UNIVERSITEIT GENT
FACULTEIT INGENIEURSWETENSCHAPPEN EN ARCHITECTUUR
Vakgroep Informatietechnologie (INTEC)

Software-Ontwikkeling I

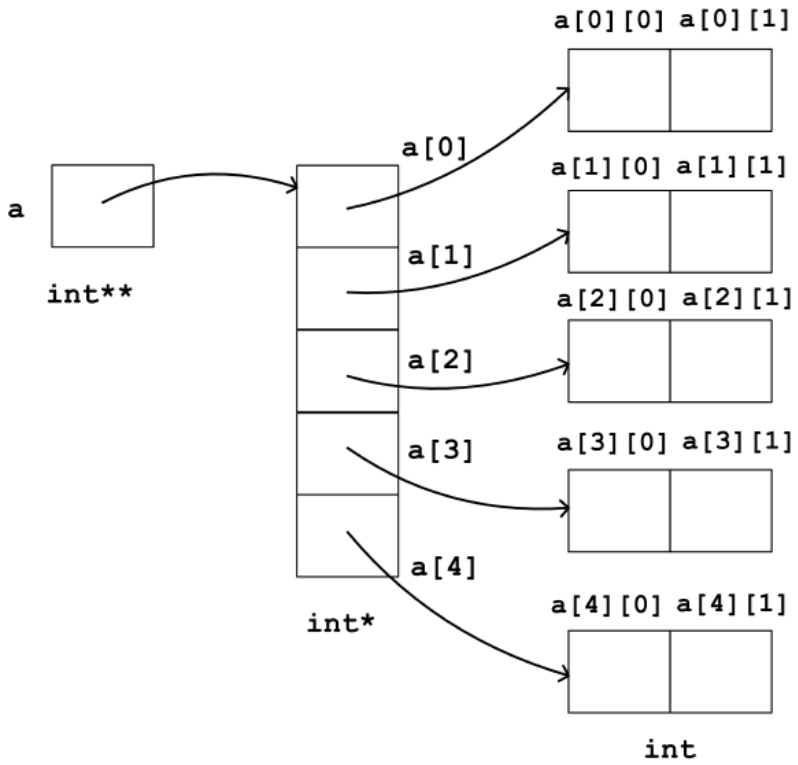
Academiejaar 2012 – 2013

SOI@intec.ugent.be

Practicum 3
Multidimensionale rijen

1. Inleiding

Dit practicum behandelt multidimensionale rijen in C (zie cursus sectie 6.6). Figuur 1 toont de geheugenstructuur van een 2-dimensionale rij. Wanneer er geheugen moet gealloceerd worden voor zo'n rij, wordt dat in 2 stappen gedaan: eerst wordt een blok gealloceerd om de wijzers naar de rijen van de matrix (in dit geval 4 wijzers) in op te slaan, daarna wordt per rij een blok gealloceerd om de elementen in de rij (in dit geval 2) op te slaan. Bij het vrijgeven van het geheugen wordt in omgekeerde volgorde gewerkt, eerst worden de rij-elementen vrijgegeven, daarna de rijen.



Figuur 1 - geheugenstructuur van een 2-dimensionale rij van gehele getallen

De eerste oefening gaat over het aanmaken van en werken met 2-dimensionale matrices. De tweede oefening gaat over het bewerken van afbeeldingen in 3-dimensionale rijen.

Belangrijk: bij het verbeteren wordt rekening gehouden met:

- efficiënt geheugenbeheer: hoe minder geheugen je gebruikt, hoe beter.
- vermijd geheugenlekken: op Minerva staat in het document 'doumenten/FAQ_VisualCppEE.pdf' de manier om je code te testen.
- hergebruik code: probeer reeds bestaande functies te hergebruiken, in plaats van code te kopiëren en plakken.
- Verander niks in de header bestanden of in `main.c`. Deze bestanden worden niet ingediend! `main.c` bevat functies om je code te testen.

