

OO-Programmierkonzepte

C oder C++

Diskussionen welche Programmiersprache besser ist, sind furchtlos. Es müssen beide Sprachen als verschiedene Werkzeuge angesehen werden.

Unterschiede

	C	C++
Compiler	gcc	g++
Ausgabe	printf	cout
Eingabe	scanf	cin

Charakteristik von C++

- C++ erlaubt prozedural, objektorientierte und generische Programmierung
- C++ ist sehr mächtig
- C++ ist eine Obermenge von C
- Syntaktisch ist C++ sehr ähnlich oder identisch zu C
- C++ ist sicherer als C

Lexikalische Elemente von C++

≙ Namen

- Bezeichner, Schlüsselwörter, Literale, Operatoren, Kommentare sind identisch zu C und werden teilweise ergänzt
- Variablen, Konstanten und Objekte sowie Funktionen werden mit Kleinbuchstaben und camelCase geschrieben.
- Klassen, Strukturen, Enums beginnen mit GROSSBUCHSTABEN und verwenden camelCase

Typkonzept

- Es gibt die gleichen Basisdatentypen wie in C, zusätzlich gibt es noch den Datentyp **bool** mit Werten (true, false)
- #define ist in C++ verpönt da der Präprozessor eine reine Textersetzung ohne Syntax und Typprüfung macht.

► Alternative

- `const double pi = 3.14159;`
- `enum { maxVersuche = 3 };` // nur für ganze Zahlen

Ausdrücke und Operatoren

- völlig identisch zu C
- zusätzliche Operatoren für z.B. Type Casting

Anweisungen

- Syntax identisch zu C
 - Syntax der Steuerstrukturen
 - Sequenz
 - Iteration (for, while, do... while)
 - Selektion (if, if...else, switch)
 - Sprunganweisungen
 - break
 - continue
 - return
 - "goto"

Streams

Ein Stream repräsentiert einen sequentiellen Datenstrom und wird mittels `<<` und `>>` Operatoren ausgeführt. Diese Operatoren können auch überladen werden. Die Standardströme sind

- `cin`: Standard-Eingabestrom `>>`
- `cout`: Standard-Ausgabestrom `<<`
- `cerr`: Standard-Fehlerausgabestrom
- `cerr`: mit `cerr` gekoppelt

Diese Streams können auch mit einer Datei verbunden werden. Diese Streams sind einen Ersatz zu `printf` und `scanf` obwohl diese auch in C++ funktionieren. `cin` und `cout` müssen immer ganz links stehen.

- Methoden für die Ausgabe von vordefinierten Datentypen

- ▶ `ostream& operator << (int n);`
- ▶ `ostream& operator << (double d);`
- ▶ `ostream& operator << (char c);`

- Für Klassen kann der Operator überschrieben werden

- ▶ `ostream& operator << (std::string str);`

- Nutzung mit `cout`

- ▶ `int i = 45;`
`cout << "Hallo" << i << endl;`

- Methoden für die Eingabe von vordefinierten Datentypen

- ▶ `istream& operator >> (int& n);` // `(double& d), (char& c)`

- Für Klassen (Operator overloading)

- ▶ `istream& operator >> (std::string& str);`

- Nutzung mit `cin`

- ▶ `double d;`
`string str;`
`cin >> d >> str;`

- Beispiel Ausgabe in Hex mittels Format Flag

- ▶ `cout << showbase << hex << 27;` // Ausgabe 0x1b
↑
Format Flag

Funktionen

Synonyme : Unterprogramm, Subroutine, Prozedur (ohne Rückgabe), Methode

Divide et impera : Teile ein grosses Problem in viel Teilprobleme

Alles was in C möglich ist geht auch in C++ zusätzlich kommen noch Operatorfunktionen, inline-Funktionen, Default-Argumente und Überladene Funktionen.

Vorteile : spart Speicher

Nachteile : Zeitverlust Overhead.

Anstatt von C-Makros welche den Overhead mit #define lösen sollten besser inline Funktionen verwendet werden. Bei **inline** Funktionen findet eine Typenprüfung statt und werden eingesetzt wenn der Codeumfang klein ist und die Funktion häufig aufgerufen wird. Der Compiler entscheidet ob die Funktion inline wird oder nicht. Nicht jeder Linker ist fähig inline Funktionen zu linkern. Rekursive und Funktionspointer werden nicht inlined.

Default-Argumente können in den Funktionsprototypen zugewiesen werden. Jedoch müssen diese immer am ende, der Initialisierungsliste sein! Default-Argumente sind sehr nützlich für Konstrukturen

Overloading ist eine Möglichkeit der gleiche Funktionsname mit verschiedenen Parametern zu verwenden. Der Returtyp wird nicht beachtet. Jede Funktion braucht eine eigene Implementation. Wenn immer möglich Default parameter einsetzen.

Pointer *

Ein Pointer ist eine Variable, welche die Adresse einer im Speicher befindliche Variable oder Funktion aufnehmen kann. Pointer verhalten sich genau gleich wie in C, sie sind somit auch Typenabhängig. Der **Null-Pointer** zeigt auf kein Objekt und wird `int* ptr = 0;` initialisiert (NULL nicht verwenden)!

Pointer mit const werden immer von Rechts nach Links gelesen

► `const char* text = str;` // Pointer auf char konstante

► `char* const text = str` // konstanter Pointer auf char

► `const char* const text = str` // konstanter Pointer auf char konstant

Ein **void-pointer** kann ohne Typcast nur anderen void-Pointern zugewiesen werden (Anders als in C). Außerdem kann der void-Pointer nicht dereferenziert (*) werden und deshalb sollte er nicht verwendet werden.

In C++ gibt es bessere Alternativen anstatt **Funktions-Pointer** zu verwenden. Funktionspointer werden mittels Interruptvektortabelle dargestellt. Der Name der Funktion kann als Adresse auf den ersten Befehl der Funktion verwendet werden (wie Array).

