EDAF50 – C++ Programming

10. The project. Templates and the standard library.

Sven Gestegård Robertz
Computer Science, LTH

2024

Outline

1. The project
2. Templates
   - Variadic templates
   - Template metaprogramming
3. The standard library
   - Time representation
   - Algorithms

Project, News

- 2–4 people per group. Use slack to find project partners.
- Develop a news server (two versions) and a text-based client.
- Write a report, hand in the report and your programs no later than Monday, May 6

A News Server and News Clients

The server keeps a database of newsgroups, containing articles. The clients connect to the server. Sample conversation:

```
news> list
1. comp.lang.java
2. comp.lang.c++
news> list comp.lang.c++
1. What is C++? From: xxx
2. Why C++?     From: yyy
news> read 2
Why C++?   From: xxx
... text ...
news>
```

A client can also create and delete newsgroups, and create and delete articles in newsgroups.

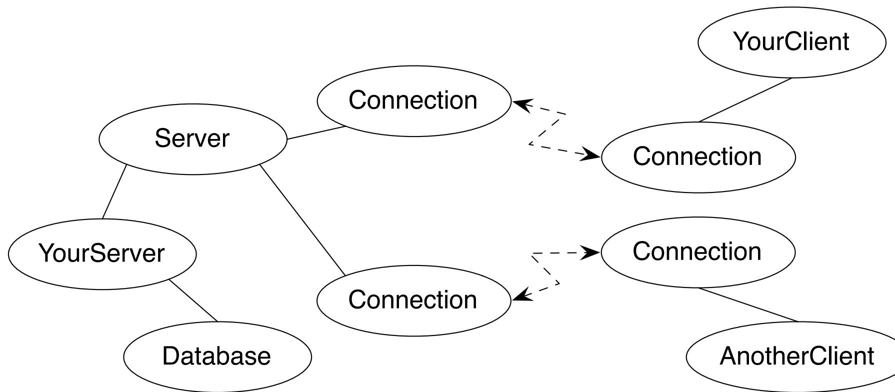
The Project: Write Server and Client

- You are to develop two versions of the server:
  - one in-memory server that forgets the data about newsgroups and articles between invocations (use the standard library containers for this database), and
  - one disk-based server that remembers the data between invocations (use files for this database)

  These versions should implement a common interface — the rest of the system should be independent of, and agnostic to, the database implementation. *Avoid duplicated code.*
- A single-threaded server is ok.
- You are to develop a client with a text-based interface. It shall read commands from the keyboard and present the replies from the server as text.
- Think about how to handle entry of multi-line articles.

System Overview

The classes `Server` and `Connection` are pre-written.



Communication Protocol

A message is a sequence of bytes. Messages must follow a specified protocol, which specifies the message format. The general form is:

```
MSG_TYPE_BYTE <data> END_BYTE
```

The protocol contains commands and answers:

```
COMMAND_TYPE <data> COM_END
ANSWER_TYPE <data> ANS_END
```

Communication Protocol
Example: List Newsgroups

List newsgroups (message to server and reply from server):

```
COM_LIST_NG COM_END
ANS_LIST_NG 2 13 comp.lang.java 15 comp.lang.c++ ANS_END
```

2 is the number of newsgroups, 13 and 15 are the unique identification numbers of the newsgroups `comp.lang.java` and `comp.lang.c++`.

Numbers and strings are coded according to the protocol:

```
string_p: PAR_STRING N char1 char2 ... charN // N is an int, sent as
num_p: PAR_NUM N                             // 4 bytes, big endian
```

Hint:

Factor out the functionality for communication on the "low protocol level" (encoding and decoding of numbers and strings).

Don't repeat yourselves.

Class Connection

```
struct ConnectionClosedException {};

/* A Connection object represents a socket */
class Connection {
      friend class Server;
public:
      Connection(const char* host, int port);

      Connection();

      virtual ~Connection();

      bool isConnected() const;

      void write(unsigned char ch) const;

      unsigned char read() const;
protected:
      void initConnection(int socket);

      //...
};
```

Class Server

```
/* A server listens to a port and handles multiple connections */
class Server {
public:
        explicit Server(int port);

        virtual ~Server();

        bool isReady() const;

        std::shared_ptr<Connection> waitForActivity() const;

        void registerConnection(const shared_ptr<Connection>& conn);

        void deregisterConnection(const shared_ptr<Connection>& conn);
};
```

Server Usage

```
while (true) {
        auto conn = server.waitForActivity();
        if (conn != nullptr) {
                try {
                        /*
                         * Communicate with a client, conn->read()
                         * and conn->write(c)
                         */
                } catch (ConnectionClosedException&) {
                        server.deregisterConnection(conn);
                        cout << "Client closed connection" << endl;
                }
        } else {
                conn = make_shared<Connection>();
                server.registerConnection(conn);
                cout << "New client connects" << endl;
        }
}
```

Provided material

On the course web page, you will find

- Classes for creating connections, including an example application.
- Test clients written in Java
  - An interactive, graphical client
  - An automated test client that runs a series of operations. Please note that this is an aid during development and not a complete acceptance test.

