Objektorientierte Programmierung
(Winter 2010/2011)

# Kapitel 3: Objektorientierte Programmierung am Beispiel

- Klassen, Attribute, Methoden
- Zeichenketten in C++
- Zeiger, Verkettete Listen





## Inhalt







## Vorbemerkung

- Dieses Kapitel ist ein Experiment.
- Es gab Kritik, dass objektorientierte Programmierung (Klassen-Deklarationen u.s.w.) erst recht spät in der Vorlesung behandelt wurden.
- Im Sinne einer systematischen Einführung braucht man aber eigentlich den Stoff vorher.
- In diesem Kapitel "springen wir einfach mitten rein".
- Später gibt es alles nochmal systematisch.

  Also keine Panik, wenn Sie nicht alles 100% verstehen.





## Objekte, Klassen (1)

- Bisher waren die Dinge, die wir in Variablen gespeichert haben, hauptsächlich ganze Zahlen.

  Oder andere, einfache, atomare Datentypen.
- Programme sollen aber Aufgaben in der realen Welt unterstützen, und enthalten deswegen Abbilder bzw. Abstraktionen von Dingen in der realen Welt.

  Abstraktion: Man lässt Details weg, die für die jeweilige Aufgabe nicht relevant sind.
- Solche Dinge haben oft mehrere verschiedene Eigenschaften (Attribute).

# Objekte, Klassen (2)

- Z.B. plane ich gelegentlich ein Großfeuerwerk.

  Etwa zur letzten Langen Nacht der Wissenschaften. Ich habe als Hobby eine Ausbildung als Großfeuerwerker gemacht und habe Befähigungsschein und Erlaubnis nach dem Sprengstoffgesetz.

- Bei einem Feuerwerk werden Feuerwerksartikel abgebrannt (z.B. Vulkane, Fontänen, Feuerwerksbatterien, Feuerwerksbomben).

  Raketen sind bei Großfeuerwerken selten, weil man wegen der etwas unberechenbaren Flugbahn und der herabfallenden Stäbe einen großen Sicherheitsabstand braucht (200m). Feuerwerksbomben werden nach dem Prinzip einer Kanone aus Abschussrohren senkrecht in den Himmel geschossen, und zerplatzen dort in Leuchtsterne und andere Effekte. Der Effekt ist also ähnlich zu Raketen.





# Objekte, Klassen (3)

- Die für die Planung wichtigen Eigenschaften eines Feuerwerksartikels sind:
  - ◇ Bezeichnung
  - ◇ Preis
  - ◇ Brenndauer
- Z.B. gibt es den Artikel "Schweizer Vulkan Magic Light", der 11,10 € kostet (plus Mehrwertsteuer), und 80 Sekunden brennt.

  Erst Stroboskop-Effekt (weißer Bengalblinker), dann goldene Fontäne mit weißen Blinksternen darin, die sich bis zur maximalen Sprühhöhe von 8m langsam aufbaut.





# Objekte, Klassen (4)

- Eine Klasse ist eine Zusammenfassung von ähnlichen Objekten, die also insbesondere die gleichen Eigenschaften haben.

  Eine Klasse ist also ein Begriff, mit denen man über eine Menge von gleichartigen Dingen der realen Welt reden kann. Auch dies ist eine Form der Abstraktion.

- Man kann Klassen auch als Blaupausen/Schablonen zur Konstruktion von Objekten verstehen.
- Oder einfach als einen strukturierten Datentyp.

  Wobei bei Datentypen immer auch die Operationen dazugehören, mit denen man mit den Elementen des Datentyps arbeiten kann.





# Objekte, Klassen (5)

- Im Beispiel gibt es also eine Klasse `Artikel` (kurz für "Feuerwerksartikel") mit den drei genannten Eigenschaften (Attributen, "data members"):

```
class Artikel {
    const char *Bezeichnung; // Zeichenkette
    int Preis; // Preis in Cent ohne MwSt
    int Dauer; // Brenndauer in Sekunden
    ...
};
```

- Ein Objekt der Klasse `Artikel` enthält also gewissermaßen diese drei Variablen.

# Inhalt







# Zeichenketten in C++ (1)

- Der C++-Compiler legt Zeichenketten aus dem Programm, wie z.B.

  `"Hello, World!\n"`

  Zeichen für Zeichen im Speicher des erzeugten Programms ab, wobei das Ende durch ein spezielles Zeichen (Null-Byte) markiert wird:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | e | l | l | o | , | | W | o | r | l | d | ! | \n | \0 |





# Zeichenketten in C++ (2)

- Steht das erste Zeichen (`'H'`) z.B. bei Adresse 1000 im Speicher, so ist klar, dass
  - ◇ das zweite Zeichen (`'e'`) bei Adresse 1001 steht,
  - ◇ das dritte bei Adresse 1002, u.s.w.,
  - ◇ bis schließlich ein Null-Byte bei Adresse 1014 das Ende der Zeichenkette markiert.

    Natürlich darf das Null-Byte dann nicht innerhalb der Zeichenkette vorkommen. ASCII 0 ist aber auch kein druckbares Zeichen.
- Wenn man mit Zeichenketten arbeiten will (z.B. sie ausdrucken), reicht es also, die Speicheradresse des ersten Zeichens zu kennen.





# Zeichenketten in C++ (3)

- Eine Variable vom Typ “`char`” (character) enthält ein einzelnes Zeichen.
- Eine Variable vom Typ “`char *`” enthält die Adresse einer Speicherstelle (Variablen) vom Typ `char`.

  Das geht für beliebige Typen, z.B. wäre `int *` die Adresse eines Speicherbereiches, in dem ein Wert vom Typ `int` abgelegt ist. Da `int`-Variablen normalerweise 4 Byte (32 Bit) belegen, merkt man sich wieder nur die Startadresse (erstes Byte). Aufgrund des Typs `int` ist klar, dass die folgenden drei Byte auch dazu gehören. Wenn man `char *` für Strings benutzt, ergibt sich nicht sofort aus dem Typ, wie lang der Speicherbereich tatsächlich ist. Weil am Ende aber immer das Null-Byte steht, funktioniert es doch.

# Zeichenketten in C++ (4)

- Daher könnte man die Bezeichnung so deklarieren:

  `char *Bezeichnung;`

  (Adresse eines Zeichens, stellvertretend für die ganze dort gespeicherte Zeichenkette).

  Man kann übrigens auch "`char* Bezeichnung;`" schreiben (mit dem Leerzeichen hinter dem "`*`"), oder "`char * Bezeichnung;`" (in der Praxis unüblich), oder sogar "`char*Bezeichnung;`" (häßlich). Die meisten C++-Programmierer verwenden die im Beispiel gezeigte Schreibweise, weil sie auch funktioniert, wenn man mehrere Variablen zusammen deklariert (man muß den "`*`" dann bei jeder Variablen wiederholen). Aber "`char* Bezeichnung;`" macht auch Sinn, denn der Typ der Variablen "`Bezeichnung`" ist "`char*`" (Adresse einer "`char`"-Speicherstelle, auch "Zeiger auf `char`" genannt, englisch "Pointer").