Referenzen &

Eine Referenz ist ein Alternativname (Alias) für ein Objekt. Referenzen können auch auf Objekte in Registern und somit nicht im adressierbaren Bereich zugreifen. Referenzen sind effizienter als Pointer, ausserdem sind sie syntaktisch einfacher. Referenzen können nicht umgehängt werden.

Referenzen werden bei Parameterübergaben und als Returntyp eingesetzt. Objekte einer Klasse oder Struktur sollten immer by reference übergeben werden.

Es darf niemals ein Pointer oder eine Referenz auf eine lokale Variable mittels return zurückgegeben werden!

Dynamische Speicherverwaltung

Speicher kann zur Laufzeit (dynamisch) vom System angefordert (alloziert) werden. Dies geschieht mit Hilfe des Operator **new**. Dieser Speicher muss explizit wieder freigegeben werden mit Hilfe des Operator **delete**. Der dynamische Speicher wird auf dem Heap abgelegt und es kann nur mittels Pointer zugriffen werden. Es ist sehr wichtig, dass der Zugriff nie verloren geht!

```
int* pint = new int;  
      ↑      ↑  
auf Stack auf Heap  
  
*pint = 23;  
  
delete pint;
```

Der Delete Operator kann auch auf den Nullpointer angewendet werden. Jedoch sollte bereits freigegebener Speicher nicht nochmals freigegeben werden, da das zu Problemen führt. Deshalb werden Pointer nach dem Delete Operator auf 0 gesetzt.

Dynamischer Speicher sollte nicht für Echtzeitprogramme eingesetzt werden. **Memory Leak** ist Speicher, auf welchen nicht mehr zugegriffen werden kann. **Garbage Collector** (Java) ist ein Abfallsammler, welcher nicht mehr benötigten Speicher freisetzt.

Arrays

Arrays sind wie in C definiert, sie beginnen bei 0 und enden bei $n-1$. In C++ kann jedoch ein Array auch dynamisch alloziert werden mit `new[]` und `delete[]`

```
int* pint = new int[100]
```

```
pint[22] = -45;
```

```
delete[] pint;
```

Arrays müssen mit Hilfe `memcmp()` bitweise verglichen werden, oder der Operator muss selbst definiert werden. `==` ist keine Standardfunktion für Arrays. Der Arrayname ist ein konstanter Pointer auf das erste Element. Auf Arraynamen können nur Operatoren `sizeof` und `&` angewendet werden. Der Arrayname ist ein nicht modifizierbarer l-Wert.

Globale Arrays werden automatisch auf 0 initialisiert, lokale nicht. Lokale nicht dynamische Arrays können mittels `{}` initialisiert werden. `int alpha[2] = {1, 4}` werden nicht alle Zahlen explizit angegeben wird der Rest mit 0 gefüllt. Wird die `[]` leer gelassen zählt der Compiler und definiert die Grösse nach Anzahl Eingaben.

Mehrdimensionale Arrays

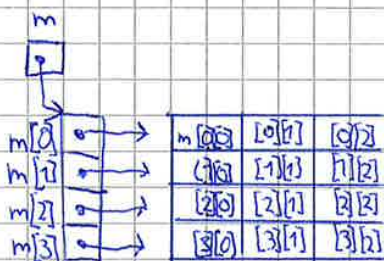
Ein mehrdimensionalen Array ist eine Matrix.

0	1	2
1	1	1
2	1	2

`int alpha[2][2] = {1, 2, 3, 4}`

1	2
3	4

Matrizen dynamisch allozieren



```
double** m = 0;
```

```
m = new double*[4];
```

```
for (int i = 0; i < 4; ++i)
    m[i] = new double[3];
```

Durch diese Aufstellung erfolgt die Zugriffe über die Arrayindizes

```
Freigabe for (int i = 0; i < 4; ++i)
        delete[] m[i];
```

```
delete[] m; // Allenfalls m auf 0 setzen
```

Übergabe von Arrays und Zeichenketten

Die Übergabe erfolgt mit einem Pointer auf das erste Element // void init (int* alpha, int size)
(int alpha[], int size)

Matrix: void printMat (double mat[][col], int m, int n);
nur erster darf leer sein.

Matrix mit Pointern (double** mat, int m, int n)

Referenz als Rückgabetype

Wird ein Parameter by value übergeben wird er nur kopiert, deshalb sollten wenn immer möglich Referenzen übergeben werden. Nur Parameter welche by reference übergeben werden können in der Funktion verändert werden.

Call by reference wird eingesetzt wenn Parameter zu groß werden (struct, class) oder wenn sie verändert werden. Durch Referenzen werden die Programme schneller. Für read only wird const verwendet.

Modul (Unit)

Grosse Probleme in kleine Unterteilen, das ist übersichtlicher und ermöglicht mehreren Personen daran zu arbeiten. Ein Baustein wird Modul oder Unit bezeichnet. Der Test wird Modultest oder als Unitest bezeichnet. Das Ziel ist möglichst schlanke Schnittstellen zwischen den Units zu erstellen und zusammengehörendes zusammenfassen. Ausserdem soll eine Hierarchie gebildet werden. Eine Unit kann in ein Gesamtsystem eingegliedert werden.

