Geheime berichten in audio files
Herkansingsopdracht CPPLS2, voorjaar 2021

Bob Polis, Avans Hogeschool, 's-Hertogenbosch

11 maart 2021

Inhoudsopgave

1 Geheime berichten

Medewerkers van geheime diensten en andere spionnen zoeken altijd naar manieren om elkaar geheime berichten te sturen. Een manier om dat te doen is door de berichten te verbergen in onschuldig uitziende data, zoals bijvoorbeeld foto's, muziekbestanden of video's. In het security vakgebied zegt men dat er dan gebruik wordt gemaakt van *covert channels*.

1.1 Hoe het werkt

Beeldinformatie in een fotobestand of audiodata in een muziekbestand bestaan in feite uit getallen. In een fotobestand gaat het om kleurwaarden van beeldpunten; in een muziekbestand staan audiosamples.

Deze getallen zijn voldoende nauwkeurig om de gewenste beeld- of geluidskwaliteit te halen. Wanneer we nu bijvoorbeeld van elk getal één onbeduidend bit gebruiken voor ons geheime bericht, kunnen we zo onze verborgen informatie verspreiden over een heleboel data in het bestand, zonder dat de beeld- of geluidskwaliteit merkbaar omlaag gaat.

Op deze manier verstopte informatie is eigenlijk niet te detecteren.

2 Wat ga je maken?

2.1 Globale omschrijving

Je gaat een programma maken dat minstens één specifiek audio bestandsformaat kan lezen en schrijven. Er zullen geldige bestanden worden aangeleverd waarin al berichten verborgen zitten.

Bij het lezen zul je specifieke bitjes in de data gaan verzamelen om daarmee het verborgen bericht te reconstrueren.

Ook zul je een gegeven bericht in zo'n bestand moeten kunnen verstoppen, op een voorgeschreven manier. Je hebt vast en zeker een programma op je systeem staan om muziek af te spelen. Daarmee kun je snel testen of de bestanden die je bouwt zich in elk geval als normale multimediabestanden gedragen.

2.2 Requirements

1. Je maakt een programma zonder grafische user interface;
2. Dat programma is geheel geschreven in C++;
3. Je programma kan argumenten vanaf de command-line krijgen waar nodig; in elk geval de input file(s);
4. Je programma schrijft uitvoer naar de *standard output*;
5. Je programma schrijft eventuele foutmeldingen, voortgang of statusberichten naar de *standard error* uitvoer;
6. Je programma moet in staat zijn data te kunnen lezen uit Wave audio files (of dat nu 'schone' data betreft, of data waarin een bericht verborgen zit);
7. Je programma moet in staat zijn om een tekstbericht in 'schone' data te verstoppen;
8. Je programma moet die data met bericht kunnen schrijven naar een bestand van hetzelfde formaat;
9. Dat tekstbericht moet in UTF-8 geëncodeerd zijn (dat betekent concreet dat je soms multi-byte tekens kunt tegenkomen);
10. Het verborgen bericht is in leesbare vorm een reeks bytes, afgesloten met een `NUL`-byte (letterlijk de waarde `0`). Daarmee is tijdens het lezen dus te bepalen wat het einde van het bericht is;
11. Van elk byte van het bericht zet je elk bit in een volgende geschikte audiosample, in de volgorde van meest naar minst significant bit.
12. In de audiobestanden moet de data als ongecomprimeerde 16-bit-samples zijn opgeslagen. In jargon heet dat *16 bit linear PCM*.
13. Van audiosamples verander je alleen het minst significante bit van elke sample naar een waarde voor jouw bericht.
14. Het resulterende bestand moet probleemloos te openen zijn in een player, waarbij de oorspronkelijke inhoud niet merkbaar in kwaliteit is afgenomen;
15. Als gelezen data een verborgen bericht bevat, moet dat bericht ontcijferd en getoond kunnen worden;

16. Je gebruikt in principe geen raw pointers, maar smart pointers uit de Standard Library. Let op: wanneer je wel raw pointers gebruikt moet je kunnen beargumenteren waarom in die situatie niet gekozen is voor een smart pointer;
17. Het is niet toegestaan om buiten een RAII class 'naked' `delete` te gebruiken;
18. Wanneer je een `std::shared_ptr` gebruikt moet je kunnen toelichten waarom je geen `std::unique_ptr` toepast;
19. We verwachten dat jouw code `const`-correct is.

