

Model for computers arbejdshukommelse

Herunder angives en model for computerens arbejdshukommelse til anvendelse i fagene Programmering C og Programmering B i gymnasiet.

Modellen tilstræbes så simpel at den er anvendelig for nye programmører.

Arbejdshukommelse i computer

Applikationer og operativ-system anvender computerens arbejdslager, også kaldet RAM. Når programmer kører opstår behov for lagring af selve programmet og de data programmet arbejder med i RAM.

RAM står for Random Access Memory, og indikerer at vilkårlige (random) steder i hukommelsen kan tilgås direkte - i modsætning til magnetbånd hvor båndet skal spoles frem eller tilbage før man kan tilgå data.

Moderne computeren er i skrivende stund typisk udstyret med 8GB eller 16GB RAM.

Arbejdshukommelsen må ikke forveksles med harddiskens lager som typisk er 256GB eller mere.

Sammenlignet med at skrive til og læse fra en harddisk så er arbejdshukommelsen meget hurtigere, selvom forskellen dog er blevet mindre de senere år med udbredelsen af SSD harddiske.

Nu er vi klar til at opstille en model for computerens arbejdshukommelse.

Til højre vises en model for arbejdshukommelsen.

Arbejdshukommelsen er opdelt i en række rum.

I computere arbejder den mindst mulige lager-enhed med to mulige tilstande, slukket eller tændt. Det gør man også i det binære talsystem, som kun har to cifre, 1 og 0. Et binært ciffer kan altså være enten 1 eller 0. Vi benævner den mindst mulige lager-enhed for bit fra det engelske "binary-digit".

Computere har så bare rigtig mange bits i deres arbejdslager, f.eks. cirka 8 milliarder bits i et 8GB RAM lager.

I stedet for at anvende hver enkelt bit for sig, tilgår man oftest lageret i klumper af 8bit (her kaldet en byte, 1 byte = 8bit), 16bit, 32bit eller 64bit. Det er relateret til når vi anvender et 32bit eller 64bit operativ-system.

I den aktuelle model for arbejdshukommelsen vælger vi at der kan være **en byte i hver lagercelle**.

Tallene til venstre for hver celle kaldes en adresse - men bliver også ofte kaldt en **reference** eller en **pointer**. Man kan sige at adressen går det muligt at udvælge en lagercelle.

Når processoren ønsker at anvende en lagercelle foregår det ved at den vælger hvilken adresse der skal være aktiv, svarende til hvilken skuffe der skal åbnes, hvorefter den kan skrive til eller læse fra lagercellen.

0000010007

0000010006

0000010005

0000010004

0000010003

0000010002

0000010001

0000010000

0000000004

0000000003

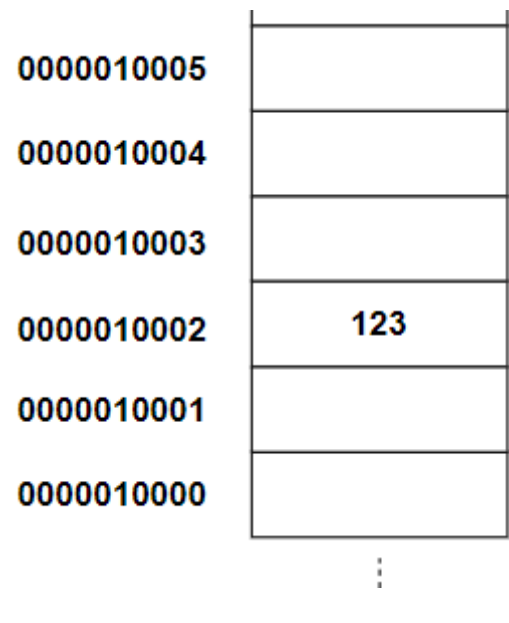
0000000002

0000000001



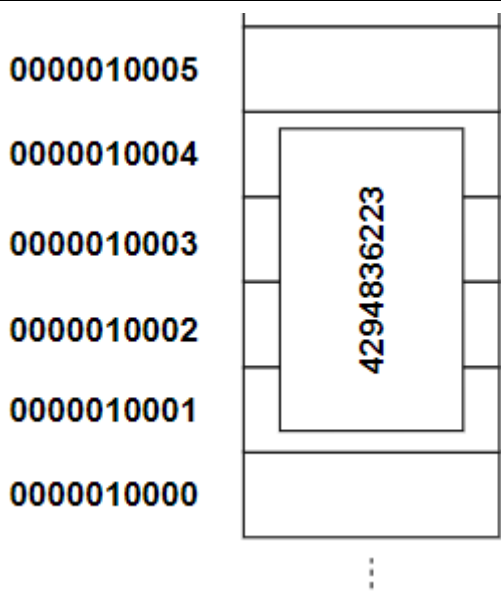
Lagring af en byte

En byte kan indeholde tallene fra 0 til og med 255.

Lad os sige at vi ønsker at gemme tallet 123 i lagercellen med adressen 0000010002. Da tallet kan være i en byte vælger vi at lagre den som data typen "byte". Det foregår lynhurtigt i følgende trin: Først sætter processoren den aktive adresse til 0000010002. Dernæst kopieres tallet 123 til lagercellen. Resultatet er vist til højre:	 The diagram shows a vertical stack of memory cells. To the left of each cell is its address in binary: 0000010005, 0000010004, 0000010003, 0000010002, 0000010001, and 0000010000. The cell at address 0000010002 contains the decimal value 123. Below the 0000010000 address, there are three vertical dots indicating further memory cells.
---	--

Lagring af heltal

Et heltal lagres normalt ved at anvende 4 byte.

Lad os sige at vi ønsker at gemme tallet 4294836223 i lagercellen med adressen 0000010001. Tallet kan ikke være i en byte (0-255), og vi vælger derfor data-typen "heltal", som fylder 4 bytes. Lagring foregår lynhurtigt i følgende trin: Først sætter processoren den aktuelle adresse til 0000010001. Dernæst kopieres tallet 4294836223 til lagercellen. Resultatet er vist til højre: Bemærk: Da data-typen heltal optager 4 byte lagres værdien "henover" 4 lagerceller (hver på 1 byte).	 The diagram shows a vertical stack of memory cells. To the left of each cell is its address in binary: 0000010005, 0000010004, 0000010003, 0000010002, 0000010001, and 0000010000. A large box labeled with the decimal value 4294836223 spans the four cells at addresses 0000010004, 0000010003, 0000010002, and 0000010001. Below the 0000010000 address, there are three vertical dots indicating further memory cells.
---	---

Lagring af kommatall

Kommatal kan lagres med forskellig præcision. Jo flere bytes der anvendes jo større præcision. F.eks. arbejder C# sproget med **float (4 byte)**, **double (8 byte)** og **decimal (16 byte)**.