Ein Programm besteht aus:

- Main
 - Header // Schnittstelle
 - Source // Implementation
- Unit

Geheimnisprinzip

In der Header wird alles beschrieben was ein Nutzer wissen muss. Die .cpp Datei sollte dem Nutzer nicht bekannt sein. Der Entwickler darf den Inhalt der Unit ändern solange die Schnittstelle gleichbleibt. In der Schnittstelle (.h) wird dem Nutzer gesagt, welche Funktionen und Daten zur Verfügung stehen. (Es braucht jedoch Funktionen um auf die Daten zuzugreifen!) Sinnvollerweise werden Inline Funktionen verwendet

```
#include Guard:      #ifndef FOO_H_
                    #define FOO_H_

                    // Declaration

                    #endif /* FOO_H_ */
```

Deklarationsreihenfolge in Headerdateien

1. Dateikommentar
2. #include < >
3. #include " " // Kein using namespace
4. Konstantendefinitionen (Enums)
5. typedefs und Definition von Strukturen
6. Allenfalls extern-Deklarationen von globalen Variablen
7. Funktionsprototypen inklusive Kommentare

Implementationsreihenfolge in Implementierungsdatei

1. Dateikommentar
2. #include < >
3. #include " "
4. Globale und statische Variablen
5. Präprozessor-Direktiven (#define)
6. Funktionsprototypen von internen Funktionen
7. Definition von Funktionen und Klassen

Unit kompilieren

```
g++ -c foo.cpp
g++ -c goo.cpp
g++ -c hoo.cpp
g++ -o foo.exe foo.o goo.o hoo.o
```

Das makefile hilft den Prozess zu vereinfachen. Beim makefile werden nur Sachen kompiliert welche geändert wurden.

Strukturen

Strukturen sind grundsätzlich identisch zu C, die Nutzung ist jedoch einfacher geregelt, die mühsamen Trippels braucht es nicht mehr.

```
struct Point
{
    double x;
    double y;
};

Point p1;
```

Gültigkeitsbereiche (Scope)

Extern ist ein Statement in welchem kein Speicherplatz reserviert wird. Es dient lediglich um globale Variablen in dieser Datei bekannt zu machen. Das gleiche passiert auch wenn die Variable im Headerdatei enthalten ist.

Globale Variablen sollten so oft als möglich vermieden werden. Muss trotzdem eine globale Variable verwendet werden sollte immer ein Modulkürzel vor die Variable gestellt werden und mit einem underscore sollte sie verbunden werden. z.B. Foo_counter. Alle globalen Variablen müssen vor der ersten Funktion definiert werden.

Namensräume

Wie in C gibt es auch in C++ vordefinierte Namensräume

- Marken
- Strukturen, Unions und enums
- Bezeichner von Variablen, Funktionen, Typen etc. - Namen

In C++ können jedoch noch zusätzlich explizite Namensräume verwendet werden (namespaces). Um auf Bezeichner des Namensraum zuzugreifen wird der scope :: Operator verwendet. Innerhalb einer Datei kann mehr als ein Namespace verwendet werden und einen Namespace kann auch auf mehreren Dateien verwendet werden.

```
namespace myLib1
{
    int i;
    void foo();
}

myLib1::i = 13;
```

Mit Hilfe von using namespace kann das ganze noch vereinfacht werden

```
int main(void)
{
    using myLib1::foo;
    foo(); // ruft myLib1::foo() auf
}
```

Falls der ganze Namensraum verwendet werden soll

`using namespace mylib1;`

Falls die Funktion `foo()` in mehreren namespace vorkommen und beide namespace include werden rekognisiert der Compiler. `using namespace` nie in Header verwenden

In C++ gibt es ebenfalls die Funktion `namespace`, welcher mehr oder weniger identisch ist zu `static` in C.

Mit dem Scope Operator kann auch auf globale Variablen zugegriffen werden.

Speicherklassen

In C++ gibt es folgende Speicherklassen

- ▶ **auto** ist default, wenn nichts geschrieben wird, Variable automatisch am ende des scopes Untertyp
- ▶ **register** Hinweis an den Compiler (wenn's geht bitte in Register ablegen)
- ▶ **static** Lokale Variablen welche den wert behalten
Auf 0 initialisiert
- ▶ **extern** Globale Variablen welche in einer anderen Datei definiert sind.
- ▶ **mutable** Um `const` zu umgehen

Typenqualifikationen

- ▶ **const** read-only
- ▶ **volatile** Der Compiler wird angewiesen keine Optimierungen vorzunehmen

Funktionsattribute

- ▶ **inline** Funktionen in die Zeilen einfügen
- ▶ **virtual** Funktionen überschreiben
- ▶ **explicit** Um eine Typenumwandlung zu vermeiden

Typdefinitionen

Typedef sind grundsätzlich, jedoch ist es nicht nötig bei struct Typedefs zu verwenden. Typedefs werden in Großbuchstaben geschrieben.

Typedefs werden auch verwendet um die Breite der Typen klar festzulegen z.B. 76 Bit auch in (stdint.h)

Typedef ist eine reine Textersetzung

Typcast

- Unsafe conversion double $\rightarrow$ int
- Safe conversion int $\rightarrow$ double

Explizite Typumwandlung (C style)

- int a = (int) 4.6;
 - int b = int(4.6);
- } // a == 4

Explizite Typumwandlung (C++ style)

- str str(const_cast<char*> str;

Explizite Typumwandlung (static_cast, dynamic_cast, reinterpret_cast)

Funktionieren gleich wie constcast

Debuggin

- Test = Fehler finden
- Debuggin = Fehler lokalisieren und beheben

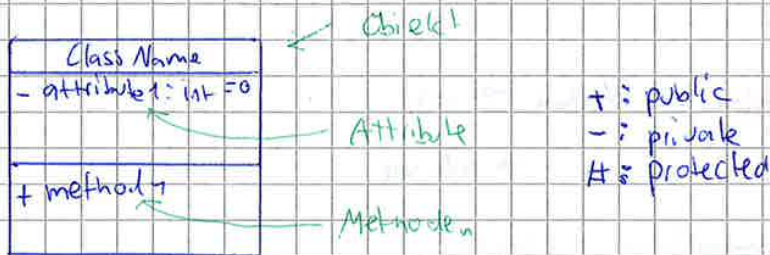
- Assertion `#include <cassert>`
`assert(i > 0);`

Abbruch wenn es den Boolischen Wert false hat. Asserts sind wirkungslos wenn ohne debugschalter kompiliert wird

Klassen

UML steht für Unified Modeling Language und ist eine graphische Modellierungssprache. UML ist Standard in Softwaremodellierung und setzt sich ein fließendes Modellierungskonzept in allen Entwicklungsphasen zum Ziel. UML integriert Datenmodellierung, Prozessmodellierung, Zustands- und Verhaltensmodellierung, Steuerfluss-Modellierung.