2. Matrix bewerkingen

In deze oefening wordt een matrix van gehele getallen aangemaakt en enkele basis matrix-bewerkingen ontwikkeld. In `matrix.h` vind je de volgende `struct`:

```
typedef struct {
      int num_rows;
      int num_cols;
      int** data;
} matrix;
```

Deze houdt het aantal rijen en kolommen van de matrix bij en een pointer naar matrix `data` in het geheugen.

Gevraagd wordt nu om volgende methodes te implementeren in `matrix.c`:

`void init_matrix(matrix* m, int num_rows, int num_cols);`

- ⇨ initialiseert een matrix met het opgegeven aantal rijen en kolommen, reserveert geheugen voor de data en initialiseert alle elementen op 0.

`void init_matrix_default(matrix* m);`

- ⇨ doet hetzelfde als de vorig methode, maar gebruikt de default variabelen in `matrix.h` voor rij- en kolomgrootte.

`void init_identity_matrix(matrix* m, int dimension);`

- ⇨ intialiseert een eenheidsmatrix met grootte `dimension`. Een eenheidsmatrix is een vierkante matrix met enen op de hoofddiagonaal en de rest allemaal nullen.

`void free_matrix(matrix* m);`

- ⇨ geeft het geheugen ingenomen door de matrix vrij.

`void print_matrix(matrix* m);`

- ⇨ print de data van de matrix op een overzichtelijke manier (hou bij het printen rekening met het feit dat getallen soms meerdere karakters kunnen bevatten. Je mag hierbij veronderstellen dat getallen maximaal 3 karakters groot zijn).

`void transpose_matrix_pa(matrix* m);`

- ⇨ transponeert de matrix. Je mag er van uit gaan dat de matrix vierkant is. Gebruik in deze functie pointer-arithmetiek en de adresseringsmethode (**(matrix+i)*) voor het adresseren van de matrixelementen, ipv de notatie *matrix[i]*.

`void transpose_matrix_sn(matrix* m);`

- ⇨ transponeert de matrix. Je mag er opnieuw van uit gaan dat de matrix vierkant is. Gebruik in deze methode de *matrix[i]* addresseringsmethode.

`void multiply_matrices(matrix* a, matrix* b, matrix* result);`

- ⇨ initialiseert matrix *`result`* met het juiste aantal rijen en kolommen en slaat het resultaat van de vermenigvuldiging van matrix `a` en `b` erin op.

```
void dynamic_expand(matrix* m, int new_num_rows, int new_num_cols);
```

- ⇨ breidt de matrix dynamisch uit (de matrix kan enkel groter worden, veronderstel dat `new_num_rows` >= huidige aantal rijen en `new_num_cols` >= huidig aantal kolommen). Het resultaat (dat na uitvoeren van de methode in `m` terecht moet komen) is een matrix waarbij alle elementen van de oorspronkelijke matrix in de linkerbovenhoek staan. De extra rijen en/of kolommen worden opgevuld met nullen. Let op: het is niet toegestaan van een kopie van de matrix te maken, je moet de bestaande rijen heralloceren.

Als alle functies zijn geïmplementeerd, kan je deze testen door de `main`-functie uit te voeren. Deze voert volgende matrix bewerkingen uit:

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 3 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 3 \\ 2 & 1 \\ 0 & 4 \end{pmatrix}$$

$$C = A \times B$$

$$C = C^T$$

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$B = C \times A$$

$$B = \begin{pmatrix} (B) & \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} \\ \begin{matrix} 0 & 0 \end{matrix} & \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} \end{pmatrix}$$

