C++ Notnagel

Ziel, Inhalt

- Ich versuche in diesem Dokument noch einmal die Dinge zu erwähnen, die mir als absolut notwendig für den C++ Unterricht und die Prüfungen erscheinen.

Grundlagen

Variablen, Datentypen und Kontrollstrukturen

Diese Themen werden im Prinz-Buch in den Kapiteln 1-6 behandelt. Dazu gehören aber auch die Kapitel 12 (Referenzen und Zeiger) und das Kapitel 21 (Speicherreservierung zur Laufzeit).

Variablendefinition

Einer Variablendefinition sieht so aus:

Datentyp Variablenname = Initialwert;

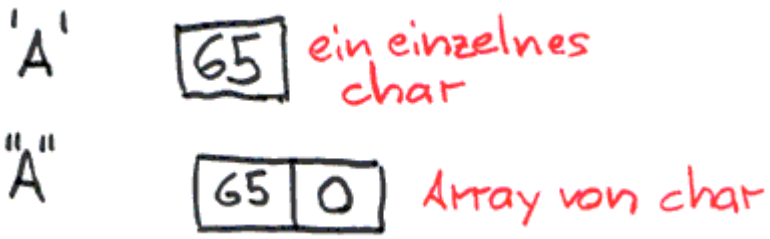
Beispiele:

```
int wert = 0;
char test = 0;
double blah = 1.0;
```

Beachte, dass der Datentyp char nicht verlangt, dass der Wert in Hochkommas steht! Häufig wird der char Typ verwendet um einzelne Zeichen zu speichern. Hierfür lässt der Compiler die Schreibweise mit Hochkommas zu, wobei der Wert dann dem ASCII-Wert des Zeichens entspricht.
Beispiele:

```
char wert = 12;
char c = ‚A’;  entspricht: char c = 65; (ASCII Wert von ‚A’)
```

Eine char Variable kann also auch für etwas anderes als Buchstaben verwendet werden, der Wertebereich beträgt -128 bis 127 und der Platzbedarf ist ein Byte. Der *unsigned char* hat den Wertebereich 0 bis 255.
Beachte auch:
Ein einzelner *char* steht immer in einfachen Hochkommas!!!
Der Datentyp von etwas was in Gänsefüsschen steht ist immer ein Array von *char,* denn der Compiler fügt immer noch eine abschliessende Null hinzu.



Konstanten

Eine Konstantendefinition ist im Grunde genommen nichts anderes als eine Variablendefinition mit dem zusätzlichen Schlüsselwort *const* vor dem Datentypen:

const Datentyp Konstantenname = Wert;

Beispiel:

```
const char wert = 0;
```

Im Gegensatz zu einer Variablen kann der Wert einer Konstante nicht mehr geändert werden.

Zeiger, Arrays und dynamischer Speicher

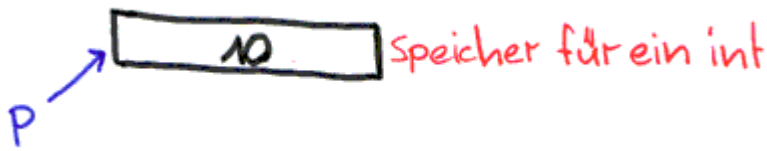
new erzeugt Zeiger

Ein *new* erzeugt immer einen Zeiger auf den Speicherplatz den man alloziert. Alloziert man Speicher für ein int ist der Datentyp int*.
Beispiel:

```
int* p = new int;
```

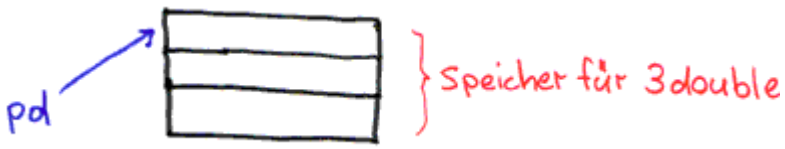
Man kann den Inhalt der Speicherstelle direkt initialisieren (in runden Klammern):

```
int* p = new int(10);
```



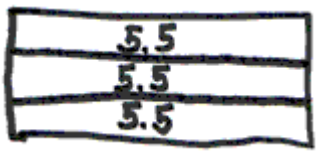
Mit *new* kann man auch mehrere Speicherstellen allozieren. Eine Aufgabe könnte heissen: Erzeuge ein Array von 3 double Werten dynamisch (mit *new)*:

```
double* pd = new double[3];
```



pd kann man nun verwenden wie ein Array. Falls die Aufgabe lautet: Setze die drei Werte auf 5.5 – können wir schreiben:

```
for(int i = 0; i < 3; ++i)
{
    pd[i] = 5.5;
}
```