Eine **Klasse** ist eine Struktur mit Funktionen und gilt als Umsetzung der Datenkapsel. Variablen werden als Objekte und Funktionen als Methoden bezeichnet, die Daten bezeichnen man als Attribute



Syntax:

```
class Classname
{
    public :
    ...
    protected :
    ...
    private :
    ...
};

classname obj1;
```

Alle Methoden und Attribute welche mit Public gekennzeichnet werden sind überall sichtbar. Die welche mit Private gekennzeichnet sind sind nur in der Klasse sichtbar (Classname.cpp). Auf Methoden wird mittels des scope(ö) Operator zugegriffen. Elementarfunktionen (Methoden) haben auf alle Attribute Zugriff.

Klassifizierung

- Konstruktor / Destruktor erzeugt/vernichtet Objekt
- Modifikator, ändert Zustand (Attribute)
- Selektor greift nur lesend zu
- Iterator Erlaubt auf Elemente eines Objekts zuzugreifen.

Methoden welche im .h File implementiert sind werden als Inlinefunktionen behandelt ('implizit' immer). Wird das Schlüsselwort **const** angehängt so kann die Methode keine Attribute ändern. **mutable** ist für Attribute welche nie const werden sollen.

this - Pointer

Der this-Pointer ist ein Pointer welchen immer auf das Objekt zeigt

z.B.

```
Article& Article::getCheaper(const Article& a) const {
    if (a.Price < price)
        return a;
    else
        return *this; // Zeigt immer auf das aktuelle
                      // Objekt in diesen Fall price
}
```

Der this-Pointer ist ein Pointer auf das eigene aktuelle Objekt welches eine Methode aufgerufen hat

- this → anyFoo();
- if (this == &obj) // Testet ob die Adressen gleich sind

Zugriffschutz bei Klassen

Innerhalb der Klasse hat jede Methoden auf alle Elemente Zugriff. Ausserhalb haben sie nur Zugriff wenn sie als public gekennzeichnet sind. Per default sind die Elemente bereits als private registriert, Trotzdem wird es gerne hingeschrieben. Protected haben Sie Zugriff auf die Elemente in einer Unterkasse

Struct und Class sind grundsätzlich das Gleiche jedoch ist per default  public und  privat verschieden. In der Praxis werden Structs nur für Attribute eingesetzt und classes für Methoden

friend ist eine Funktion um auf Elemente anderer Klassen zuzugreifen. Die Zugriffsregeln werden gebrochen und deshalb sollten sie Sparsam eingesetzt werden.

Wenn das Attribut mit static gekennzeichnet wird, dann teilen sich alle Objekte dieser Klasse eine einzige Instanz dieses Attributs, d.h. ein statisches Attribut ist nur einmal für alle Objekte einer Klasse im Speicher vorhanden. Static Elemente befinden sich ausserhalb eines Objektkontexts, jedoch können sie trotzdem mit dem Klassennamen angesprochen werden.

- .h static int nrOfObjects;
- .cpp int T::nrOfObjects = 34;
- Zugriff falls public T::nrOfObjects++;
T myT;
myT.nrOfObjects++;

Konstruktor

Der Konstruktor erzeugt ein neues Objekt ~~es~~ und hat keinen Rückgabewert.

► `Stack::Stack();` // (Default) Konstruktor

Der Konstruktor wird implizit vom System aufgerufen das funktioniert auch bei impliziten Aufrufen. Der Konstruktor wird nur aufgerufen wenn ein neues Objekt erzeugt wird.

Der Default Konstruktor kann auch selbst erzeugt werden das ist bei dynamisch allozierten Objekten enorm wichtig für eine saubere Programmierung sollten default Werte mit gegeben werden.

► `TString::TString()`
 : len(0), Str(0)

Konstrukturen mit nur einem Parameter können dazu verwendet werden, ein Objekt vom Typ `T` aus einem anderen Objekt zu erzeugen. (Typumwandlung `T*`)

Es gibt auch eine Möglichkeit Konstrukturen als explizit zu bezeichnen so werden alle impliziten Konstrukturen unterdrückt.

Der **Copy Konstruktor** kopiert Objekte und wird so definiert: `TString(const TString& s);`. Der Copy Konstruktor wird ebenfalls automatisch aufgerufen und wird bspw. einer Funktion übergeben (Niemals als Referenz).

Shallow Copy kopiert nur die Adressen aber nicht der Speicher welcher auf dem Heap liegt.

deep copy kopiert alles jedoch muss der Konstruktor selber definiert werden. Wird vor allem bei dynamischem Speicher verwendet.

Destruktor

Der Destruktor zerstört Objekte welche nicht mehr gebraucht werden vollständig. Insbesondere ist der Destruktor wichtig wenn Speicher auf dem Heap vorhanden ist. In diesem Fall muss der Destruktor selbst bestimmt werden. Der Destruktor hat ebenfalls kein Rückgabetypp. Die Reihenfolge ist die umgekehrte Reihenfolge der Konstruktoran. Der Destruktor wird folgendermaßen gekennzeichnet **virtual ~TString();**

↑
title

Konstruktor müssen immer Public sein, außer z.B. das Kopieren soll absichtlich verhindert werden.

Referenzen bei Klassen

Wird bei einer Funktion (Methode) der Returntyp by value ausgeführt, dann wird immer der copy Konstruktor ausgeführt. Deshalb sollte immer eine Referenz genommen werden. Wenn ein lokales Objekt zurück gegeben werden soll dann muss das weiterhin by value übergeben werden. **Eine Referenz auf ein lokales Objekt ist Verboten!** Andere Objekte sind auch nach Beendigung der Elementfunktion noch Gültig.

