

Kapitel 22

Templates und Exceptions

Die Funktion `swap` zum Vertauschen übergebener Parameter

Im Laufe der Veranstaltung haben wir eine Funktion `swap` verwendet, die für das Vertauschen von Parametern des Typs `int&` eingeführt worden war:

```
void swap( int& a, int& b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;
    // ...
    printf( "Vorher; %d %d\n", x, y);

    swap( x, y);

    printf( "Nachher; %d %d\n", x, y);
    return 0;
}
```

```
Vorher: 1 2
Nachher: 2 1
```

Die Signatur der Funktion schränkt die Verwendung der Funktion auf Variablen des Typs `int` ein, auch wenn die praktisch identische Nutzung auch für andere Datentypen hilfreich wäre.

Funktionen generisch nutzbar machen

Wenn wir die Funktion für andere Datentypen verwenden wollen, sind wir bisher gezwungen, spezifischen Funktionen mit einer passenden Signatur und gleicher Funktionalität neu zu implementieren.

```
void swap( long& a, long& b )
{
    long temp = a;
    a = b;
    b = temp;
}

void swap( double& a, double& b )
{
    double temp = a;
    a = b;
    b = temp;
}

void swap( datum& a, datum& b )
{
    datum temp = a;
    a = b;
    b = temp;
}
```

Die Funktionen unterscheiden sich nur durch den verwendeten Datentyp, haben aber die gleiche Funktionalität

Das implementierte Verhalten ist in allen drei Varianten identisch, sowohl für die elementaren Datentypen `long` und `double` als auch für eine selbst erstellte Datenstruktur wie `datum`.

Funktionen generisch nutzbar machen

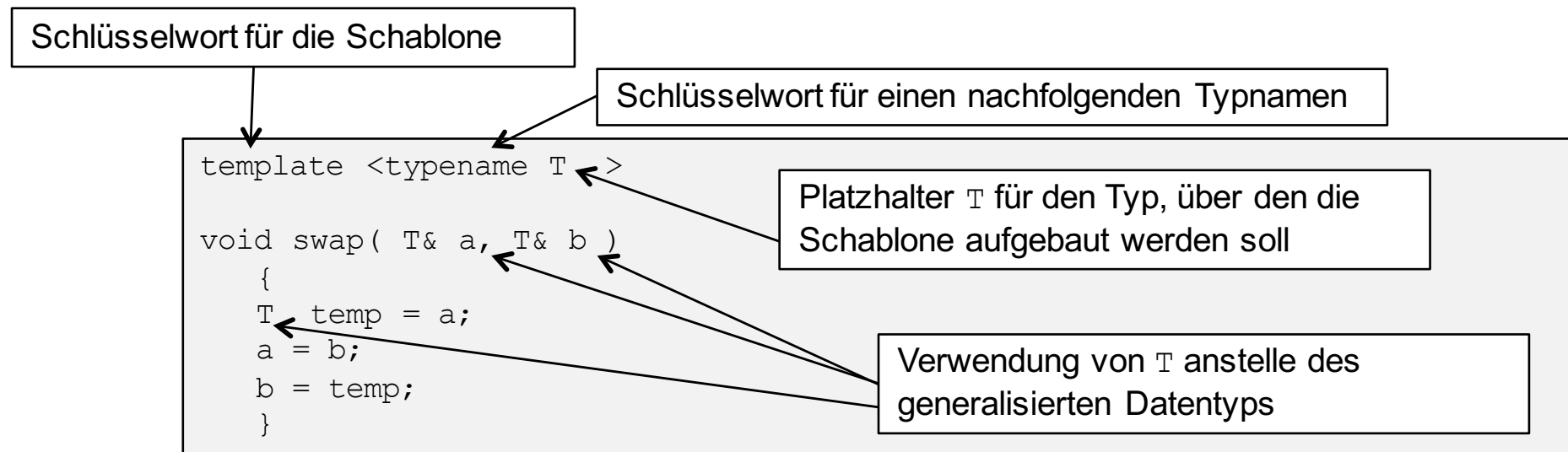
Sie haben bereits eine mögliche Lösung solcher Aufgaben unter Verwendung von Präprozessor-Makros kennengelernt, haben aber auch gesehen, wie fehlerträchtig diese Lösungen sein können. C++ bietet als typischere Alternative sogenannte Templates an. Diese Templates werden auch Vorlagen oder Schablonen genannt.

Mit den Templates kann der Compiler Funktionen und Klassen für praktisch beliebige Datentypen nach Vorgaben generieren.

Die Schablone selbst ist keine Klasse oder Funktion, sondern eine Hülle, aus der später im Generierungsprozess ein entsprechendes Element entsteht. Wir werden die gerade gezeigten Varianten von `swap` jetzt durch ein einzelnes Template ersetzen.

Anlegen eines Templates

Zur Erstellung eines Templates wird dem Compiler zuerst angezeigt, dass die folgende Funktion als Schablone verwendet werden soll:



Das Schlüsselwort `template` weist darauf hin, dass es sich nicht um fertigen Code sondern um eine Schablone handelt. Diese Schablone soll über den Typ mit dem Namen `T` aufgebaut werden. `T` steht hier als Platzhalter für den noch unbekannten, später anzugebenden Datentypen. Es folgt die eigentliche Schablone der Funktion `swap`. Die Funktion erhält als Parameter zwei Referenzen auf Variablen des Datentyps `T`. Innerhalb des Funktionsblocks wird die Funktion definiert, ebenfalls unter Zuhilfenahme des Platzhalters `T` für den Datentyp der temporären Variablen.

Verwenden des Templates

Mit der Erstellung der Schablone wird noch kein Code kompiliert. Erst wenn eine Template-Funktion erstmalig für einen bestimmten Datentyp aufgerufen wird, generiert der Compiler aus dem Template eine echte Funktion und übersetzt sie.

Zur Verwendung des Templates geben wir an, für welchen Datentyp die Funktion generiert werden soll. Vor der Kompilierung wird der Platzhalter dann durch den tatsächlichen Datentyp ersetzt:

```
template <typename T>
void swap( T& a, T& b )
{
    T temp = a;
    a = b;
    b = temp;
}

int main()
{
    double wert1 = 1.0, wert2 = 2.0;

    swap<double>( wert1, wert2 );

    printf( "Nachher; %lf %lf\n", wert1, wert2 );
    return 0;
}
```

Definition des Templates

Explizite Angabe des Datentyps über den das Template gebaut werden soll.

Aufruf der Funktion mit passenden Datentypen

Nachher; 2.000000 1.000000

Implizite Typisierung

Neben der expliziten Angabe des Datentyps kann der Compiler in vielen Fällen den benötigten Datentyp auch implizit anhand der übergebenen Argumente erkennen:

Wir haben damit eine generisch einsetzbare Funktion `swap`, die wir für alle Datentypen verwenden können, die die Zuweisung unterstützen.

Explizite Typisierung

Implizite Typisierung

```
template <typename T>
void swap( T& a, T& b )
{
    T temp = a;
    a = b;
    b = temp;
}

int main()
{
    double wert1 = 1.0, wert2 = 2.0;
    swap<double>( wert1, wert2 );

    punkt p1; punkt p2;
    swap<punkt>( p1, p2 );

    int x1 = 2, x2 = 2;
    swap( x1, x2 );

    datum d1; datum d2;
    swap( d1, d2 );
    return 0;
}
```

Ein einfacher, typisierter Stack

Im ersten Beispiel ist eine isolierte Funktion als Template bereitgestellt worden. Als Beispiel für die Verwendung einer Klasse verwenden wir einen einfachen Stack.

