Institute of Automation

C++ Basics and Applications in technical Systems

Lecture 8 - Polymorphism, template classes and exeptions

Institute of Automation
University of Bremen

14th December 2012 / Bremen

WiSe 2012/2013

VAK 01-036

Universität Bremen

Overview

Institute of Automation

Universität Bremen

Lecture schedule

Institute of Automation

Time schedule

- HK **26. Oct.** - Introduction / Simple Program / Datatypes ...
- HK **02. Nov.** - Flow control / User-Defined Data types ...
- CF **09. Nov.** - Simple IO / Functions/ Modular Design ...
- CF **16. Nov.** - C++ Pointer
- CF **23. Nov.** - Object oriented Programming / Constructors
- AL **30. Nov.** - UML / Inheritance / Design principles
- AL **07. Dec.** - Namespace / Operators
- AL **14. Dec.** - Polymorphism / Template Classes / Exceptions
- HK **11. Jan.** - Design pattern examples

Universität Bremen

Important dates

Institute of Automation

Submission of exercises

1-3 **16. Nov.** - Deadline for submission of Exercise I, 13:00

4-6 **07. Dec.** - Deadline for submission of Exercise II, 13:00

For admission to final exam you need at least 50% of every exercise sheet.

Final project

1-9 **15. Feb.** - Deadline for submission of final project, 13:00

1-9 **21. Feb.** - Student presentations of their final projects, 10:00 - 12:00, 14:00 - 17:00

Final exam

1-9 **06. Feb.** - Final exam, 10:00-12:00, H3

Bremen

Example usage of namespaces

Institute of Automation

Example

```
namespace DevLayer
{
  static int GetDevID(void);
}
```

usage version #1

```
int main()
{
  using namespace DevLayer;
  devID = GetDevID();
}
```

usage version #2

```
int main()
{
  using DevLayer::GetDevID;
  devID = GetDevID();
}
```

usage version #3

```
int main()
{
  devID = DevLayer::GetDevID();
}
```

Universität Bremen

Misc on namespaces

Institute of Automation

Attention

The usage of `using namespace ...` at global scope destroys the concept of namespaces!

Classes as namespace for static members

For static members of classes (they are not bound to a specific instance of that class) the class name builds a namespace:

```
class Device
{
public:
  static void Shutdown(void);
}
```

```
int main()
{
  Device::Shutdown();
}
```

Universität Bremen

Introduction

Institute of Automation

Definiton

Within a method `this` is a pointer to the current object and `*this` is the object itself.

Example

```
class MyClass
{
  MyClass()
  {
    std::cout << "Address of this instance: 0x"
              << static_cast<long>(this)
              << std::endl;
  }
};
```

Universität Bremen

Operators

handling of 'operator-like' methods

Example

```
Matrix a(3,3);
...                   // Set elements of a
Matrix b(3,3);
...                   // Set elements of b
a.Assign(b);          // 'Operator-like' method
```

more intuitive: `a = b;` or `Matrix a = b + c/2;`

Global vs. Element-function

By means of the output operator `<<` and the class `MyClass`

- Objective: `cout << MyClassObject;`
- Syntax:
  - 1st possibility:

    ```
    cout.operator<< (MyClassObject);
    ```

  - 2nd possibility:

    ```
    operator<< (cout, MyClassObject);
    ```

Conclusion

Only the 2nd version is really possible, i.e., the operator has to be declared as a global function.

Polymorphism

Institute of Automation

Polymorphism is:

Operations with identical names, belonging to different objects, may behave differently. Objective $\rightarrow$ Treat an object of a derived class as an object of the base class within a certain context.

You already know about one kind of polymorphism:

Static polymorphism

Overloading of functions, methods and operators.

Universität Bremen

Dynamic polymorphism

Institute of Automation

Another kind of polymorphism is:

Dynamic polymorphism

Delayed linkage
E.g. the decision which function/method call is associated with the operation will be known during runtime only. This is called runtime or dynamic polymorphism.

- In C++ polymorph methods can be used only in **inherited class hierarchies.**
- All polymorph methods must have **identical signatures.**
- A method that will be selected during runtime is called **virtual method.**

Universität Bremen

Virtual methods

Institute of Automation

The keyword `virtual`

Virtual methods of a (base) class, can be reimplemented / redefined in each derived class.

