

# Exception Handling

```

    elif operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    elif operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True

    #selection at the end -add back the deselected mirror modifier object
    mirror_ob.select= 1
    modifier_ob.select=1
    bpy.context.scene.objects.active = modifier_ob
    print("Selected" + str(modifier_ob)) # modifier ob is the active ob
    #mirror_ob.select = 0
    #new = bpy.context.selected_objects[0]
    #bpy.data.objects[new.name].select = 1
    #print("Selected" + str(new)) # new ob is the active ob

```

Handhabung von Situationen die normalerweise nicht vorgesehen sind

- Eingabe von unsinnigen Werte durch Anwender
- Verbindungsabbrüche
- Dateien können nicht geöffnet werden
- Speicherplatz kann nicht allokiert werden
- Durch den Programmierer nicht bedachte/getestete Fälle

Klassische Herangehensweise

- Funktionen liefern Fehlercodes (z.B. Rückgabewert)
- Wert nach „oben“ weiterreichen bis er behandelt wird

Code wird dadurch unübersichtlich

- Ständiger Test auf Fehlerfälle und notwendige Behandlung



# Besondere Fälle

Funktionen ohne Rückgabewerte

- Konstruktoren, Destruktoren

Fehler muss im Konstruktor erkannt und behandelt werden

- Konstruktor kennt aber den Kontext nicht
- Wie soll er den Fehler also behandeln?

Einziger Weg ist ein Programmende

- Auch nicht immer akzeptabel ...

Operatoren haben ein ähnliches Problem ...

```
Bruch::Bruch(int z, int n)
{
    if (n == 0)
    {
        cerr << "Fehler: Nenner 0" << endl;
        exit(1);
    }

    ...
}
```

# Exception Handling in C++

Bei ungeplanten Zuständen oder Fehlern wird eine sogenannte Exception („*Ausnahme*“) ausgelöst. Die Exception wird an zentraler Stelle abgefangen und behandelt.

C++	
Schlüsselwort	Verwendung
<code>try</code>	Definiert den Gültigkeitsbereich für eine Fehlerbehandlung.
<code>throw</code>	Löst eine Exception aus, auf die ein passender Exception Handler „ <i>anspricht</i> “.
<code>catch</code>	Definiert eine zentrale Fehlerbehandlungsroutine ( <b>Exception Handler</b> ) für einen definierten Gültigkeitsbereich.

# Beispiel (I)

`throw` generiert eine Instanz eines beliebigen Datentyps


Die Laufzeitumgebung unterbricht die Funktion an dieser Stelle

In der Funktionshierarchie wird ein Exception Handler gesucht, der diese Exception bearbeitet

- „Geordneter Rückzug“ – kein direkter Sprung
- In allen Blöcken werden die Destruktoren lokaler Objektinstanzen aufgerufen, sowie der belegte Speicher der lokalen Objekte freigegeben („**Stack Unwinding**“)

Mit `new` angelegte Objekte bleiben erhalten (!)

```
Bruch::Bruch(int z, int n)
{
    if (n == 0)
    {
        throw int(1);
    }
    ...
}
```



# Beispiel (II)

Für den durch `try` eingeschlossenen Block wird die durch das nachfolgende `catch` definierte Fehlerbehandlung installiert

**WICHTIG:** `catch` reagiert auf ein Objekt eines Datentyps (hier: `int`), aber nicht auf den Wert!

- `catch(const int)` wäre ausreichend gewesen
- Der Inhalt der Variablen `error` wird im Beispiel nicht verwendet (könnte aber!)

Mehrere Catch Blöcke sind möglich:

```
try {}  
catch {}  
catch {}  
...
```

Aber benötigen unterschiedliche Datentypen ...

```
...  
  
try  
{  
    Bruch a(1,0);  
  
    ...  
}  
  
catch (const int error)  
{  
    cerr << "ERROR: Nenner 0" << endl;  
    exit(1);  
}  
  
...
```

Typischerweise nutzt man Instanzen / Objekte von Klassen bzw. Fehlerklassenhierarchien

Minimalversion

- Klassen ohne Komponenten/Methoden
- Default Konstruktor

Fehlerklassen können aber auch Komponenten / Methoden beinhalten

- Z.B. zusätzliche Informationen über den Fehler oder das auslösende Objekt

Definition als Teile einer Klasse (z.B. Bruch) möglich

- Vermeidung von Namenskonflikten – Zugriff über Bereichsoperator

```
/* Fehlerklassen */
```

```
class eException {};
```

```
class eNennerNull : public eException {};
```

```
...
```