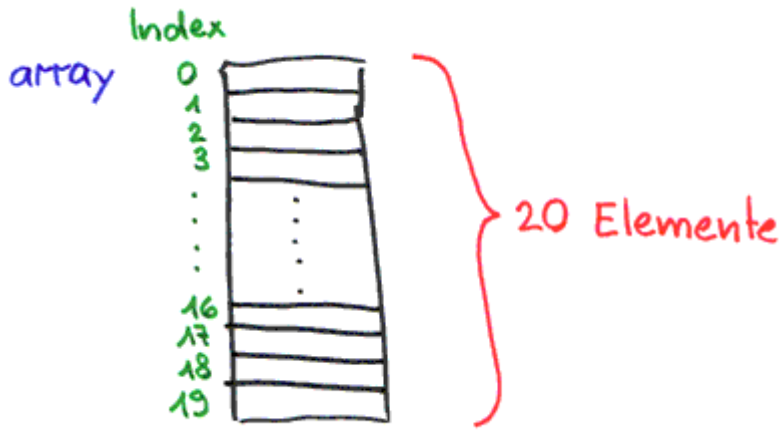
Um ein Array zu löschen, das dynamisch, also mit *new* erzeugt wurde, verwenden wir *delete* mit den eckigen Klammern:

```
delete [] pd;
```

Arraygrössen und gültige Werte für den Index

Wenn man ein Array erzeugt gibt die Zahl in den eckigen Klammen an wie viele Elemente das Array besitzt.
Beispiel mit 20 Elementen vom Datentyp long:

```
long array[20];
```



Die gültigen Werte für den Index sind 0 bis 19. Der grösste Index ist also die Anzahl Elemente minus 1!

C-Strings oder char-Arrays

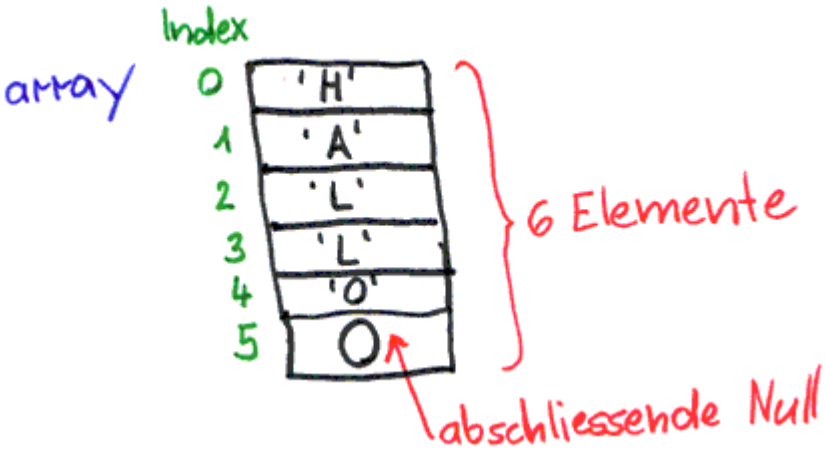
Genau wie wir Arrays von *long* Werten erzeugen können, können wir auch Arrays von char erzeugen:

```
char array[5];
```

Hierfür gelten die gleichen Regeln wie oben!

Da aber char häufig für Zeichenketten verwendet werden können wir auch folgendes schreiben:

```
char array[] = „Hallo“;
```



Wie wir wissen erzeugt jetzt der Compiler ein Array von char Werten mit den Buchstaben, die wir in Anführungszeichen gesetzt haben mit der zusätzlichen abschliessenden Null. Also hat das Array im „Hallo“-Beispiel 6 Elemente und der grösste Index ist 5!
Im Normalfall verwendet man die Zeigerschreibweise, denn Arrays sind ja auch Zeiger:

```
char* text = „Hallo“;
```

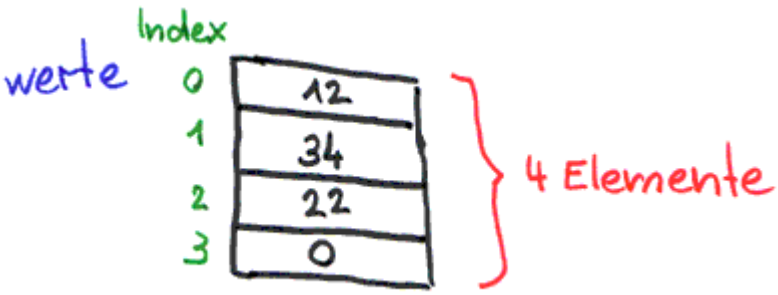
Initialisieren von Arrays

Arrays, die nicht mit *new* erzeugt wurden können direkt initialisiert werden:

Datentyp Arrayname[Anzahl] = { Initialwerte };

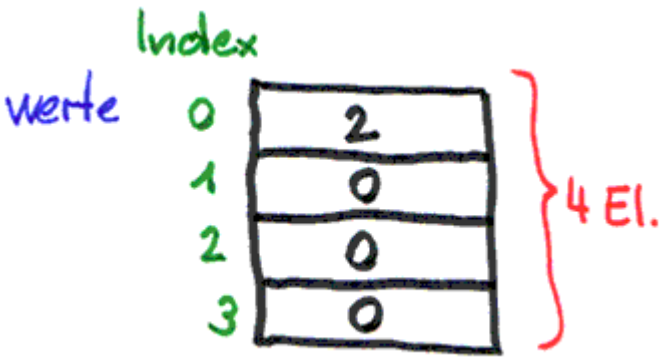
Beispiel:

```
int werte[4] = { 12, 34, 22, 0 };
```



Stehen rechts in eckigen Klammern weniger Werte als die Anzahl Elemente, werden die restlichen mit 0 initialisiert:

```
int werte[4] = { 2 };
```



Klassen

Definieren von Klassen

Heisst die Aufgabe: Definiere eine Klasse *Mischpult*. Dann sollte jeder gleich hinschreiben:

```
class Mischpult
{


hier ein wenig Platz auslassen!

};
```

Das Semikolon nicht vergessen!
Schreibe nur die Methoden hin, die in der Aufgabe verlangt werden!
Wenn die Aufgabe einen Konstruktor mit Parametern verlangt, ist es nicht immer nötig einen normalen Konstruktor zu definieren. Ein Konstruktor mit Parametern ist meistens nützlich um die Datenelemente direkt zu initialisieren. Beispiel:

```
class Mischpult
{
public:
    Mischpult(int anzahlKanaele, double preis);
private:
    int m_anzahlKanaele;
    double m_preis;
};

Mischpult::Mischpult(int anzahlKanaele, double preis)
          :m_anzahlKanaele(anzahlKanaele),
           m_preis(preis)
{
}
```

Wenn die Datenelemente nicht *const* sind ist auch folgender Konstruktor gültig:

```
Mischpult::Mischpult(int anzahlKanaele, double preis)
{
    m_anzahlKanaele = anzahlKanaele;
    m_preis = preis;
}
```

Definieren von struct

Strukturen sind eigentlich genau das gleiche wie Klassen! Im Gegensatz zu Klassen sind bei struct's alle Datenelemente *public*.
Bei Aufgaben wo es gilt eine Struktur zu definieren werden häufig gewisse

Dinge von den Datenelementen verlangt. Zum Beispiel sollen die Elemente möglichst wenig Speicher verbrauchen, oder sie müssen positiv sein. Darum gilt es die Datentypen sorgfältig auszusuchen und das Schlüsselwort *unsigned* richtig einzusetzen!

Objekte erzeugen

Ein Objekt einer Klasse wird wie eine Variable erzeugt. Beispiel:

```
Mischpult einMischpult;
```

Hier wird ein Objekt der Klasse Mischpult erzeugt. Falls die Klasse ein Konstruktor hat, wird dieser Konstruktor ausgeführt. Sobald die Variable ungültig wird, wird der Destruktor der Klasse aufgerufen, falls einer vorhanden ist.

```
if(b)
{
    Mischpult einMischpult; // der Konstruktor wird aufgerufen
    ....
    ....
} // Hier wird die Variable ungültig und der Destruktor wird
  // aufgerufen
```

Das geschieht auch in einer einfachen main – Funktion:

```
int main()
{
    Mischpult einMischpult; // der Konstruktor wird aufgerufen
    ....
    ....
    return 0; // Die Variable wird ungültig und der Destruktor
}             // wird – falls vorhanden – aufgerufen
```

Arrays von Objekten

Genauso wie man Arrays von int oder long Werten erzeugen kann, kann man Arrays von Objekten erzeugen. Beispiel:

```
int main()
{
    Mischpult array[4]; // Vier Objekte werden erzeugt
    ....
    ....
    return 0; // Vier Objekte werden zerstört
}
```

Das heisst, das auch vier mal der Konstruktor aufgerufen wird und vier mal der Destruktor! Natürlich nur wenn der Konstruktor oder der Destruktor definiert ist.