Lad os sige at vi ønsker at gemme tallet 3,141593 som et kommatall af data typen float i lagercellen med adressen 0000010001. Det foregår lynhurtigt i følgende trin: Først sætter processoren den aktuelle adresse til 0000010001. Dernæst kopieres tallet 3,141593 til lagercellen. Resultatet er vist til højre: Da et kommatall af typen float optager 4 byte lagres værdien "henover" 4 lagerceller (hver på 1 byte).	The diagram illustrates the memory layout for storing a float value. It shows a vertical column of memory cells. To the left of the cells are their addresses: 0000010005, 0000010004, 0000010003, 0000010002, 0000010001, and 0000010000. A large rectangle labeled '3,141593' spans the four cells corresponding to addresses 0000010001, 0000010002, 0000010003, and 0000010004. The cells at 0000010005 and 0000010000 are empty. Below the stack of cells, there is a vertical ellipsis '...' indicating further memory cells.
--	--

Lagring af et tegn

Et tegn er f.eks. et bogstav fra vores alfabet 'a', et 'b', et 'Å' eller det kan være et specialtegn, f.eks. '%', '*', '+', m.m. fl.

I computeren kan man "kun" lagre tal, så man har brug for en omsætningstabel, et såkaldt tegnsæt.

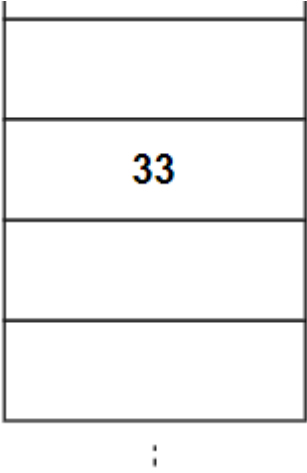
De første mange år af computerens tidsalder lagrede man tegn i en enkelt byte, dvs. som tal fra 0 til og med 255. Herunder vises uddrag fra ASCII tegnsættet (for de komplette ASCII tabeller se appendix):

Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	ESC	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32		58	:	84	T	110	n
7	BEL	33	!	59	;	85	U	111	o
8	BS	34	"	60	<	86	V	112	p
9	HT	35	#	61	=	87	W	113	q
10	LF	36	\$	62	>	88	X	114	r
11	VT	37	%	63	?	89	Y	115	s
12	FF	38	&	64	@	90	Z	116	t

Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr
13	CR	39	'	65	A	91	[	117	u
14	SO	40	(	66	B	92	\	118	v
15	SI	41	)	67	C	93	]	119	w
16	DLE	42	*	68	D	94	^	120	x
17	DC1	43	+	69	E	95	_	121	y
18	DC2	44	,	70	F	96	`	122	z
19	DC3	45	-	71	G	97	a	123	{
20	DC4	46	.	72	H	98	b	124	
21	NAK	47	/	73	I	99	c	125	}
22	SYN	48	0	74	J	100	d	126	~
23	ETB	49	1	75	K	101	e	127	DEL
24	CAN	50	2	76	L	102	f		
25	EM	51	3	77	M	103	g		

For at lagre et 'a' i hukommelsen anvendes talværdien 97. For at lagre et 'A' anvendes talværdien 65. For at lagre et 'Æ' anvendes talværdien 146.

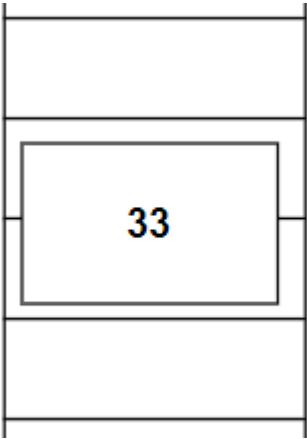
Hvilken talværdi skal du anvende hvis du ønsker at gemme et @ tegn?

Lad os sige at vi ønsker at gemme et udråbstegn (!) i lagercellen med adressen 0000010002. Det foregår lynhurtigt i følgende trin: Først sætter processoren den aktuelle adresse til 0000010002. Dernæst kopieres tallet 33 til lagercellen. Resultatet er vist til højre:	0000010003 0000010002 0000010001 0000010000 
---	--

Et problem med ovenstående er at ASCII tabellen kun kan rumme et lille antal alfabeter. Dermed får man problemer med deling af tekstdokumenter internationalt.

Derfor opfandt man Unicode og utf8. Unicode og Utf8 er store tegnsæt som vi ikke kommer nærmere ind på her. Det eneste vi skal vide er at [C# anvender 16-bit, dvs. 2-byte, Unicode til lagring af tegn i data typen char.](#)

Vi forbedrer dermed vores model af lagring af et udråbstegn (!), i C# til:

Lad os sige at vi ønsker at gemme et udråbstegn (!) i lagercellen med adressen 0000010002. Det foregår lynhurtigt i følgende trin: Først sætter processoren den aktuelle adresse til 0000010002. Dernæst kopieres tallet 33 til lagercellen. Resultatet er vist til højre: Læg mærke til at tegnet nu fylder to byte.	0000010004 0000010003 0000010002 0000010001 
--	--

... fortsættes ...

Opsummering

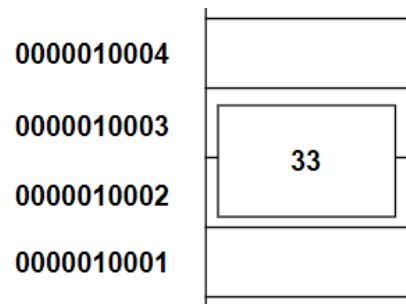
To pointer:

Pointe 1: Når vi lagrer noget i hukommelsen, f.eks. tallet 33, så gør computeren det ved at anvende en adresse, til at pege på en adressecelle, hvorefter den kan overføre tallet 33 til den aktive lagercelle.

For korrekt anvendelse af det lagrede, så skal vort program vide hvilken datatype er lagret i den aktuelle celle.

Hvis vi ikke kender datatypen, så kan lager-situationen til højre fortolkes vidt forskelligt.

Er det et heltal, en byte, tegnet '!' eller noget helt andet?



Pointe 2: Lagring af data og kendskab til datatype skal følges ad! Et lagret tal har ingen betydning før vi også kender datatypen.

... fortsætter ...

Variabelnavne i stedet for adresser

Af flere grunde er det [meget besværligt at arbejde med adressernes talværdier](#), såsom 0000010002, når der skrives computerprogrammer.

Langt de fleste programmeringssprog gør det også muligt helt at undgå adresserne - og [anvender i stedet variabelnavne](#).