# Beispiel (I)

throw „*wirft*“ ein neu erzeugtes Objekt  
der Klasse eNennerNull

```
Bruch::Bruch(int z, int n)
{
    if (n == 0)
    {
        throw eNennerNull();
    }
    ...
}
```



# Beispiel (II)

`catch (const Bruch::eException)` fängt alle Exceptions ab, die von der Klasse `eException` abgeleitet sind

- Normale Vererbung (`is_a`)

Reihenfolge ist wichtig

- Mehrere `catch` Anweisungen werden von oben nach unten abgearbeitet
- `eNennerNull` wird zuerst abgefangen und ist damit abgearbeitet
- Das zweite `catch` bekommt nur die noch offenen Exceptions (soweit vorhanden)

```
...
try
{
    Bruch a(1,0);
}
catch (const Bruch::eNennerNull)
{
    cerr << "ERROR: Nenner 0" << endl;
    exit(1);
}
catch (const Bruch::eException)
{
    cerr << "ERROR: unbekannt" << endl;
    exit(1);
}
...
```

# Nicht aufgefangene Exceptions



Was passiert, wenn eine Exception ausgelöst wird, aber an keiner Stelle darauf reagiert wird?

- D.h. keine `try-catch` Sequenz vorhanden ist oder kein Exception Handler auf die Exception passt

Suche nach einem passenden Exception Handler wird so lange fortgesetzt bis man die `main()` Funktion verlassen hat („*Stack Unwinding*“)

Falls kein passender Exception Handler gefunden wird, wird eine Default Funktion aufgerufen, die das Programm beendet (`abort()`)

