# CPLUS Eindopdracht 2021 – 2022
# Speuren met Krul

Bob Polis, Avans Hogeschool, 's-Hertogenbosch

27 september 2021

## Inhoudsopgave



## 1 Wat ga je bouwen?

Ergens op een geheime plek op het web staat een bericht dat je moet zien te vinden. Je krijgt een start-URL, waar je via een HTTP GET-request een stukje platte tekst

van ophaalt.

Deze tekst is *source code*, geschreven in een door mij verzonnen mini-taal, *Krul* genaamd, voor een door jou te bouwen *interpreter*.

Als je de code runt geeft dat een *string* als resultaat. Die string is een bestandsnaam waarmee je opnieuw een URL samenstelt, waarvandaan je een volgend stukje Krul-code kunt downloaden.

Zo ga je door, totdat je een speciale instructie in de code tegenkomt die aangeeft dat je de oplossing gevonden hebt. De laatst samengestelde string is dan het geheime bericht.

Je bouwt hiervoor een command-line tool (*console application*) in C++, die de uiteindelijke oplossing in de terminal rapporteert.

Ter controle laat je in je output steeds elk resultaat van een stukje code ook zien in de terminal, zodat je dus uiteindelijk een opsomming ziet van bestandsnamen, gevolgd door het gevonden geheime bericht.

## 2 Requirements

1. Je maakt een command-line tool (*console application*) in C++.

2. Je haalt steeds platte tekst binnen vanaf een URL die als basis
   `https://www.swiftcoder.nl/cplus/`
   heeft.

3. De eerste complete URL waar je een mini-programma van ophaalt wordt gevormd door de bestandsnaam "`start.txt`" achter de basis-URL te zetten. Dat wordt dus:
   `https://www.swiftcoder.nl/cplus/start.txt`

4. Je bouwt een interpreter voor de in dit document beschreven taal.

5. Je interpreteert het ontvangen mini-programma, wat leidt tot een string als resultaat.

6. De gevonden string is de volgende bestandsnaam die je achter de basis-URL plaatst om een nieuwe URL te krijgen. Krijg je als resultaat uit het programma de tekst "`hoera.txt`", dan wordt de volgende URL waar je code vandaan haalt dus:
   `https://www.swiftcoder.nl/cplus/hoera.txt`

7. Rapporteer de gevonden URL ter controle in de terminal.

8. Zo ga je door met programma-tekst ophalen, todat je in een programma de "`end`" instructie tegenkomt. In dat geval is de resulterende string de oplossing van deze speurtocht.

9. De enige externe library die is toegestaan is `libcurl`.[1]

## 3 Beoordeling

De beoordeling vindt plaats volgens de in de course gepubliceerde toetsmatrijs. Je maakt de opdracht individueel, en levert je werk op tijd in. Vervolgens word je opgeroepen voor een assessment in lesweek 8.

## 4 HTTP GET-requests met `libcurl`

Je gebruikt voor het doen van HTTP GET-requests een immens populaire library, `libcurl`[2] genaamd, die voor elk platform te krijgen is. Die library stelt je in staat om allerlei soorten requests te doen, zodat je bijvoorbeeld gemakkelijk webpagina's en data van API's kunt ophalen.

`libcurl` is een C-library, dus je zult in elk geval een C++-class moeten schrijven die de verbinding met jouw C++-code in goede banen leidt.

## 5 De interpreter voor Krul

De interpreter die je gaat schrijven is niet ingewikkeld, maar er zitten wel een paar leuke uitdagingen in om hem netjes geïmplementeerd te krijgen.

Krul maakt gebruik van *reversed polish notation (RPN)*[3].

Bijvoorbeeld, de volgende code:

```
2
3
add
4
mul
```

levert "`20`" als resultaat.

[1] De C++ Standard Library mag natuurlijk wel: die beschouwen we niet als *externe* library.

[2] https://curl.haxx.se/libcurl/

[3] https://en.wikipedia.org/wiki/Reverse_Polish_notation

In je implementatie gebruik je een stack om waarden op te zetten en vanaf te halen. Het is handig om daar een `std::vector` voor te gebruiken.

