Institute of Automation

C++ Basics and Applications in technical Systems

Lecture 7 - Namespaces, Operators and this-pointer

Institute of Automation
University of Bremen

07th December 2012 / Bremen

WiSe 2012/2013

VAK 01-036

Universität Bremen

Institute of Automation

Overview

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

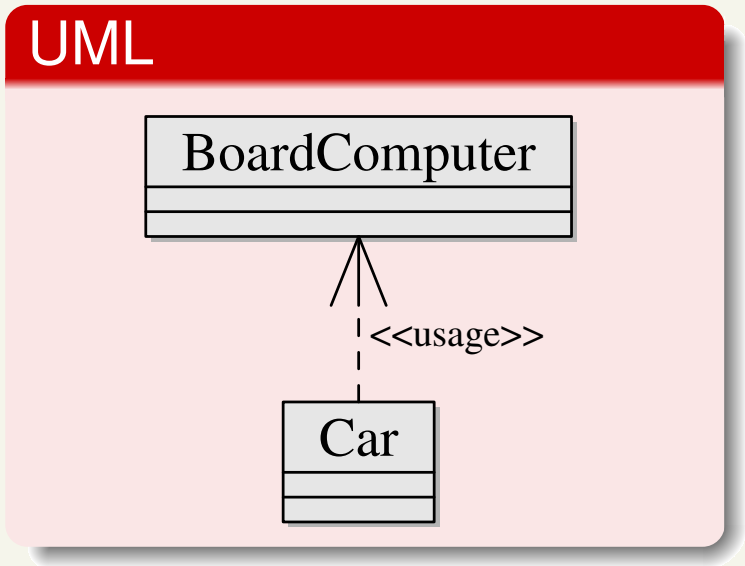
Bremen

Dependency

Dependencies are C++ `#include`-statements. In this example class `Car` uses class `BoardComputer` which is declared in file `BoardComputer.h`. It depends on it, meaning that a change in the class `BoardComputer` may cause a change in the class `Car`[1].

```
#include "BoardComputer.h"

class Car
{
public:
  void Start()
  {
    BoardComputer myComputer;
    myComputer.Enable();
  }
};
```

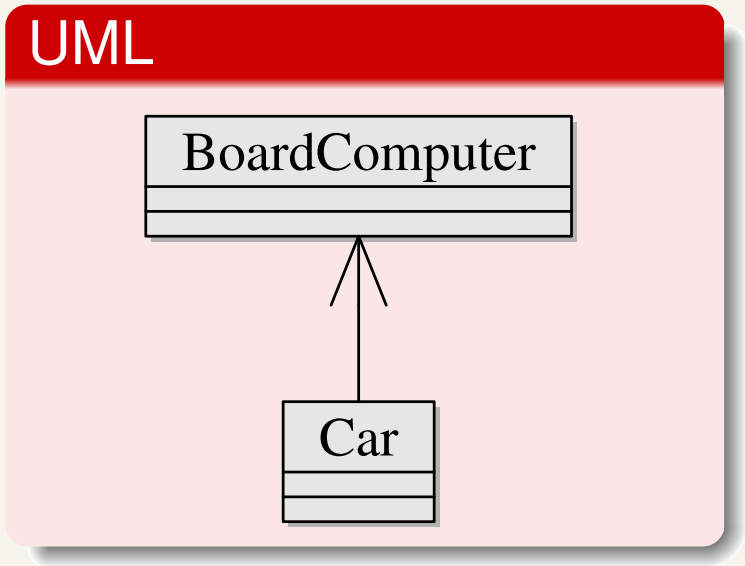


[1] This is only true as long as each class is stored in it's own module!

