Practicum C: simpele preprocessor voor C

Informatica werktuigen

Academiejaar 2018-2019

1 Gedragscode

De practica worden gequoteerd, en het examenreglement is dan ook van toepassing. Soms is er echter wat onduidelijkheid over wat toegestaan is en niet inzake samenwerking bij opdrachten zoals deze.

De oplossing en/of programmacode die ingediend wordt moet volledig het resultaat zijn van werk dat je zelf gepresteerd hebt. Je mag je werk uiteraard bespreken met andere studenten, in de zin dat je praat over algemene oplossingsmethoden of algoritmen, maar de bespreking mag niet gaan over specifieke code of verslagtekst die je aan het schrijven bent, noch over specifieke resultaten die je wenst in te dienen. Als je het met anderen over je practicum hebt, mag dit er dus *nooit* toe leiden, dat je op om het even welk moment in het bezit bent van een geheel of gedeeltelijke kopie van het opgeloste practicum of verslag van anderen, onafhankelijk van of die code of verslag nu op papier staat of in elektronische vorm beschikbaar is, en onafhankelijk van wie de code of het verslag geschreven heeft (mede-studenten, eventueel uit andere studiejaren, volledige buitenstaanders, internet-bronnen, e.d.). Dit houdt tevens ook in dat er geen enkele geldige reden is om je code of verslag door te geven aan mede-studenten, noch om dit beschikbaar te stellen via publiek bereikbare directories of websites.

Elke student is verantwoordelijk voor de code en het werk dat hij of zij indient. Als tijdens de beoordeling van het practicum er twijfels zijn over het feit of het practicum zelf gemaakt is (bvb. gelijkaardige code, of oplossingen met andere practica), zal de student gevraagd worden hiervoor een verklaring te geven. Indien dit de twijfels niet wegwerkt, zal er worden overgegaan tot het melden van een onregelmatigheid, zoals voorzien in het onderwijs- en examenreglement (zie `http://www.kuleuven.be/onderwijs/oer/`).

2 Inleiding

Een preprocessor is een programma dat data omzet naar een geschikt formaat zodat dit gebruikt kan worden als invoer voor een ander programma. De C-preprocessor (`cpp`) is een programma dat broncodebestanden in C omzet naar een geschikt formaat voor de C-compiler. Broncodebestanden bevatten vaak informatie die onnuttig is voor de compiler, zoals regels commentaar. `cpp` zal onder andere die onnuttige informatie verwijderen uit het bestand. Dit wordt weergegeven in voorbeeld 1.

```
//Bereken de som van a en b
int som(int a, int b)
{
   //hier nog wat commentaar
   return a+b;
}
```

som.c.in

$\Rightarrow$ *cpp*

```

int som(int a, int b)
{

return a+b;
}
```

som.c

Voorbeeld 1 : Preprocessor verwijdert commentaren

Naast het verwijderen van overbodige informatie kent `cpp` ook commando's. Het meest bekende preprocessorcommando is waarschijnlijk `#include <file>`. Dit commando wordt gebruikt om verschillende broncodebestanden samen te voegen.

Het commando neemt als argument een verwijzing naar een ander bestand. De preprocessor zal vervolgens de inhoud van dit tweede bestand plakken op de locatie van het `#include`-commando, zoals weergegeven in voorbeeld 2. Om de complexiteit beperkt te houden moeten jullie in dit practicum `#include` *niet* implementeren. [1]

```
#include "som.c"
int main()
{
   return som(1, 2);
}
```

include.c.ex

$\Rightarrow$ *cpp*

```

int som(int a, int b)
{

return a+b;
}

int main()
{
return som(1, 2);
}
```

include.c

Voorbeeld 2 : Voorbeeld van `#include <file>`

Het commando `#define <key> <value>` wordt gebruikt om een alias te declareren. Elk voorkomen van `key` in het bronbestand wordt vervangen door `value` in de output.

```
#define SIZE 5
int a[ SIZE ];
int main(){
  int b= SIZE ;
  for(int i=0;i< SIZE ;i++){
    b+= i;
  }
  return b;
}
```

define.c.in

$\Rightarrow$ *cpp*

```
int a[ 5 ];
int main(){
int b= 5 ;
for(int i=0;i< 5 ;i++){
b+= i;
}
return b;
}
```

define.c

Voorbeeld 3 : Voorbeeld van `#define <key> <value>`

Een volledige opsomming van alle preprocessing commando's kan je terugvinden op `https://en.wikibooks.org/wiki/C_Programming/Preprocessor_directives_and_macros`. In dit practicum wordt verwacht dat jullie een subset van deze commmando's implementeren.

