UNIVERSITEIT GENT
FACULTEIT INGENIEURSWETENSCHAPPEN EN ARCHITECTUUR
Vakgroep Informatietechnologie (INTEC)

Software Ontwikkeling I

Academiejaar 2012–2013

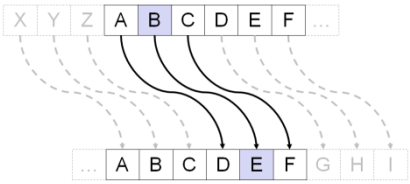
SOI@intec.UGent.be

Practicum 2

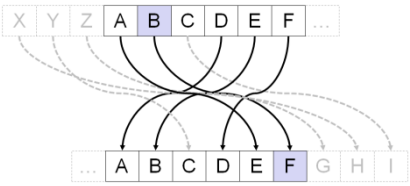
Decryptie a.d.h.v. frequentietabellen

Situering

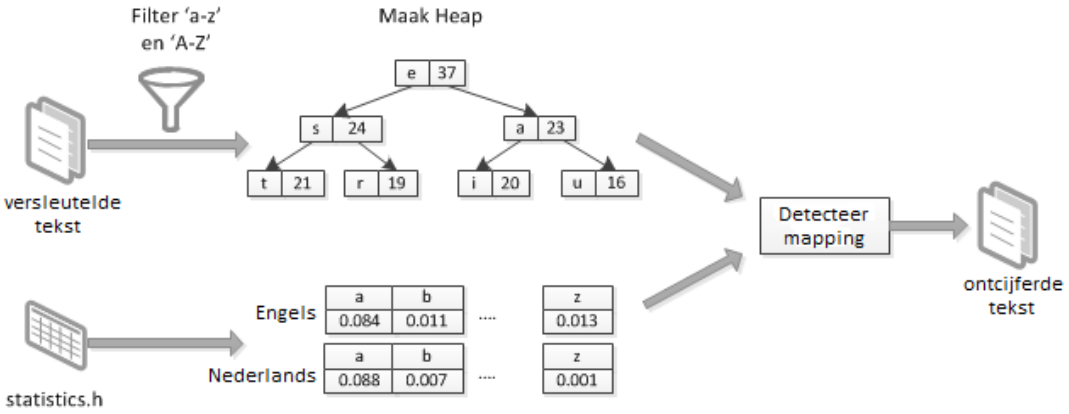
In dit practicum zullen we een frequentietabel implementeren, die kan worden gebruikt om een versleutelde tekst te ontcijferen. Een oude, bekende vorm van cryptografie is de Caesarrotatie. Bij deze vorm van cryptografie wordt het alfabet circulair geroteerd over een bepaalde afstand en wordt op basis van deze mapping een versleutelde tekst verkregen. Dit wordt in onderstaande figuur weergegeven voor een rotatieafstand 3.



Het is duidelijk dat dergelijke vorm van cryptografie erg gevoelig is voor brute-kracht aanvallen, aangezien voor een alfabet van 26 letters slechts 26 verschillende rotaties mogelijk zijn. Een algemenere vorm van cryptografie maakt gebruik van willekeurige permutaties van het alfabet. Op deze manier neemt het aantal mogelijke vercijfersleutels sterk toe tot 26!. Hierbij dient wel verzekerd te worden dat elk van de symbolen slechts 1x kan voorkomen als beeld van de permutatie (bv. indien a->d is vastgelegd is b->d niet langer mogelijk).



Toch is deze vorm van cryptografie niet als veilig te beschouwen, aangezien het gevoelig is voor een aanval gebruikmakend van frequentieanalyse. Frequentieanalyse maakt gebruik van het feit dat binnen elke geschreven taal een bepaalde frequentie waar te nemen is van letters of lettercombinaties. Hiertoe is een basiskennis van de karakteristieken van een bepaalde taal vereist. Zo komen bijvoorbeeld de e, n en a het meeste voor in het Nederlands. Als men de letterfrequentie van een versleutelde tekst vergelijkt met de normale waarden van de taal en bijvoorbeeld vaststelt dat de x, p en t het meeste voorkomen, kan men veronderstellen dat x staat voor e, p voor n en t voor a. Onder cryptoanalysten geldt de voorgestelde monoalfabetische substitutie als zeer eenvoudig.
De techniek van frequentieanalyse zal in dit practicum worden aangewend om enkele versleutelde berichten te ontcijferen. Om jullie op weg te helpen worden voor enkele talen de karakteristieken voorzien in de vorm van een array met daarin per letter de relatieve frequenties. De frequentietabel zal in dit practicum worden voorgesteld als een wachtlijn met prioriteit (heap datastructuur, meer informatie in de cursus, Appendix C, op p.247 of op Minerva). Dit laat ons toe om sequentieel telkens het meest voorkomende karakter uit de frequentietabel te halen. De gevolgde aanpak wordt in onderstaande figuur weergegeven. Om de complexiteit van dit practicum te beperken worden enkel de symbolen a-z en A-Z in rekening gebracht. Andere symbolen zoals spaties en leestekens blijven onveranderd in de geëncrypteerde en de leesbare tekst. Bovendien dient de mapping enkel voor symbolen a-z te worden vastgelegd. Deze wordt dan overgenomen voor de symbolen A-Z (bv. de mapping a->d impliceert de mapping A->D).



