Introduction to Computer Graphics
Practical Session – C++

In this second practical session you will learn how to implement a basic C++ program. When creating this pratical we assumed that you know Java. If this is not the case it should be fairly easy to understand this pratical by either asking the assistants or by using your favorite search engine. This practical will not be graded but we advice you to work through it if you are not familiar with C++.

1 Hello World

In this section we will first create a CMake to automatically generate a *Makefile* for Mac or Linux or a Visual Studio solution for Windows. First create a file called *main.cpp* and add the following code:

```
///========================== main.cpp ==========================
/// iostream provides basic input/output functionalities (cout)
#include <iostream>
/// iostream is part of the "std" namespace. The keyword "using" can
/// be employed to avoid typing std::cout throughout the code
using namespace std;

/// main is the entry point of your program. It needs to be
/// defined and always have this signature. The argc and argv
/// are used to pass parameters from command line.
int main(int /*argc*/, char** /*argv*/){
    /// @todo find out how to print "helloworld" to terminal
    /// @solution
    cout << "helloworld" << endl;

    /// The main function should always return a value.
    /// This tells the application whether it exited
    /// correctly or not (legacy error management)
    return EXIT_SUCCESS;
}
```

The *main* function is the entry point of the program and needs to always be defined. At that point you will need to add a line of code to print "helloworld" in the terminal. Your task is now to create a *CMakeLists.txt* that you will then load into CMake (as you have done in the previous pratical). Copy the following lines into the *CMakeLists.txt* file:

```
# Declaring the used CMAKE version is mandatory
cmake_minimum_required(VERSION 2.8)

# This defines the name of our project
# It will be the Visual Studio Solution name
project(Practical2)

# This creates an executable called "main.exe".
# By compiling everything that comes after (just "main.cpp" here)
add_executable(main main.cpp)
```

Look at these lines and read the comments to understand the CMake syntax. You can now load this file into CMake to generate the makefile. Compile and run you C++ program and make sure that you can see the ”helloworld” text printed in the console. More details on CMake can be found at `http://www.cmake.org/cmake/help/cmake_tutorial.html`.

2 Funtions

In this section we will learn how to create simple functions in C++.

2.1 Passing Parameters by Value

Create a function that self-incremented the value passed as parameter by 1 (using the ”++” operator) and that returns it. Call this function from the *main* function and print the original and incremented value in the terminal.

```
///========================== main.cpp ==========================
#include <iostream>
using namespace std;

int myfunction(int value){
    value = value+1;
    return value;
}

int main(int, char**){
    /// @todo
    /// 1) declare (and initialize!!) a variable "int a"
    /// 2) implement and call "myfunction(...)"
    /// 3) print the original and incremented value
    /// QUESTION) what happens if you don't initialize a?
    /// QUESTION) what happens if you declare "int a" twice?
    /// @solution
    int a = 0;
    int b = myfunction(a);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;

    return EXIT_SUCCESS;
}
```

2.2 Passing Parameters by Reference

You may recall that in Java objects are automatically passed by reference and native types (i.e., int, float, ...) are passed by value. This is not the case in C++ where references need to be declared manually.

```
///========================== main.cpp ==========================
#include <iostream>
using namespace std;

void myfunction_reference(int& val){
    val++;
}

int main(int, char**){
    /// @todo
    /// 1) declare (and initialize!!) a variable "int c"
    /// 2) implement and call "myfunction_reference(...)"
    /// 3) print the original and incremented value
    /// @solution
    int c = 0;
    myfunction_reference(c);
    cout << "c: " << c << endl;

    return EXIT_SUCCESS;
```

```
}
```

Complete the code above following the instructions written in the comments. Look at the output in the terminal. What is the difference with the code implemented in the Section 2.1. What is the meaning of the ”&” operator?

2.3 Passing Parameters by Pointer

Pointers can be considered similar to references that are more powerful but less safe. Complete this code by following the inlined instructions.

```cpp
///========================== main.cpp ==========================
#include <iostream>
using namespace std;

void myfunction_pointer(int* val){
    /// @todo increment the value by one
    /// @note val++ will not work, why?
    /// @solution
    (*val)++;
}

int main(int, char**){
    /// @todo
    /// 1) declare (and initialize!!) a variable "int d"
    /// 2) implement and call "myfunction_pointer(...)"
    ///    @attention you need to obtain a pointer from "d" using "&"
    /// 3) print the original and incremented value
    /// @solution
    int d = 0;
    myfunction_pointer(&d);
    cout << "d: " << d << endl;

    return EXIT_SUCCESS;
}
```

Homework: We *strongly* invite you to read a tutorial on pointers such as `http://www.cplusplus.com/doc/tutorial/pointers/`.

