1.1. Exception Handling

Als Exception werden Ausnahmesituationen bezeichnet, die in Folge eines Fehlers den Normalablauf des Programms unterbrechen.

Mit einer Exception-Behandlung kann die Reaktion des Programms auf einen Fehler für einen gesamten Programmblock definiert werden.

Der prinzipielle Aufbau ist dabei:

```
try {
    Anweisung1;
    Anweisung2;
    Anweisung3;
}

catch (Typ1 argument1) {
    Behandlung1;
}
catch (Typ2 argument2) {
    Behandlung2;
}
catch (...) {
    Behandlung3;
}
```

Die einzelnen Anweisungen der zusammengesetzten Anweisung nach **try** werden abgearbeitet. Wenn kein Fehler auftritt, geht das Programm nach dem letzten **catch** normal weiter. Wenn in den Anweisungen 1 bis 3 ein Fehler auftritt, und dort eine Fehlerbehandlung durch **throw** ausgelöst wird, werden die einzelnen catch-Zweige nach einer passenden Fehlerbehandlungsroutine durchsucht. Dabei werden die Deklarationen nach dem Wort catch zur Auswahl verwendet: jener catch-Zweig wird verwendet bei dem die Typen übereinstimmen.

Wird keine passende Behandlungsroutine gefunden, wird die Funktion terminate() aufgerufen. Diese ruft standardmäßig abort() auf und führt damit zum Ende des Programms. Die Funktion terminate() ist nur gedacht für die Bearbeitung von Situationen, in denen die normale Behandlung von Ausnahmen nicht mehr möglich ist. Es ist nicht möglich, nach dem Aufruf von terminate() wieder zu der Stelle

zurückzukehren, von der aus der Aufruf erfolgt ist. Der Aufruf von terminate() muß zum Ende des Programms führen.

```
#include <iostream.h>
#include <math.h>

int main () {
    int a,b;
    cout << "Type a number: ";
    cin >> a;
    cout << endl;
    try {
        if (a > 100) throw 100;
        if (a < 10) throw 10;
        throw a/3;
    }
    catch (int result) {
        cout << "Result is: " << result << endl;
        b= result +1;
    }
    cout << "b contains: " << b << endl;

    // another example of exeption use
    char zero[] = "zero";
    char pair[] = "pair";
    char notprime[] = "not prime";
    char prime[] = "prime";
    try {
        if (a==0) throw zero;
        if ((a/2)*2 == a) throw pair;
        for (int i=3; i<= sqrt(a); i++) {
            if ((a/i)*i==a) throw notprime;
        }
        throw prime;
    }
    catch (char *conclusion) {
        cout << "The number you typed is " << conclusion
            << endl;
    }
}
```

1.2. Mehrstufige Ausnahmebehandlung

Die Auslösung von Ausnahmebedingungen kann über mehrere Ebenen hinweg erfolgen. So kann innerhalb eines try-Blocks wiederum ein try-Block definiert sein, für den es auch eigene exception handler gibt.

```
g() {
    int i;
    // ...
    throw i;
    // ...
    throw "Fehler in g";
};

int main(void) {
    try {
        // ...
        try {
            // ...
            g();
        };
        catch(int zahl) {
            // Version 1
            if (i>100) throw;
            if (i>1000) throw "Das ist zuviel";
        };
        // ...
        f1();
    }
}
catch(char* text) {...}
catch(int zahl) { // Version2 }
```

Wird in `g()` eine Ausnahme ausgelöst, wird zunächst der innerste Block durchsucht, um einen passenden `catch`-Block zu finden. Im ersten Fall wird daher die `Version1` des `exception handler` benutzt. Am Ende der Bearbeitung dieses Blocks wird das Ausnahmeobjekt, das mit `throw` erzeugt wurde, wieder gelöscht. Damit ist die Fehlerbehandlung beendet, es werden keine `exception handler` weiterer Ebenen ausgeführt.

Im zweiten Fall wird das Ausnahmeobjekt die Zeichenkette „Fehler in `g`“ erzeugt. Da es im innersten Block keinen `catch`-Block mit `char*` als Argument gibt, werden nun alle umgebenden Blöcke durchsucht, bis ein passender `catch`-Block gefunden wird.