# Zeichenketten in C++ (5)

- Wenn man das so macht, bekommt man allerdings eine Warnung, wenn man eine Zeichenkettenkonstante zuweisen will.
- Das Problem ist, dass man mit der Adresse einer Speicherstelle grundsätzlich auch die Möglichkeit hätte, die Speicherstelle zu verändern.
- Bei Zeichenkettenkonstanten würde das aber nicht gehen, oder man sollte es jedenfalls lassen.

  Der Compiler könnte die Zeichenkettenkonstante in einem Speicherbereich unterbringen, der vor Änderungsoperationen geschützt ist. Er könnte auch mehrere gleiche Konstanten nur einmal speichern.





# Zeichenketten in C++ (6)

- Wenn man auf die Möglichkeit verzichtet, über die Variable ändernd auf den referenzierten Speicherbereich zuzugreifen, fügt man "`const`" so hinzu:

  `const char *Bezeichnung;`

- Das kann der Compiler dann auch kontrollieren:
  - ◇ der Versuch einer solchen Änderung wird mit einer Fehlermeldung quittiert,
  - ◇ und man kann den Inhalt dieser Variablen nur an andere Variablen weitergeben, für die der gleiche Verzicht gilt.





# Zeichenketten in C++ (7)

- Zu kompliziert? Definieren Sie sich einen eigenen Typ, der dies alles enthält:

  `typedef const char *str_t;`

  Schreiben Sie das relativ weit oben in Ihr Programm, außerhalb der Klassendeklaration. Dann können Sie den Typ `str_t` im ganzen Rest Ihres Programms verwenden.

- Nachdem Sie "`str_t`" als "String Typ" deklariert haben, können Sie die Details (vorläufig) wieder vergessen, und ihn wie einen eingebauten Typ verwenden:

  `str_t Bezeichnung;`

# Zeichenketten in C++ (8)

- Diese Art, Zeichenketten über die Adresse des ersten Zeichens zu repräsentieren, ist von C geerbt.

  Minimalistische Sprache, mit ziemlicher Nähe zur Hardware.

- Wenn Effizienz (geringe Laufzeit, wenig Speicher) im Vordergrund steht, ist diese Lösung kaum zu schlagen.

  Sie hat meiner Ansicht nach auch den Vorteil, dass man bis zur Maschinenebene herunter noch recht leicht verstehen kann, was genau vorgeht.

- C++ hat aber auch eine Alternative, nämlich die Bibliotheksklasse `string` (siehe nächste Folien).





# Die Klasse `string` (1)

- Der Vorteil der Bibliotheksklasse `string` ist, dass sie viele Funktionen zur Zeichenkettenverarbeitung mitbringt, auch mit natürlicher Operator-Notation.

  Z.B. kann man den Operator "+" für die String-Konkatenation verwenden (aneinanderhängen, zusammenfügen von Zeichenketten).

- Zwar gibt es auch für klassische C-Strings viele nützliche Bibliotheksfunktionen, aber der wesentliche Unterschied ist, dass `string` die Speicherverwaltung automatisch macht.

  Wenn man auf die klassische Art Strings aneinanderhängen will, muss man sich erst einen ausreichend grossen Speicherbereich besorgen.





# Die Klasse `string` (2)

- Die automatische Speicherverwaltung der Klasse `string` ist
  - ◇ ein Vorteil, weil man als Programmierer darüber nicht mehr nachdenken muss,
  - ◇ aber auch ein Nachteil, weil man es manuell vielleicht etwas effizienter hinbekommen hätte.
- Im Beispiel gibt es nur konstante Strings, da wirkt sich der Vorteil nicht aus.

  Der Nachteil schon: Die Zeichenketten werden unnötig kopiert. Aber der Speicher ist groß und das Programm läuft so schnell, dass die kleine Verzögerung absolut nicht spürbar ist. Außerdem funktioniert die Klasse später dann auch mit eingelesenen Strings (s.u.).





# Die Klasse `string` (3)

- Wenn Sie im Beispiel gerne `string` verwenden wollen, schreiben Sie oben bei der anderen Include-Anweisung (egal, ob davor oder dahinter):

```
#include <string>
```

- Sie können Attribut und Parameter dann mit dem Typ `string` deklarieren, z.B.

```
string Bezeichnung;
```

- Es spricht für die Sprache C++, dass ein so grundlegender Datentyp in der Sprache selbst definiert werden konnte.

# Inhalt







# Konstruktoren (1)

- Variablen müssen bekanntlich initialisiert werden, das gilt auch für die Attribute eines Objektes.
- Es ist üblich, dafür ein Programmstück zu schreiben, das man Konstruktor nennt.

  Es kann auch mehrere alternative Konstruktoren geben.
- C++ garantiert, dass der Konstruktor einer Klasse aufgerufen wird, immer wenn ein neues Objekt der Klasse erzeugt wird.

  Deswegen heißt er Konstruktor: Er konstruiert Objekte. (Genauer formt er Objekte aus dem Speicher, der ihm von C++ zur Verfügung gestellt wird — er leistet also nur einen Teil der Objekterzeugung.)





# Konstruktoren (2)

- Hat man einen Konstruktor definiert, der die Attribute initialisiert, so kann man später bei der Objekterzeugung nie mehr die Initialisierung vergessen.
- Konstruktoren können Parameter (Eingabewerte, "Argumente") haben.
- Im Beispiel möchte man, dass jedes `Artikel`-Objekt mit sinnvollen Werten für Bezeichnung, Preis, Dauer belegt wird.
- Diese Werte kann sich der Konstruktor natürlich nicht ausdenken, sondern braucht sie als Parameter.





# Konstruktoren (3)

- Beispiel für Klasse mit Konstruktor:

```
class Artikel {
    str_t Bezeichnung;
    ...
  public:
    Artikel(str_t bez, int preis, int dauer)
    {
        Bezeichnung = bez;
        Preis = preis;
        Dauer = dauer;
    }
    ...
};
```

# Konstruktoren (4)

- Ein Konstruktor besteht also aus:
  - ◇ dem Namen der Klasse,
  - ◇ in runden Klammern `(...)` den Parametern des Konstruktors, für die man beim Aufruf Eingabewerte angeben muss,

    Die Parameter werden wie Variablen deklariert, nur wird die Deklaration nicht jeweils mit "`;`" abgeschlossen, sondern die einzelnen Deklarationen werden durch "`,`" getrennt. Hat ein Konstruktor keine Parameter, schreibt man einfach "`()`".
  - ◇ und in geschweiften Klammern `{...}` dem Programmcode ("Rumpf") des Konstruktors.
- Das Schlüsselwort `public` wird später besprochen.





# Konstruktoren (5)

- Man beachte:
  - ◇ Nach den geschweiften Klammern für den Rumpf des Konstruktors (oder einer Funktion) schreibt man kein Semikolon,

    Ebenso bei der Zusammenfassung mehrerer Anweisungen durch geschweifte Klammern (es ist alles ein "Block"). Hier würde es aber keine Fehlermeldung geben, wenn man doch ein Semikolon schreibt (man hat dann eine leere Deklaration eingefügt).
  - ◇ während nach den geschweiften Klammern für die Klassendeklaration unbedingt ein Semikolon stehen muss.