Automatisch generierte Elementfunktionen (falls sie vom Programm benötigt werden und nicht explizit deklariert werden)

- ▶ Default-Konstruktor
- ▶ Copy - Konstruktor
- ▶ Zuweisungsoperator
- ▶ Adressoperator

Die **Kanonische** Form einer Klasse bezeichnet man die Form, die es erlaubt eine Klasse wie einen normalen Datentyp zu benutzen. Dabei müssen folgende Bedingungen erfüllt sein:

- ▶ Korrekter Konstruktor
- ▶ (bei dynamischen Daten) einen Zuweisungsoperator und Copier
- ▶ Einen Virtualer Destruktor.

Benutzerdefinierte Typumwandlung

Wenn zwei ganze Zahlen unterschiedlichen Typs addiert werden müssen, so ist der Additionsoperator (Auch die Anderen) vom System für folgende Varianten definiert:

- `int + int` - `int + short` - `short + int` - `short + short`

Wenn nun eine Klasse `VeryLargeInt` eingeführt wird, müssen die Operatoren selbst definiert werden. Für `int` und `VeryLargeInt` wären 24 Operatoren notwendig. Deshalb ist es besser eine Typumwandlung auf den größten Datentyp zu absolvieren. Dort sind nur noch 8 Operatoren zu definieren: `+, -, *, /, +=, -=, *=, /=`

Häufig wird die Typumwandlung mit Hilfe der Konstruktoren implementiert: `VeryLargeInt(int);` z.B. `command(const uint8_t* bytes)`

Unions

Analogisch wie eine Struktur, doch im Gegensatz zur Struktur ist nur ein einzelnes Feld jeweils aktiv (abhängig vom Typ). Die Grösse der Union ist genauso wie das grösste Feld in der Union. Bei der Union wird auch zwischen `Public` und `Private` unterschieden, jedoch sollten Sie vorzugsweise in `Public` programmiert werden um den Zusammenhang mit C zu gewährleisten.

```
Union UnionName // Union UnionName ist ein hier  
{               // selbstdefinierter Typ, der weiter verwendet  
    Public      // werden kann.  
    Private:  
        FieldType1 field1; // FieldType kann eine Union oder ein  
        FieldTypeN fieldN; // Struct sein.  
};
```

Der Programmierer muss verfolgen, welcher Typ jeweils in der Union gespeichert ist. Der Datentyp der entnommen wird, muss der sein, der zuletzt gespeichert wurde. Unions sollten zurückhaltend eingesetzt werden.

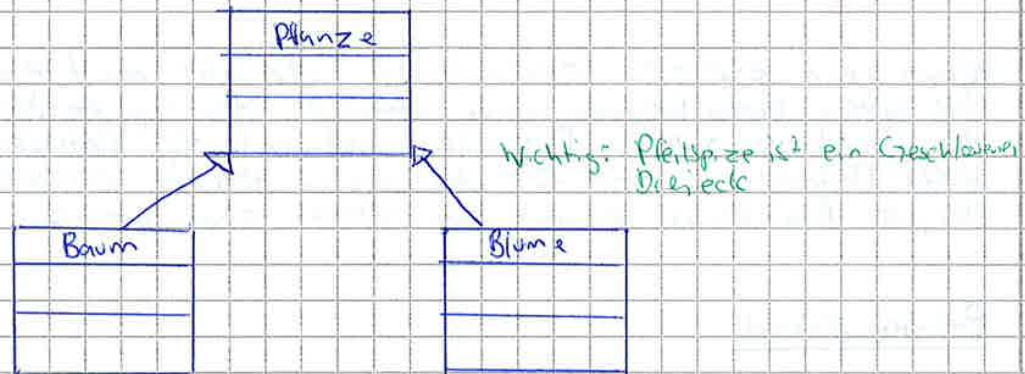
Bitfelder

Innerhalb eines `ints` können einzelne Bitgruppen definiert und angesprochen werden. Aber leider ist der C++ Standard nicht definiert, ob von links nach rechts oder von rechts nach links gelesen wird. Deshalb nicht verwenden.

```
Struct FieldName  
{  
    unsigned int a:3; // definiert 3 Bits für a  
    unsigned int b:4; // definiert 4 Bits für b  
};
```

Vererbung (Inheritance)

Die Vererbung erlaubt, neue Klassen auf Basis von bestehenden Klassen zu definieren. Dabei erbt die neue Unterklasse alle Eigenschaften der bestehenden Oberklassen. Man sagt der Oberklasse auch Basisklasse, superclass, Abstraktion oder Generalisierung. Die Unterklasse wird auch subclass oder Spezialisierung genannt. Eine Vererbung ist sinnvoll, wenn bestehende Methoden überschrieben werden oder bestehende Klassen mit zusätzlichen Attributen ^{Methoden} erweitert werden. Für die Vererbung ist wichtig, dass es eine Beziehung ist (Blume ist eine Pflanze).



Syntax:

```
class Subclass : public Super Class
{
    public
    ...
    private
};
```

Zugriff ist möglich auf alle public und protected-Elemente der Basisklasse, die Zugriffsrechte (public/protected) der Basisklasse bleiben auch in der subclass erhalten. Es gibt keinen Zugriff auf private Elemente der Basisklasse. Attribute sollte Protected generell nicht eingesetzt werden (immer private). Die Methoden können durchaus mit protected versehen werden.

Substitutionsprinzip: Ein Objekt einer Oberklasse kann Objekte einer beliebigen Unterklasse aufnehmen, jedoch kann ein Objekt einer Unterklasse keine Objekte der Oberklasse aufnehmen.

z.B. Oberclas Super;
Unterclas sub;
Super = sub;
~~Sub = Super;~~

Das Objekt der Unterklasse ist mindestens so groß wie das Objekt der Oberklasse. Es gibt keine Vererbung by reference, jedoch ist eine Aggregation by reference möglich.

