Klassen und Objekte

Einige Begriffe der objektorientierten Programmierung

- Eine **Klasse** (**class**) ist eine Zusammenfassung von Variablen und Funktionen, die auf die Datenkomponenten angewandt werden, zu einer Einheit. Diese Funktionen heißen **Methoden** oder auch **Elementfunktionen** (**member functions**).

```
class Mensch
{
public:
        void DatenErfassen(char Name[20], bool Laune);
        void StimmungAusgeben();
private:
        char name[20];
        bool traurig;
        void weinen();
};
```

- Ein **Objekt** ist ein Abbild (eine **Instanz)** einer Klasse, d. h. eine Zuweisung von Speicherplatz für die Daten der Klasse. Ein Objekt hat also zu einer Klasse die gleiche Beziehung wie eine Variable zu einem Datentyp.

```
Mensch  ich;                    //Objekt ich der Klasse Mensch
```

Wenn die **Implementierung** der Klassenmethoden außerhalb der Klassendefinition erfolgt, dann geschieht dies über den Klassennamen gefolgt von zwei Doppelpunkten und dem Funktionsnamen

```
void Mensch::weinen()
{
        cout << ":'-( \n\n";
}
```

- **Zugriffsregler** für die Komponenten einer Klasse sind:

public	für Komponenten, auf die in *main()* und allen anderen Funktionen, die die zu gehörige Klasse enthalten, zugegriffen werden kann
private	für Komponenten, die nur den Komponenten der Klasse zugänglich sind
protected	für Komponenten, auf die nur die Komponenten der Klasse und die der abgeleiteten Klassen zugreifen können

Haupteigenschaften der objektorientierten Programmierung (OOP)

- **Kapselung** (**encapsulation**): Zusammenfassung der Datenstruktur und der darauf wirkenden Funktionen zu einem Ganzen und Abschirmen derselben durch Zugriffsregler von der Außenwelt.
- **Vererbung** (**inheritance**): Ableitung von neuen Klassen aus bereits vorhandenen (dient dem hierarchischen Aufbau der Klassen)

```
class Frau : public Mensch
{       public:         void StimmungAusgeben();
        private:        void flirten()
                        {   cout << ";-)\n\n";   }
};
```

- **Polymorphie** (**polymorphism**): z. B. Eigenschaft von Variablen, verschiedene Typen anzunehmen oder Eigenschaft von Operatoren oder Funktionen, sich auf verschiedene Klassen zu beziehen und unterschiedlich interpretiert zu werden

Ein Beispielprogramm

```
//mensch.cpp
#include <iostream.h>
#include <ctype.h>
#include <string.h>

class Mensch
{
public:
        void DatenErfassen(char Name[20], bool Laune);
        void StimmungAusgeben();
private:
        char name[20];
        bool traurig;
        void weinen();
};

void Mensch::weinen()
{
        cout << ":'-( \n\n";
}

void Mensch::DatenErfassen(char Name[20], bool Laune)
{
        strcpy(Name, name);
        traurig = Laune;
}

void Mensch::StimmungAusgeben()
{
        if (traurig) weinen();
}

int main()
{
  Mensch ich;
  char MeinName[20];
  char antwort;
  bool b;
  cout << "Wie heisst du? ";
  cin >> MeinName;
  cout << "Bist du traurig, " << MeinName <<"? (J / N) ";
  cin >> antwort;
  if (toupper(antwort) == 'J') b = true;
  else b = false;
  ich.DatenErfassen(MeinName, b);
  ich.StimmungAusgeben();
  return 0;
}
```

Aufgabe 1:
Erweitern Sie die Klasse *Mensch* um die private Methode *lachen* zur Ausgabe eines Emoticons, das das Lachen symbolisieren soll, und ergänzen Sie die Methode *StimmungAusgeben* durch den Aufruf der Methode *lachen*.

Aufgabe 2:
Erweitern Sie das obige Programm um die aus der Klasse *Mensch* abgeleitete Klasse *Frau*. Definieren Sie *Frau::StimmungAusgeben* so, dass im Fall der guten Laune sowohl gelacht als auch geflirtet wird.