    Das ist kein Block, sondern eine Deklaration.





# Konstruktoren (6)

- Im Konstruktor kann man sowohl auf die Parameter zugreifen, als auch auf die Attribute der Klasse.

  Die Parameter sind Variablen, die beim Aufruf mit den übergebenen Werten automatisch initialisiert sind. Wenn der Konstruktor fertig abgearbeitet ist, werden sie wieder gelöscht (der Speicherplatz wird freigegeben). Die Attribute sind Variablen, die in dem Objekt gespeichert werden, und so lange existieren wie das Objekt.
- Der Beispielkonstruktor kopiert die Eingabewerte von den Parametern in die Attribute des Objektes.

  Bei der Zeichenkette wird nur die Adresse kopiert. Es ist dann wichtig, dass sich der Inhalt des Speicherbereiches nicht ändert. Bei konstanten Strings kann das nicht passieren. Ansonsten (Variablen mit wechselnden Eingabezeichen) wähle man doch besser den Typ `string`.





# Konstruktoren (7)

- Natürlich kann der Konstruktor mehr machen, als nur Eingabewerte 1:1 in Attribute zu kopieren:
  - ◇ Der Konstruktor könnte die Eingabewerte prüfen, z.B. negative Preise/Brenndauern zurückweisen.

    Er könnte eine Meldung ausgeben, und das Programm durch die Anweisung `exit(1);` beenden. Professioneller: `assert(preis>=0);` (Zusicherung, dazu `#include <cassert>`), siehe späteres Kapitel. Allgemein gelten häufig "Integritätsbedingungen" ("Invarianten") für eine Klasse. Der Konstruktor muss natürlich überwachen, dass sie bei der Objekt-Erzeugung nicht schon verletzt sind.
  - ◇ Oft gibt es auch Attribute, die einfach mit 0 initialisiert werden, oder mit aus den Parametern berechneten (abgeleiteten) Werten.

# Verschattung (1)

- Man kann die Attribute nennen, wie man will. Ich verwende einen Großbuchstaben am Anfang, aber dass ist nicht allgemeiner Standard.
- Wenn man die Attribute auch klein schreibt, hat man das Problem, dass sie häufig genauso heißen, wie die Parameter des Konstruktors.

  Wenn man die Parameter nicht künstlich umbenennt, und z.B. nur einbuchstabige Bezeichner verwendet.
- Wenn Attribute und Parameter gleich heißen, weiß der Compiler nicht, auf was man sich beziehen will.

  Es sind ja unterschiedliche Speicherstellen.





# Verschattung (2)

- Die Lösung ist, dass “gewinnt”, was näher deklariert ist, in diesem Fall also der Parameter. Das Attribut ist “verschattet” (nicht direkt zugreifbar).
- Eine Anweisung wie

  ```
  preis = preis;
  ```

  ist ja auch offensichtlich sinnlos: Der Wert wird von einer Variablen in die gleiche Variable kopiert.
- Wenn man Glück hat, bekommt man eine Warnung, ansonsten tut der Compiler genau dies.

  Rechner denken sich nichts, sondern tun stupide, was man ihnen sagt.





# Verschattung (3)

- Wenn Sie aber Attribute und Parameter so nennen wollen, gibt es eine Lösung. Schreiben Sie:

  ```
  this->preis = preis;
  ```
- Das Schlüsselwort “`this`” bedeutet immer das aktuelle Objekt, für das der Konstruktor also gerade aufgerufen wurde.

  Genauer: Seine Speicheradresse. Der Typ ist hier `Artikel *`.
- Mit dem Operator “`->`” kann man auf das rechts benannte Attribut von dem Objekt (Adresse) auf der linken Seite zugreifen.





# Verschattung (4)

- Deswegen ist die Situation hier klar:
  - ◇ Bei `this->preis` kann `preis` nur das Attribut sein.
  - ◇ Rechts vom Gleichheitszeichen könnte es Attribut oder Parameter sein, aber da der Parameter näher deklariert ist, gewinnt er.
- Damit geschieht also das Gewünschte:

  ```
  this->preis = preis;
  ```

  kopiert den Wert vom Parameter in das Attribut.

  Gewöhnen Sie sich einen Programmierstil an und bleiben Sie dabei (dann brauchen Sie über solche Details nicht mehr nachzudenken). In einem Team sollte man natürlich einen gemeinsamen Stil verwenden.

# Erzeugen von Objekten (1)

- Klassen sind Datentypen, also können Objekte der Klasse im Prinzip wie Variablen deklariert werden.
- Wenn wir einen Konstruktor ohne Parameter definiert hätten, wäre Folgendes möglich:

  ```
  Artikel vulkan;
  ```

  Dann ist `vulkan` ein Objekt der Klasse `Artikel`.
- Weil der Konstrukter aber drei Parameter hat (Bezeichnung, Preis, Dauer), muß man Werte dafür angeben, z.B.

  ```
  Artikel vulkan("Magic Light", 1100, 80);
  ```





# Erzeugen von Objekten (2)

- Die bei der Objektdeklaration angegebenen Werte werden entsprechend ihrer Reihenfolge in der Liste den Parametern zugewiesen.
- Weil im Konstrukter `preis` als zweiter Parameter angegeben wurde, wird der zweite Wert (`1100`) diesem Parameter zugewiesen.

  Wenn man die Werte für Preis und Dauer vertauscht, bekommt man keine Fehlermeldung vom Compiler: Beides sind `int`-Werte und können `int`-Parametern zugewiesen werden. Im Konstruktor könnte man testen, dass Brenndauern nicht unrealistisch lang sind, über 5min sind sehr selten (hier hat man nur die Wahl zwischen zwei Übeln, dies wäre ja doch eine Einschränkung). Wenn man die Zeichenkette an falscher Position übergibt, liefert der Compiler dagegen einen Typfehler.





# Erzeugen von Objekten (3)

- Jetzt soll die Klassendeklaration und die Erzeugung von Objekten mit einem vollständigen Programm ausprobiert werden.
- Da es noch nicht viel tut, bauen wir im Konstruktor eine Testausgabe ein.