Bovenstaand voorbeeld is dan gemakkelijk te verwerken: je zet de waarde `2` op de stack, dan de waarde `3`, en vervolgens roep je een functie aan om de "`add`"-instructie te interpreteren. Die haalt de twee waarden van de stack, telt ze op, en zet het resultaat "`5`" op de stack.

Daarna wordt `4` op de stack gezet, en wordt de instructie "`mul`" uitgevoerd. Die haalt twee waarden van de stack (`5` en `4` in dit geval), vermenigvuldigt ze, en zet het resultaat "`20`" op de stack.

*Let op!* Als ik schrijf "haal twee waarden van de stack", dan bedoel ik dat ze gelezen *en verwijderd* worden van de stack. Als later het resultaat van een berekening op de stack wordt gezet, vervangt dat resultaat dus in feite de twee waarden waarmee we begonnen.

*Let op!* Als je een waarde van een stack haalt, is dat de laatste die op de stack is gezet. Een stack werkt volgens het "LIFO"-pricipe: *last in, first out*.

## 5.1 Ontwerptip

Je zou de interpreter kunnen beschouwen als een soort CPU. Dat wil zeggen, als een "machientje" dat instructies verwerkt, en daarvoor een paar hulpmiddelen heeft.

Het zal handig blijken wanneer je met een integer (bijvoorbeeld een `size_t`) bijhoudt van welke regel de volgende instructie zal worden gelezen. Een echte CPU heeft daarvoor een register dat *program counter* wordt genoemd, en dat steeds naar de volgende geheugenlocatie wijst waarvandaan de volgende instructie moet worden gelezen. Dat is een simpel mechanisme, dat het ook makkelijk maakt om naar een ander stuk code te "springen". Je kent dan gewoon de locatie van de volgende instructie toe aan de program counter. De rest gaat vanzelf!

Verder heeft jouw interpreter een stack nodig voor de waarden waarmee hij rekent, zoals in bovenstaand voorbeeld al werd getoond.

Enkele andere hulpmiddelen volgen uit de beschrijving hieronder.

## 5.2 Gedetailleerde beschrijving van Krul

*Alle waarden op de stack zijn strings.*

Dat betekent dat numerieke berekeningen altijd eerst de benodigde waarden naar integer converteren voordat ermee gerekend kan worden. Ook moet een numeriek resultaat naar string geconverteerd worden alvorens het op de stack wordt gezet.

Na de laatste regel code, als er geen verdere instructies meer zijn, stopt het programma vanzelf. Als de stack niet leeg is, is de laatste waarde op de stack het eindresultaat van dit Krul-programma.

### 5.2.1 Values & Types

**`42`** *Digits: integer literal.* Zet het gegeven integer getal, geconverteerd naar string, op de stack. Er zijn alleen positieve integers of de waarde `0` toegestaan. (Voor negatieve getallen wordt een positief getal gegeven, gevolgd door een "`neg`"-instructie.) Het getal `42` is hier slechts een voorbeeld.

**`\text`** *Text to end of line.* Beschouw alle tekst na de backslash tot het einde van de regel als een letterlijke string, en zet die op de stack.

**`:label`** *Label definition.* Beschouw de tekst na de dubbele punt tot het einde van de regel als een label-naam, en onthoud die, samen met de volgende regelpositie, zodat je later naar die regel kunt springen.

**`>label`** *Label reference.* Zet de *programmaregel* die door het label wordt gerepresenteerd op de stack. Let op: je kunt zo'n *label reference* tegenkomen terwijl je de bijbehorende *label definition* nog niet hebt gezien! Dat maakt "vooruit springen" mogelijk.

**`=var`** *Variable assignment.* Beschouw alle tekst na het "`=`"-teken tot het einde van de regel als de naam van een variabele, waaraan je de huidige waarde op de stack toekent. Verwijder vervolgens de waarde van de stack.

**`$var`** *Variable reference.* Zet de waarde van de variabele op de stack.

Elk andere letterlijke tekst wordt opgevat als een instructie voor de machine. Alle mogelijke machine-instructies staan hieronder.