Konstruktoren und Destruktoren

Konstruktoren (**constructors**) sind Funktionen, die zur Erstellung (Initialisierung) eines Objektes notwendig sind. Sie haben denselben Namen wie die Klasse des Objektes und haben keine Rückgabewerte (anders als bisher wird *void* aber weggelassen). Konstruktoren können benutzerdefiniert sein. Wenn sie es nicht sind, so werden sie automatisch erstellt (**default constructor**). In unserem obigen Beispielprogramm wird der Konstruktor *Mensch()* in dem Moment aufgerufen, wo Speicherplatz für das Objekt *ich* der Klasse *Mensch* reserviert wird.
Es kann zu einer Klasse mehrere Konstruktoren geben, die sich aber in der Anzahl oder den Typen der Parameter unterscheiden müssen.

Destruktoren (**destructors**) dienen der Speicherbereinigung. Pro Klasse kann es höchstens einen benutzerdefinierten Destruktor geben. Er wird durch eine vorangestellte Tilde ~ gekennzeichnet, z. B. *~ Mensch()*. Wird der Destruktor nicht explizit angegeben, so wird er vom Compiler automatisch erstellt.

Aufgabe 3:
Gehen Sie folgendes Programm[1] Schritt für Schritt durch und versehen Sie es mit Kommentaren. Testen Sie anschließend das Programm.

```
//classMatrix1.cpp
#include <iostream.h>
class Matrix
{
        public:
                Matrix(int Breite, int Hoehe);
                ~Matrix();
                int GetBreite() const { return itsBreite; }

                int GetHoehe() const { return itsHoehe; }

                int ** Feld;
        private:
                int itsBreite, itsHoehe;

};
```

[1] Nach einem Programm von Paul Müller (Jan 2006)

```
Matrix::Matrix(int Breite, int Hoehe)
{
        int i, j;
        itsBreite = Breite;
        itsHoehe = Hoehe;
        Feld = new int*[Breite];

        for (i = 0; i < Breite; i++)
        {
                Feld[i] = new int[Hoehe];
        }
        for (i = 0; i < Breite; i++)
        {
                for (j = 0; j < Hoehe; j++)
                {       Feld[i][j] = 0;         }
        }
}

Matrix::~Matrix()
{
        int i;
        for (i = 0; i < itsBreite; i++)
        {
                delete [] Feld[i];
        }
        delete [] Feld;
}

void matrixAddition (Matrix& Ma1, Matrix& Ma2, Matrix& SuMa);
//Matrizen werden an Funktion matrixAddition übergeben, "&" ist ein Muss!

int main ()
{
        Matrix A(3,5);
        Matrix B(3,5);
        Matrix S(3,5);

        A.Feld[0][0]=4;
        B.Feld[0][0]=12;

        matrixAddition(A, B, S);

        cout << S.Feld[0][0] << endl;
        return 0;
}
```

```
void matrixAddition(Matrix& Ma1, Matrix& Ma2, Matrix& SuMa)
{
        if(Ma1.GetBreite() == Ma2.GetBreite() && Ma1.GetHoehe() == Ma2.GetHoehe())
        {
                int i, j;
                for (i = 0; i < Ma1.GetBreite(); i++)
                {
                        for (j = 0; j < Ma1.GetHoehe(); j++)
                        {
                                SuMa.Feld[i][j] = Ma1.Feld[i][j] + Ma2.Feld[i][j];
                        }
                }
        }
        else
        {       cout << "Matrizen haben nicht die gleiche Dimension!!!";       }
}
```

Aufgabe 4:

a) Ergänzen Sie Konstruktor und Destruktur des obigen Programms durch Ausgabe von Text auf dem Bildschirm, wie z. B. „*Jetzt wurde der Konstruktor aufgerufen*“.
b) Ergänzen Sie das Programm durch eine Funktion zur Vielfachenbildung von Matrizen und rufen Sie diese an geeigneter Stelle auf.
c) Wandeln Sie das Programm so ab, dass der Typ der Matrizen über Tastatur eingegeben werden kann.