Nu hvor vi kender de underliggende mekanismer, kan vi simplificere vores hukommelses-model yderligere. Vi gør det i tre trin:

Ændring	Før	Efter simplificering
Trin 1: Dels ved at undlade adresserne og i stedet tilføje variabelnavn og datatype. Til højre har vi brugt variablen med navnet myExMark som har data typen char.		
Trin 2: vi undlader hvor mange byte de enkelte datatyper anvender, og får resultatet til højre.		
Trin 3: Vi går væk fra princippet om at lagercellerne skal ligge på en lang akse, nede fra og opefter. I stedet kan lagercellerne placeres frit, se eksemplet til højre. I eksemplet anvender programmet variablen myExMark af data typen char. Og programmet anvender variablen med navnet pi af data typen float.		

Værdi-datatyper og reference datatyper

Alle moderne programmeringssprog anvender direkte eller indirekte begreberne værdi-datatyper og reference-datatyper. C# arbejder direkte med begreberne.

Men hvad betyder det i forhold til vores hukommelsesmodel?

Værdi-datatyper

Værdi-datatyper (eng: value types) indeholder selv data.

I C# gælder det for datatyperne: `int`, `float`, `double`, `decimal`, `bool`, `char`, `enum`, `struct` og `tuple` (avanceret: det gælder også hvis de gøres "nullable").

Værdi-datatyper kaldes også værdi-datatyper fordi deres værdi kopieres når man anvender tildeling med lighedstegnet. Eksempel:

```
int a, b;  
a = 2; b = 3;  
a = b;
```

Her kopieres værdien i b til a, så a til slut indeholder værdien 3.

Reference-datatyper

Reference-datatyper (eng: reference types) indeholder ikke selv data, men indeholder i stedet en reference (dvs. hukommelses-adresse), der peger på hvor de egentlige data er lagret.

I C# gælder det for datatyperne: `string`, `array`, `list`, `object`, `class`, `interface`, `delegate`, `record` og `dynamic`.

Reference-datatyper kaldes reference-datatyper fordi deres reference (adresse) kopieres når man anvender tildeling med lighedstegnet. Eksempel:

```
string str1, str2;  
str1 = new string("Tekst1");  
str2 = str1;
```

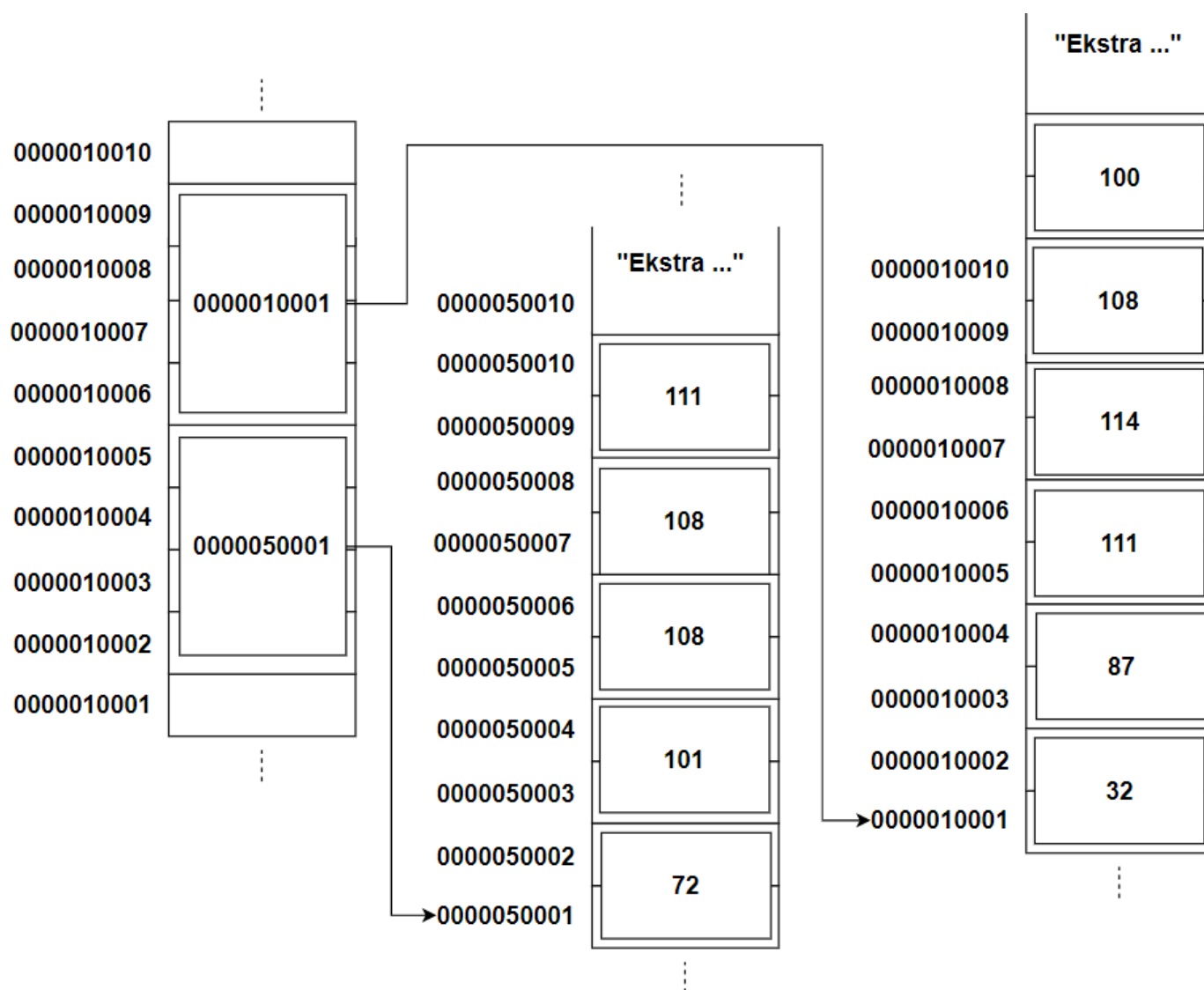
Her kopieres adressen som str1 peger på over i str2.

... fortsættes ...

Hukommelsesmodel for string (tekst)

De eksempler vi har set på i vores hukommelsesmodel indtil nu har alle været værdi-datatype. Nu viser vi et eksempel med to strenge, dvs. tekst. C# gemmer strenge/tekst i datatypen string som er en reference datatype.

Herunder vises model for lagring af to string-variable som henholdsvis er lagret på adressen 0000010002 og på adressen 0000010006.