- usage of virtual methods is called run-time polymorphism
- statement: `virtual <method declaration>;`

Example

```
class Point
{
public:
  virtual void Move(int iX,
                    int iY);
};
```

Example

```
class Rectangle
  : public Point
{
public:
  void Move(int iX, int iY);
};
```

Bremen

Usage example

Institute of Automation

Example

```
class Point
{
public:
  virtual void Move(int iX,
                    int iY);
};

class Rectangle
  : public Point
{
public:
  void Move(int iX, int iY);
};
```

During runtime the correct method is called automatically!

Example

```
Point * pElement;
Rectangle newRectangle(...);
Point newPoint(...);

pElement = &newPoint;
pElement->Move(...);

pElement = &newRectangle;
pElement->Move(...);
```

Universität Bremen

Virtual method background

- Type-informations are stored for each object internally.
- Difference between „normal“ and „virtual“ method is only observable if pointer or reference to base class is used for access.
- Call to non-virtual methods depends on the type of the pointer or reference, virtual method calls depend on the object type.
- A method declared as virtual is used as **interface definition** for all derived classes.

Rule-to-Remember

Non-virtual member functions should not be overloaded within derived classes.

Example **without** `virtual` method

Institute of Automation

Example

```
class Vehicle {
public:
  void Travel() {
    std::cout << "no vehicle"
    " specified" << std::endl;
  }
};
```

Example

```
class Car : public Vehicle {
public:
  void Travel() {
    std::cout << "we travel"
    " by car" << std::endl;
  }
};
```

Example

```
Car newCar;
Vehicle *pVehicle(&newCar);
pVehicle->Travel(); // output: no vehicle specified
```

Universität Bremen

Example with `virtual` method

Institute of Automation

Example

```
class Vehicle {
public:
  virtual void Travel() {
    std::cout << "no vehicle"
    " specified" << std::endl;
  }
};
```

Example

```
class Car : public Vehicle {
public:
  void Travel() {
    std::cout << "we travel"
    " by car" << std::endl;
  }
};
```

Example

```
Car newCar;
Vehicle *pVehicle(&newCar);
pVehicle->Travel(); // output: we travel by car
```

Universität Bremen

Usage of polymorphism for containers

```
// vehicle park
class Aircraft : public Vehicle { ... };
class Ship : public Vehicle { ... };
class Balloon : public Vehicle { ... };
```

```
// storing all vehicles in a std::vector with pointer to base class
std::vector<Vehicle*> vehiclePark;
```

```
// fill vector
vehiclePark.push_back(new Car);
vehiclePark.push_back(new Aircraft);
vehiclePark.push_back(new Ship);
vehiclePark.push_back(new Balloon);
```

```
// access elements of vector
vehiclePark[0]->Travel(); // output: we travel by car
vehiclePark[1]->Travel(); // output: we travel by aircraft
vehiclePark[2]->Travel(); // output: we travel by ship
vehiclePark[3]->Travel(); // output: we travel by balloon
// do not forget to delete all elements of vehiclePark ...
```

Institute of Automation

Use a `virtual` destructor

Example

```
class Vehicle {
public:
  virtual ~Vehicle() {
    std::cout << "vehicle"
    " destroyed" << std::endl;
  }
};
```

Example

```
class Car : public Vehicle {
public:
  ~Car() {
    std::cout << "car"
    " destroyed" << std::endl;
  }
};
```

Example

```
Car * pCar(new Car);
Vehicle * pVehicle(pCar);
delete pVehicle; // output: car destroyed
                 //         vehicle destroyed
```

Universität Bremen

Institute of Automation

Dynamic Cast

Sometimes it is necessary to access the derrived class from an interface. This is possible by dynamic casting the pointer to the derrived class.

Usage

```
DerrivedClass* pDerrivedClass =
    dynamic_cast< DerrivedClass* >( pBaseClass );
```

If pBaseClass is not a pointer to a DerrivedClass object, the dynamic_cast returns a NULL pointer.
Warning: static_cast does not type check the cast. Returned pointer may be invalid and can crash the program on using!

Example

```
Car* pCar = dynamic_cast< Car* >( pVehicle ) ;
```

Bremen

Summarization

Remember

- Every class, from which should be inherited, should have a virtual destructor.
- Every class, which has at least one virtual method (incl. the destructor) is called polymorphic type.
- A virtual and overridden method is only accessible from its derivatives. E.g.:

```
void Aircraft::Travel() { Vehicle::Travel(); }
```

- The `virtual` statement is only used in the declaration / header file.
- A method that was declared `virtual` should also be declared `virtual` in all derived children to improve readability.