Ausgangspunkt für das Template ist eine einfache Stack-Klasse, die bis zu 100 Datenelemente vom Typ `int` speichern kann:

```
class intStack
{
private:
    int stck[100];
    int top;

public:
    intStack() { top = 0; }
    int push( int element );
    int pop();
    int isEmpty(){ return top == 0; }
};
```

Array mit festem Typ
und fester Größe als
Datenspeicher

Die unterstützten
Operationen der Stack-
Klasse und deren
Implementierung

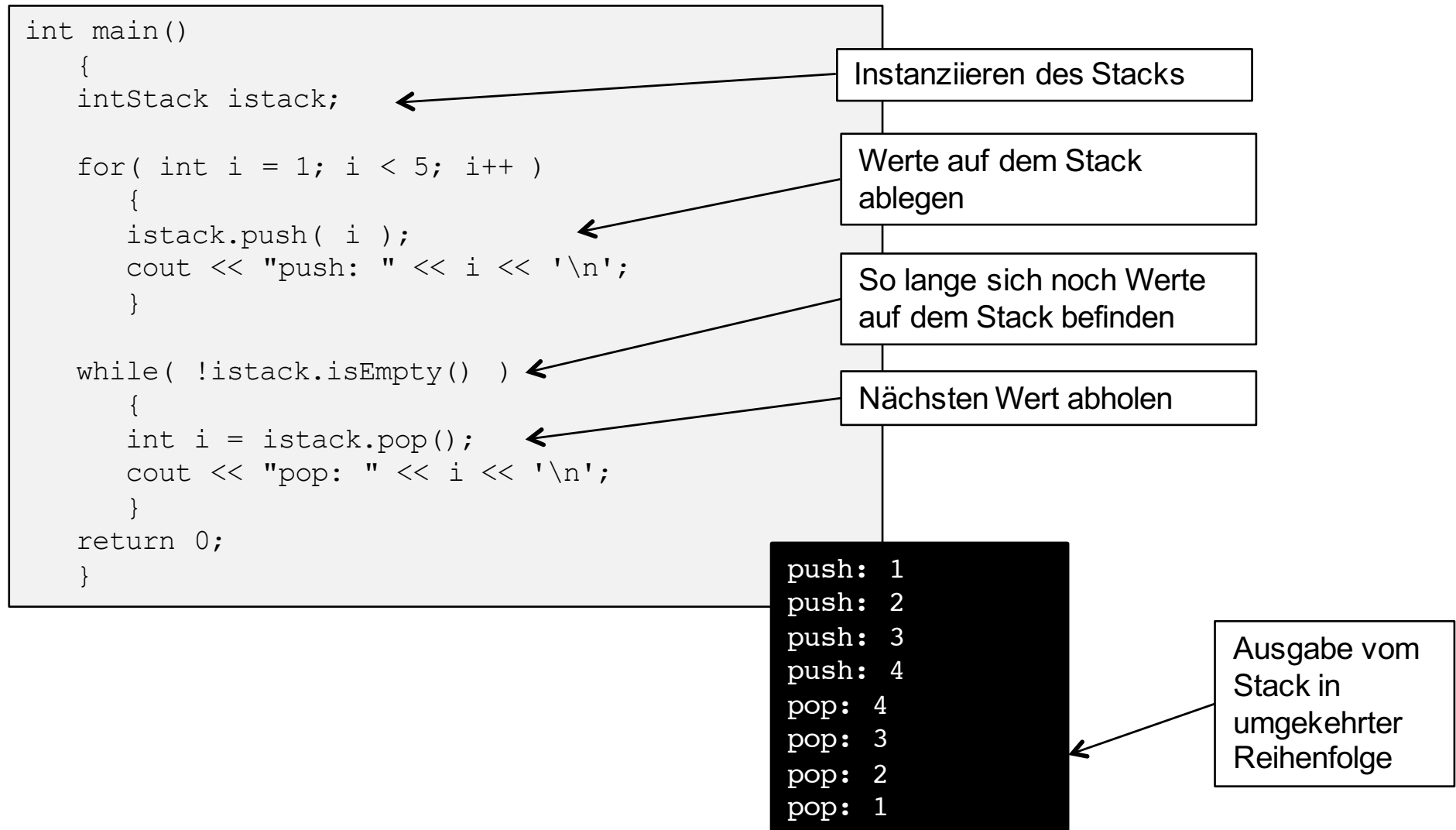
```
int intStack::push( int element )
{
    if( top < 100 )
    {
        stck[top] = element;
        top++;
        return 1;
    }
    return 0;
}
```

Limit des Arrays
noch nicht erreicht?

```
int intStack::pop()
{
    if( top > 0 )
        top--;
    return stck[top];
}
```

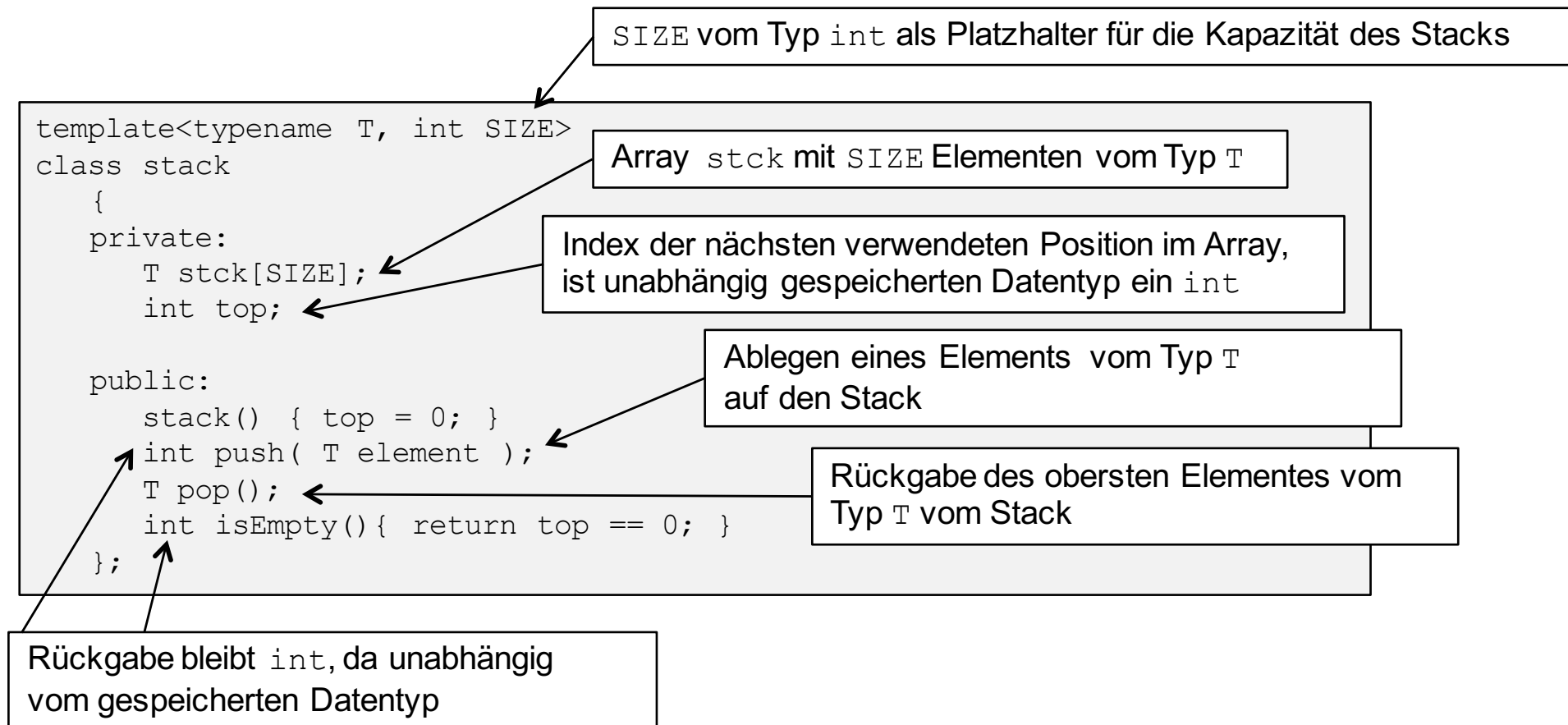

Verwendung des einfachen Stacks

Die Verwendung unseres typisierten Stacks erfolgt dann beispielsweise so:



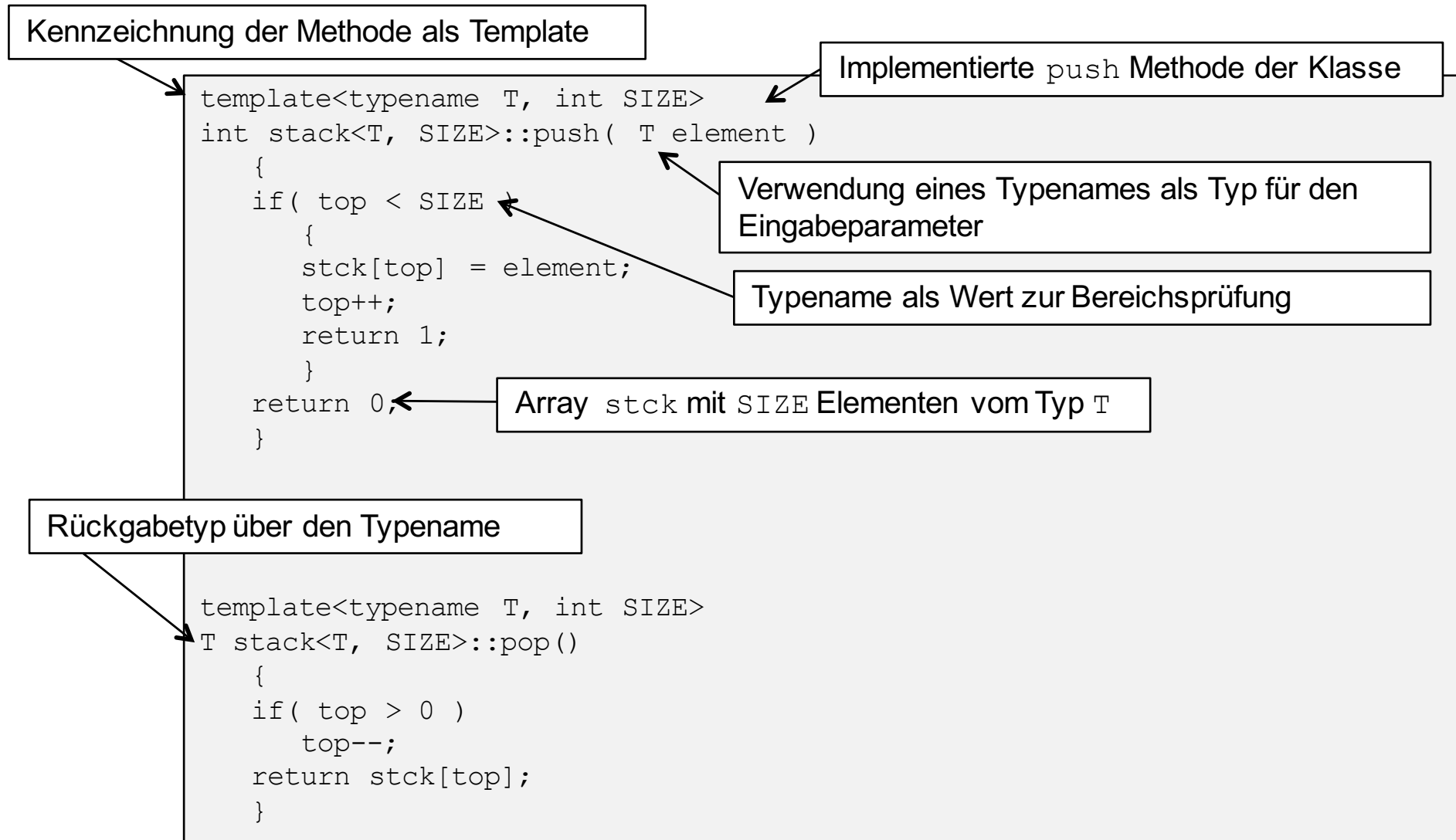
Stack-Klasse als Template

Auf Basis der Klasse `intstack` wollen wir nun eine generische Klasse erstellen, für die wir den Datentyp und die Anzahl der zu speichernden Elemente beliebig konfigurieren können. Dazu ersetzen wir den auf dem Stack zu speichernden Datentyp `int` an den entsprechenden Stellen durch den Platzhalter `T`. Das Template der Klassendeklaration sieht damit so aus:



Die Methoden der generalisierten Stack-Klasse

Die zur Klassendeklaration gehörigen Methoden haben damit folgende Implementierung:



Verwendung der generalisierten Stack-Klasse

Die generalisierte Stack-Klasse kann nun mit unterschiedlichen Datentypen und Stackgrößen verwendet werden.

```
int main()
{
    stack<int, 4 >istack;
    for( int i = 1; i <= 5; i++ )
    {
        istack.push( i );
        cout << "push: " << i << '\n';
    }

    while( !istack.isEmpty() )
    {
        int i = istack.pop();
        cout << "pop: " << i << '\n';
    }

    stack<datum, 3 >dstack;
    for( int i = 1; i <= 5; i++ )
    {
        datum dat( i, 6, 2015 );
        dstack.push( dat );
        cout << "push: " << dat << '\n';
    }

    while( !dstack.isEmpty() )
    {
        datum dat = dstack.pop();
        cout << "pop: " << dat << '\n';
    }
    return 0;
}
```

Instanziierung eines Stacks aus dem Template mit 4 Elementen für Integer-Werte

Instanziierung eines Stacks mit 3 Elementen für Instanzen des Typs datum

Jeweils 5 Elemente per push übertragen, aber nur SIZE Elemente entnehmbar

```
push: 1
push: 2
push: 3
push: 4
push: 5
pop: 4
pop: 3
pop: 2
pop: 1
push: 1.6.2015
push: 2.6.2015
push: 3.6.2015
push: 4.6.2015
push: 5.6.2015
pop: 3.6.2015
pop: 2.6.2015
pop: 1.6.2015
```

Hinweise zur Verwendung von Templates

Durch die entsprechende Verwendung der Platzhalter in einem Template werden implizit Annahmen über die Typen gemacht, mit denen das Template später verwendet werden kann.

Unsere Template-Klasse für einen einfachen Stack kann generell mit allen Datentypen aufgerufen werden, mit denen sie sich nach Ersetzung des Platzhalters auch übersetzen lässt. Die Methoden der Klasse setzen allerdings durch die Übergabe per Wert einen korrekt arbeitenden Copy-Konstruktor voraus.

Bei der Definition der Templates muss mit besonderer Sorgfalt vorgegangen werden. Zum Zeitpunkt der Erstellung ist ja noch unbekannt, für welche Klassen der Generierungsprozess zukünftig verwendet werden wird.

Insbesondere sollten Templates immer „geschlossen“ arbeiten und keine Seiteneffekte haben. Über solche Seiteneffekte können ansonsten nicht zusammenhängende Klassen ungewollt voneinander abhängig werden und interagieren. Verwendet ein Template beispielsweise globale Variablen, können sich verschiedene, aus dem gleichen Template generierte Klassen gegenseitig beeinflussen, ohne dass dieser Zusammenhang unmittelbar sichtbar ist.