Ovenstående, noget avancerede udgave af vores hukommelsesmodel, viser at variablene for strengene, i søjlen til venstre, i virkeligheden ikke indeholder strengene, men kun indeholder adresser på hvor strengene faktisk er lagret.

I C# er data typen string i virkeligheden en klasse. Instanser af string er dermed objekter. Med objekterne følger der, ud over tegnene i strengen, "noget ekstra", som vi ikke kommer ind på her, men det er vist ovenfor med "Ekstra ...".

Kan du finde ud af hvilken tekst de to strenge indeholder?

Simplificeret hukommelsesmodel for tekst (string)

Her vises en simplificeret model af lagring af to string-variable i C#.

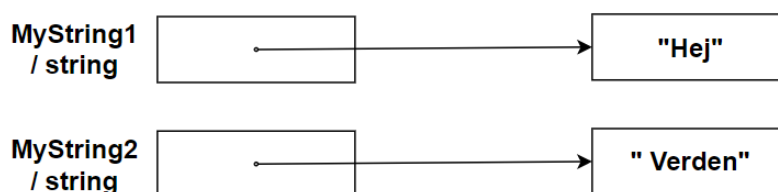
Adresserne er igen udeladt og erstattet af variabelnavne.

Variablenes indhold, adresser, er erstattet af pile, som peger på andre lager-celler hvor den egentlige tekst er placeret.

Lagring af de to strenge foregår i to trin.

Trin 1: Først erklæres de to variable `MyString1` og `MyString2`, begge af datatypen `string`. `MyString1` og `MyString2` er lagerceller der kan pege på adresser i hukommelsen.

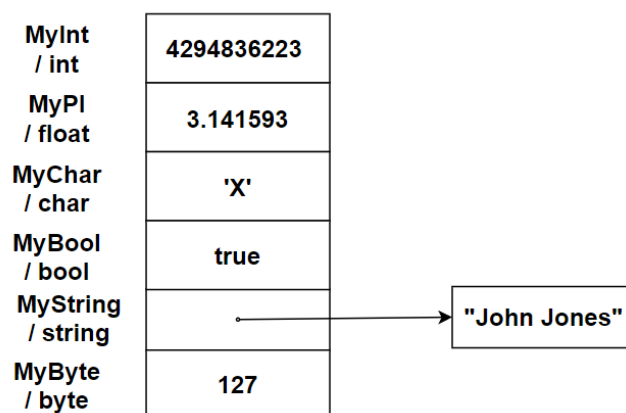
Trin 2: Dernæst, når streng-variablene tildeles en værdi foregår det ved at tegnene i strengen lagres andet sted i hukommelse hvorefter `MyString1` og `MyString2` sættes til at pege på hvor strengene faktisk er.



... fortsættes ...

Opsummering af hukommelsesmodel: Nu med værdi-datatyper og reference datatyper

Herunder vises eksempel med en blanding af datatyper.



Læg mærke til at værdi-datatyperne indeholder data selv, mens reference-datatypen, her en string, indeholder en adresse, en pil, der peger på et andet sted i lageret hvor data opbevares.

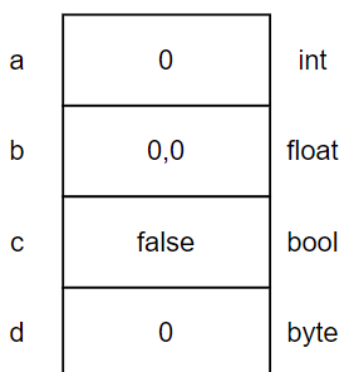
C# kode og hukommelsesmodel

Eksempel med værdi-datatyper

I C# kan alle værdi-datatyper erklæres med syntaksen "<datatype> <variabelnavn>", f.eks.:

```
int a;
float b;
bool c;
byte d;
```

Nu udarbejdes den tilhørende hukommelsesmodel. Hver erklæring af variabel, reserverer plads i hukommelsen og "sætter mærkat" også kaldet variabel-navn på hukommelsesområdet. C# sørger desuden for at gemme en "standard-værdi", også kaldet default værdi, i lagercellerne. Den resulterende (simplificerede) hukommelsesmodel for ovenstående kode er:

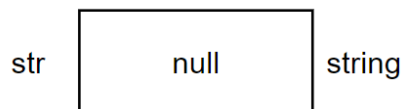


Eksempel med reference-datatype

I C# kan alle reference-datatyper erklæres med tilsvarende syntaks "<datatype> <variabelnavn>":

```
string str;
```

Men her knyttes variabelnavnet til en lagercelle som kan pege på et område i hukommelsen, men pladsen til data er ikke reserveret i hukommelsen endnu og lagercellen peger derfor på "null" - som er begrebet for ikke at pege på noget. Den tilhørende hukommelsesmodel er:

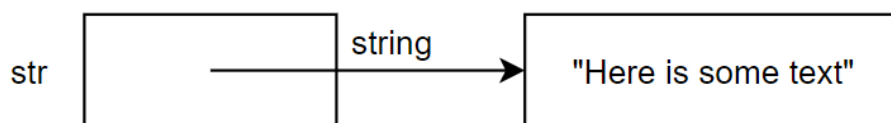


For at reservere hukommelse, til en reference-variabels data, kan man i C# altid anvende instruktionen "new <datatype>()". Her er eksempel for tekst-streng (string):

```
str = new string("Here is some text");
```

Herved sker der tre ting: (1) der reserveres hukommelse, og (2) teksten "Here is som text" kopieres ind i dette lager-område, og (3) str sættes til at pege på dette område.

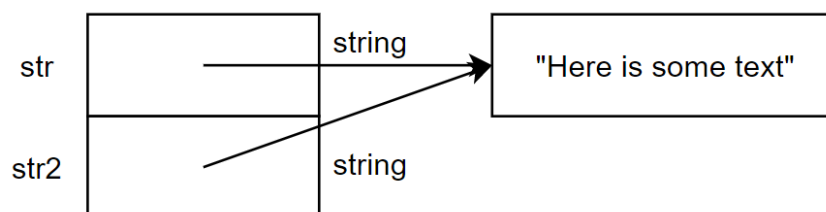
Den tilhørende hukommelsesmodel er nu.



Flere reference-variable kan pege på samme indhold. Afvikles f.eks. følgende instrukser:

```
string str2;  
str2 = str;
```

Så er den tilhørende hukommelsesmodel:



Instruksen `str2 = str;` har altså blot kopieret adressen i `str` over i `str2`. Og ikke som man kunne tro, kopieret tekst-indholdet til et nyt område i hukommelse til `str2`.

C# og = operatoren

I C# betyder "=" operatoren, læs hvad der står på højre side, og kopier det ind i variabelen på venstre side.