Voor het detecteren van de mapping kunnen voor deze opgave volgende aannames verondersteld worden:

- De frequenties zijn, zowel in de statistieken als in de heap, uniek per karakter. Er komen dus geen 2 karakters voor met eenzelfde frequentie.
- Bijgevolg kan, zowel voor de statistieken als voor de heap, een totale ordening van karakters worden vastgelegd.
- Dit betekent dat de mapping, die door de combinatie van beide ordeningen kan worden bekomen, ten allen tijde uniek is.

Opgave

Met deze opgave worden vijf bestanden geleverd:

- `main.c`: om jullie op weg te zetten worden de `main` en `actions` functies volledig meegeleverd. Deze roept, naast enkele hulpfuncties om te testen op memory leaks (zie ook "Appendix A: Opsporen van Memory leaks in MS Visual C++" achteraan deze opgave), een functie op die, gegeven een bestandsnaam, een tekstbestand inleest (door jullie te implementeren functie `read_file()` in `main.c`). Vervolgens worden uit dit bestand de gebruikte taal en de geëncrypteerde tekst onderscheiden en weergegeven (reeds meegegeven code). Daarna worden de functies, `create_ft()` en `decrypt()`, opgeroepen om dan ten slotte het dynamisch gealloceerde geheugen vrij te geven.
- `frequency_table.c`: dit bestand bevat bijna alle functies die jullie moeten implementeren in dit practicum. Jullie zijn vrij om hulpfuncties te implementeren, maar pas hierbij het header bestand NIET aan.
- `frequency_table.h`: het header-bestand voor de frequentietabel. (Dit headerbestand mag niet aangepast worden)
- `statistics.h`: de relatieve frequenties per taal kunnen hier worden teruggevonden. (Dit headerbestand mag niet aangepast worden)
- `Tekst1-2.txt`: de invoerbestanden hebben het volgende formaat [taal]#[geëncrypteerde tekst]. Bijvoorbeeld: "Engels#Zel iwnrtmc …"

frequency_table.h

De frequentietabel is een wachtlijn met prioriteiten (Heap datastructuur). Deze wachtlijn wordt geïmplementeerd zoals voorgesteld in de cursus op p.248. De elementen in deze wachtlijn zijn koppels, bestaande uit een karakter `c` en getal `n`, dat aantoont hoe vaak `c` voorkomt in de tekst. Dit koppel wordt voorgesteld door de volgende struct:

```
typedef struct{
	char c;
	unsigned int n;
} ft_entry;
```

De frequentietabel zelf wordt dan voorgesteld door de volgende `struct` en bevat de `array` van `ft_entry` koppels en het aantal elementen in de wachtlijn.

```
typedef struct{
	ft_entry* ft;
	unsigned int size;
} frequency_table;
```

statistics.h

De statistieken worden voorgesteld door middel van volgende struct en bevatten naast de naam van de taal ook een `array` van relatieve frequenties per letter in het alfabet. Zo bevat `frequencies[0]` de relatieve frequentie van de letter 'a' en `frequencies[25]` de relatieve frequentie van de letter 'z'.

```
typedef struct{
	char* language;
	const double frequencies[26];
} stat;
```

Merk hierbij wel op dat de `frequencies` als een constante gedefinieerd zijn. Indien je enige aanpassingen zou moeten doen aan de `array`, zal je de inhoud moeten kopiëren in een nieuwe array.

main.c

Implementeer de volgende functies:

```
char* read_file(const char* filename);
```

Met deze functie zal je een bestand, geïdentificeerd door `string filename`, kunnen inlezen en teruggeven als een `string` (zie Appendix B). Hierbij zal je gebruik moeten maken van de volgende functie, die de lengte van het bestand geeft:

```
unsigned long get_file_length(FILE* ifp);
```

