Praktikum: SystemC
C++-Labs

Joachim Falk (joachim.falk@fau.de)

Friedrich-Alexander-Universität Erlangen-Nürnberg

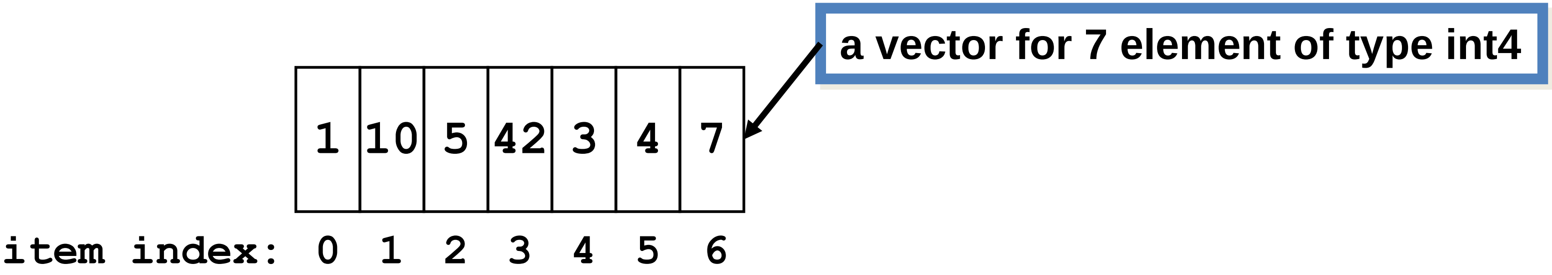
Agenda

1. **Writing a Vector Class**
2. **Constructor, References, Overloading**
3. **Templates, Virtual Functions**
4. **Standard Template Library (ADVANCED)**
5. **Smart Pointer (ADVANCED)**