  Nachdem man probiert hat, dass alles wie erwartet funktioniert, kann man sie wieder löschen.
- Komplettes Programm, Version 0.1 (Teil 1):

  ```
  #include <iostream>
  using namespace std;
  typedef const char *str_t;
  ```





# Erzeugen von Objekten (4)

- Komplettes Programm, Version 0.1 (Teil 2):

```
class Artikel {
    str_t Bezeichnung;
    int Preis; // Preis in Cent ohne MwSt
    int Dauer; // Brenndauer in Sekunden
  public:
    Artikel(str_t bez, int preis, int dauer)
    {   // Testausgabe, spaeter entfernen:
        cout << "Konstruktor: " << bez << "\n";
        Bezeichnung = bez;
        Preis = preis;
        Dauer = dauer;
    }
};
```

# Erzeugen von Objekten (5)

- Komplettes Programm, Version 0.1 (Teil 3):

```
int main()
{
    cout << "Feuerwerk-Planungssoftware 0.1\n";

    Artikel vulkan("V. Magic Light", 1100, 80);
    Artikel goldbatt("CB Flower Wall", 6200, 16);

    cout << "(Fortsetzung folgt)\n";
    return 0;
};
```





# Inhalt







# Zugriffsschutz (1)

- Bisher kann man mit den `Artikel`-Objekten noch nichts anfangen, da es nach der Erzeugung keine Möglichkeit gibt, auf die Attribute zuzugreifen.
- Die Attribute sind nämlich "privat", und nur für Programmcode innerhalb der Klasse zugreifbar.
- Dort gibt es aber bisher nur den Konstruktor.
- Was privat und was öffentlich ist, bestimmen die Schlüsselworte "`private:`" und "`public:`". Schreibt man nichts, ist "`private:`" die Voreinstellung.





# Zugriffsschutz (2)

- Wäre der Konstruktor nicht explizit als "`public:`" markiert, wäre er nicht von außen zugreifbar.

  Wir hätten dann also nicht im Hauptprogramm (`main`) Objekte der Klasse anlegen können. Oft bieten Klassen mit einem privaten Konstruktor eine andere Funktion an, um Objekte zu erzeugen. Es gibt auch Klassen, bei denen es gar keinen Sinn macht, dass Objekte erzeugt werden (wenn die Klasse nur eine Sammlung von Funktionen sein soll).

- Die Angabe "`public:`" oder "`private:`" gilt für alle folgenden Attribute oder Funktionen der Klasse, bis wieder eins dieser Schlüsselworte verwendet wird.

  Man braucht sie also nicht bei jedem einzelnen Attribut oder jeder einzelnen Funktion zu wiederholen.

# Zugriffsschutz (3)

- Ein Möglichkeit wäre nun, “`public:`” ganz vorne zu schreiben, dann ist alles von außen zugreifbar:

```
class Artikel {
  public:
    str_t Bezeichnung;
    int Preis; // Preis in Cent ohne MwSt
    int Dauer; // Brenndauer in Sekunden
    Artikel(str_t bez, int preis, int dauer)
    {
        Bezeichnung = bez;
        Preis = preis;
        Dauer = dauer;
    }
};
```





# Zugriffsschutz (4)

- Wenn $X$ ein Objekt der Klasse $C$ ist, und $A$ ein Attribut von $C$, kann man mit der Schreibweise

  $X$.$A$

  auf das Attribut $A$ im Objekt $X$ zugreifen.

  Vorausgesetzt natürlich, der Zugriffsschutz erlaubt es, d.h. dieser Ausdruck steht entweder in der Klasse $C$ selbst, oder $A$ ist als “`public:`” deklariert (diese Regeln werden später noch etwas erweitert).

- Ist $X$ dagegen die Adresse eines Objektes der Klasse $C$, hat also den Typ $C$`*` (“Zeiger auf $C$”), so schreibt man (wie oben schon bei `this`)

  $X$`->`$A$





# Zugriffsschutz (5)

- Wenn man die Attribute als “öffentlich” deklariert hat, kann man im Hauptprogramm z.B. schreiben:

```
cout << "Preis des Vulkans: " << vulkan.Preis;
```

- Man kann aber auch schreiben:

```
vulkan.Preis = -100;
```

  (Attribute funktionieren ja wie Variablen).

- Es wäre dann sinnlos, im Konstruktor die Parameter auf sinnvolle Werte zu prüfen, wenn die Attribute später doch jederzeit ohne Eingriffsmöglichkeit geändert werden könnten.





# Zugriffsschutz (6)

- Deswegen ist es üblich, Attribute immer `private:` zu deklarieren, und Zugriffe nur über Funktionen (“Methoden der Klasse”) zu erlauben.
- Der Programmcode dieser Funktionen kann dann Integritätsbedingungen (Invarianten) überwachen.
- Es hängt von der Anwendung ab, auf welche Attribute überhaupt ändernd zugegriffen werden muss.

  Manche Attribute werden im Konstruktor initialisiert und behalten dann den gleichen Wert für den Rest des Programmlaufs. In diesem Fall bietet man zwar eine Lesefunktion an (zum Abfragen des Wertes), aber keine zum Schreiben (Ändern) des Wertes. Eine Variable würde dagegen immer beide Möglichkeiten bieten.

# Methoden (1)

- Es ist ein wesentliches Kennzeichen der objektorientierten Programmierung, dass man Daten zusammen mit Funktionen zum Zugriff auf die Daten "kapselt", d.h.
  - ◇ beides ist zusammen definiert, und

    Dass man Daten immer im Zusammenhang mit Funktionen zu ihrer Verarbeitung sehen muss, war nicht neu: Dies ist z.B. ein Kernpunkt der Theorie abstrakter Datentypen.
  - ◇ man kann auf die Daten nur über die in der Klasse definierten Funktionen zugreifen.

    Dazu müssen die Daten, also die Attribute, natürlich "`private:`" deklariert sein.





# Methoden (2)

- In einer Klasse definierte Funktionen werden in der objektorientierten Programmierung normalerweise "Methoden" genannt.

  Solche Dinge führten zu der Kritik, dass die objektorientierte Programmierung nur neue Namen für existierende Konzepte einführte.
- Funktionen kennen wir von der Funktion `main`, dem Hauptprogramm.
- Im Unterschied dazu haben in einer Klasse definierte Funktionen (Methoden) Zugriff auf die Attribute des aktuellen Objektes.





# Methoden (3)

- Beginnen wir mit einer Methode, die die Bezeichnung eines Artikels liefert:

```
class Artikel {
        str_t Bezeichnung;
        int Preis; // Preis in Cent ohne MwSt
        int Dauer; // Brenndauer in Sekunden
    public:
        // Konstruktor (s.o.):
        Artikel(str_t bez, int preis, int dauer)
            { ... }
        // Lesezugriff auf Attribut Bezeichnung:
        str_t bezeichnung() const
            { return Bezeichnung; }
```





# Methoden (4)

- Eine Methodendefinition besteht aus
  - ◇ dem Kopf (mit Ergebnistyp, Methodenname, Parameterliste, ggf. `const`), und
  - ◇ dem Rumpf (ein Block, Folge von Anweisungen).
- Im Beispiel
  - ◇ heißt die Methode "`bezeichnung`",
  - ◇ liefert eine Zeichenkette (Wert von Typ "`str_t`"),
  - ◇ hat keine Parameter (Eingabewerte),
  - ◇ ändert den Objektzustand nicht (`const`), und
  - ◇ der Rumpf besteht nur aus der `return`-Anweisung.

