Themen

- C++ type casting
  - static_cast
  - dynamic_cast
  - const_cast
  - reinterpret_cast
- Ausnahmen - Exceptions
  - try...catch
  - throw
- Default-Argumente bei Funktionen

C typecasting

```
char a='B';
cout << a << endl;
cout << (int)a << endl;
```

C typecasting und const

```
const Bruch* bruch=new Bruch(1,5);
bruch->setNenner(6); // FEHLER
((Bruch*)bruch)->setNenner();
```

C typecasting

```
const Bruch* bruch=new Bruch(1,5);
const Quadrat* quadrat=
      (const Quadrat*)bruch;
quadrat->getUmfang(); // FEHLER
```

C typecasting

- ist gefährlich, weil man die Regeln kennen muss, nach denen der Compiler die Daten ineinander umwandelt
- sollte man deshalb sehr sparsam verwenden
- benutzerdefinierte Datentypen werden über Konstruktoren ineinander überführt

Konstruktor Beispiel

```
class A
{
   A(const B& b);
};

B b;
A a1=(A)b;
A a2=A(b);
```

Konstruktor Beispiel 2

```
void print(string s)
{
   cout << s;
}

const char* str="Test";
print(str);
```

Konstruktor Beispiel 2

```
class string
{
  string(const char* str);
};

string s1=”Test”;
string s2=string(“Test”);
```

C typecasting / Konstruktorschreibweise

```
char a='\n';
cout << a << endl;
cout << int(a) << endl;
```

C typecasting / Konstruktorschreibweise

- ist für einfache Datentypen und Instanzen dasselbe wie die andere Schreibweise

```
char c='w';
int a=int(c); // == int a=(int)c;
```

- nicht anwendbar auf Zeiger und Referenzen

```
A* a=new A();
B* b=B*(a);        // FALSCH
```

C++ typecasting Operatoren

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

Warum C++ Casts

- prüfen teilweise die Ziel- und Quelldatentypen, um Fehler zu erkennen und zu vermeiden
- geben dem Programmierer die Möglichkeit genauer auszudrücken, auf welche Art die Daten umgewandelt werden sollen
- sind im Quellcode leichter zu erkennen und zu finden

static_cast<A>(B)

- Dient dazu verwandte Datentypen ineinander umzuwandeln
- Der Compiler überprüft bereits beim Compilieren, ob die beiden Datentypen verwandt sind
- sind die Datentypen nicht verwandt, gibt der Compiler einen Fehler aus

static_cast Beispiel 1/3

```
class A{ ... };
class B : public A { ... };
class C { ... };

A* a=new A();
B* b=new B();
C* c=new C();

A* x=static_cast<A*>(b);
// identisch zu A* x=(A*)b;
```

static_cast Beispiel 2/3

```
class A{ ... };
class B : public A { ... };
class C { ... };

A* a=new A();
B* b=new B();
C* c=new C();

B* x = static_cast<B*>(a);
// identisch zu B* x=(B*)a;
// syntaktisch korrekt, aber a ist
// keine Instanz vom Typ B !!!
```

static_cast Beispiel 3/3

```
class A{ ... };
class B : public A { ... };
class C { ... };

A* a=new A();
B* b=new B();
C* c=new C();

C* x=static_cast<C*>(a); // ERROR
```

static_cast vs. C cast

```
int i=5;
int* pI=&i;
float f;
float* pF;

f=(float)i;
f=static_cast<float>(i);
pF=(float*)pI;
pF=static_cast<float*>(pI); // Fehler
```

dynamic_cast<A>(B)

