

Përmbajtje: Gjuha c Code format: language cpp dhe xcode theme

Sintaksa:	5
Sekuenca e largimit	5
Variablat janë kontejnerë për ruajtjen e vlerave të të dhënave.	5
Deklaroni variabla të shumta x = y = z = 50;	6
Variablat Boolean	6
Emrat e variablave C - Identifikuesit	7
Specifikuesit e formatit – vendmbajtës për printim variabla, ne funksionin printf sipas tipit.	7
Llojet bazë të të dhënave	7
Llojet e të dhënave të karaktereve C	8
ascii	8
C Llojet e të dhënave numerike	9
Numrat shkencorë	9
Vendos saktësinë decimale	10
Merrni madhësinë e memories	10
Kujtesa sizeof(myChar)	10
Lloje te zgjeruara	11
Konvertimet (int to float)	11
Konstantet C - variabla që nuk duhet të ndryshojnë -read-only.	12
C Operatorët - përdoren për të kryer operacione në variabla dhe vlera.	12
Rendi i Operacioneve	13
Operatorët bitwise – Detaje	13
>>= dhe <<= n pjesëtim ose shumëzim me 2n	14
◆ ^= (XOR assignment) fik e ndez	15
= bitwise OR () për të shtuar opsione pa i fikur ato ekzistuese	15
&= bitwise AND (&) për të filtruar disa bite specifike	17
C Nëse ... Ndryshe	18
if - Kushtet dhe deklaratat nëse	18
Else Deklarata tjetër	18
Përdorimi i një variabli Boolean	18
else if - Deklarata tjetër nëse – shtohet kusht	19
Ternary Operator (trash) Shkurtore - If...Else	19
Nested if – if i ndërfutur	19
Switch – ne vend te if .. else Deklarata e ndërrimit	20
break - Fjala kyçe e pushimit	21
Ciklet - loops	22
While Loop / cikli ndërsa	22
Cikli Do/While - është variant i ciklit while	22
Shembull – nr invers	23
Cikli For	23
Break / continue ndal dhe Vazhdo	24
 continue	24

Vektorët - grupet(numra)groups ose Vargjet(tekst)/strings / Vlera te shumëfishta te njëjtit tip ne te njëjtin variabël	25
Vendosni madhësinë e grupit	25
Merrni numrin e elementeve	25
Cikli përmes një grupi (elementet numra) / vargu (elementet tekst)	26
Bërja e cikleve më të mira	26
Shembull i jetës reale	27
Vektorët - Grupe shumëdimensionale	27
Vektorët - grupe dydimensionale	27
Hyni në elementët e një grupi 2D	28
Ndryshoni elementet në një grup 2D	28
Ciklo nëpër një grup 2D	28
Grupet tredimensionale	29
Kur të përdorni grupe shumëdimensionale	29
Detyra – vektor numrash = grup:	29
Vektorët strings = Vargjet C	30
Vargjet e aksesit	30
Modifikoni vargjet	31
Cikël në varg - <code>int length = sizeof(carName) / sizeof(carName[0]);</code>	31
Një mënyrë tjetër për të krijuar vargje	32
Dallimet	32
Vargjet - Karakteret speciale - sekuenca largimit – vazhdim	32
Funksionet e vargut C – string functions	33
1. Gjatësia e vargut	33
2. Vargjet e lidhura	33
3. Kopjo vargjet	34
4. Krahasoni vargjet	34
scanf - Hyrja e përdoruesit	35
Hyrje të shumta	36
Adresa e kujtesës - operatori i referencës &	38
Pointers – krijim variabli pointer me operatorin *	38
De=*-referencë=& – operatori i de-referencës *	39
pointers dhe grupet = vektorët numerikë	40
shënjeshtër pointers dhe vargjet	40
Si lidhen pointers /shënjeshtër me vargjet	41
Aritmetika e pointerave /shënjeshtërve	44
Rritja dhe zvogëlimi	44
Zbritja e pointer (distanca)	44
Aritmetika e treguesit varet nga tipi ose lloji	45
Cikle me pointera	46
Gabimet e zakonshme që duhen shmangur	46
C Pointer to Pointer	46
Funksionet C	48

Funksionet e paracaktuara	48
Krijoni një funksion	48
Thirrni një funksion	49
Llogaritni shumën e numrave	50
Parametrat e funksionit C	50
Parametra të shumtë	51
Kaloni vargjet si parametra funksioni	52
Vlerat e kthimit	53
Fushëveprimi.....	55
Shtrirja lokale	55
Shtrirja globale	56
Emërtimi i variablave	57
Përfundim	58
Deklarata dhe përkufizimi i funksionit C	58
Po në lidhje me parametrat	59
Funksionet që thërrasin funksione të tjera	60
Funksioni inline	60
Shembull i funksionit në linjë	61
Kur të mos përdorni Inline:	61
Funksioni i brendshëm vs i rregullt	62
Dallimet kryesore:	62
Rekursioni.....	62
Llogaritni faktorialin me rekursion	64
Pointeri /shënjeshtër i funksionit.....	64
Deklarimi i një pointer funksioni	65
Caktimi i një funksioni një treguesi	65
Thirrja e një funksioni përmes një pointer	65
Shembull i pointer të funksionit	65
Kalimi i një funksioni si argument	66
Grupi /vargu i treguesit pointer të funksionit	67
Treguesi i funksionit vs funksioni normal	68
Përmbledhje.....	68
Funksioni i thirrjes mbrapsht	68
Shembull i thjeshtë i thirrjes mbrapsht	69
Callback me Parametra	69
Thirrje të shumta për rikthim	70
Shembull nga Jeta Reale: Përdorimi i Callback-eve në qsort()	70
Përmbledhje.....	71
C Files - Menaxhimi i Skedarëve dhe Si Të Krijoni Skedarë.....	71
Trajtimi i Skedarëve	71
Krijo një Skedar	72
Shkruaj në një skedar	73

Shtoni përmbajtje në një skedar	75
Lexo një Skedar	75
Praktika e Mirë	77
Strukturat	79
Krijo një Strukturë	79
Anëtarët e Strukturës së Aksesit	79
Po për vargjet ose fijet në struktura?	80
Sintaksa më e thjeshtë	81
Strukturat e Kopjimit	82
Modifiko Vlerat	82
Shembull nga jeta reale	83
Sfida: Ruaj moshën e një personi	84
Struktura të Ndërlidhura	85
Strukturat dhe treguesit	85
Kalimi i treguesve të strukturës për funksionet	86
Pse të përdorni Struct Pointers?	86
C Unions	87
Deklaro një union / Bashkim	87
Anëtarët e Access Union	88
Madhësia e një unioni	88
Sqarim me shembull:	89
struct vs union	89
Kur përdoren unionet	89
C typedef	90
Pse të përdorni typedef?	90
Typedef me strukturë	90
Mbushja e strukturës	92
Pse?	93
Si të Reduktohet Mbushja	94
Pse Ka Rëndësi	94
Strukturat kundrejt unioneve	95
Përmbledhje	96
C Enums – Lista specifike	96
Ndryshimi i Vlerave	97
Enum në një deklaratë çelës / switch	98
Typedef me Enum	98
Sfida: Ruani një ditë jave duke përdorur një Enum	99
Menaxhimi i Memories C	100
Pse është e rëndësishme të dihet?	101
Memoria Statike	101
Memoria Dinamike	102
Shënime	103

Memoria e grumbullit	103
Makro janë me te shpejte se funksionet.....	104

Gjuha c ka formatim kodimi ne laboratorët, me përjashtim lab **8, 10**

Sintaksa:

Shënim: Çdo deklaratë C përfundon me një pikëpresje ; udhëzimet janë deklarata.

Rreshta te përsëritur:

```
#include <stdio.h>    //skedar koke-na lejon të përdorim funksione hyrëse/dalëse
int main() {          //Programi juaj fillon të funksionojë këtu

...
    char name[] = "John";
    printf("Hello %s", name);

...
    return 0;          /*përfundon programin/funksionin main dhe dërgon një vlerë përsëri në sistemin
operativ. Kthimi zakonisht do të thotë "gjithçka funksionoi".*/
}
```

Sekuenca e largimit

Sekuencat e arratisjes fillojnë me një prerje të kundërt dhe përfaqësojnë karaktere speciale që nuk mund të shtypen drejtpërdrejt.

\\ Fut një karakter të kundërt (\)

\ Fut një karakter me thonjëza të dyfishta

Komentet me shumë rreshta fillojnë dhe përfundojnë me ./**/

Variablat janë **kontejnerë** për ruajtjen e vlerave të të dhënave.

Mendoni për një variabël **si një kuti me emër ku mbani një vlerë** që mund të përdoret më vonë.

Në C, variablat duhet të kenë një **lloj** specifik, i cili i tregon programit se çfarë lloj të dhënash mund të ruajë variabli.

Sintaksa: *type variableName = value;*

Shenja e barabartë përdoret për t'i caktuar një vlerë ndryshore.=

Këshillë: Emrat e variablave duhet të jenë kuptimplotë, kështu që kodi juaj është më i lehtë për t'u kuptuar.

1. int - ruan numra të plotë (numra të plotë), të tilla si ose 123-123
2. float - ruan numrat me dhjetore, të tilla si ose 19.99-19.99
3. char - ruan një karakter të vetëm, si p.sh. Personazhet janë të rrethuar nga **thonjëza të vetme**'a' 'B'

Variablat kanë për qëllim të ndryshojnë gjatë ekzekutimit të një programi.

Ju madje mund të kopjoni vlerat në variabla që janë deklaruar më herët pa një vlerë:

Mund të përdorni operatorë matematikorë për të punuar me variabla.

Ju gjithashtu mund të përditësoni një variabël duke përdorur vlerën e saj aktuale: Vlera e mëparshme mbishkruhet

Deklaroni variabla të shumta $x = y = z = 50$;

Për të deklaruar më shumë se një variabël të të njëjtit lloj, përdorni një listë të **ndarë me presje**:

Shënim: Të gjitha variablat duhet të jenë të të njëjtit lloj. Ju nuk mund të përzieni llojet në të njëjtën deklaratë.

```
int x = 5, y = 6, z = 50;
printf("%d", x + y + z);
```

Ju gjithashtu mund t'i caktoni të **njëjtën vlerë** shumë variablave në të njëjtën kohë:

```
int x, y, z;
x = y = z = 50;
printf("%d", x + y + z);
```

Variablat Boolean

Mund të ketë vetëm një nga dy vlerat, si:

- PO / JO
- NDEZUR / FIKUR
- E VËRTETË / E RREME

Në C, lloji nuk është një lloj i integruar i të dhënave, si ose .bool int char. Është prezantuar në C99 dhe duhet të **përfshini** skedarin e mëposhtëm të kokës për ta përdorur atë:

Mos harroni të përfshini skedarin e kokës kur punoni me variabla. `<stdbool.h>bool`

- **#include <stdbool.h>**

Një variabël boolean deklarohet me fjalën kyçe dhe mund të marrë vlerat ose :

```
bool true false
```

```
bool isProgrammingFun = true;
bool isFishTasty = false;
```

Para se të përpiqeni të printoni variablat booleanë, duhet të dini se vlerat booleane kthehen si **numra të plotë**:

- **1** (ose çdo numër tjetër që nuk është 0) përfaqëson **true**
- **0** përfaqëson **false**

Prandaj, mund të përdorni specifikuesin e formatit për të printuar një vlerë booleane: **%d**

Booleanët janë baza për të gjitha krahasimet dhe kushtet.