Som vi har set ovenfor, så virker denne fortolkning uanset om det er værdi-variable eller reference-variable, og som sådan er "=" operatoren relativ simpel.

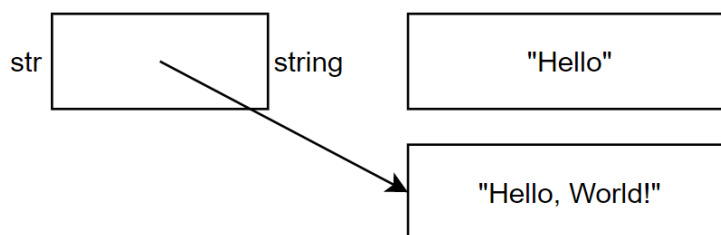
Streng (string) er immutable

I C# er string data typen, immutable. Immutable betyder "kan ikke ændres".

Når man alligevel ændrer en tekst-strengs indhold, f.eks. ved at sammensætte to tekst-streng med følgende kode:

```
str = new string("Hello");  
str = str + ", World!";
```

Da tekst-indholdet "Hello" ikke kan ændres, opretter instrukserne istedet en ny tekst og sætter str til at pege på det, som vist herunder:



Nu anvendes lageret med "Hello" i ikke længere, og hurtigt derefter finder Garbage Collectoren ud af frigive denne hukommelse til anden anvendelse.

Opsummering af C# og hukommelsesmodel

I det følgende forbindes følgende instruks-eksempler til data-modellen.

```
int tal;           // Erklæring af værdi-datatype
string str;        // Erklæring af reference-datatype
str = new string("Hello"); // Reference til data i hukommelse.
```

Pointer:

- En erklæring af en værdi-datatype variabel, f.eks. `int tal;`, gør, at
 - Der reserveres plads, dvs. lagerceller, i arbejdshukommelse med plads til netop denne datatype.
 - At variabelnavnet knyttes til denne plads i hukommelsen.
 - At der placeres en værdi i lagercellen, angives ikke en start-værdi anvendes default værdi for datatypen, typisk 0, 0.0 eller false. Angives en start-værdi anvendes den.
 - **Variablen indeholder data.**
- En erklæring af en reference-datatype variabel, f.eks. `string str;`, gør at
 - Der reserveres plads, dvs. lagerceller, i arbejdshukommelse med plads til en adresse - også kaldet reference eller pointer.
 - At variabelnavnet knyttes til denne plads i hukommelsen.
 - At der placeres NULL i lagercellen.
 - **Variablen indeholder reference - eller NULL** som betyder at der ikke peges på en plads i hukommelsen (endnu).
 - **Der er endnu ikke reserveret plads til variabelens indhold**, f.eks. "Hello".
- Anvendelse af new, f.eks. `str = new string("Hello");`, gør at
 - Der reserveres plads, dvs. lagerceller, i arbejdshukommelsen, med plads til det aktuelle indhold, her strengen med indholdet "Hello".
 - Adressen på dette indhold kopieres ind i reference variabelen, her `str`.

Bemærk: C# tilbyder ofte simple måder at reservere plads på end med `new`, men de er alle "genveje" til instruksen "new <datatype>()".

Stak, Heap og Garbage Collector

Når et computerprogram kører anvender det to områder af hukommelsen forskelligt.

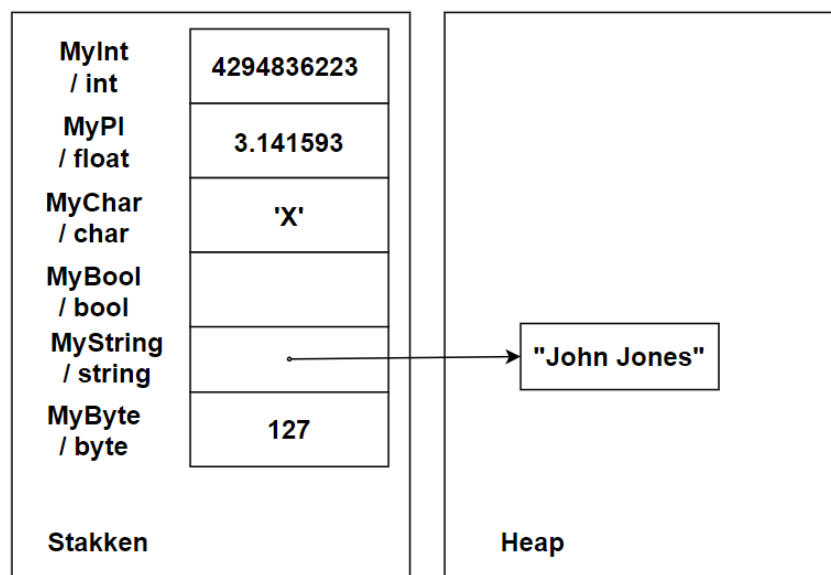
Det ene hukommelsesområde kaldes stakken (eng: Stack) og det andet kaldes "Heap".

Forskellige programmeringssprog anvender de to områder lidt forskelligt, men hovedtrækkene er ens. Her tages udgangspunkt i hvordan C# anvender områderne.

Bemærk, der er ingen fysisk forskel på de to hukommelsesområder. Den eneste forskel er den måde programmet anvender områderne på - nemlig til at løse to forskellige problemer. Det vil være for kompliceret at gå nærmere ind på problemerne i denne note, og i stedet fokuserer vi på at opstille en simpel model for stakken og for heap'en.

Groft sagt kan vi sige at *instans-variable* og *statiske variable* altid lagres i heap, mens *lokale variable* typisk lagres på stakken.

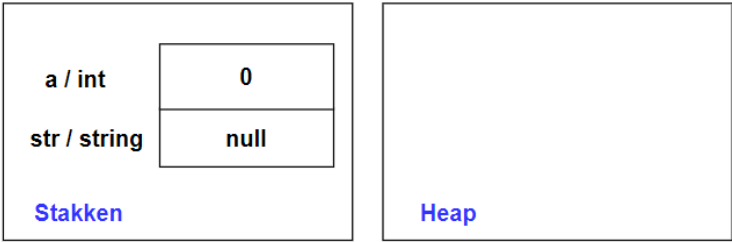
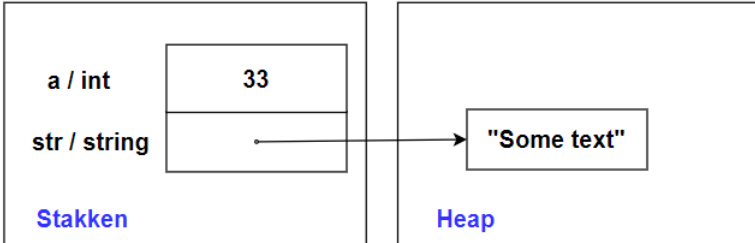
Eksemplet vist ovenfor vil derfor lagres som vist herunder:



Man kan sige at

- Heap'en indeholder et tilfældigt virvar af objekter, inklusiv deres variable og statiske variable, og at
- stakken indeholder en ordnet "stak" af variable.