Report and submission

- Write the report, preferably in English, follow the instructions.
- Create a directory with your programs (only the source code – don't include any generated files) and a Makefile.
- Write a README file (text) with instructions on how to build and test your system.
- Submission:
  1. The report in PDF format.
  2. The README file.
  3. The program directory, as a tar, tar.gz or .zip archive.
     - *Make sure that executables or object files are not included.*
     - *Avoid swedish characters, spaces, and special characters (`+`,`*`,`?`, ...) in file and directory names.*
  4. Submission instructions will be published on the course web, under Project.

the <filesystem> header

- standardised interface to the filesystem
- introduced in C++-17

```
path                                current_path
directory_entry                     absolute
directory_iterator                  relative
recursive_directory_iterator        exists
file_status                         status
file_type                           permissions
perms                               copy
                                    remove
                                    rename
```

Variadic templates

A function template can take a variable number of arguments

```
void println() { base case: no argument
    cout << endl;
}

template <typename T, typename... Tail>
void println(const T& head, const Tail&... tail)
{
    cout << head << " ";     Print the first element
    println(tail...);        recursion: print the rest
}

void test_variadic()
{
    string a{"Hello"};
    int b{10};
    double c{17.42};
    long d{100};

    println(a,b,c,d);

}
```

Template metaprogramming

- Write code that is executed *by the compiler, at compile-time*
- Common in the standard library
  - As optimization: move computations from run-time to compile-time
  - As utilities: e.g., `type_traits`, `iterator_traits`
- Metafunction: a class template containing the result
- Standard library conventions:
  - Type results: type member named `type`
  - Value results: value member named `value`

Template metaprogramming
Example of compile-time computation

```
template <int N>
struct Factorial{
    static constexpr int value = N * Factorial<N-1>::value;
};

template <>
struct Factorial<0>{
    static constexpr int value = 1;
};

void example()
{
    Show<int, Factorial<5>::value>{};
}
```

Result of the *meta-function call* as a compiler error:

```
error: invalid use of incomplete type 'struct Show<int, 120>'
     Show< int, Factorial<5>::value >{};
```

Template metaprogramming
Example of templates for getting values as compiler errors

- Trick: use a template that doesn't compile to get information about the template parameters through a compiler error.
- Can be useful for debugging templates.
- To get the type parameter `T`:

```
template <typename T>
struct ShowType;
```

- To get a value (`N`) of type `T`:

```
template <typename T, T N>
struct Show;
```

What is a value

The semantics of a value often include

- a quantity
- a number
- a unit

E.g `int length = 2;`

- two meters?
- two millimeters?

*Including quantity and unit in the **type** helps avoid mistakes.*

Time representation

- A "time value" can be either
  - A duration – a time interval
  - A point in time
    - relative to a particular *clock*
- Different units
  - seconds
  - milliseconds
  - nanoseconds
  - *manual conversion error prone*
- Different semantics
  - `duration + duration = duration`
  - `duration - duration = duration`
  - `time_point + duration = time_point`
  - `time_point - duration = time_point`
  - `time_point - time_point = duration`
  - `time_point + time_point =` *error*

Time representation

<chrono>

- Uses the type system to denote
  - if a value is a duration or a point in time
  - the unit used (seconds, milliseconds, etc.)
  - which clock a point in time is relative to
    - `system_clock` – wall clock time
    - `steady_clock` – stopwatch
- Uses compile-time computations for
  - conversions between units
    - implicit conversions when safe
    - explicit conversions when loosing information
    - E.g. `duration_cast<seconds>(milliseconds)`

Time representation
<chrono>

A duration is

- an *integer value* and
- *a ratio* (the number of seconds between two values).

```
std::chrono::nanoseconds duration</*signed int, at least 64 bits*/,
                                          std::nano>
std::chrono::microseconds duration</*signed int, at least 55 bits*/,
                                          std::micro>
std::chrono::milliseconds duration</*signed int, at least 45 bits*/,
                                           std::milli>
std::chrono::seconds duration</*signed integer, at least 35 bits*/>
std::chrono::minutes duration</*signed integer, at least 29 bits*/,
                                    std::ratio<60>>
std::chrono::hours   duration</*signed integer, at least 23 bits*/,
                                    std::ratio<3600>>
```

`std::ratio` provides compile-time rational arithmetic

Demo

Algorithms and references

The standard algorithms take function objects *by value*:

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f);

template< class InputIt, class UnaryPredicate >
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p);
```

How to handle *stateful function objects*?

Demo

std::ref

<functional> defines helper functions std::ref and std::cref:

```
template< class T >
std::reference_wrapper<T> ref(T& t) noexcept;

template< class T >
std::reference_wrapper<const T> cref( const T& t ) noexcept;
```

that return a CopyConstructible and CopyAssignable wrapper around a reference:

```
template< class T >
class reference_wrapper {
public:
    reference_wrapper& operator=(const reference_wrapper&) noexcept;
    operator T&() const noexcept;
    T& get() const noexcept;

    template< class... ArgTypes >
    typename std::result_of<T&(ArgTypes&&...)>::type
    operator() ( ArgTypes&&... args ) const;
};
```

Suggested reading

References to sections in Lippman

Overloading and templates 16.4

Variadic templates 16.4

Template specialization 16.5