Do të mësoni më shumë rreth [kushteve \(nëse... tjetër\)](#) në kapitullin tjetër.

Shembull

Dalja "Mjaftueshëm i vjetër për të votuar!" nëse është **më i madh ose i barabartë me** . Përndryshe nxjerr "Nuk është mjaftueshëm i vjetër për të votuar.":**myAge18**

```
int myAge = 25;
int votingAge = 18;

if (myAge >= votingAge) {
    printf("Old enough to vote!");
} else {
    printf("Not old enough to vote.");
}
```

Shembull

```
// Create boolean variables
bool isProgrammingFun = true;
bool isFishTasty = false;

// Print boolean values
printf("%d", isProgrammingFun); // Prints 1 (true)
printf("%d", isFishTasty);      // Prints 0 (false)
```

Emrat e variablave C - Identifikuesit

Të gjitha **variablat** C duhet të **identifikohen** me **emra unikë**.

Këta emra unikë quhen **identifikues**.

Identifikuesit mund të jenë emra të shkurtër (si x dhe y) ose emra më përshkrues (mosha, shuma, totalVolume).

- **E vlefshme:** , , age_counttotalSum
- `// Student data`
`int studentID = 15;`
`int studentAge = 23;;`

Për të printuar variabla në C, duhet të përdorni diçka të quajtur [specifikues formati](#).

Specifikuesit e formatit – vendmbajtës për printim variabla, ne funksionin printf sipas tipit.

1. Ju mund të mendoni për një specifikues formati si një **vendmbajtës** që i tregon C se çfarë lloj vlere do të printohet.
2. Specifikuesit e formatit përdoren së **bashku me funksionin për të printuar variabla.printf()**
3. Një specifikues formati fillon gjithmonë me një shenjë përqindjeje, e ndjekur nga një shkronjë.%

Llojet bazë të të dhënave

Data Type	Size	Description	Example
int <code>%d</code> or <code>%i</code>	2 or 4 bytes	Stores whole numbers, without decimals	1
float <code>%f</code> or <code>%F</code>	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits	1.99
double <code>%lf</code>	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits	1.99
char <code>%c</code> <code>%s</code>	1 byte	Stores a single character/letter/number, or ASCII values	'A'

Llojet e të dhënave të karaktereve C

nëse jeni të njohur me ASCII, mund të përdorni vlerat ASCII për të shfaqur disa karaktere. Vini re se këto vlera nuk janë të rrethuara me citate (), Siç janë numrat:"

```
char a = 65, b = 66, c = 67;
```

```
printf("%c", a);
```

```
printf("%c", b);
```

```
printf("%c", c);
```

Nëse përpigjeni të ruani më shumë se një karakter të vetëm, ai do të printojë vetëm të fundit Karakteri:
Shembull

```
char myText = 'Hello';
```

```
printf("%c", myText);
```

jep vetëm **o** ne ekzekutim

Për të ruajtur karaktere të shumta (ose fjalë të tëra), përdorni [vargje](#) (për të cilat do të mësoni më shumë në një kapitull të mëvonshëm):

```
char myText[] = "Hello";
```

```
printf("%s", myText);
```

ascii

[HTML ASCII Reference](#) 0 - 31 Control characters (see below) 65-90 germa te mëdha 97-122 lower

case

~ 126 tilde, @at, & 38 ampersand, (40 left parenthesis

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

C Llojet e të dhënave numerike

float vs. double

Saktësia e një vlere të pikës lundruese tregon se sa shifra mund të ketë vlera Pas pikës decimale. Saktësia e është gjashtë ose shtatë Shifra decimale, ndërsa variablat kanë një saktësi prej rreth 15 shifrash. Prandaj, shpesh është më e sigurt për t'u përdorur për shumicën e llogaritjeve - por vini re se ajo merr dy herë më shumë [kujtesë](#) se (8 bytes vs. 4 bytes). **float double doublefloat**

Numrat shkencorë

Një numër pikash lundruese mund të jetë edhe një numër shkencor me një "e" për të treguar fuqinë e 10:

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

```
int main() {
```

```
float f1 = 35e3;    // 35 * 10^3 = 35000
double d1 = 12E4;   // 12 * 10^4 = 120000
```

```
printf("%f\n", f1);
printf("%lf", d1);
```

```
return 0;
```

```
}
```

```
35000.000000
```

```
120000.000000
```

Ka specifikues të formatit të ndryshëm për çdo lloj të dhënash. Ja disa prej ata:

Format Specifier Data Type

%d or %i **int**

%f or %F	float
%lf	double
%c	char
%s	Used for strings (text) , which you will learn more about in a later chapter

Vendos saktësinë decimale

[Llojet e të dhënave C Saktësia dhjetore](#)

Nëse dëshironi të hiqni zerot shtesë (përcaktoni saktësinë decimale), mund të përdorni një pikë (.) pasuar nga një numër që specifikon se sa shifra duhet të tregohen Pas pikës decimale:.

```
float myFloatNum = 3.5;
```

```
printf("%f\n", myFloatNum); // Default will show 6 digits after the decimal
point
printf("%.1f\n", myFloatNum); // Only show 1 digit
printf("%.2f\n", myFloatNum); // Only show 2 digits
printf("%.4f", myFloatNum); // Only show 4 digits
```

19.99 me **%.01** do japë 20 pra rrumbullakos nr tek te qindëshe.

Merrni madhësinë e memories

Kujtesa sizeof(myChar)

[C Më shumë lloje të dhënash \(lloje të zgjeruara\)](#)

[Madhësia C e operatorit](#)

Përdorimi i llojit të duhur të të dhënave për qëllimin e duhur do të **kursejë kujtesën** dhe do të **përmirësojë performancën** e programit tuaj.

It is because the compiler expects the sizeof operator to return a %lu.

Eshtë më e sigurt për t'u përdorur long unsigned int (%lu)

Lu=long unsign

Me cilin specifikues formati përdoret shpesh për të shfaqur **rezultatit e bajtave**?printfsizeof

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

```
int main() {
```

```
int myInt;
```

```
float myFloat;
```

```
double myDouble;
```

```
char myChar;
```

```
printf("%zu\n", sizeof(myInt));
```

```
printf("%zu\n", sizeof(myFloat));
```

```
printf("%zu\n", sizeof(myDouble));
printf("%zu\n", sizeof(myChar));
```

```
return 0;
}
```

```
4
4
8
1
```

Nëse keni një sërë vlerash 500, sa bajta memorie do të zërë? char

☒ 500 bytes

☐ 1000 bytes

☐ 250 bytes

☐ 4 bytes

[Dërgo përgjigje »](#)

Përdorimi i llojit të duhur të të dhënave për qëllimin e duhur do të **kursejë memorie** dhe do të **përmirësojë performancën** e programit tuaj.

Lloje të zgjeruara

[C Më shumë lloje të dhënash \(lloje të zgjeruara\)](#)

Shënim: Këto lloje të zgjeruara përdoren kryesisht kur keni nevojë për kontroll shumë specifik mbi përdorimin e memories ose diapazonin e numrave.

Shënim: Për programimin e përditshëm, , , dhe zakonisht janë të mjaftueshme. **int float double char**

[C Më shumë lloje të dhënash \(lloje të zgjeruara\)](#)

Konvertimet (int to float)

[Madhësia C e operatorit](#)

Ekzistojnë dy lloje të konvertimit në C:

- **Konvertimi implicit** / i nënkuptuar (automatikisht)
- **Konvertimi eksplicite** / i qarte (manualisht)

// Automatic conversion: float to int

```
int myInt = 9.99;
printf("%d", myInt);           // 9   gabim
```

```
float sum = 5 / 2;
printf("%f", sum);             // 2.000000   gabim
```

5 dhe 2 janë **ende integers** në divizionin. Në këtë rast, ju duhet të konvertoni manualisht vlerat integer në Vlerat e pikës lundruese.

// Manual conversion: int to float

```
int num1 = 5;
int num2 = 2;
float sum = (float) num1 / num2;
```

```
printf("%f", sum); // 2.500000
printf("%.1f", sum); // 2.5
```

Konstantet C - variabla që nuk duhet të ndryshojnë-read-only.

Kjo mund të bëhet me fjalën kyçe, e cila e bën një variabël të **pandryshueshëm** dhe **vetëm për lexim**:const

```
const int myNum = 15; // myNum will always be 15
myNum = 10; // error: assignment of read-only variable 'myNum'
```

Redaktori i W3Schools Tryit:

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

```
int main() {
    const int myNum = 15;           //fjala kyçe const e bën read-only variablin myNum
    myNum = 10;

    printf("%d", myNum);
    return 0;
}
```

Afishimi në ekzekutim:

```
prog.c: Në funksionin 'main':
prog.c:5:18: gabim: caktimi i variablit vetëm për lexim 'myNum'
5 | myNum = 10;
  | ^
```

c Operatorët - përdoren për të kryer operacione në variabla dhe vlera.

C i ndan operatorët në grupet e mëposhtme:

1. Operatorët aritmetikë: 7: + - / * ++ -- %
2. Operatorët e caktimit: 11: =, +=, -=, /=, *=
3. Operatorët e krahasimit: >, <, >=, <=,
4. Operatorët logjikë: 3: && || ! [C Operatorët logjikë](#) shembull login
5. Operatorët bitwise 5

	Operator	Shembull	Ekuivalent me	Shpjegim
1	=	x = 5	x = 5	Cakton vlerën 5 te variabli x.
2	+=	x += 3	x = x + 3	Shton 3 te vlera ekzistuese e x.
3	-=	x -= 3	x = x - 3	Zbrit 3 nga vlera ekzistuese e x.
4	*=	x *= 3	x = x * 3	Shumëzon x me 3.
5	/=	x /= 3	x = x / 3	Pjesëton x me 3.