Eine weitere Möglichkeit ist schließlich die ausdrückliche Weitergabe einer Ausnahme an den nächsten umgebenden Block. In dem `exception handler Version1` mit `int` als Objekttyp wird mit `throw` das Ausnahmeobjekt weitergegeben an die nächste Ebene. In diesem Fall wird bei Verlassen des `catch`-Blocks also das Ausnahmeobjekt nicht gelöscht. Es wird der `catch`-Block `Version2` ausgeführt, wie wenn die `int`-Ausnahme in der Funktion `f1()` ausgelöst worden wäre. Die Weitergabe an einen umgebenden Block ist allerdings nicht beschränkt auf die Ausnahme, die empfangen wurde, es ist auch möglich, innerhalb eines `catch`-Blocks beliebige Ausnahmen zu erzeugen. Als Beispiel wird im inneren `catch`-Block zur Behandlung von `int`-Ausnahmen auch eine Ausnahme mit dem Text „Das ist zuviel“ erzeugt, die im umgebenden Block bearbeitet werden muß.

Auf diese Weise kann die Behandlung von Ausnahmen über mehrere Ebenen hinweg erfolgen. Wird in einer Ebene kein passender `exception handler` gefunden, so wird auf der jeweils nächsten Ebene gesucht. Wird auf diese Weise auch auf der höchsten Ebene kein `exception handler` gefunden, so kommt es zu einem Laufzeitfehler.

Achtung: Exception Handling sollte immer mit Bedacht und nur an sinnvollen Stellen eingesetzt werden. Durch das `exception handling` soll es daher auf keinen Fall dazu kommen, daß quasi `goto`-Verzweigungen wieder in der Programmierung eingesetzt werden.

1.3. Namespaces

1.3.1. Übersicht

1. Ein Namespace ist ein Deklarationsbereich, der benannt werden kann.
2. Bezeichner müssen nur innerhalb eines Namespace eindeutig sein.
3. Ein Namespace kann über die Grenzen von Quelldateien hinweg in mehreren Teilen deklariert werden.
4. Namespaces können geschachtelt werden.

1.3.2. Deklaration

Beispiel:

```
namespace A {  
  
    int i;  
    void f();  
  
}
```

1.3.3. Zugriff

Beispiel:

```
A::i=5;  
A::f();
```

1.3.4. Module mit Namespace

1. Jedes Modul innerhalb eines Namespace implementieren
2. Namen im Modul entsprechend wählen,
z.B.: Modul Queue -> Funktion Init und nicht QueueInit nennen
3. Den Namespace eines Moduls nicht außerhalb des Moduls erweitern.
4. Den Namespace eines moduls nicht mit `using` importieren.
Der Import von Modulen erfolgt über `#include`
5. Aliases für namespaces vermeiden

1.3.5. Beispiel - Headerdatei

```
#ifndef _PIXEL
#define _PIXEL

/* Module Description :
   This module contains the declaration of two
   classes:
   class Location describes screen locations in X
   and Y coordinates
   class Action describes whether a point is
   hidden or visible
*/
/*-----*/

namespace Pixel {

// Exported declarations

typedef int XP;
typedef int YP;

class Location {
    // Attributes
protected:
    XP xpXPos;
    YP ypYPos;

    // Attribute access methods
public:
    XP xpGetXPos();
    YP ypGetYPos();

    // Public methods
public:
    Location (XP xpInitialXPos, YP ypInitYPos);

};    /* end of class Location */
```

```
class Action : public Location {
    // Attributes
protected:
    bool bVisible;

    // Constructor
public:
    Action (XP xpInitialXPos, YP ypInitialYPos);

    // Public methods
public:
    void Show();
    void Hide();
    Boolean bIsVisible();
    void MoveToPosition (XP xpNewXPosition,
                        YP ypNewYPosition);

}; /* end of class Action */

} // end of namespace
#endif // _PIXEL
```

