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).

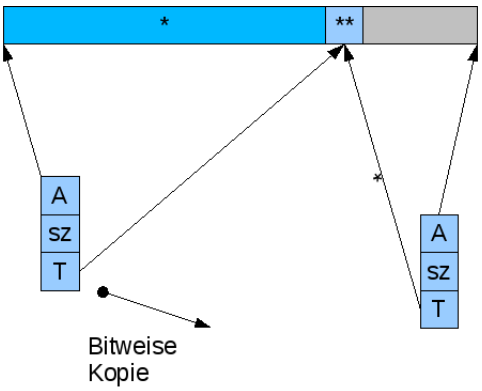
Beispiel: int_stack

```
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’

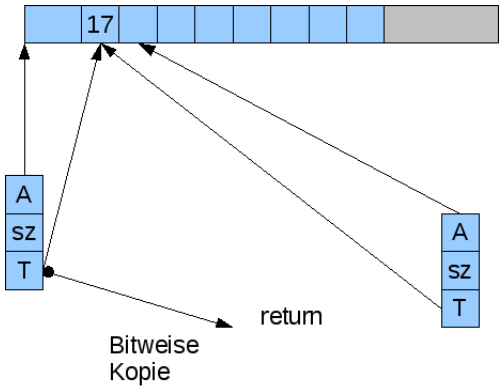
Resultatrückgabe

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

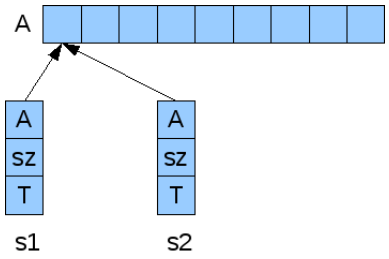
Wertzuweisung: x = y;

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).

2.1.1 Allgemeinvererbung:

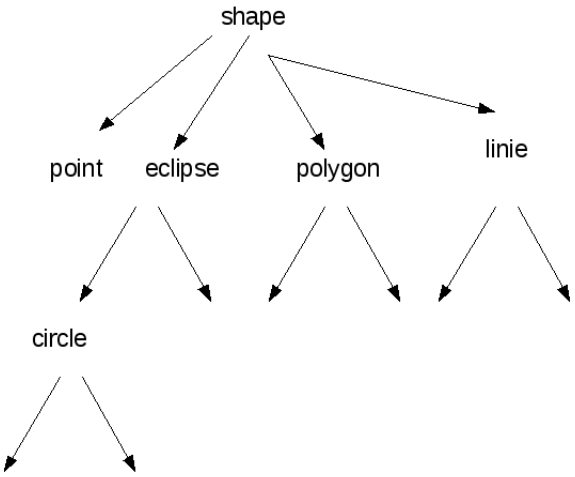
wir leiten Klasse A (rechteck) von einer Basis-Klasse B (polygon) ab

Syntax

```
class A: public B //oder private
{/* alle public (private members von B

    sind auch public (private) in A */
    A-Teil;
    //zusätzliche Daten & Operationen
    //Operationen überschreiben

};
```

B(Basisklasse) →A(Abgeleitete Klasse)

Abbildung 10: Ableitungsbaum

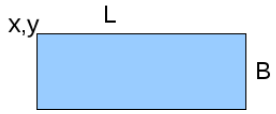


Beispiel: Ableitungsbaum

```
class polygon
{

    private // Daten
    public:
    polygon ( int n, double * x; double * y)
    ~polygon ();
    double umfang() const;
    double flaeche() const;
```

```
        void rotate (double phi);
        void translate (double dx, double cy);
        void draw ( ... );

    };
    class rechteck: public polygon
    {

        private
        double L,B; // Länge und Breite
        public
        rechteck (double xm double y, double B, double L);
        //-> polygon Konstruktor)
```

Abbildung 11:




```
        //alle Operationen von Polygon werden geerbt
        double ratio() const { return L/B; }
        double flaeche() const { return L*B; }
        double umfang() const { return 2*(A+B); }

    }
```

8.06.09

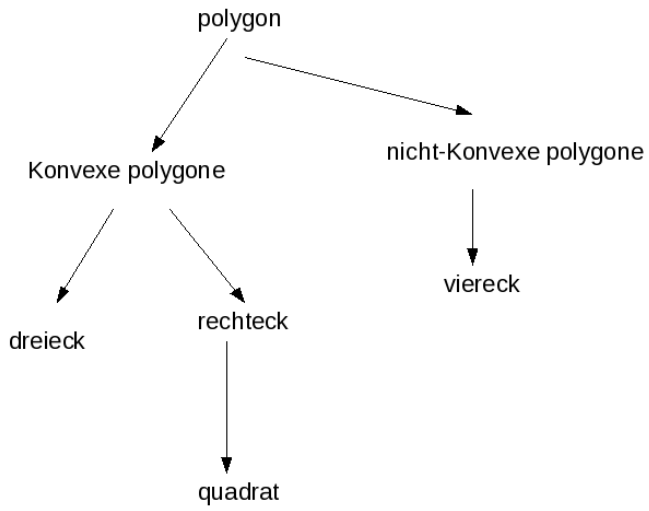
allgemeines Objekt: polygon

Spezialisierung "ist ein" rechteck Vererbung

```
    class polygon { //Basisklasse

        double area (); //Fläche
        ...

