Eindopdracht CPPLS2

Avans Hogeschool

4 november 2019

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 enkele image en/of audio bestandsformaten kan lezen en schrijven. Bij het lezen zul je specifieke bitjes in de data gaan verzamelen en samen pakken om daar een verborgen bericht uit te halen. Er zullen geldige bestanden worden aangeleverd waarin al berichten verborgen zitten. Ook zul je een gegeven bericht in zo'n bestand moeten kunnen verstoppen, op een voorgeschreven manier. Je hebt vast en zeker programma's op je systeem staan om images te tonen en muziek af te spelen. Daarmee kun je snel testen of de bestanden die je bouwt zich in elk geval als normale multimediabestanden gedragen.

Pluspunten kun je halen met een implementatie in de vorm van plugins (*shared libraries* of *DLLs*) voor het lezen en schrijven van de verschillende bestandsformaten.

Zie voor een precieze omschrijving paragraaf 2.3, 'Requirements', op pagina 7.

2.2 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.

De bestandsformaten die we gaan implementeren zijn *binaire* formaten (dus *geen tekst*).

We moeten dan onder andere in staat zijn om integers (en floating-point-getallen) van verschillende grootte (8-, 16-, 32- of 64-bits) te lezen en te

schrijven (§2.2.2), waarbij we ook nog onderscheid maken tussen *signed* en *unsigned* integers (§2.2.1).

Bovendien kunnen die getallen, als ze uit meerdere bytes bestaan, in *little-endian* of *big-endian* vorm worden opgeslagen. Dat gaat over de volgorde van de bytes: van voor naar achter, of andersom (§2.2.3).

Verder moeten we natuurlijk voldoende gedetailleerde informatie hebben over de bestandsformaten die we willen kunnen lezen en schrijven. Wij gebruiken alleen *ongecomprimeerde* bestandsformaten, omdat we daar meer ruimte hebben om (vrijwel) ongemerkt bitjes aan te passen.

Hieronder staan de benodigde kennis en vaardigheden verder toegelicht.

2.2.1 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 vergelijkt. Het kan namelijk goed fout gaan:

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

2.2.2 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>`[1], waarin constanten staan gedefinieerd voor alle numerieke limieten).

[1] https://en.cppreference.com/w/cpp/types/numeric_limits

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>`[2], waarin types worden gedefinieerd voor specifieke groottes (in bytes), zoals `int16_t`, `int32_t`, `uint32_t` en dergelijke.

2.2.3 *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.[3] 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 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
```

[2] https://en.cppreference.com/w/cpp/header/cstdint
[3] https://en.wikipedia.org/wiki/Endianness

