EDAF50 – C++ Programming

12. Recap. About the project.

Sven Gestegård Robertz
Computer Science, LTH

2021
Outline

1. The project

2. Classes and inheritance
   - Scope
   - const for objects and members

3. Rules of thumb

4. Advice
2–4 people per group. Use the Canvas discussion 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 Tuesday, April 21.
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.
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.
The classes Server and Connection are pre-written.
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 of commands and answers:

```
COMMAND_TYPE <data> COM_END
ANSWER_TYPE <data> ANS_END
```
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: write a class to handle the communication on “low protocol level” (encoding and decoding of numbers and strings).

Don’t repeat yourselves.
struct ConnectionClosedException {};  
/* A Connection object represents a socket */
class Connection {
public:
    Connection(const char* host, int port);
    Connection();
    virtual ~Connection();
    bool isConnected() const;
    void write(unsigned char ch) const;
    unsigned char read() const;
};
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);
};
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;
    }
}
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.
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, tar-ed and gzip-ped. Don’t bury the report inside the gzip file.
4. Submission instructions will be published on the course web, under Project.
The scope of a derived class is nested inside the base class
- Names in the base class are visible in derived classes
- if not hidden by the same name in the derived class
- Use the scope operator :: to access hidden names
- Name lookup happens at compile-time
  - static type of a pointer or reference determines which names are visible (like in Java)
  - Virtual functions must have the same parameter types in derived classes.
No function overloading between levels in a class hierarchy

```cpp
struct Base{
    virtual void f(int x) { cout << "Base::f(int): " << x << endl; }
};
struct Derived : Base{
    void f(double d) { cout << "Derived::f(double): " << d << endl; }
};

void example() {
    Base b;
    b.f(2);    Base::f(int): 2
    b.f(2.5);  Base::f(int): 2 (as expected)
    Derived d;
    d.f(2);    Derived::f(double): 2
    d.f(2.5);  Derived::f(double): 2.5

    Base& dr = d;
    dr.f(2.5); Base::f(int): 2
    dr.f(2);   Base::f(int): 2
}
```
Function overloading and inheritance

Make functions visible using `using`

```cpp
struct Base{
    virtual void f(int x) { cout << "Base::f(int): " << x << endl; }
};
struct Derived : Base{
    using Base::f;
    void f(double d) { cout << "Derived::f(double): " << d << endl; }
};

void example() {
    Base b;
    b.f(2); // Base::f(int): 2
    b.f(2.5); // Base::f(int): 2

    Derived d;
    d.f(2); // Base::f(int): 2
    d.f(2.5); // Derived::f(double): 2.5
}
```
class Vector {
public:
    Vector() =default;
    Vector(int s): size{s}, elem{new T[size]} {}
    T* begin() {return elem.get();}
    T* end() {return begin()+size;}
    // functionality for growing...
private:
    std::unique_ptr<T[]> elem{nullptr};
    int size{0};
};

*Error! size is uninitialized when used to create the array.*

- If a member has both *default initializer* and a member initializer in the constructor, the constructor is used.
- Vector() =default; is necessary to make the compiler generate a default constructor.
- Members are initialized *in declaration order.* (Compiler warning if member initializers are in different order.)
Constructors
Special cases: zero or one parameter

class KomplextTal {
public:
    KomplextTal(): re{0}, im{0} {}
    KomplextTal(const KomplextTal& k): re{k.re}, im{k.im} {}
    KomplextTal(double x): re{x}, im{0} {}
    // ...
private:
    double re;
    double im;
};
Constructors
Implicit conversion

```cpp
struct Foo {
    Foo(int i) : x{i} { cout << "Foo(" << i << ")\n"; }
    Foo(const Foo & f) : x(f.x) { cout << "Copying Foo(" << f.x << ")\n"; }
    Foo & operator=(const Foo & f) {
        x = f.x;
        return *this;
    }
    int x;
};

void example()
{
    int i = 10;
    Foo f = i;    // Foo(10) (an optimized away copy(move) construction)
    f = 20;      // Foo(20)
    Foo g = f;   // Copying Foo(20)
}
```
Constructors
Default constructor

Default constructor
A constructor that can be called without arguments
- May have parameters with default values
- Automatically defined if *no constructor is defined*
  (in declaration: =default, cannot be called if =delete)
- If not defined, the type is *not default constructible*
Constructors
Copy constructor

- Is called when initializing an object
- Is *not called* on assignment
- Can be defined, otherwise a standard copy constructor is generated (default, delete)
- default copy constructor
  - Is automatically generated if not defined in the code
  - exception: if there are members that cannot be copied
  - shallow copy of each member
The parameter \( v \) is default copy constructed: the value of each member variable is copied.

