anders gesagt wird, dann werden Klassenobjekte bitweise kopiert (jedes Datenfeld).

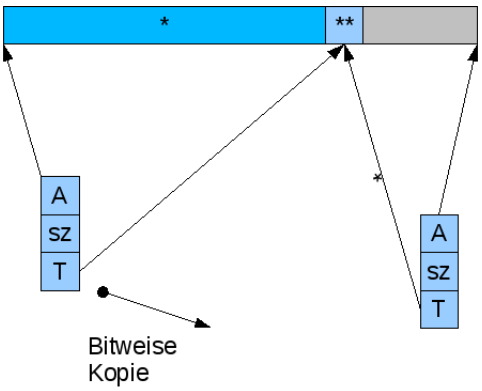
Beispiel: int_stack

```
void F (stack s)
{
    s.pop(); *
    s.push (17); **
}
```

Aufruf stack S;

F(S”)

Abbildung 7: Aufruf von stack S



⇒Aufruf: F(s’) verändert s’

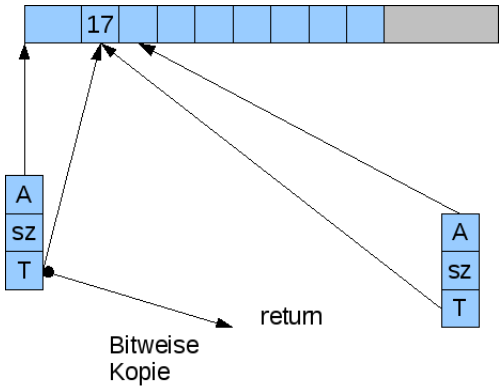
Resultatrückgabe

Beispiel

```
int_stack F (int x)
{
```

```
    int_stack S;
    S.push (x)
    return S;
}
int y = F(17).top();
```

Abbildung 8: push-Operation



Konstruktor von int_stack gibt Feld am Ende von F für d.h. Feld existiert nicht außerhalb von F - Widerspruch!

⇒Bitweise Kopie reicht für Komplexe Klassen nicht aus

Lösung in C++ Definition eines copy-Konstruktors der immer dann verwendet wird, wenn eine Kopie gemacht werden muss (bei Übergabe)

Syntax für Klasse T

```
class T
{
    T (const T & x);
    /*erzeugt eine Kopie von x
    durch entsprechende Initialisierung
    der Datenfelder von T.
```

für int_stack

```
class int_stack { const int_stack & s )
{
    sz = S.sz;
    T = S.T;
```

```
        A = new int [sz];
        for (int i = 0; i < sz; i++)
        {
            A[i] 0 S.A[i];
        }
    }
```

Wertzuweisung x = y;

Default: x wird bitweise durch y überschreiben

Probleme

1. wie beim copy-Konstruktor
2. altes Objekt wird nicht zerstört

Destruktor

Lösung in C++ Definiere einzelne Wertzuweisungs-Operatoren →allgemein überladen von Operatoren (nachlesen!)

25.05.09

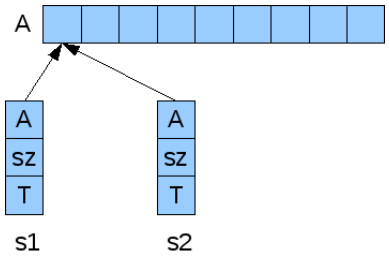
Wertzuweisung: x = y;

Default: Datenfelder von x werden bitweise durch die Datenfelder von y überschrieben.

Gleiches Problem wie beim copy constructor in_stack s1, int_stack s2

s1 = s2; ⇒

Abbildung 9: s1=s2



zeigen auf dasselbe Feld.

Lösung: Überladen des "=" Operators, ...

(→Übung Nachher wie Operatoren in c++ überladen werden)

Syntax: Memberfunktion für eine Klasse T

```
class T
{
    public
    T & operator = (const T &);
```

Aufruf: T x,y; x = y; wird vom Compiler durch x.operator=(y); ersetzt.

$a = b = c; \Rightarrow a = (b = c); \Rightarrow$a.operator=(b.operator=(c));

Implementierung von x.operator = (y) muss

1. x zerstören (wie im Destruktor)
2. Kopie von y erzeugen (wie Copy-Konstruktor)

im Beispiel:

```
class int_stack
{
    puclic: int_stack & operator =(const int_stack);
    int_stack & int_stack :: operator = (const int_stack S)
    {
        delete[] A; //Zerstörung
        sz = S.sz;
        T = S.T;
        A = new int [sz]
        for ( int i = 0; i < sz; i++) A[i] = S.A[i];
        return *this;
    }
```

Problem: S=S; delete [] A serstört S.

Einfache Lösung: teste, ob this == & S;

⇒1. Zeile des Operators:

```
if (this == & S) return * this;
```

andere wichtige Operatoren (je nach Typ):

- Test auf Gleichheit

```
class T
{
    bool operator == ( const T & x);
    bool operator < ( const T & x);
    bool operator > ( const T & x);
    bool operator != ( const T & x);
    ...
```

```
}
```

- arithmetische Operatoren: +, -, *, / ...
- spezielle Operatoren:
  - Feldzugriff: operarot []
  - Funktionsaufruf operator ()

  Übung: Ein/Ausgabe in C++ →streams nachlesen!
- I/O-Operatoren << (Ausgabe), >> (Eingabe)

Beispiel: Konsolen I/O

```
cin >> x; // Eingabe
cout << x << endl; // Ausgabe
```

2 Parametrisierte Datentypen oder generische Datentypen

d.h. verwendbar für beliebige Elementtypen

Beispiel: stack von int, double, string, ...

2 Strategien:

1. Ableitung (Vererbung)
2. Templates

Ziel: Wiederverwendung von Programm-Code.

2.1 Vererbung und dynamisches Binden

Situation Man braucht einen Datentyp A der ähnlich zu einem bereits definierten Typ B (→Klasse) ist. A soll

- einen Teil der Daten/Operationen von B wiederverwenden
- andere hinzufügen
- einige verändern (auf andere Weise implementieren)

Beispiel: →Spezialisierung Datentyp Polygon existiert, wir wollen Datentyp Rechteck definieren.

Da jedes Rechteck ein Polygon ist ("ist ein" Relation), können alle Polygon-Operationen übernommen werden. Einige Operationen können effizienter implementiert werden. (z.B. Flächeninhalt), andere sind nur für Rechtecke definiert (z.B. Seitenverhältnis).