1.3.6. Beispiel – Modul-Implementierung

```
/* Module Description :

The module Pixel contains the implementation of
two classes:
class Location describes screen locations in X
and Y coordinates
class Action describes whether a point is
hidden or visible
*/
/*-----*/

// Includes
#include "GRAPHICS.H" // Borland Graphics Lib.
#include "PIXEL.H"    // Module Interface Def.

namespace Pixel {

/*-----*/
/* Member functions for Pixel::Location      */
/*-----*/

Location::Location (XP xpInitialXPos,
                    YP ypInitialYPos)
/*
 * Purpose : Constructor
 * Precon  : None
 * Actions : Initialises the Pixel X and Y
              co-ordinate positions
 * Postcon : Object co-ordinates set to known
              initial values
 */
{
    xpXPos = xpInitialXPos;
    ypYPos = ypInitialYPos;
} // Location::Location
```

```

XP Location::xpGetXPos()
{
    return (xpXPos);
} // Location::xpGetXPos

```

```

YP Location::ypGetYPos()
{
    return (ypYPos);
} // Location::ypGetYPos

```

```

/*-----*/
/* Member functions for Pixel::Action          */
/*-----*/

```

```

Action::Action (XP xpInitialXPos,
                YP ypInitialYPos)
                : Location (xpInitialXPos,
                            ypInitialYPos)

```

```

/*
* Purpose : Constructor
* Precon  : None
* Actions : Initialises the Pixel X and Y co-
              ordinate positions via
*           its Base class and make pixel
              invisible by default
* Postcon : Object set to known initial values
*/

```

```

{
    bVisible = false;
} // Action::Action

```

```

void Action::Show ()
/*
 * Purpose : Make a pixel visible
 * Precon  : Graphics system must have been
             initialised
 * Actions : Turns on the pixel it the default
             colour.
 * Postcon : Pixel visible
 */
{
    bVisible = true;
    putpixel (xpXPos, ypYPos, getcolor());
} // Action::Show

```

```

void Action::Hide ()
/*
 * Purpose : Make a pixel hidden
 * Precon  : Graphics system must have been
             initialised
 * Actions : Turns off the pixel by making it
             the background colour.
 * Postcon : Pixel hidden
 */
{
    bVisible = false;
    putpixel (xpXPos, ypYPos, getbkcolor());
} // Action::Hide

```

```

Boolean Action::bIsVisible ()
/*
 * Precon  : None
 * Actions : Returns true if Pixel is visible.
 * Postcon : None
 */
{
    return bVisible;
} // Action::bIsVisible

```

```

void Action::MoveToPosition (XP xpNewXPosition,
                             YP ypNewYPosition)
/*
 * Purpose : To show a pixel in a new position
 * Precon  : Graphics system must have been
             initialised
 * Actions : Turn off the old pixel
 *           Save new positions
 *           Turn on the pixel at the new
             position.
 * Postcon : Object's screen co-ordinates have
             been updated.
 */
{
    Hide();
    xpXPos = xpNewXPosition;
    ypYPos = ypNewYPosition;
    Show();
} // Action::MoveToPosition

}

```

1.3.7. Beispiel – Modul-Import

```
/*  Module Description :

    Main application to demonstrate the
    PixelLocation and PixelAction classes.
    Compile and link using Borland Project file
    TESTPIXEL.PRJ.
*/
/*-----*/

// Includes
#include "PIXEL.H"
#include "GRAPHICS.H" // Borland Graphics Libray
#include "CONIO.H"

/*
 * The program turns on a pixel and waits for a
 * key. When keypressed, the pixel turned on is
 * moved to a new position. On the next keypress
 * the program terminates.
*/

// Local functions
int main()
{
    // Declarations
    const Pixel::XP xpInitialX = 100;
    const Pixel::YP ypInitialY = 50;

    const Pixel::XP xpNextX = 300;
    const Pixel::YP ypNextY = 150;

    int graphmode;
```

```

// declare an object APoint of class
// Pixel::Action and initialise its location
Pixel::Action APoint (xpInitialX,
                      ypInitialY);

// initialise the graphics system
// See Borland C++ Programmer's Guide page 236
int graphdriver = DETECT;

initgraph (&graphdriver, &graphmode,
          "c:..\\bgi");

// move a point across the screen
APoint.Show();
getch();
APoint.MoveToPosition (xpNextX, ypNextY);
getch();

APoint.Hide();
getch();

closegraph();
return 0;
} // main

```