3 Beoordeling

Voldoe je aan bovenstaande requirements, dan heb je een 6,0.

3.1 Bonuspunten

Onderstaande punten zijn geen requirements. Je kunt er voor kiezen ze mee te nemen voor extra punten.

1. Ondersteuning voor meer dan één bestandformaat. (Als je Wave kunt lezen en schrijven, probeer dan het lezen en schrijven van AIFF te implementeren. Als dat is gelukt, probeer dan het lezen en schrijven van CAF te implementeren.)
    - Een extra bestandsformaat kunnen lezen: +0,5 punt;
    - Een extra bestandsformaat kunnen schrijven: +0,5 punt;
2. Bouw ondersteuning in voor 24-bit samples. Voor elk bestandsformaat waarvoor je dat hebt gerealiseerd: +0,5 punt.
3. Je mag de ondersteuning voor je verschillende bestandsformaten onderbrengen in plugins (DLLs), die op runtime worden geladen. Je zou plugins in een eigen directory kunnen zetten die je bij het opstarten van je programma scant, zodat op die manier vanzelf blijkt welke bestandsformaten ondersteund worden. Als je een plugin per bestandsformaat hebt gerealiseerd: +1,0 punt.

3.2 Minpunten

- Als je bij de implementatie van de basiseisen zaken niet helemaal compleet of correct gebouwd hebt, worden er punten afgetrokken: −1 punt per gemist item.
- C++-idioom: we veronderstellen dat je C++ correct gebruikt. Er zal puntenaftrek plaatsvinden wanneer dat niet het geval is. Bijvoorbeeld:
  - Parameter passing: *primitives* en `std::shared_ptr` horen by value doorgegeven te worden; *objects* moeten by (const) reference: −0,5 punt per geval wanneer niet.
  - Const correctness: methods zijn `const` wanneer ze een instantie niet wijzigen, en objecten worden als `const` reference doorgegeven wanneer ze niet gewijzigd worden door die method: −0,5 punt per geval wanneer niet op die manier.
  - We willen geen *naked* `delete` zien, behalve in zelf geschreven RAII classes: −1 punt per *naked* `delete`.

  De assessor kan naar eigen inzicht op dit gebied punten aftrekken voor zaken die hier niet expliciet genoemd staan, maar die je redelijkerwijs zou moeten weten.

4 Tips

1. Geef de voorkeur aan classes en functies uit de Standard Library boven eigengemaakte code!
2. Denk aan de juiste modus waarmee je bestanden opent. Het zijn binaire bestanden!
3. Je hoeft lang niet alle metadata uit de bestanden te kunnen interpreteren. Bij het lezen is slechts van belang dat je de *data* kunt vinden. Bij het schrijven kun je alle metadata ongewijzigd overnemen (kopiëren, dus). Het idee is immers dat je de algehele structuur ongewijzigd laat, en slechts bepaalde bits van de audio data een door jou gekozen waarde geeft.
4. Als je een bruikbaar bestand krijgt aangeboden weet je nog niet of daar een verborgen bericht in zit. Je gaat dus gewoon lezen zoals gespecificeerd, totdat je het afsluitende NUL-byte bent tegengekomen. Als het een geldig bericht is zal de resulterende string dus geldige UTF-8 zijn. Als er geen bericht in de data zat, is dat waarschijnlijk niet het geval.