[1] Dit voorbeeld gebruikt het `#include`-commando met als argument een `.c`-bronbestand. In de praktijk is dit "bad practice", en gebruik je dit commando enkel om header-files (`.h`) te includen.

3 Achtergrond

3.1 Structuur project

De skeletcode en nodige tools en tests voor dit project kan je terugvinden op GitHub in deze repository: `https://github.com/KobeVrancken/iw-2018-c.git`. Haal deze repository op met het commando `git clone`.

3.1.1 Repository

Het kan zijn dat we gedurende de komende weken extra testbestanden toevoegen aan deze repository. We voegen enkel bestanden toe die jullie verder kunnen helpen. Wijzigingen aan de submission checker of het testscript zullen ook via deze repository doorgegeven worden. Vergeet dus niet om regelmatig een `git pull` uit te voeren! Voor belangrijke wijzigingen zullen we zeker een announcement plaatsen, maar niet voor kleine zaken.

De repository bestaat uit verschillende folders. We bespreken in deze sectie de bestanden in de `src`-folder. De andere folders worden toegelicht in verdere secties.

3.1.2 list.c en list.h

In dit practicum maken we gebruik van een gelinkte lijst van aliassen ((`key`, `value`)-paren). Deze gelinkte lijst wordt geïmplementeerd in de bestanden `list.c` en `list.h`. De meegeleverde implementatie in `list.c` is echter onvolledig. Als eerste stap van dit practicum is het de bedoeling dat jullie dit bestand vervolledigen. Nadien zal deze datastructuur gebruikt worden bij het implementeren van de preprocessor.

3.1.3 cpp.c

`cpp.c` is het hoofdbestand in dit practicum. Hier vind je onder andere de main-functie terug. Je zal in dit bestand moeten werken nadat je `list.c` hebt vervolledigd.

De meegeleverde `main`-functie zal bij de start van het programma de verschillende command-line argumenten opvragen. Elke command-line argument wordt vervolgens verwerkt door dit door te geven aan de functie `void option(char c);`. Extra command-line argumenten kunnen toegevoegd worden door deze functie uit te breiden.

Na het parsen van de command-line argumenten zal de main-functie het gevraagde invoerbestand uitlezen en de inhoud regel per regel doorgeven aan de functie `void process_line(char* line);`. Implementeer de preprocessor door deze functie te vervolledigen[2].

3.1.4 util.c en util.h

In deze bestanden worden enkele hulpfuncties gedefinieerd die gebruikt worden bij het uitlezen van bestanden en bij het geven van error messages. Aan deze bestanden moeten jullie niets aanpassen.

3.1.5 Makefile

Een Makefile is een bestand waarin men door middel van regels kan aangeven exact op welke manier de build van een project uitgevoerd moet worden. Wanneer men het commando `make` uitvoert in een folder met een Makefile, worden de verschillende commando's gedefinieerd in de Makefile

[2]Het is sterk aan te raden met hulpfuncties te werken. Schrijf niet de volledige preprocessor enkel in deze functie!

door middel van van deze regels uitgevoerd. Een simpele tutorial over het gebruik van Makefiles kan je terugvinden op `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`.

In dit practicum wordt verwacht dat jullie volledige project door middel van het commando `make` gebuild kan worden. Het resultaat van dit commando moet een werkend uitvoerbaar bestand `cpp-iw` zijn. Net na het clonen van de repository zou dit het resultaat moeten zijn van het make-commando:

```
$ cd iw-2018-c
$ make
$ ./iw-cpp
The awesome IW C preprocessor
Usage:  cpp-iw [-h -e] file.c
Recognized options
   -h display this help message
   -e terminate preprocessing on #error
   $
```

De meegeleverde Makefile is voldoende indien jullie zelf geen andere bestanden aanmaken. Wanneer jullie zelf wel eigen bestanden toevoegen moet de Makefile ook aangepast worden.

3.2 Standard streams

Bij het uitvoeren van een programma worden er standaard drie communicatiekanalen opengesteld voor communicatie tussen de gebruiker en het programma: de `stdin`, `stdout` en `stderr`. De `stdin` wordt gebruikt om invoer van de gebruiker door te geven aan het programma. De `stdout` wordt gebruikt om uitvoer van het programma te tonen. De `stderr` wordt gebruikt om foutmeldingen te tonen.