### 5.2.2 Integer operaties

**`add`** *Optellen.* Haal 2 waarden van de stack, converteer naar integer, tel ze op, converteer naar string, en zet het resultaat weer op de stack.

**`sub`** *Aftrekken.* Haal 2 waarden van de stack, converteer naar integer, trek de tweede van de eerste af, converteer het resultaat naar string, zet het op de stack.

Bijvoorbeeld, de volgende code:

```
5
3
sub
```

Levert `2` als uitkomst, en dat wordt (als string) op de stack gezet.

**mul** *Vermenigvuldigen.* Haal 2 waarden van de stack, converteer naar integer, vermenigvuldig ze, converteer naar string, en zet het resultaat op de stack.

**div** *Delen met afkappen.* Haal 2 waarden van de stack, converteer naar integer, deel de tweede door de eerste, converteer het resultaat naar string en zet dit op de stack.

Bijvoorbeeld, de volgende code:

```
8
2
div
```

levert `4` als uitkomst, en dat wordt (als string) op de stack gezet.

**mod** *Modulo.* Haal 2 waarden van de stack, converteer naar integer, bereken `tweede % eerste`, converteer het resultaat naar string en zet dat op de stack.

Bijvoorbeeld, de volgende code:

```
7
3
mod
```

levert `1` als resultaat, en dat wordt (als string) op de stack gezet.

**neg** *Negate.* Haal één waarde van de stack, converteer naar integer, keer het teken om (van plus naar min of omgekeerd), converteer het resultaat naar string en zet op de stack.

**abs** *Absolute waarde.* Haal één waarde van de stack, converteer naar integer, bereken de absolute waarde, converteer het resultaat naar string en zet op de stack.

**inc** *Increment.* Haal één waarde van de stack, converteer naar integer, tel er één (`1`) bij op, converteer naar string en zet op de stack.

**dec** *Decrement.* Haal één waarde van de stack, converteer naar integer, trek er één (`1`) van af, converteer naar string en zet op de stack.

#### 5.2.3 String operaties

**dup** *Duplicate.* Lees één waarde van de stack (haal hem er *niet* af), en zet deze nogmaals op de stack.

**rev** *Reverse.* Haal één waarde van de stack, draai de string om, en zet het resultaat op de stack.

**slc** *Substring.* Haal 3 waarden van de stack, achtereenvolgens *to*, *from*, en de *waarde* waarvan we een substring willen nemen. De *from*- en *to*-waarden worden naar integer geconverteerd, zodat je ze op de juiste wijze kunt toepassen in de `substr()`-method van `std::string`. De *to*-waarde is de string-index die nèt niet meer meedoet. Het is dus van-tot, niet van-tot-en-met.

*Let op*: dit is niet helemaal hetzelfde als wat `substr()` verwacht.

Bijvoorbeeld, de volgende code:

```
\abcdefghijklm
3
6
slc
```

levert "`def`" als resultaat.

**idx** *Indexeren.* Haal 2 waarden van de stack: eerst een index die je naar integer converteert, dan de string-waarde waaruit een character wordt genomen. Zet het correcte character als string op de stack.

Dus:

```
\tekst
2
idx
```

levert "k" als resultaat.

**cat** *Concatenation.* Haal 2 waarden van de stack, bouw een string door de tweede achter de eerste te plaatsen en zet het resultaat op de stack.

Dus:

```
\hello
\ world
cat
```

levert "`hello world`" als resultaat.

**`len`** *Lengte.* Haal één waarde van de stack, bepaal de length van deze string, converteer die naar string en zet op de stack.

**`rot`** *Roteren.* Haal één waarde van de stack, voer er een rot-13[4] transformatie op uit en zet het resultaat op de stack.

**`enl`** *Add a newline.* Haal één waarde van de stack, voeg daar een newline (`'\n'`) aan toe, en zet het resultaat op de stack.

#### 5.2.4 Tests & Jumps

**`gto`** *Goto.* Haal één waarde van de stack, interpreteer die als programmaregel, en zorg ervoor dat de volgende instructie vanaf die regel zal worden gelezen.