5. Zorg dat je eerst de meegeleverde file(s) kunt lezen, en daar een geldige tekstboodschap uit kunt halen. Als dat lukt weet je dat je de juiste aanpak hebt. Je kunt dan het schrijven implementeren, dat je vervolgens kunt testen door de opgeleverde files te lezen, en af te spelen in een player.

6. Mocht je er voor kiezen om een plugin-architectuur toe te voegen, maak dan een RAII class om het management van plugins te regelen.

7. Als je meer dan alleen bestandsnamen als command-line argumenten accepteert, kijk dan naar `getopt` (uit `<unistd.h>`). Als je ook GNU long options will ondersteunen, gebruik dan `getopt_long` (uit `<getopt.h>`).

8. Onder Windows is het minder gebruikelijk (en is de ondersteuning ook minder) voor command-line opties. Je kunt er daarom ook voor kiezen om opties in een configuratiebestand te zetten en dat in te lezen. Dat is nog steeds een handige manier om het runtime-gedrag van je programma te kunnen beïnvloeden zonder opnieuw te bouwen.

Veel succes!

5 Benodigde kennis

Om deze opdracht tot een goed einde te brengen zul je enkele technieken moeten beheersen die met *low level programming* in C++ te maken hebben. Daarmee helpen we je hier een beetje op weg. Maar eerst iets over de audio bestandsformaten.

5.1 Bestandsformaten

De bestandsformaten die we gaan implementeren zijn *binaire* formaten (dus *geen tekst*). Vergeet dus niet om files te openen in *binary mode*.

5.1.1 Data format vs. file format

De vorm waarin we audio data opslaan heet het *data format*. Verwar dat dus niet met het *file format* (bestandformaat).

De in de requirements genoemde *16 bit linear PCM* is een *data format* dat door heel veel *file formats* wordt ondersteund. De bestandsformaten waar we naar gaan kijken (Wave, AIFF en CAF) kunnen audio data op die manier opslaan.

Toevallig zijn alle drie deze bestandsformaten in staat om *meerdere* data formats op te slaan. Ze ondersteunen bijvoorbeeld ook diverse compressie-formaten.

Er zijn natuurlijk ook bestandsformaten die slechts één specifieke vorm opslaan: denk bijvoorbeeld aan `mp3` files. De audio data die daarin staat is altijd volgens de *mp3*-standaard geëncodeerd.

5.1.2 Chunks

Al deze bestandsformaten werken op vergelijkbare wijze: ze slaan verschillende stukken data in aparte blokken op (vaak *chunks* genoemd). Elke chunk begint met een *identifier*, gevolgd door een integer die het aantal bytes aangeeft van de data die daarna komt, en daarna de data waar het eigenlijk om gaat.

Een chunk bestaat dus uit 3 delen:

- *chunk identifier* (4 bytes), meestal leesbare ASCII-tekens;
- *chunk size* (4 bytes bij Wave en AIFF, 8 bytes bij CAF);
- *chunk data* (*n* bytes, waarbij *n* de chunk size is).

Door data op deze manier te organiseren kunnen er willekeurige "blokjes" data in de file staan, waarbij de chunk identifier aangeeft om wat voor data het gaat.

Het mooie hiervan is dat wanneer je een onbekende chunk identifier tegenkomt je die chunk gemakkelijk kunt overslaan, omdat je dankzij de aangegeven grootte weet hoeveel bytes je moet skippen. Hierbij komt de `seekg`-method van `std::ifstream` goed van pas.

Een programma dat een dergelijk file format schrijft mag de benodigde chunks in willekeurige volgorde wegschrijven. Een programma dat het file format moet kunnen lezen moet er dus rekening mee houden dat de volgorde niet gegarandeerd is!

Je kunt zoiets dus aanpakken door bijvoorbeeld actief op zoek te gaan naar een bepaalde chunk, als je die nodig hebt. Of je kunt eenmalig een inventarisatie maken van alle chunks in de file, zodat je later (bijvoorbeeld in een `std::map`) kunt terugzoeken waar ergens in de file een bepaalde chunk staat, en hoe groot die is.