```
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";
}
```

2.2.4 Individuele bits manipuleren

In ons programma moeten we individuele bits kunnen lezen en schrijven. We gebruiken daarvoor de bitmanipulatie-operatoren van C++.

Meer informatie over de achterliggende concepten is te vinden in Wikipedia-artikelen over bit manipulatie, bitwise operations en bit maskers.

In het boek wordt er in §11.1.2 vanaf pagina 274 ook het een en ander over uitgelegd.

Zoek de volgende dingen uit:

- hoe je bij een integer variabele in C++ een gewenst bit kunt

  - *setten* (naar `1` forceren),
  - *resetten* (naar `0` forceren),
  - of *inverteren* (van `0` naar `1` of omgekeerd);

- hoe je uit een gegeven integer waarde een specifiek bit kunt lezen, of, daaraan gerelateerd,

- hoe je kunt *testen* of een bit de waarde `0` of `1` heeft.

2.2.5 *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>`[4] 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[5].

Een klein addertje onder het gras: een C++-compiler verhaspelt de namen van functies en classes[6], 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.

[4] https://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html
[5] https://stackoverflow.com/questions/53530566/
[6] Dat heet officieel *name mangling*.

2.2.6 Bestandsformaten

Er zijn vele tientallen bestandsformaten voor audio of image data. Wij kijken naar twee audio formats en twee image formats: AIFF[7] (`*.aif`) en Wave[8] (`*.wav`); en PNG[9] (`*.png`) en TIFF[10] (`*.tif`).

In AIFF en Wave kun je vergelijkbare (meta-)data kwijt, hoewel de data bij elk op hun eigen manier wordt opgeslagen. De data in een AIFF-bestand is in big-endian volgorde opgeslagen; in een Wave-bestand is dat little-endian. PNG data staat in big-endian volgorde in een bestand, en in een TIFF-bestand kan de data zowel little-endian als big-endian zijn (dat staat aangegeven in de eerste bytes van de file).

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 grootte in bytes.

Het mooie hiervan is dat wanneer je een onbekende chunk identifier tegenkomt je die chunk 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.

Deze bestandsformaten zijn in staat om audio- of imagedata op te slaan van diverse formaten. Om goed onderscheid te kunnen maken gebruiken we de termen *file format* en *data format*.

Een Wave-file, bijvoorbeeld, kan onder andere 16-bits stereo ongecomprimeerde audio data opslaan. Dat kan net zo goed in een AIFF-file. Beide ondersteunen dus dit data format. Ook kunnen ze allebei volgens A-law en μ-law geëncodeerde data opslaan. Wat dat precies voor data formats zijn doet er even niet toe; het punt is dat beide file formats deze data formats ondersteunen.

De consequentie is dat er dus in de metadata terug te vinden is wat het data format van de opgeslagen audio is. Hier moet je dus rekening mee houden, omdat je in deze opdracht alleen een paar simpele data formats gaat ondersteunen.

Een vergelijkbaar verhaal gaat op voor de image formats.

2.3 Requirements

1. Je maakt een programma zonder grafische user interface;

2. Dat programma is geheel geschreven in C++;

[7] http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/AIFF/AIFF.html
[8] http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html
[9] https://www.w3.org/TR/2003/REC-PNG-20031110/
[10] https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf

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 uit minstens één bestandsformaat data te kunnen lezen (of dat nu 'schone' data betreft, of data waarin een bericht verborgen zit);

7. Dat bestandsformaat moet AIFF, Wave, PNG of TIFF zijn;

8. Je programma moet in staat zijn om een tekstbericht in 'schone' data te verstoppen;

9. Je programma moet die data met bericht kunnen schrijven naar een bestand van hetzelfde formaat;

10. Dat tekstbericht moet in UTF-8 geëncodeerd zijn (dat betekent concreet dat je soms multi-byte tekens kunt tegenkomen);

11. 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;

12. Van elk byte van het bericht zet je elk bit in een volgende geschikte data-eenheid (audiosample, pixelwaarde), in de volgorde van meest naar minst significant bit.

13. In de audiobestanden moet de data als ongecomprimeerde samples zijn opgeslagen. In jargon heet dat *linear PCM*.

14. In imagebestanden moet de data als ongecomprimeerde pixel-data zijn opgeslagen. Waarschijnlijk is RGB of RGBA[11] het meest gebruikelijk.

15. In audiobestanden moeten samples minstens 16-bit zijn.

16. Van audiosamples verander je alleen het minst significante bit van elke sample naar een waarde voor jouw bericht.

[11] Rood, Groen, Blauw, Alpha (transparantie)

17. Van pixel-data verander je alleen het minst significante bit van het blauw-kanaal. Anders ga je kleurenruis zien. In plaats van blauw zouden we natuurlijk elk ander kanaal kunnen kiezen, maar door deze afspraak worden bestanden onderling uitwisselbaar.

18. Als je programma meerdere bestandsformaten ondersteunt, mogen dat uiteraard ook andere dan de hierboven genoemde zijn, zolang de data ongecomprimeerd is (dus *geen* MP3 of JPEG).

19. Het resulterende bestand moet probleemloos te openen zijn in een player/viewer, waarbij de oorspronkelijke inhoud niet merkbaar in kwaliteit is afgenomen;

20. Als gelezen data een verborgen bericht bevat, moet dat bericht ontcijferd en getoond kunnen worden;

21. 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;

22. Het is niet toegestaan om buiten een RAII class 'naked' `new` of `delete` te gebruiken;

23. Wanneer je een `std::shared_ptr` gebruikt moet je kunnen toelichten waarom je geen `std::unique_ptr` toepast;

24. We verwachten dat jouw code `const`-correct is.

2.4 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.

2. Je mag de ondersteuning voor je verschillende bestandsformaten onderbrengen in plugins, die op runtime worden geladen.

3. 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.

2.5 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 image- of audio data een door jou gekozen waarde geeft.

4. Begin met een eenvoudig bestandsformaat, bijvoorbeeld Wave. Je hoeft niet zo heel veel te lezen en interpreteren om er achter te komen waar de audio data staat, en of het gewone, ongecomprimeerde data is.

5. 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.

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>`).

3 Beoordeling

We zijn nog bezig om de rubric fijn te slijpen, maar in grote lijnen komt het hier op neer:

Je haalt een voldoende wanneer

- je een Wave-bestand kunt lezen, èn
- daar een eventueel verborgen bericht uit kunt halen, èn

- je een Wave-bestand kunt schrijven, èn
- daar een bericht in kan verbergen.

Een hoger punt krijg je onder andere door

- hetzelfde kunstje uit te halen met AIFF, PNG en/of TIFF;
- file formats te ondersteunen met plugins;
- nette toepassing van C++-idioom.

Er zal een rubric in Blackboard worden gepubliceerd waarin de exacte scores en wegingen staan.

Veel succes!