Association

If a class `BoardComputer` is associated[2] to a class `Car`, this means that the class `Car` has access to an object of type `BoardComputer` via a pointer. It does not own this linked object, so it must not e.g. delete it.

```
class Car
{
public:
  void TestBoardComputer(
   BoardComputer * pComputer);
};
```



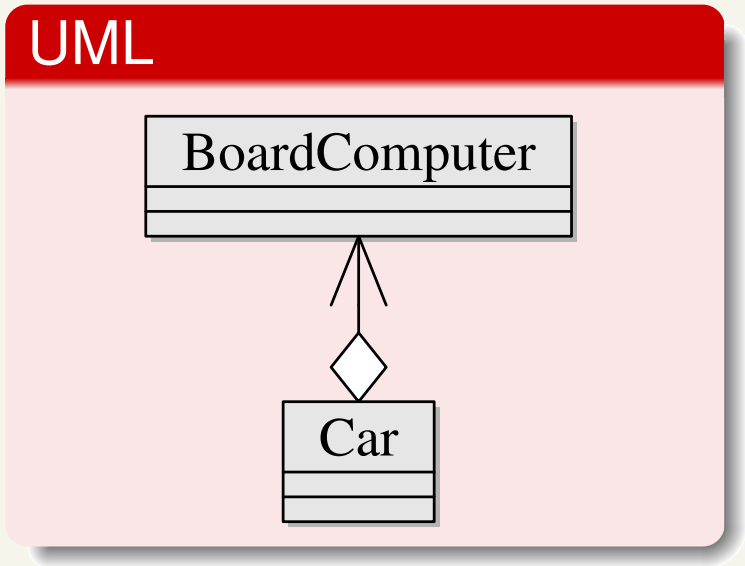
[2] Association is a special form of dependency!

Aggregation

If a class `BoardComputer` is aggregated[3] to a class `Car`, this means that class `BoardComputer` is a member of class `Car`. The ownership is via a pointer, so the linked object must be deleted by the class `Car` after usage.

```
class Car
{
public:
  void SetBoardComputer(
   BoardComputer * pComputer);

private:
  BoardComputer * m_pMyComputer;
};
```



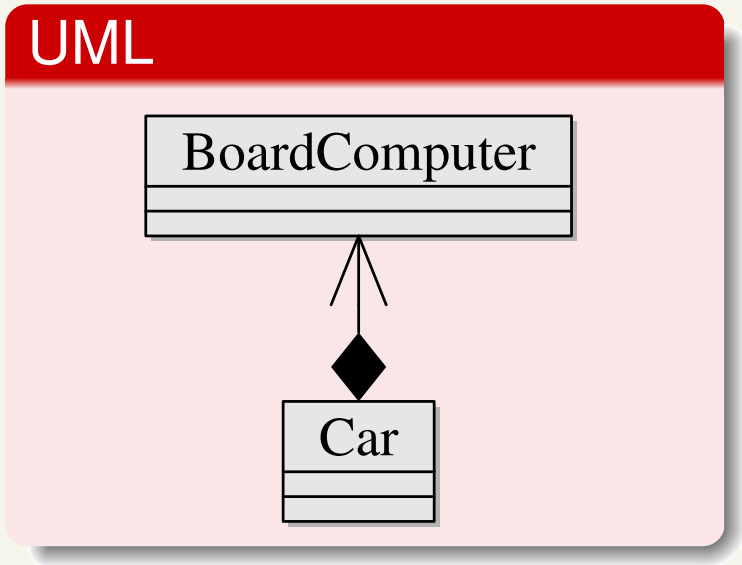
[3] Aggregation is a special form of association!

Composition

Institute of Automation

If a class `BoardComputer` is composited[4] by a class `Car`, this means that class `BoardComputer` is a member of class `Car`. The ownership is via an actual object (not a pointer).

```
class Car
{
private:
  BoardComputer m_MyComputer;
};
```