3. Image processing

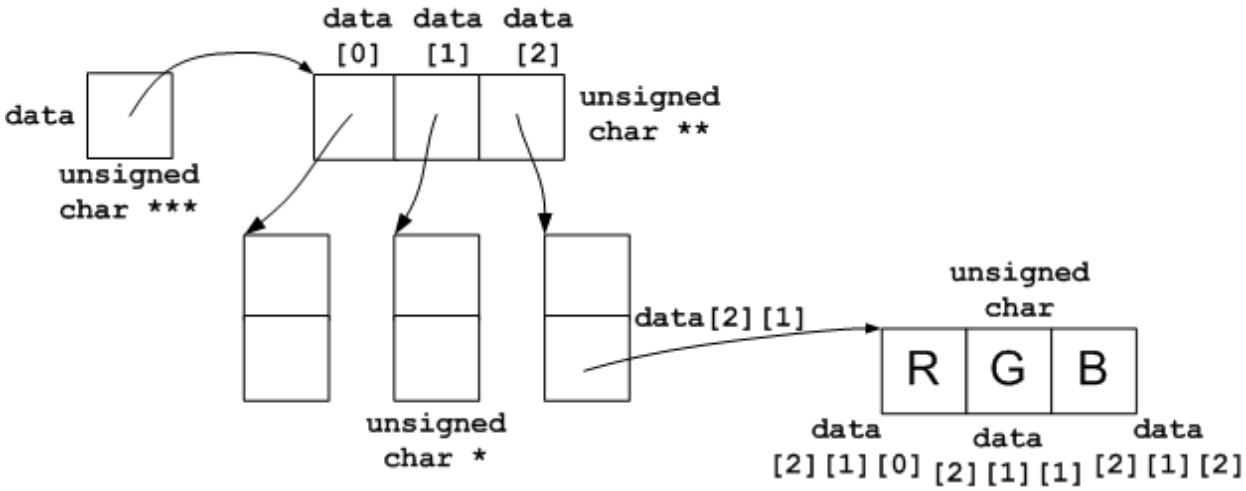
Bitmap-afbeeldingen zijn een 2-dimensionaal raster van punten (of pixels). Per pixel worden 3 kleurcomponenten bijgehouden, de zogenaamde RGB waarden (RGB staat voor rood, groen en blauw). Een R, G of B waarde is voor 24-bits kleuren een byte groot (dus een getal tussen 0 en 255). In deze oefening zullen we afbeeldingen inladen in een 3-dimensionale rij (denk ruimtelijk: drie 2-dimensionale matrices na elkaar, eentje per kleurcomponent R, G en B) en manipuleren. In `image.h` vind je de volgende `struct`:

```
typedef struct {
      int width;
      int height;
      unsigned char*** data;
} image;
```

Deze houdt de breedte en hoogte van de afbeelding bij. Merk op dat de grootte van de derde dimensie niet hoeft bijgehouden te worden, gezien deze steeds 3 is. Als datatype voor de kleurcomponenten werd gekozen voor unsigned char omdat die exact 1 byte groot zijn, en dus 256 waarden in het interval [0-255] kunnen bevatten.

Figuur 2 toont de geheugenstructuur van het data element van zo'n afbeelding voor breedte 3 en hoogte 2.

Let op: bij afbeeldingen duidt de eerste coordinaat naar de kolom en de tweede naar de rij (dus omgekeerd in vergelijking met matrices). Dus `data[x][y]` wijst naar de R,G en B waarden van de pixel op de `x`e kolom en `y`e rij.
Het element `data` in de struct image, van het type `unsigned char***`, wijst naar een rij van type `unsigned char**` met lengte `width`. De elementen in die rij (`data[x]`) wijzen op hun beurt naar een rij van het type `unsigned char*` met lengte `height`. De elementen in deze rijen (`data[x][y]`) wijzen op hun beurt naar een RGB rij met lengte 3. Voor de eenvoud staat in de figuur maar 1 van de 6 RGB rijen afgebeeld.



Figuur 2 - geheugenstructuur van een image

Implementeer de volgende functies in `image.c`:

```
void init_image(image* im, int width, int height);
```
⇨ initialiseert een image met de opgegeven breedte en hoogte, reserveert geheugen voor de kleurcomponenten.