5.1.3 Wave file format

Wave[1] (`*.wav`) files zijn ontwikkeld door Microsoft, en worden door heel veel programma's en libraries ondersteund.

Zoals je in de online beschrijving kunt zien zijn er 2 specifieke chunks minimaal nodig om een geldige Wave file te maken: de `'fmt '` chunk en de `'data'` chunk.

In de eerste staat de meta-data, zoals:

[1] http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

- wat het data format is;
- hoe groot elke sample is (bijvoorbeeld 16 bit);
- wat de sample rate is (bijvoorbeeld 44100 Hz);
- hoeveel kanalen er zijn (1 = mono, 2 = stereo);
- enzovoorts.

Gelukkig hoef je in dit project niets te doen met deze meta-data, tenzij je expliciet wilt controleren of het data format aan de eisen voldoet. En als je voor het bonus-punt gaat waarbij je ook 24-bit samples wilt ondersteunen, zul je natuurlijk wel de sample size echt moeten checken.

De feitelijke audio data staat in de `'data'` chunk. In de bij deze opdracht geleverde voorbeeldbestanden, waarin al verborgen berichten zijn verstopt, begint de audio data in de `'data'` chunk steeds op file offset 44, maar daar mag je dus niet van uit gaan! Je hoort netjes op zoek te gaan naar waar de `'data'` chunk precies in de file staat, en waar de audio data dan precies begint.

Alle informatie in een Wave file is in *litte endian* volgorde opgeslagen. Voor een uitleg over *little endian* en *big endian*, zie: §5.5, pag. 12.

5.1.4 AIFF file format

AIFF[2] (`*.aif`) files zijn door Apple bedacht. De naam staat voor *Audio Interchange File Format*.

De informatie hierin is op vergelijkbare wijze opgeslagen als in Wave files, alleen is alle data in *big endian* volgorde opgeslagen, en hebben de chunks andere identifiers.

De meta-data staat in de `'COMM'` chunk, en de audio data in de `'SSND'` chunk. Let op: de `'COMM'` chunk zit anders in elkaar dan de `'fmt '` chunk van een Wave file!

5.1.5 CAF file format

CAF[3] (`*.caf`) files zijn door Apple bedacht. De naam staat voor *Core Audio File*.

Dit is een veel nieuwer file format dan Wave en AIFF. CAF lost enkele problemen op van de oude bestandsformaten.

Wave en AIFF zijn zo oud (ze stammen uit de jaren 80), dat het gebruik van 4-byte integers om chunk sizes aan te geven een limiet geeft aan de bestandsgrootte

[2] http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/AIFF/AIFF.html

[3] https://developer.apple.com/library/archive/documentation/MusicAudio/Reference/CAFSpec/CAF_overview/CAF_overview.html

van 2 GB. Dat is in moderne tijden te klein. Ook ondersteunen ze niet op een standaard manier het gebruik van meer dan 2 audiokanalen, zoals bij surround sound nodig is.

Het belangrijkste verschil tussen CAF enerzijds, en Wave en AIFF anderzijds, is dat in een CAF file de chunk sizes integers van 8 bytes zijn (64 bits, dus). Voor de rest is de aanpak weer vergelijkbaar met wat we eerder gezien hebben. Natuurlijk heeft CAF zijn eigen chunk identifiers, met specifieke indelingen.

Bij CAF zit de meta-data in een `'desc'` chunk, en de audio data in een `'data'` chunk.

De informatie in een CAF file kan zowel in *litte endian* als *big endian* volgorde zijn opgeslagen, afhankelijk van het gebruikte *data format*.

5.2 Individuele bits manipuleren

Een bit kan `0` of `1` zijn. Om bitmanipulaties te begrijpen, moet je eerst weten hoe een integer getal als bits wordt gerepresenteerd.