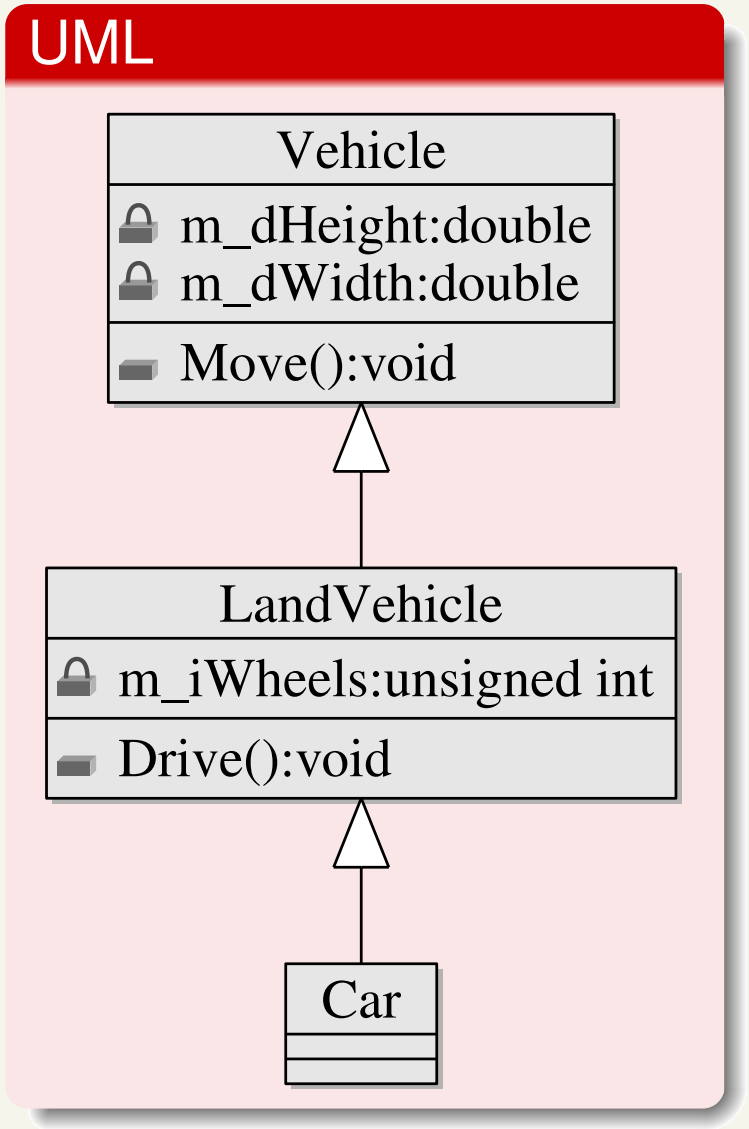
[4] Composition is a special form of association!

Inheritance terminology

- Top level class is called ***base class***.
- The class from which is inherited is called ***parent class***.
- The class that inherits is called ***derived*** or ***child class***.

The child class inherits from its parent class

- the characteristics (element variables)
- the behavior (member functions)



Inheritance of access rights

The weakest access right for the members of the inherited class is set to the access type of the inheritance.

Member \ Inheritance	`public`	`protected`	`private`
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`private`	`private`	`private`

Static class members

Institute of Automation

Static member variables are

- used to create class-global states/variables.
- valid for all objects of that class and only exist once in the memory.
- used e.g. for counting object instances or other shared information like common graphical origin on the screen.

Static methods

- for class-global operations not bound to a certain object (e.g., to operate on class related (static) data)

Universität Bremen

Implicit complexity of software

Institute of Automation

Frederick P. Brooks, Jr. in 1987:

„Complexity is one of the basic characteristics of software ...“

Reasons?

- Complexity of the problem.
- Problems to control the process of development.
- Requirements to the flexibility of software.
- In contrast to continuous systems it is difficult to describe the behaviour of discrete event systems.

Universität Bremen

Characteristics of complex systems

Institute of Automation

- Often complexity appears in form of hierarchies (subsystem $\rightarrow$ subsystem $\rightarrow$ ... $\rightarrow$ elementary component).
- The definition of elementary components (primitives) is arbitrary and depends on the designer of the system.
- Normally, relations inside components are stronger than between components.
- Typically, hierarchical systems consist only of a small number of different kinds of subsystems.
- It is for sure that a correct working complex system has been created by means of less complex and smaller subsystems.

Universität Bremen

Creation of sub systems in C++

Institute of Automation

Things you already know...

- **Modules in C++:**
  Header + implementation file (*.h/cpp)
- **Data as well as data-structures are encapsulated inside modules:**
  It is not allowed to offer access to internal data-structures by means of functions.
- **Principle of information hiding:**
  The user of the module can only access as much as necessary and however little as possible.