Abstract classes

Institute of Automation

Sometimes it is necessary to declare a method in a base class, which must be implemented in each derived class, but not in the base class itself.

Pure virtual method declaration syntax:

```
virtual <method declaration> = 0;
```

Example

```
class Plugin {
public:
  virtual Icon * GetIcon(void) = 0;
};
```

Classes with at least one pure virtual method are called abstract classes.

Universität Bremen

Example

Institute of Automation

Example

```
class Vehicle {
public:
   virtual void Travel() = 0;
   // Method is pure virtual,
   //  no implementation
   //  definition here!
};
```

Example

```
class Car : public Vehicle {
public:
   void Travel()
   // This method must be
   // implemented here!
   {
      std::cout << "we travel"
      " by car" << std::endl;
   }
};
```

Example

```
Car newCar; // OK, does compile
Vehicle newVehicle; // NOT OK, error from compiler!
```

Remember...

- A class with at least one pure virtual method cannot be instantiated directly, only their derivatives.
- An abstract class with only pure virtual methods is normally used for **interfaces**.

Usage...

Abstract classes are often used as common interfaces in class hierarchies.

Small exercise

Create a class hierarchy with a base class for 2-dimensional shapes: rectangle, triangle and circle:

- The base class should be an interface (abstract class).
- Create a class for each shape.
- Each class for a shape should have a `Draw()` method, which outputs „Drawing <shape type>“.
- Store different objects of shapes in one `std::vector` and test the `Draw()` method within a `main()` function.

General idea of templates

A template:

- enables to construct methods without defining parameter types.
- is easier to write.
- is easier to understand.
- is type-safe.

Your already know about function templates (lecture 3):

```
template <class tType> void Swap(tType &A, tType &B);
```

Repetition: function-templates

- Function templates for unspecified data types
- Generic usable algorithms can be made available
- **Declaration and definition must be in the header-file!**

Definition

```
template <class tType>
void Swap(tType &A, tType &B)
{
  tType Temp = A;
  A = B;
  B = Temp;
}
```

Example

```
int iNumber1;
int iNumber2;
iNumber1 = 1;
iNumber2 = 2;

Swap(iNumber1, iNumber2);
```

Class templates

Institute of Automation

Beside function templates it is also possible to create class templates and method templates.

Usage for abstract data-types like:

- graphs
- sets
- queues
- lists
- trees
- matrices

You already know two template classes: `std::vector` and `std::complex`. The first with one and the second with two template arguments.

Universität Bremen

Institute of Automation

Example template class

```
#ifndef SIMPLE_STACK_H
#define SIMPLE_STACK_H
#include <vector>                  // std::vector to store elements
```

```
template <class T>
class SimpleStack
{
```

```
public:
        SimpleStack();                  // Constructor
        const T& Top(void) const;       // Method to get top element
        void Pop(void);                 // Method to remove top element
        void Push(const T & x);         // Method to add element
```

```
private:
        std::vector<T> m_Array;         // std::vector for stack
```

```
};
```

```
#endif
```

Universität Bremen

Usage of template class

Institute of Automation

Example

```
SimpleStack< unsigned int > SStack ; // Constructs SimpleStack
                                     //  with type unsigned int

SStack.Push( 1 ) ;                   // Fills stack
SStack.Push( 2 ) ;
SStack.Push( 3 ) ;

cout << SStack.Top() << endl ;      // Print out top element
SStack.Pop() ;                      // Removes from stack
```

Universität Bremen

Institute of Automation

Code organization with templates I

Problem:

Datatype in template only specified during instantiation.
$\rightarrow$ Declaration + definition have to be included in compilation unit (impl/cpp-file) of template-instantiation.

Solution 1:

Declaration + definition in header-file:

- Normal approach
- Attention: code bloat in large projects!!

Universität Bremen

Code organization with templates II

Institute of Automation

Solution 2:

Separate declaration „Stack.h“ and definition „Stack.cpp“ and inline implementation in separate file e.g. „Stack.inl“.