# Try & Catch – oder doch nicht?

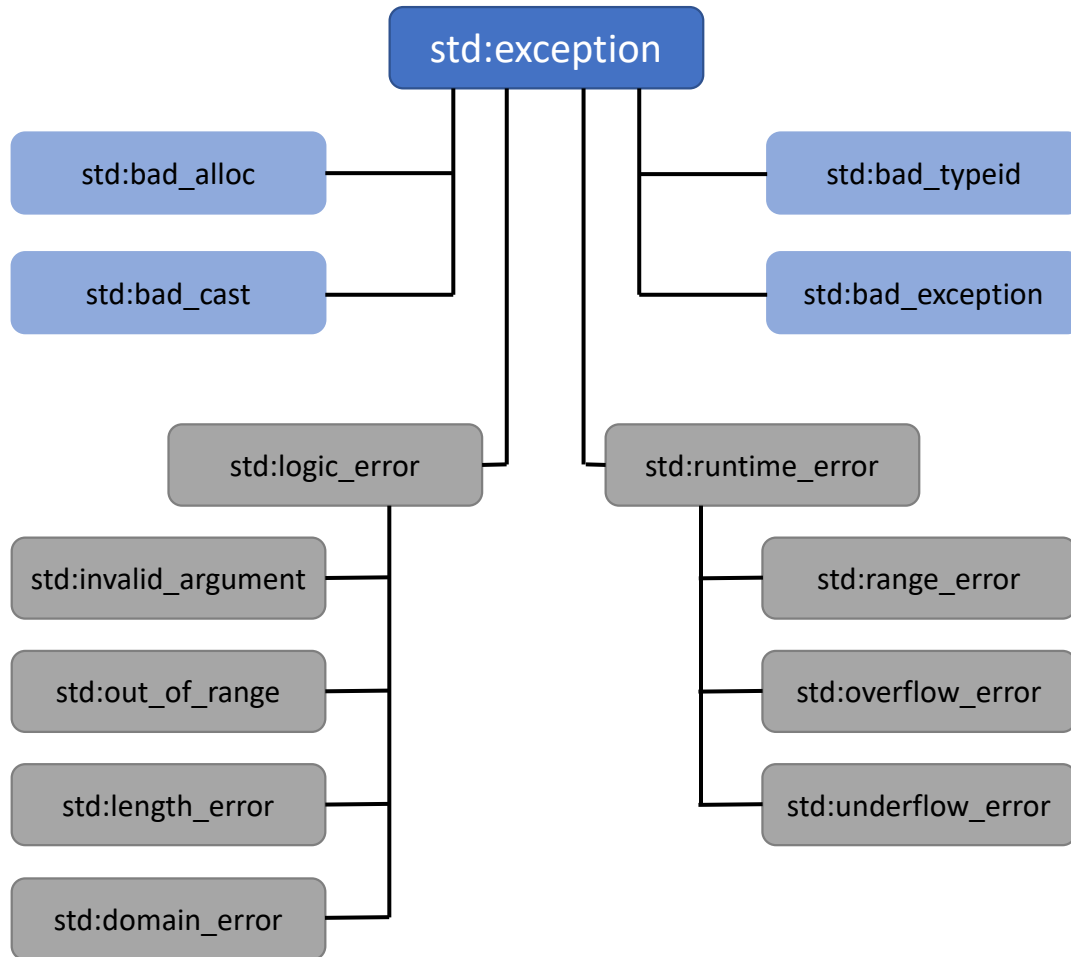


- Ein C++ Programm ohne (oder mit fast keinem) Exception Handling ist potentiell eine tickende Zeitbombe (Thema Konstruktoren)
- Ein Programm, in dem man vor lauter `try`'s den Quellcode nicht mehr erkennen kann, ist schlecht wartbar
- Gesamter Quelltext in einem riesigen `try`-Block ist auch nicht sinnvoll

## Empfehlungen:

- Gefährliche Stellen und Stellen mit zu erwartenden Fehlern in `try`-Blocks einfassen
- Mittels `catch` Block Exceptions beheben, im Zweifelsfall eine Fehlermeldung ausgeben
- In flachen Funktionshierarchien kann eine traditionelle Fehlerbehandlung immer noch die beste Methode sein

# C++ Standard Exceptions



Klassenhierarchie als Bestandteil der Standard Klassenbibliothek

- Benötigt: `#include <exception>`

Die Konstruktoren der grauen Klassen benötigen den Fehlertext als Argument

Beispiel:

- `throw out_of_range("ausserhalb");`

Klasse stellt virtuelle Funktion:  
`char* what()` – liefert Fehlertext

Kann auch als Basis für weitere Vererbung dienen (spezielle Exceptions für eigene Programme)

- `what()` Funktion kann überschrieben werden

- **Exception Handling** ist ein Sprachmittel zur Behandlung von Fehlern und Ausnahmen, die im Programmfluss auftreten.
  - Es kann insbesondere zur Fehlerbehandlung in Konstruktoren eingesetzt werden, da dabei keine Parameter oder Rückgabewerte verwendet werden.
- Tritt eine Exception auf, wird ein entsprechendes Objekt erzeugt und es werden alle übergeordneten Blöcke geordnet verlassen (**Stack Unwinding**), bis das Objekt zur Fehlerbehandlung empfangen und damit die Ausnahme behandelt wird.
  - Wird eine Exception nicht behandelt, ist das ein Programmfehler, der mit einem außergewöhnlichen Abbruch durch `abort()` behandelt wird.
- Typischerweise sind Exceptions Objekte, für die entsprechende Klassen deklariert werden.
  - Die Fehlerklassen können hierarchisch angeordnet werden, um allgemeine Fehlerarten durch spezielle Fehlerarten verfeinern oder verschiedene Fehlerarten zusammenfassen zu können.
- Die C++ Standardklassenbibliothek stellt eine Klassenhierarchie von Exceptions zur Verfügung die einfach verwendet und auch erweitert werden kann.



# Templates





# Typgebundene Variablen (I)

C/C++ verwenden typgebundene Variablen

Funktionen die für mehrere Typen funktionieren sollen, müssen dadurch mehrfach implementiert werden

```
int max(int, int)
{ ... }
```

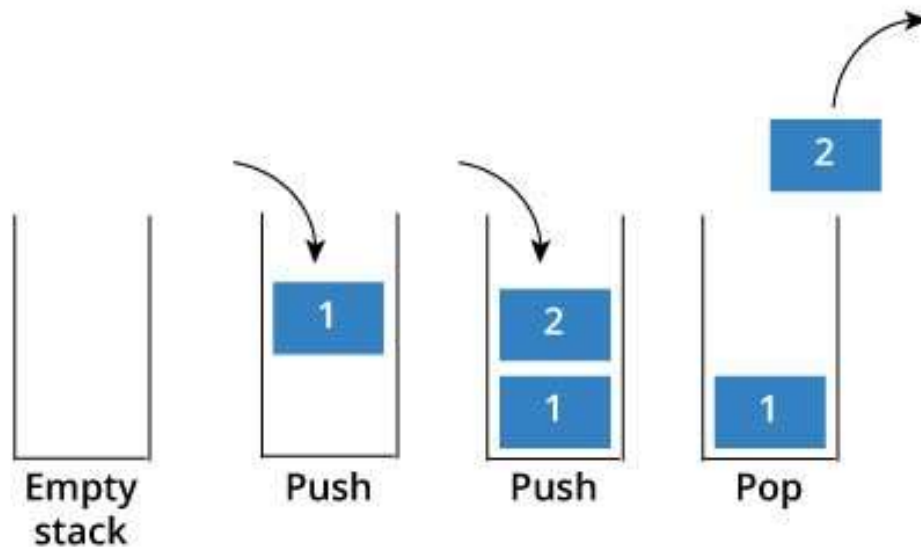
```
float max(float, float)
{ ... }
```

```
Bruch max(Bruch, Bruch)
{ ... }
```