Universität Bremen

Repetition: Module structure tree

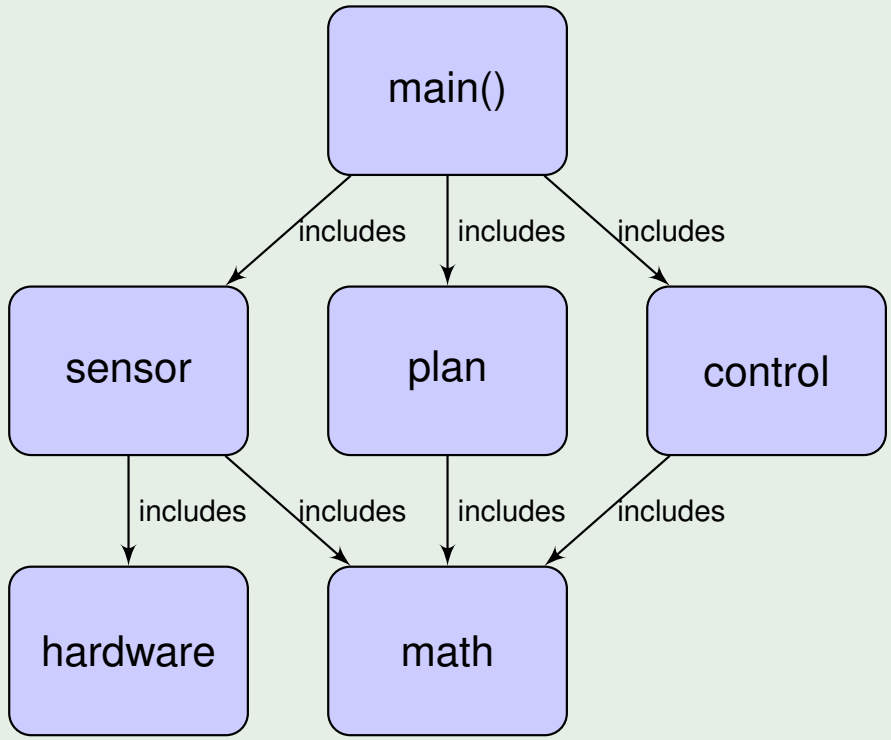
Institute of Automation

The arrangement of modules is arbitrary!

Rules of thumb:

- principle of cohesion
- functional decomposition
- encapsulation of interfaces (Hard- or Software)
- logical context (e.g. math library)

Example



Universität Bremen

Introduction of namespaces

Institute of Automation

Definition

A namespace

- is a mechanism for expressing logical grouping and a scope of visibility and validity.
- prevents name-conflicts between modules.
- is almost similar in form and function to a `struct`.
- differs from a struct in the following ways:
  - begins with the keyword namespace (instead of struct)
  - cannot be instantiated
  - may be defined only at the file scope or immediately within another namespace
  - does not end with „;“

Universität Bremen

Syntax of namespaces

Namespace can be used for:

- simplification of names.
- simplification of module assignments.

Example

```
namespace Math
{
  float Sin(float fArg);
  ...
}
...
float fResult = Math::Sin(90.0);
```

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

Small exercise

Institute of Automation

- Create the namespaces `Numbers` and `Letters`.
- Implement in both namespaces the function `Display()`:
  - `Numbers::Display()` should display „Number was called“
  - `Letters::Display()` should display „Letter was called“
- In the main function call both of the `Display()` functions.
- Try the three usage versions for the two namespaces!

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

Another this-Pointer example

Institute of Automation

The this-pointer can be used to check if an instance is passed to itself:

Example

```
Matrix::Assign(const Matrix & toBeAssigned)
{
  if (this != &toBeAssigned)
  {
    // Code for assignment
  }
  // Assigning an object to itself is non-sense
  // (and sometimes dangerous)
}
```

Universität Bremen

Motivation

Institute of Automation

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;`

Universität Bremen

Syntax

An operator can be overloaded as

- a global function.

```
ReturnType operator# (list of arguments)
{Code}
```

- an element-function.

```
ReturnType classname::operator# (list of arguments)
{Code}
```

represents all possible operator symbols of C++ (e.g., + or <<)

Operator as function-call

Institute of Automation

Elementfunction	Syntax	Replaced by
No	`X # Y`	`operator# (X,Y)`
No	`# X`	`operator# (X)`
No	`X #`	`operator# (X,0)`
Yes	`X # Y`	`X.operator# (Y)`
Yes	`# X`	`X.operator# ()`
Yes	`X #`	`X.operator# (0)`
Yes	`X = Y`	`X.operator= (Y)`
Yes	`X [Y]`	`X.operator[] (Y)`
Yes	`X ->`	`(X.operator-> () ) ->`