Wanneer we een programma uitvoeren met behulp van een shell (zoals `bash`), zullen standaard de `stdout` en de `stderr` als uitvoer getoond worden in de (pseudo)-terminal. Het is mogelijk om deze streams door te verwijzen naar een bestand, door middel van shell commando's. De preprocessor in dit practicum zal het verwerkte bestand printen naar de `stdout`. Foutmeldingen of debug-informatie wordt geprint naar de `stderr`.

Indien we de `stdout` van de preprocessor doorwijzen naar een bestand, wordt het resultaat van ons programma meteen in een bestand bewaard. Daarnaast houden we zo de foutmeldingen en debug-informatie gescheiden van de echte output van het programma.

De `stdout` van een programma doorverwijzen naar een bestand doen we met behulp van het `>`-teken:

```
$ cpp-iw invoer.c.in > uitvoer.c
```

Het is ook mogelijk de stderr van een programma door te verwijzen naar een bestand:

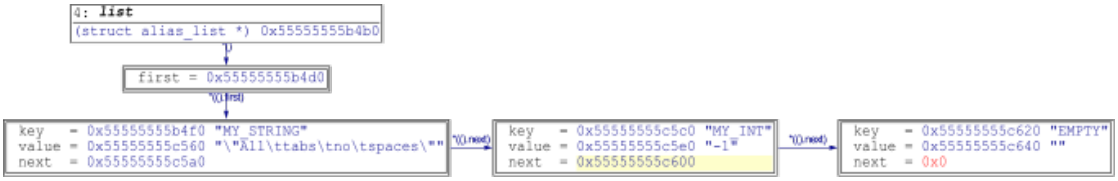
```
$ cpp-iw invoer.c.in 2> errorfile
```

Je kan beide ook combineren:

```
$ cpp-iw invoer.c.in > uitvoer.c 2> errorfile
```

In dit practicum is het de bedoeling dat de verschillende streams op de correcte manier gebruikt worden. Foutmeldingen en debug-messages schrijf je naar `stderr`, de output van de preprocessor schrijf je naar `stdout`. Maak gebruik van `printf` om te printen naar de `stdout`. Maak gebruik van de meegeleverde functie `void pr_error(const char* fmt, ...)` om te printen naar de `stderr`. Deze functie werkt op dezelfde manier als `printf`. Meer informatie over de verschillende streams kan je ook terugvinden in de allereerste oefenzitting Linux van dit vak.

3.3 Linked lists



Figuur 1: (Mogelijke) uitvoer van de debugger `ddd` voor de alias-list

Het bestand `list.c` definieert een gelinkte lijst van aliassen. Deze datastructuur is reeds besproken in het hoorcollege en de oefenzittingen. We geven hier kort nog wat extra informatie.

Het type `struct alias_list` bevat een verwijzing naar het eerste element van een lijst van aliassen. Indien het veld `first` van deze struct `0x0` (`NULL`) bevat is deze lijst leeg. In het andere geval bevat het veld `first` het geheugenadres van de eerste `struct alias`, zoals je kan zien in figuur 1.

Elke `struct alias` bevat drie velden. De velden `key` en `value` bevatten telkens verwijzingen naar het overeenkomstige (key, value)-paar. Het veld `next` bevat het geheugenadres van het volgende `struct alias` element, of `0x0` (`NULL`) indien dit het laatste element van de lijst is.

Figuur 1 bevat een grafische weergave van deze datastructuur in het geheugen. Deze weergave hebben we verkregen door met behulp van de debugger `ddd` de modeloplossing uit te voeren op het bestand `define-ext.c.in`. Zo ziet het geheugen van de modeloplossing na uitvoering van het `#define`-commando op regel 3.

3.4 Tokenizen van input

De reeds meegeleverde (onvolledige) implementatie van de functie `void process_line(char* line)` maakt gebruik van de functie `char *strtok(char *str, const char *delim)`. In deze sectie wordt kort uitgelegd hoe dit werkt. De werking van de meegeleverde code begrijpen zal van pas komen wanneer jullie deze willen uitbreiden.