Writing a Vector Class

1. Writing a simple vector class

a vector is an one-dimensional array of objects



a vector for 7 element of type int4

	1	10	5	42	3	4	7
item index:	0	1	2	3	4	5	6

start with a simple object

integer values - type int

to make future changes easier use a typedef - t_vector

Writing a Vector Class

1. Writing a simple vector class

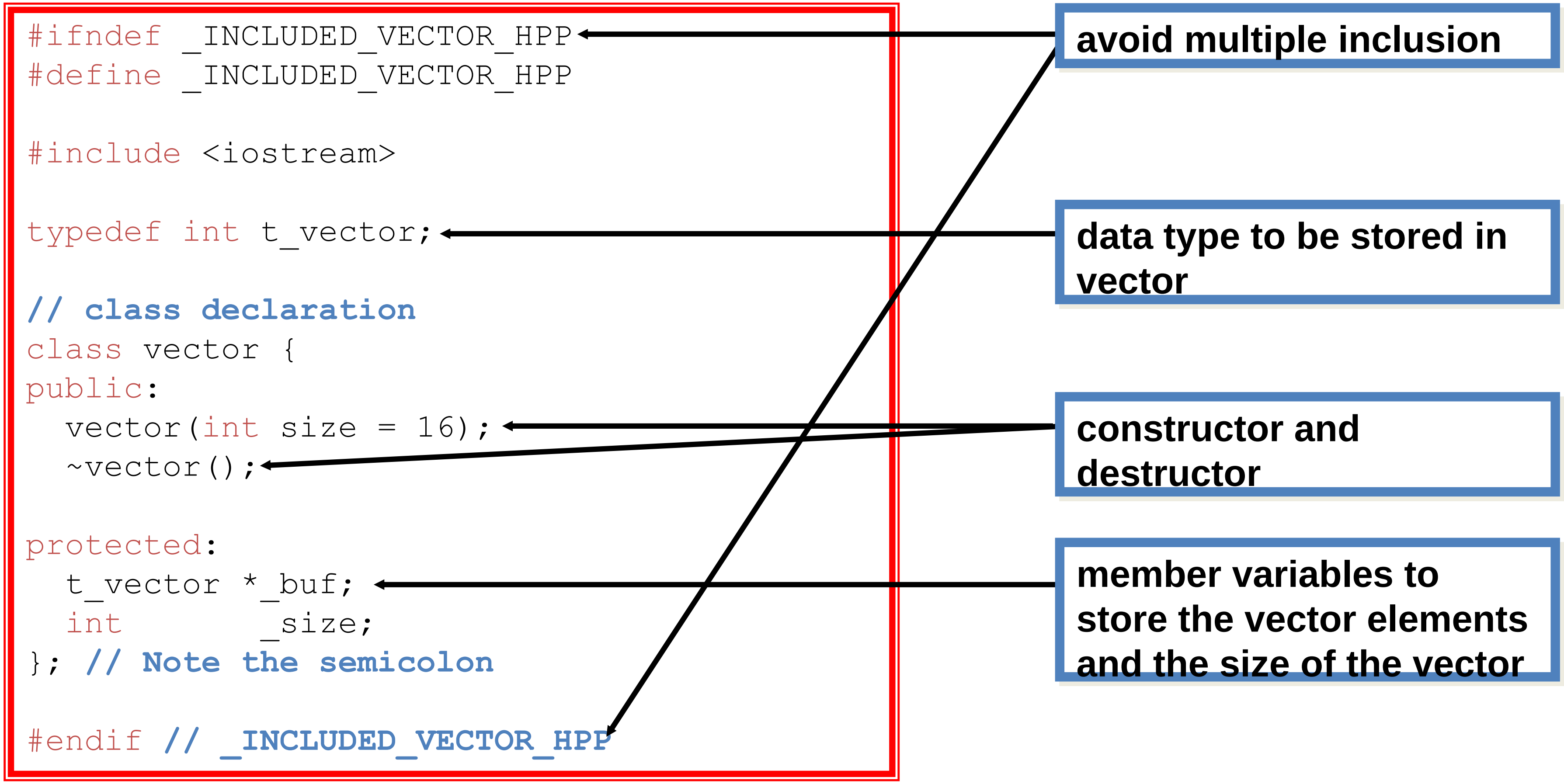
6. provide methods to

   create a vector of given size

   read/write to/from that vector (implemented later)

   destroy a vector without memory leakage

Vector Class| Header



```
#ifndef _INCLUDED_VECTOR_HPP
#define _INCLUDED_VECTOR_HPP

#include <iostream>

typedef int t_vector;

// class declaration
class vector {
public:
  vector(int size = 16);
  ~vector();

protected:
  t_vector *_buf;
  int       _size;
}; // Note the semicolon

#endif // _INCLUDED_VECTOR_HPP
```

avoid multiple inclusion

data type to be stored in vector

constructor and destructor

member variables to store the vector elements and the size of the vector

Vector Class| Implementation

```
// use header from previous slide
#include "vector.hpp"

vector::vector(int size) { // constructor
  _size = size;
  _buf = new t_vector[_size];

  for(int idx = 0;idx < _size;++idx) {
    _buf[idx] = -1;
  }
  std::cout << "vector of size: "
            << _size << " created [ ";
  for(int idx = 0; idx < _size;++idx) {
    std::cout << _buf[idx] << " ";
  }
  std::cout << "]" << std::endl;
}

vector::~vector() { // destructor
  delete[] _buf;
  std::cout << "vector of size: "
            << _size << " deleted”
            << std::endl;
}
```