Universität Bremen

Restrictions

Institute of Automation

- overloading only for ordinary C++ operators possible

`+`	`-`	`*`	`/`	`%`	`^`
`&`	`|`	`~`	`!`	`=`	`<`
`>`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`
`||`	`++`	`--`	`->*`	`,`	`->`
`[]`	`()`	`new`	`delete`	`new[]`	`delete[]`

- no overloading of `.` , `.*` , `::` , `?:` and other symbols like `$` etc.
- no possibility for definition of new operators
- priority rules cannot be changed
- At least one of the arguments has to be a `class`-object or the operator function has to be an element function.

Universität Bremen

Global vs. Element-function

Institute of Automation

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.

Universität Bremen

Output operator

output `operator<<` by the example of the class `Rational`

Example

```
std::ostream& operator<< (std::ostream& out,
                          const Rational& rationalObj)
{
    out << rationalObj.GetEnumerator()
        << "/" << rationalObj.GetDenominator();
    return out;
}
```

Return of a reference to `ostream` to allow concatenation.

Example

```
std::cout << "r1 = " << rationalObj_1 << std::endl;
```

Assignment operator

Institute of Automation

Purpose:

- necessary for classes with references and pointers

Syntax (`A=B` with dynamic memory):

- prevention of self assignment
- deallocation of occupied memory of object `A`
- allocation of new memory for object `B` in size of object `A`
- copy of content from object `B` to object `A`

Universität Bremen

Example assignment operator

```
1  MyVector& MyVector::operator= (const MyVector& vectorObj)
2  {

1    if( this != &vectorObj )                    // check if assignment
2    {                                           // to itself

1      delete[] m_DynArray;                      // release memory

1      m_Number = vectorObj.GetNumber();
2      m_DynArray = new int[m_Number];           // reserve memory

1      int iCount;

1      for( iCount = 0; iCount < m_Number; iCount++ )
2      {
3        m_DynArray[iCount] = vectorObj.m_DynArray[iCount];
4      }

1    }

1    return *this;
2  }
```

Operator += for Rational

Institute of Automation

```
Rational& Rational::operator+= (const Rational& rationalObj)
{
  m_Enumerator = m_Enumerator * rationalObj.m_Denominator +
                 m_Denominator * rationalObj.m_Enumerator;

  m_Denominator = m_Denominator * rationalObj.m_Denominator;

  trim();

  return *this;
}
```

Return of a reference to `Rational` to allow concatenation.

Example

```
rationalObj_1 += rationalObj_2 + 2;
```

Universität Bremen

Operator +

Purpose:

- binary operator
- addition commutative, i.e., `x + y = y + x`
- arguments shall not be changed
- operator versions:
  - operator+ (x,y) - global operator
  - x.operator+ (y) - operator as member function

Institute of Automation

Operator + (possible cases)

addition of two rational numbers: `z = x + y`

- `z = x.operator+ (y)`
- `z = operator+ (x,y)`

addition of a rational and a natural numbers: `z = x + 3`

- `z = x.operator+ (3)`
- `z = operator+ (x,3)`

addition of a natural and a rational numbers: `z = 3 + y`

- `z = 3.operator+ (y)`
- `z = operator+ (3,y)`

Conclusion

Only realization as a global function is valid for the arithmetic operator +.

Bremen

Operator + for Rational

Institute of Automation

Already overloaded `operator+=` will be used for overloading `operator+` for the class `Rational`.

```
Rational operator+ (const Rational& ratObj_1,
                    const Rational& ratObj_2)
{
  Rational tempRational = ratObj_1;
  return tempRational += ratObj_2;
}
```

Universität Bremen

Index-operator

Task: `x = a[i];` or `a[i] = x;` with secure access to the elements

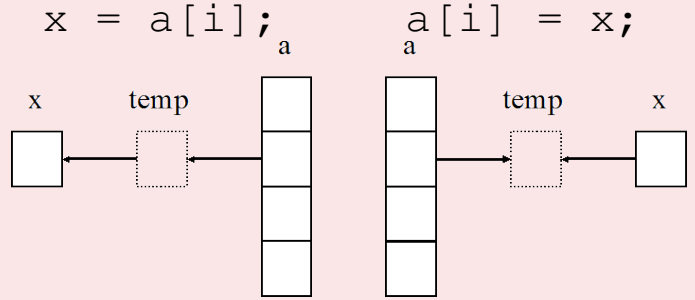
possible realization:

```
T operator[] (int index)
{
  ...
}
```

