581251 Software Design (C++) Course Exam 19 Dec 2014

The course includes an obligatory C++ programming project.

You can answer the following questions either in Finnish/Swedish or in English. If you are answering in Finnish/Swedish, give the original technical English terms for your own translations, when appropriate.

1. Explain (shortly) the following C++ related terms, concepts or idioms. Explain what the concept is and why is it important. [3p. each]
    a. Type safety
        i. every object is used only according to its type (defined operations)
        ii. each operation programmed to leave the object with a valid value
        iii. Other valid approved observations
            - explanations about static and dynamic type safety and/or good examples of safety violations and their consequences
            - C++ is *not* statically *nor* dynamically type safe, *nor* is it strongly typed
    b. Class invariant
        i. A rule for what constitutes a valid value. Checks the internal state of the object.
        ii. operations of the class should always leave the object with a valid value (variant is true both before and after the operation has been applied)
    c. Virtual destructor
        i. If a class is possibly used as a base class (derived from), its destructor must be defined as virtual
        ii. Ensures that polymorphic objects are correctly destroyed when deleted via a base class pointer, because the runtime system will use dynamic dispatch (late binding) to start destruction by calling the destructor of the most derived type of the object (the type/class used when the object was created)
    d. const member function
        i. a member function of a type that does not change the state (data members) of an object of that type
        ii. helps compiler to check at compile time that code using the objects does not accidentally call state changing operations when that is not intended (e.g. accessing objects by const reference)

    [12 points]

2. Explain the following items concerning C++ related concepts, programming techniques or language constructs
    a. How is the C-style *assert* facility different from *exceptions* in C++? Which one would you use in checking a *pre-condition* of a function and which one would you use in checking a *post-condtion*? Why? [4 p.]
        i. assert aborts the executing program immediately
        ii. throwing exceptions gives the program a chance to catch the exception and try to recover from the error situation and continue execution

iii. Exceptions are good for pre-condition checks, because pre-condition check fails are typically caused by the caller trying to use the function inappropriately. the caller should get a chance to try again (external errors)
iv. Asserts are better for post-condition checks and class invariant checks, that indicate internal faults (bugs) in the executing component/function. There is probably no meaningful way to continue.

b. Copy constructor and move constructor. Give an example of both in C++. [6p.]
   i. Copy constructor is used to create and initialize an object by copying the state (data members) of another object of the same type given as a parameter (by const reference). It is up to the programmer to define the depth of copying the data members. The default copy constructor generated by the compiler performs a shallow copy (bitwise copy of data members).
   ii. Move constructor is used to create and initialize an object by taking over the (dynamic) resources reserved and owned by another object of the same type given as a parameter (by rvalue reference) and then clearing the related data members of the other object. The idea is to take possession of ("steal") the resources acquired by the other object and leave it in a state where it can be safely destroyed.
   iii. Example code: see for example the code for Vector in Slideset 2 (slides 40 & 46)

c. STL iterator concept: what is it and how the algorithms in STL depend on it? [4p.]
   i. Iterator is a type that provides pointer like access to the items of a collection or container data type. Key ops *p, p++, p==q and p!=q
   ii. Iterator can be used to traverse the items using a simple post-increment operator (++)
   iii. The container must provide operation begin() to return iterator to the first item and end() to a special iterator of one past the last item.
   iv. The algorithms access the containers via iterators. That's how the same algorithms (find, sort etc.) can be applied to all container types that provide a compatible iterator (there is no dependende to any particular container types).

[14 p.]

3. The following function `make_X()` may leak memory if either `X::configure()` or `X::start()` throws an exception. Explain why memory may be leaked. Is there any risk of other leaks that might occur (hint: think about the other resources possibly acquired by X)?

   Let's assume that X is from a legacy code library and we cannot change the implementation of X. Write two versions of `make_X()` that show two different ways to avoid the memory leak problem. Note that objects of class X must always be handled via pointers (the objects cannot be copied) but you are allowed to change the return type of `make_X()`. Make sure that your versions are exception neutral; the exceptions thrown by X are derived from std::exception. Which version is better in your opinion? [14 p.]

```
X* make_X(const vector<bool>& config_bits)
```

```
{
     X* p_x = new X();
     p_x->configure(config_bits);
     p_x->start();
     return p_x;
}
```

If either operation throws, the function make_X is exited immediately, nothing is returned to the caller (exception handling is in process), and there is no way anymore (no pointer) to access the X object created by new. So, the memory for the object stays reserved but inaccessible, which means it is leaked (cannot be reused). Because the object is never deleted, all dynamic resources reserved by X in its constructor (dynamic objects created by X, file handles, database connections etc.) are also never released (not until the whole process running the program exits). This is a possibility to consider, although the details how X manages its resources are not given.

Two ways to solve the problem:

```
// (1)
X* make_X2(const vector<bool>& config_bits)
{
     X* p_x = new X();
     try {
               p_x->configure(config_bits);
               p_x->start();
     } catch (std::exception& e) {
               delete p_x; // delete (and release resources)
               throw;      // re-throw the same exception -
     }                     // required for exception neutrality
     return p_x;           // if no exceptions
}
//(2a) – wrap the raw pointer in a shared pointer
// This is also exception neutral. When propagating the
```

```
// exception, all local variables are properly destructed
// and the X object wrapped in a shared_ptr is deleted.
std::shared_ptr<X> make_X3(const vector<bool>& config_bits)
{
    std::shared_ptr<X> p_x(new X());
    p_x->configure(config_bits);
    p_x->start();
    return p_x;
}
// or (2b) – wrap the raw ptr inside a unique pointer
// Note - don’t _have_ to change the return type, but it is a
// good idea to keep raw pointers inside smart pointers!
X* make_X4(const vector<bool>& config_bits)
{
    std::unique_ptr<X> p_x(new X());
    p_x->configure(config_bits);
    p_x->start();
    return p_x.release();
}
// Returning a unique pointer containing the raw pointer is
// also fine (due to move constructor semantics), but shared
// pointers are more natural for holding raw pointers across
// different scopes
```

(Note that a 3rd way could be to plug-in a garbage collector to the program.)

Using the smart pointers is more elegant and less error prone. When using the try-catch , you still have to *remember to (explicitly) delete* p_x.