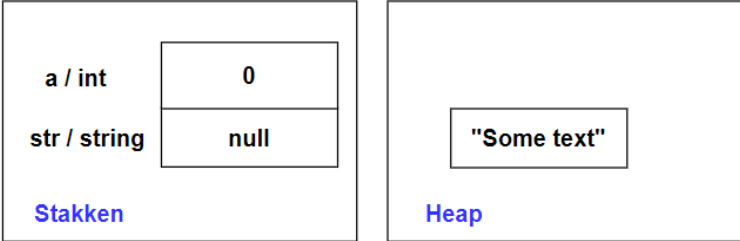
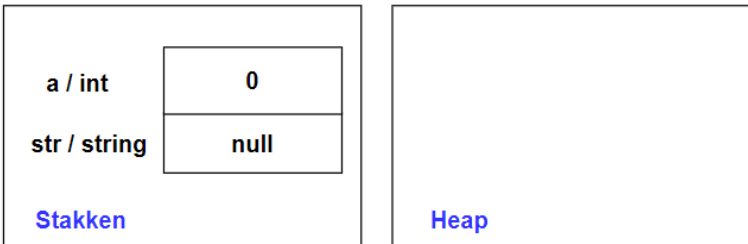
Herunder sammenholdes programkode med hvad der sker i vores hukommelsesmodel:

Antag din kode indeholder følgende: <pre>int a; string str;</pre> Så sker der følgende i hukommelsen	
Lidt senere i koden er der følgende instrukser: <pre>a = 33; str = new string("Some text");</pre>	

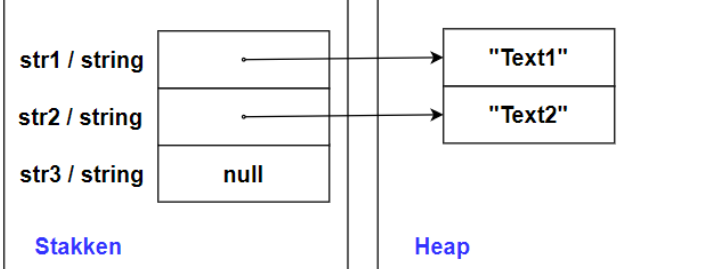
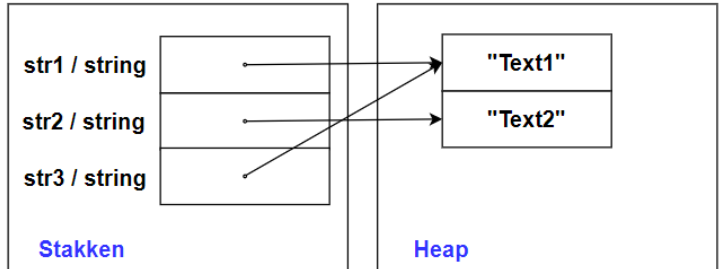
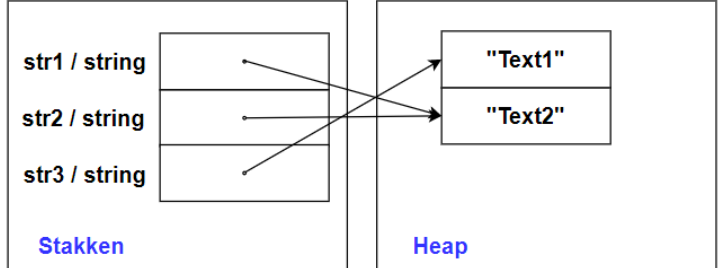
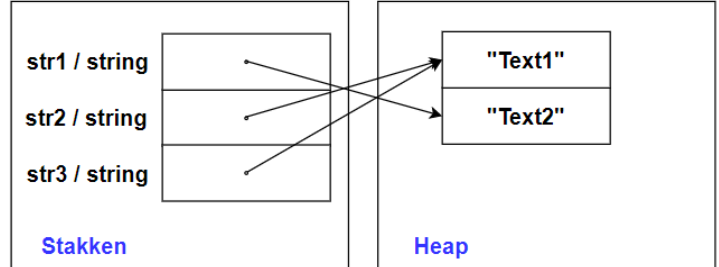
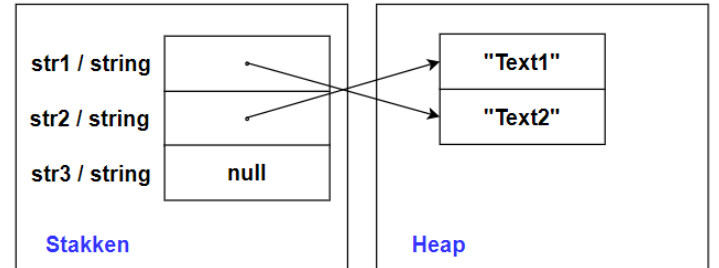
Lidt om Garbage Collectoren i .NET / C#

Garbage Collectoren, GC'en, rydder op i Heap'en.

GC'en holder øje med at der fra stakken peges på alle reserverede områder i Heap.

Hvis man på et tidspunkt mister adressen på noget i heap - f.eks. med koden: <pre>str = null;</pre> Så er der ikke længere noget der peger på "Some text" i Heap'en.	
Her går Garbage Collectoren så ind og rydder op, på den måde at lagret hvor "Some text" ligger, frigives til anden brug. Det viser vi i vores hukommelsesmodel ved at slette "Some text" i heap'en.	