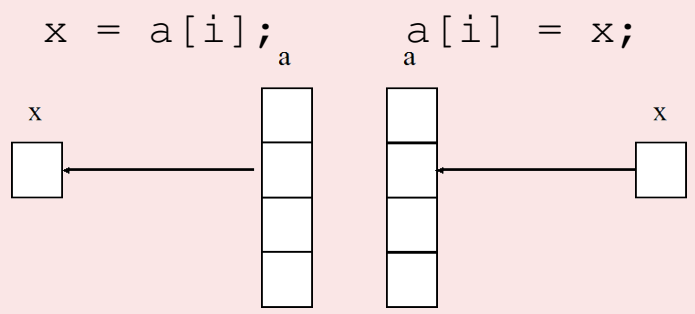
optimal realization:

```
T& operator[] (int index)
{
  assert( index >= 0 &&
          index < m_Number );
  return m_Start[index];
}
//constant objects...
const T& operator[] (int index) const
{ ... }
```

return of an object



return of an address



Small exercise

Institute of Automation

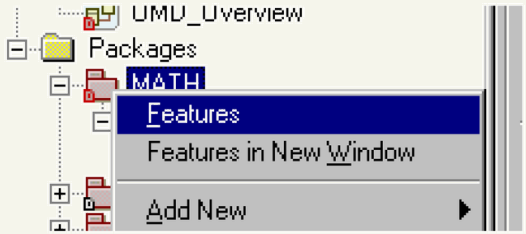
- Create the namespace `Math`.
- Design and implement a class `Rational` in `Math`.
- This class shall offer:
  - standard-constructor.
  - set- and get-methods for enumerator and denominator (division by zero?).
- Implement in `Math` the following operators:
  - + and * for addition and multiplication of two `Rational` objects.
  - `operator<<` for displaying the value held by a `Rational` instance.
  - Introduce an operator for assignment.
  - Introduce also operators for += and *=.
- Test the class.

Universität Bremen

Institute of Automation

Namespaces

- Introduce Namespace in Rhapsody

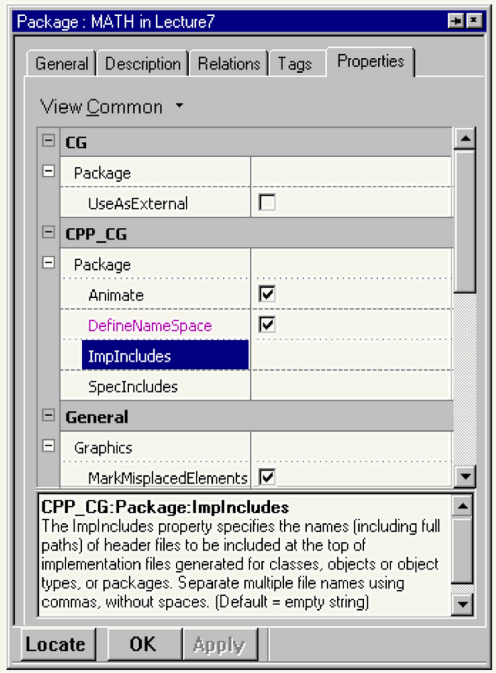



```
namespace MATH {

    // ************************
    class CRational  {

    public :

        ...
    };
}
```



Universität Bremen

Institute of Automation

Arithmetic Operators (Ia)