`strtok` zal de invoerstring `str` opsplitsen in verschillende substrings, op de positie waar de string `delim` voorkomt. Dit doet `strtok` door elk voorkomen van de delimiter te vervangen door een null byte. *De invoerstring wordt dus bewerkt!*

De return value van `strtok` verwijst naar de eerste verkregen substring. Wanneer je vervolgens `strtok` nog eens oproept met als eerste parameter `NULL` en met dezelfde `delim`, zal de return value verwijzen naar de tweede substring. Je kan dit blijven herhalen tot de return value `NULL` is. Op dat moment heb je alle substrings overlopen. Meer informatie over deze functie kan je vinden door in je terminal onderstaand commando in te voeren.

```
$ man strtok
```

4 Functionaliteit

Deze sectie beschrijft de volledige functionaliteit die verwacht wordt in dit practicum. De codevoorbeelden die in deze sectie gegeven worden moeten exact op de voorgestelde manier verwerkt worden door jullie programma. Functionaliteit die niet in deze sectie gespecificeerd wordt is niet verplicht. Stel bij twijfel een vraag op het discussieforum!

4.1 Commentaar en witruimte

4.1.1 Witruimte

Alle witruimte aan de start van een regel wordt weggehaald. De meegeleverde implementatie doet dit reeds correct. Je moet er wel voor zorgen dat dit blijft werken.

```
int␣main(){
␣␣return␣1;
␣␣␣␣}
```

whitespace.c.in

⇒ *cpp*

```
int␣main(){
return␣1;
}
```

whitespace.c

Voorbeeld 4 : Witruimte

4.1.2 Single-line commentaar: //

Zodra je in een regel `//` tegenkomt, is de rest van de regel commentaar. Die commentaar mag niet voorkomen in het uitvoerbestand. Indien een volledige regel in het invoerbestand bestaat uit commentaar zal het uitvoerbestand op die plaats een lege lijn bevatten. Je mag er vanuit gaan dat, indien `//` in het midden van een regel voorkomt, er altijd een spatie voor zal staan. In dat geval is de lijn in het uitvoerbestand uiteraard niet leeg, maar wordt de commentaar gewoon verwijderd.[3]

```
//␣Single␣line␣comments␣should
    ↪ ␣be␣replaced␣with␣a␣
    ↪ space
int␣main(){
␣//Even␣if␣there␣is␣no␣
    ↪ whitespace␣behind␣//
␣return␣-1;␣//remove␣this
␣//You␣can␣assume␣that␣there␣
    ↪ is␣always␣either␣
    ↪ whitespace␣or␣a␣new␣line
    ↪ ␣character␣before␣//
}
```

single-comments.c.in

⇒ *cpp*

```

int␣main(){

return␣-1;

}
```

single-comments.c

Voorbeeld 5 : Single-line commentaar

4.1.3 Multi-line commentaar: /* */

Zodra je in een regel `/*` tegenkomt, zijn alle karakters tot en met het eerstvolgende voorkomen van `*/` commentaar. Al deze commentaar wordt integraal verwijderd. Je mag opnieuw aannemen dat voor elk voorkomen van `/*` of `*/` altijd een spatie zal staan.

[3] Gevallen waarbij commentaarkarakters in strings voorkomen (bijvoorbeeld `char* s = "string met // commentaar"`) mogen jullie buiten beschouwing laten. De automatische tests zullen nooit bestanden bevatten waarin dit voorkomt. Dit geldt ook voor multi-line comments.

```
/* Multi line comments

   Should be replaced as well
   */
int /*in the middle*/ main(){
 /*remove me*/ return 0;
}
```

multi-comments.c.in

⇒ *cpp*

```

int main(){
return 0;
}
```

multi-comments.c

Voorbeeld 6 : Multi-line commentaar

4.1.4 Tab-karakter

Elk voorkomen van het tab-karakter moet men vervangen door een spatie (voorbeeld 7), tenzij preprocessor commando's dit apart specificeren (zoals `#define`).

```
//	Replace all tabs
	int main() {
	//In code and comments!
		return 0;
}
```

tab.c.in

⇒ *cpp*

```

int main() {

return 0;
}
```

tab.c

Voorbeeld 7 : Tab-karakter

4.2 Commando's

Elk commando start met het karakter `#`. Ze kunnen enkel aan het begin van een regel voorkomen. Het is wel mogelijk dat commando's voorafgegaan worden door tabs of spaties. Regels waarin commando's voorkomen worden volledig verwijderd in het uitvoerbestand. We vervangen deze dus *niet* door een spatie. Dit geldt voor alle commando's. Alle commando's in deze sectie moeten ondersteund zijn.