Operator	Shembull	Ekuivalent me	Shpjegim
6 %	<code>x %= 3</code>	<code>x = x % 3</code>	Merr mbetjen e pjesëtimit të x me 3.
7 &	<code>x &= 3</code>	<code>x = x & 3</code>	Kryen AND bitwise midis x dhe 3.
8 `	<code>= `</code>	<code>`x</code>	<code>= 3`</code>
9 ^	<code>x ^= 3</code>	<code>x = x ^ 3</code>	Kryen XOR bitwise midis x dhe 3.
10 >>=	<code>x >>= 3</code>	<code>x = x >> 3</code>	Zhvendos bitët e x djathtas me 3 pozicione.
Zhvendosja djathtas është e barabartë me pjesëtimin e vlerës me 2 ⁿ .			zhvendos bitet e variablit x djathtas për n pozicione dhe rezultatin e ruan përsëri te x
11 <<=	<code>x <<= 3</code>	<code>x = x << 3</code>	Zhvendos bitët e x majtas me 3 pozicione. 2 ⁿ n=3

Një përdorim tjetër i zakonshëm është kontrollimi nëse një fjalëkalim është mjaft i gjatë:

```
int passwordLength = 5;

printf("%d\n", passwordLength >= 8); // 0 (false), too short
printf("%d\n", passwordLength < 8); // 1 (true), needs more characters
```

Shembull i jetës reale: Ndjekja e kursimeve

Operatorët e caktimit mund të përdoren gjithashtu në skenarë të jetës reale. Për shembull, mund të përdorni operatorin për të mbajtur gjurmët e kursimeve kur shtoni para në një llogari:+=

Rendi i Operacioneve

Këtu janë disa operatorë të zakonshëm në C, nga prioriteti më i lartë në atë më të ulët:

- **()** - Kllapa
- *****, **,** - Shumëzimi, Pjesëtimi, Moduli **/%**
- **+**, **-** - Mbledhja, Zbritja
- **>**, **<**, **<=**, **>=** - Krahasimi
- **==**, **!=** - Barazia
- **&&** - Logjike DHE
- **||** - OR logjike
- **=** - Detyra

Operatorët bitwise – Detaje

Operator	Veprimi kryesor	Përdorim tipik
&=	Heq/filtron bite	Heq një flag
=	Shton bite	Aktivizon një flag
^=	Ndërron bite	Toggle një flag

👉 Këta operatorë janë shumë të përdorur në programim të sistemeve, kur menaxhohen **opsione, leje, konfigurime** ose kur punohet **direkt me hardware**.

>>= dhe <<= n pjesëtim ose shumëzim me 2^n

```
int x = 40; // në binar: 101000
x >>= 3;    // zhvendos 3 pozicione djathtas: 000101
// Rezultati: 5 në decimal
```

Pra, $40 \gg= 3$ jep 5, sepse 40 pjesëtohet me $2^3 = 8$.

👉 Ky operator përdoret shpesh për operacione të shpejta matematikore (pjesëtim me fuqi të 2) ose për manipulim të biteve në programim të ulët (low-level programming).

Operatori i kundërt <<= (zhvendosje majtas), që është si shumëzim me fuqi të 2?

Shembull të plotë ku përdoren të tre operatorët (&=, |=, ^=) për menaxhimin e **lejeve të një përdoruesi**:

```
#include <stdio.h>

#define FLAG_READ  0x01 // 0001
#define FLAG_WRITE 0x02 // 0010
#define FLAG_EXEC  0x04 // 0100

int main() {
    int permissions = 0; // fillimisht pa leje

    // Aktivizojmë READ dhe WRITE me |=
    permissions |= FLAG_READ; // 0001
    permissions |= FLAG_WRITE; // 0011 (READ + WRITE)

    printf("Lejet fillestare: %d\n", permissions);

    // Heqim WRITE me &=
    permissions &= ~FLAG_WRITE; // 0001 (vetëm READ)
    printf("Pas heqjes së WRITE: %d\n", permissions);

    // Ndërrojmë EXEC me ^=
    permissions ^= FLAG_EXEC; // 0101 (READ + EXEC)
    printf("Pas toggle të EXEC: %d\n", permissions);

    // Ndërrojmë përsëri EXEC me ^=
    permissions ^= FLAG_EXEC; // 0001 (vetëm READ)
    printf("Pas toggle të dytë të EXEC: %d\n", permissions);

    return 0;
}
```

Çfarë ndodh:

- |= aktivizon flamujt (shton leje).
- &= heq flamujt (filtron leje).
- ^= ndërrohet (toggle) flamujt, duke i ndezur ose fikur.

👉 Ky kombinim është shumë i përdorur në sistemet operative, programim të pajisjeve dhe aplikacione ku menaxhohen **opsione binare**.

💠 ^= (XOR assignment) fik e ndez

- Shprehja $x \oplus y$ është e barabartë me $x = x \oplus y$.
- Përdoret për të **ndërruar (toggle)** një flag: nëse është aktiv e fik, nëse është fik e ndez.

```
int permissions = FLAG_READ | FLAG_WRITE; // 0011
permissions ^= FLAG_WRITE; // 0001 (WRITE u fik)
permissions ^= FLAG_WRITE; // 0011 (WRITE u ndez përsëri)
```

Përdorimi tipik

- Manipulimi i **bitëve** në programe të nivelit të ulët.
- Kontrolli i **flamujve** (flags) në sisteme operative ose pajisje hardware.
- Optimizim kur duhen operacione logjike të shpejta mbi bitë.

👉 Me pak fjalë: $x \&= y$ është ekuivalente me $x = x \& y$.

Të gjithë operatorët e tjerë të caktimit bitwise (|=, ^, <<=, >>=)

|= bitwise OR (|) për të **shtuar opsione** pa i fikur ato ekzistuese

Sqarime:

sa del 0101 | 0011

0101 // kjo është 5 në binar

0011 // kjo është 3 në binar

Operacioni **bitwise OR (|)** krahason çdo bit:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

```
0101
0011
----
0111
```

👉 Ky është një shembull tipik se si | përdoret për të kombinuar vlera në nivel bit.

👉 Ja një shembull praktik me **flamuj (flags)** ku përdoret operatori |=:

```
#include <stdio.h>

#define READ    0x01    // 0001
#define WRITE   0x02    // 0010
#define EXEC    0x04    // 0100

int main() {
    int permissions = 0;

    // Aktivizojmë të drejtën për READ
    permissions |= READ;    // permissions = 0001

    // Aktivizojmë edhe WRITE
    permissions |= WRITE;   // permissions = 0011

    // Kontrolllojmë nëse kemi të drejtën për WRITE
    if (permissions & WRITE) {
```

```

        printf("WRITE është aktiv!\n");
    }

    return 0;
}

```

```

#include <stdio.h>

#define READ    0x01    // 0001
#define WRITE   0x02    // 0010
#define EXEC    0x04    // 0100

int main() {
    int permissions = 0;

    // Aktivizojmë të drejtën për READ
    permissions |= READ;    // permissions = 0001

    // Aktivizojmë edhe WRITE
    permissions |= WRITE;    // permissions = 0011

    // Kontrolllojmë nëse kemi të drejtën për WRITE
    if (permissions & WRITE) {
        printf("WRITE është aktiv!\n");
    }

    return 0;
}

```

Çfarë ndodh këtu:

- Fillimisht `permissions = 0` (asnjë e drejtë).
- `permissions |= READ;` ndez bitin e leximit → 0001.
- `permissions |= WRITE;` ndez edhe bitin e shkrimit → 0011.
- Kontrolli `permissions & WRITE` kthen **true**, sepse biti i shkrimit është aktiv.

👉 Pra, **|=** përdoret për të **shtuar opsione** pa i fikur ato ekzistuese. Shumë i dobishëm kur punojmë me **maskat e të drejtave** (permissions) ose konfigurime që kombinohen me bite.

Le të marrim një shembull grafik me **opsione vizuale (flags)**, ku përdoret `|=` për të ndezur efekte të ndryshme në një program grafik:

```

#include <stdio.h>

#define BOLD      0x01    // 0001
#define ITALIC    0x02    // 0010
#define UNDERLINE 0x04    // 0100
#define COLOR     0x08    // 1000

int main() {
    int style = 0;

    // Aktivizojmë BOLD dhe UNDERLINE
    style |= BOLD;        // style = 0001
    style |= UNDERLINE;    // style = 0101

    // Aktivizojmë edhe ngjyrën
    style |= COLOR;        // style = 1101
}

```

```
// Kontrolllojmë cilat opsione janë aktive
if (style & BOLD)      printf("Teksti është BOLD\n");
if (style & ITALIC)    printf("Teksti është ITALIC\n");
if (style & UNDERLINE) printf("Teksti është UNDERLINE\n");
if (style & COLOR)     printf("Teksti ka NGJYRË\n");

return 0;
}
```

Çfarë ndodh këtu:

- Fillimisht `style = 0` (pa efekte).
- Me `!=` ndezim **BOLD**, **UNDERLINE** dhe **COLOR** pa fikur opsionet e mëparshme.
- Kontrolli `style & FLAG` tregon cilat efekte janë aktive.

👉 Ky është një shembull tipik se si `!=` përdoret për të **kombinuar opsione** në një variabël, duke e bërë kodin më të thjeshtë dhe fleksibël.

&= bitwise AND (&) për të filtruar disa bite specifike

Operatori **&=** në gjuhën C është një **operator i caktimit bitwise**. Ai kombinon operatorin bitwise **AND (&)** me operatorin e caktimit (`=`). `&=` për të kontrolluar flag-e (flamuj) në programim

Si funksionon:

- Shprehja `x &= y` është e barabartë me `x = x & y`.
- Pra, merr vlerën aktuale të `x`, bën një operacion **AND bitwise** me `y`, dhe rezultatin e ruan përsëri te `x`.

```
int x = 6; // në binar: 0110
x &= 3;    // në binar: 0011
// Rezultati: 0010 (që është 2 në decimal)
```

👉 Ky operator përdoret shpesh për të **filtruar disa bite specifike** të një variabli, duke i lënë vetëm ato që përputhen me maskën (`y`).

```
#include <stdio.h>

#define FLAG_READ    0x01    // 0001
#define FLAG_WRITE   0x02    // 0010
#define FLAG_EXEC    0x04    // 0100

int main() {
    int permissions = FLAG_READ | FLAG_WRITE; // 0011 (READ + WRITE)

    // Duam të kontrollojmë nëse ka leje për WRITE
    if (permissions & FLAG_WRITE) {
        printf("Ka leje për WRITE\n");
    }

    // Tani heqim lejen për WRITE duke përdorur &=
    permissions &= ~FLAG_WRITE; // 0011 & 1101 = 0001 (vetëm READ)

    if (permissions & FLAG_WRITE) {
        printf("Ka leje për WRITE\n");
    } else {
        printf("Nuk ka më leje për WRITE\n");
    }
}
```

```
    return 0;
}
```

Çfarë ndodh këtu:

- Fillimisht, permissions ka **READ + WRITE**.
- Kontrollon me permissions & FLAG_WRITE → rezulton e vërtetë.
- Pastaj përdorim permissions & ~FLAG_WRITE për të hequr lejen WRITE.
- Në fund, kontrolli tregon se WRITE nuk është më i lejuar.