Dus:

```
>main
gto
```

vervolgt het interpreteren van instructies vanaf de regel die met het label `:main` wordt aangeduid (elders in de code).

*Let op*: het kan best zijn dat `:main` *verderop* in de code staat. Je moet labeldefinities dus al ergens op een rijtje hebben staan voordat je een programma gaat interpreteren.

Dit betekent dat het handig is als je een *two-pass* interpreter maakt. Eén keer door de code voor alle labels, daarna code uitvoeren.

**`geq`** *Goto if equal.* Haal 3 waarden van de stack, achtereenvolgens: *label-waarde*, *val2* en *val1*. Als *val1* en *val2* gelijk zijn, vervolg het interpreteren van instructies dan vanaf de locatie die gerepresenteerd wordt door de *label-waarde*. Zo niet, ga dan gewoon door met de volgende instructie.

Dus:

```
\hoi
\daar
>groet
geq
```

[4] https://en.wikipedia.org/wiki/ROT13

De strings “hoi” en “daar” zijn niet gelijk, dus de volgende instructie wordt niet uitgevoerd vanaf de plek direct na `:groet`, maar gewoon direct na de `geq`-instructie.

Let op: deze instructie doet een string-vergelijking!

**`gne`** *Goto if not equal.* Haal 3 waarden van de stack, achtereenvolgens: *label-waarde*, *val2* en *val1*. Als *val1* en *val2* ongelijk zijn, vervolg het interpreteren van instructies dan vanaf de locatie die gerepresenteerd wordt door de *label-waarde*. Zo niet, ga dan gewoon door met de volgende instructie.

Dus:

```
\hoi
\daar
>groet
gne
```

zal ervoor zorgen dat de volgende instructie vanaf de regel direct na `:groet` zal worden uitgevoerd, want de strings zijn niet gelijk.

Let op: deze instructie doet een string-vergelijking!

**`glt`** *Goto if less.* Haal 3 waarden van de stack, achtereenvolgens: *label-waarde*, *val2* en *val1*. Converteer *val1* en *val2* naar integer. Als *val1* kleiner is dan *val2*, vervolg het interpreteren van instructies dan vanaf de locatie die gerepresenteerd wordt door de *label-waarde*. Zo niet, ga dan gewoon door met de volgende instructie.

Dus:

```
2
3
>kleiner
glt
```

`2` is kleiner dan `3`, dus inderdaad zal de volgende instructie worden gelezen vanaf de regel direct na `:kleiner`.

Let op: deze instructie vergelijkt integers!

**`gle`** *Goto if less or equal.* Haal 3 waarden van de stack, achtereenvolgens: *label-waarde*, *val2* en *val1*. Converteer *val1* en *val2* naar integer. Als *val1* kleiner of gelijk is aan *val2*, vervolg het interpreteren van instructies dan vanaf de

locatie die gerepresenteerd wordt door de *label-waarde*. Zo niet, ga dan gewoon door met de volgende instructie.

Dus:

```
42
42
>kleiner-of-gelijk
gle
```

De twee integers zijn gelijk, dus de sprong wordt uitgevoerd. We gaan niet verder op de regel na `gle`, maar vanaf de regel die door `:kleiner-of-gelijk` wordt aangeduid.

Let op: deze instructie vergelijkt integers!

**`ggt`** *Goto if greater.* Haal 3 waarden van de stack, achtereenvolgens: *label-waarde*, *val2* en *val1*. Converteer *val1* en *val2* naar integer. Als *val1* groter is dan *val2*, vervolg het interpreteren van instructies dan vanaf de locatie die gerepresenteerd wordt door de *label-waarde*. Zo niet, ga dan gewoon door met de volgende instructie.

Dus:

```
10
37
>groter
ggt
```

`10` is niet groter dan `37`, dus in dit voorbeeld gaan we niet verder vanaf de regel na `:groter`, maar vanaf de regel direct na de `ggt`-instructie.

Let op: deze instructie vergelijkt integers!