Schlecht für Wartung – gleicher Fehler muss in mehreren Funktionen korrigiert werden



## Beispiel Verwaltung eines Stacks



**LIFO** = Last In First Out

**FILO** = First In Last Out

Gleiche Problematik bei Container Klassen

- **Container Klassen** verwalten Objekte beliebiger Klassen (z.B. verkettete Listen, assoziative Arrays, Stack, Bäume, etc.)

Für den Programmierer der Container Klasse sollte es uninteressant sein, für welche Objekte seine Klasse verwendet wird

Für den Anwender der Container Klasse haben die zu verwaltende Objekte die höchste Bedeutung

Ideal wäre, wenn man der Containerklasse einen Typ als Parameter übergeben könnte ...



# Templates (I)

**Templates** sind Schablonen

- Können für verschiedene Datentypen verwendet werden

Deklaration beginnt mit: `template <class ...>`

- `T` wird damit zum Typparameter erklärt
- Mehrere Typparameter sind möglich

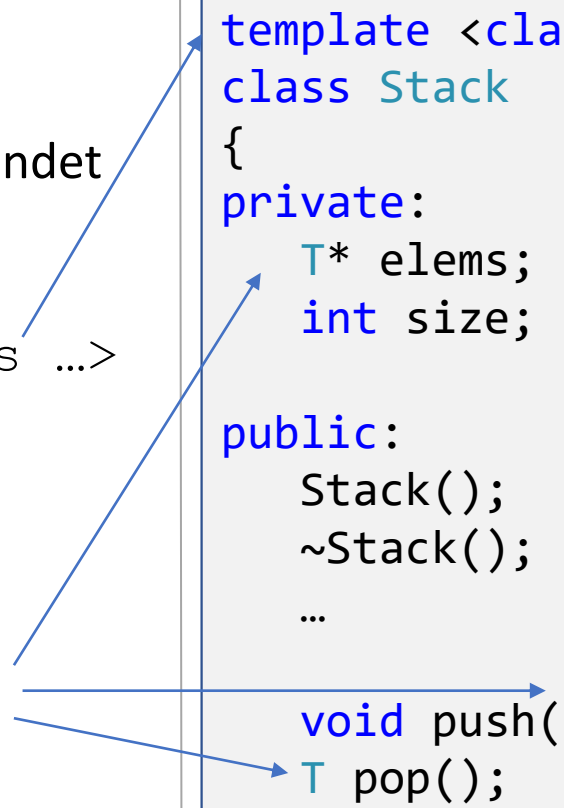
Verwendung des Typparameters an Stelle des „*unbekannten*“ Datentyps

Compiler verwendet das Template um bei Bedarf die notwendige Klasse/Funktion zu erzeugen

```
template <class T>
class Stack
{
private:
    T* elems;
    int size;

public:
    Stack();
    ~Stack();
    ...

    void push(T&);
    T pop();
    ...
};
```



# Templates (II)

Funktionsimplementierung beginnen ebenfalls mit  
`template <class T>` Deklaration

Normaler Zugriff auf die Attribute oder Elemente  
des Typparameters

Verwendete Operatoren müssen von dem  
„*unbekannten*“ Datentyp unterstützt werden

```
template <class T>
Stack<T>::Stack()
{
    numElems = 0; size 10;
    elems = new T[size];
}

...

template <class T>
T Stack<T>::pop()
{
    if(numElems == 0)
    {...}
    return(elems[--numElems]);
}
```