In verband met het inlezen van tekstbestanden in C kan een voorbeeld worden teruggevonden in "Appendix B: File IO in C" achteraan deze tekst. Indien je problemen ondervindt bij het inladen van de tekstbestanden, kijk dan eerst of deze bestanden op de juiste plaats staan (In dezelfde map als het [projectnaam].vcxproj bestand).

frequency_table.c

Implementeer de volgende functies, met exact dezelfde signatuur als de declaraties in `frequency_table.h`:

```
frequency_table* ft_create(const char* encrypted);
```

Deze functie zal voor de geëncrypteerde tekst geïdentificeerd door `string encrypted` een frequentietabel maken. Na het initialiseren van de `frequency_table struct`, moet je alle karakters één voor één inlezen uit de tekst en zo de frequentietabel opbouwen. Maak hiervoor gebruik van de onderstaande functie `ft_insert`.

```
void ft_insert(const char c, frequency_table* ft);
```

Deze functie dient om een karakter toe te voegen aan de frequentietabel. Bij het opstellen van deze tabel dien je enkel rekening te houden met de letters van het alfabet (a-z en A-Z), waarbij hoofdletters worden omgezet in kleine letters. Andere karakters zoals spaties, tabs, cijfers en taalspecifieke elementen als ç, é, â, … mogen worden genegeerd. De functie werkt zoals uitgelegd in de cursus op p.248-250 en zal dus gebruik maken van het `fixup` algoritme. Merk wel op dat als een bepaald karakter `c` reeds ingevoerd werd, enkel zijn aantal voorkomens (`n`) verhoogd moet worden.

```
ft_entry* ft_getmax(frequency_table* ft);
```

Met deze functie wordt het karakter met het meeste aantal voorkomens uit de frequentietabel – hier dus uit de wachtlijn – gehaald. Als er geen elementen in de wachtlijn zitten moet de `NULL`-pointer teruggegeven worden. Deze methode is volledig gebaseerd op de werkwijze beschreven op p.248-250 van de cursus en zal dus gebruik maken van het `fixdown` algoritme. Om te zorgen dat iedereen dezelfde uitkomst heeft, spreken we hier enkele modaliteiten af:

- Tijdens de `fixup` functie zal een `ft_entry fc` enkel met zijn voorvader `p` gewisseld worden als:

  `fc.n > p.n`

- Tijdens de `fixdown` functie zal een `ft_entry fc` met zijn grootste kind `child` omgewisseld worden als:

  `fc.n < child.n`

Het kan handig zijn om aparte functies voor het `fixup` en `fixdown` algoritme te maken, zo kan je gebruik maken van recursie om deze functies te implementeren. Hiervoor werden de volgende functies gedeclareerd in het headerbestand `frequency_tabel.h` (met k gedefinieerd zoals in de cursus op p.248):

```
void ft_fixup(ft_entry* fc, const unsigned int k,
              const frequency_table* ft);

void ft_fixdown(ft_entry* fc, const unsigned int k,
              const frequency_table* ft);
```

Naast deze hulpfuncties is er nog een bijkomende hulpfunctie gedeclareerd. Deze dient ook geïmplementeerd en gebruikt te worden. Deze hulpfunctie zal twee elementen uit de frequentietabel van plaats wisselen:

```
void ft_swap(ft_entry* fc1, ft_entry* fc2);
```

Om het ontcijferen gemakkelijker te maken werd er nog een laatste hulpfunctie gedeclareerd. Deze functie zal de positie van het grootste element in de array teruggeven:

```
unsigned int get_max_index(double* list, unsigned int size);
```

Nu we over een ingevulde frequentietabel beschikken kunnen we overgaan tot het ontcijferen van de tekst:

```
char* decrypt(frequency_table* ft, const char* language,
              const char* encrypted);
```

Deze functie neemt een frequentietabel, een `string` die de taal voorstelt en de geëncrypteerde tekst als input en geeft de gedecrypteerde tekst terug. Hierbij dien je gebruik te maken van de frequentiestatistieken meegegeven in `statistics.h`. In deze datastructuur kunnen per taal de relatieve frequenties worden gevonden van de letters die voorkomen in het alfabet (ongeacht of het kleine of grote letters zijn). De bedoeling is om de voorkomens uit de frequentietabel te vergelijken met de voorkomens van de taal in de `stat statistics`. Zo komt de meest voorkomende letter in de frequentietabel (en dus de geëncrypteerde tekst) overeen met de meest voorkomende letter in `statistics`. Met behulp van een tabel kan je dan voor elke letter een vertaling opstellen. Hou bij het vertalen zeker rekening met hoofdletters en kleine letters. Hoofdletters in de geëncrypteerde tekst dienen ook hoofdletters te blijven in de leesbare tekst.
Ten slotte moet het geheugen, ingenomen door de frequentietabel, weer vrijgegeven worden via volgende functie:

```
void ft_free(frequency_table* ft);
```

Nog een belangrijk laatste puntje. Ten allen tijden moet je telkens zo zuinig mogelijk omspringen met het geheugen (m.a.w. niet meer geheugen gebruiken dan strict nodig).