**`gge`** *Goto if greater or equal.* Haal 3 waarden van de stack, achtereenvolgens: *label-waarde*, *val2* en *val1*. Converteer *val1* en *val2* naar integer. Als *val1* groter of gelijk is aan *val2*, vervolg het interpreteren van instructies dan vanaf de locatie die gerepresenteerd wordt door de *label-waarde*. Zo niet, ga dan gewoon door met de volgende instructie.

Dus:

```
92
56
>groter-of-gelijk
gge
```

In dit voorbeeld is `92` groter dan `56`, dus wordt de sprong uitgevoerd: we gaan verder vanaf de regel direct na `:groter-of-gelijk`.

Let op: deze instructie vergelijkt integers!

#### 5.2.5 Functies

**`fun`** *Functie.* Bewaar de locatie van de volgende instructie op een aparte stack (de *call stack*), en voer een `gto` uit.

**`ret`** *Return.* Haal de laatste locatie van de *call stack*, en zorg dat de volgende instructie vanaf die locatie wordt uitgevoerd.

#### 5.2.6 Eindoplossing

**`end`** *End of search.* Waarde op de stack is het uiteindelijke geheime bericht. Deze instructie komt alleen in de laatste file die je ophaalt voor.

### 5.3 Langer voorbeeld

```
3               zet 3 op de stack
=cnt            haal 3 van de stack en ken toe aan cnt
\Hello, world   zet "Hello, world" op de stack
enl             voeg daar een newline aan toe
=hello          ken het resultaat toe aan hello
\               zet een lege string op de stack
=result         ken die toe aan result
:loop           definieer de volgende stap als locatie loop
$result         zet result op de stack
$hello          zet hello op de stack
cat             maak er één string van
=result         ken die toe aan result
$cnt            zet cnt op de stack
dec             trek daar 1 van af
=cnt            ken dat weer toe aan cnt
$cnt            zet cnt op de stack
0               zet 0 op de stack
>loop           zet een verwijzing naar locatie loop op de stack
ggt             als cnt > 0, ga naar loop, anders verder
$result         zet result op de stack — dit is het eindresultaat
```

Het eindresultaat is een string die uit 3 regels tekst bestaat:

```
Hello, world
Hello, world
Hello, world
```

## 6 Tips

- Laat je niet afschrikken door de lijst *instructies* die je moet implementeren, dat is minder werk dan je denkt. Elke instructie is steeds een korte functie met een paar regels recht-toe-recht-aan code.
- In vervolg op de eerste tip: velen van jullie zijn geneigd om een Command design pattern toe te passen. Dat is natuurlijk prima, maar levert in dit geval wel onnodig veel werk op. Dit is C++, geen design pattern vak! Houd het simpel: maak gewoon een interpreter class waar alles in zit.
- Een stack kun je makkelijk met een `std::vector`[5] implementeren. Gebruik de methods `push_back()` en `pop_back()` om objecten op de stack te zetten of er van af te halen. Gebruik `back()` om de laatste waarde op de stack te lezen zonder die er van af te halen.
- Voeg eventueel een "**`out`**" instructie aan de interpreter toe, die de laatste waarde op de stack naar `std::cout` schrijft. Dat is handig voor testen en debuggen.
- Houd de verbinding open totdat je klaar bent met alle requests. Dat maakt je programma aanzienlijk sneller dan wanneer je per request een verbinding maakt, request uitvoert, en weer afsluit.
- Als je wat makkelijker je eigen interpreter wilt kunnen testen met lokale Krul-scripts, maak hem dan geschikt voor het lezen uit lokale files.
- Sta commentaar toe, door alle tekst vanaf "`#`" tot het einde van de regel te interpreteren als commentaar. POSIX-gebruikers kunnen daardoor een *she-bang line* boven in hun scripts zetten:

  ```
  #!/usr/bin/env krul
  ```

  Als je dan een test-script ook nog executable maakt, kun je het zonder toevoegingen lokaal uitvoeren.
- Als je tenslotte nog een "**`err`**" en "**`in`**" instructie toevoegt kun je volwaardige commandline scripts maken. Dit is handig voor een test-suite van unit tests!

[5] https://en.cppreference.com/w/cpp/container/vector