allocate storage for vector elements

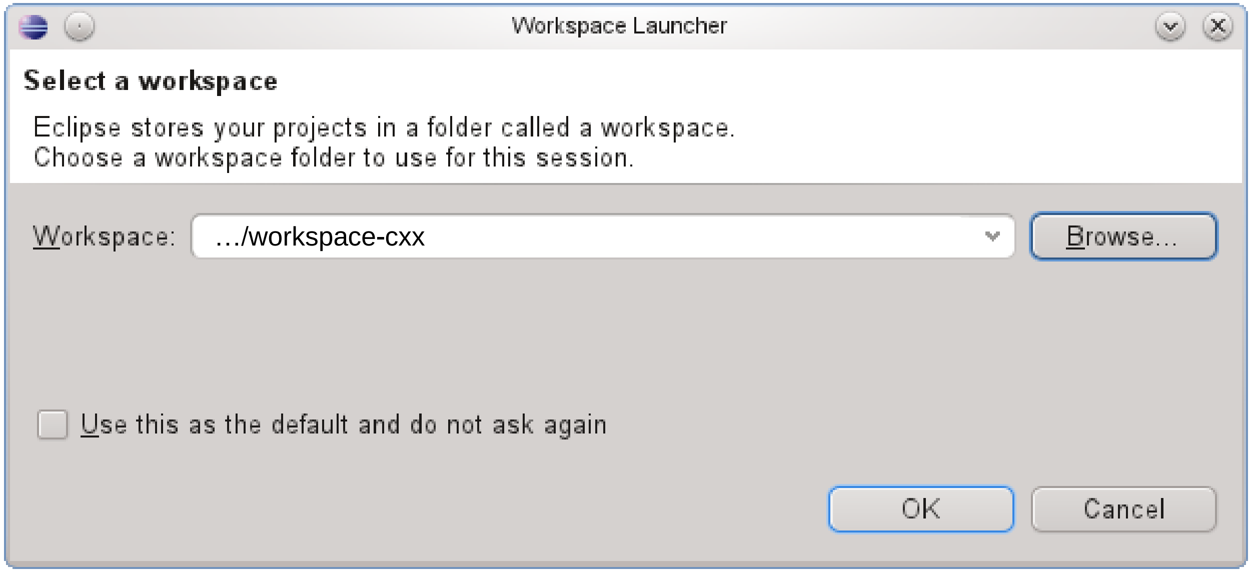
data type to be stored in vector

initialize vector elements to known value

free the storage allocated by the vector elements

Vector Class| Compile and Run

1. Open the Eclipse workspace “workspace-cxx”



Workspace Launcher

Select a workspace

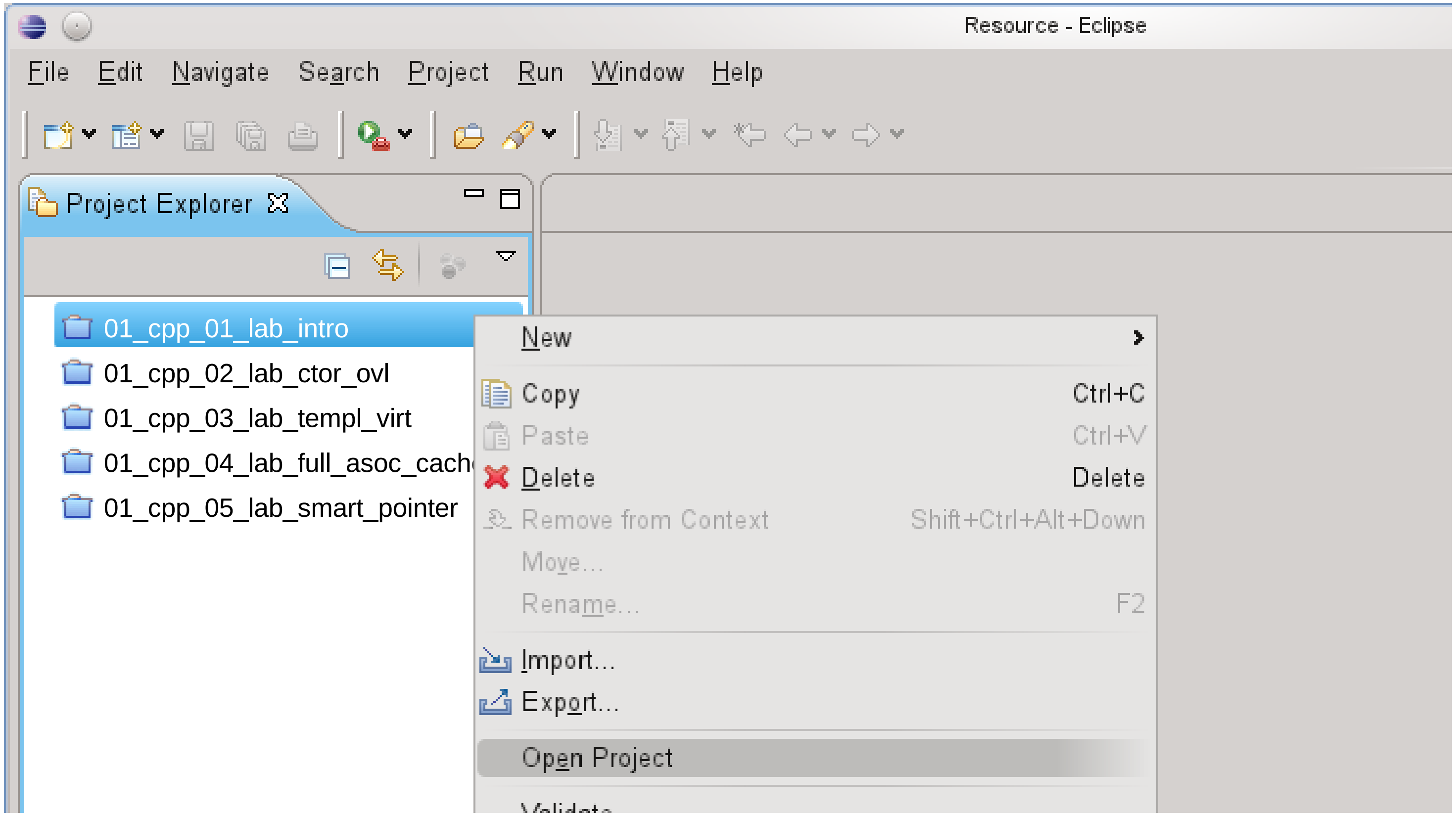
Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: …/workspace-cxx

