Agenda

C++ Introduction

1. Writing a Vector Class Introduction
2. Labs

Writing a simple vector class

- a vector is an one-dimensional array of objects
- start with a simple object
  - integer values - type `int`
  - to make future changes easier use a typedef - `t_vector`
- provide methods to
  - create a vector of given size
  - read/write to/from that vector (implemented later)
  - destroy a vector without memory leakage

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

a vector for 7 element of type `int`

C++ - Writing a Vector Class Introduction

```
#ifndef VECTOR_H
#define VECTOR_H

#include <iostream>

typedef int t_vector;

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

protected:
  t_vector* _buf;
  int _size;

};

#endif // VECTOR_H
```

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

Fraunhofer Institut Integrierte Schaltungen

C++ - Writing a Vector Class Introduction

```
#include "vector.h“

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

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

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

allocate storage for vector elements

initialize vector elements to known value

free the storage allocated by the vector elements

Fraunhofer Institut Integrierte Schaltungen

Agenda

C++ Introduction

1. Writing a Vector Class Introduction
2. Labs

   2.1 Introduction

   2.2 Constructor, References, Overloading

   2.3 Templates, Virtual Functions

   2.4 Standard Template Library (ADVANCED)

   2.5 Smart Pointer (ADVANCED)

C++ lab_intro - Problem and Task

Compiling and running a simple C++ program

- Directory: `lab_intro`
- compile and run the program using
  - `make`
  - `lab_intro.x`
- modify `main.cpp` to
  - instantiate vectors of size 2,5 and 10
  - try to explicitly call the destructor of one vector

Agenda

C++ Introduction

C++ lab_ctor_ovl - Problem

For the vector class

- a constructor with an optional parameter for the initial value is needed (default = 0)
- a function with two arguments that reads values from the vector is needed
  - argument 1: a reference to the value to be read
  - argument 2: the index of the value to be read
  - the function has to implement a range check for the index argument
- two operators have to be implemented for the vector
  - `vector& operator=(const vector& rhs);`
  - `vector& operator+=(const vector& rhs);`

C++ lab_ctor_ovl - Task

Constructor, References and Overloading

- directory: `lab_ctor_ovl`
- in `vector.h`
  - extend the function prototype of the constructor to take two arguments (vector size and initial value)
  - give the function prototype for the new `read()` function that takes two arguments (value and index)
- in `vector.cpp`
  - implement the element initialization in the constructor
  - implement the new `read()` method
  - implement the `operator=()`
  - implement the `operator+=()`
- compile run and debug your program using
  - `make`
  - `lab_ctor_ovl.x`

Agenda

C++ Introduction

1. Writing a Vector Class Introduction
2. Labs

    2.1 Introduction

    2.2 Constructor, References, Overloading

    2.3 Templates, Virtual Functions

    2.4 Standard Template Library

    2.5 Smart Pointer

C++ lab_templ_virt - Problem

Making the `vector` class a template class

- modify the `vector` class to be a template class that can store an arbitrary data type

Creating a class hierarchy for graphical objects

- pure virtual base class `graph_obj`
- declares a method `area()` to return the area
- a concrete implementation derived from `graph_obj` (e.g. a rectangle) has to implement that method

Storing graphical objects within the `vector` class

- use the new template version of the `vector` class to store graphical objects (e.g. a rectangle)

C++ lab_templ_virt - Task 1

Class Templates, Virtual Methods and Classes

- directory: `lab_templ_virt`
- 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. That has already been transferred to a template style
  - Remember: the complete class implementation of a template class has to reside in the header file
- compile run and debug your program using
  - `make -f Makefile_vector`
  - `tst_vector.x`

C++ lab_templ_virt - Task 2

Class Templates, Virtual Methods and Classes

- directory: `lab_templ_virt`
- in `graph_obj.h`
  - implement a class `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.
- compile run and debug your program using
  - `make -f Makefile_graph_obj`
  - `tst_graph_obj.x`

C++ lab_templ_virt - Task 3

Class Templates, Virtual Methods and Classes

- directory: `lab_templ_virt`
- 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
- compile run and debug your program using
  - `make`
  - `lab_templ_virt.x`

Agenda

C++ Introduction

1. Writing a Vector Class Introduction
2. Labs

   2.1 Introduction

   2.2 Constructor, References, Overloading

   2.3 Templates, Virtual Functions

   2.4 Standard Template Library

   2.5 Smart Pointer

C++ lab_full_asoc_cache - Problem

Problem:

- Associative hardware caches have fixed sizes and given replace strategies
- The C++ STL provides associative container classes, but these do not have a fixed size and no replace strategy

Idea:

- Implement a fully associative cache as a template class `full_asoc_cache<>`, that uses the `map<>` container class from the STL
- The data types for the *key* and for the *entry* are given as template parameters
- The size of the cache (the number of cache-lines) is given as constructor parameter
- To simplify the implementation, inserting a new entry into a full cache replaces a *random* cache line

C++ lab_full_asoc_cache - Task

Task:

- directory: `lab_full_asoc_cache`
- implement following methods in `full_asoc_cache.h`
  - `bool get(const TAG_T&, ENTRY_T&);`
    - use the method `find()` from the class `map<>`
  - `void insert(const TAG_T&, const ENTRY_T&);`
    - use the `operator[]` from the class `map<>`
  - `void erase(const TAG_T&);`
  - `void clear();`

Compilation:

- compile, run and debug your program using
  - `make`
  - `./stl.x`

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

C++ Introduction

1. Writing a Vector Class Introduction
2. Labs

   2.1 Introduction

   2.2 Constructor, References, Overloading

   2.3 Templates, Virtual Functions

   2.4 Standard Template Library

   2.5 Smart Pointer

C++ lab_smart_pointer - Problem

Problem:

- Unlike Java, C++ provides no built-in garbage collector that deletes unreferenced objects and thus eliminate memory leaks
- Smart pointers that manage reference counts for every allocated object are able to know when the last reference to an object is gone and thus delete the object

Idea:

- Implement a template class `smart_ptr<>` that represents a pointer to a given object type `T`
- The Copy Constructors and the Assignment Operators have to manage the reference counts
- The Destructor and the Assignment Operators may delete the referenced object
- A `smart_ptr<>` can be created from a pointer to an object of type `T`
- A common *reference count* value is allocated if the pointer is not `0` (what is the default value)

Fraunhofer Institut Integrierte Schaltungen

C++ lab_smart_pointer - Task

Task:

- directory: `lab_smart_pointer`
- in `smart_ptr.h`
  - Implement a constructor to create a `smart_ptr<T>` from a pointer `T*`
  - Implement the copy constructors and the assignment operator with *reference counting*
  - Implement the destructor and *avoid memory leaking*
  - Implement the missing operators to create a complete smart pointer

Compilation:

- compile, run and debug your program using
  - `make`
  - `./smart_ptr.x`

Output:

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