Auch wenn diese Hinweise jetzt vielleicht abschreckend wirken, generell erhöhen Templates die Code-Qualität. Die Parameter sind zur Kompilierungszeit bekannt, bei der Übersetzung kann eine Typüberprüfung stattfinden und manuellen Kopien des Codes werden vermieden.

Da das Template zur Übersetzung für alle Code-Elemente zur Verfügung stehen muss, wird es typischerweise in einer Header-Datei abgelegt und zu den verwendenden Quellen inkludiert.

Templates der Standard-Template-Library (STL)

C++ bietet dem Benutzer in der Standard-Template-Library eine große Zahl hilfreicher Templates in Form sogenannter Container

Zu diesen Vorlagen und Containern gehören unter anderem:

- **vector**, dynamisches Array
- **list**: doppelt verkettete Liste
- **queue**: Warteschlange (FIFO)
- **stack**: Stapel (LIFO)
- **set**: Menge
- **string**: String

Diese Container sind bei der Erstellung eigener Programme sehr hilfreich. Die Verwendung der STL erleichtert nicht nur viele Aufgaben, sondern fast immer auch die Codequalität.

Fehler und Ausnahmebehandlung

Die Behandlung von Fehlern wurde bisher absichtlich nur am Rande behandelt. Dieses Vorgehen ist bewusst gewählt, um den Code übersichtlich zu halten und den jeweiligen Kern der Beispiele herauszuarbeiten. Die Erkennung und Behandlung aller Fehlersituationen macht den Code schnell „unlesbar“.

Das gilt bei realen Programmsystemen natürlich auch, nur kann die Fehlerbehandlung hier nicht einfach ignoriert werden.

Wir haben bisher Fehlersituationen über bestimmte Fehlercodes als Rückgabewerte der Funktionen mitgeteilt. Das gibt dem Aufrufer zum einen die Möglichkeit, Fehlersituationen komplett zu ignorieren. Zudem stößt dieses Verfahren schnell an seine Grenzen, wenn Fehlersituationen über mehrere Funktionen hinweg übermittelt werden müssen.

Innerhalb der Aufrufhierarchie wuchern dann die unterschiedlichen Fehlerbehandlungen schnell so um den Programmcode herum, dass der eigentliche Zweck des Codes kaum noch zu erkennen ist.

Um den Umgang mit Ausnahmesituationen zu erleichtern, ist es sinnvoll, die Ausnahmebehandlung in zwei Schritte aufzuteilen, nämlich die **Ausnahmeerkennung** und die nachfolgende **Ausnahmebehandlung**.

Ausnahmeerkennung

Eine Funktion kann eine Ausnahme- oder Fehlersituation meist leicht **erkennen**. Typische Situationen sind:

- Division durch 0
- Bereichsüberschreitung bei einem Zugriff auf ein Array
- Misserfolg einer Dateioperation
- Fehlschlag bei der Speicherallokation mit `malloc` oder `new`

Wenn eine Funktion eine Datei nicht einlesen kann, deren Name ihr übergeben wurde, ist dies leicht festzustellen. Dies gilt auch für die anderen genannten Fehlerfälle.

Wie der erkannte Fehler allerdings **behandelt** werden soll, ist in der Regel schwieriger zu entscheiden. Meist hat die Funktion dazu gar nicht genügend Informationen zum Kontext. Dies gilt insbesondere in Bibliotheksfunktionen, die am Ende einer Kette von Funktionsaufrufen liegen:

Soll das Einlesen noch einmal mit einer anderen Datei versucht werden, wird deren Name vom Benutzer erfragt? Muss das Programm sofort abgebrochen werden, oder kann das Problem vielleicht sogar ignoriert werden?

Die Frage der Ausnahmebehandlung muss also typischerweise an einer Stelle im Programm beantwortet werden, die den Kontext kennt und diese Frage entscheiden kann. Diese Stelle wird dabei oft deutlich weiter „oben“ in der Aufrufhierarchie liegen.

Ausnahmebehandlung

Die Anforderungen an die Behandlung einer Ausnahme sind für Hilfsprogramme anders als an ein Computerspiel, für eine Anwendungssoftware anders als für die Software eines Linienflugzeugs.

Auf die Frage, wie eine Ausnahme behandelt werden muss, wird es also keine allgemeine Lösung geben. Diese Frage muss in jedem Programm anhand der Anforderungen neu beantwortet werden.

Was vereinheitlicht werden kann, ist der Weg, auf dem die Information über das Auftreten eines Problems transportiert wird – von der Stelle der Erkennung zu der Stelle, an der das Problem behandelt werden kann.

Fehler und Ausnahmebehandlung

C++ bietet mit der **Ausnahmefallbehandlung**, dem sogenannten **Exception Handling** die Möglichkeit, Fehlersituationen über Aufrufhierarchien hinweg konsistent zu behandeln.

Eine Funktion kann eine Ausnahmesituation melden, dies wird auch als **Werfen** einer Ausnahme bezeichnet. Eine Funktion, die eine Ausnahme wirft, beendet damit ihre reguläre Ausführung. Weiterer Code, der auf das Werfen nachfolgt, wird nicht mehr ausgeführt.

Die geworfene Ausnahme wird durch die Aufrufhierarchie nach „oben“ weitergereicht, bis sie **gefangen** und behandelt wird. Der Fänger in der Funktion sein, die den ursprünglichen Aufruf gemacht hat, sie kann in der Hierarchie aber auch mehrere Funktionsaufrufe entfernt liegen.

Die geworfenen Ausnahmen

Es wurde bereits erwähnt, dass Ausnahmen geworfen werden können. Dabei ist noch nicht geklärt worden, welche Form eine Ausnahme eigentlich hat.

Ausnahmen werden typischerweise als Instanz speziell dafür deklarierter Klasse oder Klassenhierarchien geworfen. Eine der einfachsten Klassen die als Ausnahme geworfen werden kann, könnte so aussehen:

```
class ausnahme
{
    public:
        ausnahme() {}
};
```