# Methoden (5)

- Die `return`-Anweisung beendet die Ausführung der Methode und definiert den Ergebniswert.
- Der Ergebniswert wird für den Aufrufer kopiert, der Aufrufer bekommt auf diese Weise nur lesenden Zugriff auf das Attribut (und nicht die Variable selbst).
- Z.B. kann man im Hauptprogramm jetzt schreiben:

```
cout << vulkan.bezeichnung() << "\n";
```

- Zuweisungen werden dagegen zurückgewiesen:

```
vulkan.bezeichnung() = "Ueberraschungsvulkan";
```

```
fw.cpp:32: error: lvalue required as left operand of assignment
```





# Methoden (6)

- Man kann genauso für die beiden anderen Attribute Zugriffsfunktionen definieren:

```
class Artikel {
    ...
    // Preis des Artikels ohne MwSt (netto):
    int preis() const { return Preis; }

    // Brenndauer des Artikels in Sekunden:
    int dauer() const { return Dauer; }
};
```





# Methoden (7)

- Üblich ist auch, die Lesefunktion für das Attribut $A$ "`get_`$A$" nennen.

  Falls man dem Stil folgt, die Attributnamen mit Kleinbuchstaben zu beginnen (so wie für Variablennamen üblich), kann man die Zugriffsfunktionen nicht genauso nennen. Wenn das Attribut z.B. "`preis`" heisst, und man versucht, die Methode auch "`preis`" zu nennen, bekommt man eine Fehlermeldung ("conflicts with previous declaration"). Man kann den gleichen Namen zwar in verschiedenen Gültigkeitsbereichen verwenden (als Parameter im Konstruktor und als Attribut in der Klasse), aber hier wäre der gleiche Name für zwei verschiedene Dinge, beide direkt in der Klasse, verwendet. Es wäre für den Compiler dann nicht eindeutig, was gemeint ist, wenn der Name später im Programm auftaucht. Deswegen gibt dies gleich bei der Deklaration einen Fehler. Bei diesem Stil wäre der übliche Name für die Methode dann "`get_preis`" (das deutsch-englische Mischmasch ist nicht hübsch, richtig lösen kann man das nur durch "alles Englisch").





# Methoden (8)

- Wenn man in der Anwendung ein Attribut $A$ ändern muss, führt man dafür entsprechend eine Methode `set_`$A$ ein, z.B.:

```
void set_preis(int preis) { Preis = preis; }
```

- Diese Methode hat einen Parameter (für den neuen Wert), und keinen Ergebniswert, ausgedrückt mit dem speziellen Typ "`void`" (leer). Ist ist nicht mit `const` deklariert, da sie ein Attribut ändert.

  In er Methode sollte man eigentlich wieder prüfen, dass der Preis nicht negativ ist.

# Methoden (9)

- Neben den obigen Standard-Lese- und ggf. Schreib-Methoden kann man noch beliebige weitere Methoden einführen, die für die Anwendung wichtig sind.
- Z.B. ist der Preis mit 19% MwSt (Brutto-Preis) ein abgeleitetes Attribut:

```
int brutto_preis() const
    { return ceil(Preis * 1.19); }
```

Damit die Funktion `ceil` (aufrunden, "ceiling") definiert ist, muß man oben `#include <cmath>` schreiben. Damit erhält man Zugriff auf die "C-Mathematik-Bibliothek". in der es auch Funktionen wie `sqrt`, `sin`, `exp`, `log`, `log10`, etc. gibt. Die Funktion `ceil` liefert einen `double` Gleitkomma-Wert, der hier automatisch durch Abschneiden der Nachkommastellen in eine ganze Zahl umgewandelt wird.





# Methoden (10)

- Es ist ein wichtiges Prinzip der Objektorientierung (und auch schon der Modularen Programmierung und von abstrakten Datentypen), dass
  - ◇ man das Interface (Schnittstelle, von außen zugreifbare Funktionen) möglichst konstant hält,
  - ◇ während man die Implementierung (wie die Methoden intern programmiert sind) ändern kann.

    Wenn die Berechnung des Bruttopreises nicht so einfach wäre, wäre es eine Alternative, ein zusätzliches Attribut für den Bruttopreis einzuführen, und ihn nur einmal (im Konstruktor), ggf. auch in `set_preis`, zu berechnen. Von außen wären diese beiden Implementierungen ununterscheidbar.





# Deklarations-Reihenfolge (1)

- Normalerweise müssen Variablen und Funktionen deklariert sein, bevor man sie verwenden kann.

  Der Compiler liest das Programm von oben nach unten und gibt eine Fehlermeldung aus, wenn z.B. ein Variablenname vorkommt, den er noch nicht kennt. Deklarationen machen ihm den Namen bekannt.
- Bei Klassendeklarationen ist diese Regel dagegen etwas abgeschwächt, weil sie (mindestens konzeptuell) in zwei Phasen verarbeitet werden:
  - ◇ Zuerst die Deklarationen der Attribute, Methoden (Methodenkopf), u.s.w.,
  - ◇ danach die Methodenrümpfe.





# Deklarations-Reihenfolge (2)

- Konkret kann man in einem Methodenrumpf auf ein Attribut zugreifen, das erst weiter unten deklariert wird (in der gleichen Klassendeklaration).
- Damit Anwender der Klasse nur den für Sie relevanten Teil der Deklaration lesen müssen, schreibt man oft den öffentlichen Teil (Schnittstelle) zuerst, und danach den privaten Teil (Implementierung).

  Später wird erläutert, wie man die Methodenrümpfe getrennt von der Klassendeklaration aufschreiben kann (zumindest teilweise). Sie sind auch Teil der Implementierung.
- Das ist aber eine Geschmacksfrage.

# Beispiel (1)

- Vollständige Klassendeklaration (Teil 1):

```
class Artikel {
  public:

    // Konstruktor:
    Artikel(str_t bez, int preis, int dauer)
    {
        Bezeichnung = bez;
        Preis = preis;
        Dauer = dauer;
    }

    // Lesezugriff auf Attribut Bezeichnung:
    str_t bezeichnung() const
        { return Bezeichnung; }
```





# Beispiel (2)

- Vollständige Klassendeklaration (Teil 2):

```
// Preis des Artikels ohne MwSt (netto):
int preis() const { return Preis; }

// Preis des Artikels mit MwSt (brutto):
int brutto_preis() const
    { return ceil(Preis * 1.19) + 0.1; }

// Preis aktualisieren:
void set_preis(int preis) { Preis = preis; }

// Brenndauer des Artikels in Sekunden:
int dauer() const { return Dauer; }
```





# Beispiel (3)

- Vollständige Klassendeklaration (Teil 3):

```
    private:
      str_t Bezeichnung;
      int Preis; // Preis in Cent ohne MwSt
      int Dauer; // Brenndauer in Sekunden
};
```

- Wollen Sie die Objekte mit << druckbar machen, schreiben Sie ausserhalb der Klassendeklaration:

```
ostream& operator<<(ostream& s, const Artikel& A)
    { return s << A.bezeichnung(); }
```

Hier wird nur die Bezeichnung ausgegeben, mehr zur Übung.