👉 Pra, &= përdoret shpesh për të **filtruar ose hequr bitet specifike** nga një variabël, duke e bërë shumë të dobishëm në menaxhimin e flamujve, opsioneve ose konfigurimeve.

C Nëse ... Ndryshe

if- Kushtet dhe deklaratat nëse

C ka deklaratat e mëposhtme të kushtëzuara:

- Përdorni për të specifikuar një bllok kodi që do të ekzekutohet, nëse një kusht i specifikuar është **if true**
- Përdorni për të specifikuar një bllok kodi që do të ekzekutohet, nëse i njëjti kusht është **else false**
- Përdorni për të specifikuar një **kusht të ri** për t'u testuar, nëse **kushti i parë** është **else if false**
- Përdorni për të specifikuar **shumë blloqe alternative** të kodit që do të ekzekutohen **switch**

Else Deklarata tjetër

Sintaksa

```
if (condition) {
    // block of code to be executed if the condition is true
} else {
    // block of code to be executed if the condition is false
}
```

Përdorimi i një variabli Boolean

Ju gjithashtu mund ta ruani kushtin në një variabël boolean dhe ta përdorni atë në deklaratë. Kjo mund ta bëjë **kodin më të lehtë** për t'u lexuar: **if...else**

Shembull

```
int time = 20;

bool isDay = time < 18;

if (isDay) {
    printf("Good day.");
} else {
    printf("Good evening.");
}
```

else if - Deklarata tjetër nëse – shtohet kusht

Sintaksa

```
if (condition1) {  
    // block of code to be executed if condition1 is true  
} else if (condition2) {  
    // block of code to be executed if condition1 is false and condition2 is true  
} else {  
    // block of code to be executed if both conditions are false  
}
```

Shembull

```
int time = 16;  
  
if (time < 12) {  
    printf("Good morning.");  
} else if (time < 18) {  
    printf("Good day.");  
} else {  
    printf("Good evening.");  
}
```

// Outputs "Good day."

Nëse vlera e do të ishte 22, asnjë nga kushtet nuk do të ishte e vërtetë, dhe programi do të printonte "Mirëmbrema".

Ternary Operator (trash) Shkurtore- If...Else

Ekziston edhe një dorë stenografike, e njohur si **operatori trash** sepse përdor tre operandë.

if...else

Operatori treshe *kthen një vlerë* bazuar në një kusht: nëse kushti është , ai kthen vlerën e parë; përndryshe, ai kthen vlerën e dytë.**true**

Mund të përdoret për të zëvendësuar shumë rreshta kodi me një rresht të vetëm dhe shpesh përdoret për të zëvendësuar deklaratat e thjeshta: **if...else**

Sintaksa

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Shembull

```
int time = 20;  
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

Nested if – if i ndërfutur

Ju lejon të kontrolloni për një gjendje vetëm nëse një gjendje tjetër është tashmë. if true

Sintaksa

```
if (condition1) {  
    // code to run if condition1 is true
```

```
if (condition2) {  
    // code to run if both condition1 and condition2 are true  
}  
}
```

Shembull

```
int x = 15;  
int y = 25;  
  
if (x > 10) {  
    printf("x is greater than 10\n");  
  
    // Nested if  
    if (y > 20) {  
        printf("y is also greater than 20\n");  
    }  
}
```

Rezultati:

```
x is greater than 10  
y is also greater than 20
```

Shembull

```
int age = 20;  
bool isCitizen = true;  
  
if (age >= 18) {  
    printf("Old enough to vote.\n");  
  
    if (isCitizen) {  
        printf("And you are a citizen, so you can vote!\n");  
    } else {  
        printf("But you must be a citizen to vote.\n");  
    }  
} else {  
    printf("Not old enough to vote.\n");  
}
```

Switch – ne vend te if .. else Deklarata e ndërrimit

Në vend që të shkruani **shumë** deklarata **if..else**, mund ta përdorni deklaratën **switch**

Deklarata **switch** zgjedh një nga shumë blloqet e kodit që do të ekzekutohet:

Sintaksa

```
switch (expression) {  
    case x:  
        // code block  
        break;  
    case y:  
        // code block
```

```
break;  
default:  
    // code block  
}
```

Kështu funksionon:

- Shprehja **switch** vlerësohet një herë
- Vlera e shprehjes krahasohet me vlerat e secilës **case**
- Nëse ka një përputhje, ekzekutohet blloku i lidhur i kodit
- Deklarata **break** shpërthen nga blloku i ndërrimit dhe ndalon ekzekutimin
- Deklarata **default** është **opsionale** dhe specifikon kod për t'u ekzekutuar **nëse nuk ka përputhet me rastet**.

Shembulli i mëposhtëm përdor numrin e ditës së javës për të llogaritur emrin e ditës së javës:

Shembull

```
int day = 4;  
  
switch (day) {  
  
    case 1:  
        printf("Monday");  
        break;  
    case 2:  
        printf("Tuesday");  
        break;  
    case 3:  
        printf("Wednesday");  
        break;  
    case 4:  
        printf("Thursday");  
        break;  
    case 5:  
        printf("Friday");  
        break;  
    case 6:  
        printf("Saturday");  
        break;  
    case 7:  
        printf("Sunday");  
        break;  
}  
  
// Outputs "Thursday" (day 4)
```

break - Fjala kyçe e pushimit

Kur C arrin një fjalë kyçe, ajo del nga blloku i çelësit. **break**

Kjo do të ndalojë ekzekutimin e më shumë testimit të kodit dhe rasteve brenda blloku.

Kur gjendet një ndeshje dhe puna ka mbaruar, është koha për një pushim. Nuk ka nevojë për më shumë testime.

Një break / pushim mund të kursejë shumë kohë ekzekutimi sepse "injon" ekzekutimin të gjithë pjesës tjetër të kodit në bllokun e çelësit switch te ndërrimit.

Ciklet- loops

Loops mund të ekzekutojnë një bllok kodit për sa kohë që një kusht i specifikuar është i vërtetë.

Ciklet janë të dobishme sepse kursejnë kohë, reduktojnë gabimet dhe e bëjnë kodin më të lexueshëm.

While Loop / cikli ndërsa

Laku while përsërit një bllok kodit për sa kohë që një kusht i specifikuar është: true

Sintaksa

```
while (condition) {  
    // code block to be executed  
}
```

Shembull

```
int i = 0;  
  
while (i < 5) {  
    printf("%d\n", i);  
    i++;  
}
```

Shënim: Mos harroni të rrisni variablin **i++** e përdorur në gjendje (), përndryshe Laku nuk do të mbarojë kurrë!

Cikli Do/While - është variant i ciklit while

Cikli **do/while** do të ekzekutojë bllokun e kodit një herë, përpara se të kontrollojë nëse kushti është i vërtetë, atëherë do të përsëritet cikli për sa kohë që kushti **while** është i vërtetë.

Sintaksa

```
do {  
    // code block to be executed  
}  
while (condition);
```

Shembulli i mëposhtëm përdor një lak. Laku do të jetë gjithmonë ekzekutuar të paktën një herë, edhe nëse kushti është i rremë, sepse blloku i kodit ekzekutohet përpara se të testohet gjendja:

do/while

Shembull

```
int i = 0;

do {
    printf("%d\n", i);
    i++;
}
while (i < 5);
// Mos harroni të rrisni variablin e kushtit, përndryshe cikli nuk do të mbarojë kurrë!
```

Shembull – nr invers

```
// A variable with some specific numbers
int numbers = 12345;

// A variable to store the reversed number
int revNumbers = 0;

// Reverse and reorder the numbers
while (numbers) {
    // Get the last number of 'numbers' and add it to 'revNumber'
    revNumbers = revNumbers * 10 + numbers % 10;
    // Remove the last number of 'numbers'
    numbers /= 10;
}
```

Cikli For

Kur e dini saktësisht sa herë dëshironi të kaloni nëpër një bllok kodin, përdorni lakun në vend të një cikli: **forwhile**

Sintaksa

```
for (expression 1; expression 2; expression 3) {
    // code block to be executed
}
```

Shprehja 1 ekzekutohet (një herë) përpara ekzekutimit të bllokut të kodit.

Shprehja 2 përcakton kushtin për ekzekutimin e bllokut të kodit.

Shprehja 3 ekzekutohet (çdo herë) pasi të jetë ekzekutuar blloku i kodit.

Shembull Shembulli më poshtë do të printojë numrat 0 deri në 4:

```
int i;

for (i = 0; i < 5; i++) {
    printf("%d\n", i);
}
```

Shembull i shpjeguar

- Deklarata 1 vendos një variabël përpara se të fillojë cikli: **int i = 0**

- Deklarata 2 përcakton kushtin që cikli të ekzekutohet: . Nëse kushti është i vërtetë, laku do të fillojë përsëri, nëse është i rremë, laku do të përfundojë. `i < 5`
- Deklarata 3 rrit një vlerë sa herë që është ekzekutuar blloku i kodit në lak: `i++`

Break / continue ndal dhe Vazhdo

Në gjuhën C, `break` dhe `continue` janë dy fjalë kyçe që përdoren për të kontrolluar rrjedhën e ekzekutimit brenda cikleve (`for`, `while`, `do...while`) dhe strukturave `switch`.

break

- Përdoret për të **dalë menjëherë** nga një cikël ose nga një `switch`.
- Kur ekzekutohet, cikli ndërpritet dhe programi vazhdon me kodin pas tij.

Shembull:

```
c
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // ndalon ciklin kur i = 5
    }
    printf("%d\n", i);
}
```

continue

- Përdoret për të **kapërcyer pjesën e mbetur të trupit të ciklit** dhe për të kaluar direkt tek iterimi tjetër.
- **Nuk e ndalon ciklin**, thjesht e anashkalon pjesën e mbetur të kodit për atë iterim.

Shembull:

```
c
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // kalon tek iterimi tjetër kur i = 5
    }
    printf("%d\n", i);
}
```

👉 Ky kod do të printojë numrat 0 deri 9, por **do të anashkalojë 5**.

Mirë të mbani mend:

- `break` = ndaloni lakun plotësisht.
- `continue` = kapërceni këtë raund, por vazhdoni ciklin.

Ju gjithashtu mund të kombinoni dhe `break continue`

- Ky shembull anashkalon printimin 2 dhe ndalon ciklin në 4:

Shembull

```
int i;
```

```
for (i = 0; i < 6; i++) {  
    if (i == 2) {  
        continue;  
    }  
    if (i == 4) {  
        break;  
    }  
    printf("%d\n", i);  
}
```

Vektorët- grupet(numra)groups ose [Vargjet\(tekst\)/strings](#) / Vlera te shumëfishta te njëjtit tip ne te njëjtin variabël

Për të krijuar një grup, përcaktoni llojin e të dhënave (si) dhe specifikoni emrin të grupit të ndjekur nga **klapa katrore** []. **int**

Për të hyrë në një element të vargut, referojuni numrit të tij **të indeksit**.