Parameterloser Konstruktor

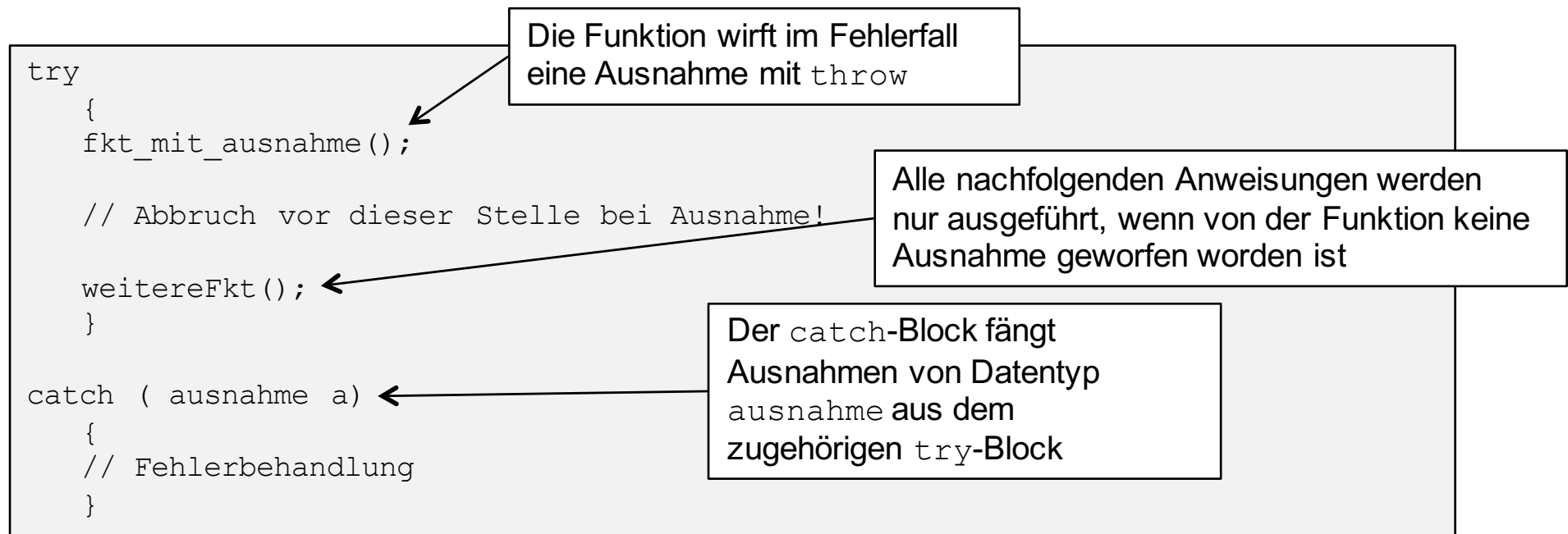


Von einer Ausnahme als Basisklasse, werden typischerweise passende Kindklassen abgeleitet, die weitere Informationen zum Fehlerfall enthalten. Ein Beispiel dafür werden wir später sehen, vorerst werden wir es aber bei dieser einfachen Variante belassen.

Das Exception Handling im Code

Das zentrale Element der Ausnahmefallbehandlung ist der `try-catch`-Block. Im `try`-Block ruft das Programm eine Funktion „versuchsweise“ auf. Wird aus der aufgerufenen Funktion heraus keine Ausnahme geworfen, wird der Code, der dem Aufruf im `try`-Block folgt, normal ausgeführt.

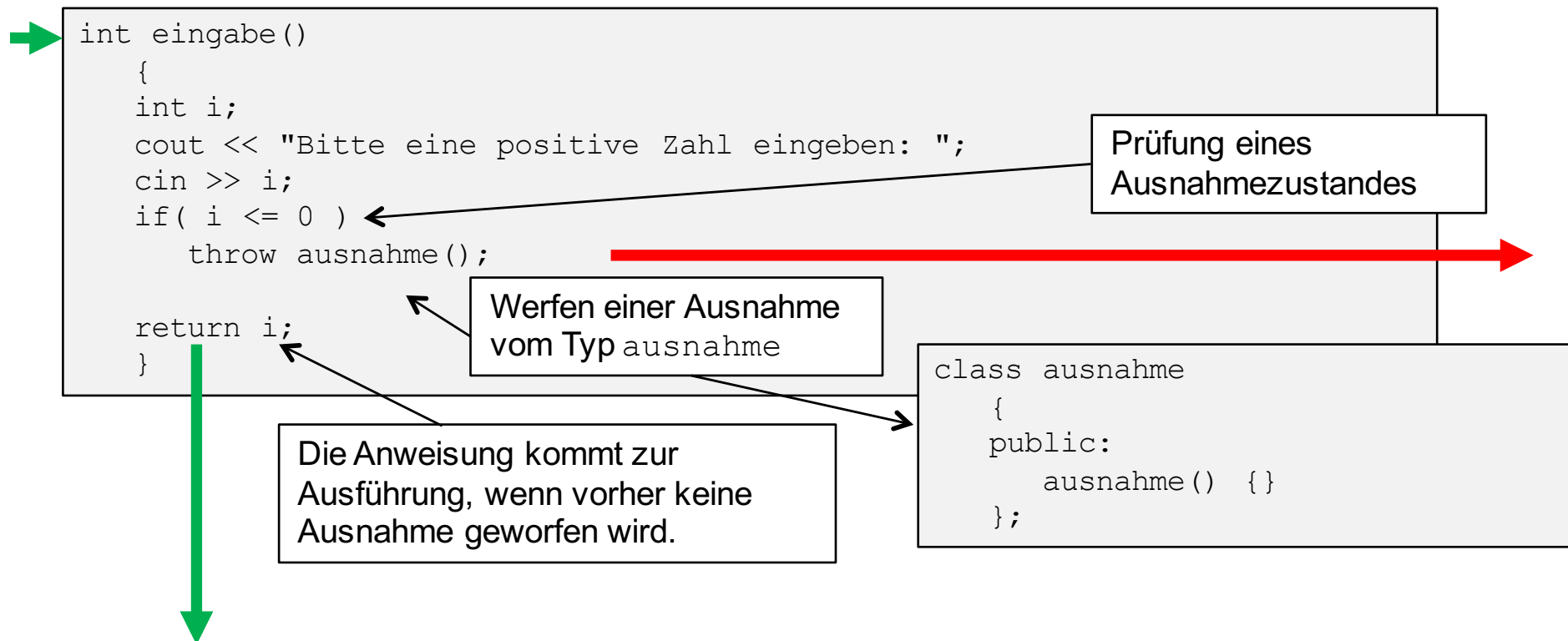
Eine eventuell geworfene Ausnahme wird vom `catch`-Block gefangen, der den eigentlichen Fänger für die Ausnahme bildet. Die Fehlerbearbeitung ist in diese `catch`-Blöcke ausgelagert. Wenn aus einer Funktion im `try`-Block heraus eine Ausnahme geworfen wird, wird der weitere Programmablauf der Funktion sofort abgebrochen. Der zugehörige `catch`-Block fängt die Ausnahmen des Datentyps für den er erstellt wurde und kann die Ausnahme nun behandeln.



Werfen einer Ausnahme

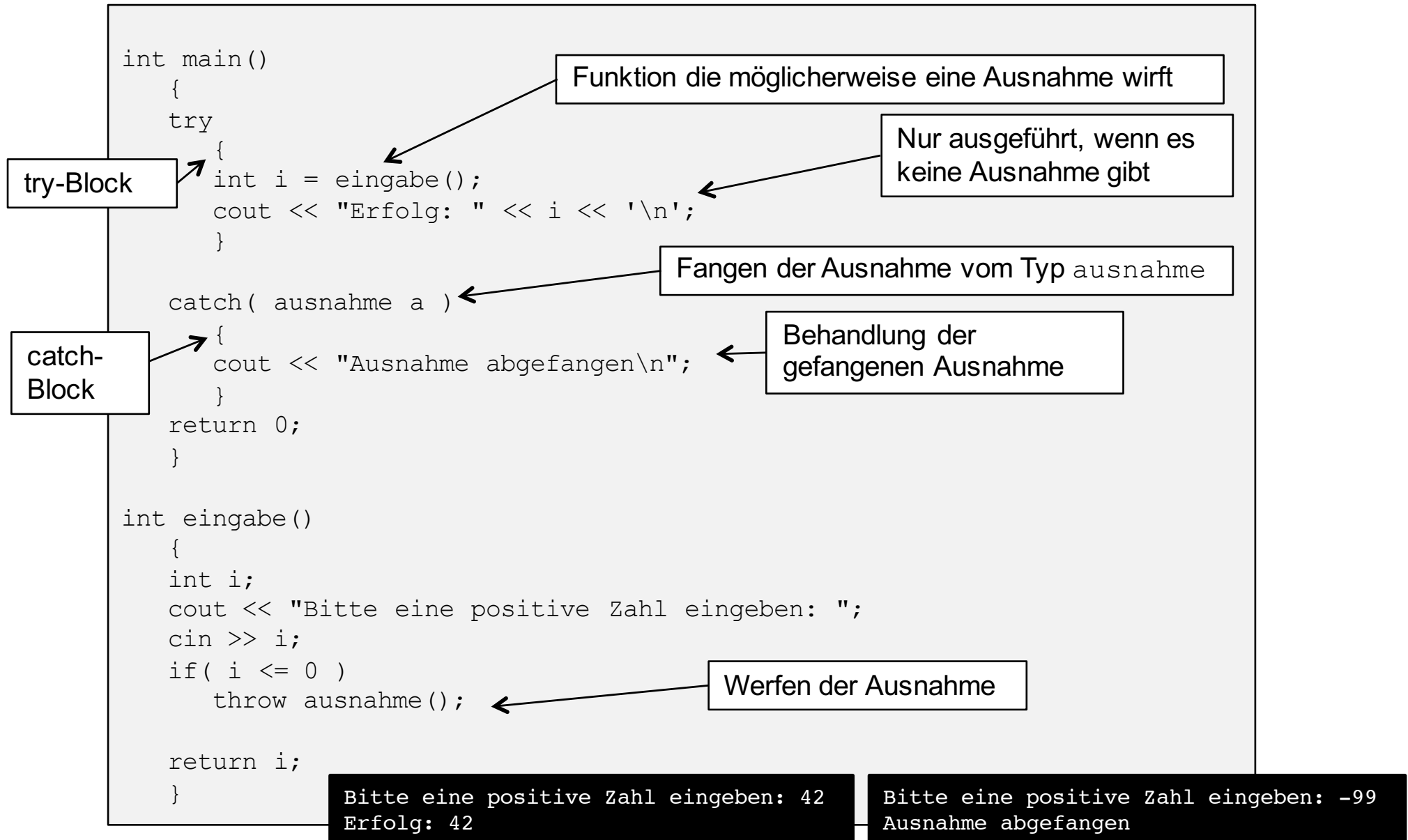
Eine Funktion kann jederzeit eine Ausnahme werfen. In diesem Fall wird die reguläre Ausführung sofort beendet. Die Ausnahme wird dann entlang der Aufrufhierarchie nach oben gegeben, bis sie in einem `catch`-Block gefangen wird.