Es gibt eine Möglichkeit bei Unterklassen die Attribute der Basisklasse direkt initialisieren. Der Konstruktor initialisiert explizit nur die Attribute der eigenen Klasse. Deshalb ist es sinnvoll der Konstruktor der Oberklasse aufzurufen anstatt jedes Attribut einzeln zu initialisieren.

```
z.B. Book : Book { const string & aName  
                int aCode  
                const string & aAuthor  
                const string & aIsbn }  
Article(aName, aCode,  
        author(aAuthor), isbn(aIsbn))  
}
```

Wenn kein expliziter Copykonstruktor definiert wird erzeugt das System einen eigenen, welcher ebenfalls den Copykonstruktor der Basisklasse aufruft. Das Gleiche ist mit den Destruktoren. Jeder Destruktor ist nur für die eigenen Attribute zuständig. Der, der Basisklasse kommt am Schluss zum Einsatz.

Polymorphismus

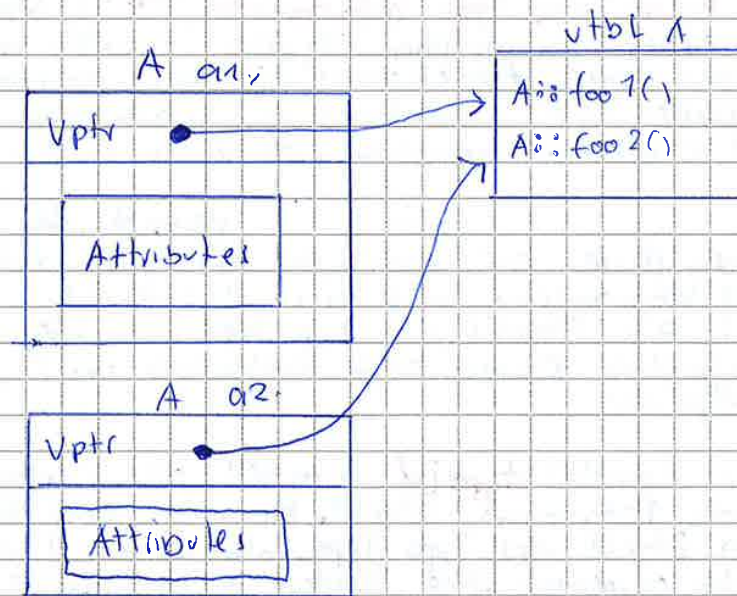
Bei normaler Weise wird bereits zur Compilezeit festgelegt, welcher Elementfunktionscode ausgeführt wird. **Static binding**. Soll dies erst zur Laufzeit bekannt werden nennt man das **Dynamic binding** oder late binding. Dieses Konzept nennt sich Polymorphismus. Um Funktionen dynamisch zu gestalten müssen sie zwingend mit **virtual** deklariert werden. Sowohl in der Basisklasse (notwendig) als auch in der Unterklasse (empfohlen). Virtual sollte nur eingesetzt werden, wenns wirklich notwendig ist (Nur wenn Methoden überschrieben werden).

Bei dynamischen Aufrufen erfolgt die Auswahl der richtigen Funktion aufgrund des tatsächlichen Datentyps. Dies ist mit **Overhead** verbunden, welches bei statischen Aufrufen nicht der Fall ist.

Eine Klasse welche mit mindestens eine virtuelle Funktion hat heißt virtuell. Virtuelle Klassen bewirken einen Mehraufwand für den Compiler und sind langsamer in der Ausführung. Konstruktor sind nie virtuell, Destruktoren hingegen sobald eine Methode virtuell ist.

In der **Virtual Function Table (vtbl)** vermerkt das System die Reihe nach die Adressen der für eine Klasse gültigen virtuellen Methoden. Das System legt für jede polymorphe Klasse eine vtbl an. Jedes Objekt einer polymorphen Klasse enthält einen Virtual Pointer **vptr** auf die vtbl der entsprechenden Klasse.

Beispiel für vtbl



Abstrakte Klassen werden dort eingesetzt wo normale Vererbungen nicht ausreichen. Beispiel ein Kreis ist ein Spezialfall einer Ellipse ist aber nicht sinnvoll ihn so zu programmieren, da er sonst Sachen erbt die er nicht will. Deshalb ist es sinnvoll eine Basisklasse und zwei vtbl Klassen zu programmieren jedoch ist die Basisklasse nicht vollständig und wird deshalb abstrakte Klasse genannt. Es können keine Objekte von abstrakten Klassen gebildet werden.

In C++ können rein virtuelle Funktionen deklariert werden, die in der Basisklasse nicht von einer Definition begleitet werden.

```
virtual double getArea() = 0;
```

Klassen, die mindestens eine rein virtuelle Funktion haben sind abstrakte Klassen. Diese Klasse kann dann nur durch eine Vererbung nutzbar gemacht werden.

Manchmal ist es auch sinnvoll eine abgeleitete Klasse von mehreren verschiedenen Basisklassen zu erben, das ganze nennt man dann **Mehrfachvererbung**.

```
class SingingWaiter : public Waiter, public Singer
```

Bei den Konstruktoren müssen unbedingt die Reihenfolge beachtet werden. Der der Zuerst steht wird zuerst ausgeführt. Bei den Methoden ist darauf zu achten dass sie immer mit Oberklasse:: ausgeführt werden da sonst eine Mehrdeutigkeit besteht.