□ Use this as the default and do not ask again

Vector Class| Compile and Run

1. Then open the 01_cpp project in there



Lab “01_cpp_01_lab_intro” 1/1

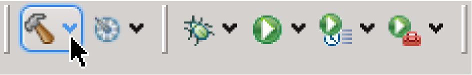
1. **Modify main.cpp to**

   instantiate vectors of size 2, 5 and 10

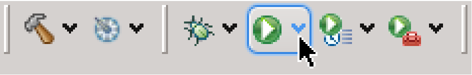
   explicitly call the destructor of one vector

2. **Compile and Run the program using the eclipse**

   “Build” menu



   “Run” menu



Agenda

1. **Writing a Vector Class**
2. **Constructor, References, Overloading**
3. **Templates, Virtual Functions**
4. **Standard Template Library (ADVANCED)**
5. **Smart Pointer (ADVANCED)**

Lab “01_cpp_02_lab_ctor_ovl”

For the vector class

1. a constructor with the size parameter and an optional parameter for the initial value is needed (default = 0)
2. the **read** method with two arguments that reads values from the vector is needed
   1. Argument 1: a reference to the value to be read
   2. Argument 2: the index of the value to be read
   3. The function has to implement a range check for the index argument

Lab “01_cpp_02_lab_ctor_ovl”

For the vector class

3. two operators have to be implemented

```
vector &operator   =(const vector &rhs);
vector &operator +=(const vector &rhs);
```

(implements pointwise addition; check if both vectors are of equal length)

References and Overloading

1. References and Overloading

1. in **vector.h**

   Extend the function prototype of the constructor to take two arguments (vector size and initial value) and declare the function prototype for the new read function that takes two arguments (value and index).

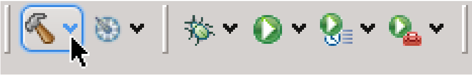
2. in **vector.cpp**

   Implement the element initialization in the constructor, the new **read** method, the **operator=**(), and the **operator+=**().

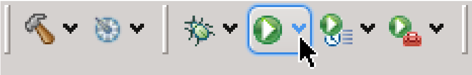
References and Overloading

2. Compile and Run the program using the eclipse

“Build” menu



“Run” menu



Agenda

1. Writing a Vector Class
2. Constructor, References, Overloading
3. **Templates, Virtual Functions**
4. **Standard Template Library (ADVANCED)**
5. **Smart Pointer (ADVANCED)**

Lab “01_cpp_03_lab_templ_virt”

Converting the vector to a template

Modify the vector class to be a template class that can store an arbitrary data type.

Test with class hierarchy of graph objects

Pure virtual base class **graph_obj** declares a method **area** to return the area. Concrete implementations derived from **graph_obj** (e.g., a **rectangle** and a **circle**) have to implement that method.