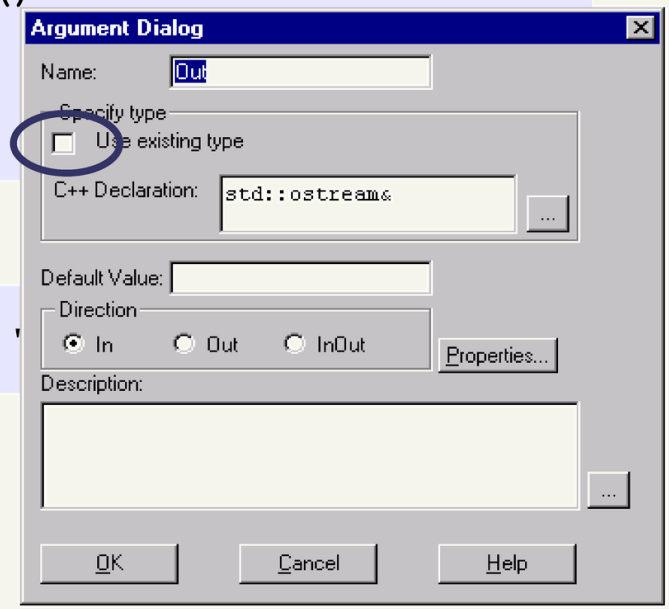
- Output `operator<<` by the example of the class `CRational`

```
std::ostream& operator<<(std::ostream& Out, const CRational& Rational) {

    Out   << "r = " << Rational.GetEnumerator()
          << "/" << Rational.GetDenominator();

    return Out;
}
```

```
std::cout << "r1 = " << Rational1 << ", r2 = "
```



Universität Bremen

Institute of Automation

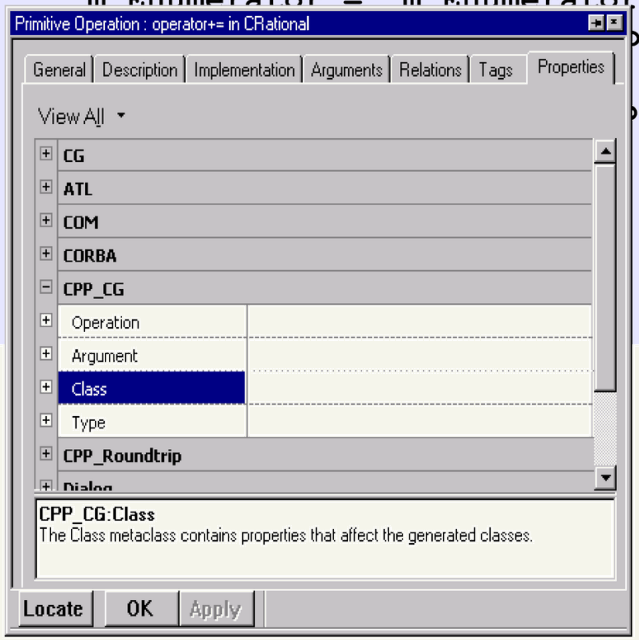
Arithmetic Operators (IIa)

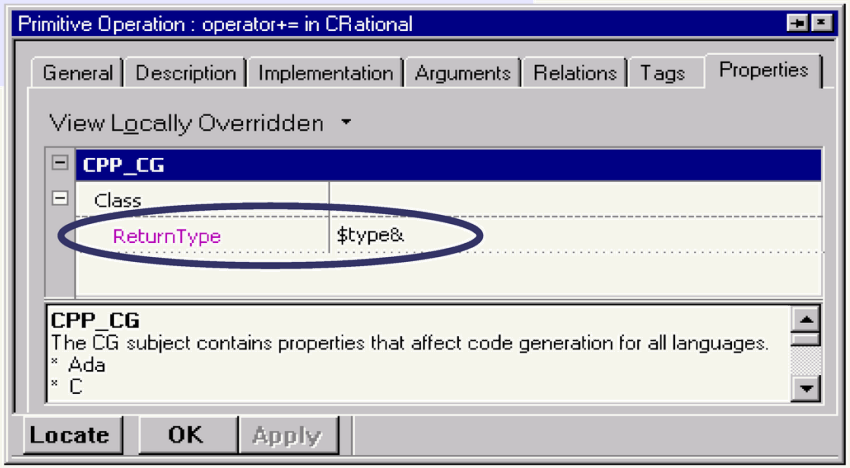
- '+=' - operator for **Rational**

```
CRational& CRational::operator+=(const CRational& Rational) {

    m_Enumerator =  m_Enumerator * Rational.m_Denominator +
                    r * Rational.m_Enumerator;

                    r * Rational.m_Denominator;
}
```





Universität Bremen