# Templates (III)

Bei Verwendung wird einfach der benötigte Typ eingesetzt

Objekt kann dann mit allen Komponenten/Methoden des Templates verwendet werden

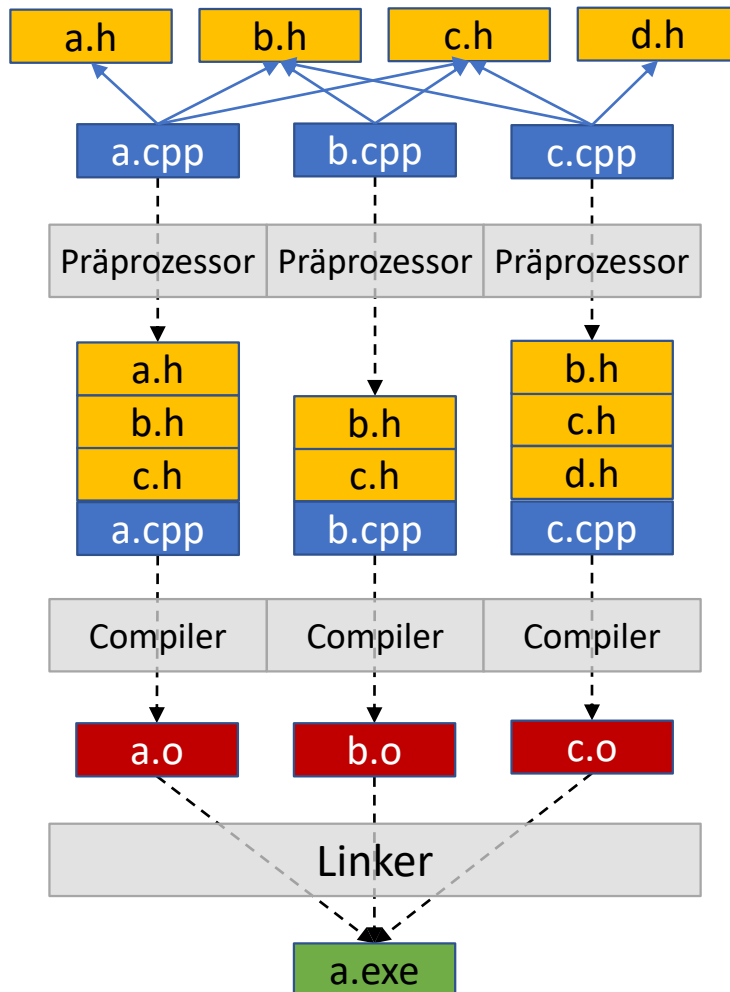
Zur Vereinfachung / besseren Lesbarkeit wird oft ein `typedef` verwendet

```
typedef Stack<int> intStack;
```

```
Stack<int> iStack;  
Stack<Bruch> bStack;
```

...

# Übersetzungsvorgang



## Mehrere Schritte

- Präprozessor

- Ersetzt die `#include` Anweisung einer Quellcode Datei durch die entsprechende Header Datei
- Präprozessor Direktiven werden ausgeführt und aufgelöst (`#define`, `#ifdef`, etc.)

- Compiler

- Übersetzt eine Quellcode Datei in (optimierte) Maschinsprache
- Resultat ist ein Object File - noch nicht ausführbar
- Zugriff auf Funktionen/Variablen aus anderen Object Files oder Standardbibliotheken sind nicht aufgelöst
- Compiler betrachtet immer nur eine (!) Quellcode (cpp) Datei

- Linker

- Verknüpft die verschiedenen Object Files untereinander und mit den verwendeten Standardbibliotheken – offene Referenzen werden dabei aufgelöst
- Generiert das Executable

## Üblicher Quellcode Struktur

- Je Klasse eine separate .cpp und .h Datei
- Also auch eine separate .cpp und .h Datei für die Template Klasse
- Eine main.cpp Datei (in der `Stack<int>` verwendet wird)

Compiler läuft für alle .cpp ohne Fehler durch

Aber Linker meldet:

***cannot find Stack<int>::Stack<int>()***

Er findet den Konstruktor nicht ...

## Was passiert hier?

- Compiler übersetzt die main Datei
  - Sieht die Verwendung von `Stack<int>`
  - Muss einen Konstruktor aufrufen - hat selbst aber keinen C++ Code dazu
  - Generiert offene Referenz, die der Linker aufzulösen hat (wie bei `strcpy()` oder anderen Funktionen aus Bibliotheken)
- Compiler übersetzt die cpp Datei des Templates
  - Keine Information über den benötigten Typ
  - Daher wird nur wenig Maschinencode erzeugt (nur generische Teile – aber kein Konstruktor)
- Linker findet keinen Code für die offene Referenz
  - Code wurde ja auch an keiner Stelle vom Compiler erzeugt