Store rectangles and circles within vector<rectangle> and vector<circle>

Templates

1. Templates

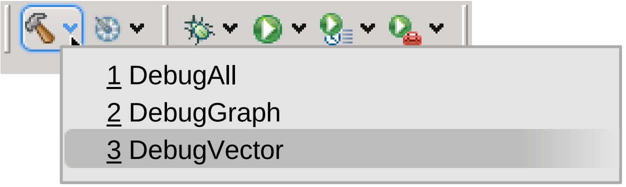
in **vector.h**

- modify the code to make vector a template class
  vector<T>
- **Hint: In our original code we used t_vector as a typedef for the data type to store in the vector!**
- **Hint: Have a look at the constructor, as it has already been transferred to a template style!**
- **Hint: Remember that the complete class implementation of a template class has to reside in the header file!**

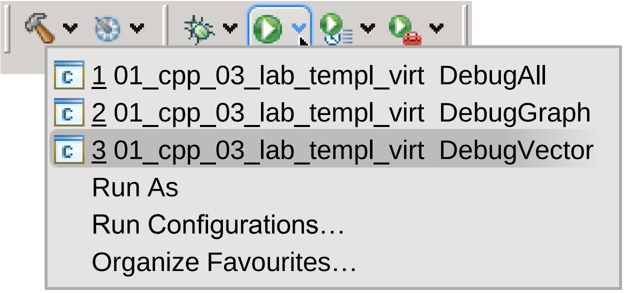
Templates

2. Compile and Run the program using the eclipse

DebugVector “Build” menu entry



DebugVector “Run” menu entry


1 01_cpp_03_lab_templ_virt DebugAll
2 01_cpp_03_lab_templ_virt DebugGraph
3 01_cpp_03_lab_templ_virt DebugVector
Run As
Run Configurations…
Organize Favourites…

Virtual Methods

3. Virtual Methods

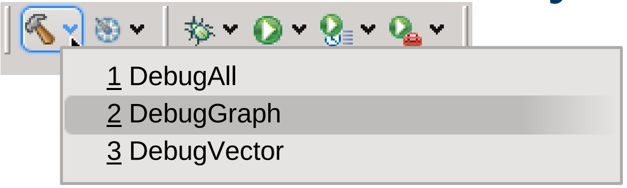
in **graph_obj.h**

- implement a class circc (for circle) that inherits from the virtual base class graph_obj
- the constructor should take the radius as an optional argument (default = 0.0)
- implement the method **area**()
- **Hint: Don't forget to implement a destructor as well!**

Virtual Methods

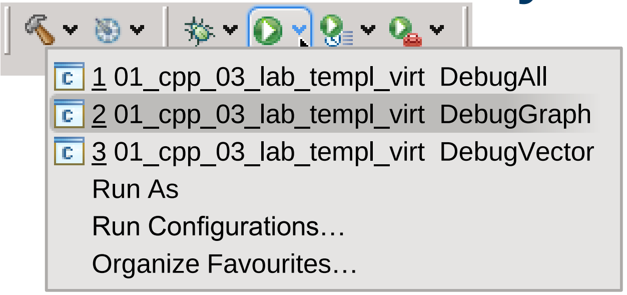
4. Compile and Run the program using the eclipse

DebugGraph “Build” menu entry



1 DebugAll
2 DebugGraph
3 DebugVector

DebugGraph “Run” menu entry



1 01_cpp_03_lab_templ_virt DebugAll
2 01_cpp_03_lab_templ_virt DebugGraph
3 01_cpp_03_lab_templ_virt DebugVector
Run As
Run Configurations…
Organize Favourites…

Integration Test

5. Integration Test

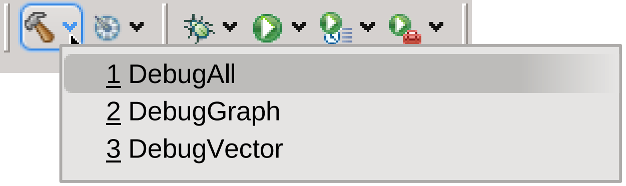
in **main.cpp**

- instantiate a vector of **rect** with 2 elements, the elements should have width=1, height=2
- instantiate a vector of **circ** with 3 elements, the elements should have radius=2