2.4 Templated Function

Similar to Java templates are part of the C++ language. Templates are a very powerfull tool allowing to define a function that uses generic types. Look at the following code to understand how a templated function is defined.

```cpp
///========================== main.cpp ==========================
#include <iostream>
using namespace std;

template <class Derived>
void myfunction_templated(Derived& val){
    /// @todo divide val by two
    /// @solution
    val/=Derived(2.0);
}

int main(int, char**){
    /// @todo
    /// 1) declare (and initialize!!) a variable "int e"
    /// 2) declare (and initialize!!) a variable "float f"
    /// 3) implement and call "myfunction_template(...)"
    /// 4) force a call to the "float" version of the template
    ///    function even though you pass an integer as parameter
    /// @solution
    int e = 2.0;
    float f = 2.0;
    myfunction_templated(e);
    myfunction_templated(f);
    cout << e << endl;
    cout << f << endl;

    return EXIT_SUCCESS;
}
```

In this example the template type will be automatically inferred from the function parameter. However, a tempate function can be called explicitly with a fix type in the following manner:

```
        myfunction_template<float>(...);
```

3 Memory Allocation

Contrary to Java **the memory is not automatically managed** and it will be your duty to allocate and deallocate the memory. The allocation can be done using the C++ operator *"new"* and the deallocation using *"delete"*. Note that for an array the operator *"delete []"* needs to be called instead. Read this code very carefully and try to understand the difference between automatic (stack) versus dynamic (heap) allocation `http://www.cplusplus.com/doc/tutorial/dynamic/`.

```
///========================== main.cpp ==========================
#include <iostream>
using namespace std;

int main(int, char**){
    /// @todo Read carefully to understand memory management
    {
        /// This memory is allocated on the stack (at compile-time)
        float array_1[10]; ///< the size *must* be a constant!!
        /// This memory is allocated on the heap (at runtime)
        float* array_2 = new float[10];

        /// @note array-2 needs to be deleted manually
        /// Otherwise we will have a MEMORY LEAK!!!!
        delete[] array_2;
    }
    /// array-1 will be automatically de-allocated here

    return EXIT_SUCCESS;
}
```

4 Classes

Create a file *Shape.h* and a file *Shape.cpp* and add them to the *CMakeLists.txt*. These two files contain the definition of your first C++ class.

4.1 Member Function

In this section you will implement member functions of a C++ class `http://www.cplusplus.com/doc/tutorial/classes/`. Similar to Java a class contains a *constructor* that is called when the object is created. As in C++ the memory is not managed a class also contains a *destructor* that is called when the object is deleted (for example the destructor should be used to delete previously allocated memory). A dummy default constructor and destructor are automatically generated by the compiler if none are provided.

Another difference from Java to C++ is that the classes are usually not defined in a single file but split in an *h* file where the functions are declared and a *cpp* file where the functions are defined. Look at the following code and try to understand how to implement a C++ class. Add a member function that ouputs the name of the class and add a destructor that deletes the memory allocated by the constructor (remember that you will need to use *delete []* and not *delete* in this case). Modify the main function following the inlined instructions.

```
///============================ Shape.h ==========================
#pragma once
#include <iostream>

class Shape{
protected:
    float* data;
public:
    Shape(int size);
    /// @todo Declare the destructor here
    virtual ~Shape();
    /// @todo Declare "member_function()" here
    void member_function();
};
/// @note DO NOT forget the semicolon!!!
```

```
///========================== Shape.cpp =========================
#include "Shape.h"

Shape::Shape(int size){
    /// @attention memory allocated here!!!
    data = new float[size];
}

/// @todo Define the destructor here
/// 1) print "~Shape()" to terminal (to know which destructor is called)
/// 2) deallocate the memory pointed to by the member "data"
Shape::~Shape(){
    std::cout << "~Shape()" << std::endl;
    delete[] data;
}

/// @todo Define "member_function()" here
void Shape::member_function(){
    std::cout << "Shape::member_function()" << std::endl;
}

/// @todo Define "virtual_member_function()" here
/// The function prints "Shape::virtual_member_function()"
```

```
///========================== main.cpp ==========================
#include <iostream>
#include "Shape.h"
using namespace std;

int main(int, char**){
    /// @todo
    /// 1) Create a class Shape (Shape.h/.cpp)
    ///    - define an appropriate constructor
    ///    - add a public member function
    ///      - declare the function in "Shape.h"
    ///      - define the function in "Shape.cpp"
    ///      - the function body prints "Shape::member_function()"
    /// 2) Instantiate an object of class Shape
    /// 3) Call the member function
    /// @solution
    Shape shape(10);
    shape.member_function();

    return EXIT_SUCCESS;
}
```