# Inhalt

# Fortsetzung des Beispiels (1)

- Der Abbrennplan für ein Feuerwerk legt fest, welche Artikel zu welchem Zeitpunkt gezündet werden sollen (ggf. auch mehrere gleichzeitig).

  Damit wird insbesondere auch die Reihenfolge der Artikel festgelegt.
- Der Abbrennplan ist eine Folge von Punkten.
- Zu jedem Punkt (Zündung) gibt es folgende Daten:
  - ◇ Laufende Nummer im Plan.
  - ◇ Zeitpunkt der Zündung.
  - ◇ Artikel
  - ◇ Menge (falls mehrere Exemplare des Artikels).





# Fortsetzung des Beispiels (2)

- Die laufende Nummer im Plan ist oft die Kanalnummer für eine elektrische Zündanlage.

  Großfeuerwerke werden heute fast nur noch elektrisch gezündet.
- Der Zeitpunkt der Zündung wird hier in Sekunden seit dem Start des Feuerwerks gespeichert.

  In der Realität würde man für bestimmte Effekte schon mindestens eine Auflösung von Zehntelsekunden benötigen.
- Nicht selten werden mehrere Exemplare eines Artikels gleichzeitig gezündet.

  Z.B. nicht einen Vulkan, sondern $\geq 3$ in einer Linie parallel zum Publikum. Zur Vereinfachung behandeln wir nicht den Fall gleichzeitiger Zündung verschiedener Artikel.





# Fortsetzung des Beispiels (3)

- Der Abbrennplan für ein Mini-Feuerwerk könnte also z.B. so aussehen:

| Nr. | Zeit | Artikel | Stück |
|---|---|---|---|
| 1 | 0:00 | Vulkan Magic Light | 3 |
| 2 | 0:10 | Feuertopf Silberweide | 1 |
| 3 | 0:12 | Feuertopf Silberweide | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ |

Real wären etwa 40–80 Punkte bei einem mittleren Feuerwerk ohne Musik. Bei großen musiksynchronen Feuerwerken könnten es auch Hunderte von Punkten sein. Eine weitere Information, die hier zur Vereinfachung weggelassen wurde, ist die Position, an der ein Artikel abgebrannt wird: Z.B. der erste Feuertopf links, der zweite rechts.





# Fortsetzung des Beispiels (4)

- Nennen wir die Klasse für einen Punkt im Abbrennplan “`Zuendung`”.

  Natürlich sollte man möglichst aussagekräftige Namen wählen, die auch für andere Personen verständlich sind (vielleicht arbeiten später weitere Programmierer in diesem Projekt mit). Ich weiss nicht, ob mir das hier gelungen ist. Übrigens ist auch “`Artikel`” mehrdeutig: Gemeint ist eigentlich “Artikeltyp”, von dem es mehrere Exemplare geben kann.
- Neu ist nun, dass eine Beziehung zwischen Objekten der Klasse `Zuendung` und Objekten der Klasse `Artikel` dargestellt werden muss.

  Es ist eine ganz wesentliche Information, welcher Artikel(-typ) zu einem bestimmten Zeitpunkt gezündet werden soll.

# Fortsetzung des Beispiels (5)

- Es muss möglich sein, dass verschiedene `Zuendung`-Objekte auf das gleiche `Artikel`-Objekt verweisen.

  Verschiedene Exemplare des gleichen Artikels können zu unterschiedlichen Zeitpunkten gezündet werden.

- Man kann daher in den `Zuendung`-Objekten nicht ein `Artikel`-Objekt abspeichern.

  Allgemein wäre es möglich, dass ein Attribut selbst ein Objekt ist.

- Die Lösung ist, in den `Zuendung`-Objekten jeweils nur die Adresse eines `Artikel`-Objektes zu speichern (also eben einen Verweis/Zeiger auf ein `Artikel`-Objekt), und nicht das Objekt selbst.





# Fortsetzung des Beispiels (6)

- Am besten definiert man sich einen entsprechenden Zeiger-Typ (Adressen von Artikel-Objekten):

  ```
  typedef Artikel *art_t;
  ```

  Der Suffix "`_t`" ist nicht allgemeiner Standard: Man schaue sich verschiedene Stile an, und entscheide sich dann für einen.

- Ist `A` ein Wert vom Typ `art_t`, kann man z.B. die Methode `bezeichnung` folgendermaßen aufrufen:

  ```
  A->bezeichnung()
  ```

- In der Programmierpraxis arbeitet man häufig mit Adressen von Objekten, und nicht direkt mit den Objekten selbst: "`->`" kommt häufiger vor als "`.`".





# Verkettete Listen (1)

- Damit könnte man die Klasse `Zuendung` jetzt im gleichen Stil wie die Klasse `Artikel` deklarieren.
- Wir benötigen allerdings noch eine dritte Klasse `Plan` (für Abbrennpläne).

  Man hätte die Klasse vielleicht auch `Feuerwerk` nennen können.

- Ein `Plan` enthält im wesentlichen eine Liste von `Zuendung`-Objekten.

  Ein `Zuendung`-Objekt ist ja gerade ein Punkt im Abbrennplan.

- Es gibt verschiedene Arten, wie man diese Liste implementieren kann.





# Verkettete Listen (2)

- Eine häufig verwendete Methode zur Programmierung von Listen ist, das jedes Listenelement-Objekt (hier Objekt der Klasse `Zuendung`) einen Verweis auf das nächste Element der Liste enthält.

  Etwas unschön ist dabei, dass die `Zuendung`-Objekte verändert werden, um eine Funktionalität zu erbringen, die logisch erst zum `Plan` gehört, und auch nur mit einem `Plan`-Objekt zusammen funktioniert. Es gibt natürlich Alternativen, die aber auch gewisse Schwächen haben.

- Diese Methode heißt "Verkette Listen" und gehört auch noch zum "Kleinen $1\times 1$ der Programmierung".

  Man könnte aber sagen, dass es in der dritten Semesterwoche etwas früh dafür ist. Aber dieses Kapitel ist ein Experiment.

# Verkettete Listen (3)

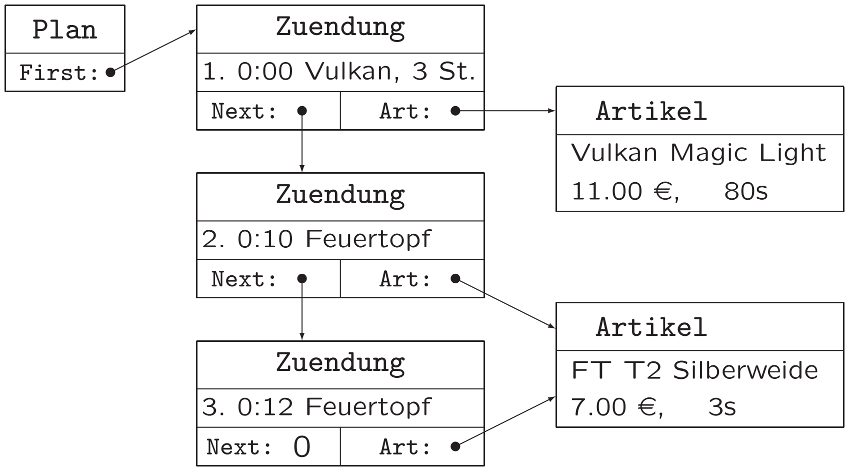
- Ein Objekt der Klasse `Plan` enthält die Adresse der ersten `Zuendung` im Plan ("Listen-Anker").
- Im ersten `Zuendung`-Objekt ist die Adresse des zweiten gespeichert, darin die Adresse des dritten, u.s.w.
- Im letzten `Zuendung`-Objekt ist eine sonst nicht benutzte Adresse "`0`" gespeichert (Ende-Markierung).