4.2.1 #warning <message>

Wanneer de preprocessor het `#warning`-commando tegenkomt, moet de meegegeven boodschap geprint worden naar `stderr`.

```
#warning Deze boodschap naar
    ↪ stderr!
int main(){
 return -1;
 #warning En ook deze
    ↪ boodschap.
}
```

warning.c.in

⇒ *cpp*

```
int main(){
return -1;
}
```

warning.c

Voorbeeld 8 : Het warning-commando

Wanneer we voorbeeld 8 uitvoeren met `cpp-iw` moet de terminal er als volgt uitzien:

```
$ cpp-iw warning.c.in > warning.c
```

```
warning:  Deze boodschap naar stderr!
warning:  En ook deze boodschap.
        $
```

4.2.2 #error <message>

Wanneer de preprocessor het `#error`-commando tegenkomt moet, net zoals voor het `#warning`-commando, de meegegeven boodschap geprint worden naar `stderr`. Indien de preprocessor werd uitgevoerd met `-e` als argument, moet de preprocessor stoppen bij het eerste voorkomen van een `#error`-commando. Indien `-e` niet wordt meegegeven gaat de executie gewoon door, zoals voor het `#warning`-commando.

```
1 #error␣Deze␣boodschap␣naar␣
      ↪ stderr!
2 int␣main(){
3 ␣return␣-1;
4 ␣#error␣Deze␣enkel␣zonder␣-e␣
      ↪ flag
5 }
```

error.c.in

⇒ *cpp*

```
1 int␣main(){
2 return␣-1;
3 }
```

error.c

Voorbeeld 9 : Het error-commando

Wanneer we voorbeeld 9 uitvoeren met `cpp-iw` *met* de error flag moet de terminal er als volgt uitzien (in dit geval is error.c leeg):

```
        $ cpp-iw -e error.c.in > error.c
error:  Deze boodschap naar stderr!
        $
```

Wanneer we voorbeeld 9 uitvoeren met `cpp-iw` *zonder* de error flag moet de terminal er als volgt uitzien:

```
        $ cpp-iw error.c.in > error.c
error:  Deze boodschap naar stderr!
error:  Deze enkel zonder -e flag
        $
```

4.2.3 #define <key> <value>

Het commando `#define <key> <value>` moet ondersteund worden, waarbij `<value>` alles bevat dat na `<key>` komt, tot het einde van de regel. Voor dit commando is het de bedoeling dat elk voorkomen van `<key>` dat na dit commando voorkomt letterlijk vervangen wordt door `<value>`.[4]

De tokens `#define`, `<key>` en `<value>` kunnen onderling gescheiden worden door tabs of spaties. Tabs die voorkomen in `<value>` worden niet verwijderd.

[4] Je mag aannemen dat na elk voorkomen van `<key>` een spatie of tab voorkomt. In plaats van `CONSTANTE;` zullen we bijvoorbeeld telkens `CONSTANTE ;` schrijven in invoerbestanden.

```
#define MY_STRING "All tabs
    ↪ no spaces"
#define MY_INT -1
#define EMPTY
#define MAIN int main(){
EMPTY EMPTY EMPTY
MAIN
char** s = MY_STRING;
return MY_INT;
}
```

define-ext.c.in

⇒ *cpp*

```

int main(){
char** s = "All tabs no
    ↪ spaces";
return -1;
}
```

define-ext.c

Voorbeeld 10 : Het define-commando

Indien `#define`-commando een `<key>` gebruikt die reeds eerder was gedefinieerd, moet gekeken worden naar de meegegeven `<value>`. Indien deze nieuwe waarde hetzelfde is als de oude waarde mag het commando genegeerd worden. Indien de nieuwe waarde verschilt van de oude waarde moet er een waarschuwing geprint worden. De nieuwe waarde van de key wordt wel bewaard. Dit gedrag wordt weergegeven in voorbeeld 11.

```
#define MY_INT 1
#define MY_INT 1
#define MY_INT 2
int main(){
return MY_INT;
}
```

define-double.c.in

⇒ *cpp*

```
int main(){
return 2;
}
```

define-double.c

Voorbeeld 11 : Het define-commando wanneer een value hergebruikt wordt (*zonder* error-flag)