- When \( f() \) returns, the destructor of \( v \) is executed:
  ```cpp
  (delete[] elem;)
  ```
- The array pointed to \textit{by both copies} is deleted. Disaster!
If a class implements any of these:

1. Destructor
2. Copy constructor
3. Copy assignment operator

it (quite probably) should implement (or `delete`) all three.

*If one of the automatically generated does not fit, the other ones probably won’t either.*
“Rule of three five”
Canonical construction idiom, from C++11

If a class implements any of these:

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. Move constructor
5. Move assignment operator

it (quite probably) should implement (or \texttt{=delete}) \textit{all five}.

\textit{and possibly an overloaded swap function.}
Constant objects

- `const` means “I promise not to change this”

- Objects (variables) can be declared `const`
  - “I promise not to change the variable”

- References can be declared `const`
  - “I promise not to change the referenced object”
  - a `const&` can refer to a non-`const` object
  - common for function parameters

- Member functions can be declared `const`
  - “I promise that the function does not change the object”
  - A `const` member function *may not call non-`const` member functions*
  - Functions can be overloaded on `const`
Operator overloading syntax:

```
return_type operator⊗ (parameters...)
```

for an operator ⊗ e.g. == or +

For classes, two possibilities:

- as a member function
  - if the order of operands is suitable
    E.g., ostream& operator<<(ostream&, const T&)
    cannot be a member of T

- as a free function
  - if the public interface is enough, or
  - if the function is declared friend
Conversion operators
Exempel: Counter

Conversion to int

```cpp
struct Counter {
    Counter(int c=0) : cnt{c} {};
    Counter& inc() {++cnt; return *this;}
    Counter inc() const {return Counter(cnt+1);}
    int get() const {return cnt;}
    operator int() const {return cnt;}

private:
    int cnt{0};
};
```

Note: `operator T()`.

- no return type in declaration (must obviously be T)
- can be declared `explicit`
Constructors
Member initialization rules

class Bar {
public:
    Bar() = default;
    Bar(int v, bool b) : value{v}, flag{b} {}  
private:
    int value {0};
    bool flag {true};
};

▶ If a member has both default initializer and a member initializer in the constructor, the constructor is used.

▶ Members are initialized in declaration order. (Compiler warning if member initializers are in different order.)

▶ Bar() =default; is necessary to make the compiler generate a default constructor (as another constructor is defined).
Constructors
Special cases: zero or one parameter

```cpp
class KomplextTal {
public:
    KomplextTal() : re{0}, im{0} {}
    KomplextTal(const KomplextTal& k) : re{k.re}, im{k.im} {}
    KomplextTal(double x) : re{x}, im{0} {}
    //...

private:
    double re;
    double im;
};
```
default constructor  copy constructor  converting constructor
struct Foo{
    Foo(int i) : x{i} { cout << "Foo(" << i << ")\n"; }
    Foo(const Foo& f) : x(f.x) { cout << "Copying Foo(" << f.x << ")\n"; }
    Foo& operator=(const Foo& f) { cout << "Foo = Foo(" << f.x << ")\n";
        x=f.x;
        return *this;
    }
    int x;
};

void example()
{
    int i=10;

    Foo f = i;       Foo(10) (conversion + optimized away copy/move)
    f = 20;          Foo(20)
        Foo = Foo(20) (would move if operator=(Foo&&) defined)
    Foo g = f;       Copying Foo(20)
Conversion operators
Exempel: Counter

**Conversion to int**

```cpp
struct Counter {
    Counter(int c=0) : cnt{c} {}
    Counter& inc() {++cnt; return *this;}
    Counter inc() const {return Counter(cnt+1);}
    int get() const {return cnt;}
    operator int() const {return cnt;}
private:
    int cnt{0};
};
```

**Note:** `operator T()`.
- no return type in declaration (must obviously be `T`)
- can be declared `explicit`
rules of thumb, “defaults”

- Iteration, *range for*
- *return value optimization*
- call by value or reference?
- reference or pointer parameters? (without transfer of ownership)
- default constructor and initialization
- resource management: RAII and *rule of three (five)*
- be careful with type casts. Use *named casts*
use range for

```cpp
for(auto e : collection) { or (const) reference
   // ...
}
```

Use range for for iteration over *an entire* collection:

- safer and more obvious code
- no risk of accidentally assigning
  - the iterator
  - the loop variable
- no pointer arithmetic

Works on any type T that has

- member functions T::begin() and T::end(), or
- free functions begin(T) and end(T)
- with proper `const` overloads
return value optimization (RVO)

The compiler may optimize away copies of an object when returning a value from a function.

- *return by value* often efficient, also for larger objects
- RVO allowed *even if the copy constructor or the destructor has side effects*
- avoid such side effects to make code portable
Rules of thumb for function parameters

parameters and return values, “reasonable defaults”

- return by value if not very expensive to copy
- pass by reference if not very cheap to copy (Don’t force the compiler to make copies.)
  - input parameters: const T&
  - in/out or output parameters: T&
parameters: reference or pointer?