5.2.1 Getalrepresentaties

Wat voor bitpatroon is dan bijvoorbeeld de waarde 42? Dat is `101010`, en dat heet de *binaire* weergave van deze waarde.

In feite is dit een representatie in een tweetallig stelsel. Wij zijn normaliter gewend aan een *decimale* representatie, dus een tientallig stelsel.

De waarde van elk cijfer wordt bepaald door de positie: de `2` in `42` is `2` waard, en de `4` is `40` waard. Opgeteld is dat `42`.

Meer wiskundig: $42 = 2 \times 10^0 + 4 \times 10^1$

De binaire weergave is in een tweetallig stelsel, en heeft dus grondtal `2`. We kunnen dan opschrijven: $101010 = 0\times2^0+1\times2^1+0\times2^2+1\times2^3+0\times2^4+1\times2^5$.

5.2.2 Bitmanipulatie: setten, resetten, inverteren of testen van individuele bits

In ons programma moeten we individuele bits kunnen lezen en schrijven. We gebruiken daarvoor bij voorkeur de bitmanipulatie-operatoren van C++, die de taal heeft geërfd van C.

Dat zijn: de *bitwise OR*-operator (`|`), de *bitwise AND*-operator (`&`), de *bitwise NOT*-operator (`~`), en de *bitwise XOR*-operator (`^`).

Behalve bij de `NOT`-operatie combineren ze twee integer waarden, waarbij steeds een `OR`-, `AND`- of `XOR`-operatie wordt uitgevoerd met overeenkomstige bits. Hieronder staan de *waarheidstabellen*, die aangeven wat de uitkomsten zijn bij verschillende combinaties van bits.

```
0011        0011        0011        <-- eerste getal
0101        0101        0101        <-- tweede getal
---- OR     ---- AND    ---- XOR
0111        0001        0110        <-- uitkomst
```

Dat leidt tot de volgende mogelijkheden:

Wanneer je een specifiek bit van een integer variabele de waarde `1` wil geven (*setten*), dan kun je die variabele met bitwise OR combineren met een constante, die op de plek van het overeenkomstige bit een `1` heeft staan, en verder alleen maar bits met de waarde `0`. Zo'n constante wordt vaak een *masker* genoemd.

Wanneer je een specifiek bit van een integer variabele de waarde `0` wil geven (*resetten*), dan kun je die variabele met bitwise AND combineren met een masker dat uit alleen maar bits met de waarde `1` bestaat, behalve bij het bit dat moet worden gereset.

Wanneer je wilt weten of een specifiek bit van een integer variabele de waarde `0` of `1` heeft, kun je het met bitwise AND combineren met een masker waarin alle bits `0` zijn, behalve het te testen bit. De uitkomst is dan `0` als het bit `0` is, en ongelijk aan `0` als het bit `1` is.

Het volgende code-fragment laat zien hoe je dat gebruikt.

```
uint8_t val {42};     // 00101010
uint8_t mask1 {1};    // 00000001
uint8_t mask2 {0xFD}; // 11111101

// using OR with mask1 will set bit 0
std::cout << (val | mask1) << '\n'; // will output 43 (00101011)

// using AND with mask2 will reset bit 1
std::cout << (val & mask2) << '\n'; // will output 40 (00101000)

// using AND with mask1 is used to test bit 0
if (val & mask1) {
    std::cout << "bit 0 has value 1\n";
} else {
    std::cout << "bit 0 has value 0\n"; // this will be printed
}
```

5.2.3 Shift operatoren

Wat ook nog handig kan zijn voor het werken met maskers is het *opschuiven van bits* naar links of rechts. Daarvoor erft C++ uit C de *shift operatoren*: **<<** en **>>**, voor schuiven naar links, respectievelijk rechts.

```
uint8_t mask {1};       // 00000001
uint8_t sl {mask << 3}; // 00001000, want 3 posities naar links
uint8_t sr {sl >> 1};   // 00000100, want 1 positie naar rechts

// combineren met toekenning mag ook
mask <<= 1; // 00000010, want 1 positie naar links
```

Aan de kant waar je naar toe schuift, "valt er een bitje uit", dat ben je dus kwijt. Aan de andere kant schuift er altijd een `0` naar binnen. Uitzondering: wanneer je een *signed* integer waarde naar rechts schuift, wordt er een `1` in geschoven wanneer het getal negatief is.