```
int myNumbers[] = {25, 50, 75, 100};  
printf("%d", myNumbers[0]);           //jep 25
```

```
int myNumbers[] = {10, 25, 50, 75, 100};  
printf("%lu", sizeof(myNumbers));     // Prints 20  
4 x 5 (4 bytes eshte tipi integer x 5 elements) = 20 bytes.
```

Vendosni madhësinë e grupit

Një mënyrë tjetër e zakonshme për të krijuar grup /vargje (tekst), është të specifikoni madhësinë e grupit dhe të shtoni Elementet më vonë:

Shembull

```
// Declare an array of four integers:  
int myNumbers[4];  
  
// Add elements  
myNumbers[0] = 25;  
myNumbers[1] = 50;  
myNumbers[2] = 75;  
myNumbers[3] = 100;
```

Duke përdorur këtë metodë, **duhet të dini paraprakisht numrin e elementeve të grupit**, në mënyrë që programi të ruajë memorie të mjaftueshme.

Ju nuk jeni në gjendje të ndryshoni madhësinë e grupit pas krijimit.

Merrni numrin e elementeve

Nëse dëshironi të zbuloni se sa elementë ka një grup, mund të përdorni këtë formulë, e cila ndan madhësinë totale të grupit me madhësinë e një elementi:

Shembull

```
int myNumbers[] = {10, 25, 50, 75, 100};  
  
int length = sizeof(myNumbers) / sizeof(myNumbers[0]);  
  
printf("%d", length); // Prints 5
```

Cikli përmes një grupi (elementet numra) / vargu (elementet tekst)

Ju mund të përdorni një lak për të kaluar nëpër elementët e një grupi duke shkruar madhësinë e grupit në kushtin e lakut (në këtë shembull grupi ka 4 elementë, kështu që ne përdorim). Megjithatë, kjo nuk është ideale, pasi do të funksionojë vetëm për grupe të një madhësie të caktuar: `for i < 4`

Shembull

```
int myNumbers[] = {25, 50, 75, 100};  
int i;  
  
for (i = 0; i < 4; i++) {  
    printf("%d\n", myNumbers[i]);  
}
```

Bërja e cikleve më të mira

Në [kapitullin e mëparshëm](#), ju mësuat se si të llogarisni numrin e elementeve në një grup duke përdorur formulën. Tani mund ta përdorim atë për të shkruar cikle që funksionojnë për grupe / vargje të çdo madhësie. Kjo është më fleksibël dhe më e qëndrueshme: `sizeof`

Shembull

```
int myNumbers[] = {25, 50, 75, 100};  
  
int length = sizeof(myNumbers) / sizeof(myNumbers[0]);  
int i;  
  
for (i = 0; i < length; i++) {  
    printf("%d\n", myNumbers[i]);  
}
```

Ky cikël do të funksionojë automatikisht pavarësisht se sa elementë ka grupi!

Përmbledhje: Përdorni gjithmonë formulën kur kaloni nëpër vargje. Kjo i bën ciklet tuaja të përshtaten automatikisht me madhësinë e grupit. `sizeof`

Shembull i jetës reale

Për të demonstruar një shembull praktik të përdorimit të grupeve, le të krijojmë një program që llogarit mesataren e moshave të ndryshme:

Shembull

```
// An array storing different ages
int ages[] = {20, 22, 18, 35, 48, 26, 87, 70};

float avg, sum = 0;
int i;

// Get the length of the array
int length = sizeof(ages) / sizeof(ages[0]);

// Loop through the elements of the array
for (i = 0; i < length; i++) {
    sum += ages[i];
}

// Calculate the average by dividing the sum by the length
avg = sum / length;

// Print the average
printf("The average age is: %.2f", avg);
```

Vektorët- Grupe shumëdimensionale

Në kapitullin e mëparshëm, ju mësuat për [vargjet](#), të cilat njihen edhe si **grupe me një dimension**. Këto janë të shkëlqyera dhe diçka që do ta përdorni shumë gjatë programimit në C. Megjithatë, nëse dëshironi të ruani të dhënat si një formë tabelare, si një tabelë me rreshta dhe kolona, duhet të njiheni me **grupet shumëdimensionale**.

Një grup shumëdimensional është në thelb një grup grupesh.

Vargjet mund të kenë çdo numër dimensionesh. Në këtë kapitull, ne do të prezantojmë më të zakonshmet; Vargje dydimensionale (2D).

Vektorët- grupe dydimensionale

Një grup 2D njihet gjithashtu si matricë (një tabelë rreshtash dhe kolonash).

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

Dimensioni i parë përfaqëson numrin e rreshtave **[2]**, ndërsa i dyti dimension i përfaqëson numrin e kolonave **[3]**. Vlerat vendosen në renditja e rreshtave dhe mund të vizualizohet kështu:

	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	1	4	2
ROW 1	3	6	8

Hyni në elementët e një grupi 2D

Për të hyrë në një element të një grupi dydimensional, duhet të specifikoni indeksin numri i rreshtit dhe kolonës.

Kjo deklaratë akseson vlerën e elementit në **rreshtin e parë (0)** dhe **kolonën e tretë (2)** të grupit **të matricës**.

Shembull

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
printf("%d", matrix[0][2]); // Outputs 2 pra shto me 1 per rresht e kolone
```

Shembull

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
matrix[0][0] = 9;

printf("%d", matrix[0][0]); // Now outputs 9 instead of 1
```

Ndryshoni elementet në një grup 2D

Për të ndryshuar vlerën e një elementi, referojuni numrit të indeksit të elementit në secilin prej dimensioneve:

Shembulli i mëposhtëm do të ndryshojë vlerën e elementit në **rreshtin e parë (0)** dhe kolonën e **parë (0)**:

Shembull

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
matrix[0][0] = 9;

printf("%d", matrix[0][0]); // Now outputs 9 instead of 1
```

Ciklo nëpër një grup 2D

Për të kaluar nëpër një grup shumëdimensional, ju duhet një cikël për secilin prej dimensioneve të grupit.

Shembull

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

```
int i, j;
for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
        printf("%d\n", matrix[i][j]); //pa & !!!
    }
}
```

Grupet tredimensionale

Ju gjithashtu mund të deklaroni grupe me më shumë se dy dimensione:

Shembull

```
// A 3D array with 2 blocks, each with 4 rows and 3 columns
```

```
int example[2][4][3];
```

Kjo krijon një grup 3D me:

- 2 blloqet (indeksi i parë)
- 4 rreshta për bllok (indeksi i dytë)
- 3 Kolonat për rresht (indeksi i tretë)

Kur të përdorni grupe shumëdimensionale

Vargjet shumëdimensionale janë të dobishme kur të dhënat tuaja janë të rregulluara në rreshta dhe kolona, si një tabelë, rrjet ose matricë.

Çdo dimension shtesë shton një nivel tjetër strukture:

- **Grupet 2D** (si) janë të shkëlqyera për ruajtjen e gjërave të tilla si rezultatet, tabelat e lojërave ose tabelat `int scores[3][4]`
- **Grupet 3D** (si) mund të përfaqësojnë struktura më komplekse si një grup tavolinash ose nivelesh në një lojë `int cube[2][3][4]`

Detyra – vektor numrash = grup:

Sfida: Krijoni një grup të thjeshtë

Testoni të kuptuarit tuaj të **grupeve C** duke plotësuar një sfidë të vogël kodimi.

Udhëzime

Brenda, kryeni hapat e mëposhtëm: `main()`

1. Deklaroni një grup të emërtuar me 5 elementë `int numbers`
2. Vendosni elementin e parë () në `numbers[0]10`

3. Vendosni elementin e fundit () në `numbers[4]` 50
4. Printoni elementin e parë dhe të fundit duke përdorur `printf`

Zgjidhje:

```
#include <stdio.h>

int main() {
    // 1) Declare an int array named numbers with 5 elements
    int numbers[5];
    // 2) Set numbers[0] to 10
    numbers[0]=10;
    // 3) Set numbers[4] to 50
    numbers[4] =50;
    // 4) Print numbers[0]
    printf("\n%d", numbers[0]);
    // 5) Print numbers[4]
    printf("\n%d", numbers[4]);
    return 0;
}
```

10

50

Vektorët strings = Vargjet C

Vargjet përdoren për ruajtjen e tekstit/karaktereve.

Për shembull, "Hello World" është një varg karakteresh.

Ndryshe nga shumë gjuhë të tjera programimi, C nuk ka një lloj vargu: **String type** për të krijuar lehtësisht variabla vargjesh. Në vend të kësaj, duhet të përdorni llojin `char` dhe të krijoni një vektor karakteresh për të bërë një varg në C:

```
char greetings[] = "Hello World!";
```

Vini re se duhet të përdorni thonjëza të dyfishta ("").

Për të nxjerrë vargun, mund të përdorni funksionin `printf()` së bashku me specifikuesin e formatit `%s` për t'i thënë C se tani po punojmë me vargje:

Shembull

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

Vargjet e aksesit

Meqenëse vargjet janë në të vërtetë grupe në C, mund të përdorni një varg duke iu referuar numrit të indeksit të tij brenda kllapave katrore. `[]`

Ky shembull printon **karakterin e parë (0)** në **përshëndetje**:

Shembull

```
char greetings[] = "Hello World!";
printf("%c", greetings[0]);           //jep H
```

Vini re se duhet të përdorim specifikuesin e formatit për të printuar një **karakter të vetëm.%c**

Modifikoni vargjet

Për të ndryshuar vlerën e një karakteri specifik në një varg, referojuni numrit të indeksit dhe përdorni **thonjëza teke**: ' '

Shembull

```
char greetings[] = "Hello World!";
greetings[0] = 'J';
printf("%s", greetings);
// Outputs Jello World! instead of Hello World!
```

Cikël në varg - `int length = sizeof(carName) / sizeof(carName[0]);`

Ju gjithashtu mund të kaloni nëpër karakteret e një vargu, duke përdorur një ciklin: **for**

Shembull

```
char carName[] = "Volvo";
int i;

for (i = 0; i < 5; ++i) {
    printf("%c\n", carName[i]);
}
```

Dhe siç specifikuam në kapitullin e [ciklit të vargjeve](#), duhet të përdorni *formulën sizeof* (në vend të manualisht shkruani madhësinë e grupit në kushtin e lakut) për ta bërë lakun më të qëndrueshëm: `(i < 5)`

Shembull

```
char carName[] = "Volvo";
int length = sizeof(carName) / sizeof(carName[0]);
int i;

for (i = 0; i < length; ++i) {
    printf("%c\n", carName[i]);
}
Jep:
V
o
l
v
o
```

Një mënyrë tjetër për të krijuar vargje

Në shembujt e mësipërm, ne përdorëm një "varg fjalë për fjalë" për të krijuar një varg variabël. Kjo është mënyra më e lehtë për të krijuar një varg në C.