  Während C++ sonst Zeiger (Adressen) und Zahlen sauber auseinander hält (man muß schon explizit eine Umwandlung fordern), ist die Zahlkonstante `0` eine Ausnahme: C++ wandelt sie automatisch in jeden gewünschten Zeigertyp um, und garantiert, dass nie ein Variable (und damit auch kein Objekt) an dieser Adresse gespeichert wird. Diese spezielle Adresse ("Null-Zeiger", "Nil") ist also sicher von allen echten Adressen zu unterscheiden.





# Verkettete Listen (4)








# Verkettete Listen (5)

- Damit verkettete Listen und andere, noch interessantere Zeiger-Strukturen möglich sind, gibt es noch eine Ausnahme von der Regel "Alles muß vor der Verwendung deklariert sein":
  - ◇ Ein Zeiger `P` auf ein Objekt der Klasse `C` darf mit

    ```
    class C *P;
    ```

    deklariert werden, auch wenn die Klasse `C` noch nicht deklariert ist.

    Damit kann die Klasse `C` insbesondere ein Attribut haben, dass ein Zeiger auf ein Objekt der Klasse `C` ist, auch wenn die Klasse `C` noch nicht vollständig deklariert ist.





# Adressoperator

- Für die verketteten Listen benötigt man häufig die Speicheradresse von Objekten.
- Allgemein erhält man von jeder Variablen `V` durch Voranstellen von "`&`" die Speicheradresse (d.h. `&V` ist ein Zeiger/Pointer auf die Variable `V`).
- Sei folgende Deklaration gegeben:

  ```
  Artikel vulkan("Magic Light", 1100, 80);
  ```

  D.h. `vulkan` ist ein Objekt der Klasse Artikel.
- Dann ist `&vulkan` die Speicheradresse dieses Objektes vom Typ `Artikel*` (hier abgekürzt `art_t`).

# Objekterzeugung mit `new` (1)

- Bisher wurden Objekte angelegt, indem Variablen des Klassentyps deklariert wurden.
- Damit wäre aber nur eine feste Anzahl von Objekten möglich (unabhängig von der Eingabe).

  Solange man auf die Objekte nur über die Variablennamen zugreifen kann, ist ohnehin klar, dass man nur mit einer im Programm festgelegten Anzahl von Objekten arbeiten kann. Mit verketteten Listen braucht man nicht mehr für jedes Objekt einen Variablennamen: Man kann Objekte auch über die Verkettungsstruktur erreichen. Damit machen "anonyme" Objekte Sinn.

- Mit dem `new`-Operator kann man Objekte "dynamisch" erzeugen, z.B. in einer Schleife.





# Objekterzeugung mit `new` (2)

- Z.B. wird mit folgender Anweisung ein neues Objekt der Klasse `Artikel` erzeugt:

```
art_t art = new Artikel("KB 100mm Blinker",
                        990, 3);
```

- Die Variable `art` enthält jetzt die Speicheradresse eines neuen `Artikel`-Objektes.
- Nachdem man diese Adresse z.B. in einer verketteten Struktur gespeichert hat, kann man den Inhalt von `art` mit einer neuen Adresse überschreiben.





# Objekterzeugung mit `new` (3)

- Die Lebensdauer von mit `new` erzeugten Objekten ist nicht an Variablen gebunden, in denen die Adresse gespeichert wird.

  Diese Objekte existieren bis zum Programmende, oder bis zur expliziten Löschung (s.u.). Natürlich sollten alle Objekte ausgehend von benannten Variablen über Verzeigerungsstrukturen erreichbar sein. Ansonsten kann man auf die Variablen nicht mehr zugreifen, und sie belegen nutzlos Speicher. In manchen Sprachen werden solche Objekte automatisch gelöscht ("Garbage Collection"), in C++ nicht.

- Bei Bedarf kann man das Objekt durch folgende Anweisung löschen (d.h. den Speicher freigeben):

```
delete art;
```





# Klasse `Zuendung` (1)

- Klasse `Zuendung`, Teil 1 (Konstruktor):

```
class Zuendung {
  public:

    // Konstruktor:
    Zuendung(int nr, int zeit, art_t art,
             int stueck = 1)
    {
        Nr = nr;
        Zeitpunkt = zeit;
        Art = art;
        Stueck = stueck;
        Next = 0;
    }
```

# Klasse Zuendung (2)

- Klasse Zuendung, Teil 2 (Zugriffsfunktionen):

```
// Nr der Zeile im Abbrennplan (Kanal):
int nr() const { return Nr; }

// Zeitpunkt der Zuendung in Sek. vom Start:
int zeitpunkt() const { return Zeitpunkt; }

// Artikel, der gezuendet wird:
art_t artikel() const { return Art; }

// Stueckzahl (gleichzeitig gezuendet):
int stueck() const { return Stueck; }
```





# Klasse Zuendung (3)

- Klasse Zuendung, Teil 3 (verk. Liste, Attribute):

```
    // Naechster Eintrag im Abbrennplan:
    class Zuendung *next() const { return Next; }

    // Setzen des Verweises auf naechsten Eintrag:
    void set_next(class Zuendung *z) { Next = z; }

  private:
    int Nr;
    int Zeitpunkt;
    art_t Art;
    int Stueck;
    class Zuendung *Next;
};
```





# Klasse Zuendung (4)

- Deklaration eines Zeigertyps:

  ```
  typedef class Zuendung *zuendung_t;
  ```

- Im Konstruktor ist ein Defaultwert (Voreinstellung) für den vierten Parameter (Stückzahl) angegeben, nämlich 1.

  Gibt man jetzt bei der Objektkonstruktion nur drei Werte an, so ist das kein Fehler, sondern für den letzten Parameter wird automatisch der Wert 1 genommen.

- `Next` wird im Konstruktor auf den Nullzeiger gesetzt (Ende der Liste), mit `set_next` kann später ein Element angehängt werden (siehe Klasse `Plan`).





# Klasse Zuendung (5)

- Ich hätte das Attribut für den Artikel gerne `Artikel` genannt, aber dann kommt man mit dem Klassennamen durcheinander.

