Makefile

Rakenduse ehitamise süsteemid

Erinevad lahendused:

- GNU Make - Laialdaselt kasutusel UNIX põhistel süsteemidel, vaikimisi toetatud pea kõikjal
- CMake - Fookuses OS-vaheline tugi, laialdaselt kasutusel IDE-de köögipoolel
- Meson - Fookuses lihtsus
- …

Lihtsustab rakenduse ehitamist

- Tavaliselt tehtav ühe käsuga - nt `make`, `cmake --build`, `meson compile`

Soovituslik kui sul

- On rohkem kui üks .c koodifail
- Kood jagatud kaustadesse
- Kasutad kolmanda osapoole teeke
- Kasutad mitmest käsust koosnevat kompileerimist või täiendavaid lippe kompileerimiseks

GNU Make ja Makefile

Klassikaline ehitussüsteem

Laialdane levik, fookus UNIX-põhistel süsteemidel, kuid töötab ka nt Windowsis, macOSis

- Kasutusel ka teiste ehitussüsteemide poolt - nt CMake ise ei ehita, vaid loob Makefile'i ehitamiseks

Makefile sisaldab endas retsepte (*recipe*) tarkvaratoote ehitamiseks

Retseptide sisuks on tavalised *shell-i* käsud

- ehk siis kõiksugu käsud mida võid ka niisama käsurealt käivitada
- seetõttu kasutatav ka väljaspool rakenduse kompileerimist

Võimaldab kompileerida vaid muudetud sisuga faile

- Kontrollib, millal viimati muudeti faile, millest retsept sõltub
- Kompileerib vaid need failid, mida on muudetud või retseptid, mis sõltuvad muudetud failidest

GNU Make kättesaadavus

Laboriarvutites

- Paigaldatud

Sinu enda Linux

- Tuli koos `build-essential` pakiga, mille paigaldasid keskkonda üles seades
- Eraldi paigaldatav (Nt Debiani-põhisel distributsioonil: `apt install make`)

Windows

- Saadaval pakina `GNU Make`
- Paigaldatav näiteks paketihaldurist Chocolatey (`choco install make`)

MacOS

- Tuleb koos arendustööriistadega (Developer tools, nt paigaldades xCode)

Retsept ja retsepti ehitamine

Retsepti struktuur

- Retsepti nimi, millele järgnevad failid ja teised retseptid, millest see sõltub
- Kõigil retsepti käskudel on üks *tab ees*

```
recipe_name: dependencies
[tab] action line(s)
```

Retseptid kirjutatakse faili nimega `Makefile`

Retsepti ehitamiseks kasutame käsku `make`, mis käivitatakse samast kaustast kus on `Makefile`

Ilma argumendita käivitatakse failist esimene retsept

Retsept nimega `all` kirjeldab tavaliselt seda kuidas ehitada nullist kogu tarkvaratoode

Käivitamaks ükskõik millist teist retsepti kirjutame `make recipe_name`

Näide (1)

```
all:
    gcc -o program -Wall -Wextra source.c
```

Retsepti käskudele peab eelneva *tab*

```
risto@risto-lt:~$ make
gcc -o program -Wall -Wextra source.c
```

Selle retsepti käivitamiseks saame kirjutada kas `make` või `make all`

Vaikimisi trükitakse terminali kõik read, mis käivitati

Näide (2)

```
all:
    gcc -o program -Wall -Wextra source.c

chatty:
    @echo Starting build
    gcc -o program -Wall -Wextra source.c
    @echo Finished
```

Ühes failis võib olla mitu retsepti

Retseptil võib olla mitu rida

```
risto@risto-lt:~$ make
    gcc -o program -Wall -Wextra source.c

risto@risto-lt:~$ make chatty
    Starting build
    gcc -o program -Wall -Wextra source.c
    Finished
```

Vaikimisi käivitatakse esimene retsept failist. Teisi retsepte saab käivitada kutsudes neid nimepidi.

`@` lülitab välja käsu ekraanile trükkimise. `echo` on samas käsk teksti ekraanile trükkimiseks

NB! Sõltuvusena võib olla vajalik lisada ka päisefaile!