Example

```
// Stack.h
template<class T>
class Stack
{
  void Push(T & value);
};

#include "Stack.inl"
```

Example

```
// Stack.inl
template<class T>
void Stack<T>::Push(T & value)
{
  // ...
}
```

Universität Bremen

Templates and polymorphism

You already know:

Run-time (dynamic) polymorphism

Polymorphism with virtual methods is called **run-time polymorphism.**

With respect to templates a new type of polymorphism is:

Compile-time polymorphism

Polymorphism with templates is called parametric or **compile-time polymorphism.**

Small exercise

Institute of Automation

- Create a simple stack template class with methods to add, remove and get elements from the top of the stack.
- Create a test application to test your stack template class.

Example

```
template <class T>
class SimpleStack
{
public:
  const T& Top(void) const;      // Method to get top element
  void Pop(void);              // Method to remove top element
  void Push(const T & x);           // Method to add element
private:
  std::vector<T> m_Array;           // std::vector for stack
};
```

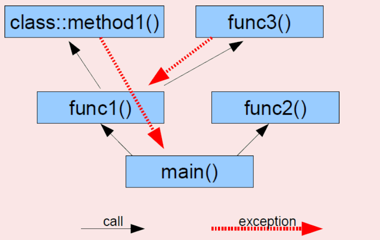
Universität Bremen

Exception handling

Institute of Automation

- Exception can / shall not be handled within the scope of its occurrence.
- Program abort in case of exception is often unnecessary and not user-friendly.

Call graph



Concept: Parent scope is informed about error:

- and does exception handling.
- forwards exception notification to own parent scope.

Universität Bremen

Usage of exceptions for error handling I

Institute of Automation

Exception handling is mainly used for error handling.

- Code for error handling can be separated from „normal“ code.
- Complexity in code is reduced.

statement for callee

```
throw (<exception object>);
throw (); // no exceptions
```

Without the throw statement any class can be thrown ⇒ unexpected for caller.

statement for caller

```
try {
  // error prone code
}
catch (<exception type>
        [object name])
{
  // error handling
}
```

Universität Bremen

Usage of exceptions for error handling II

Institute of Automation

Declaration

```
class MyClass {
public:
  void Method(void)
    throw (ExceptionType1,
           ExceptionType2);
};
```

Hint: Don't forget the exception specification in your declaration and definition!

Definition

```
void MyClass::Method(void)
  throw (ExceptionType1,
         ExceptionType2)
{
  // ...
  if (bVeryBad)
    throw ExceptionType1;
  if (bVeryVeryBad)
    throw ExceptionType2;
  // ...
  return;
}
```

Universität Bremen

Institute of Automation

Example

Example

```
double Div(double dDividend,
           double dDivisor)
{
  std::string Exception(
    "division by 0");

  if (dDivisor == 0.0)
  {
    // throw exception
    throw (Exception);
  }

  // return result
  return dDividend
         / dDivisor;
}
```

Example

```
double dNumberA;
double dNumberB;
std::cin >> dNumberA;
std::cin >> dNumberB;

try {
  std::cout << Div(dNumberA,
                   dNumberB)
            << std::endl;
}
catch (std::string error) {
  std::cout << error
            << std::endl;
}
```

Universität Bremen

Summarization

Rules to remember

- Enclose each function / method that can throw an exception in a `try`-block.
- Catch each kind of exception and handle it or forward it using throw again.
- Use `catch(...){}` to catch all exceptions anonymously.
- If there is no object type given for `throw`, no exception can be thrown.
- Without the throw statement any class can be thrown $\Rightarrow$ unexpected for caller.

Uncaught exceptions

Institute of Automation

- If an exception is thrown, but not caught, the application will be terminated!
- Take care for standard exceptions thrown by the `new`-operator: `std::bad_alloc`.

Example

```
double * pValue(NULL);
try {
  pValue = new double(0.0);
}
catch (std::bad_alloc) {
  std::cout << "not enough memory available"
            << std::endl;
}
```

Universität Bremen

Institute of Automation

Grouping exceptions

Often it is useful to group exceptions depending on the occurred error. A possible and good approach is to use inheritance for own exception types.

Example

```
// General error/exception type
class MathErr { };

// Exception type for division by zero
class DivisionZeroErr : public MathErr { };
// Exception type for overflow
class OverflowErr : public MathErr { };
// Exception type for underflow
class UnderflowErr : public MathErr { };
```

Universität Bremen

Catch different exceptions

Institute of Automation

Be careful: The order of `catch()`-statements is important when using exceptions based on an inheritance hierarchy.

