Übergabe von Kommandoargumenten

Beim Programmaufruf können Kommandoargumente übergeben werden. Diese sind in der Funktion `main` über den C-Vektor `argv` zugänglich, auf dessen Komponenten jeweils die Startadressen der Kommandoargumente gespeichert sind. In der Komponente 0 ist die Startadresse des Programmnamens abgelegt, die Anzahl der Kommandoargumente einschließlich des Programmnamens steht in der ganzzahligen Variable `argc`.

Bsp.: Kommandoargumente ausgeben

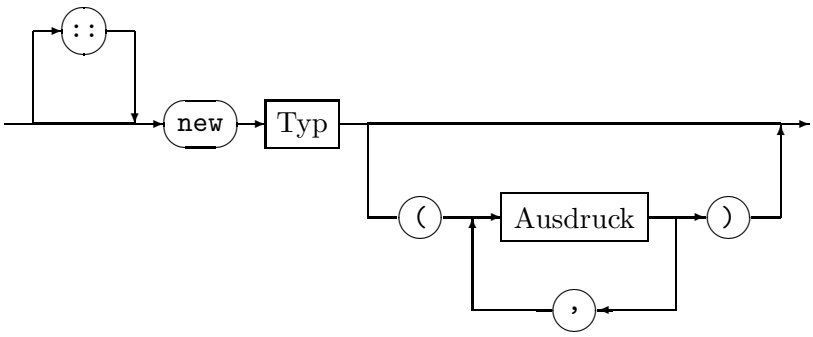
```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
   /* argc:  Zahl der Kommandoargumente
             (>=1 weil argv[0] auf Programmnamen zeigt)
      argv[argc] = 0    */
  int i;
  for (i=0; i<argc; ++i)
    cout << "Argument " << i << ": " << argv[i] << endl;
  return 0;
}
```

Dynamische Speicherreservierung (`<new>`)

`new`-Ausdruck[2]:



Der Operator `new` versucht ein Datenobjekt vom angegebenen Typ zu erzeugen und liefert im Erfolgsfall einen *Zeiger* darauf (bei C-Vektoren einen Zeiger auf die Komponente zum Index 0), andernfalls wird die Ausnahme `bad_alloc` signalisiert (Voreinstellung).

Die Lebensdauer der erzeugten Objekte hängt *nicht* von der Lebensdauer des Blocks ab, in dem der Ausdruck ausgewertet wurde.

Die optionale Ausdrucksliste kann wie bei Vereinbarungen zur Initialisierung verwendet werden, allerdings *nicht* bei der dynamischen Erzeugung von C-Vektoren.

Bei der dynamischen Erzeugung von mehrdimensionalen C-Feldern müssen alle Dimensionen evtl. mit Ausnahme der ersten konstant sein.

Der angegebenene Datentyp muß ggf. in Klammern gesetzt werden, wenn eine Verletzung der Vorrangregeln zu befürchten ist.

[2] ohne Placement-Syntax

`::new` benutzt den allgemeinen anstelle des typspezifischen `new`-Operator. Zum Löschen muß dann entsprechend `::delete` verwendet werden.

Bsp.: Dynamische Speicherreservierung einer skalaren Größe bzw. für $a \in \mathbb{R}^{m,n}$ und $x \in \mathbb{R}^n$

```
#include <new>

using namespace std;

int main()
{
  int *ip;
  ip = new int;
  *ip = 4;
     :
}
```

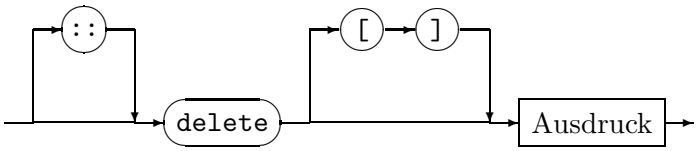
```
#include <new>

using namespace std;

int main()
{
  int m,n;
  double **a, *x;
  cout << "m n ? " ;
  cin >> m >> n ;
  x = new double[n];
  a = new double*[m];

  for (int i=0; i<m; ++i)
    a[i] = new double[n];
       :
}
```

`delete`-Ausdruck :



Der Operator `delete` gibt den Speicherplatz frei, auf den der Operandenausdruck zeigt. Dieser muß von einem früheren `new`-Aufruf stammen oder `0` sein. Letzteres bewirkt allerdings nichts.