- dient dazu, Zeiger auf eine Basisklasse auf Zeiger einer abgeleiteten Klasse umzuwandeln
- Compiler überprüft zur Laufzeit ob B eine vollständige Instanz von A ist.
- anwendbar auf Zeiger und Referenzen
- nur anwendbar auf polymorphe Klassen (Klassen mit virtuellen Methoden)
- im Fehlerfall wird bei Zeigern 0 zurückgegeben
- bei Referenzen wird im Fehlerfall eine Ausnahme (Exception) ausgelöst

dynamic_cast Beispiel 1/3

```
class A{ ... };
class B : public A { ... };
class C { ... };

A* a=new A();
A* b=new B();
C* c=new C();

B* x=dynamic_cast<B*>(b); // x=b;
```

dynamic_cast Beispiel 2/3

```
class A{ ... };
class B : public A { ... };
class C { ... };

A* a=new A();
A* b=new B();
C* c=new C();

B* x=dynamic_cast<B*>(a); // x=0
```

dynamic_cast Beispiel 3/3

```
class A{ ... };
class B : public A { ... };
class C { ... };

A* a=new A();
A* b=new B();
C* c=new C();

C* c=dynamic_cast<C*>(a); // c=0;
```

const_cast<A>(const A)

- dient vor allem dazu ein konstantes (const) Objekt in ein nicht konstantes Objekt umzuwandeln (bzw. umgekehrt)
- anwendbar auf Zeiger und Referenzen

const_cast Beispiel

```
void print(char* c)
{
   cout << c;
}

const char* str="Test\n";
print(str); // FEHLER
print( const_cast<char*>(str) );
```

reinterpret_cast<A>(B)

- dient dazu den Inhalt eines Speicherbereiches neu zu interpretieren
- meist nicht portabel
- sehr gefährlich!!! - Uminterpretierung ohne Prüfung
- nur in Ausnahmefällen verwenden!!!

reinterpret_cast Beispiel

```
float f=34.5f;
unsigned int i,iMax=sizeof(f);
unsigned char* c=
  reinterpret_cast<unsigned char*>(f);

for(i=0;i<iMax;++i)
{
  cout << (int)c[i];
}
// Ausgabe von Architektur abhängig
```

Ausnahmen - Exceptions

- dienen dazu dem laufenden Programm anzuzeigen, dass ein (nicht kritischer) Fehler aufgetreten ist und gibt dem Programm die Gelegenheit auf diesen Fehler zu reagieren
- viele Methoden der C++ Bibliothek und der Standard Template Library (STL) benutzen Ausnahmen, um Fehler anzuzeigen oder den Benuzer darauf aufmerksam zu machen, dass eine Funktion nicht erfolgreich ausgeführt werden konnte

Wo treten Exceptions auf?

- anfordern von Speicher
- dynamic_cast von Referenzen
- arbeiten mit Streams
- Standard Template Library (STL)
- ...

Behandeln von Exceptions

- Exceptions werden mit **`try...catch`** behandelt
- der Codeblock nach der **`try`** Anweisung enthält die Anweisungen, die zu einer Exception führen können
- die **`catch`** Anweisung enthält den Code zur Fehlerbehandlung für eine mehr oder weniger spezifische Exception
- Jede **`try`** Anweisung kann eine oder mehrere **`catch`** Anweisungen besitzen

Exception Behandlung Beispiel

```
try
{
   unsigned int* t=new int[2000000000];
}
catch(std::bad_alloc e)
{
    // do something
}
```

Exception Behandlung Beispiel

```
try{
   // mach was
}
catch(std::bad_alloc e){
    // nicht genug Speicher
}
catch(std::exception e){
   // etwas Anderes ist schief
   // gegangen
}
```

Exceptions

- jede Exception, die nicht durch eine **`catch`** Anweisung abgefangen wird, wird an die nächst höhere Anweisungsschicht weitergereicht
- Exceptions, die nirgends behandelt werden, führen zum Abbruch des Programmes
- die **`catch(...)`** Anweisung fängt alle Exceptions ab

