Programmieren

Projektorintierte Programmierung II

Dynamische Instanzen

Um eine Instanz dynamisch zu erzeugenm verwenden wir „new“.
new ruft impliziet einen Konstruktor.
Bsp.:

```
Class c_bruch
{
     private:
     int zaehler;
     int nenner;

     public:
     c_bruch (int _zaehler, int _nenner),
     ...
};

void main (void)
{
     c_bruch * bruch = new c_bruch(1,3);
}
```

Um eine dynamische Instanz wieder zu löschen, verwenden wir „delete“.
delete ruft impliziet den Destruktor.

Dynamische Felder:
Bsp.:

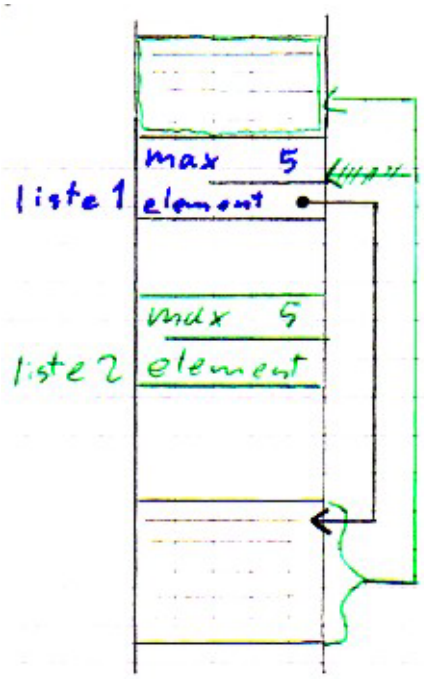
```
c_bruch *liste = new c_bruch[5];    Standart Konstr. nötig
liste[2].methode ...; // (liste+2) → meth.
delete[] liste;
```

Kopieren von (dynamischen) Instanzen:
Bsp.:

```
class c_liste
{
     private:
     int *element;
     int max;

     public:
     c_liste (int _max);
     int get (int index);
     void set (int index, int wert);
     ~c_liste();
};

c_liste::c_liste (int _max)
```


Text: M. Wilke
Zeichnungen: M. Löschinger

```
{
max = _max;
element = new int[max];
}

int c_liste ::get (int index)
{
      return
      ((index>=0) && (index<max))? element[index]
      :
      -1;
}

void c_liste::set(int iindex, int wert)
{
      if ((index>=0)&&((index<max))
            element[index] = wert;
}

c_liste::~c_liste()
{
      delete[] element;
}
void main (void)
{
      c_liste liste1(5);                  1)
      c_liste liste2 = liste1;            2)
      …     //c_liste liste2 (liste1) ;
}
```



(Bild 5.1)

Text: M. Wilke
Zeichnungen: M. Löschinger

Der Kopierkonstruktor:

Problem:
Der Standart-Kopierkonstruktor kopiert die Attribute bitweise. Dies ist immer dann problematisch, wenn Attribute dynamisch belegt werden.
Lösung:
Eine geeignetere Version des Kopierkonstruktors.

```
class c_liste
{
     private:
     int *element;
     int max;

     public:
     c_liste (int _max);
     c_liste (const c_liste &_liste);
     int get (int index);
     void set (int index, int wert);
     ~c_liste();
     c_liste & operator= (const c_liste &_liste);
};

c_liste::c_liste (const c_liste &_liste)
{
     max = _liste.max;
     element = new int[max];
     for (int i-0; i<max; i++)
     {
          element[i]= _liste.element[i];
     }
}
```

Es bleibt das Problem:

```
void main (void)
{
     c_liste liste1(5);
     c_liste liste2;
     liste2 = liste1;
```

Bei der Zuweisung besteht das gleiche Problem

```
}
```

Neudefinition des Zuweisungsoperators:

```
c_liste & c_liste::operator= (const c_liste &_liste)
{
     int *tmp = element;
     if (&_liste != this)          //?
     {
          element = new int[_liste.max];
          if (element == 0)
```


Text: M. Wilke
Zeichnungen: M. Löschinger

```
            {
                element=tmp;
                return *this;
            }
            max = _liste.max;
            for (int i=0; i<max; i++)
                element[i] = _liste.element[i];
            if (tmp != 0)
                delete[] tmp;
        }
        return *this;
}
```

Polymorphie:

Polymorphie bezeichnet die Möglichkeit, in einer Objekt-Variablen nicht nur eine Instanz der Klasse, um deren Typ die Variabele ist, zu speichern, sondern auch eine Instanz einer beliebigen, davon abgeleiteten Klasse.

Bsp.:

```
class c_element
{
        public:
        c_element &get_ref(void);
        virtuel   void ausgabe(void);
};

class c_int_element : public c_element
{
        private:
        int wert;

        public:
        c_int_element(void);
        c_int_element(int_wert);
        virtuel   void ausgabe(void);
};

class c_liste
{
        private:
        c_element *element;
        int max;

        public:
        c_liste(int _max);
        void ausgabe(void);
        c_element&get(int index);
};

c_element::get_ref(void)
{
```


Text: M. Wilke
Zeichnungen: M. Löschinger

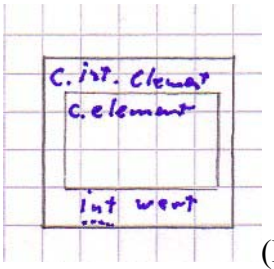
```
     return *this;
}

void c_element::ausgabe(void)
{
}

c_int_element::c_int_element(void):wert(0)
{
}

c_int_element::c_int_element(int wert):wert(_wert)
{
}

void c_int_element::ausgabe(void)
{
     printf („%d", wert);
}

c_liste::c_liste(int _max)
{
     max = _max ;
     element = new c_int_element[max] ;
}

void c_liste ::ausgabe(void)
{
     int i;
     for (i=0; i<max; i++)
          element[i].ausgabe();
}

c_element &c_liste::get(int index)
{
     if ((index>=0) && (index<max))
          return element[index].get_ref();
     else
          return 0;
}

void main(void)
{
     c_liste liste(5);
     liste ausgebe();
}
```

Test

```
I.)  void main(void)
     {
          c_element *e1 = new c_element;
```

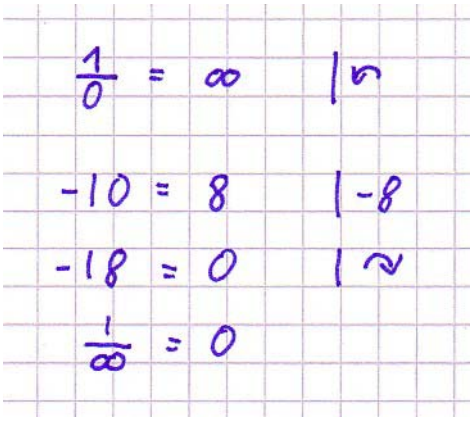

Text: M. Wilke
Zeichnungen: M. Löschinger

```
            c_element *e2 = new c_int_element;
            c_element *e3 = 0;

            e3=e1;
            e3=e2;
            e1=e3;
            e2=e3;
      }
II.)  void main(void)
      {
            c_element e1;
            c_int_element e2;
            c_element e3;

            e1=e2;
            e3=e2;      Polymorphie !
            e1=e3;
            e2=e3;      X
      }
```



(Bild5.2)

Bertrams mathematische Weisheiten:



Text: M. Wilke
Zeichnungen: M. Löschinger