ICPT verslag

Barnhoorn, Mees
S1127445

13-01-2024
Hogeschool Leiden, ICPT, opleiding Informatica

Inhoudt

Voorwoord

Bij dit verslag horen opdrachten, aangezien het op het moment van schrijven nog onduidelijk is hoe deze ingeleverd moeten worden zijn deze mogelijk in de bijlage te vinden. Als alternatieve locatie voor de opdrachten zijn deze ook naar github geupload op de volgende link: https://github.com/dev-mmb/ICPT. Niet alle code voorbeelden komen uit deze opdrachten, dit is vooral zo bij de C++ voorbeelden. Dit komt omdat er geen C++ opdracht was om voorbeelden uit te halen. Dit geld ook voor een aantal C# voorbeelden. Er was geen goed voorbeeld voor het verschil tussen parameter passing methoden aanwezig in de opdrachten. Deze zijn dus ook apart bijgeschreven.

1. Historie

Een lambda is een concept in programmeertalen waar een functie zonder naam gedefinieerd kan worden. In onderstaande code is bijvoorbeeld een functie genaamd ‘functionName’ gedefinieerd, dit is een normale functie. In tegenstelling tot deze functie wordt er ook een ‘anonieme’ functie gedefinieerd en in de variabele ‘lambda’ gestopt.

```
void fuctionName() {};
lambda = () => {};
```

In dit hoofdstuk wordt de geschiedenis van de ‘lambda’ of ‘anonieme functie’ beschreven, ook wordt er voor een aantal moderne talen beschreven hoe de lambda in deze talen werkt. Dit is niet een uitputtende lijst aangezien vrijwel elke moderne programmeertaal het concept van de lambda kent.

1.1 Lambda Calculus

Het concept van de lambda komt uit de ‘lambda calculus’, een van de fundamentele theorieën over ‘computability’ bedacht door Alonzo Church in de jaren 1930 (unc.edu, sd). Binnen de lambda calculus is alles een functie, er zijn geen ‘primitieve’ types zoals booleans of integers. Als de programmeur deze wil gebruiken zullen ze eerst uit een functie moeten komen. Door het enkel gebruiken van functies welke geen ‘instructies’ uitvoeren zoals in een ‘imperative’ taal (java, c++) zijn er ook geen ‘side effects’ aanwezig. Het is onmogelijk dat een functie data verandert, het heeft simpelweg een input en een output. In ‘pure’ lambda calculus mag er zelfs maar een enkel argument aan een functie geven worden, dit is echter niet altijd zo in daadwerkelijke programmeertalen.

Een functie in lambda calculus is gedefinieerd als: f = λ input. Output. Waar de functie f een input krijgt en een output geeft. Zo heeft de functie a = λ x. x, een input x en geeft deze x weer terug. Dus a met input ‘1’ geeft weer ‘1’ terug, eens gelijks geeft de functie b = λ x.x+1, met 1 als input, 2 als output.

Volgens Alonzo Church kan deze calculus alles wat een computer ook kan (cse.psu.edu, sd).

1.2 Lisp

In de late jaren 1950 heeft John McCarthy de eerste versies van Lisp gedefinieerd. Deze gebruikten de lambda notatie van Church. Lisp wordt vandaag de dag nog steeds gebruikt als, Common Lisp, Scheme en elisp. Volgens (McCarthy, 1979, pp. 173-197) had de lambda calculus weinig impact op het ontwikkelen van Lisp:

> "To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ -notation of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions."

Buiten het gebruik van de lambda notatie heeft lisp ook het gebruik van een ‘garbage collector’, ‘closures’ voor static scoping, conditionele expressies voor het gebruik van recursion en higher order operations voor lists gepioneerd. Aspecten die door veel moderne talen nog steeds gebruikt worden.

1.3 FP

In 1978 definieerde John Backus FP (Function programming) (math.bas.bg, sd) (Backus, 1978). FP is niet een 'echte' programmeertaal, aangezien het niet een vaste inhoudt en geen input/output heeft. FP is meer een familie van talen, beschreven door Backus als 'FP systems'. De bekende FP taal is een instantie van deze 'FP systems' gedefinieerd door Backus in (Backus, 1978).

Een programma in FP is een set van functies. Deze functies hebben geen losstaande variabelen of expliciet gedefinieerde parameters. Elke functie heeft altijd een enkele parameter, deze parameter heeft dus geen naam nodig. Nieuwe functies kunnen opgebouwd worden uit oude functies, een programma in FP wordt vervolgens gebouwd door een van deze functies uit te voeren met data als input. Rond 1980 hebben Backus en zijn collega's de taal FL gemaakt. FL is een taal gebaseerd op de filosofie van FP.

1.4 Algol

Algol 60 implementeert geen lambda's, wel gebruikt de compiler 'Call by name' (depaul.edu, sd) wat hier op lijkt. Als een functie met parameters wordt aangeroepen in Algol 60, dan wordt voor elke parameter een 'thunk' (jargon300/thunk.html, sd) gegenereerd. Een 'thunk' is een tweetal anonieme functies die de waarde van de parameter ophalen (get) en veranderen (set). Dus ondanks dat Algol 60 geen lambda's heeft, word er wel een soort gelijk mechanisme door de compiler gebruikt. Echter maakt dit het wel onmogelijk om te redeneren over functies, aangezien iedere functie met parameters nu altijd side effects heeft. Dit is ook de reden dat 'Call by name' niet meer gebruikt wordt (Cartwright, sd). Algol 68 had lambda's vanaf het moment dat het uit kwam.

1.5 C

Lambda's zijn niet officieel een onderdeel van C, echter zijn deze wel in GCC geïmplementeerd als een 'non standard' feature. Via de volgende macro kunnen lambda's geimplementeerd worden in C:

```
#define lambda(lambda$_ret, lambda$_args, lambda$_body)\
({\
lambda$_ret lambda$__anon$ lambda$_args\
lambda$_body\
&lambda$__anon$;\
})
printf("%f\n", average_apply(lambda(float,(float x),{ return 2*x; })));
```

Zonder de 'gcc -O2' compiler optie, werkt deze code niet en resulteert in een 'seg fault' (Williams, 2019).

1.6 Functionele talen

Lambda's zijn een concept binnen het functionele programmeren. Het is dus niet gek dat de meeste functionele talen het concept van de lambda ondersteunen. Een aantal van de grote

functionele talen met ondersteuning voor de lambda zijn: Erlang (1986), Haskell (1990), OCaml (1996), Scala (2003), F# (2005), Clojure (2007).

1.7 Java, C#, C++ en JS

In 2014 zijn lambda's aan Java toegevoegd (Warburton, 2024). C# kreeg lambda's in 2007 (Code Maze, 2024). C++ kreeg lambda's met het uitkomen van C++ 11 in 2011. Kortom de meeste 'grote' talen hebben tegenwoordig lambda's. Dit komt mogelijk door de groeiende populariteit van JavaScript, een taal waar soms meer lambda's worden gebruikt dan normale functies. Tegenwoordig hebben vrijwel alle nieuwe talen lambda's als een feature.

2. Compiler en Toolchain (C)

De C compiler kent een aantal stappen die doorlopen worden om van C code een 'executable' (uitvoerbaar programma) te maken. Deze stappen zijn:

1. Pre Compiler
2. Compiler
3. Assembler
4. Linker

In dit hoofdstuk zal elk van deze stappen aan de hand van voorbeelden worden uitgelegd. Deze voorbeelden zijn in de bijlage te vinden onder 'c_toolchain', deze bestaan uit de 'main.c', 'pre_compiler.c', 'main.s', 'main.o', en 'main.out' bestanden.

2.1 Pre Compiler/ processor

Bij de eerste stap die de C compiler maakt worden 'comments' uit de code verwijderd, een comment is een stuk tekst in de code welke vaak gebruikt wordt om de code te beschrijven. Deze comments zijn onnodig voor de werking van het programma en worden dus door de pre compiler verwijderd.

Macros

De C taal kent het concept van 'Macros', een soort meta-taal die uitgevoerd wordt door de pre processor. Een macro begint altijd met een '#' (hashtag). De meest gebruikt vorm van macros is de 'define' macro.

```
#define TEN 10
#define SUM(a, b) (a + b)
```

De define macro definieert het woord of de functie aan de linker kant van de macro, als de waarde aan de rechter kant. In het voorbeeld hierboven wordt bijvoorbeeld 'TEN' als '10' gedefinieerd. 'SUM' wordt vervolgens gedefinieerd als 'a + b', waar 'a' en 'b' de inputs van de macro zijn.