`delete []` dient zur Freigabe des von C-Vektoren belegten Speicherplatzes und darf nur in diesem Zusammenhang verwendet werden.

Bsp.: Freigabe einer skalaren Größe bzw. von Vektoren/Matrizen

```
     :
  delete ip;
     :
```

```
          :
  delete [] x;
  for (int i=0; i<m; i++)
    delete [] a[i];
  delete [] a;
          :
```

Klassenobjekte werden gelegentlich dynamisch alloziert, vorzugehen wäre wie folgt.

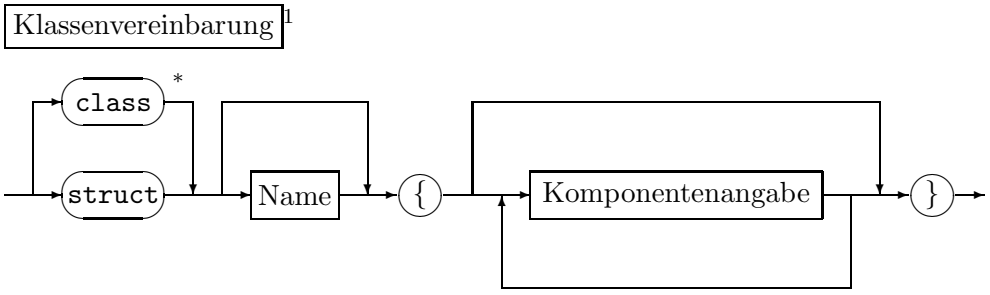
Bsp.: Dynamische Allokation und Deallokation eines STL-Vektors (unüblich!)

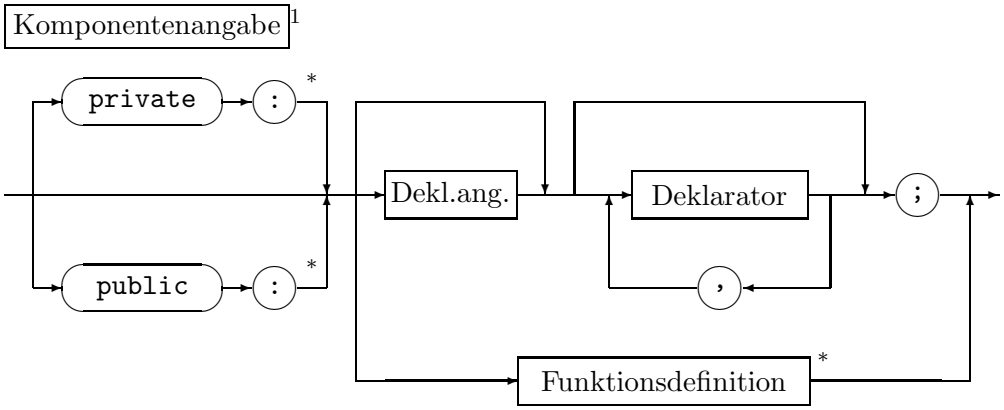
```
     :
int main()
{
  int n;

  vector<double> *xp;
  xp = new vector<double>(n);
  cout << "Laenge von *xp: " << (*xp).size() << endl;

  delete xp;
     :
}
```

Einfache Klassen





Klassen sind zusammengesetzte Datentypen, die von einem Benutzer oder in einer externen Bibliothek definiert sind und deren Komponenten über ihren Namen ansprechbar sind. Neben Datenkomponenten (Attribute) gehören auch Komponentenfunktionen (Methoden, andere Bez.: Elementfunktionen, Memberfunktionen) zu einer Klasse. Zusätzlich können in einer Klasse auch *befreundete* Funktionen vereinbart sein, sie zählen aber nicht zu den Komponentenfunktionen. Zugriffsattribute regeln, welche Funktionen auf die Klassenkomponenten zugreifen können.