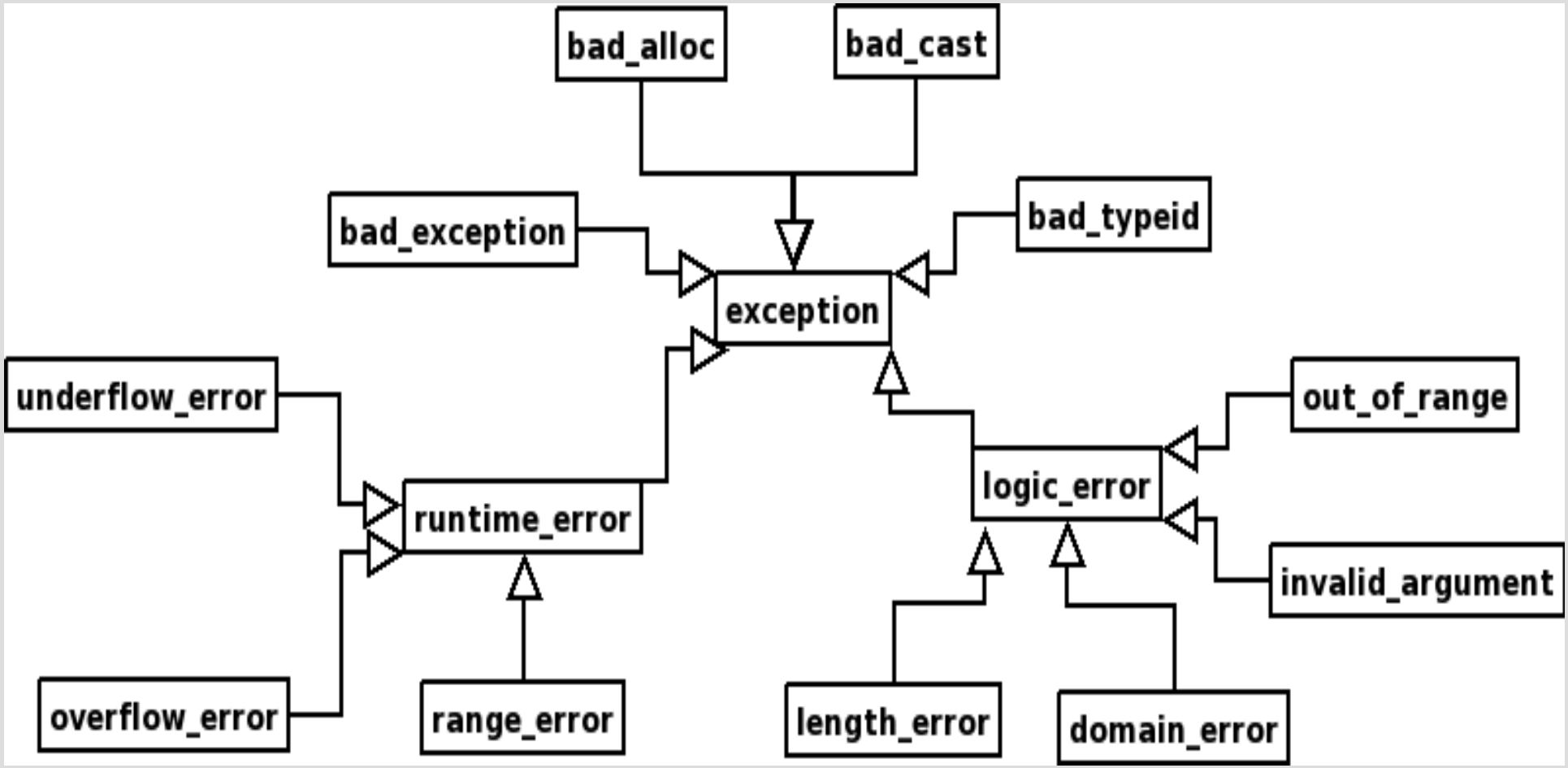
Exception Behandlung Beispiel

```
try{
   // mach was
}
catch(...){
    // Upps, schnell aufräumen
}
```