De macros worden vervolgens door de pre compiler omgezet in de waardes die zij definiëren. Zo wordt 'SUM(TEN, 2)' omgezet in '10 + 2'.

```
int result = SUM(TEN, 2);
int result = (10 + 2);
```

De pre processor wordt ook gebruikt voor '#include'. Als de pre processor een include tegenkomt, dan zoekt deze naar de file binnen de haakjes na de '#include' en plakt deze in zijn geheel in de file waar de include in stond.

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

De pre processor kan zelfs condities uitvoeren. Hiervoor word een '#ifdef' gebruikt. De ifdef checkt of een waarde gedefinieerd is met een #define. Als dit niet zo is dan wordt de code binnen het ifdef blok verwijderd. Na de ifdef kan ook een '#else' blok gedefinieerd worden, op deze manier kunnen er if else chain gebruikt worden voor de pre compiler. Dit is bijvoorbeeld handig om enkel code uit te voeren in debug mode. Een ifdef eindigt altijd met een '#endif'.

```
#ifdef DEBUG
    printf("Debug mode\n");
#else
    printf("Release mode\n");
#endif
```

Gebruik van de pre processor

De pre compiler kan gebruikt worden met de gcc commando 'gcc -E'. Dit print het resultaat van de pre compiler naar de console.

Het probleem met de pre processor is dat het geen enkele error checken doet. In het volgende stuk code wordt de ADD-macro gedefinieerd, deze is fout want er staat een extra haakje in die niet correct gesloten wordt.

```
#define TEN 10
#define ADD(a, b) ((a + b)

int main(int argc, char **argv) {
    int x = ADD(1, 2);
    return 0;
}
```

De preprocessor is dom en plakt simpelweg het extra haakje ook in de code, waardoor de code vervolgens niet meer compileert. Als de ADD-macro in een apart bestand staat dan wordt dit nog best een klus om op te sporen. Aangezien de error niets over deze macro zegt en enkel een syntax error geeft.

```
int main(int argc, char **argv) {
    int x = ((1 + 2);
    return 0;
}
```

2.2 Compiler

Tijdens de compiler stap wordt de C code omgezet in assembly (.s bestand). Dit kan gedaan worden met de -S flag 'gcc -S main.c'. De volgende assembly code wordt gegenereerd met dit commando:

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 2
    .globl  _main                           ; -- Begin function main
    .p2align    2
_main:                                  ; @main
    .cfi_startproc
; %bb.0:
    sub sp, sp, #48
    .cfi_def_cfa_offset 48
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    mov w8, #0
    str w8, [sp, #12]                   ; 4-byte Folded Spill
    stur    wzr, [x29, #-4]
    stur    w0, [x29, #-8]
    str x1, [sp, #16]
    adrp    x0, l_.str@PAGE
    add x0, x0, l_.str@PAGEOFF
    bl  _printf
    ldr w0, [sp, #12]                   ; 4-byte Folded Reload
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    add sp, sp, #48
    ret
    .cfi_endproc
                                        ; -- End function
    .section    __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
    .asciz  "Hello, world!\n"

.subsections_via_symbols
```

Tijdens deze stap worden ook syntax errors en types gecheckt. Het is belangrijk om bij deze stap de errors te checken aangezien na deze stap de originele code verloren gaat. Er is enkel nog assembly code over. Deze stap is in tegenstelling tot de preprocessor ook systeem afhankelijk. Voor een Intel chip moet andere assembly worden gegenereerd dan voor een AMD-chip. Ook CPU-architectuur maakt hier een verschil, ARM assembly zien er anders uit dan X86 instructies. Het maakt ook uit of de er voor 64 of 32 bit wordt gecompileerd, dit heeft namelijk ook invloed op hoe de assembly er uit moet zien.

2.3 Assembler

Tijdens de assembler stap wordt de zojuist gegenereerde assembly omgezet in machine code (binaire code) zodat de computer deze kan snappen. Deze code wordt opgeslagen in object files (.o/.obj bestanden). Deze stap kan uitgevoerd worden met het volgende commando: 'gcc -c main.s -o main.o'.

2.4 Linker

Tijdens de linking fase wordt externe code aan de zojuist geassembleerde code toegevoegd. Als er externe libraries worden gebruikt dan kent ons programma nog niet de code die in deze libraries gedefinieerd staat. Het weet enkel dat deze definities uit een library komen. De library bestaat ook uit object files, zo is het niet nodig om een al gemaakte library opnieuw te compileren en assembleren.

De linker kan gerund worden met het volgende commando: 'gcc main.o -o main.out', als de '-o' flag niet gebruikt wordt om het output bestand een naam te geven dan krijgt deze automatisch de naam 'a.out'. Dit '.out' bestand is onze gemaakte executable, op windows zou dit dan ook een '.exe' extensie hebben om uitgevoerd te worden.

3. Memory management (C – Stratego)

In de meeste moderne talen is het niet meer nodig om handmatig geheugen te managen, echter moet dit in oudere en of talen met een focus op snelheid nog wel vaak.

Als een programma start dan krijgt het van de OS (operating system eg. Windows, MacOS) een blok met geheugen. Dit blok wordt opgedeeld in drie segmenten de stack, de heap en het tekst segment. Het tekst segment is waar het programma zelf te vinden is. De stack is een snel maar klein stuk geheugen en de Heap is een groot maar relatief langzaam stuk geheugen. Deze hebben beiden hun eigen functie, werking en doel.

3.1 Werking en doel van de stack

De stack is meestal letterlijk als een stack datastructuur geïmplementeerd. Een stack is een LIFO (last in, first out) structuur. Het kan gezien worden als een stapel waar data boven op wordt gedaan, als er data verwijderd moet worden van deze stapel dan moet eerst de data die bovenop ligt ervan af worden gehaald.
In het geval van een stack die gebruikt wordt voor geheugen is de data die op de stack gedaan worden zogenaamde 'stack frames'. Als een functie wordt aangeroepen dan worden de parameters van deze functie toegevoegd aan de stack. Vervolgens wordt de functie uitgevoerd, als er een variabele wordt aangemaakt dan wordt deze toegevoegd aan de huidige stack frame.