  Deswegen würde ich die Klasse in meinen Programmen `artikel_c` nennen, den Zeigertyp entsprechend `artikel_t`. Das schien mir für den Anfang dieser Präsentation aber zu technisch, und ist auch nur einer von vielen möglichen Programmierstilen. Informieren Sie sich über andere Stile, und klären Sie für sich, wie Sie Namenskonflikte vermeiden oder ggf. damit umgehen wollen. Selbstverständlich wäre es möglich gewesen, das Attribut `Artikel` zu nennen, man darf den gleichen Namen nur nicht im gleichen Gültigkeitsbereich für unterschiedliche Dinge verwenden. Dann wäre aber die Klasse `Artikel` verschattet gewesen. Man muss zwar in der Klasse `Zuendung` nicht direkt auf sie zugreifen, so dass das Beispiel funktioniert hätte, aber das wäre doch sehr unübersichtlich (schlechter Stil).

# Klasse Plan (1)

- Die wesentliche Funktionalität von Plan ist:
  - Eine Methode add, mit der man ein Zuendung-Objekt zum Plan hinzufügen kann (als jeweils letztes Element der Liste).
  - Eine Methode gesamtpreis, die die Summe aller Einzelpreise berechnet, und dazu die Liste der Zuendung-Objekte durchläuft.
- Natürlich wären in dieser Klasse weitere Analyse- und Ausgabe-/Visualisierungs-Funktionen denkbar, dafür fehlt uns jetzt aber die Zeit.





# Klasse Plan (2)

- Klasse Plan, Teil 1 (Konstruktor):

```
class Plan {
  public:

    // Konstruktor:
    Plan()
    {
        First = 0;
        Last = 0;
    }

    // Zugriff auf Liste (erstes Zuendung-Objekt):
    zuendung_t z_liste() const { return First; }
```





# Klasse Plan (3)

- Klasse Plan, Teil 2 (Hinzufügen zur Liste):

```
// Anhaengen an Liste:
void add(zuendung_t z)
{
    // Zeiger auf z setzen.
    // Erstes Element speziell behandeln:
    if(First == 0) // Liste ist bisher leer.
        First = z;
    else
        Last->set_next(z);

    // z ist nun letztes Element:
    Last = z;
}
```





# Klasse Plan (4)

- Klasse Plan, Teil 3 (Berechnung Gesamtkosten):

```
// Summe der Kosten aller Artikel (netto):
int gesamtkosten() const
{
    int kosten = 0;
    zuendung_t z = First;
    while(z != 0) {
        kosten = kosten + z->stueck() *
                          z->artikel()->preis();
        z = z->next();
    }
    return kosten;
}
```

# Klasse Plan (5)

- Statt “while(z != 0)” hätte man auch “while(z)” schreiben können, weil in C/C++ der Nullpointer als false, und alle anderen als true behandelt werden.
- Wenn man mit verketteten Listen arbeitet, ist eine Schleife wie die obige typisch.

  Man beginnt mit dem Listen-Anker (erstes Element), macht weiter, solange das Listen-Ende noch nicht erreicht ist (Null-Pointer), und setzt die Laufvariable (z) am Ende des Schleifendurchlaufs auf das jeweils nächste Listenelement.
- Üblicherweise wird es als for-Schleife geschrieben:

```
for(zuendung_t z = First; z; z = z->next())
```





# Klasse Plan (6)

- Bei der Programmierung sollten sich solche Muster mit der Zeit einprägen, so dass man nicht jedes Mal neu überlegen muss, wie man eine Schleife zum Durchlaufen einer verketteten Liste schreibt.
- Ein anderes typisches Muster in diesem Beispiel ist die “Akkumulatorvariable” kosten.
- Sie wird mit 0 initialisiert und dann werden die Kosten jedes einzelnen Punktes (im Abbrennplan) immer auf den aktuellen Wert aufaddiert.
- So enthält sie am Ende die Summe aller Kosten.





# Klasse Plan (7)

- Klasse Plan, Teil 4 (Attribute):

```
  private:
    zuendung_t First; // Erstes Element der Liste
    zuendung_t Last; // Letztes Element der Liste
};
```

- Natürlich hätte man auch noch weitere Daten erfassen können, wie das Datum oder den Anlass des Feuerwerks.
- Aufgabe: Berechnen Sie die Gesamtdauer des Feuerwerks als das Maximum von Zuendzeitpunkt + Brenndauer des Artikels (über allen Punkten).





# Hauptprogramm (1)

- Hauptprogramm mit Testaufruf (Teil 1):

```
int main()
{
    cout << "Feuerwerk-Planungssoftware 0.1\n";

    Artikel vulkan("V. Magic Light", 1100, 80);
    Artikel feuertopf("FT Silberweide", 700, 3);

    cout << "Bruttopreis Vulkan (in Cent): ";
    cout << vulkan.bruttopreis() << "\n";

    cout << "Bruttopreis Feuertopf (in Cent): ";
    cout << feuertopf.bruttopreis() << "\n";
```

# Hauptprogramm (2)

- Hauptprogramm mit Testaufruf (Teil 2):

```
    Plan mini;
    mini.add(new Zuendung(1, 0, &vulkan, 3));
    mini.add(new Zuendung(2, 10, &feuertopf));
    mini.add(new Zuendung(3, 12, &feuertopf));

    cout << "Gesamtkosten (netto, in Cent): ";
    cout << mini.gesamtkosten() << "\n";

    return 0;
};
```

- Beachte: Keine Variablen für Zündungen nötig, Objekte werden direkt in die Liste eingehängt.





# Hilfsprozedur (1)

- Natürlich sollen die Geldbeträge wie üblich in Euro und Cent ausgegeben werden.
- Das leistet die Prozedur auf der nächsten Folie.
- Sie ist außerhalb der Klassen deklariert, da
  - ◇ Geldbeträge in verschiedenen Klassen auftauchen
  - ◇ und ein Geldbetrag kein typisches Objekt ist (es ist ein Wert eines Datentyps).

    Objekte haben immer eine Identität, es könnte zwei verschiedene Beträge mit gleichem Wert geben. Wenn man es als Wert sieht, ist 1 Euro dagegen immer gleich 1 Euro. Man könnte natürlich eine Klasse für Geldbeträge definieren (mit Operatoren und Typumwandlungen, fortgeschrittene Übung für später).





# Hilfsprozedur (2)

- Prozedur zum Drucken eines Centbetrags als Euro und Cent:

```
void drucke_euro(int betrag)
// betrag in cent, nicht negativ
{
    int euro = betrag / 100;
    int cent = betrag % 100; // Divisionsrest
    cout << euro;
    cout << '.';
    if(cent < 10)
        cout << '0';
    cout << cent;
};
```





# Hilfsprozedur (3)

- Die Prozedur muss deklariert sein, bevor sie aufgerufen wird, also spätestens vor `main`.

  Wenn man diese Prozedur ganz vorne deklariert (nach den `#include`-Anweisungen), könnte man sie auch in den Klassen aufrufen.

- Beim Aufruf wird kein Objekt angegeben, da diese Prozedur keine Methode einer Klasse ist, z.B.

```
cout << "Gesamtkosten (netto, in Euro): ";
print_euro(mini->gesamtkosten());
cout << "\n";
```

- Die Gesamtkosten des Feuerwerks (Cent-Betrag) werden an die Prozedur `print_euro` übergeben.