4.2 Inheritance

You have most probably seen inheritance during your Java course. In Java the functions are automatically declared *virtual*, however this is not the case for C++, where you need to use the *virtual* keyword to do so. If the word *virtual* does not mean anything to you then you probably need to read the tutorial `http://www.cplusplus.com/doc/tutorial/polymorphism/`. Maybe this example will also help you to understand. Follow the inline instructions and look at the terminal output. What should you do for the desctructor?

```cpp
///=========================== Shape.h ==========================
#pragma once
#include <iostream>

class Shape{
private:
    float* data;
public:
    Shape(int size);
    /// @todo Declare the destructor here
    virtual ~Shape();
    /// @todo Declare "member_function()" here
    void member_function();
    /// @todo Declare "virtual_member_function()" here
    virtual void virtual_member_function();
};

/// This is a derived class of Shape
class Square : public Shape{
public:
    Square(int size):Shape(size){}

    ~Square(){ std::cout << "~Square()" << std::endl; }

    void virtual_member_function(){
        std::cout << "Square::virtual_member_function()" << std::endl;
    }
};

/// This is another derived class of Shape
class Circle : public Shape{
public:
    Circle(int size):Shape(size){}

    ~Circle(){ std::cout << "~Circle()" << std::endl; }

    void virtual_member_function(){
        std::cout << "Circle::virtual_member_function()" << std::endl;
    }
};
```

```cpp
///========================= Shape.cpp =========================
#include "Shape.h"

Shape::Shape(int size){
    /// @attention memory allocated here!!!
    data = new float[size];
}

/// @todo Define the destructor here
/// 1) print "~Shape()" to terminal (to know which destructor is called)
/// 2) deallocate the memory pointed to by the member "data"
Shape::~Shape(){
    std::cout << "~Shape()" << std::endl;
    delete[] data;
}

/// @todo Define "member_function()" here
void Shape::member_function(){
    std::cout << "Shape::member_function()" << std::endl;
}
```

```
/// @todo Define "virtual_member_function()" here
/// The function prints "Shape::virtual_member_function()"
void Shape::virtual_member_function(){
    std::cout << "Shape::virtual_member_function()" << std::endl;
}
```

```
///========================== main.cpp ==========================
#include <iostream>
#include "Shape.h"
using namespace std;

int main(int, char**){
    /// @todo read the code below and execute the program
    Shape* shape_ptr  = new Shape(10);
    Shape* square_ptr = new Square(10);
    Shape* circle_ptr = new Circle(10);

    /// @note after this point the program only works on pointers of type Shape*!!
    /// The program doesn't need to know how "Square" and "Circle" are defined.

    /// @note Classical member functions *do not* use Polymorphism
    /// consequently they will only call the member function from "Shape"
    cout << endl << "Public members" << endl;
    shape_ptr->member_function();
    square_ptr->member_function();
    circle_ptr->member_function();

    /// @note Virtual member functions use Polymorphism
    /// even though we only have a pointer to Shape, the specialized
    /// functions will be called!
    cout << endl << "Virtual members" << endl;
    shape_ptr->virtual_member_function();
    square_ptr->virtual_member_function();
    circle_ptr->virtual_member_function();

    /// @note Note how only "~Shape" is called. This is EXTREMELY bad if memory
    ///       was allocated in one of the subclasses as it would result in a leak.
    /// @todo what do you need to do to call the correct (inherited) destructors?
    /// @solution the destructor needs to be declared virtual!
    cout << endl << "Destructors" << endl;
    delete shape_ptr;
    delete square_ptr;
    delete circle_ptr;

    return EXIT_SUCCESS;
}
```

5 STL

The Standard Template Library (STL) is a software library provided with C++. This library contains useful classes for C++, such as containers. In this section you will use the *vector* container (similar to the Java *vector* container) to store one instance of each class. You will need to use an iterator to iterate over the vector (`http://www.cplusplus.com/reference/iterator/`).

```
///========================== main.cpp ==========================
#include <iostream>
#include "Shape.h"
#include <vector> ///< we are going to use the STL std::vector
using namespace std;

int main(int, char**){
    /// @todo
    /// 1) fill the vector with one instance of each class
    /// 2) iterate over the vector and call the virtual member function
    /// 3) empty the vector (and delete memory!!!!)
    typedef std::vector<Shape*> ShapesVector;
    ShapesVector vec;
    /// @solution
    /// 1) fill the vector
    vec.push_back(new Shape(10));
    vec.push_back(new Square(10));
    vec.push_back(new Circle(10));

    /// 2) iterate over the vector and call the virtual member
    for(ShapesVector::iterator it=vec.begin(); it!=vec.end(); it++)
        (*it)->virtual_member_function();

    /// 3) empty the vector and delete memory
    for(ShapesVector::iterator it=vec.begin(); it!=vec.end(); it++)
        delete (*it); ///<frees memory
    vec.clear(); ///< @attention only now it's safe to empty the vector!

    return EXIT_SUCCESS;
}
```

If you do not know which container to use for the task at hand you can peek at this diagram http://i.stack.imgur.com/kQnCS.png.