```
int baz(int a, int b) {
    int z = a + b;
    return z;
}
```

```
void foo(int x) {
    int y = 0;
    int c = baz(x, y);
}
foo(1);
```

Als de bovenste functie ‘foo’ uitgevoerd wordt, dan wordt eerst een nieuwe stack frame aangemaakt. Vervolgens wordt x = 1 aan deze frame toegevoegd. De functie wordt doorlopen en y = 0 wordt aan de huidige stack frame toegevoegd. Als de ‘baz’ functie aangeroepen wordt dan wordt weer een nieuwe stack frame aangemaakt.

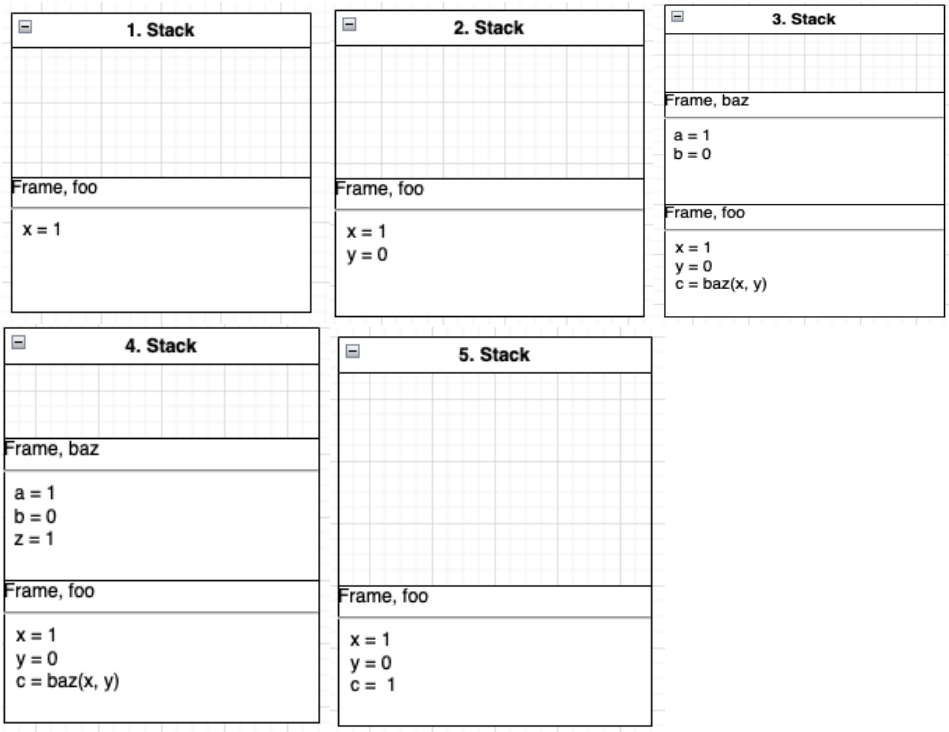


Figure 1 De Stack

1. Als de functie ‘foo’ aangeroepen wordt dan wordt deze toegevoegd aan de stack, de parameter x krijgt binnen de stack frame de waarde 1.
2. Als vervolgens ‘y = 0’ wordt uitgevoerd dan wordt y en de waarde binnen y toegevoegd aan de huidige stack frame.
3. De ‘baz’ functie wordt aangeroepen en krijgt een nieuwe stack frame. De parameters a en b krijgen hun waarde gekopieerd van de x en y waardes waarmee de ‘baz’ functie is aangeroepen.
4. De z variabele wordt aangemaakt en krijgt de waarde ‘a + b’, dit is 1. Dus z = 1 wordt toegevoegd aan de bovenste stack frame.
5. De ‘baz’ functie komt op een return, dus wordt de ‘baz’ stack frame verwijderd van de bovenkant van de stack. De waarde c krijgt de waarde die ‘baz’ teruggeeft.

Als er data van de stack wordt gebruikt hoeft er nauwelijks gezocht te worden, er kan simpelweg in de bovenste stack frame gekeken worden, als de gezochte variabele hier niet wordt gevonden kan naar de volgende stack frame gegaan worden. Hierdoor krijg je ook een soort 'scope' waar de variabelen van de 'foo' frame gebruikt kunnen worden in de 'baz' functie. Maar de variabelen uit de 'baz' functie kunnen niet gebruikt worden in de 'foo' functie.

Het verwijderen van data is ook een stuk sneller dan de Heap op deze manier, er hoeft niet gezocht te worden naar een specifiek adres. Als een functie klaar is, dan is de bovenste stack frame niet meer nodig en kan deze verwijderd worden. Het grote nadeel van de stack is dat deze relatief klein is. Het is ook niet mogelijk om 'permanente' data op de stack te gebruiken. Als de functie waar de data in is gedefinieerd klaar is dan is de data ook meteen weg.

In het volgende voorbeeld wordt het adres van z in de pointer 'p' opgeslagen. Als vervolgens de 'p' pointer gedereferenced wordt om de opgeslagen waarde te bekijken dan komt er 'undefined behavior' (cppreference.com, 2023) voor, aangezien de waarde van 'z' verwijderd is toen de stack frame van 'baz' verwijderd werd. Het is dus mogelijk dat er nu geen data meer aanwezig is op het adres van z.

```
int* p = nullptr;
void baz(int a, int b) {
    int z = a + b;
    p = &z;
}
void foo(int x) {
    baz(x, y);
    int c = *p; // z is out of scope, so its data is possibly gone
}
foo(1);
```

Stack overflow

De stack heeft maar een vaste hoeveelheid geheugen tot diens beschikking. Als dit overschreden wordt dan komt er een 'stack overflow' error voor en crasht het programma. Dit kan simpelweg gedaan worden met de volgende functie:

```
int add(int n) {
    return n + add(n + 1);
}
```

Er is geen eind conditie voor de recursie van deze functie. Deze zal dus voor eeuwig nieuwe stack frames aan blijven maken.

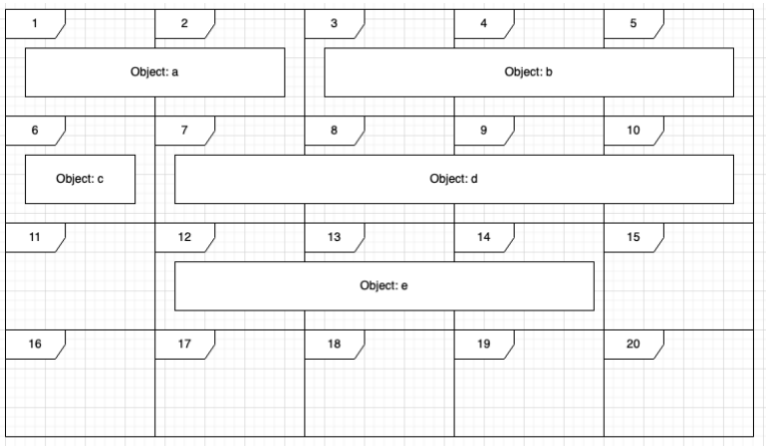
3.2 Werking en doel van de Heap

De heap is een grote 'hoop' met data die gebruikt wordt door het programma, als deze data aangemaakt wordt dan moet deze ook handmatig verwijderd worden (in C tenminste). Bij

talen met een ‘garbage collector’ zoals java en C# wordt dit automatisch gedaan door een apart programma dat met de taal komt.

In C wordt er data aan de heap toegevoegd met de ‘malloc’ functie, deze functie neemt een hoeveelheid data als parameter en geeft een pointer naar de zojuist aangemaakte data terug. Om deze data vervolgens weer te verwijderen moet de ‘free’ functie met als parameter het gegeven adres als pointer aangeroepen worden. In C++ worden hier de new/new[] en delete/delete[] keywords voor gebruikt. Het is in C++ ook mogelijk om zogenaamde ‘smart pointers’ te gebruiken, dit is dan een wrapper om de new/delete keywords (cppreference.com, 2023).

```
void memory() {
    // in C
    int* c = malloc(1 * sizeof(int));
    free(c);
    // in C++
    int* cpp = new int();
    delete cpp; // use delete[] if q is an array allocated with new int[size]
}
```

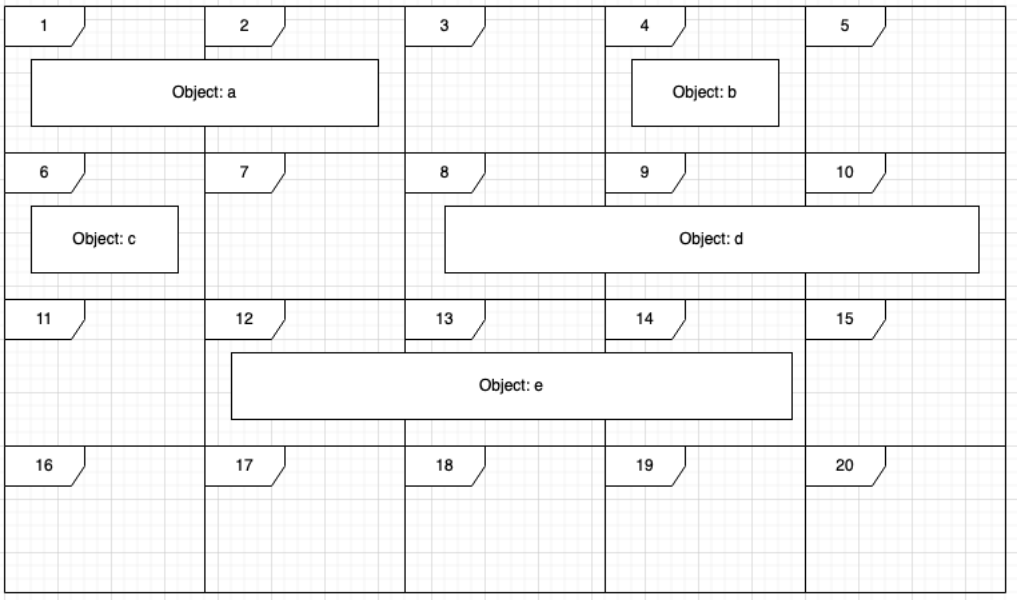


Elk object op de heap heeft een adres, dit is het adres van de eerste byte van het object. Zo is het adres van object: b, in deze heap ‘3’. De pointer naar dit object wijst naar dit adres. Zie figuur 6.

Figuur 1 De Heap

Heap fragmentation

Als er veel data op de heap wordt aangemaakt en weer verwijderd, dan ontstaat een gefragmenteerde heap. De objecten op de heap staan als het ware niet meer netjes naast elkaar. Er ontstaan gaten tussen de data. Als er nu een nieuw object aangemaakt moet worden dat niet in een van deze gaten past. Dan zal deze aan het einde van de heap worden geplakt. Dit is inefficiënt en zorgt voor een heap die veel meer ruimte in beslag neemt dan nodig is.



Figuur 2 Een gefragmenteerde Heap

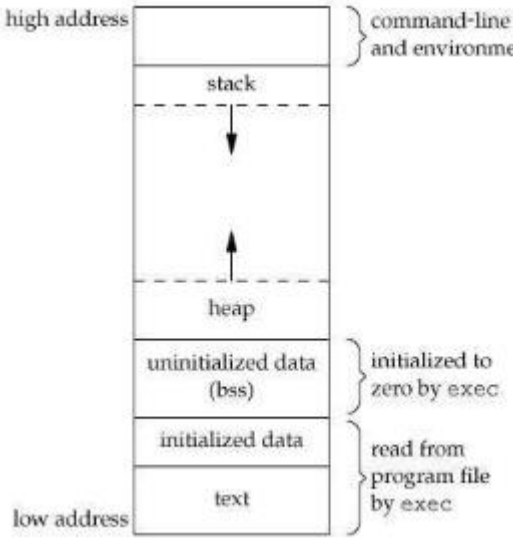
In figuur 7 is te zien dat de heap gefragmenteerd is, er zijn gaten tussen de data ontstaan. Als er nu een blok met een grootte van 2 moet worden toegevoegd, dan zal deze aan het einde van de heap op adres 15 moeten worden gezet.

3.4 Verschil stack en Heap

Kort gezegd is de stack snel en klein terwijl de heap groot en sloom is. Het zal altijd sneller zijn om data op te halen van de stack dan van de heap, hier is namelijk minder zoekwerk voor nodig. Het grote probleem aan de stack is dat deze snel vol zit en dat hier niet gemakkelijk ‘globale’ data op gezet kan worden. Hier tegenover staat wel dat het managen van het geheugen van de stack automatisch gaat, in tegenstelling tot de heap.

Als data veel moet veranderen dan is het gebruik van de stack optimaal. Denk hier dan aan bijvoorbeeld increment counters in loops. Voor grotere objecten kan beter de heap worden gebruikt. Zo komt de stack niet te vol te zitten.

3.5 Functie, voor- en nadelen van static memory



Figuur 3 geheugen opslag in een C programma (Ferres, 2007)

Een conventioneel C programma is opgedeeld in meerdere segmenten, de heap en de stack zijn twee van deze segmenten zoals te zien is in figuur 8. Static memory is data die aan het begin van het programma wordt aangemaakt. In java wordt dit via een 'static' keyword gedaan. Deze data wordt opgeslagen in de 'initialized data' en 'uninitialized data (bss)' segmenten.

Volgens (Ferres, 2007) word "All the global, static and constant data" opgeslagen in het 'initialized data segment'. En wordt "All the uninitialized data" opgeslagen in het 'uninitialized data segment'.

In de meeste object georiënteerde talen wordt het 'static' keyword gebruikt om globale data aan te maken binnen een class. In talen die minder strikt alle data in klassen willen hebben kan globale data vaak simpelweg als een variabele buiten een functie of klasse worden aangemaakt. Zo moet er in C# bijvoorbeeld een 'static' variabele binnen een klasse aangemaakt worden.