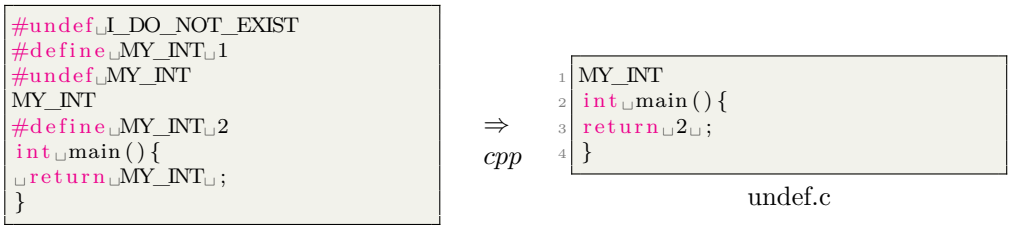
Wanneer we voorbeeld 11 uitvoeren met `cpp-iw` moet de terminal er als volgt uitzien:

```
    $ cpp-iw define-double.c.in > define-double.c
warning:  'MY_INT' redefined
    $
```

Maak voor de implementatie van dit commando gebruik van de gelinkte lijst van aliassen in `list.c`.

4.2.4 #undef <key>

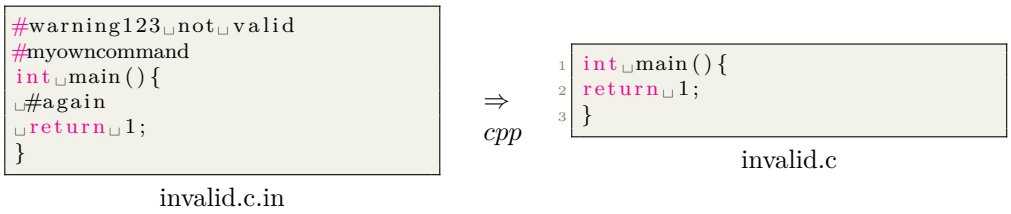
Met `#undef <key>` kunnen reeds gedefinieerde keys terug verwijderd worden. Nadien kan de key hergebruikt worden in een ander `#define`-commando. Wanneer `#undef <key>` wordt uitgevoerd op een onbestaande key wordt het commando genegeerd. Er wordt in dat geval geen warning noch error getoond. Wanneer men voorbeeld 12 uitvoert wordt er geen waarschuwing of foutmelding getoond.



Voorbeeld 12 : Het undef-commando

4.2.5 Onbekende commando's

Alle commando's die gebruikt worden maar niet ondersteund worden door je implementatie geven een foutmelding. Indien de error-flag actief is moet de uitvoering van de preprocessor stoppen.



Voorbeeld 13 : Onbekende commando's

Wanneer we voorbeeld 13 uitvoeren met `cpp-iw` moet de terminal er als volgt uitzien:

```
$ cpp-iw invalid.c.in > invalid.c
error:  invalid preprocessing directive:  warning123
error:  invalid preprocessing directive:  myowncommand
error:  invalid preprocessing directive:  again
$
```

Wanneer we voorbeeld 13 uitvoeren met `cpp-iw` *met* de error-flag actief moet de terminal er als volgt uitzien:

```
$ cpp-iw -e invalid.c.in > invalid.c
error:  invalid preprocessing directive:  warning123
$
```

In dit geval is het uitvoerbestand leeg.

4.2.6 Bonus

Indien je klaar bent met het volledige practicum mag je altijd eigen functionaliteit toevoegen. Voor inspiratie kan je kijken naar de echte C-preprocessor `cpp`. Het kan ook interessant zijn om te zoeken naar een betere datastructuur dan de gelinkte lijst om aliassen te bewaren.

Als je uitbreidingen implementeert, voorzie dan de optie `-u` en zorg ervoor dat deze extra functionaliteit enkel uitgevoerd wordt indien deze flag *expliciet* meegegeven wordt. Zo voorkom je problemen met de automatische tests.

Extra functionaliteit is niet nodig om de maximumscore te behalen op dit practicum. Het kan er wel voor zorgen dat eventueel puntenverlies door kleine fouten in andere delen van de code

opgehaald kan worden. Begin hier niet aan, tenzij alle tests slagen.