Man kan ændre det en reference-variabel peger på

Antag du har følgende programkode: <pre>string str1, str2, str3; str1 = new string("Text1"); str2 = new string("Text2");</pre> Efter afviklingen ser hukommelsesmodellen ud som følger:	 Stakken Heap
Dernæst køres instruksen: <pre>str3 = str1;</pre> Hvorved adressen i str1 kopieres over i str3. Nu er hukommelsesmodellen ændret til:	 Stakken Heap
Dernæst køres instruksen: <pre>str1 = str2;</pre> Hvorved adressen i str2 kopieres over i str1. Nu er hukommelsesmodellen ændret til:	 Stakken Heap
Dernæst køres instruksen: <pre>str2 = str3;</pre> Hvorved adressen i str3 kopieres over i str2. Nu er hukommelsesmodellen ændret til:	 Stakken Heap
Nu har vi ikke behov for str3 peger på noget mere: <pre>str3 = null;</pre> Nu er hukommelsesmodellen ændret til:	 Stakken Heap

Metoder / Funktioner - parametre og argumenter

Vi skal først se lidt på hvad en metode er. En metode har følgende egenskaber:

- De har et navn som anvendes til at kalde dem,
- Når man "kalder" en metoden, betyder det at man kører instrukserne i metoden.
- Metoder kan modtage variable, som lokalt betegnes for metodens parametre.
- Metoder kan desuden have lokale variable, som kan hjælpe metoden til at gøre det den skal.
- Metoder returnerer en variabel (i C familien af programmeringssprog klares det med "return" instruksenen).

Bemærk: Forskellige programmeringssprog anvender forskellige navne for ovenstående. Udover navnet metode, støder du ofte også på navnene funktion eller procedure.

Stakken og metoder

Stakken (Stack) er det hukommelsesområde som blandt andet løser problemet med at metoder skal kunne kalde sig selv, også kaldet rekursion. Man kan derved ikke afsætte et fast område i hukommelse til anvendelse ved kørsel af en bestemt metode. I stedet gør man følgende.

Når en metode kaldes, så oprettes de variable den arbejder på, på stakken. Man siger at variable push'es til stakken. Her kan metoden så anvende variablene til at udføre dens opgave.

Når metoden er færdig og returnerer, fjernes dens variablene fra stakken. Man siger at variable pop'es af stakken.

Allokering af variablene på stakken sker automatisk; du skal som programmør ikke sørge for dette; det klarer compileren for dig.

Lidt teknisk kan man sige at stakken groft set anvendes som en LIFO (last-in-first-out) data struktur. Den sidst kaldte funktion skal afslutte før metoden før den kan afslutte og så videre.

... fortsættes ...

Lad os prøve at bytte om på to tal og udskrive dem ved hjælp af en metode. Samtidig opbygger vi en model af hvordan stakken anvendes.

Først skrives en simpel Main metode:

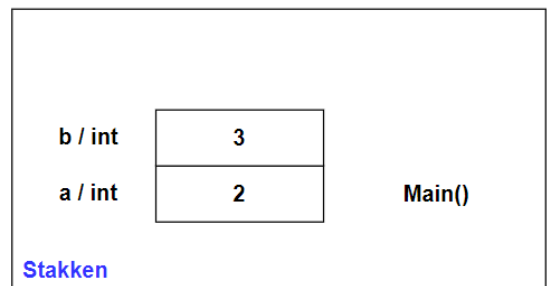
```
static void Main()
{
    int a, b;
    a = 2; b = 3;

}
```

Variablene a og b er lokale (for Main) og placeres dermed på stakken.

Stak-modellen er vist til højre:

Stakken oprettes nede fra og op. I dette tilfælde er a erklæret først, så den er placeret på stakken først. Dernæst er b erklæret, så den placeres ovenpå a.



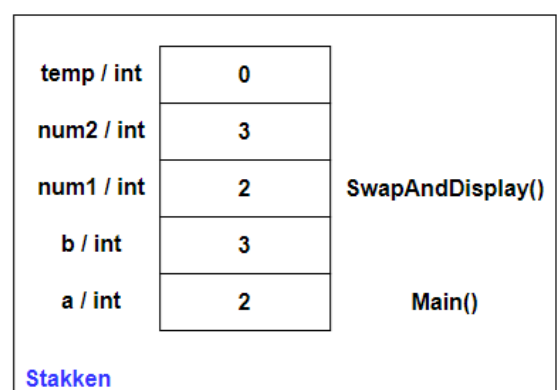
Nu tilføjes en metode SwapAndDisplay()

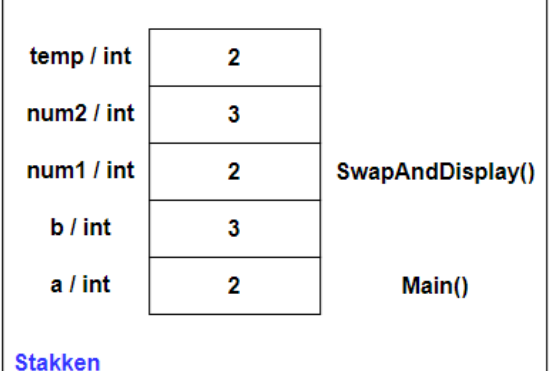
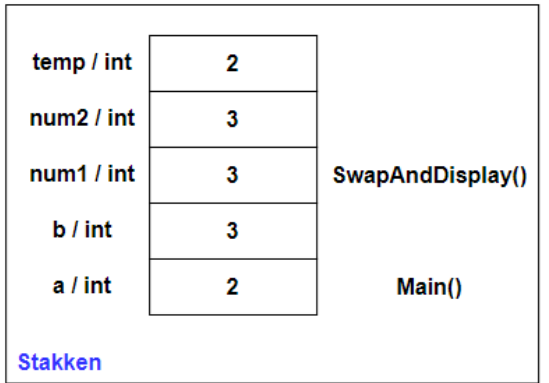
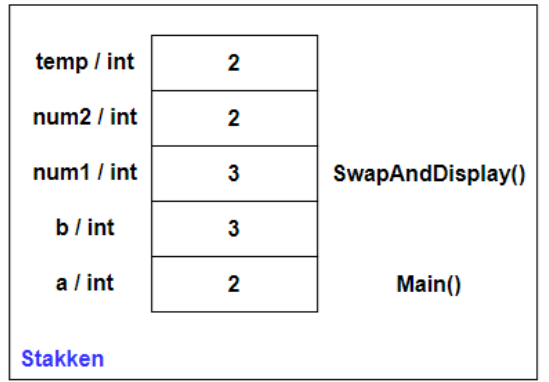
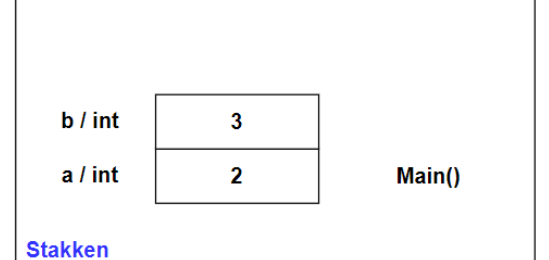
```
static void SwapAndDisplay(int num1, int num2)
{
    int temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
    Console.WriteLine("First number is {0}", num1);
    Console.WriteLine(", and the second is {0}", num2);
    return;
}
```

