TDDD38 Advanced Programming in C++

Aim – or, what does “advanced” stand for?

- good knowledge about constructs and mechanisms in the programming language C++ and their use
- good knowledge about the C++ standard library, both content and design principles, and how to use its components effectively
- ability to design and implement usable, correct, error-safe, non-trivial classes, including polymorphic class lattices (hierarchies)
- ability to design and implement advanced program components – template based, policy design, function objects, …
- it’s ***not*** a systems design course, or a problem solving course, or alike, but there will be problems to solve!

Prerequisite

- good knowledge and skills in programming in at least one procedural and/or object-oriented language
- knowledge about fundamentals of object-oriented programming (class, derivation/inheritance, polymorphism)

No previous experience of C++, C or Java?

- get acquainted a.s.a.p. with C++ basics
- attend lecture 2 and 3

Organization

- basically a self-study course – good self-discipline is necessary
  - study theory!
  - do exercises!
  - do it in time!
- lectures – the core language is covered in the first study period, standard library in the second study period
  - single class design
  - operator overloading
  - derived classes, inheritance, polymorphism, RTTI,…
  - exception handling, exception classes
  - templates
  - standard library – strings, streams, standard exceptions, containers, iterators, algorithms, function objects, related utilities,…
  - design patterns, policy design, template meta programming, style, …
- no lessons – no compulsory labs
  - optional exercises – strongly recommended!
  - it is absolutely necessary to practice a lot – and enough on a Unix system (IDA's Linux Mint system) – and using g++
- limited computer resources scheduled
  - assistance by email to TDDD38@ida.liu.se (or tommy.olsson@liu.se), or by appointment
  - you are also welcome to look me up in my office

Examination

Computer exam – the only compulsory item

- four times a year – December, March/April, May/June, and August – five hours
- five theory questions – 1 points each
- four programming problems – 5 points each
  - basic stuff is of course required
  - class design, derivation, operator overloading, templates, exception handling
  - standard library – strings, stream I/O, containers, iterator, algorithms, function objects, related utilities, etc.
  - techniques and style
- marking – 25 points maximum
  - 19-25 – 5/A
  - 15-18 – 4/B
  - 11-14 – 3/C (corresponds to correctly solving two programming problems and getting one theory question right).
  - 0-10 – U/FX
- means of assistance
  - cplusplus.com Reference (available in a Chromium web browser)
- important to be familiar with the exam system (Unix), some available text editor (Emacs), and the GCC compiler (g++)
  - only simple file handling required
  - sufficient experience in how to read, interpret and act upon compiler and linker error and warning messages

Information, literature, etc.

- course home page: http://www.ida.liu.se/~TDDD38/
  - information about examination – the three last given exams are always available
  - timetable (link to LiTH timetable server)
  - lecture plan and content
  - lecture slides, code examples
  - exercises
  - C++ links
  - contact information
- course literature – basically your choice – some recommendations:

  C++ Primer, Fifth edition (2012). Lippman, Lajoie, Moo. (downloadable)

  The C++ Programming Language, Fourth edition (2013), Stroustrup.

  The C++ Standard Library, A Tutorial and Reference, 2/E (2012), Josuttis, N. M. (downloadable)

  cplusplus.com (tutorial, reference, articles, etc.) – Reference part is means of assistance at exam
- Friday fun!

Lecture plan for the forthcoming lectures

Lecture 2-3

- basic stuff – data types, variables and constants, declarations, expressions and operators, statements, functions,…
- strings, initializer lists, tuples, streams, string streams

Lecture 4–5

- single class design and operator overloading

Lecture 6–7

- derived classes, inheritance, polymorphism, RTTI
- exception handling

Lecture 8-9

- templates
- namespaces
- preprocessor

Lecture 10-12 (13)

- standard library
  - containers – iterators – algorithms – function objects, lambda expression
  - related utilities and such (e.g. std::pair)
  - maybe some more…

And now some comments to the course content…

C++ history and future

- one of the most popular programming languages
- development started in 1979 – originally named *C with Classes*
- renamed *C++* in 1983
- *C++98* – the first ISO standard
- *C++03* – C++98 was amended by the 2003 technical corrigendum (TC)
- *C++11* – current standard (formerly known as C++0x, since it was expected to be released before 2010)
  - a “new” C++
  - a new C++ programming style is developing
- *C++14* – minor revision targeted for late 2014, but delayed
  - mainly bug fixes and small improvements
  - one new major feature can be expected – maybe **static if** (compile time **if** statement)
- *C++1y* – major revision targeted for 2017
  - will not be the last…
- Each revision
  - adds more power
  - makes constructs more general
  - increases efficiency even more
  - makes C++ easier to use