## Inklusion

- Template Methoden, die abhängig vom verwendeten Typ sind, kommen in die Header Datei
  - Nur Template unabhängiger Code verbleibt in der cpp Datei
  - Potentiell ist die cpp Datei leer und kann entfernt werden
- Compiler hat damit bei jeder Verwendung des Templates die notwendigen Funktionsdefinitionen zur Verfügung
  - Entsprechend wird der Typ spezifische Code generiert
- Linker sortiert potentiell mehrfach generierten Code aus
  - Für jede cpp Datei, die den gleichen Typ verwendet, generiert der Compiler die gleichen Methoden

### Vorteil

- Einfach zu handhaben und flexibel

### Nachteil

- Lange Übersetzungszeiten in großen Projekten

## Explizite Instanziierung

- Erfordert: die benötigten Typen sind dem Programmierer des Templates bekannt
- Üblicher Setup mit cpp und h Datei für das Template
- Instanziierung für die benötigten Datentypen am Ende der Template cpp Datei mit der folgenden Anweisung

```
template Stack<int>;
```

- Compiler erzeugt damit bei der Übersetzung der Template cpp Datei den entsprechenden Code für diesen Typ
- Linker findet damit Code für die notwendigen Methoden

### Vorteil

- Schnellere Übersetzungszeiten

### Nachteil

- Nutzung nur noch für die instanziierten Typen möglich

# Templates für Funktionen

Templates sind auch für einzelne Funktionen möglich

- Unabhängig von Klassen

Compiler erzeugt bei Verwendung mit einem konkreten Typ automatisch Maschinencode

Thematik Inklusion vs. Explizite Instanziierung ist identisch zu Klassentemplates

```
template <class T>
const T& maxWert(const T& a, const T& b)
{
    return(a > b ? a : b);
}

...

void vergleich()
{
    int x, y, z;
    Bruch a, b, c;

    x = maxWert(y, z);
    a = maxWert(b, c);
    ...
}
```

- Funktionen und Klassen können als **Templates** (Schablonen) für noch nicht festgelegte Typen definiert werden.
- Durch Aufruf oder Deklaration eines Templates mit konkreten Typen werden Templates dann als Code realisiert und können für die Typen verwendet werden.
  - Dabei wird für jeden Type, für den ein Template verwendet wird, eigener Code generiert.
- Mit Template-Klassen können insbesondere **Containerklassen** in Bezug auf den Typ der verwalteten Objekte parametrisiert werden.
- Bezüglich der Quellcode Struktur kommt dabei entweder das Verfahren **Inklusion** oder die **Explizierte Instanziierung** zum Einsatz.





# Klassenbibliothek

```
elif operation == "MIRROR_Z":
    mirror_xsd.use_x = False
    mirror_xsd.use_y = False
    mirror_xsd.use_z = True
```

```

#selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob

```

Mehrere aufeinander abgestimmte Klassen, die in einer oder mehreren Hierarchien implementiert sind



Vorteil Wiederverwendbarkeit

- Für die gleiche Funktionalität können Klassen mehrfach verwendet werden
- Klassen mit ähnlicher Funktionalität können durch Vererbung so implementiert werden, dass gleiche Teile nur einmal implementiert werden müssen
- Weniger Quellcode – bessere Wartbarkeit
- Bibliothek Code schon getestet – weniger fehleranfällig

## MYTHOS

*Verwendung einer objektorientierten Sprache bedeutet automatisch Wiederverwendbarkeit.*

## REALITÄT

*Wiederverwendbarkeit erfordert Arbeit und muss bereits beim Design beachtet werden!*

Entwurf einer wiederverwendbaren Klasse / Bibliothek ist schwieriger als der Entwurf eines normalen Programms

Programm	Bibliothek
Lösung eines besonderen Problems in einem besonderen Kontext	Lösung für ein Set von Problemen, die in einer Vielzahl von Projekten anzutreffen sind
Genaue Annahmen über die Umgebung möglich	Muss in einer Vielzahl von Kontexten arbeitsfähig sein ohne die Umgebung genau zu kennen