Indienen: belangrijk

- Dit practicum dient individueel opgelost te worden. Plagiaat wordt niet getolereerd.
- Zip je broncode bestanden (main.c en frequency_table.c) in een bestand `naam_voornaam_nr.zip` waarbij nr het nummer van het practicum is, bvb. `volckaert_bruno_2.zip`. Gebruik een standaard zipformaat, geen rar of 7zip.
- Zet in je bronbestand bovenaan in commentaar je naam en de naam van het bestand.
- Respecteer de signatuur van de opgave. Het veranderen van de header-bestanden is NIET toegestaan. Doe je dit toch, dan zal zich dat reflecteren in je score.
- Stuur je finale (geen tussenversies) oplossing door middel van de dropbox op Minerva door naar Bruno Volckaert, ten laatste op **zondag 21 oktober, 23:59** (deadline).

Veel succes!

Appendix A: Opsporen van Memory leaks in MS Visual C++

De geheugenchecker in Visual C++ Express Edition kan je gebruiken als volgt. Je includeert in het betreffende bestand (meestal de main) volgende 2 headerfiles:

```
#include <stdlib.h>
#include <crtdbg.h>
```

Op de plaats waar je wil nagaan of er geheugen verloren gaat zet je volgende lijn

```
_CrtDumpMemoryLeaks();
```

Als je het programma in debug modus runt, dan krijg je na afloop van je programma melding of er geheugenlekken gevonden zijn, soms kan je uit die melding ook onmiddellijk afleiden waar het lek zit.

Let wel, als je variabelen op de stack aanmaakt (=statisch geheugen) in dezelfde functie als waar je `_CrtDumpMemoryLeaks();` statement staat, dan zullen deze variabelen ook gezien worden als *memory leaks* (aangezien ze nog niet out-of-scope gegaan zijn en dus nog geheugen innemen). Een goede manier om na te gaan of er geheugenlekken zijn is bvb:

```
void actions(...){
      //declaraties
      //allocaties op heap (via malloc/new/... = dynamisch geheugen)
      //uitvoeren programma
}
int main(){
      actions(...);
      _CrtDumpMemoryLeaks();
}
```

Zo ben je zeker enkel melding te krijgen van 'echte' geheugenlekken. Elk geheugenlek heeft een nummer dat je kan zien in de debug-output.

Verder kan je, eens je het nummer van een geheugenlek hebt, ook je programma aanpassen zodat het stopt met uitvoeren op de plaats waar het geheugen dat het lek veroorzaakt, gealloceerd wordt. Dit kan uitermate handig zijn bij het debuggen! Hiertoe voeg je in het begin van je programma volgende regel toe: `_CrtSetBreakAlloc(mem_leak_nr);`

Appendix B: File IO in C

Hieronder kan je een voorbeeld terugvinden van het gebruik van file IO in C:

```
#include <stdio.h>

#include <stdlib.h>
#include <crtdbg.h>

int main(int argc, char** argv) {
	char* lijn;
	FILE* file;
	/*_CrtSetBreakAlloc(mem_leak_nr);*/

	if(argc != 2){
		printf("No file name specified!\nExiting program...\n");
		return EXIT_FAILURE;
	}

	file = fopen(argv[1], "w");
	if(!file){
		printf("Fout bij het openen van %s!\n", argv[1]);
		return EXIT_FAILURE;
	}

	fputs("Dit is een IO test...", file);
	fclose(file);

	file = fopen(argv[1], "r");

	if(!file){
		printf("Fout bij het openen van %s!\n", argv[1]);
		return 1;
	}

	lijn = (char*)malloc(256*sizeof(char));
	fgets(lijn, 256,file);
	printf("Ingelezen lijn: %s\n", lijn);
	fclose(file);

	free(lijn);
	_CrtDumpMemoryLeaks();
	return EXIT_SUCCESS;
}
```