Duhet të keni parasysh gjithashtu se mund të krijoni një varg me një grup karakteresh. Ky shembull do të prodhojë të njëjtin rezultat si shembulli në fillim të kësaj faqeje:

Shembull

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!', '\0'};
printf("%s", greetings);
```

Pse përfshijmë karakterin `\0` në fund? Kjo është i njohur si "karakter i përfundimtar null", dhe duhet të përfshihet gjatë krijimit të vargut duke përdorur këtë metodë. Ai i thotë C se ky është fundi i vargut.

```
char greetings[] = {'H', 'i', '\0'};
char greetings2[] = "Hi";
```

```
printf("%zu\n", sizeof(greetings)); // Outputs 3
printf("%zu\n", sizeof(greetings2)); // Outputs 3
```

Dallimet

Dallimi midis dy mënyrave të krijimit të vargjeve, është se metoda e parë është më e lehtë për të shkruar, dhe nuk keni nevojë të përfshini karakterin, pasi C do ta bëjë atë për ju. `\0`

Ju duhet të vini re se madhësia e të dy grupeve është e njëjtë: Ata të dy kanë **3**

karaktere (hapësira gjithashtu llogaritet si karakter meqë ra fjala), duke përfshirë karakterin: `\0`

Vargjet- Karaktere speciale- sekuenca largimit – vazhdim

Për shkak se vargjet duhet të shkruhen brenda thonjzave, C do ta keqkuptojë këtë varg, dhe gjeneroni një gabim:

```
char txt[] = "We are the so-called "Vikings" from the north.";
```

Zgjidhja për të shmangur këtë problem, është përdorimi i **karakterit backslash escape**.

`\'` Fut një karakter me thonjëza të dyfishta `\\` Fut një karakter të kundërt (`\`)

Sekuencat e largimit fillojnë me një prerje të kundërt dhe përfaqësojnë karaktere speciale që nuk mund të shtypen drejtpërdrejt.

Escape character	Result	Description
<code>\'</code>	<code>'</code>	Single quote
<code>\"</code>	<code>"</code>	Double quote

\\ \ Backslash

Shembull - Sekuenca fut një thonjëzë të dyfishtë në një varg: \"

```
char txt[] = "We are the so-called \"Vikings\" from the north.";
```

Shembull - Sekuenca fut një thonjëzë të vetme në një varg: \'

```
char txt[] = "It\'s alright.";
```

Shembull - Sekuenca fut një prerje të vetme të kundërt në një varg: \\

```
char txt[] = "The character \\ is called backslash.";
```

\n New Line

\t Tab

\0 Null

Funksionet e vargut C – string functions

Për t'i përdorur ato, duhet të përfshini skedarin e kokës në programin tuaj: <string.h>

```
#include <string.h>
```

1. Gjatësia e vargut

Për shembull, për të marrë gjatësinë e një vargu, mund të përdorni funksionin: **strlen()**

Shembull

```
char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%zu\n", strlen(alphabet)); // jep 26 sillet ndryshe me sizeof
printf("%zu\n", sizeof(alphabet)); //jep 27 se përfshin edhe \0
```

Është gjithashtu e rëndësishme që të dini se **sizeof** gjithmonë do të kthejë madhësinë e memories (në bajt / bytes), dhe jo gjatësinë aktuale të vargut:

Shembull

```
char alphabet[50] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%zu\n", strlen(alphabet)); // 26
printf("%zu\n", sizeof(alphabet)); // 50
```

```
printf("%zu\n", sizeof(myChar)); = 1 (byte)
```

2. Vargjet e lidhura

Për të lidhur (kombinuar) dy vargje, mund të përdorni funksionin: **strcat()**

Vini re se madhësia e **str1** duhet të jetë mjaft e madhe për të **ruajtur rezultatin e dy vargje të kombinuara** (20 në shembullin tonë).

Shembull

```
char str1[20] = "Hello ";
char str2[] = "World!";
```

```
// Concatenate str2 to str1 (result is stored in str1)
strcat(str1, str2);
```

```
// Print str1
printf("%s", str1);
```

ose:

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

int main() {
    char str1[20] = "Hello ";
    char str2[] = "World!";

    // Concatenate str2 to str1 (the result is stored in str1)
    strcat(str1, str2);

    // Print str1
    printf("%s", str1);           //Hello World!
    printf("\n%zu", sizeof(str2)); //del 7 = 6 +   \o byte

    return 0;
}
```

3. Kopjo vargjet

Për të kopjuar vlerën e një vargu në një tjetër, mund të përdorni funksionin: **strcpy()**

Shembull

```
char str1[20] = "Hello World!";
char str2[20];
```

```
// Copy str1 to str2
strcpy(str2, str1);
```

```
// Print str2
printf("%s", str2);
```

4. Krahasoni vargjet

Për të krahasuar dy vargje, mund të përdorni funksionin. **strcmp()**

Ai kthehet nëse dy vargjet janë të barabarta, përndryshe një vlerë që nuk është 0:0

Shembull

```
char str1[] = "Hello";
char str2[] = "Hello";
char str3[] = "Hi";
```

```
// Compare str1 and str2, and print the result
printf("%d\n", strcmp(str1, str2)); // Returns 0 (the strings are equal)
```

```
// Compare str1 and str3, and print the result
printf("%d\n", strcmp(str1, str3)); // Returns -4 (the strings are not equal)
```

```
0
-4
```

```
#include <stdio.h>

int main() {

    // Write your code here
    char message[]="Hello";
    printf("%s",message);

    return 0;
}
```

Udhëzime

Brenda, kryeni hapat e mëposhtëm:**main()**

1. Deklaroni një variabël të quajtur që duhet të ruajë një varg**message**
2. Ruani tekstin në varg**Hello**
3. Printoni vargun duke përdorur specifikuesin e duhur të formatit

scanf- Hyrja e përdoruesit

Ju tashmë keni mësuar se **printf()** përdoret për të **nxjerrë vlerat** në C.

Për të marrë **të dhënat e përdoruesit**, mund të përdorni funksionin:**scanf()**

Shembull

Nxjerr një numër të futur nga përdoruesi:

```
// Create an integer variable that will store the number we get from the user
int myNum;

// Ask the user to type a number
printf("Type a number: \n");

// Get and save the number the user types
scanf("%d", &myNum);

// Output the number the user typed
printf("Your number is: %d", myNum);
```

Funksioni **scanf()** merr **dy argumente**: specifikuesin e formatit të variablit **%d** (në shembullin e mësipërm) dhe operatori i referencës (**&myNum**), i cili ruan adresën e memories së ndryshores.

Këshillë: Do të mësoni më shumë rreth adresave dhe [funksioneve](#) të [kujtesës](#) në kapitullin tjetër.

Hyrje të shumta

Funksioni `scanf()` gjithashtu lejon shumë inputet (një numër i plotë dhe një karakter në shembullin e mëposhtëm):

Shembull

```
// Create an int and a char variable
int myNum;
char myChar;

// Ask the user to type a number AND a character
printf("Type a number AND a character and press enter: \n");

// Get and save the number AND character the user types
scanf("%d %c", &myNum, &myChar);

// Print the number
printf("Your number is: %d\n", myNum);

// Print the character
printf("Your character is: %c\n", myChar);
```

Shembull - Nxjerr emrin e një përdoruesi:

```
// Create a string
char firstName[30];

// Ask the user to input some text
printf("Enter your first name: \n");

// Get and save the text
scanf("%s", firstName);

// Output the text
printf("Hello %s", firstName);
```

Shënim: Kur punoni me **vargje /string** në **, duhet të specifikoni madhësinë e vargu/grupi** (ne kemi përdorur një numër shumë të lartë, 30 në shembullin tonë, por të paktën atëherë jemi sigurisht se do të ruajë karaktere të mjaftueshme për emrin e parë), dhe ju nuk keni për të përdorur operatorin e referencës `()&`.

Megjithatë, funksioni `scanf()` ka disa kufizime: ai merr parasysh **hapësirën** (hapësira e bardhë, skedat, etj) si një **karakter përfundimtar**, që do të thotë se mund të shfaqë vetëm një fjalë të vetme (edhe nëse shkruani shumë fjalë). Për shembull:

Shembull

```
char fullName[30];

printf("Type your full name: \n");
scanf("%s", &fullName);
```

```
printf("Hello %s", fullName);

// Type your full name: John Doe
// Hello John
```

Nga shembulli i mësipërm, do të prisnit që programi të printonte "John Doe", por printon vetëm "John".

Kjo është arsyeja pse, kur punojmë me vargje, ne shpesh përdorim funksionin `fgets()` për të **lexuar një rresht teksti**. Vini re se duhet të përfshini sa vijon argumentet: emri i variablit të vargut, (*string_name*) dhe : `sizeof` dhe `stdin`

Shembull

```
char fullName[30];

printf("Type your full name: \n");
fgets(fullName, sizeof(fullName), stdin);

printf("Hello %s", fullName);

// Type your full name: John Doe
// Hello John Doe
```

Përdorni funksionin `scanf()` për të marrë një fjalë të vetme si hyrje dhe përdorni `fgets()` për fjalë të shumta.

```
#include <stdio.h>

int main() {
    // Create a string
    char fullName[30];

    // Ask the user to input some text (full name)
    printf("Shkruani emrin tuaj të plotë dhe shtypni enter:: \n");

    // Get the text
    fgets(fullName, sizeof(fullName), stdin);

    // Output the text
    printf("Hello %s", fullName);

    return 0;
}
```

Shkruani emrin tuaj të plotë dhe shtypni enter:

Përsëritje aaa bbbb ccc

Adresa e kujtesës - operatori i referencës &

Kur krijohet një variabël në C, variablit i caktohet një adresë memorie.

Adresa e kujtesës është **vendndodhja** e vendit ku ruhet variabli **në kompjuter**.

Kur i caktojmë një vlerë ndryshore, ajo ruhet në këtë adresë memorie.