    };
    class rechtechk: public polygon //abgeleitete Klasse
    {

        double L,B;

    ...
    double area () { return L*B;} //spezielle Implementierung
    double ratio () { return L/B;} //spezielle Operation
```

Im Allgemeinen Ableitungsbaum

Abbildung 12: Ableitungsbaum



2.1.2 Typverträglichkeit

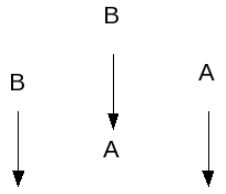
Typverträglichkeit zwischen Basisklasse & abgeleiteter Klasse.

Einer Veriable (Parameter) vom Typ B* (Pointer) oder B& (Referenz) kann ein Wert vom Typ A* bzw. A& zugeordnet werden.

Beispiel:

```
polygon * p = new rechteck (...);

    rechteck R ( ... );
    p = & R

Funktion double flaeche (polygon & P)
{ return P.area();}
```

Abbildung 13: Typverträglichkeit



Aufruf

```
polygon poly ( ... )
double f = flaeche(poly);
rechteck rect ( ... );
f = flaeche (rect)
```

Achtung Im Rumpf der Funktion ist nur der Polygon-Teil des Rechtecks sichtbar, d.h. die Funktion behandelt rect wie ein Polygon.

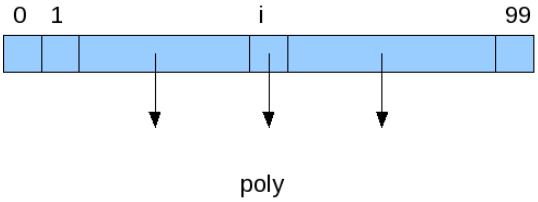
Insbesondere ist der Aufruf *P.area()* immer ein Aufruf der Methode *polygon::area()*!

→statische Bindung.

Besseres Beispiel Feld von Polygonen (Pointer auf Polygone)

```
A = new polygon * [100];
double gesamt_fläche (polygon ** A, int n)
{
    double sum  = 0;
    for ( int i = 0; i < n ; i++ )
        sum += A[i] -> area();
        return sum;
}
```

Abbildung 14: Feld von Polygonen



Feld A kann nun Pointer auf alle von Polygon abgeleiteten Klassen enthalten!

(→polymorphes Feld)

```
A[0] = new polygon (...);
A[1] = new rechteck (...);
A[2] = new dreieck (...);
...
```

statische Bindung Funktion *gesamt_flaeche* verwendet für alle Objekte *polygon::area* (der Basisklasse) obwohl es die effizienteren Implementierungen für abgeleitete Klassen gibt.

Lösung in C++ virtuelle Methoden für dynamische Bindung ("wirkliche" typ bestimmt zu Laufzeit welche Methode verwendet wird)

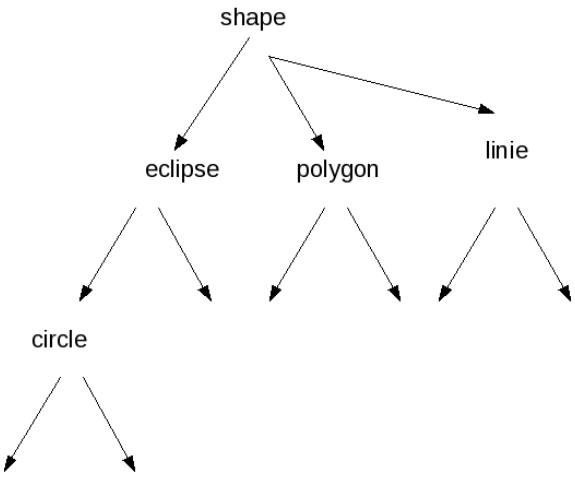
```
class polygon
{
    virtual double area() const;
    class rechteck:public polygon
    ...
    double area () const; // optional: virtual
```

Jetzt wird in Funktion *gesamt_flaeche* die richtige Methode "*area*" aufgerufen.

2.1.3 Abstrakte Klassen_oder Interfaces durch rein virtuelle Funktion

Beispiel Basisklasse *shape* (im Zeichenprogramm)

Abbildung 15: Basisklasse




```
class shape {
    virtual double area() const = 0;
    virtual void draw (...) const = 0;
    virtual void write (ostream &);
    virtual void read (ostream &);
```

Keine Objekte von abstrakten Klassen

```
shape s; //Compiler Fehler!
shape * s = new circle (...); //OK!
A = new shape*[n];
A[0] = new circle ( ...);
A[1] = new polygon ( ...);
class editor {
    void push (shape * p)
    void draw () {
        for (i = 1; i < n; i ++ ) A [i] -> draw ();
```

Vorteil: Flexibilität

Man kann sehr leicht neue Objekttypen hinzufügen, ohne den Code für die editor-Klasse zu verändern.

2.1.4 Anwendung in Algorithmen

Sortieren (Quicksort, Heapsort)

- generische Implementierung. Soll für alle Typen mit Vergleich funktionieren.

Lösung Interface (abstrakte Basisklasse)

```
class comparable {
    virtual bool lte (const compatable & c) const = 0;
    };
```

Sortieralgorithmus Quicksort (comparable** A, int n)

```
//Vergleich A[i] <= A[j]
    A[i] -> lte(*A[j]);
```

Quicksort akzeptiert Felder von Pointern auf alle Klassen, die von comparable abgeleitet sind (d.h. die das Interface implementieren)

Beispiel

class point:public comparable {

```
    double x, y;
    public
    bool lte (const comparable & c)
    { //Vorbedingung: c ist vom Typ point c.
    if x < (point &) c.x ... ; // Type casting (Übung)
```

Nun kann Feld von Zeigern auf Point mit Quicksort sortiert werden.