Um zu vermeiden dass bei Mehrfachvererbung eine gemeinsame Basisklasse mehrfach im Speicher vorkommt werden die Oberklassen mit **virtual** gekennzeichnet.

```
class SuperHero : virtual public ComicCharacter
```

Mit **Run-Time Type Information (RTTI)** kann der (dynamische) Typ eines polymorphen Objektes ermittelt werden. Der Mechanismus funktioniert nur bei polymorphen Klassen. Es stehen drei Operatoren zur Verfügung, welche sehr viel Overhead brauchen. (`dynamic_cast`, `typeid`, `type_info`)

`dynamic_cast<SuperHero*>(p)` versucht den pointer `p` in einen pointer auf ein Objekt des Typs `SuperHero` umzuwandeln. Die Umwandlung wird nur durchgeführt, wenn `p` tatsächlich auf ein Objekt vom Typ `SuperHero` bzw. auf eine davon abgeleitete Klasse zeigt, andernfalls wirds zu einem Nullpointer.

Der Operator **`typeid`** ermittelt den dynamische Datentyp eines Polymorphen Objekts. Er gibt eine Referenz auf ein Objekt des Typs `type_info` zurück. Diese Klasse beinhaltet unter anderem eine Methode `name()`, welche den Namen der Klasse zurückgibt.

```
cout << "p ist ein " << typeid(*p).name() << " Objekt";
```

RTTI soll nur äußerst selten eingesetzt werden.

Überladen von Operatoren

Das Wort überladen wird hier nicht im Sinne von "zu viel laden" sondern im Sinne von "darüberladen, neu definieren" verwendet. Mit Operator overloading kann ein bestehender Operator neu definiert werden. Operator overloading ist nicht objektorientiert aber enorm praktisch. Die Operatoren, welche überschrieben werden können, können in der üblichen Operatornotation als auch in der Funktionsnotation verwendet werden. Um Operator overloading einzusetzen müssen folgende Randbedingungen eingehalten werden:

- Anzahl der Operanden muss gleich sein
- Die Priorität des Operators kann nicht geändert werden.
- Es können keine neuen Operatoren eingeführt werden
- Default Argumente sind nicht möglich

Der neu definierte Operator wird als Elementfunktion implementiert, damit ist der Zugriff auf private und protected Attribute der Klasse möglich.

```
class TString
{
    bool operator < (const TString& s) const;
}

bool TString::operator < (const TString& s) const
{
    // Implementation
}
```

nur 1 Argument das zweite ist `*this`

Bei der Anwendung muss auf der linken Seite zwingend ein `TString` sein.

Der Neudefinierte Operator kann genau gleich wie der normale Operator verwendet.

```
TString s1 = "Hallo";  
TString s2 = "Hello";  
if (s1 < s2) // s1 < s2 entspricht s1.operator <(s2)  
{ ... }
```

Jedoch steht auf der linken Seite s1 an dieser Seite muss ein Objekt der Klasse TString sein auf der rechten Seite könnte auch "s1" stehen.

Muss der Operator auf der linken Seite genau das können muss der Operator nicht als Elementfunktion definiert werden dadurch hat er aber keinen Zugriff auf private und protected Attribute. Deshalb muss die Funktion als friend deklariert werden.

```
class TString  
{  
    friend bool operator <(const TString& s1,  
                          const TString& s2);  
};  
  
bool operator <(const TString& s1, const TString& s2)  
{  
    ...  
}
```

(2 Argumente da der Operator ausserhalb einer Klasse definiert ist.)

Um zu vermeiden, dass für ein Operator unzählige Varianten mit Typen definiert werden müssen kann auf benutzerdefinierte Typumwandlung zurückgegriffen werden.

Zuweisungs - (=), Index[] , Funktionsaufruf () und Zeigeroperator (->) müssen immer als Elementfunktion implementiert werden

Der Zuweisungsoperator muss selbst definiert werden sobald die Klasse Elemente auf dem Heap enthält. Der Indexoperator kann definiert werden um einfacher auf die einzelnen Positionen zuzugreifen (Achtung es gibt zwei verschiedene Varianten)

Ein **Stream** repräsentiert einen sequentiellen Datenstrom und ist mit << und >> Operatoren definiert. Für selbstdefinierte Klassen können die Operatoren überladen werden. C++ stellt die Standardströme **cin, cout, cerr, clog**

Vordefiniert

Ostream& operator <<(int n); // auch Double und Char

Für Strings müssen diese Operatoren jedoch überschrieben werden

Ostream& operator <<(std::string str);

Das gleiche gilt auch für **Istream& operator >>(....)**. Diese Operatoren werden normalerweise als friend deklariert.

Templates

Templates kommen zum Einsatz wenn der Gleiche Algorithmus mit unterschiedlichen Typen verwendet werden soll. Bis jetzt musste immer eine eigene Funktion geschrieben und getestet werden. Oder es wurde mit dem nicht sinnvollen void-pointer gearbeitet werden.

Die Generische Programmierung mit Templates wird jedes Problem nur einmal gelöst unabhängig vom Typ.

(**Single Source-Prinzip**). Template Definition erzeugen grundsätzlich noch keine Zeile Code. Erst wenn das Template verwendet wird, wird Code für diesen Typ erzeugt.

Templates sind somit ideal für Bibliotheken (im Binärfile sind nur Sachen welche wirklich gebraucht werden)

Templates verwenden den Typ als "Variable". Templates sind keine Funktionsdefinition, so beschreiben dem Compiler nur, wie er den Code definieren soll. d.h. der Compiler nimmt den konkret verwendeten Typ, ersetzt diesen in das Template ein und compiliert den so erhaltenen Code. Das ganze wird **early binding** genannt.

Vor den Funktionsnamen wird das Schlüsselwort **template** gefolgt von einer in spitzen Klammer eingeschlossene Parameterliste. Die Parameterliste enthält eine (nicht leere) Liste von Typ- und Klassenparameter, die mit dem Schlüsselwort **class** oder Typenamen. Einzelne Parameter werden mit Komma getrennt.