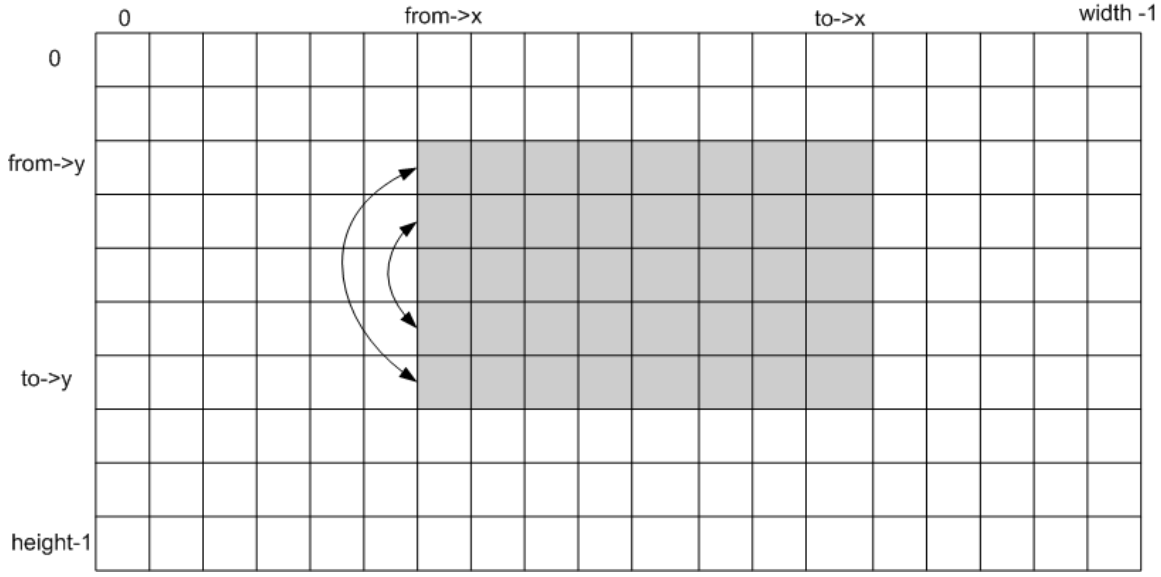
```
void init_image_default(image* im);
```
⇨ doet hetzelfde als de vorig functie, maar gebruikt de default variabelen in `image.h` als breedte en hoogte.

```
void free_image(image* im);
```
⇨ geeft het geheugen vrij van de afbeelding

```
void negative_image(image* im, pixel* from, pixel* to);
```
⇨ inverteert alle kleurcomponenten binnen de opgegeven rechthoek (inclusief de grenzen!) van de afbeelding. Bekijk hiervoor de `struct pixel` in `image.h`. De `pixel* from` geeft de linkerbovenhoek aan van de rechthoek, de `pixel* to` de rechteronderhoek. Het inverse van een kleurcomponent X is 255 – X. Zo wordt bijvoorbeeld zwart (0,0,0) omgezet naar wit (255,255,255).

```
void flip_image_vertically(image* im, pixel* from, pixel* to);
```

⇨ spiegelt de afbeelding verticaal binnen een bepaalde rechthoek (inclusief de grenzen!) van de afbeelding. Figuur 3 - verticaal spiegelen van een rechthoek in een afbeelding toont grafisch op pixelniveau wat er gebeurt tijdens deze operatie. Enkel de pixels in het grijs (binnen de rechthoek) veranderen. Alle lijnen worden gespiegeld om een horizontale as in het midden van de rechthoek. In het voorbeeld zal de middelste lijn ongewijzigd blijven omdat de hoogte van de rechthoek (`to->y - from->y`) oneven is. Voor een even hoogte zullen geen pixels ongewijzigd blijven.



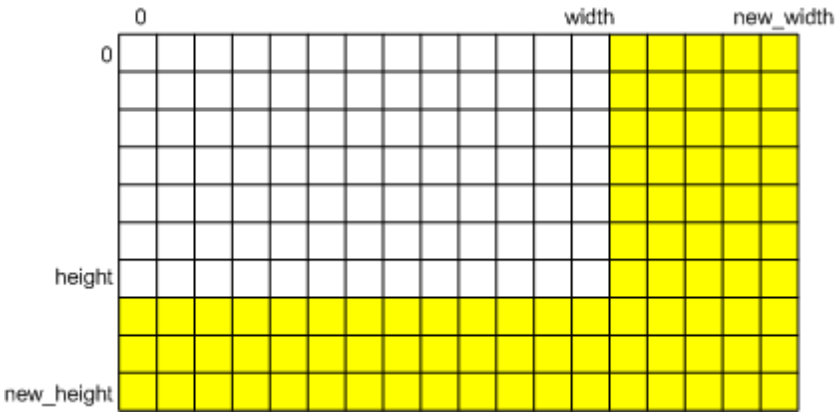
Figuur 3 - verticaal spiegelen van een rechthoek in een afbeelding

```
void flip_image_horizontally(image* im, pixel* from, pixel* to);
```