Näide (3)

```
all: source1.o source2.o
    gcc -o program source1.o source2.o

source1.o: source1.c
    gcc -c source1.c

source2.o: source2.c
    gcc -c source2.c
```

Esimene retsept sõltub teistest retseptidest

Need retseptid sõltuvad koodifailidest. Need kompileeritakse uuesti vaid pärast sõltuvate failide uuendamist

```
risto@risto-lt:~$ make
gcc -c source1.c
gcc -c source2.c
gcc -o program source1.o source2.o
risto@risto-lt:~$ make
gcc -o program source1.o source2.o
```

Muutujad

Muutuja deklareerimiseks anna neile nimi ja väärtus

```
CC = gcc

CFLAGS = -Wall -Wextra -Wconversion
```

Muutuja kasutamiseks kirjuta `$(VARNAME)`. Nt: `$(CFLAGS)`

Eksisteerivad erinimelised muutujad, mis langevad kokku vaikimisi reeglitega (***implicit rules***). Neid reegleid kasutatakse kui faili kompileerimiseks **retsept puudub**

https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html#Implicit-Variables

Näide (4)

```
CC = gcc
CFLAGS = -Wall -Wextra –Wconversion
BIN = my_program

all: source1.o source2.o
  $(CC) –o $(BIN) source1.o source2.o
```

source1.c ja source2.c kompileerimiseks puuduvad retseptid. Küll aga source1.o ja source2.o on märgitud sõltuvustena. Nende kompileerimiseks kasutatakse vaikimisi reegleid muutujatest CC ja CFLAGS.

```
risto@risto-lt:~$ make
gcc -Wall -Wextra -Wconversion   -c -o source1.o source1.c
gcc -Wall -Wextra -Wconversion   -c -o source2.o source2.c
gcc –o my_program source1.o source2.o
```

Märka, et lippe ei kasutatud `all` retseptiga kompileerimisel! Lippe ei lisatud kuna selle kompileerimiseks oli käsitsi kirjutatud retsept.

Näide (5)

Retsepti `clean` kasutatakse enamasti ajutiste failide puhastamiseks, mis tekivad kompileerimise käigus. Mõnel juhul likvideerib see ka programmi enda. Sageli võib puhastamiseks olla mitu retsepti.

```
CC = gcc
CFLAGS = -Wall -Wextra -Wconversion
BIN = my_program
all: source1.o source2.o
  $(CC) -o $(BIN) source1.o source2.o

.PHONY: clean
clean:
  rm *.o $(BIN)
```

.PHONY on märksõna mida kasutatakse retseptides, mis ei loo uusi faile. See väldib konflikte olukorras, kus mõnel failil on sama nimi.

```
risto@risto-lt:~$ make clean
rm *.o my_program
```

Näide eelmisest nädalast

```
CC = gcc-12
CFLAGS = -g -Wall -Wextra -Wconversion
LDFLAGS = -lcurl
BIN = covid_stats

all: data_processor.o file_helper.o main.o
    $(CC) -o $(BIN) data_processor.o file_helper.o main.o $(CFLAGS) $(LDFLAGS)

file_helper.o: file_helper.c file_helper.h
    $(CC) -c file_helper.c $(CFLAGS)

data_processor.o: data_processor.c data_processor.h
    $(CC) -c data_processor.c $(CFLAGS)

main.o: main.c main.h
    $(CC) -c main.c $(CFLAGS)

.PHONY: clean
clean:
    rm *.o $(BIN)
```

Veel märkusi

Kaetud sai vaid kõige esmased teadmised

Retseptide käsitsi kirjutamine töötab vaid väikeste projektide käigus

- Tavaliselt kasutatakse üldistatud retsepte, mis töötavad ükskõik kui paljude failidega
- Need kohandatakse projektile sobilikuks (nt lipud kolmanda osapoole teekidele)

Suurte projektide jaoks kasutatakse sageli IDEsid, mis suudavad ise välja mõelda kuidas rakendusi kompileerida (vajavad siiski kohandamist – nt teegid)

Geany toetab mitme faili kompileerimist kasutades Makefile-i

- shift+f9

Vaata ka näidisprojekte koos Makefile'iga

Logimine

Logimine

Logimist kasutatakse enamasti jälgitavuse, analüütika ning silumise eesmärkidel

Iga logitav sündmus omab ajatemplit

Logitavatel sündmustel on sageli erinevad tasemed, kasutatav tase seadistatav

Logimise keerukus ja kohustus on sageli seotud toote äriloogika ja juriidiliste nõuetega

Logifailide säilitamine ja organiseerimine sõltub äriloogikast

- Näiteks võid kirjutada iga kord uue faili, lisada eelmise lõppu või kombineerida seda.
- Logisid võib jagada kaustadesse (nt kuu lõikes).
- Logide kustutamine / üle kirjutamine võib olla keelatud (jäädavalt, teatud perioodil, …)
- Vanade logide eemaldamine võib olla oluline kettaruumi säästmiseks
- Mida keerukam rakendus, seda olulisemaks logimine muutub

Logimine vajab täiendavat kettapinda, kasutab protsessori tsükleid ning lisab keerukust tarkvarasse.

Näide: logimise tasemed (VMWare)

Log Level Setting	Description
None	Disables logging.
Error	Logging limited to error messages.
Warning	Error messages plus warning messages are logged.
Info	Default setting on ESX/ESXi and vCenter Server systems. Errors, warnings, plus informational messages about normal operations are logged. Acceptable for production environments.
Verbose	Can facilitate troubleshooting and debugging. Not recommended for production environments.
Trivia	Extended verbose logging. Provides complete detail, including content of all SOAP messages between client and server. Use for debugging and to facilitate client application development only. Not recommended for production environments.