```
public class Data {
    public static int p = 0;
}
```

Terwijl in C en C++ de p variabele op zichzelf kan staan. Hier hoeft dan niet meer het 'static' keyword voor worden gebruikt, dit is wel weer nodig als p in een klasse of functie gedefinieerd zou zijn. Ook heeft 'static' een aantal andere mogelijke functies afhankelijk van waar het gebruikt wordt binnen C++ (cppreference.com, 2020).

```
int p = 0;
```

Het grote nadeel maar ook voordeel van static memory is dat het niet meer kan verdwijnen, het wordt aangemaakt als het programma start, en wordt pas verwijderd als het programma sluit. Dit kan natuurlijk ook een goed ding zijn aan de hand van waar het voor gebruikt moet worden. Bepaalde data in een programma veranderd nou eenmaal nooit, denk hier bijvoorbeeld aan constante waardes waar mee vergeleken moet worden of nummers als PI die altijd hetzelfde zullen zijn. Static memory kan ook geoptimaliseerd worden aangezien het nooit veranderd, dit kan niet met de heap.

3.6 Stratego

Als opdracht voor dit hoofdstuk moest er een stratego spel gebouwd worden. Het moest mogelijk zijn om stukken te verplaatsen, vijandige stukken aanvallen en het mocht niet

mogelijk zijn om de waardes van de stukken van de tegenstander te zien. Verder mocht er enkel met pointers gewerkt worden.

Om de waardes van de stukken op te slaan is er een enum gemaakt genaamd ‘PIECES’, hierin worden de namen van elk stuk gebonden aan hun waardes.

Het bord wordt opgeslagen als een struct met een twee dimensionale array van pointers genaamd ‘tiles’. Aangezien het mogelijk moet zijn om deze array op runtime een lengte te geven wordt deze opgeslagen als een ‘pointer naar een pointer naar een pointer’. De eerste pointer verwijst naar de ‘width’ array, de tweede pointer verwijst naar de ‘height’ array en de derde pointer verwijst naar de data die opgeslagen wordt in deze arrays.

```
typedef struct Board  {
    struct Tile ***tiles;
    int w;
    int h;
} Board;
```

Er zijn verscheidende functies gemaakt om het werken met het bord simpeler te maken. De ‘constructBoard’ functie is een constructor voor het bord, ‘seedBoard’ vult het bord met ‘Tile’ objecten, ‘move’ verplaatst een ‘Tile’ en ‘deleteBoard’ verwijderd alle data in het bord en het bord zelf.

```
Board *constructBoard();
Board *seedBoard(Board * board);
void move(Board * board, int fromX, int fromY, int toX, int toY);
void deleteBoard(Board * board);
```

Om het bord aan te maken moeten er drie malloc calls gedaan worden:

```
Board* board = malloc(sizeof(Board));
// code …
board->tiles = malloc(sizeof(Tile) * board->w);
for (int x = 0; x < board->w; x++) {
            board->tiles[x] = malloc(sizeof(Tile) * board->h);
            // code …
```

Een bord is gevuld met ‘Tile’ objecten. Elk van deze objecten heeft een ‘value’ en slaat op of het van de speler is.

```
typedef struct Tile {
    enum PIECE value;
    bool isPlayer;
} Tile;
```

Uiteindelijk worden ‘Tile’ objecten aangemaakt met de volgende constructor functie:

```
Tile *constructTile(PIECE value, bool isPlayer);
```

Door bovenstaande functies te gebruiken is het mogelijk om het stratego spel te simuleren.

5. Expressions and Assignment Statements (C++)

Binnen c++ is een expression: "… a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects" (Google Inc, 2020, p. 86). Deze expressions worden geclassificeerd als: glvalue, rvalue, lvalue, xvalue en prvalue. Deze classificatie zal niet verder gebruikt worden binnen dit document, voor meer informatie over de verschillende classificaties zie de c++ standaard pagina 87: (Google Inc, 2020, p. 87).

Expressions zien er binnen c++ als volgt uit:

expression operator expression.

Een expression kan dus dus meerdere sub expressions bevatten. Echter kan een expression ook een 'literal' value zijn oftewel een waarde als 0 of "Hello World!".

5.1 Relational and Boolean expressions

Boolean expressions zijn expressions met een boolean als uitkomst. Relational expressions vallen onder de boolean expressions maar gebruiken een set specifieke operatoren hier voor.

Relational expressions

Volgens de c++ standaard (Google Inc, 2020, p. 134) een relational expression als volgt gedefinieerd:

The relational operators group left-to-right. [Example: a<b<c means (a<b)<c and not (a<b)&&(b<c). — end example]
relational-expression:

compare-expression
relational-expression < compare-expression
relational-expression > compare-expression
relational-expression <= compare-expression
relational-expression >= compare-expression

Deze syntax komt erop neer dat een relational expression bestaat uit een compare expression, of een relational expression gevolgd door een operator (<, >, <=, >=) gevolgd door een compare expression. Een compare expression wordt gevormd door een serie verdere expressies die weer op hun beurt gevormd worden door expressies. Dit gaat door tot shift expressions, additive expressions, multiplicative expressions, pm expressions, cast expressions en unary expressions voor gekomen zijn. Om ervoor te zorgen dat dit hoofdstuk niet te lang wordt kan er van uit gegaan worden dat een compare expression een literal of een berekening is.

Hier uit is dus op te vatten dat een relational expression in c++ een expression is die een van de volgende operators gebruikt: < (minder dan), > (groter dan), <= (minder dan of gelijk aan) en >= (groter dan of gelijk aan). In code zou dit er op de volgende manier uit zien:

```
1 > 2; // false
2 < 3; // true
1 >= 1; // true
```

```
5 <= 3; // false
```

Elk cijfer in deze code kan natuurlijk voor een berekening of andere expressie waar een nummer uit voort komt worden vervangen.

Een relational expression kan volgens de c++ standaard enkel werken met ‘arithmatic’ (nummers), enumeration of pointer types. En zal altijd een boolean als resultaat hebben (Google Inc, 2020, p. 134).