5 Tests

5.1 Gelinkte lijst

In `list.c` staat een methode `int alias_list_test()` die gebruikt wordt om de functionaliteit van de gelinkte lijsten te testen. Vul deze methode aan met eigen tests voor de methodes die je zelf hebt geïmplementeerd. Het is aan te raden je eigen functies uitgebreid te testen. Fouten in deze code kunnen voor bugs zorgen in je preprocessor die lastig te detecteren zijn.

5.2 Preprocessor

In de repository vinden jullie het script `test-script`. Dit script kan gebruikt worden om reeds geschreven functionaliteit te testen. Het script compileert je programma met de makefile en kopieert de `.in`-bestanden uit de testfolder naar een tijdelijke folder in `/tmp`. Vervolgens voert hij je programma uit op alle `.in` bestanden en genereert zo de overeenkomstige `.out`-bestanden. Ten slotte wordt gekeken of het verkregen `.out`-bestand identiek is aan de meegeleverde referentie (het normale `.c`-bestand). Indien het bestand verschilt wordt de output van het `diff`-commando op deze twee bestanden getoond.[5] De `stderr` wordt op gelijkaardige manier gegenereerd en vergeleken met de referentie in de `.err`-files.

Indien je het script uitvoert zonder argumenten zal het uitgevoerd worden voor alle `.in`-bestanden in de directory tests. Je kan ook eigen tests toevoegen.

Je kan het script ook uitvoeren met als argument een specifiek bestand dat je wil testen. In dit geval moet je geen extensie meegeven. Om bijvoorbeeld whitespace.c.in te testen voer je dus het volgende commando uit:

```
$ ./test-script whitespace
==================
whitespace.c
==================
stdout:  SUCCESS
stderr:  SUCCESS
$
```

De volledige folder test je dus simpelweg met het commando

```
$ ./test-script
```

Je kan ook opteren om de output van het `diff`-commando in een meer visuele vorm te krijgen door gebruik te maken van `vimdiff`. Dit doe je als volgt:

```
$ VIMDIFF=1 ./test-script
```

Als reactie op vragen op het discussieforum kan het zijn dat er nieuwe testbestanden worden toegevoegd aan de git repository. Vergeet dus niet om van tijd eens een `git pull` uit te voeren om te kijken of er nieuwe test cases te vinden zijn.

[5] Voer `man diff` uit in een terminal om meer informatie te krijgen over het `diff`-commando

6 Strategie en tips

In deze sectie geven we enkele tips bij het oplossen van dit practicum.

6.1 Volgorde

Bij het oplossen van het practicum is het aan te raden onderstaande volgorde te volgen. Controleer tussen elke stap of je de geïmplementeerde functionaliteit kan testen. Het is slim om ook telkens je eigen tests toe te voegen.

1. Implementeer de lege functies in `list.c`. Zorg ervoor dat alle list tests slagen.
2. Zorg ervoor dat single-line comments correct afgehandeld worden
3. Zorg ervoor dat multi-line comments correct afgehandeld worden
4. Zorg ervoor dat het tabkarakter correct verwijderd wordt
5. Implementeer het warning-commando
6. Implementeer het error-commando. Voeg hiervoor de `-e` flag toe.
7. Implementeer het define-commando
8. Implementeer het undef-commando
9. (optioneel) Implementeer bonusfunctionaliteit naar keuze

6.2 Tijdsplanning

Begin ruim op tijd aan dit practicum. Indien je pas op zondag 16 december begint neem je een groot risico, zelfs indien je de materie goed beheerst.

Het is een goed idee om ervoor te zorgen dat je vóór de tweede oefenzitting van `C` stap 1 zeker afgewerkt hebt, zodat je tussen de tweede en de derde oefenzitting aan de preprocessor zelf kan werken. *Na de derde oefenzitting zijn er nog maar twee dagen om je submissie af te werken!*

In dit practicum bieden wij een submission checker aan waarmee je je opgave kan controleren op geldigheid. Reken voldoende tijd uit om met behulp van de submission checker jou submissie te controleren *op de machines van het departement* (dit kan via ssh of door fysiek aan te melden). Indien je hierbij een probleem ontdekt om 23:50 kan het goed zijn dat je dit niet op tijd opgelost krijgt.

Voor vragen die op het discussieforum gesteld worden vóór vrijdag 14 december 16:00 garanderen wij een tijdig antwoord. Nadien kunnen we niet garanderen dat je vragen of mails nog beantwoord worden voor de deadline.