Example

```
try {
  // execute Div() with arbitrary values
  dResult = Div(dNumberA, dNumberB)
}
catch (DivisionZeroErr) {
  // handle division by zero errors
}
catch (OverflowErr) {
  // handle overflow errors
}
catch (MathErr) {
  // handle all math errors
}
```

Institute of Automation

Transporting data with exceptions

Using classes enables to add further functionality to exception types.

Example

```
class Error_T {
  int m_iErrNum;
  std::string m_sError;
public:
  Error_T(int iErrNum,
          std::string sError)
    : m_iErrNum(iErrNum),
      m_sError(sError) {}
  void PrintMessage(void) {}
};
```

Example

```
if ( ... very bad things ...)
{
  throw Error_T(13, "Friday");
}
```

Example

```
try
{
  // ...
}
catch (const Error_T& error)
{
  error.PrintMessage();
}
```

Bremen

Small exercise

Institute of Automation

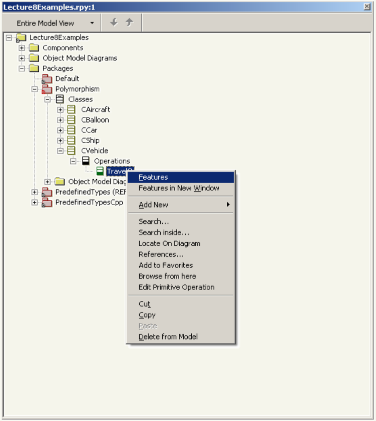
Extend the class `Rational` from lecture #7 about exceptions for error cases (e.g. division by zero):

- create own exception types
- use exception specifications
- test for error cases the behavior with and without catching the exceptions

Universität Bremen

Institute of Automation

Polymorphism (XI) – Rhapsody

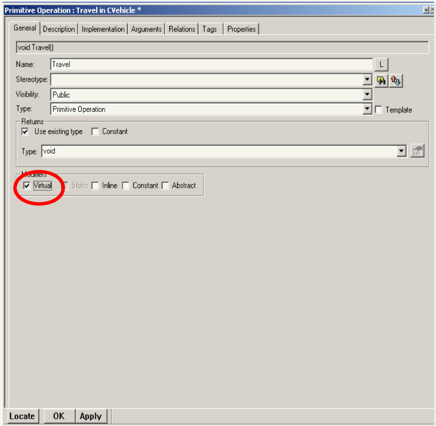


- Creating virtual methods:
  - Select a method (in Rhapsody: operation) with right click
  - Choose “**Features**”

Universität Bremen

Institute of Automation

Polymorphism (XII) – Rhapsody

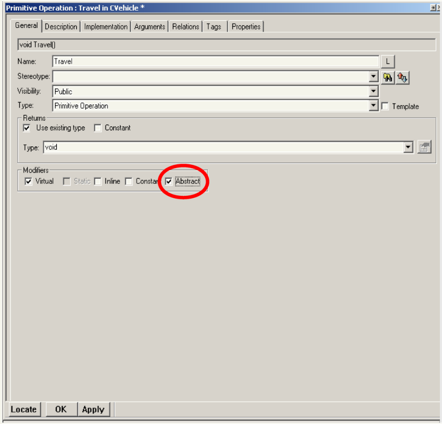


– Select the “**virtual**” modifier under “**General**”

Universität Bremen

Institute of Automation

Polymorphism (XII) – Rhapsody

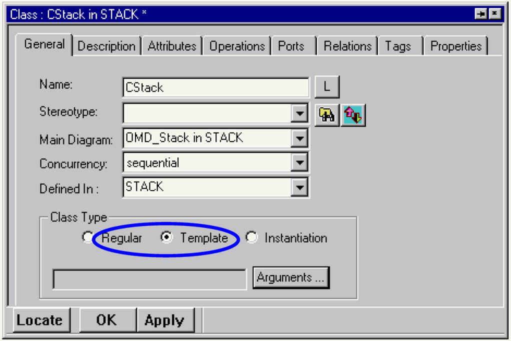


- Creating pure virtual methods:
  - additionally select the “**Abstract**” modifier under “**General**”