.h file

```
template<typename ElemType>  
ElemType minimum(ElemType elemField[], int fieldSize);
```

oder:

```
template<typename A, typename B>  
int foo(A a, B b, int i);
```

inline



inline steht hier falls es gebraucht wird

Funktions-Templates können auch explizit mit einem Typ qualifiziert werden

```
int i = minimum<int>(IF, sizeof(IF)/sizeof(it[0]));
```

Außerdem können Funktions Templates auch mit normalen Funktionen **überladen** werden. Der Compiler sucht die beste Version. (Vorsicht! und nicht im Zusammenhang mit default Parameter verwenden)

Klassen-Templates sind mit Typen oder Konstanten parametrisierbare Klassen. Im Gegensatz zu Funktions-Templates können in Klassen Templates auch die Attribute der Klassen mit variablen Typen ausgestattet sein. Ein Klassen Template kann auch von Ausdrücken abhängig sein, welche zu Compile Zeit aufgelöst werden. Die Syntax ist analog zu den Funktionstemplate.

Deklaration

```
template < typename ElemType, int size = 100 > // default
class Stack
{
public:
    Stack();
    ~Stack();
    ElemType pop();
    void push (const ElemType & elem)
private:
    ElemType elems[size];
    int top;
};
```

Definition

```
template < typename ElemType, int size > // default nicht erneut verwenden
void Stack < ElemType, size >::push (const ElemType & elem)
{
    // Implementation
}
```

Natürlich kann auch mehr als ein Typ pro Klasse verwendet werden. Ausserdem können Klassentemplates analog zu den Funktionstemplate explizit mit einem Typ qualifiziert werden.

Der Schlüsselbegriff **export** wird im C++ 11 nicht mehr unterstützt.

Weil bei der Definition und Deklaration nur die Schablone entsteht muss das .cpp file mittels **ugly** **include** auch ins .h file eingefügt werden, aber nicht umgekehrt. Eine Bessere Version ist ein .t file zu erstellen mit der ganzen Template class.

Exceptions

Am Anfang ist sicherzustellen das Exception und Error nicht das Gleiche ist. Error bedeutet "etwas ist falsch programmiert" und **Exception** ist eine abnormale aber vorhersehbare und mögliche Bedingung bei der Programmausführung. Wir sprechen hier über Exception Handling.

Es gibt drei Mögliche Reaktionen auf Ausnahmen

- Ignorieren
- Programmabbruch (**Exception Detection**)
- Exceptioncodes

Es gibt die Möglichkeit den Exception code als Rückgabewert oder Referenz mit if und else zurückzugeben, jedoch ist das sehr unschön und unleserlich. Diese sollten auch nicht global sein.

Das **Ziel für Exception Handling** ist dass der Normale Programmablauf nicht durch Exceptions gestört werden. Der Normalfall soll einfach gelesen werden und der Ausnahmefall ist klar und einfach geregelt. Außerdem sollte der Overhead möglichst klein sein.

Syntax:

```
try
{
    // Code, welcher Exceptions auswerfen könnte
}
catch (const MyException& &ex)
{
    // Wenn ein Objekt der Klasse ↑ ausgeworfen wurde wird es hier
    gefangen.
}
```

Ausnahmen können mit dem Schlüsselwort **throw** explizit ausgeworfen werden. Bei einem throw befehlt wird das Programm sofort unterbrochen und beim Fänger weitergemacht, Dabei werden alle lokalen Objekte wieder zerstört. Geworfen wird ein beliebiges Objekt. Normalerweise werden für Exception eigene Klassen verwendet (z.B. Standardklasse exception)

In der Praxis wird zwischen logic_error und runtime_error unterschieden

Die Handler (Fänger) können Hierarchisch aufgebaut werden wichtig ist jedoch dass der Allgemeine am Schluss steht. Wenn kein passender Handler gefunden wird, wird die Funktion **terminate()** aufgerufen, welche das Programm beendet. Soll ein Handler alle Exceptions auffangen müssen drei Punkte (...) in der Klammer sein

```
catch (...)
{
}
}
```

Die Funktion **unexpected()** bricht das Programm ebenfalls ab

Always throw exceptions by value, and catch them by const reference. `catch (const Xcpt& e) {}`

Vereinzelte Programmiersprachen nutzen auch System Exception jedoch C++ nicht um die performance zu erhalten. Einzelne Systemumgebungen betreiben dennoch Exception Mapping in C++.

Preprocessor

Der Preprocessor wird vor der eigentlichen Übersetzung in Binärcode aktiviert, er führt textuelle Manipulationen von Source-Dateien durch (Makro-Substitution, Einfügen von Dateien und bedingte Übersetzung). Der Output des Preprocessors wird dem eigentlichen Compiler übergeben. Die Direktiven des Preprocessors beginnen immer mit **#**. Der Preprocessor ist zeilenorientiert.

- #define** Definition von Makros
Definition von Konstanten
Der Compiler führt eine reine Textersetzung durch
#define, haben keinen Scope
- #undef** Macht die Definition rückgängig.
- #include** Vollständiges Einfügen einer Datei
- #line** Direktes Setzen der Nummerierung von Sourcecode-Zeilen. Durch die optionale Angabe des Dateinamen lässt sich der Compiler einen neuen Dateinamen unterschreiben
- #error** Sofortiger Abbruch des Compile-Vorgangs und Ausgabe einer Fehlermeldung
- #pragma** Der Compiler ignoriert unbekannte #pragma-Direktiven. Jeder Compiler reagiert nur auf für ihn bestimmte Direktiven. Diese sind jedoch nicht portabel

Bedingungsanweisungen sind nach folgendem Schema aufgebaut

[Bedingungsprüfung]

Direktiven

beliebige viele #elif-Gruppen mit Direktiven

Optional ein #else mit Direktiven

#endif

// Mögliche Bedingungsprüfungen sind: #if #ifdef #ifndef

#if 0

// Auskommentieren des gesamten hier stehenden Codes.

#endif

Richtig eingesetzt kann der Preprocessor sehr nützlich sein, jedoch gibt es oft Alternativen, welche ebenso effizient jedoch viel sicherer sind. Deshalb sollten Preprocessor-Anweisungen zurückhalten eingesetzt werden.