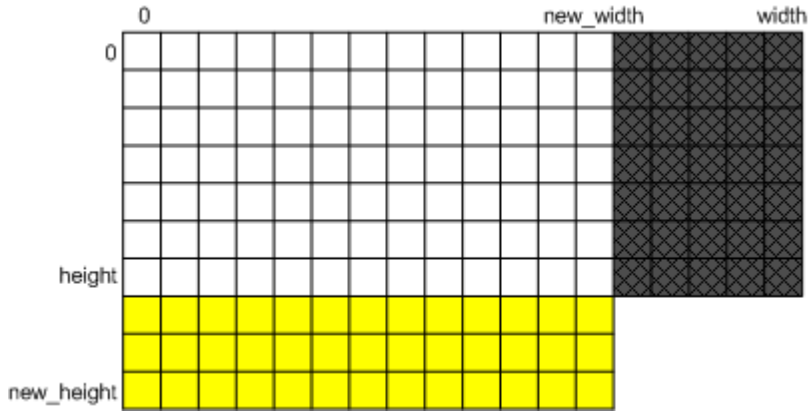
⇨ spiegelt de afbeelding horizontaal binnen een bepaalde rechthoek (inclusief de grenzen!) van de afbeelding. Analoog aan de vorige functie, maar nu wordt gespiegeld om een verticale as.

```
void dynamic_resize_image (image* im, int new_width, int new_height,
unsigned char r, unsigned char g, unsigned char b);
```

⇨ verandert de grootte van de afbeelding dynamisch. Hierbij geven `new_width` en `new_height` de nieuwe grootte van de afbeelding aan en geven `r`, `g` en `b` de kleurwaarden van de nieuwe pixels aan, indien deze er zijn (in het andere geval worden pixels weggesneden en verlies je dus kleurdata). Let hierbij op dat (integenstelling tot `dynamic_expand` bij de matrices), de breedte en hoogte zowel kunnen groter als kleiner worden (er zijn dus 4 mogelijke situaties). Vermijdt het kopiëren van de kleurdata. Figuur 4 toont een voorbeeld waarbij de breedte en hoogte beiden groter worden, Figuur 5 toont een voorbeeld waarbij de breedte kleiner en de hoogte groter wordt.



Figuur 4 - dynamisch de grootte van een afbeelding veranderen – voorbeeld 1



Figuur 5 - dynamisch de grootte van een afbeelding veranderen – voorbeeld 2

De functies voor het inladen en opslaan van afbeeldingen uit een bestand zijn reeds aanwezig, hier mag niets aan veranderd worden:

```
void load_image(image* im, char* file_name);
void save_image(image* im, char* file_name);
```

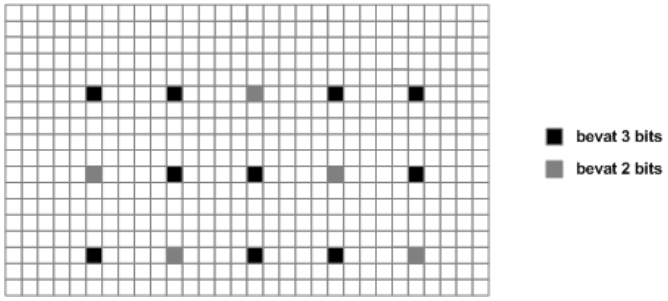
Als alle functies geïmplementeerd zijn, kan je deze testen door de `main`-methode uit te voeren. Deze zal een afbeelding (`input.ppm`) inlezen, enkele vooraf gedefinieerde bewerkingen hier op uitvoeren en het resultaat opslaan in `output.ppm`. Zorg dat de `input.ppm` afbeelding bij je bronbestanden staat! Het resultaat wordt ook op die locatie opgeslagen. Om te testen, mag je eventueel (tijdelijk) enkele `save_image` oproepen aan je main functie toevoegen om je tussentijdse resultaten te kunnen bekijken.