Universität Bremen

Institute of Automation

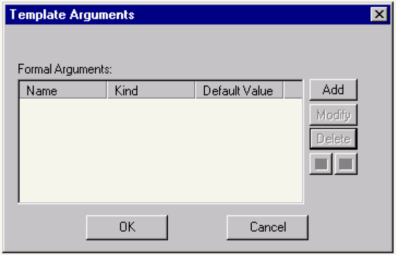
Definining template classes

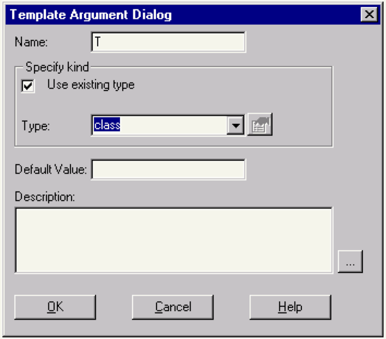


Universität Bremen

Institute of Automation

Settings for template arguments

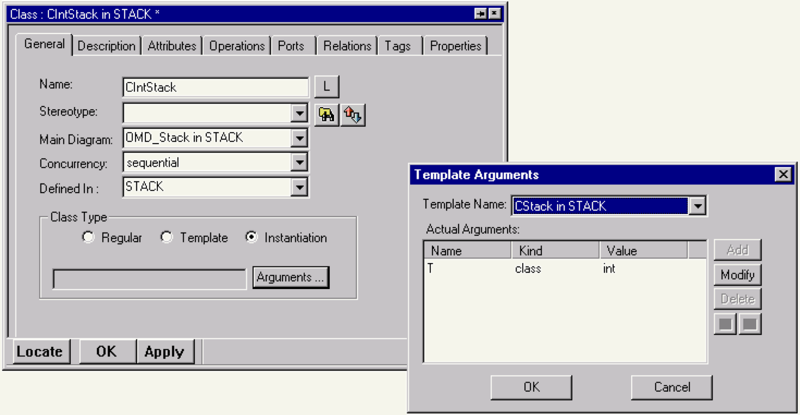




Universität Bremen

Institute of Automation

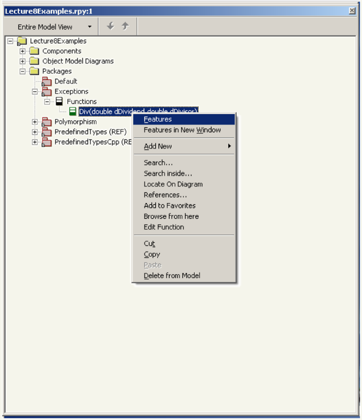
Template instantiation



Universität Bremen

Institute of Automation

Exception Handling (XII) – Rhapsody

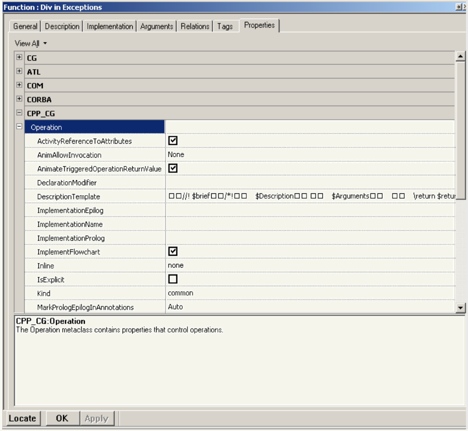


- Adding exception specifications to methods / functions
  - Select a method or function with right click
  - Choose “**Features**”

Universität Bremen

Institute of Automation

Exception Handling (XIII) – Rhapsody

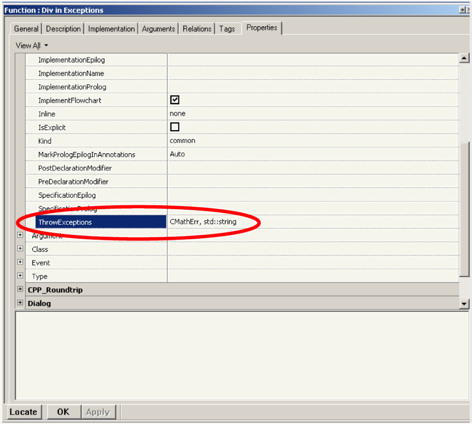


- Under "Properties" select category "**CPP_CG**"
- Select "**Operation**"

Universität Bremen

Institute of Automation

Exception Handling (XIIV) – Rhapsody



– Enter in the property **“ThrowExceptions”** a list of coma separated exception types

Universität Bremen