- required parameter: pass reference
- optional parameter: pass pointer (can be nullptr)

```c
void f(widget & w)
{
    use(w); // required parameter
}

void g(widget * w)
{
    if(w) use(w); // optional parameter
}
```
Default constructor and initialization

- (automatically generated) default constructor (=`default`) does not always initialize members
  - *global variables* are initialized to 0 (or corresponding)
  - *local variables* are not initialized

```c
struct A { int x; };
```

```c
int a; // a is initialized to 0
A b;   // b.x is initialized to 0
```

```c
int main() {
    int c; // c is not initialized
    int d = int(); // d is initialized to 0
    A e;       // e.x is not initialized
    A f = A(); // f.x is initialized to 0
    A g{};     // g.x is initialized to 0
}
```

- *always initialize variables* (with value or empty `{}`)
- *always implement default constructor* (or =`delete`)
RAII: Resource acquisition is initialization

- Allocate resources for an object in the constructor
- Release resources in the destructor
- Simpler resource management, no naked `new` and `delete`
- Exception safety: destructors are run when an object goes out of scope
- **Resource-handle**
  - The object itself is small
  - Pointer to larger data on the heap
  - Example, our Vector class: pointer + size
  - Utilize move semantics
- `unique_ptr` is a `handle` to a specific object. Use *if you need an owning pointer*, e.g., for polymorph types.
- Prefer specific `resource handles` to smart pointers.
Smart pointers: unique_ptr

Example

```cpp
struct Foo {
    int i;
    Foo(int ii=0) : i{ii} { std::cout << "Foo(" << i << ")\n"; }
    ~Foo() { std::cout << "~Foo(" << i << ")\n"; }
};

void test_move_unique_ptr()
{
    std::unique_ptr<Foo> p1(new Foo(1));
    {
        std::unique_ptr<Foo> p2(new Foo(2));
        std::unique_ptr<Foo> p3(new Foo(3));
        // p1 = p2; // error! cannot copy unique_ptr
        std::cout << "Assigning pointer\n";
        p1 = std::move(p2);
        std::cout << "Leaving inner block...\n";
    }
    std::cout << "Leaving program...\n";
}
```

Foo(2) survives the inner block as p1 takes over ownership.

---

Rules of thumb

12. Recap. About the project.

Foo(1)
Foo(2)
Foo(3)
Assigning pointer
~Foo(1)
Leaving inner block...
~Foo(3)
Leaving program...
~Foo(2)
Advice

Resource management

- Resource management: RAII and rule of three (five)
- Avoid “naked” new and delete
- Use constructors to establish invariants
  - throw exception on failure

for polymorph classes

- Copying often leads to disaster.
- delete
  - Copy/Move-constructor
  - Copy/Move-assignment
- If copying is needed, implement a virtual clone() function
classes
- only create member functions for things that require access to the representation
- as default, make constructors with one parameter explicit
- only make functions virtual if you want polymorphism

polymorph classes
- access through reference or pointer
- A class with virtual functions must have a virtual destructor.
- use override for readability and to get help from the compiler in finding mistakes
- use dynamic_cast to navigate a class hierarchy
Advice

.safer code
► initialize all variables
► use exceptions instead of returning error codes
► use *named casts* (if you must cast)
► only use `union` as an implementation technique inside a class
► avoid pointer arithmetics, except
  ► for trivial array traversal (e.g., `++p`)
  ► for getting iterators into built-in arrays (e.g., `a+4`)
  ► in very specialized code (e.g., memory management)

.use compiler warnings (consult your compiler manual)
- `Wall` - `Wextra` - `Werror` - `pedantic` - `pedantic-errors`
- `Wold` - `style-cast` - `Wnon-virtual-dtor` - `Wconversion` - `Wshadow`
- `Wtype-limits` - `Wtautological-compare` - `Wduplicated-cond`

*The compiler manual gives a comprehensive list of dangerous constructs.*
The standard library

- use the standard library when possible
  - standard containers
  - standard algorithms
- prefer std::string to C-style strings (char[])
- prefer containers (e.g., std::vector<T>) to built-in arrays (T[])
- consider standard algorithms instead of hand-written loops

Often both
- safer and
- more efficient
than custom code
The standard containers

- use `std::vector` by default
- use `std::forward_list` for sequences that are usually empty
- be careful with iterator invalidation
- use `at()` instead of `[]` to get bounds checking
- use `range for` for simple traversal
- initialization: use `()` for sizes and `{}` for list of values
- use `emplace_back` instead of `push_back` of a temporary
- use member functions (not algorithms) for `std::map` and `std::set`
Write code that is correct and easily understandable

Good luck on the exam

Questions?