Object-oriented programming

Three main components – objects, inheritance, polymorphism

- objects
  - classes are used to model objects
  - in C++ we have two syntactic choices, **class** or **struct**
  - only difference is related to default member and base member access – **private** for **class** – **public** for **struct**
  - *rule of thumb:* use **struct** for behaviourless aggregates with public members only, **class** otherwise
- inheritance
  - code can be reused by derivation, *base class – subclass*
  - creates related classes/objects – a subclass object is also a base class object regarding type, an “is a” relationship
  - derivation is typically a question of specialization – a subclass can have more state and more functionality
  - C++ supports four ways to derive from a base class – **public** base, **protected** base, **private** base, and **virtual** base
- polymorphic behaviour
  - an object reference (pointer or reference) may at different times refer to objects of different type
  - the same member function call may at different times give different result, depending on the type of the related object
- polymorphic behaviour is optional in C++
  - a member functions must be declared **virtual** to be able to behave polymorphic
  - objects must be referred to by pointers or references
- dynamic type checking and dynamic type conversion, RTTI
  - sometimes it’s required

Important characteristics for class types in C++

```
{
    string s1;        // an object declaration, in some block, string is a class type
}
```

- classes in C++ are *not* reference types
  - `s1` stores an object of type `string` – it's not a reference to such an object
  - a `string` object is automatically created and initialized when the declaration of `s1` is elaborated
  - when the execution exits the declaration block, `s1` goes out of scope – the object is destroyed – memory is reclaimed
- class types have the same basic semantics as fundamental types, e.g.
  - require *copy semantics* not found in most other object-oriented languages – in C++11 *move semantics* is also available

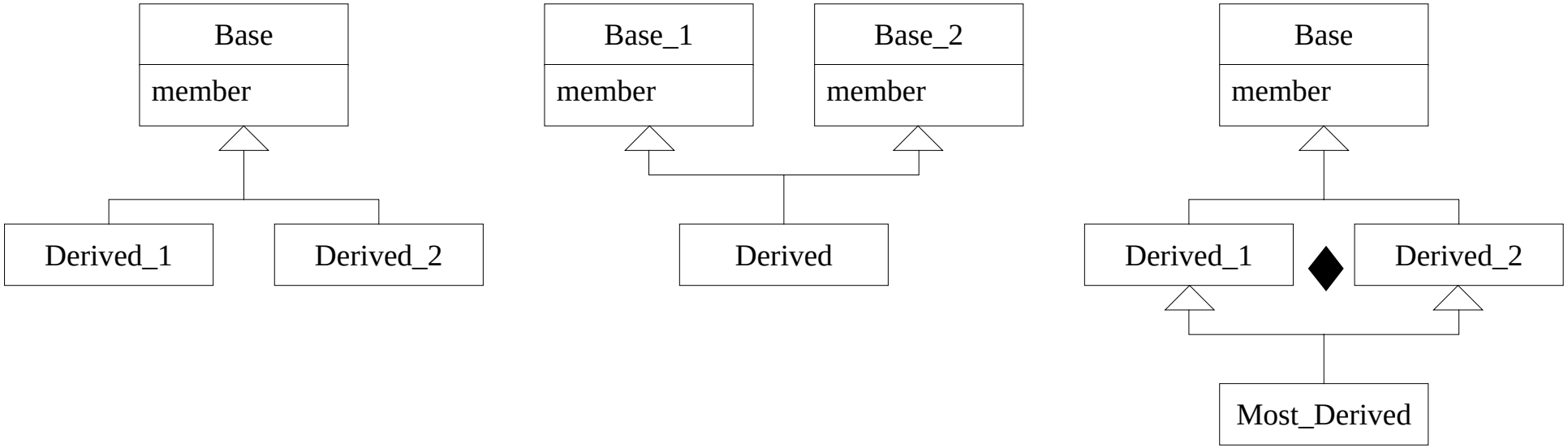
```
string s2{s1};      // initialization          string s2(s1);          string s2 = s1;
s2 = s1;            // assignment
```

- great care must be taken when designing classes in C++
  - initialization – *default constructor*, argument passing constructors
  - copying – *copy constructor*, *copy assignment operator*, *move constructor*, *move assignment operator*
  - destruction – *destructor*
- to mimic e.g. Java we use *pointers* an dynamic memory allocation (memory has to be deallocated explicitly when no longer needed)

```
string* message{ new string{ "Hello world!"} };
…
delete message;
```

Inheritance and classes derivation

- class derivation is an essential features of object-oriented design
  - new classes can be defined from exciting classes
  - code is reused – members are inherited
- C++ supports “all” variants
  - *single* inheritance (single base class)
  - *multiple* inheritance (multiple base classes), which can lead to
  - *repeated* inheritance (DDD – the *Deadly Diamond of Derivation*, ♦), which is supported by
  - *virtual* inheritance (virtual base class) – how many Base subobjects are there to be in Most_Derived?



Operator overloading

```
ostream& operator<<(ostream& os, const T& x)
{
    // write x to os
    return os;
}
```

- important for construction of fully featured data types
  - assignment
  - indexing
  - any other operator for which there is a natural interpretation of its use
- function objects rely on the possibility to overload the function call operator – **operator**()
  - function objects are important components in the standard library – lambda expressions are implemented as function objects
  - can act as function
  - can carry state
  - possible to overload for class types and for compound types (**enum** types, pointer types, etc.)

```
struct fun
{
    void operator()() const { cout << "This is fun!\n"; }
};

fun()();          // not the same parentheses…          fun{}()
```

Templates

Extremely powerful construct in C++.