Wir betrachten das Werfen einer Ausnahme am Beispiel. Die Funktion `eingabe` soll vom Benutzer eine positive Zahl ermitteln. Wird eine solche eingegeben, ist der Ablauf ungestört. Bei Eingabe einer negativen Zahl wirft die Funktion eine Ausnahme von Typ erstellten Klasse `ausnahme` und verlässt damit die Funktion vor dem eigentlichen Ende.



Fangen der geworfenen Ausnahme

Die Funktion wird mit einem entsprechenden `try-catch`-Block gerufen:



Programmablauf ohne Ausnahme

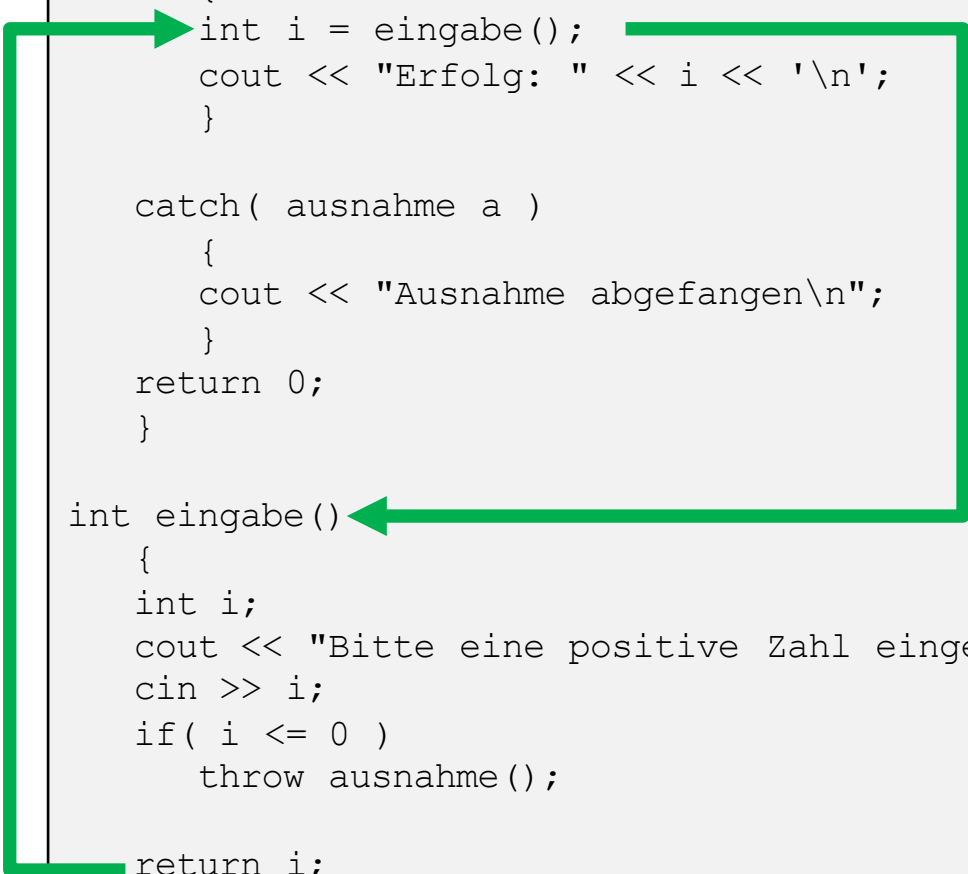
Die Funktion wird mit einem entsprechenden `try-catch`-Block gerufen:

```
int main()
{
    try
    {
        int i = eingabe();
        cout << "Erfolg: " << i << '\n';
    }

    catch( ausnahme a )
    {
        cout << "Ausnahme abgefangen\n";
    }
    return 0;
}

int eingabe()
{
    int i;
    cout << "Bitte eine positive Zahl eingeben: ";
    cin >> i;
    if( i <= 0 )
        throw ausnahme();

    return i;
}
```



```
Bitte eine positive Zahl eingeben: 42
Erfolg: 42
```