Source: https://pubs.vmware.com/vsphere-50/index.jsp?topic=%2Fcom.vmware.wssdk.pg.doc_50%2FPG_ChA_Diagnostics.19.4.html

Näide: WinSCP tekitatud logi

```
2020-02-10 16:06:28.639 Using SFTP protocol.

2020-02-10 16:06:28.639 Doing startup conversation with host.

2020-02-10 16:06:28.684 SFTP version 3 negotiated.

2020-02-10 16:06:28.684 We believe the server has signed timestamps bug

2020-02-10 16:06:28.684 We will use UTF-8 strings until server sends an invalid UTF-8 string
as with SFTP version 3 and older UTF-8 strings are not mandatory

2020-02-10 16:06:28.684 Changing directory to "/".

2020-02-10 16:06:28.686 Trying to open directory "/".

2020-02-10 16:06:28.687 Getting current directory name.

2020-02-10 16:06:28.747 Listing directory "/".

2020-02-10 16:06:28.961 Startup conversation with host finished.

2020-02-10 16:07:47.364 Changing directory to "StudentsHome ".

2020-02-10 16:07:47.369 Trying to open directory "/StudentsHome ".
```

Näide: Võimalik laboriülesande logi

```
2020.02.10 15:18:44 INFO: Program started
2020.02.10 15:18:44 ERROR: Opening input file "grades.txt" failed. Exiting.
2020.02.10 15:19:09 INFO: Program started
2020.02.10 15:19:09 INFO: Input file opened
2020.02.10 15:19:09 INFO: Input file closed, scanned 3 entries
2020.02.10 15:19:09 WARNING: Can't calculate average for 'Logistics' - no grades
2020.02.10 15:19:09 INFO: Program exit
2020.02.10 15:24:33 INFO: Program started
2020.02.10 15:24:33 INFO: Input file opened
2020.02.10 15:24:33 WARNING: The file too large be stored in memory, reading halted
2020.02.10 15:24:33 INFO: Input file closed, scanned 10 entries
2020.02.10 15:24:33 WARNING: Can't calculate average for 'Logistics' - no grades
2020.02.10 15:24:36 INFO: Program exit
```

Kellaaja pärimine

Lisa `<time.h>` teek

Andmetüübid

- `time_t` täisarv, enamjaolt UNIXi aeg
- `struct tm` aja struktuur, liikmeteks sekundid, minutid, tunnid, päevad, ..

`time_t` ja `struct tm` võivad sisaldada ükskõik millist legaalset kellaaega

- Nendega on võimalik ka erinevad operatsioonid ajaga

Hetke kellaaeg on süsteemi kellaajast sõltuv

- Praeguse kellaaja pärimiseks kasutame `time()`

Logimise rakendamisel pööra tähelepanu suve/talveajale, lisasekunditele, seaduste muutumisele (kella keeramisest loobumine, …)

- Oluline kriitilistes ja globaalsetes rakendustes
- Näiteks võib süsteemis kasutada UTC aega ning kohandada regioonile

Näide 1: ctime

```
Time is: Sun Feb 27 13:51:38 2022


-----------------
(program exited with code: 0)
Press return to continue
```

```c
#include <stdio.h>
#include <time.h>

int main (void)
{
    time_t currentTime;
    time(&currentTime);
    printf ("Time is: %s\n", ctime(&currentTime));
    return 0;
}
```

`ctime()` teisendab UNIXi aja loetavale kujule. Formaat on ettemääratud ja pole muudetav.

`Www Mmm dd hh:mm:ss yyyy`

Näide 2: kohandatud vorming

```
Time is 14:06 EET
Today is 27. February


------------------
(program exited with code: 0)
Press return to continue
```

```c
#include <stdio.h>
#include <time.h>

int main (void)
{
    time_t currentTime;
    struct tm *currentTimeStruct;
    char buf[64];

    time(&currentTime);
    currentTimeStruct = localtime(&currentTime);

    // %r is for HH:MM and %Z shows timezone if available
    strftime(buf, 64, "%R %Z", currentTimeStruct);
    printf("Time is %s\n", buf);

    // %d - day without preceding 0, %b - full month name
    strftime(buf, 64, "%d. %B", currentTimeStruct);
    printf("Today is %s\n", buf);
    return 0;
}
```

`strftime()` käsitsi koostada kellaaja ja kuupäeva vormingu

Parameetrid:

- Puhver kuhu ajatempel kirjutada (sõne)
- Puhvri pikkus
- Koostatav vorming
- `tm struct` tüüpi viit

Abiks vormingu koostamisel

- `man strftime`
- https://www.cplusplus.com/reference/ctime/strftime/