Për të hyrë në të, përdorni operator (&) e **referencës** dhe rezultati përfaqëson **vendin** ku ruhet variabla:

Shembull

```
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

Shënim: Adresa e kujtesës është në formë heksadecimale (0x..). Ndoshta nuk do të marrë të njëjtin rezultat në programin tuaj, pasi kjo varet nga ku Variabla ruhet në kompjuterin tuaj.

Duhet të keni parasysh gjithashtu se shpesh **&myAge** quhet "**pointers**" ose "tregues". Një tregues në thelb ruan adresën e kujtesës së një ndryshoreje si vlerën e saj. Për të printuar vlerat e pointers /treguesit, ne përdorim **specifikuesin e formatit**. **%p**

Do të mësoni shumë më tepër rreth **treguesve** në kapitullin tjetër.

Pse është e dobishme të dini adresën e kujtesës?

Treguesit janë të rëndësishëm në C, sepse na lejojnë të manipulojmë të dhënat në memorien e kompjuterit - **kjo mund të zvogëlojë kodin dhe të përmirësojë performancën**.

Treguesit janë një nga gjërat që e bëjnë C-në të dallohet nga të tjerët gjuhët e programimit, si **Python** dhe **Java**.

Pointers – krijim variabli pointer me operatorin *

Ju mësuat nga kapitulli i mëparshëm, se ne mund të marrim **adresën e kujtesës** së një variabli me **Operator reference**: **&**

Shembull

```
int myAge = 43; // an int variable

printf("%d", myAge); // Outputs the value of myAge (43)
printf("%p", &myAge); // Outputs the memory address of myAge (0x7ffe5367e044)
```

Një pointer/ **tregues ose shenjues** është një **variabël** që ruan **adresën e kujtesës** së një variabli tjetër si vlerën e saj.

Një **variabël pointer/shënjes shënjon datatipin** (si **int**) të se njëjtit **tip të dhënash** dhe **krijohet** me operatorin. *

Adresa e variablit me të cilin po punoni i caktohet treguesit:

Shembull

```
#include <stdio.h>

int main() {
    int myAge = 43; // An int variable
    int* ptr = &myAge; // A pointer variable, with the name ptr, that stores the address of myAge

    // Output the value of myAge (43)
    printf("%d\n", myAge);

    // Output the memory address of myAge (0x7ffe5367e044)
    printf("%p\n", &myAge);

    // Output the memory address of myAge with the pointer (0x7ffe5367e044)
    printf("%p\n", ptr);

    return 0;
}
```

43
0x7ffe5367e044
0x7ffe5367e044

Shembull i shpjeguar

Krijoni një variabël pointer /shenjues me emrin **ptr**, që **tregon një int** variabël (**myAge**). Vini re se lloji i shenjuesit duhet të përputhet me llojin e variablit që jeni duke punuar me (**int** në shembullin tonë).

Përdorni operatorin **&** për të ruajtur adresën e memories së variablit **myAge** dhe caktoni atë treguesit pointer.

Tani, **ptr** mban vlerën **myAge** e adresës së kujtesës.

De=*-referencë=& – operatori i de-referencës *

Në shembullin e mësipërm, ne përdorëm variablin pointer për të marrë adresën e memories së një variabli (përdoret së bashku me operatorin **e referencës &**).

Ju gjithashtu mund të merrni **vlerën** e variablit në të cilin tregon pointer, duke përdorur operatorin (operatorin **e dereferencës**): *

Vini re se shenja * mund të jetë konfuze këtu, pasi bën **dy gjëra të ndryshme** Në kodin tonë:

- Kur përdoret në deklaratë (**int* ptr**), **krijon një variabël pointer /shenjues**.
- Kur nuk përdoret në deklaratë, vepron si **operator de-reference**.

1. Cili operator përdoret për të marrë **vlerën** e ruajtur në adresën e memories që tregon një tregues? *
2. Cili është qëllimi i operatorit? & Për të marrë **adresën e memories** së një **variabli**

Mirë të dini: Ka dy mënyra për të deklaruar treguesin variablat në C:

```
int* myNum;
int *myNum;
```

Shënime mbi pointerat / treguesit

Pointer janë një nga gjërat që e bëjnë C-në të dallohet nga të tjerët gjuhët e programimit, si [Python](#) dhe [Java](#).

Ato janë të rëndësishme në C, sepse na lejojnë të manipulojmë të dhënat në memorien e kompjuterit. Kjo mund të zvogëlojë kodin dhe të përmirësojë performancën. Nëse jeni njohur me strukturat e të dhënave si **listat, pemët dhe grafikët**, duhet të dini se treguesit janë veçanërisht të dobishme për zbatimin e tyre. Dhe ndonjëherë edhe duhet të përdorni tregues, për shembull kur punoni me [skedarë](#) dhe [menaxhimin e memories](#).

Por kini kujdes; Treguesit duhet të trajtohen me kujdes, pasi është e mundur të dëmtohen të dhënat e ruajtura në adresa të tjera memorie.

[pointers dhe grupet](#) = vektorët numerikë

[shënjesit pointers dhe vargjet](#)

Ju gjithashtu mund të përdorni tregues për të hyrë në [grupe](#).

Konsideroni grupin e mëposhtëm të numrave të plotë:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};
```

Ju mësuat nga [kapitulli i grupeve](#) se mund të kaloni nëpër elementët e grupit me një lak: **for**

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
    printf("%d\n", myNumbers[i]);
}
```

Rezultati:

```
25
50
75
100
```

Në vend që të printojmë vlerën e secilit element të grupit, le të printojmë adresën e memories së secilit element të grupit:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
    printf("%p\n", &myNumbers[i]);
}
```

Rezultati:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
```

Vini re se numri i fundit i adresës së kujtesës së secilit prej elementeve është ndryshe, me një shtesë prej 4.

Kjo është për shkak se madhësia e një lloji **int** është zakonisht **4 bajt**, mbani mend:

Shembull

```
// Create an int variable
int myInt;

// Get the memory size of an int
printf("%zu", sizeof(myInt));
```

Rezultati:

```
4
```

Pra, nga "shembulli i adresës së kujtesës" më sipër, mund të shihni se **përpiluesi rezervon 4 bajt** memorie për çdo element të grupit, që do të thotë se l gjithë grupi zë 16 bajt ($4 * 4$) memorie:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the size of the myNumbers array
printf("%zu", sizeof(myNumbers));
```

Rezultati:

```
16
```

Si lidhen pointers /shënjes me vargjet

Ok, pra, cila është marrëdhënia midis treguesve dhe vargjeve? Epo, në C, **emri i vektorit**, është në fakt një shënjes / pointer tek **elementi i parë i** vektorit (grupit).

Të hutuar? Le të përpiqemi ta kuptojmë më mirë këtë dhe të përdorim "shembullin tonë të adresës së kujtesës" më sipër përsëri.

Adresa e kujtesës së elementit të parë është e njëjtë me **emrin e vektorit (grupit)**:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the memory address of the myNumbers array
printf("%p\n", myNumbers);

// Get the memory address of the first array element
printf("%p\n", &myNumbers[0]);
```

Rezultati:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f0
```

Kjo në thelb do të thotë që ne mund të punojmë me vargje përmes treguesve!

Si? Meqenëse myNumbers është një tregues për elementin e parë në myNumbers, mund të përdorni operatorin për të hyrë në të:*

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the first element in myNumbers
printf("%d", *myNumbers);
```

Rezultati:

```
25
```

Për të hyrë në pjesën tjetër të elementëve në myNumbers, mund të rrisni treguesin/grupin (+1, +2, etj):

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the second element in myNumbers
printf("%d\n", *(myNumbers + 1));

// Get the value of the third element in myNumbers
printf("%d", *(myNumbers + 2));

// and so on..
```

Rezultati:

```
50
75
```

Ose kaloni nëpër të:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
int i;

for (i = 0; i < 4; i++) {
    printf("%d\n", *(ptr + i));
}
```

Rezultati:

```
25
50
75
100
```

Është gjithashtu e mundur të ndryshoni vlerën e elementeve të grupit me tregues:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};

// Change the value of the first element to 13
*myNumbers = 13;

// Change the value of the second element to 17
*(myNumbers + 1) = 17;

// Get the value of the first element
printf("%d\n", *myNumbers);

// Get the value of the second element
printf("%d\n", *(myNumbers + 1));
```

Rezultati:

```
13
17
```

Kjo mënyrë e punës me vargjet mund të duket pak e tepërt. Sidomos me vargje të thjeshta si në shembujt e mësipërm. Megjithatë, për grupe të mëdha, mund të jetë shumë më efikas për të hyrë dhe manipuluar grupet me pointera.

Konsiderohet gjithashtu më e shpejtë dhe më e lehtë për të hyrë në [grupe dydimensionale](#) me pointer.

Dhe meqenëse vargjet janë në të vërtetë grupe, mund të përdorni gjithashtu pointer për të hyrë tek [vargjet](#).

Tani për tani, është mirë që e dini se si funksionon kjo. Por siç specifikua në kapitullin e mëparshëm; **pointerat (shenjues) duhet të trajtohen me kujdes**, pasi është e mundur të mbishkruhen të dhëna të tjera të ruajtura në kujtesë.

Aritmetika e pointerave /shënjesve

Aritmetika e pointer do të thotë ndryshimi i vlerës së një pointer (shënjesi) për ta bërë atë të tregojë një element tjetër në memorie.

Siç e pamë në faqen e [mëparshme](#), elementët e grupit ruhen pranë njëri-tjetrit në memorie. Pra, nëse një tregues tregon një element, shtimi e zhvendos atë në tjetrin: **1**

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};
int *p = myNumbers; // points to myNumbers[0]

printf("%d\n", *p); // 25
printf("%d\n", *(p + 1)); // 50
printf("%d\n", *(p + 2)); // 75
printf("%d\n", *(p + 3)); // 100
```

Rritja dhe zvogëlimi

Ju mund të lëvizni pointerin me **++** dhe **--** (dhe me **+=** / **-=**):

Shembull

```
int myNumbers[3] = {10, 20, 30};
int *p = myNumbers; // myNumbers[0]

printf("%d\n", *p); // 10
p++; // move to myNumbers[1]
printf("%d\n", *p); // 20
p--; // back to myNumbers[0]
printf("%d\n", *p); // 10

p += 2; // jump to myNumbers[2]
printf("%d\n", *p); // 30
```

Kjo tregon se ju mund të lëvizni pointerin ashtu si një numërues: sa herë që përdorni **p++** ose **p--** Ai kalon në elementin tjetër ose të mëparshëm në vektor.

Zbritja e pointer (distanca)

Mund të zbritni dy pointera që tregojnë elementë në **të njëjtin grup** për të zbuluar se sa elementë janë midis tyre:

Shembull