```
1 > "hallo"; // error

3 < 3.14; // ok

p > q; // ok if p and q are pointer types and point to the same array
```

Als pointers vergeleken worden in een relational expression dan moeten zij altijd naar dezelfde array verwijzen, als dit niet zo is dan komt er undefined behaviour voor. In het geval dat dit wel zo is dan zal de index van de data binnen de array waar de pointers naar verwijzen worden vergeleken volgens de normale relational operator regels.

Boolean expressions

Boolean expressions zijn expressions die als uitkomst een boolean hebben. Hier onder vallen natuurlijk de relational expressions maar ook equality expressions, de logical and expression en de logical or expression. De syntax voor elk van deze expressions is volgens de c++ standaard als volgt:

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression
(Google Inc, 2020, p. 134)

logical-and-expression :
    inclusive-or-expression
    logical-and-expression && inclusive-or-expression
(Google Inc, 2020, p. 136)

logical-or-expression :
    logical-and-expression
    logical-or-expression || logical-and-expression
(Google Inc, 2020, p. 136)

Er wordt hier duidelijk dan een boolean expression een van de volgende operators gebruikt: ==, !=, &&, || of een van de relational expression operators. Deze zien er als volgt uit in code:

```
true || false; // true

false || false; // false

true && true; // true;

true && false; // false
```

```
1 == 1; // true
1 == 2; // false
1 != 1; // false
1 != 2; // true
```

De == en != operatoren werken volgens de c++ standaard met: “arithmetic, enumeration, pointer, or pointer-to-member type, or type std::nullptr_t” en hebben als resultaat een boolean (Google Inc, 2020, p. 134).

In de c++ standaard (Google Inc, 2020, p. 136) wordt er over de && operator het volgende gezegd:

> “*The && operator groups left-to-right. The operands are both contextually converted to bool (7.3). The result is true if both operands are true and false otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is false.*”

Hieruit is op te maken dat de && operator true als resultaat heeft als beide expressions waar het op werkt true zijn, anders zal deze op false uit komen. Als de eerste expression false is dan wordt de tweede niet nagekeken.

Net als de && operator wordt ook de || operator beschreven door de c++ standaard:

> “*The || operator groups left-to-right. The operands are both contextually converted to bool (7.3). The result is true if either of its operands is true, and false otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to true.*”

Echter in tegenstelling tot de && operator geeft || enkel true als bijde of een van dienst sub expressies true is. Ook wordt er net als bij de && operator enkel de tweede expressie uitgevoerd als de eerst false is.

5.2 Expression evaluation

Binnen C++ wordt er met ‘operator precedence’ (welke operator eerst wordt uitgevoerd) en ‘order of evaluation’ (In welke volgorde een expression wordt uitgevoerd) gewerkt.

Operator precedence

De volgorde van operators wordt niet in de c++ standaard aangegeven, deze wordt bepaald door de grammatica van de taal zelf. Zoals onder het kopje ‘boolean expressions’ te zien is staat de ‘equality expression’ (==, !=) hoger in de gramatica boom dan de ‘logical and’ (&&). Deze zal dus ook eerder uitgevoerd worden.

Table 1 Een deel van de C++ operator precedence tabel (cppreference.com, 2024)

Precedence	**Operator**	**Associativity**
2	a++ a— a() a[] . ->	Left to right
3	++a –a ! ~ (type) *a &a new delete	Right to left
5	a*b a/b a%b	Left to right
6	a+b a-b	Left to right
9	< <= > >=	Left to right
10	== !=	Left to right

14	&&	Left to right
15	||	Left to right
16	= += -= *= /= a?b:c	Right to left

Volgens de C++ standaard mag een c++ compiler de volgorde van uitvoering voor associatieve en commutatieve operatoren veranderen:

> "The implementation may regroup operators according to the usual mathematical rules only where the operators really are associative or commutative." (Google Inc, 2020, p. 86)

Het is binnen C++ dus niet gegarandeerd dat bij de expressie '1 + 2 + 3' eerst 1 + 2 of 2 + 3 wordt uitgevoerd. Dit is echter enkel zoals het gegarandeerd is dat een overflow niet een exception veroorzaakt.

Als een operator niet associatief of commutatief is dan zal diens 'associativity' worden gebruikt om een volgorde te bepalen. De = (asignment) operator is bijvoorbeeld 'right to left' associatief, dat betekend dat de expressie a = b = c als a = (b = c) zal worden uitgevoerd.

Order of evaluation

De order of evaluation binnen c++ is 'unspecified' deze is niet gegarandeerd (cppreference.com, 2024). Het is echter wel belangrijk om de order of evaluation niet met operator precedence of operator associativity te verwarren. Operator precedence gaat strict over de volgorde waar in onderdelen van een expression worden geparsed. Order of evaluation gaat over in welke volgorde expressions worden uitgevoerd.

In het volgende voorbeeld heeft operator associativity of precedence geen effect, de call operator '()' zal eerst worden uitgevoerd en vervolgens wordt de + operator uitgevoerd. Echter is het niet gegarandeerd dat a() of b() eerst wordt uitgevoerd.

```
int a() { std::cout << "1"; return 0; }
int b() { std::cout << "2"; return 0; }
a() + b();
```

Als a() eerst wordt uitgevoerd dan zal er 12 op de console verschijnen, is dit b() dan zal er 21 op de console staan.

5.3 Type casting

C++ kent expliciet en impliciet type conversies. Type casting valt onder expliciete conversie. Er zijn vijf soorten casts in C++: De C style cast, de const cast, de static cast, de dynamic cast en de reinterpret cast.

Static Cast

De meest veilige cast in C++ is de static cast. Deze cast kan niet om const heen casten (Google Inc, 2020, p. 116). Dit betekend dat de volgende code correct is:

```
bool b = true;
int i = static_cast<int>(b);
```

Maar dat het niet mogelijk is om van const af te komen met een static_cast:

```
const int i = 0;
```

```
int& b = static_cast<int&>(i); //error
```
Dit geeft de volgende error: Binding reference of type 'int' to value of type 'const int' drops 'const' qualifier.

Als het mogelijk is om een type impliciet te converteren dan zal een static_cast altijd werken. Dit werkt ook met ‘conversion constructors’, neem de klasse A en B. B heeft een conversie constructor met als parameter A:
```
class A {};
class B {
public:
    B(const A& a) {}
};
```
Aangezien a impliciet naar b converteert is het ook mogelijk om dit met een static cast te doen:
```
A a{};
B b = a; // ok
B b2 = static_cast<B>(a); // ok
```
Deze impliciete conversie gaat zelfs door meerdere conversies. Als er een extra C klasse wordt aangemaakt die impliciet vanuit B te converteren is:
```
class C {
public:
    C(const B& b) { }
};
```
Dan is het mogelijk om van een A klasse met een static cast een C klasse aan te maken.
```
A a{};
C c = static_cast<C>(a);
```

Reinterpret Cast

De reinterpet cast is speciaal gemaakt voor programmeurs die van gevaar houden. Een reinterpret cast kan niet de impliciete conversies doen waar een static cast toe in staat is. Het kan ook niet const weg casten (Google Inc, 2020, p. 118). De reinterpret cast wordt in plaats daarvan gebruikt om het type van een pointer te veranderen. Er worden geen checks gedaan om zekerheid te bieden dat het ook daadwerkelijk mogelijk is om de conversie uit te voeren. Zo is de volgende code compleet legaal:
```
std::string a = "hello world";
auto m = reinterpret_cast<std::map<int, int>*>(&a);
```
Hier wordt een string geherinterpreteerd als een hashmap. Als er nu geprobeerd wordt om de hashmap te gebruiken dan treedt er undefined behaviour op.

Const Cast

De const cast heeft de enkele functie om het mogelijk te maken een const variabele te kunnen veranderen. Door een pointer naar een const variabele (niet een const pointer naar een variabele!) te veranderen naar een normale pointer waardoor het mogelijk wordt de

waarde waar de pointer naar wijst te veranderen. De originele waarde veranderd hier niet door van waarde:

```
const int i = 1;
const int* p = &i;
int* r = const_cast<int*>(p);
(*r) = 2; // p == 2
```

In deze code wordt de pointer naar de constante variabele p gecast naar de normale pointer r. R is niet const dus mag veranderd worden, dit veranderd de waarde waar p naar wijst, maar niet de waarde i.

Dit kan volgens de standaard echter wel tot undefined behaviour leiden:

> "Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a const_cast that casts away a const-qualifier may produce undefined behavior" (Google Inc, 2020, p. 119)

Dynamic Cast

De dynamic cast converteert een pointer of een reference naar een klasse naar een ander type hoger op de overerving hierarchy. Neem de B en C klassen die beide van A overerven:

```
class A {};
class B : public A {};
class C : public A {};
```

Het is mogelijk om C omhoog te casten naar A maar niet om C zijwaarts naar B te casten:

```
C c{};
A* a = dynamic_cast<A*>(&c); // ok
B* b = dynamic_cast<B*>(&c); // error, C does not derive B
```

C Style Cast

De C Style Cast is de cast die C++ heeft geërfd van C, deze is aangepast om in de achtergrond de C++ casts die in de voor gaande kopjes besproken zijn aan te roepen. De C style cast heeft hierdoor ook dezelfde syntax als de cast die in C wordt gebruikt:

```
int i = 1;
bool b = (int)i;
```

Als een C style cast wordt uitgevoerd dan wordt er volgens de C++ standaard (Google Inc, 2020, p. 130) in de achtergrond de volgende volgorde van C++ style casts uitgevoerd:

1. Const Cast
2. Static Cast
3. Static Cast gevolgd door Const Cast
4. Reinterpret Cast
5. Reinterpret Cast gevolgd door Const Cast

Het wordt afgeraden om de C Style Cast te gebruiken in C++ aangezien deze geen mogelijkheid tot controle geeft aan de programmeur. Het is te makkelijk om per ongeluk een verkeerde cast te gebruiken. Het is mogelijk om const weg te casten, klassen naar compleet

ongerelateerde klassen te casten of zelfs om een pointer naar elke andere mogelijke pointer te casten.

5.4 Overload operators

Het is mogelijk om in C++ operatoren te 'overloaden', oftewel een klasse een specifieke implementatie van deze operatoren te geven. De syntax hiervoor is als Volgt:

```
class A {
public:
   A& operator+(const int& v) { return *this; }
};
```

Dit maakt het mogelijk om de addition (+) operator te gebruiken met objecten van het type A en integers:

```
A a{};
a = a + 1; // ok
a = a + "Hello World!"; // error
```

Volgens de C++ standaard volgen overloaded operatoren de normale regels voor syntax en order evaluation:

> "Overloaded operators obey the rules for syntax and evaluation order specified in 7.6, but the requirements of operand type and value category are replaced by the rules for function call." (Google Inc, 2020, p. 86).

Ook is het niet mogelijk om voor ingebouwde types de operatoren te overloaden:

> "overloading shall not modify the rules for the built-in operators, that is, for operators applied to types for which they are defined by this Standard" (Google Inc, 2020, p. 86)

Het is niet mogelijk om de '.', '.*', '::' en '?:' operatoren te overloaden (Google Inc, 2020, p. 338).

7. Subprograms & Implementing subprograms, Abstract Data Types and Encapsulation Constructs (C#)

Binnen dit hoofdstuk zullen er een aantal concepten binnen de C# taal worden uitgelegd.

7.1 Parameter-passing Methods

C# kent drie soorten types: Value, Reference en Pointer Types (Microsoft Corporation, 2024). Pointer types zijn enkel in zogenoemde 'unsafe' code beschikbaar en komen overeen met C/C++ pointers. Value types bevatten direct hun waarde, in tegenstelling tot reference types die een referentie naar een waarde bevatten. Klassen in C# zijn altijd reference types, terwijl structs en 'simple types' (C++ primitives) value types zijn.

De C# documentatie definieerd de verschillende typen parameters als volgt (Microsoft Corporation, 2024):

- *Pass by value* means **passing a copy of the variable** to the method.
- *Pass by reference* means **passing access to the variable** to the method.
- A variable of a *reference type* contains a reference to its data.
- A variable of a *value type* contains its data directly.

Er zijn dus vier manieren om een parameter aan een functie tegeven in C#: pass value by value, pass value by reference, pass reference by value en pass reference by reference. In code zien die er ieder op deze manier uit:

```
public void PassValueTypeByValue(int i) {} // pass value type by value
public void PassValueTypeByReference(ref int i) {} // pass value type by reference
public void PassReferenceTypeByValue(Class c) {} // pass reference type by value
public void PassReferenceTypeByReference(ref Class c) {} // pass reference type by reference
```

De C/C++ versie van elk van deze methodes is als volgt:

```
void passValueTypeByValue(int i) {} // pass primitive by value
void passValueTypeByReference(int& i) {} // pass primitive by reference
void passReferenceTypeByValue(int* c) {} // pass pointer by value
void passReferenceTypeByReference(int& c) {} // pass primitive by reference
```

Pass Value Type by Value

Als een parameter een value type zoals een integer, boolean of struct is en deze zonder het 'ref' of 'out' keyword wordt gebruikt, dan is er sprake van pass value type by value. De waarde van de parameter wordt gekopieerd als de functie begint. Het is niet mogelijk om de originele waarde waar mee de functie is aangeroepen te veranderen.

Dit werkt hetzelfde als in C++, echter kent C++ niet het verschil tussen value en reference types, als de gegeven parameter pass by value is dan zal de gegeven waarde altijd gekopieerd worden of dit nu een object of primitive is.

Pass Value Type by Reference

Een value type welke als een reference parameter aan een functie wordt gegeven zal niet gekopieerd worden. Als deze binnen de functie veranderd wordt dan zal de waarde buiten de functie waar de functie mee is aangeroepen ook veranderen.

Neem de functie Foo welke een integer (value type) als reference parameter binnen krijgt en vervolgens deze naar 1 veranderd.

```
void Foo(ref int i) { i = 1; }
```

Als Foo vervolgens aangeroepen wordt met een reference naar de variabele ‘value’ (let op het gebruik van het ‘ref’ keyword):

```
int value = 0;
Foo(ref value);
bool b = value == 0; // false
```

Dan is vervolgens de waarde die aan de functie gegeven is veranderd.

Pass Reference Type by Value

Als een reference type (class of array) als value (zonder het ref keyword) als parameter wordt gebruikt, dan worden rerassignments van de parameter niet buiten de functie gereflecteerd, aangezien de reference gekopieerd wordt. Echter worden veranderingen binnen de klasse zelf wel gereflecteerd.

```
void Bazz(Class c) {}
```

Als een member variabele van c wordt veranderd binnen Bazz dan zal dit buiten de functie ook zo zijn veranderd.
Dit komt overeen met hoe pointers in C++ werken. De pointer zelf wordt gekopieerd en zal dus niet na een assignment ook buiten de functie veranderen, echter kunnen waardes waar de pointer naar wijst wel veranderd worden:

```
void bazz(Class * c) {} // pass pointer by value
void bazz(int* c) {} // pass pointer by value
```

Pass Reference Type by Reference

Een reference type welke gepast word als reference werkt precies het zelfde alsof er een value type by reference word gebruikt. De parameter kan veranderd worden binnen de functie en dit zal buiten de functie gereflecteerd worden. Hier voor zal dus ook het ‘ref’ keyword gebruikt moeten worden:

```
public void Bar(ref Class c) {}
```

7.2 Stack & Recursion Aspects

Simpel gezecht is recursion al seen functie zichzelf aanroept, dit kan direct zijn maar ook met meerdere andere functies als tussenstappen. In C# ziet dit er als volgt uit:

```
public int Factorial(int num) {
        return num * Factorial(num - 1);
}
```

Zoals wordt uitgelegd in hoofdstuk ‘3.1 Werking en doel van de stack’ wordt er elke keer als er een functie wordt aangeroepen een nieuwe ‘stack frame’ aan de stack toegevoegd. Dit gebeurt nog steeds in recursieve functies. Hieruit volgt dat als een recursieve functie geen eind conditie heeft deze eeuwig door zal gaan waardoor de stack vol komt te zitten. Dit is een stack overflow error.

Bovenstaande functies hebben geen eind conditie en zullen dus het programma laten crashen. Dit is op te lossen met de volgende toevoeging:

```
int Factorial(int num) {
    if (num == 0)
        return 0;
    return num * Factorial(num - 1);
}
```

Als de parameter num nu de waarde ‘0’ bereikt dan komt de functie tot een eind.

7.3 Closures

Het is mogelijk om bij het gebruik van een lambda functie variabelen buiten de functie aan te passen. Als er een variabele van buiten de lambda binnen de lambda wordt gebruikt, dan slaat C# een reference naar deze variabele op (shaikh, 2024). Binnen de lambda wordt deze variabele dan ook als een ‘ref’ behandeld, zie ‘7.1 Parameter-passing Methods’. Een lambda kan in C# op een van de volgende manieren aangemaakt worden:

```
Action l = delegate () {};
Action l2 = () => { };
```

In de volgende code wordt een closure gebruikt om de ‘i’ variabele binnen de lambda genaamd ‘closure’ te veranderen:

```
int i = 1;
var closure = () => { i++; };
closure();
```

Als nu de functie ‘closure’ wordt aangeroepen dan zal de expressie ‘i++’ worden uitgevoerd op een reference naar de i variabele. De waarde van ‘i’ buiten de lambda zal dus ook veranderd zijn.

7.4 User-defined Abstract Datatypes, Parameterized Abstract Datatypes en Opdracht

Het is in C# mogelijk om een ‘anonieme’ representatie van een type te gebruiken, er is dan niet bekend wat dit type ook daadwerkelijk zal zijn. Dit is vooral handig om datastructuren te maken die alle soorten type data moeten kunnen opslaan. Deze vorm van abstractie wordt binnen C# ‘generics’ genoemd (Microsoft Corporation, 2024).

Generics

Om een generic klasse aan te maken in C# hoeven enkel ‘<>’ haakjes na de klassen naam gezet te worden, hier na kunnen zoveel type parameters als nodig binnen de haakjes gezet worden en gebruikt worden binnen de klasse:

```
class GenericClass<T> {
    public void WithParameter(T t) {}
    public T WithReturnType(T t) { return t; }
    public T variable;
}
```

Het is ook mogelik om enkel een functie generic te maken, dit gebeurt met dezelfde syntax:

```
void WithRestriction<T>() where T : IComparable {}
```
De ‘where’ zorgt er hiervoor dat het type T altijd van IComparable moet overerven.

Het is nu mogelijk om een ‘GenericClass’ aan te maken met een type parameter welke gereflecteerd wordt in de functies binnen de klasse:
```
var cInt = new GenericClass<int>();
var cString = new GenericClass<string>();

cInt.WithParameter(1); // ok
cString.WithParameter(1); // error, Argument type 'int' is not assignable to parameter type 'string'
```
De restriction via het ‘where’ keyword werkt ook op deze manier:
```
WithRestriction<string>(); // ok, string is comparable
WithRestriction<Object>(); // error, The type 'object' must be convertible to 'System.IComparable' in order to use it as parameter 'T'
```

Opdracht
Onder dit hoofdstuk valt natuurlijk ook een opdracht. Voor deze opdracht zijn een Stack, een Queue en een binary search tree geimplementeerd in C# met generics. De code voor deze opdracht is te vinden in de bijlage onder ‘C#_abstract_datatypes’. Voor de opdracht zijn een aantal van de besproken technieken gebruikt. Zo zijn generics en recursion in de volgende functie gebruikt:
```
int? HighestValueIndex(ref BinarySearchTreeData<T> tree, int root) {
// code …
var child = HighestValueIndex(ref tree, right);
// code …
}
```
En wordt een closure in de Delete functie van de binary search tree gebruikt:
```
var values = tree.Items.Where(item => item.CompareTo(value) != 0).ToArray();
```
Er wordt hier een lambda mee gegeven aan de ‘Where’ functie welke de ‘value’ variabele gebruikt, dit is een variabele van buiten de functie en dit teld dus als een closure.

8. Concurrency (Go)
Concurrency is een manier om binnen een programma taken tegelijkertijd te laten draaien. Dit wordt ook wel asynchonous programmeren genoemd.

8.1 Subprogram-Level Concurrency Discussion
In veel oude talen is een ‘sub-program’ een ander woord voor een functie of een method. Concurrency kan op het niveau van een enkele functie gebruikt worden. Dit wordt veel gebruikt binnen Javascript en Go. Door het toe passen van concurrency op deze manier wordt het makkelijker om programma’s ‘responsive’ te maken, een user interface zal altijd beter werken als deze niet bevriest als het programma data aan het verwerken is.

Bij het gebruik van concurrency ligt natuurlijk wel altijd gevaar op de loer. Aangezien het onmogelijk is om te weten welk sub programma welke taak wanneer af heeft zal het ook niet mogelijk zijn om de gedeelde data te synchroniseren. Hier zal bij het gebruik van concurrency altijd rekening mee gehouden moeten worden.

Binnen Go is het mogelijk om een sub progam concurrent aan te roepen met het 'go' keyword. Na dit keyword moet een functie aangeroepen worden, deze functie wordt vervolgens in een aparte 'go routine' uitgevoerd terwijl de main routine (het hoofdprogramma) door loopt. Dit is te zien in de volgende code:

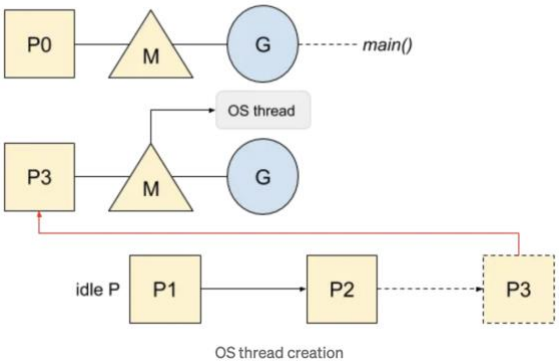
```
go func() {
    for {
        serverLoop();
    }
}();
```

Hier wordt het 'go' keyword gebruikt met een lambda. Deze lambda bevat een loop waar binnen een functie aangeroepen wordt. De lambda wordt direct aangeroepen en de

8.2 Thread Concepts and Implementation

Een thread is een 'execution context', dit is alle data die een cpu nodig heeft om een instructies uit te voeren. Aangezien een thread instructies uit kan voeren, wordt het ook mogelijk om binnen een programma een aparte thread te draaien. Deze wordt dan 'tegelijkertijd' met het hoofdprogramma door de cpu uitgevoerd.

Nu kan een CPU core natuurlijk eigenlijk maar een ding tegelijkertijd, echter is het toch mogelijk om meerdere threads op dezelfde core te laten draaien. De CPU is snel genoeg dat als de instructies van meerdere threads door elkaar worden gedraaid het dan lijkt alsof de threads tegelijkertijd draaien.



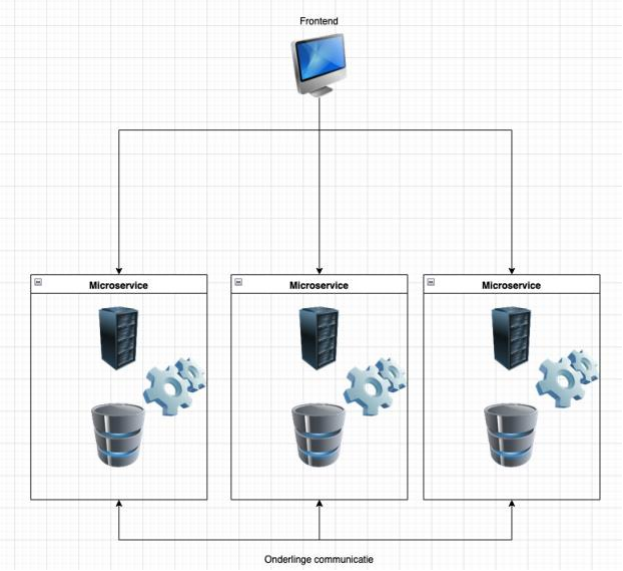
Figuur 4OS thread creation (Blanchon, 2024)

Binnen Go wordt er met het P M G model gewerkt. G staat voor een goroutine, M is een thread en P is een processor. Als er een goroutine start wordt er eerst een P (processoren) voor de routine geindexeerd. Vervolgens wordt er een P wakker gemaakt waar, deze P maakt een M (thread) aan waar de G (goroutine) op kan werken.

Het blijkt dus dat goroutines eigenlijk gewoon software threads zijn. Dit is ook de reden dat goroutines zo veel sneller op te starten zijn dan traditionele threads binnen bijvoorbeeld java.

8.3 Co-operative Processing

Co-operative processing kan vergeleken worden met een thread op een aparte computer draaien. Dit is natuurlijk niet een aparte thread, maar dezelfde problemen komen kijken bij een netwerk van meerdere computers die samenwerken als bij multi threading. Co-operative processing komt voor bij bijvoorbeeld microsystem architectuur, en als er communicatie tussen de frontend en backend van een webapplicatie plaats vind.



Bij het gebruik van een microservice architectuur wordt er onderling tussen services gecommuniceerd. Er wordt ook tussen de front-end en de verschillende microservices gecommuniceerd. Deze communicatie is asynchroon en heeft dus dezelfde voordleen en nadelen die bij het gebruik van multithreading komen kijken.

Figuur 5 Een simpele microservice architectuur

Voor de opdracht die bij dit hoofdstuk hoort is er een webshop in Go gemaakt. Deze webshop bestaat uit een front-end en een backend die asynchroon met elkaar communiceren. Op de backend draait er een aparte go routine die de voorraad van producten die binnen de webshop te krijgen zijn bijhoudt. Deze voorraad is via HTTP aanvragen in te zien en te veranderen. Via de volgende code worden de HTTP aanvragen geregeld:

```
router.GET("/products", getProducts);
router.GET("/products/:name", getProductByParamName);
router.POST("/products/:name", buyProduct);
```

En de webshop loop wordt in een goroutine gedraaid:

```
go func() {
    for {
        serverLoop();
    }
}();
```

Vervolgens draaid in de frontend een loop die producten koopt:

```
for {
    loop();
    sleepAmount := calculateNextSleepAmount();
    time.Sleep(sleepAmount);
}
```

8.4 Synchronization methods

Als threads data delen dan is het nodig dat deze threads gesynchroniseerd werken, anders is het mogelijk dat er nare problemen als race conditions of deadlocks voorkomen.

WaitGroup

Een manier om te wachten tot meerdere threads klaar zijn met hun taak is een 'WaitGroup'. Dit is een object die bijhoudt of threads al klaar zijn. Een WaitGroup heeft het aantal threads nodig waar die op moet wachten:

```
var wg sync.WaitGroup
wg.Add(1);
go func()  {
    defer wg.done();
}();
wg.Wait()
```

Door in elke thread wg.done gebruiken wordt er aan de WaitGroup gesignaleerd dat er een thread klaar is. wg.Wait kan vervolgens in de main thread aangeroepen worden om die thread te laten wachten tot alle threads binnen de WaitGroup klaar zijn.

Channels

Een channel kan gezien worden als een FIFO (first in first out) queue die wordt gebruikt door go routines om te communiceren. Een channel kan waardes toevoegen aan de queue, en uit de queue halen. Dit gaat via de volgende syntax:

```
c := make(chan int); // make a channel
c <- 1; // add a value to the queue
value := <-c; // read from the queue
```

Door dat het toevoegen en lezen uit een channel veilige operaties zijn voor threads om te doen, wordt het mogelijk om op een veilige manier data tussen threads te versturen.

Mutex

Een Mutex is een van de standaardmethoden die gebruikt worden om threads veilig te maken. Go ondersteund deze ook. Als een Mutex 'gelocked' is dan zal deze een andere thread die deze probeerdt te 'locken' laten wachten tot de Mutex weer 'geunlocked' is. Een Mutex wordt binnen Go op de volgende manier gebruikt:

```
var productsMutex sync.Mutex; // create a mutex
```

```
productsMutex.Lock(); // lock the mutex
// .. code
productsMutex.Unlock(); // unlock the mutex
```

9. Reflectie

Tijdens het schrijven van dit verslag heb ik ontzettend veel geleerd. Het onderzoeken van verschillende programmeertalen is altijd al een interesse van mij geweest dus ik vond het erg leuk om dit voor een schoolvak te kunnen doen. Af en toe had ik er moeite mee dat het

document maar dertig pagina's mocht zijn, af en toe had ik graag dieper op de taal in willen gaan.

Het hoofdstuk waar ik het meeste van geleerd heb is zeker hoofdstuk: 5. Expressions and Assignment Statements (C++). Tijdens het schrijven van dit hoofdstuk was ik onder de indruk dat er 'goede' bronnen gebruikt moesten worden. Vervolgens heb ik voorbeelden uit de C++ standaard gebruikt. Dit was erg leuk om te onderzoeken en om af en toe compleet niets van te snappen (l, r, gl en x values etc.). Helaas heeft deze diepere uitleg er wel voor gezorgd dat dit hoofdstuk mogelijk iets te lang is geworden, waardoor de andere hoofdstukken wat verkort zijn.

De lessen waren leuk om bij te wonen, deze waren af en toe wat lang maar de stof die behandeld is was erg interessant. Vooral de les door Heiko over Scala was erg leuk, ik vond het dan ook erg jammer dat er geen hoofdstuk over functioneel programmeren geschreven moest worden.

Al met al vond ik dit een leuk vak om te volgen, de richtlijnen over wat er nou precies in het verslag moest waren af en toe iet wat onduidelijk. Maar dit heeft er ook voor gezorgd dat het verslag interessanter was om te maken. Het was niet nodig om over precies het zelfde te schrijven als medestudenten.

Literatuur

unc.edu. (sd). *Lambda*. Opgehaald van cs.unc.edu: https://www.cs.unc.edu/~stotts/723/Lambda/overview.html

cse.psu.edu. (sd). *Brief history of functional programming*. Opgehaald van https://www.cse.psu.edu/~gxt29/historyOfFP/historyOfFP.html

McCarthy, J. (1979). *History of Lisp.* Artificial Intelligence Laboratory Stanford University.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*.

math.bas.bg. (sd). *fp.html*. Opgehaald van math.bas.bg: http://www.math.bas.bg/bantchev/place/fp.html

depaul.edu. (sd). *ichu/csc447/notes/wk4/ps.html*. Opgehaald van condor.depaul.edu: https://condor.depaul.edu/ichu/csc447/notes/wk4/ps.html

Cartwright, C. (sd). *Principles of Programming Languages Lecture 16 Boxes as Values and Call-by-Reference*. Opgehaald van cs.rice.edu: https://www.cs.rice.edu/~javaplt/411/21-summer/Lectures/16.pdf

jargon300/thunk.html. (sd). Opgehaald van ftp.informatik.rwth-aachen.de: http://ftp.informatik.rwth-aachen.de/jargon300/thunk.html

Wijngaarden, A. V. (1975). Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, 1-236.

Koster, C. (sd). *A SHORTER HISTORY OF ALGOL68.* Nijmegen: University of Nijmegen.

LINDSEY, C. H. (sd). PAPER:A HISTORYOF ALGOL 68.

Williams, A. (2019, September 11). *lambdas-for-c-sort-of*. Opgehaald van hackaday.com: https://hackaday.com/2019/09/11/lambdas-for-c-sort-of/

cppreference.com. (2023, october 31). *lambda*. Opgehaald van en.cppreference.com: https://en.cppreference.com/w/cpp/language/lambda

cppreference.com. (2023, september 18). *w/c/language/behavior*. Opgehaald van cppreference.com: https://en.cppreference.com/w/c/language/behavior

cppreference.com. (2023). *w/cpp/memory*. Opgehaald van cppreference.com: https://en.cppreference.com/w/cpp/memory

Ferres, L. (2007). *Memory management in C: The heap and the stack.* Universidad de Concepción, Department of Computer Science.

cppreference.com. (2020, April 10). *w/cpp/keyword/static*. Opgehaald van cppreference.com: https://en.cppreference.com/w/cpp/keyword/static

Google Inc. (2020). *Working Draft, Standard for Programming Language C++.* Google Inc.

cppreference.com. (2024, 1 5). */w/cpp/language/operator_precedence*. Opgehaald van cppreference.com: https://en.cppreference.com/w/cpp/language/operator_precedence

cppreference.com. (2024, 1 5). *w/cpp/language/eval_order*. Opgehaald van cppreference.com: https://en.cppreference.com/w/cpp/language/eval_order

Microsoft Corporation. (2024, 01 07). */en-us/dotnet/csharp/language-reference/keywords/method-parameters*. Opgehaald van learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters

Microsoft Corporation. (2024, 01 07). *en-us/dotnet/csharp/language-reference/language-specification/types*. Opgehaald van learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types

shaikh, i. (2024, 01 08). *the-magic-of-c-closures*. Opgehaald van medium.com: https://medium.com/swlh/the-magic-of-c-closures-9c6e3fff6ff9

Microsoft Corporation. (2024, 01 08). *fundamentals/types/generics*. Opgehaald van learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics

Blanchon, V. (2024, 01 11). *go-goroutine-os-thread-and-cpu-management-2f5a5eaf518a*. Opgehaald van medium.com: https://medium.com/a-journey-with-go/go-goroutine-os-thread-and-cpu-management-2f5a5eaf518a

Warburton, R. (2024, 01 13). *Java-8-Lambdas-A-Peek-Under-the-Hood/*. Opgehaald van infoq.com: https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood/

Code Maze. (2024, 01 13). *lambda-expressions-in-csharp/*. Opgehaald van code-maze.com: https://code-maze.com/lambda-expressions-in-csharp/