Exceptions auslösen

- Exceptions werden mit der **`throw`** Anweisung ausgelöst.
- Als Argument bekommt die **`throw`** Anweisung ein beliebiges Objekt (auch einfache Datentypen wie **`int`**, **`double`**, etc)
- normalerweise sollte das Objekt Informationen über den Fehler liefern
- <stdexcept> definiert die Klasse std::exception und einige davon abgeleitete Klassen

<stdexcept>


bad_alloc
bad_cast
bad_exception
bad_typeid
exception
underflow_error
out_of_range
logic_error
runtime_error
invalid_argument
overflow_error
range_error
length_error
domain_error

throw Beispiel

```
try
{
    throw Bruch(3,4);
}
catch(Bruch& b)
{
    // Fehlerbehandlung
}
```

throw Beispiel

```
try
{
   throw
    std::out_of_range("Wert zu
 niedrig.");
}
catch(std::out_of_range& e)
{
   // Fehlerbehandlung
}
```

throw ohne Argument

- dient dazu die ursprüngliche Exception erneut auszulösen
- wird verwendet, wenn der Exception Handler Code enthält, um einen bestimmten Status wieder herzustellen, aber nicht adequat auf die Exception reagieren kann

throw Beispiel

```
try{
   // mach was
}
catch(...){
    // Upps, schnell aufräumen
    throw;  // Exception
            // weiterreichen
}
```

throw in der Funktionsdeklaration

- mit **`throw`** kann man angeben, welche Exceptions eine Funktion auslösen kann

```
void run() throw
    (bad_alloc,bad_cast);
```

`run` löst nur Exceptions vom Typ `bad_alloc` oder `bad_cast` aus oder Exceptions, die von diesen abgeleitet sind

- Die Definition der Funktion muss dieselbe **`throw`** Anweisung enthalten

throw in der Funktionsdeklaration

- löst eine Funktion eine andere als die angegebenen Exceptions aus, so wird `std::unexpected()` aufgerufen, welche das Programm beendet.
- eine Funktion ohne `throw` Anweisung in der Deklaration darf beliebige Exceptions auslösen
- eine Funktion, welche keine Exception auslösen darf, enthält eine leere Liste von erlaubten Exceptions

```
void run throw ();
```

throw in Funktionsdeklarationen

- wird eine Methode in einer abgeleiteten Klasse überschrieben, welche in der Basisklasse eine Liste von erlaubten Exceptions angibt, so kann die Methode in der abgeleiteten Klasse lediglich dieselben, oder ein Teilmenge davon auslösen

Exceptionlisten in abgeleiteten Klassen

```
class A{
    void run() throw(bad_cast, bad_alloc);
};

class B : public A{
    void run(); // FEHLER
};
```

Exceptionlisten in abgeleiteten Klassen

```
class A{
void run()
    throw(bad_cast, bad_alloc);
};

class B : public A{
void run() throw (bad_cast); // OK
};
```

Default Argumente

- C++ erlaubt es Default Argumente bei der Deklaration einer Funktion/Methode anzugeben
- diese Argumente müssen beim Benutzen der Funktion nicht angegeben werden, der Compiler ersetzt fehlenden Argumente durch die in der Deklaration angegebenen Default-Werte
- die Default-Werte werden bei der Definition der Funktion nicht angegeben

Beispiel – Default Argumente

```
unsigned int fakultaet
                (unsigned int n=4);
...
unsigned int f1=fakultaet(3);
cout << f1 << endl;
f1=fakultaet();
cout  << f1 << endl;
```

```
6
24
```

Default Argumente

- Default-Argumente einer Funktion müssen immer am Ende der Funktionsargumente stehen

```
double power(double exponent=2,
    double radikand); // FALSCH
```

- Durch Default-Argumente kann es sein, dass ein Funktionsaufruf nicht mehr eindeutig ist

Default Argumente

```
unsigned int fakultaet(unsigned int n=4);

unsigned int fakultaet();

          ...

unsigned int i=fakultaet(); // ???
```