Klassen und Vererbung

Von einer Klassen erben

Um eine neue Klasse zu definieren, die von einer anderen ableitet genügt es folgende Schreibweise zu verwenden. Beispiel:

```
class Abgeleitet : public Basis
{
public:
....
private:
....
};
```

Ein Objekt der Klasse *Abgeleitet* ist auch ein Objekt der Klasse Basis. Die Datenelemente, die in der Basisklasse definiert sind, sind automatisch in der abgeleiteten Klasse auch vorhanden.
Es ist nicht nötig, sondern falsch die Datenelemente noch mal zu definieren (obwohl es grundsätzlich möglich ist)! Ein zusätzliches Datenelement kann aber definiert werden um die abgeleitete Klasse zu erweitern.

Konstruktor der Basisklasse

Häufig hat die Basisklasse einen Konstruktor mit Parametern. Im Konstruktor der abgeleiteten Klasse muss dieser Konstruktor in der Initialisierungsliste aufgerufen werden. Beispiel:

```
class Basis
{
    public:
        Basis(int daten);
    private:
        int m_daten;
};
Basis::Basis(int daten)
{
    m_daten = daten;
}

class Abgeleitet : public Abgeleitet
{
    public:
        Abgeleitet(int daten, double zusatzsdaten);
    private:
        double m_zusatzdaten;
};
Abgeleitet::Abgeleitet(int daten, double zusatzdaten)
           :Basis(daten)
{
    m_zusatzdaten = zusatzdaten;
}
```

Konstruktor, Destruktor

Wird ein Objekt erzeugt, wird zuerst der Konstruktor der Basisklasse aufgerufen. Wird ein Objekt zerstört, wird zuerst der Destruktor der abgeleiteten Klasse aufgerufen.

```
Basisklassen Konstruktor
Abgeleitete Klasse Konstruktor

Abgeleitete Klasse Destruktor
Basisklassen Destruktor
```

Virtuelle Methoden

Die Virtualität spielt nur eine Rolle wenn ein Zeiger oder eine Referenz auf ein Objekt verwendet wird. Ein Zeiger auf ein Objekt der Basisklasse kann auch auf ein Objekt der abgeleiteten Klasse zeigen. Beispiel:

```
Basis* p = new Abgeleitet;
```

Hier wird ein Objekt der Klasse *Abgeleitet* erzeugt. Zuerst wird der Basisklassenkonstruktor und danach der Konstruktor der abgeleiteten Klasse aufgerufen.
Werden jetzt Methoden aufgerufen werden im Normalfall die Methoden der Basisklasse aufgerufen, auch wenn die Methoden in der abgeleiteten Klasse redefiniert wurden.
Wurde eine Methode jedoch in der Basisklasse virtuell (mit *virtual*) definiert, wird die Methode der abgeleiteten Klasse aufgerufen. Das gilt auch für den Destruktor.

```
Basis* p = new Abgeleitet;
delete p;
```

Beim *delete* wird eigentlich nur der Destruktor der Basisklasse aufgerufen, ausser der Destruktor ist virtuell! Dann wird zuerst der Destruktor der abgeleiteten Klasse und danach der Basisklassendestruktor aufgerufen.
Also :
virtuell -> redefinierte Methode der abgeleiteten Klasse
nicht virtuell -> Methode der Klasse des Zeigers wird aufgerufen

Verschiedene Fragen

Hier werden häufig Funktionen verlangt.

Funktionen

Funktionen können kleine nette Helfer sein, um gewisse Berechnungen und derartiges zu leisten. Eine Funktion hat immer diese Form:

Datentyp Funktionsname(Argumente);

Der Datentyp gibt an, was die Funktion für einen Wert zurückgibt. Das sind häufig Funktionen, die irgendwelche Berechnung anstellen. Die Argumente sind mit Datentyp und Variablenname zu definieren. Beispiel:

```
#include <string>
using std::string;

string ErzeugeMail(const string& username,
                   const string& domain)
{
    string result = username + string("@") + domain;
    return result;
}
```

Es gibt auch Funktionen die kein Ergebnis liefern. Die tun nur etwas mehr oder weniger nützliches. Dann ist der Rückgabetyp *void.* Beispiel:

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

void SagHallo(int anzahl)
{
    for(int i = 0; i < anzahl; ++i)
    {
        cout << "Hallo" << endl;
    }
}
```

Wenn die Aufgabe lautet:
Schreibe eine Funktion *SagHallo* ist damit gemeint, dass die Funktion den Namen *SagHallo* hat. Es muss dann nicht eine main Funktion geschrieben werden.