6.3 Discussieforum

Alle vragen moeten gesteld worden op het discussieforum. Controleer dit regelmatig, zelfs indien je zelf geen vragen hebt. Vaak lopen medestudenten tegen gelijkaardige problemen aan. Indien je het discussieforum actief in de gaten houdt, ben je op de hoogte van de moeilijke delen van het practicum. Dit zorgt ervoor dat je des te sneller kan werken. Je moet zo geen tijd verspillen aan problemen die andere studenten reeds hebben opgelost. Zo stond er voor het Visual Basic-practicum een uitgewerkt voorbeeld voor de sleepinteractie op het discussieforum. Velen onder jullie hadden veel tijd kunnen uitsparen indien jullie dit toen gezien hadden. Om op de

hoogte te blijven van iedere post in het forum kan je op Toledo op “subscribe” klikken. Zo krijg je een e-mail wanneeer iemand een bericht plaatst.

7 Afspraken

7.1 Indienen

Je practicum moet **ten laatste op zondag 16 december 2018 om 23:59** ingeleverd worden.

Genereer hiervoor een zip-bestand van je projectfolder:

```
$ zip -r iw_<voornaam>_<naam>_<r-nummer>.zip folder-naam/
$
```

Controleer met behulp van de submission checker in de folder `tools` na of je opgave voldoet aan de minimale voorwaarden voor dit practicum.

```
$ ./submission-checker /home/r0xxxxx/pad/naar/file/iw_<voornaam>_<naam>_<r-nummer>.zip
De submissie voldoet aan de minimale vereisten!
$
```

De submission checker werkt in de Linux-omgeving aangeboden op het departement. Andere omgevingen gebruiken doe je onder eigen verantwoordelijkheid. Jullie hebben allen toegang, via ssh en fysiek, tot de machines in de computerleslokalen in 200A. Voer de submission checker zeker uit op deze machines. *Indien het niet werkt op deze machines, ook al werkt het wel op je eigen machine, voldoet je submissie niet aan de minimale vereisten!* Hier zullen geen uitzonderingen op worden gemaakt.

7.1.1 Laattijdig indienen

Dit practicum wordt gequoteerd op 20 punten. Bij een laattijdige inzending trekken we in dit practicum een punt voor elk uur. Voor een submissie tussen 00:00 en 00:59 verlies je één punt. Tussen 01:00 en 01:59 verlies je twee punten. Indien je indient om 19:00 of later kan je geen punten meer scoren.

7.1.2 Onvolledig indienen

Samen met dit practicum leveren we een submission checker waarmee je jou zipfile met oplossing kan controleren. Indien je een programma indient dat niet wordt aanvaard door de submission checker *op de machines van het departement* kan je geen punten scoren op dit practicum. Hier maken we geen uitzonderingen op. Controleer dus zeker of je ingediende bestand op Toledo aanvaard wordt door deze tool! De submission checker zal code die niet compileert niet aanvaarden. Het heeft geen zin om code in te dienen die we niet kunnen compileren.

Het is niet moeilijk om te voldoen aan de voorwaarden van de submissie checker. Indien je zipbestand een correcte naam heeft en een Makefile bevat in de top-level folder waarmee je het programma cpp-iw kan genereren *op de machines van het departement*, wordt je submissie aanvaard. Wanneer je meteen na het clonen van de git-repository met de opgave deze folder zou zippen met een correcte naam, zou dit bijvoorbeeld aanvaard worden door de checker (maar uiteraard geen punten opleveren).

7.2 Verdediging

Er zal opnieuw een verdediging plaatsvinden van dit project, waar jullie zelf uitbreidingen zullen moeten implementeren. Je krijgt enkel een verdedigingsmoment toegewezen indien je code aanvaard wordt door de submission checker. De verdediging is een examenmoment, behandel dit ook zo. Zorg ervoor dat je op tijd aanwezig bent op je toegekende moment. Indien je om eender welke reden niet op tijd op de verdediging kan geraken, neem dan zo snel mogelijk contact op met het assistententeam zodat er gezocht kan worden naar een oplossing. *Niet komen opdagen op je verdediging, zonder enige verwittiging, zal resulteren in een 0 op dit practicum.* Indien je door ziekte je verdediging niet kan halen, bezorg je je ombudspersoon een doktersattest. Via de ombudspersoon kan dan een inhaalverdediging vastgelegd worden.

Veel succes!