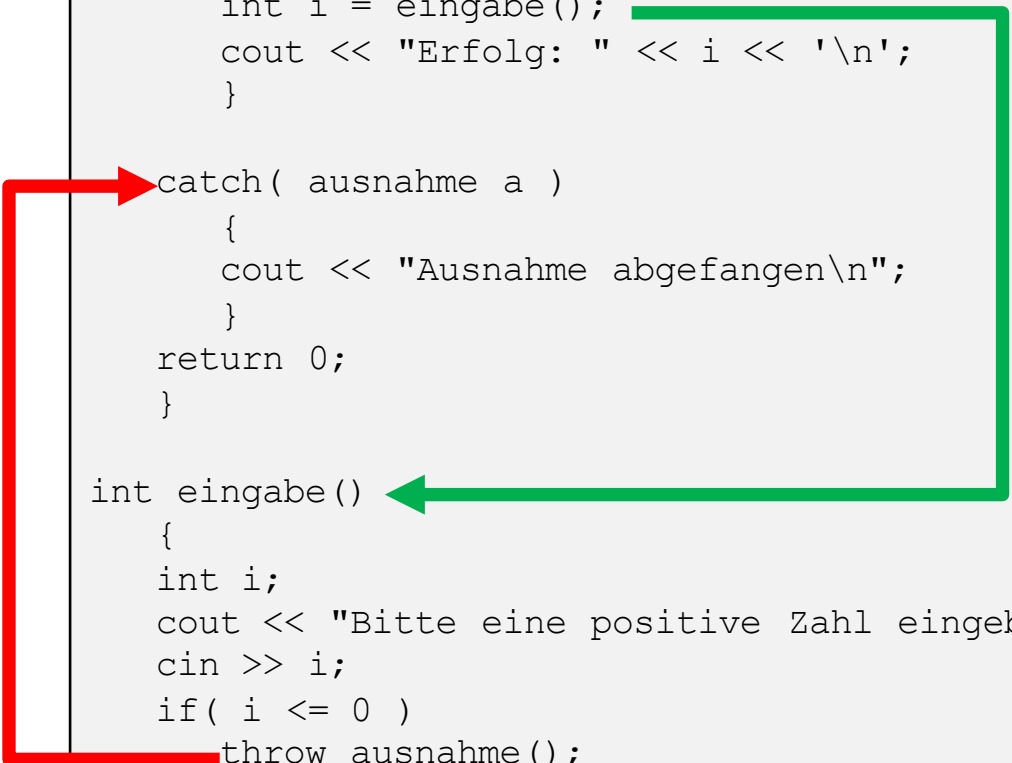
Programmablauf mit Ausnahme

Die Funktion wird mit einem entsprechenden try-catch-Block gerufen:

```
int main()
{
    try
    {
        int i = eingabe();
        cout << "Erfolg: " << i << '\n';
    }
    catch( ausnahme a )
    {
        cout << "Ausnahme abgefangen\n";
    }
    return 0;
}

int eingabe()
{
    int i;
    cout << "Bitte eine positive Zahl eingeben: ";
    cin >> i;
    if( i <= 0 )
        throw ausnahme();

    return i;
}
```



Bitte eine positive Zahl eingeben: -99
Ausnahme abgefangen

Spezialisierung der Klasse ausnahme

Bisher liefert die Klasse `ausnahme`, über ihre bloße Existenz hinaus, keine weitere Information. Wir werden von ihr eine spezialisiert Klasse ableiten, die Information liefert, die bei der Ausnahmebehandlung verwendet werden kann. Wir erstellen die Kindklasse und passen `eingabe` so an, das sie die Kindklasse mit zusätzlicher Information als Ausnahme wirft:

```
class ausnahme_bereich: public ausnahme
{
private:
    int wert;

public:
    ausnahme_bereich( int w ) { wert = w; }
    int getWert() { return wert; }
};
```

Die Klasse abgeleitet von der Basisklasse `ausnahme`

Zusätzliche Information zur Ausgabe für den Benutzer

```
int eingabe()
{
    int i;
    cout << "Bitte eine positive Zahl eingeben: ";
    cin >> i;
    if( i <= 0 )
        throw ausnahme_bereich( i );
    return i;
}
```

Werfen der erstellten Kindklasse `ausnahme_bereich` mit Zusatzinformation zum fehlerhaften Wert

Fangen der Klasse ausnahme und der Kindklassen

Wir passen unser Hauptprogramm ebenfalls an. Zur detaillierten Ausnahmebehandlung können auf einen `try`-Block auch mehrere `catch`-Blöcke folgen. Der erste `catch`-Block mit passendem Datentyp wird aufgerufen. Durch die Nutzung passender Vererbungshierarchien kann so eine gestaffelte Abfrage erfolgen und auch „unspezifischere“ Ausnahmen können bereits berücksichtigt werden, selbst wenn sie vielleicht erst in späteren Varianten einer Funktion geworfen werden.

```
int main() {  
    try  
    {  
        int i = eingabe();  
        cout << "Erfolg: " << i << '\n';  
    }  
  
    catch( ausnahme_bereich a )  
    {  
        cout << "Bereichsfehler mit Eingabe:" << a.getWert() << "\n";  
    }  
  
    catch( ausnahme a )  
    {  
        cout << "Ausnahme abgefangen\n";  
    }  
    return 0;  
}
```

Aufruf der Funktion die Ausnahmen vom Typ
der Klasse `ausnahme_bereich` wirft

Fangen der Ausnahmen `ausnahme_bereich` und
spezifische Behandlung

Fangen aller weiteren Ausnahmen vom Typ
`ausnahme`

Vollständiges Beispiel

Das komplette angepasste Beispiel sieht damit folgendermaßen aus:

```
int main() {
    try
    {
        int i = eingabe();
        cout << "Erfolg: " << i << '\n';
    }

    catch( ausnahme_bereich a )
    {
        cout << "Bereichsfehler mit Eingabe:" << a.getWert() << "\n";
    }

    catch( ausnahme a ) ←
    {
        cout << "Ausnahme abgefangen\n";
    }
    return 0;
}

int eingabe()
{
    int i;
    cout << "Bitte eine positive Zahl eingeben: ";
    cin >> i;
    if( i <= 0 )
        throw ausnahme_bereich( i );
    return i;
}
```

Fangen aller weiteren Ausnahmen vom Typ `ausnahme` ist möglich, die Ausnahmen würden ebenfalls gefangen und nicht zum Programmende führen

Ausnahmen über eine Aufrufhierarchie

Bisher haben wir Ausnahmen direkt in der aufrufenden Funktion gefangen. Um den Ablauf über die Aufrufhierarchie hinweg zu betrachten, erstellen wir drei nacheinander aufgerufene Funktionen, bei denen die dritte in einigen Fällen eine Ausnahme wirft:

```
void test1( int i )  
{  
    cout << "Aufruf von test2(" << i << ")\n";  
    test2( i );  
}
```

Aufruf der Funktion test2

```
void test2( int i )  
{  
    cout << "Aufruf von test3(" << i << ")\n";  
    test3( i );  
}
```

Aufruf der Funktion test3

```
void test3( int i )  
{  
    if( i % 3 == 0 )  
        return;  
    else if( i % 3 == 1 )  
    {  
        cout << "AUSNAHME1 wird geworfen\n";  
        throw ausnahme_bereich( 1 );  
    }  
    else if( i % 3 == 2 )  
    {  
        cout << "AUSNAHME2 wird geworfen\n";  
        throw ausnahme_bereich( 2 );  
    }  
}
```

Regulärer Rücksprung aus der Funktion

Ausnahme mit dem Parameter 1

Ausnahme mit dem Parameter 2

Aufruf der Funktionen

Der Aufruf der Funktionen erfolgt über `test1` aus dem Hauptprogramm

```
int main()
{
    for( int i = 0; i < 4; i++ )
    {
        try{
            cout << "Aufruf von test1(" << i << ")\n";
            test1( i );
            cout << "Kein Ausnahmefall aufgetreten\n";
        }

        catch( ausnahme_bereich a )
        {
            cout << "main faengt: " << a.getWert() << '\n';
        }
    }
    return 0;
}
```

Den Ablauf sehen wir in der folgenden Darstellung.

```
int main()
{
    for( int i = 0; i < 4; i++ )
    {
        try{
            cout << "Aufruf von test1(" << i << ")\n";

            test1( i );

            cout << "Kein Ausnahmefall aufgetreten\n";

        }
        catch(ausnahme_bereich a )
        {
            cout << "main faengt: " << a.getWert()<<'\n'
        }
    }
}
```

```
class ausnahme_bereich: public ausnahme
{
    // ...
};
```

```
void test1( int i )
{
    cout << "Aufruf von test2(" << i << ")\n";
    test2( i );
}
```

```
void test2( int i )
{
    cout << "Aufruf von test3(" << i << ")\n";
    test3( i );
}
```

```
void test3( int i )
{
    if( i % 3 == 0 )
        return;
    else if( i % 3 == 1 )
    {
        cout << "AUSNAHME1 wird geworfen\n";
        throw ausnahme_bereich( 1 );
    }
    else if( i % 3 == 2 )
    {
        cout << "AUSNAHME2 wird geworfen\n";
        throw ausnahme_bereich( 2 );
    }
}
```

```
Aufruf von test1(0)
Aufruf von test2(0)
Aufruf von test3(0)
Kein Ausnahmefall aufgetreten
Aufruf von test1(1)
Aufruf von test2(1)
Aufruf von test3(1)
AUSNAHME1 wird geworfen
main faengt: 1
Aufruf von test1(2)
Aufruf von test2(2)
Aufruf von test3(2)
AUSNAHME2 wird geworfen
main faengt: 2
Aufruf von test1(3)
Aufruf von test2(3)
Aufruf von test3(3)
Kein Ausnahmefall aufgetreten
```