# C++ Standard Klassenbibliothek

Standardisierung durch das ANSI/ISO Komitee

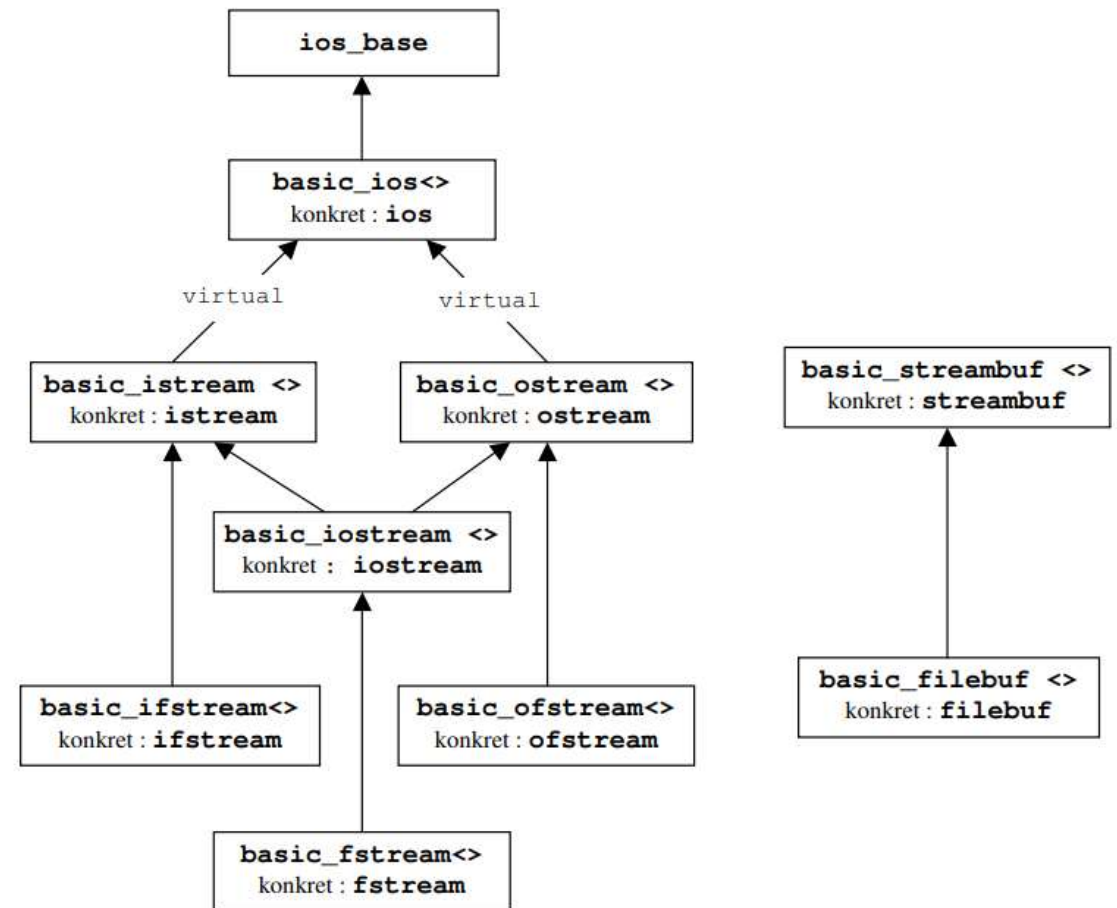
Basiert u.a. auf der Standard Template Library (STL)  
von Hewlett-Packard

Sammlung von

- Generischen Containern (Behälterklassen)
- Generische Zeichenketten (Strings)
- Eingabe und Ausgabe (Streams)
- Ausnahmen (Exceptions)
- ...

Implementiert mit Hilfe der C++ Sprache

Namespace: std



# C++ Standard - Container

Verwaltung anderer Datentypen und Objekte

Formuliert als Template Klassen

Zugriff über Methoden und Iteratoren

Der Container „*besitzt*“ die Elemente

- Lebenszeit des gespeicherten Objekts endet mit dem Container

Sequentielle Container	
vector	Felder dynamischer Größe
array	Felder fester Größe
list	Doppelt verkettete Listen
forward_list	Einfach verkettete Listen
queue	Warteschlangen
deque	Warteschlangen mit 2 Enden
priority_queue	Warteschlangen mit Prioritäten
stack	Stapel
Geordnete assoziative Container	
set / multi_set	Mengen
map / multi_map	Assoziative Felder
Ungeordnete assoziative Container (Hashmaps/Hashsets)	
unordered_set / unordered_multiset	Mengen
unordered_map / unordered_multimap	Assoziative Felder

## „Intelligente“ Zeiger

- Zugriff auf Elemente eines Containers
- Iteration über die Elemente eines Containers