Tijdens het practicum gebruiken we het `.ppm` bestandsformaat. Windows heeft geen standaard viewer voor dit formaat. Om het resultaat te bekijken hebben we een extra tool nodig: IrfanViewer. Deze kun je hier downloaden: http://users.atlantis.ugent.be/dvrslype/iview430_setup.exe

Voor mac gebruikers bestaat er ToyViewer of Xee. Ubuntu ondersteunt `.ppm` out of the box.

4. Steganografie

Steganografie is de wetenschap van het verbergen van boodschappen in afbeeldingen. Onder andere al-Qaeda gebruikte deze technieken om berichten onder hun leden te verspreiden, zonder dat deze door de CIA konden onderschept worden. Er bestaan veel methodes om dit aan te pakken, wij kiezen voor een van de simpelste varianten:
We plaatsen een rooster van 30 bij 30 pixels over de afbeelding. Enkel op de kruispunten van dit rooster passen we de laatste bit (de least significant bit genoemd) van de kleurcomponenten aan. We gebruiken de laatste bit, omdat die minst invloed heeft op het uiteindelijk kleur. We proberen dus verstoring van het visuele beeld te vermijden. Op die manier kunnen we 3 bits van onze boodschap verbergen per pixel op het rooster. Gezien een `unsigned char` 1 byte aan opslag nodig heeft, zullen we elk karakter verdelen over 3 pixels. De eerste 3 bits in de eerste pixel, bit 4 tot en met 6 in de tweede pixel en de laatste 2 bits in de derde pixel. In Figuur 6 zie je een voorbeeld van zo'n rooster (in dit geval van 5x5 voor de eenvoud, in het practicum gebruiken we 30x30). De donkere pixels bevatten 3 bits van de boodschap (in zowel de R, de G als de B kleurcomponent). De lichtere pixels op het rooster bevatten slechts 2 bits van de boodschap (enkel in de R en G kleurcomponent). Rekening houdend met het feit dat we elke versleutelde boodschap laten afsluiten met de 0x00 – byte (zoals bij strings), kunnen dus in totaal 4 karakters in deze afbeelding verborgen worden.



Figuur 6 - bits verborgen in image

Bereken eens hoeveel karakters (rekening houdend met het 30x30 rooster en de stop-byte) kunnen verborgen worden in de opgave figuur. Bereken dit ook eens voor het geval we elke pixel gebruiken (dus zonder rooster).

- Vul de functie `decode_message` aan, je hoeft slechts 8 regels code toe voegen, namelijk oproepen naar de functie `extractLeastSignificantBit`. Deze functie neemt de laatste bit van een `unsigned char`, shift de bits eventueel naar rechts en voegt de bit toe aan het samen te stellen karakter.
- Implementeer de functie `encode_message` volledig zelf. Maak gebruik van de functie `writeLeastSignificantBit`. Deze functie neemt van een `unsigned char` een bepaalde bit (aangegeven door bitIndex), shift deze bit helemaal naar rechts (zodat we 0x01 of 0x00 uitkomen) en verbergt deze bit in een kleurcomponent.
- Vul in de main methode, je voornaam en naam in in de `encode_message` call. Als alles goed is gegaan zouden we je naam moeten kunnen decoderen, dus dien naast je source code ook je output afbeelding in.
- Meer info over binaire operaties en bitshifts in de slides bij de cursus H7 en sectie 7.7.

5. Indienen

- Zip je broncode bestanden (`matrix.c en image.c`) en je output image (`output.ppm`) in een bestand `naam_voornaam_3.zip` bvb. `volckaert_bruno_3.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 aanbrengen van veranderingen hieraan is NIET toegestaan. Doe je dit toch, dan zal dat zich reflecteren in je score.
- Stuur je oplossing door middel van de dropbox op Minerva door naar Bruno Volckaert, ten laatste op **zondag 28 oktober 2012, 23u59**.

Veel succes!