5.2.4 Meer info

Meer informatie over de achterliggende concepten is te vinden in Wikipedia-artikelen over bitwise operations[4] en bit maskers[5].

5.3 Werken met *signed* en *unsigned* integers

Integers zijn standaard signed, dat wil zeggen dat hun hoogste bit wordt gebruikt als plus- of min-teken. Als je expliciet wilt aangeven dat ze alleen waarden van 0 of groter kunnen hebben, kun je het keyword `unsigned` bij de declaratie opnemen, zoals in: `unsigned int value = 0;`

Of een integer signed of unsigned is heeft consequenties voor:

- het bereik van mogelijke waarden, bijvoorbeeld voor 16-bits integers: van `-32768` tot en met `32767` (signed) of `0` tot en met `65535` (unsigned),
- gedrag bij bit-shift-operatoren (`>>` en `<<`).

Een compiler zal ook warnings geven wanneer je een signed integer met een unsigned integer vergelijkt. Het kan namelijk goed fout gaan:

```
void fun(unsigned int a) {
    int b = 0;
    if (a < b) {
        std::cout << "Never here\n";
    }
}
```

[4] https://en.wikipedia.org/wiki/Bitwise_operation
[5] https://en.wikipedia.org/wiki/Mask_(computing)

5.4 Werken met verschillende maten integer en floating point

Normaliter zul je nauwelijks geïnteresseerd zijn in hoe integers en floating point-getallen intern worden gerepresenteerd, behalve misschien wanneer je met rekenwerk buiten de kleinste of grootste mogelijke waarde dreigt te komen (zie daarvoor ook `<limits>`[6], waarin constanten staan gedefinieerd voor alle numerieke limieten).

In dit project hebben we echter te maken met bestandsformaten waarvoor heel precies is vastgelegd uit hoeveel bytes elke numerieke waarde bestaat. We moeten dus ook in onze code in staat zijn daarmee om te gaan. In §6.2 van het boek staan de mogelijkheden voor fundamentele data types uitgelegd. Zorg dat je weet hoe het zit met `char`, `int`, `short`, `long`, `long long`, enz. Kijk ook naar de beschrijving van de header `<cstdint>`[7], waarin types worden gedefinieerd voor specifieke groottes (in bytes), zoals `int16_t`, `int32_t`, `uint32_t` en dergelijke.

5.5 *Little-endian* en *big-endian* byte order

Integer- of floating-point-waarden bestaan meestal uit meerdere bytes. (Alleen `char` is één byte groot.)

Elke processor heeft een eigen volgorde waarin de bytes in het geheugen worden geplaatst.[8] Als je nu een programma maakt dat een integer naar een binair bestand schrijft, zullen de bytes waaruit die integer bestaat, ook *in die volgorde* in het bestand terecht komen.

Als je die integer op een later moment uit het bestand terugleest, zal dat alleen het juiste getal opleveren als de computer waarmee je dat doet een processor heeft die de bytes *in dezelfde volgorde* zet als de processor in de computer waarmee het bestand werd geschreven.

Je moet dus weten wat de volgorde was, om in staat te zijn de juiste waarde uit het bestand te krijgen. Want als je tegelijkertijd weet wat de gebruikelijke volgorde is voor de computer waar je programma op draait, kun je, indien nodig, zelf de bytes *omdraaien*.

Wij gaan werken met bestandsformaten waarbij nauwkeurig is vastgelegd (in de officiële beschrijving van het formaat) of data als little-endian of big-endian is opgeslagen.