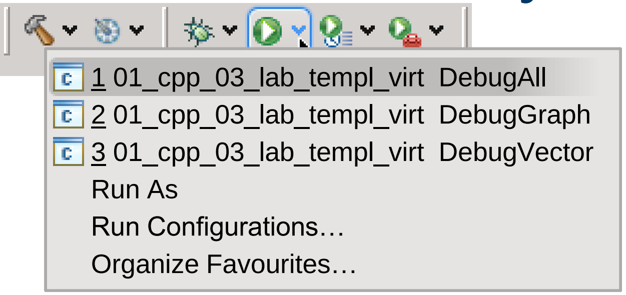
Integration Test

6. Compile and Run the program using the eclipse

DebugAll “Build” menu entry



DebugAll “Run” menu entry


1 01_cpp_03_lab_templ_virt DebugAll
2 01_cpp_03_lab_templ_virt DebugGraph
3 01_cpp_03_lab_templ_virt DebugVector
Run As
Run Configurations…
Organize Favourites…

Agenda

1. Writing a Vector Class
2. Constructor, References, Overloading
3. Templates, Virtual Functions
4. **Standard Template Library (ADVANCED)**
5. **Smart Pointer (ADVANCED)**

Lab “…04_lab_full_asoc_cache”

Problem

Associative hardware caches have fixed sizes and a replacement strategy

The C++ STL provides associative container classes, but these do not have a fixed size and no replacement strategy

Lab “…04_lab_full_asoc_cache”

Idea

Implement a fully associative cache as a template class **full_asoc_cache**<>, that uses the **map**<> container from the STL.

1. The data types for the key and for the entry are given as template parameters.
2. The size of the cache (the number of cache-lines) is given as constructor parameter.
3. To simplify the implementation, inserting a new entry into a full cache replaces a random cache line.

Lab “…04_lab_full_asoc_cache” 1/1

1. Standard Template Library

in **full_asoc_cache.h** implement following methods

```
bool get(const TAG_T&, ENTRY_T&);
```

Hint: Use the method find() from the class map<>

```
void insert(const TAG_T&, const ENTRY_T&);
```

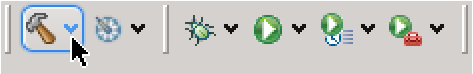
Hint: Use the operator[] from the class map<>

```
void erase(const TAG_T&);
void clear();
```

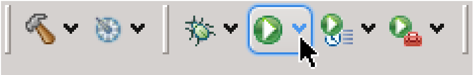
Lab "…04_lab_full_asoc_cache" 1/1

2. Compile and Run the program using the eclipse

"Build" menu



"Run" menu



Output:

```
re:10.1 im:0
re:12.1 im:0.2
re:14.1 im:0.4
Re:15.1 im:0.5
```

Agenda

1. Writing a Vector Class
2. Constructor, References, Overloading
3. Templates, Virtual Functions
4. Standard Template Library (ADVANCED)
5. **Smart Pointer (ADVANCED)**

Lab “01_cpp_05_smart_pointer”

Problem

Unlike Java, C++ provides no built-in garbage collector that deletes unreferenced objects, thus eliminating memory leaks.

Smart pointers manage reference counts for every allocated object and are, thus, able to know when the last reference to an object is gone to trigger deletion of the object.

Lab “01_cpp_05_smart_pointer”

Idea

1. Implement a template class **smart_ptr**<> that represents a pointer to a given object type **T**
2. Copy Constructors and the Assignment Operators have to manage the reference counts
3. The Destructor and the Assignment Operators may delete the referenced object
4. A **smart_ptr**<> can be created from a pointer to an object of type T
5. A common reference count value is allocated if the pointer is not NULL

Lab “01_cpp_05_smart_pointer”

1. Smart Pointer

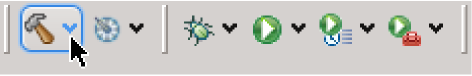
in **smart_ptr.h**

1. implement a constructor to create a **smart_ptr**<**T**> from a pointer **T***
2. implement the copy constructors and the assignment operator with reference counting
3. implement the destructor and avoid memory leaking
4. implement the missing operators to create a complete smart pointer

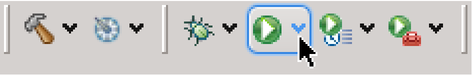
Lab “01_cpp_05_smart_pointer”

2. Compile and Run the program using the eclipse

“Build” menu



“Run” menu



Output:

```
*ptr3 = black-colored car with speed 12.0416
*ptr4 = silver-colored jet with speed 100.125
*ptr5 = 42
```