I Main metoden kalder vi ovenstående SwapAndDisplay ved:

```
static void Main()
{
    int a, b;
    a = 2; b = 3;
    SwapAndDisplay(a, b);
}
```

Og inden første instruks i SwapAndDisplay er kørt, er vores stak-model følgende:



Når SwapAndDisplay() har afviklet instruksen: <pre>temp = num1;</pre> Så har den lokale variabel temp fået værdien 2.	
Når SwapAndDisplay() har afviklet instruksen: <pre>num1 = num2;</pre> Så har den lokale parameter num1 fået værdien 3.	
Når SwapAndDisplay() har afviklet instruksen: <pre>num2 = temp;</pre> Så har den lokale parameter num2 fået værdien 2. Og dermed er værdierne byttet rundt i num1 og num2 og vi kan skrive dem ud: <pre>First number is 3, and the second is 2</pre>	
Nu er metoden SwapAndDisplay() færdig og returnerer med instruksen: <pre>return;</pre> Derved nedlægges den del af stakken som blev brugt af SwapAndDisplay().	

Læg mærke til, at selvom værdierne blev byttet inde i metoden SwapAndDisplay, så er argumenterne a og b som blev kopieret til metoden urørte.

Ønsker du at bytte værdierne i a og b kan det gøres ved at overføre adressen på a og b i stedet for.

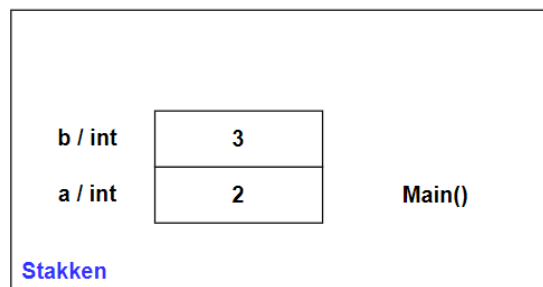
Lad os skrive en metode som kan bytte om på to tal

Det ændrede program er vist herunder:

```
static void Main()
{
    int a, b;
    a = 2; b = 3;
    Swap(ref a, ref b);
}

static void Swap(ref int num1, ref int num2)
{
    int temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
    return;
}
```

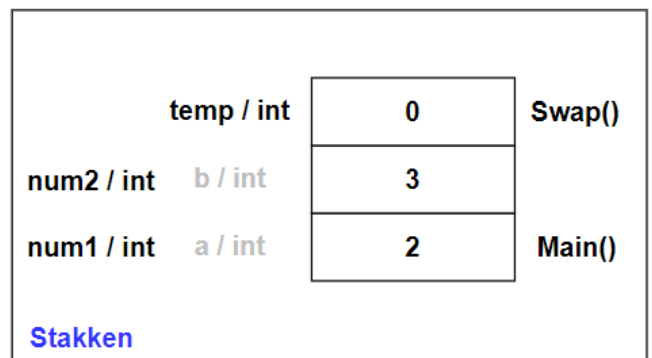
Inden kørsel af Swap, ser stakken ud som følger:



Efter kaldet af Swap, men lige inden første instruks er kørt (i Swap) ser vores stak-model ud som følger:

Læg mærke til at num1 og num2 ikke er nye variable, men blot referencer til samme sted i hukommelsen som a og b.

Den eneste nye variable på stakken er temp.



Efter kørsel af <pre>temp = num1;</pre> Ser stakken ud som følger	temp / int 2 Swap() num2 / int b / int 3 num1 / int a / int 2 Main() Stakken
Efter kørsel af <pre>num1 = num2;</pre> Ser stakken ud som følger:	temp / int 2 Swap() num2 / int b / int 3 num1 / int a / int 3 Main() Stakken
Efter kørsel af <pre>num2 = temp;</pre> Ser stakken ud som følger:	temp / int 2 Swap() num2 / int b / int 2 num1 / int a / int 3 Main() Stakken
Efter metoden returnerer med: <pre>return;</pre> Ser stakken ud som følger: Læg mærke til at variablene a og b fra den kaldende metode Main, nu er byttet om.	b / int 2 a / int 3 Main() Stakken

Appendix 1: ASCII tabeller

Regular ASCII Chart (character codes 0 – 127)											
000	(nul)	016	► (dle)	032	sp	048	0	064	@	080	P
001	☺ (soh)	017	◄ (dc1)	033	!	049	1	065	A	081	Q
002	☹ (stx)	018	↑ (dc2)	034	"	050	2	066	B	082	R
003	♥ (etx)	019	!! (dc3)	035	#	051	3	067	C	083	S
004	♦ (eot)	020	¶ (dc4)	036	\$	052	4	068	D	084	T
005	‡ (enq)	021	§ (nak)	037	%	053	5	069	E	085	U
006	♠ (ack)	022	— (syn)	038	&	054	6	070	F	086	V
007	• (bel)	023	↑ (etb)	039	'	055	7	071	G	087	W
008	■ (bs)	024	↑ (can)	040	(	056	8	072	H	088	X
009	(tab)	025	↓ (em)	041	)	057	9	073	I	089	Y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z
011	♂ (vt)	027	+ (esc)	043	+	059	:	075	K	091	[
012	♀ (np)	028	- (fs)	044	,	060	<	076	L	092	\
013	(cr)	029	* (gs)	045	-	061	=	077	M	093	]
014	♂ (so)	030	▲ (rs)	046	.	062	>	078	N	094	^
015	※ (si)	031	▼ (us)	047	/	063	?	079	O	095	_

Extended ASCII Chart (character codes 128 – 255)											
128	Ç	143	À	158	h	172	¼	186	¶	200	ℓ
129	ü	144	Á	159	i	173	½	187	§	201	ℓ
130	é	145	Â	160	j	174	¾	188	¶	202	ℓ
131	â	146	Ã	161	k	175	»	189	¶	203	ℓ
132	ä	147	Ä	162	l	176	¼	190	¶	204	ℓ
133	å	148	Å	163	m	177	½	191	¶	205	ℓ
134	ä	149	Ö	164	n	178	¾	192	¶	206	ℓ
135	ç	150	Ø	165	ñ	179	¼	193	¶	207	ℓ
136	è	151	Ù	166	o	180	½	194	¶	208	ℓ
137	ë	152	Ú	167	p	181	¾	195	¶	209	ℓ
138	ì	153	Û	168	q	182	¼	196	¶	210	ℓ
139	í	154	Ü	169	r	183	½	197	¶	211	ℓ
140	î	155	Ý	170	s	184	¾	198	¶	212	ℓ
141	ï	156	Þ	171	t	185	¼	199	¶	213	ℓ
142	ä	157	ÿ								