Bij *little-endian* komt het minst significante byte eerst; bij *big-endian* komt het meest significante byte eerst. Intel-processoren hebben een little-endian architectuur; big-endian kom je bijvoorbeeld tegen bij PowerPC. Ook wanneer je binaire data over een TCP/IP-netwerk stuurt gebeurt dat in big-endian volgorde. Dat

[6] https://en.cppreference.com/w/cpp/types/numeric_limits
[7] https://en.cppreference.com/w/cpp/header/cstdint
[8] https://en.wikipedia.org/wiki/Endianness

wordt *network byte order* genoemd.

Mocht je op runtime willen achterhalen wat de byte-order van de huidige processor is, kun je onderstaande header (`byte_order.hpp`) gebruiken:

```
#ifndef byte_order_h
#define byte_order_h

namespace su {

    enum class byte_order {
        little_endian,
        big_endian
    };

    byte_order cur_byte_order() {
        const unsigned short val
            {*reinterpret_cast<const unsigned short *>("az")};

        return val == 0x617AU ?
            byte_order::big_endian :
            byte_order::little_endian;
    }

}

#endif /* byte_order_h */
```

In de functie `cur_byte_order` wordt gebruik gemaakt van de techniek om via twee verschillende typen pointers naar dezelfde data te kijken, en die data dus te 'herinterpreteren'. Vandaar de lelijke `reinterpret_cast`. *Low level tricks* vereisen nu eenmaal dit soort 'hacks'. De source file hieronder laat zien hoe je de enumeratie en de functie kunt gebruiken.

```
#include <iostream>
#include "byte_order.hpp"

int main() {
    std::cout << "Your processor has a " <<
        (su::cur_byte_order() == su::byte_order::little_endian ?
         "little" : "big") << " endian architecture\n";
}
```

5.6 *Shared libraries* gebruiken als plugin (bonus)

Een shared library is een bestand met executable code, die op runtime in een draaiend programma geladen kan worden. Daarmee kun je dus plugins realiseren.

Vanuit architectuur-oogpunt is het gebruik van plugins een mooi voorbeeld van maximale modulariteit. Je kunt er bijvoorbeeld voor kiezen om in elke plugin een class te implementeren, die is afgeleid van een *pure abstract base class*. Dan kan je hoofdprogramma, dat de plugins inlaadt, concrete objecten (laten) instantiëren uit de plugin, die vervolgens met de API worden aangesproken die in de abstract base class is vastgelegd. Dat heeft als voordeel dat je eigenlijk maar één functie hoeft te exporteren, namelijk eentje die als *factory* voor de in de plugin verpakte class fungeert. Het hoofdprogramma zoekt dan die functie op en roept hem aan om daaruit een instantie terug te krijgen.

Op UNIX/Linux-systemen heb je de header `<dlfcn.h>`[9] nodig, en moet je programma gelinkt worden met `libdl` die standaard in het systeem zit. Je zult gebruik maken van de functies `dlopen`, `dlclose`, `dlsym` en `dlerror`. Hiermee kun je een shared library openen, sluiten, namen opzoeken (van functies bijvoorbeeld), en error messages opvragen als er iets mis ging.

Op Windows werkt het anders[10].

Een klein addertje onder het gras: een C++-compiler verhaspelt de namen van functies en classes[11], waardoor opzoeken van de juiste naam in een shared library lastig wordt (en compiler-afhankelijk). Het is daarom een beter idee om de functie(s) die je direct zou willen benaderen vanuit je hoofdprogramma *C-linkage* te geven. Dat doe je door '`extern "C"`' vóór de declaratie van een te exporteren functie te zetten. Een uitleg daarover staat in het boek in §15.2.5 vanaf pagina 428.

[9] https://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html

[10] https://stackoverflow.com/questions/53530566/

[11] Dat heet officieel *name mangling*.