## Implementierung

- Überladen von Operatoren (\*, ++, ==)
- Zusätzliche Methoden

Iterator Klassen werden durch den jeweiligen Container bereitgestellt

Nicht jeder Container stellt alle Iteratoren zur Verfügung

Iteratoren	Verwendung
<i>Eingabe-Iteratoren</i>	Lesender Zugriff für einen einzelnen Durchlauf
<i>Ausgabe-Iteratoren</i>	schreibender Zugriff für einen einzelnen Durchlauf
<i>Forward-Iteratoren</i>	sequenzieller Zugriff mit relativem Bezug auf Iteratoren, in eine Richtung
<i>Bidirektionale Iteratoren</i>	wie Forward-Iteratoren jedoch in beide Richtungen
<i>Iteratoren mit wahlfreiem Zugang</i>	wahlfreier Zugriff, auch mit Index-Operator ([])



## *„Funktionen mit Manipulationsvorschriften“*

- Werden auf Container angewendet
- Zugriff auf den Container über Iteratoren

Enthalten Standardalgorithmen der Informatik

- Sortieralgorithmen
- Erzeugung von Zufallszahlen
- ...

Algorithmen	Verwendung
<code>for_each</code>	Wendet eine Operation auf alle Elemente eines Datensatzes an
<code>transform</code>	transformiert einen Datensatz mit einer Funktion in einen anderen
<code>copy</code>	kopiert den Datensatz in einen anderen
<code>sort</code>	sortiert den Datensatz
<code>find</code>	sucht nach einem bestimmten Element in einem Datensatz
<code>search</code>	sucht nach einer Elementreihe in einem Datensatz
...	



- Microsoft Foundation Classes (MFC)
  - Entwicklung von grafischen Benutzeroberflächen (Windows)
- Framework Class Library (FCL)
  - .NET/Common Language Infrastructure (CLI) Standard Libraries
- Qt
  - Plattformübergreifenden Entwicklung von Programmen und grafischen Benutzeroberflächen
  - Umfangreiche Funktionen zur Internationalisierung, Datenbankfunktionen und XML-Unterstützung
  - Erhältlich für eine große Zahl an Betriebssystemen bzw. Grafikplattformen wie X11 (Unix-Derivate), macOS, Windows, iOS und Android

Und viele andere ...



- Wiederverwendbarkeit ist kein Automatismus, der durch die Verwendung einer objektorientierten Sprache entsteht.
  - Wiederverwendbarkeit erfordert Arbeit und muss bereits beim Design berücksichtigt werden.
- Es gibt verschiedene **Klassenbibliotheken**, die die Anwendungsentwicklung unterstützen können.
  - Die C++ Standard Klassenbibliothek ist Teil des C++ Compilers.
  - Andere Klassenbibliotheken unterstützen spezielle Anwendungsgebiete (Windows, .NET, Qt, etc.).



# Abschluss

```
elif operation == "Mirror_Y":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = True  
    mirror_mod.use_z = False  
elif operation == "Mirror_Z":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = False  
    mirror_mod.use_z = True  
  
#selection at the end -add back the deselected mirror modifier object  
mirror_ob.select= 1  
modifier_ob.select=1  
bpy.context.scene.objects.active = modifier_ob  
print("Selected" + str(modifier_ob)) # modifier ob is the active ob  
#mirror_ob.select = 0  
#new = bpy.context.selected_objects[0]  
#bpy.data.objects[new.name].select = 1  
#print("Selected" + str(new) + " is the active ob")
```

# Das Ende der Vorlesung ...

## ■ Grundlagen der objektorientierten Programmierung

- Paradigma und Grundbegriffe (Objekte, Klassen, Vererbung, Polymorphismus)
- UML / Objektorientierter Entwurf

## ■ Objektorientierte Programmierung in C++

- Von C zu C++
- Die Grundbausteine von C++
- Klassen, Objekte, Datenkapselung
- Vererbung
- Polymorphismus
- Exception Handling
- Templates
- Klassenbibliothek



# ... aber der Beginn für mehr!



- Weitere C++ Sprachelemente
  - Inline Funktionen zur Laufzeitverbesserung
  - Friend Funktionen als Hilfsmittel bei globalen Überladen
  - Statische Klassenkomponenten
  - Weitergehende Nutzung von Initialisierungslisten
  - Weitergabe von Exceptions
  - Zusätzliche Parameter für Templates
  - ...
- Klassenbibliotheken
  - Elemente der Standardklassenbibliothek
  - Windows Klassenbibliothek
  - ...



```
elif operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
elif operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

#selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob
#mirror_ob.select = 0
#new = bpy.context.selected_objects[0]
#new.data.objects[mirror_ob.name].select = 0
```