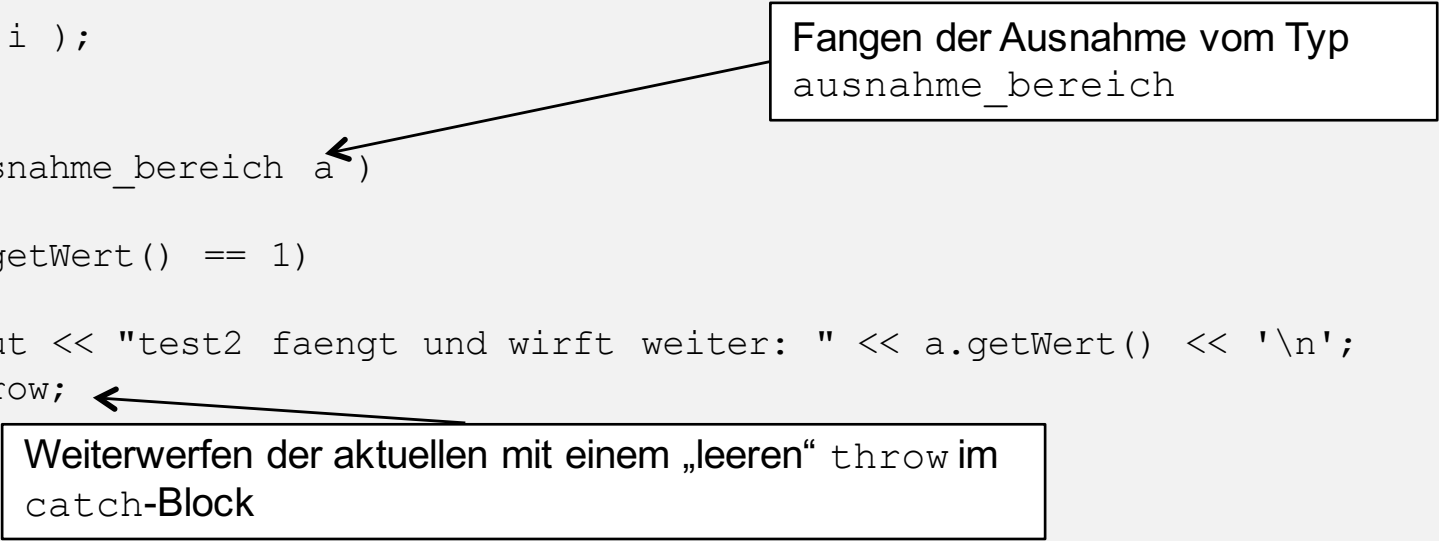
Fangen und weiterwerfen

In dem gerade gezeigten Beispiel wurden die Ausnahmen nur im Hauptprogramm gefangen. Generell hat aber jede Funktion in der Hierarchie die Möglichkeit, Ausnahmen zu fangen, auszuwerten und zu behandeln.

Als Beispiel haben wir hier eine Modifikation der Funktion `test2`, die bestimmte Ausnahmen fängt und bearbeitet. Die „Bearbeitung“ erfolgt hier nur in Form einer Ausgabe. Wenn die Bearbeitung nicht möglich oder abgeschlossen ist, kann die Funktion die Ausnahme wieder „weiterwerfen“.

```
void test2( int i ) // Modifiziert
{
    cout << "Aufruf von test3(" << i << ")\n";
    try{
        test3( i );
    }

    catch( ausnahme_bereich a )
    {
        if (a.getWert() == 1)
        {
            cout << "test2 faengt und wirft weiter: " << a.getWert() << '\n';
            throw;
        }
    }
}
```

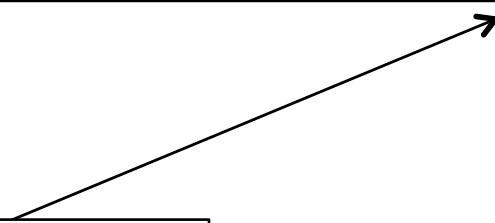


Ablauf mit modifizierter Funktion test2

Mit der modifizierten Funktion `test2` stellt sich der Ablauf des gesamten Programms dann so dar:

```
int main() {  
    for( int i = 0; i < 4; i++ )  
    {  
        try{  
            cout << "Aufruf von test1(" << i << ")\n";  
            test1( i );  
            cout << "Kein Ausnahmefall aufgetreten\n";  
        }  
        catch( ausnahme_bereich a )  
        {  
            cout << "main faengt: " << a.getWert() << '\n';  
        }  
    }  
    return 0;  
}
```

Behandlung der gefangenen
Ausnahme in `test2` und erneutes
Werfen der Ausnahme



```
Aufruf von test1(0)  
Aufruf von test2(0)  
Aufruf von test3(0)  
Kein Ausnahmefall aufgetreten  
Aufruf von test1(1)  
Aufruf von test2(1)  
Aufruf von test3(1)  
AUSNAHME1 wird geworfen  
test2 faengt und wirft weiter: 1  
main faengt: 1  
Aufruf von test1(2)  
Aufruf von test2(2)  
Aufruf von test3(2)  
AUSNAHME2 wird geworfen  
Kein Ausnahmefall aufgetreten  
Aufruf von test1(3)  
Aufruf von test2(3)  
Aufruf von test3(3)  
Kein Ausnahmefall aufgetreten
```


Bemerkungen zur Verwendung von Exceptions

Im Rahmen der Ausnahmebehandlung werden für Instanzen die als automatische Variablen in Funktionen angelegt wurden, (hier beispielsweise in `test1`, `test2` und `test3`) die entsprechenden Destruktoren aufgerufen, wenn die Funktion per `throw` verlassen wird. In den Funktion gegebenenfalls dynamisch angelegte Instanzen werden auf diesem Weg nicht automatisch beseitigt.

Wie auch in den Abbildungen leicht zu erkennen war, wird durch das Exception-Handling in dem Programm ein zweiter Kontrollfluss erzeugt. Die Wege, die der Kontrollfluss nimmt, sind nicht immer leicht nachzuvollziehen.

Sie sollten die Ausnahmebehandlung immer nur zur gezielten Behandlung von Ausnahmefällen verwenden und nicht versuchen, hier einen weiteren Programmablauf unterzubringen.

Die Klasse `exception` für Ausnahmen

In der C++ Standardbibliothek ist bereits eine Basisklasse `exception` enthalten. Diese Klasse wird von den bereits erwähnten Templates der STL genutzt. Alle Ausnahmen, die aus der STL-Bibliothek geworfen werden, sind vom Typ dieser Basisklasse oder einer ihrer Kindklassen. I

Die Klasse enthält geeignete Konstruktoren sowie Destruktor, Copy-Konstruktor und Zuweisungsoperator. Insbesondere können der Klasse im Konstruktor Zeichenketten als Informationen mitgegeben werden. Die Klasse kann auch als Basisklasse für eigenen Hierarchien verwendet werden.

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

```
class exception {
public:
    exception();
    exception( const exception& );
    exception(const char * const &);
    exception& operator= ( const exception& );
    virtual ~exception() throw( );
    virtual const char* what() const throw( );
}
```

Parameterloser Konstruktor

Konstruktor mit einer
Zeichenkette

Methode zur Rückgabe einer
im Konstruktor übergebenen
Zeichenkette