Operationen:

`class` C `{...};`	Klassendefinition (außerhalb von Fkt.)
C c	Variablenvereinbarung (nach Voreinst. initial.)
C c`(`$init_1$`,`$init_2$`,...)`	Variablenvereinbarung (initialisiert)
C`()`	Temporärobjekt in Ausdrücken (nach Voreinst. initial.)
C`(`$init_1$`,`$init_2$`,...)`	Temporärobjekt in Ausdrücken (initialisiert)
c`.`$name$	Komponentenauswahl
c`.`f`(...)`	Aufruf einer Komponentenfunktion
cp`->`$name$	entspricht `(*`cp`).`$name$, sofern cp Zeiger auf C
cp`->`f`(...)`	entspricht `(*`cp`).`f`(...)`, sofern cp Zeiger auf C

Records

Klassen sind aus den bereits in C vorhandenen Records entstanden. Ein Record (Verbund) ist ein zusammengesetzter Datentyp, der nur aus Datenkomponenten besteht. Der Datentyp der Komponenten ist entweder wieder ein Record oder ein eingebauter Datentyp (einfacher Datentyp, C-Vektor, Zeiger bzw. geeignete Kombinationen). Der Zugriff auf die Komponenten ist nicht eingeschränkt. Records werden in C mit dem Schlüsselwort `struct` ohne Zugriffsattribute vereinbart.

*nur C++
[1]vereinfacht

Bsp.: Record für komplexe Zahlen

```
#include <iostream>
#include <new>
using namespace std;

struct Complex {double re,im;};     // Recorddefinition (ausserhalb von Fkt.)

Complex add(Complex w, Complex z)
{
  z.re += w.re; z.im += w.im;
  return z;
}

int main()
{
  Complex v={1.0,2.0}, *wp, z;
  wp = new Complex;
  cout << "Re(w) Im(w): ";
  cin >> (*wp).re >> (*wp).im;

  z=add(v,*wp);

  cout << "v = (" << v.re   << "," << v.im   << ")" << endl;
  cout << "w = (" << wp->re << "," << wp->im << ")" << endl;
  cout << "z = (" << z.re   << "," << z.im   << ")" << endl;

  return 0;
}
```

Unterschiede Klassen - Records

- Das Zugriffsattribut `private` erlaubt nur den Komponentenfunktionen und den mit `friend` innerhalb der Klasse vereinbarten Funktionen den Zugriff auf die so gekennzeichneten Klassenkomponenten.
  `public` gestattet allen Funktionen die Verwendung der Klassenkomponenten.
  Für `struct` ist `public` und für `class` `private` voreingestellt.

- Innerhalb von Klassen können Funktionen vereinbart werden. Funktionen *ohne* die Deklarationsangabe `friend` sind Komponentenfunktionen (member functions). Sie werden für Klassenvariable (Objekte) mit den Komponentenzugriffsoperatoren `.` bzw. `->` aufgerufen und haben auch Zugriff auf die `private`-Komponenten.

- Funktionen, die in einer Klasse als `friend` vereinbart werden, haben ebenfalls Zugriff auf die `private`-Komponenten von Objekten dieser Klasse. Sie gehören nicht zur Klasse und werden wie normale Funktionen aufgerufen.

- Konstruktoren sind Komponentenfunktionen, die denselben Namen wie ihre Klasse tragen. Für sie darf kein Ergebnistyp, auch nicht `void`, angegeben werden.
  Ihr Aufruf dient der Initialisierung der Klassenobjekte und erfolgt zu diesem Zeitpunkt.

- Destruktoren sind Komponentenfunktionen, deren Name aus Zeichen `~` und dem Klassennamen gebildet werden. Wie bei Konstruktoren darf kein Ergebnistyp angegeben werden. Ihr Aufruf dient der Freigabe der Klassenobjekte und erfolgt am Ende der Lebensdauer (Blockende/Programmende/delete).