- for creating reusable program components – function templates and class templates

```
template <typename T> T fun(const T& a);

template <typename T, size_t N> class array;
```

- supports e.g. policy design
  - a policy used by a type (class) can be separated into one or more policy classes, which can be supplied as a template parameter

```
template <typename T, class Allocator = allocator<T>> class vector;
```

- supports template metaprogramming
  - function templates can be used to let the compiler generate source code
  - recursive, purely static template functions can perform compile-time evaluation
- is an object-oriented construct – static (compile-time) polymorphism
  - a function template represents a whole family of functions
  - a class template represents a whole family of classes, data types are pure static (compile-time) constructs
- the standard library depend heavily on templates, in many cases in combination with derivation
  - the new feature *variadic templates* is widely used in the implementation of the standard library

```
template <typename... Types> class tuple;
```

Exception handling

- conceptually fairly simple, but should be used with care.
  - prefer traditional, local error handling if possible
  - good for reporting errors from within software components
  - place error handlers with care – avoid “exception handling spaghetti”
  - practice exception-safe (error-safe) programming
- C++ implements the *termination model* – the alternative to the *resumption model*
  - at least one block is terminated
  - exceptions are propagated backwards along the dynamic call chain
  - stack objects will be destroyed implicitly and properly when blocks are exited
  - heap objects must be taken responsibility for by the programmer (smart pointers is one possibility)

```
try
{
    ...
    if (disaster) throw some_exception{"help!"};    // supposed subclass to std::exception
    ...
}
catch (const exception& e)                          // “polymorphic catch”
{
    cout << e.what() << endl;
}
```

Namespaces

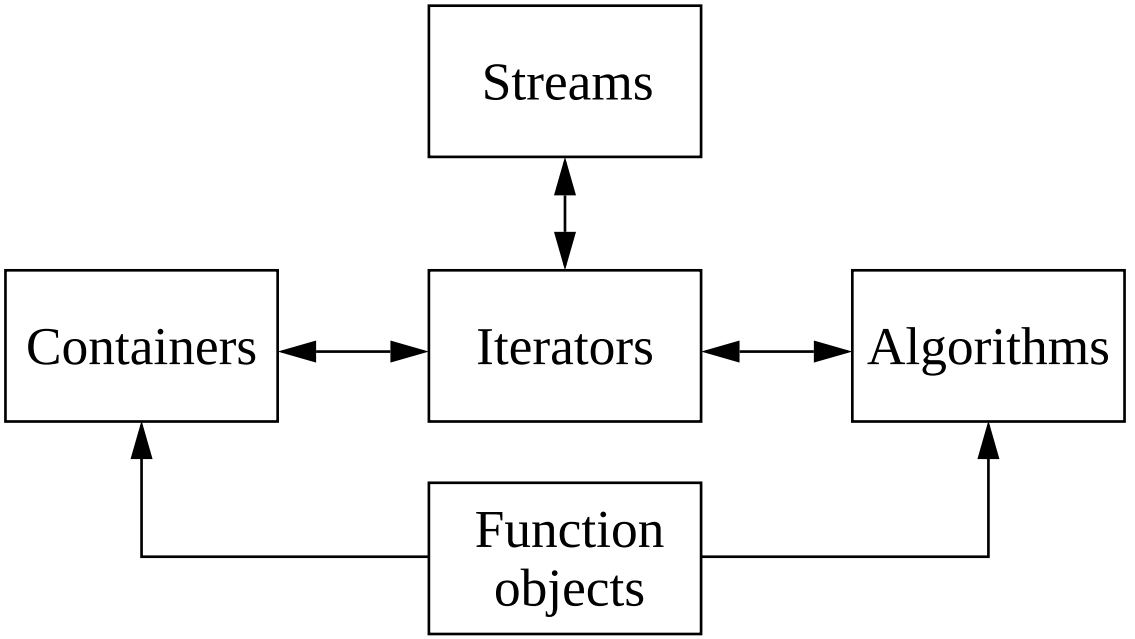
A simple module construct.

- the class has module properties, but it’s not enough
  - was one of the last things introduced before the first standard C++98 was published
- important in several aspects
  - for handling potential name collisions
  - modularisation – a type and it’s operations, e.g., should be encapsulated in the same namespace
  - function name look up – ADL look up (argument-dependent look up) – is related to namespaces

```
namespace std
{
    // namespace member declarations…
}

using namespace std;        // using directive

using std::member;          // using declaration

... std::member ...         // qualified name
```

Standard Library

Data structure and algorithm part



- String
- Streams
- String streams
- Related utilities
- Interesting implementation, e.g.
  - templates
  - derivation
  - policy design

And, of course, a lot of basic stuff to be mastered

The “C part”.

- Lexical conventions
- Translation model – compilation and linking
- Data types and type conversion
- Declarations and definitions
- Expression and operators
- Statements
- Functions and parameter passing
- Basic standard library components
- I/O and file handling

Topics for lecture 2 and 3.