```
int myNumbers[5] = {10, 20, 30, 40, 50};
int *start = &myNumbers[1]; // points to 20
int *end = &myNumbers[4];    // points to 50

printf("%ld\n", end - start); // 3 elements apart
```

Këtu:

- **start** tregon për elementin e dytë (20).
- **end** tregon për elementin e pestë (50).
- **end - start** jep numrin e elementeve midis tyre: 3.

Shënim: Zbritja e pointer funksionon vetëm kur të dy pointrat shënojnë në të njëjtin vektor. Rezultati matet në elementë, jo në **bajt/byte**.

Aritmetika e treguesit varet nga tipi ose lloji

Jo të gjithë pointerat lëvizin në të njëjtën mënyrë.

Kur shtoni **1** në një pointer, ai **lëviz** me hap përpara **sipas madhësisë së gjësë** që tregon - jo vetëm me 1 bajt.

Për shembull:

- Një pointer **int*** lëviz me madhësinë e një numri të plotë (zakonisht 4 bajt).
- Një pointer **char*** lëviz sipas madhësisë së një karakteri (1 bajt).

Pra, nëse të dy treguesit fillojnë në adresën e kujtesës: **1000**

- **int*** → **p + 1** do të lëviznin për të adresuar **1004**
- **char*** → **p + 1** do të lëviznin për të adresuar **1001**

Kjo tregon se lëvizja e pointer **varet nga lloji i të dhënave** për të cilat tregon - jo në numrin që shtoni:

Shembull

```
int myNumbers[2] = {1, 2};
char letters[] = "Hi"; // 'H', 'i', '\0'

int *pi = myNumbers; // int pointer
char *pc = letters;  // char pointer

printf("%p\n", (void*)pi);
printf("%p\n", (void*)(pi + 1)); // moves by sizeof(int) (4 bytes)
printf("%p\n", (void*)(pi + 2)); // moves by sizeof(int) (4 bytes)

printf("%p\n", (void*)pc);
printf("%p\n", (void*)(pc + 1)); // moves by 1 byte
```

Cikle me pointera

Në [kapitullin e mëparshëm](#), ju mësuat se si të kaloni nëpër një vektor duke përdorur `*(ptr + i)`

Tani le të shohim një mënyrë tjetër - duke **lëvizur vetë pointerin** brenda ciklit. Sa herë që pointer rritet (`p++`), ai lëviz në elementin tjetër në memorie:

Shembull

```
int myNumbers[4] = {25, 50, 75, 100};
int *p = myNumbers;    // start of array

for (int i = 0; i < 4; i++) {
    printf("%d\n", *p);
    p++; // move to next element
}
```

Ja çfarë ndodh në çdo lak:

- `*p` jep vlerën aktuale të elementit.
- `p++` zhvendos treguesin në elementin tjetër në grup.
- Nuk nevojitet indeks i grupit () - treguesi mban gjurmët e pozicionit. `i`

Këshillë: Kjo mënyrë e ciklit është e zakonshme kur punoni drejtpërdrejt me kujtesën, pasi vetë treguesi lëviz nëpër grup në vend që të përdorë një numër indeksi.

Gabimet e zakonshme që duhen shmangur

- **Përdorimi i llojit të gabuar:** Mos harroni se lëvizja e pointer varet nga lloji i tij. An `int*` lëviz në hapa 4-bajt, por një `char*` lëviz 1 bajt në të njëjtën kohë. Përzierja e tyre do të tregojë vendndodhjet e gabuara të kujtesës.
- **Pointerat e painicializuar:** Gjithmonë sigurohuni që pointeri të tregojë diçka reale përpara se ta përdorni. Përdorimi i një treguesi që nuk tregon askund mund të prishë programin tuaj.
- **Duke dalë jashtë kufijve:** Asnjëherë mos e lëvizni pointerin përtej fundit të vektorit ose para se të fillojë. I vetmi pozicion i sigurt "jashtë" është *një hap pas fundit*, dhe kjo është vetëm për krahasimin e pointerave - jo për qasje në vlerat.

Dhe kini kujdes; Treguesit duhet të trajtohen me kujdes, pasi është e mundur të dëmtohen të dhënat e ruajtura në adresa të tjera memorie.

C Pointer to Pointer

Ju gjithashtu mund të keni një **tregues që tregon një tregues tjetër**. Kjo quhet **tregues për treguesin** (ose "tregues i dyfishtë").

Mund të tingëllojë konfuze në fillim, por është vetëm një nivel tjetër i indrejtimit: një tregues që ruan *adresën e një treguesi tjetër*.

Mendoni kështu: Një tregues normal është si një shënim me një adresë në të. Një tregues për treguesin është si një shënim tjetër që ju tregon se ku mbahet ajo shënim i parë.

Shënim: Treguesi në tregues nuk është diçka që duhet ta përdorni shpesh si fillestar. Megjithatë, mund ta shihni në programe më të avancuara, Pra, është mirë të kuptosh se çfarë do të thotë dhe si funksionon.

Le të shohim një shembull të thjeshtë për të kuptuar se si funksionon kjo:

Shembull

```
int myNum = 10;           // normal variable
int *ptr = &myNum;        // pointer to int
int **pptr = &ptr;        // pointer to pointer
```

```
printf("myNum = %d\n", myNum);
printf("*ptr = %d\n", *ptr);
printf("**pptr = %d\n", **pptr);
```

Rezultati:

```
myNum = 10
*ptr = 10
**pptr = 10
```

Ja çfarë ndodh hap pas hapi:

- **myNum** mban vlerën **.10**
- **ptr** mban adresën e **.myNum**
- **pptr** mban adresën e **.ptr**
- ***ptr** jep vlerën e **.myNum**
- ****pptr** gjithashtu jep vlerën e , duke kaluar nëpër të dy treguesit.**myNum**

Pra:

- ***ptr** = vlera e **myNum**
- ****pptr** = vlera e të dy niveleve **myNum**

Funksionet C

Një funksion është një bllok kodi i cili funksionon vetëm kur thirret.

Ju mund të kaloni të dhëna, të njohura si parametra, në një funksion.

Funksionet përdoren për të kryer veprime të caktuara, dhe ato janë të rëndësishme për ripërdorimin e kodit: Përcaktoni kodin një herë dhe përdorni atë shumë herë.

Funksionet e paracaktuara

Pra, rezulton se tashmë e dini se çfarë është një funksion. Ju e keni përdorur atë gjatë gjithë kohës ndërsa studioni këtë tutorial!

Për shembull, `main()` është një funksion, i cili përdoret për të ekzekutuar kodin, dhe `printf()` është një funksion; përdoret për të nxjerrë/printuar tekst në ekran:

Shembull

```
int main() {  
    printf("Hello World!");  
    return 0;  
}
```

Krijoni një funksion

Për të krijuar (shpesh referuar si *deklarim*) funksionin tuaj, specifikoni emrin e funksionit, të ndjekur nga kllapa `()` dhe kllapa kaçurrela : `{ }`

Sintaksa

```
void myFunction() {  
    // code to be executed  
}
```

Shembulli i shpjeguar

- `myFunction()` është emri i funksionit
- `void` do të thotë që funksioni nuk ka një vlerë e kthimit. Do të mësoni më shumë rreth vlerave të kthimit më vonë në kapitullin tjetër
- Brenda funksionit (trupit), shtoni kodin që përcakton se çfarë duhet të bëjë funksioni

Thirrni një funksion

Funksionet e deklaruar nuk ekzekutohen menjëherë. Ato "ruhen për ta përdor më vonë", dhe do të ekzekutohen kur të thirren.

Për të thirrur një funksion, shkruani emrin e funksionit të ndjekur nga dy kllapa `()` dhe një pikëpresje `;`

Në shembullin e mëposhtëm, `myFunction()` përdoret për të printuar një tekst (veprimin), kur thirret:

Shembull

Brenda `main`, Thirret: `myFunction()`

```
// Create a function
void myFunction() {
    printf("I just got executed!");
}

int main() {
    myFunction(); // call the function
    return 0;
}

// Outputs "I just got executed!"
```

Një funksion mund të quhet disa herë:

Shembull

```
void myFunction() {
    printf("I just got executed!");
}

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

// I just got executed!
// I just got executed!
// I just got executed!
```

Llogaritni shumën e numrave

Ju mund të vendosni pothuajse gjithçka që dëshironi brenda një funksioni. Qëllimi i funksionit është të ruani kodin dhe ta ekzekutoni atë kur ju nevojitet.

Ashtu si në shembullin e mëposhtëm, ne kemi krijuar një funksion për të llogaritur shumën e dy numrave. Sa herë që jeni gati për të ekzekutuar funksionin (dhe për të kryer llogaritjen), thjesht e thërrisni atë:

Shembull

```
void calculateSum() {  
    int x = 5;  
    int y = 10;  
    int sum = x + y;  
    printf("The sum of x + y is: %d", sum);  
}  
  
int main() {  
    calculateSum(); // call the function  
    return 0;  
}  
  
// Outputs The sum of x + y is: 15
```

Ky ishte vetëm një shembull për të demonstruar një funksion të thjeshtë me deklarata të ndryshme në C. Fuqia e vërtetë e një funksion zbulohet në kapitullin tjetër, kur kaloni "parametra" në të. Kjo lejon funksionin të llogarisë shumën e **çdo** numri, në vend që të kufizohet në vlerat fikse 5 dhe 10.

Parametrat e funksionit C

Informacioni mund t'u kalojë funksioneve si **parametër**. Parametrat veprojnë si **variablat** brenda funksionit.

Parametrat specifikohen pas emrit të funksionit, brenda kllapave. Mund të shtoni sa më shumë parametra që dëshironi, thjesht ndani ato me presje:

Sintaksa

```
returnType functionName(parameter1, parameter2, parameter3) {  
    // code to be executed  
}
```

Në shembullin e mëposhtëm, funksioni merr një **varg karakteresh** me **name** si parameter. Kur thirrret funksioni, ne kalojmë një emër, i cili përdoret brenda funksionit për të printuar "Hello" dhe emrin e secilit person:

Shembull

```
void myFunction(char name[]) {
    printf("Hello %s\n", name);
}

int main() {
    myFunction("Liam");
    myFunction("Jenny");
    myFunction("Anja");
    return 0;
}

// Hello Liam
// Hello Jenny
// Hello Anja
```

Kur një **parametër** i kalohet funksionit, ai quhet **argument**. Pra, nga shembulli i mësipërm: **name** është **parametër**, ndërsa **Liam**, **Jenny** dhe **Anja** janë **argumente**.

Parametra të shumtë

Brenda funksionit, mund të shtoni sa më shumë parametra që dëshironi:

Shembull

```
void myFunction(char name[], int age) {
    printf("Hello %. You are %d years old.\n", name, age);
}

int main() {
    myFunction("Liam", 3);
    myFunction("Jenny", 14);
    myFunction("Anja", 30);
    return 0;
}

// Hello Liam. You are 3 years old.
// Hello Jenny. You are 14 years old.
// Hello Anja. You are 30 years old.
```

Nëse marrim parasysh shembullin "**Llogaritni shumën e numrave**" [Nga faqja e mëparshme](#), ne mund të bëjmë një program më të qëndrueshëm duke përdorur parametrat e funksionit:

Shembull

```
void calculateSum(int x, int y) {
    int sum = x + y;
```

```
printf("The sum of %d + %d is: %d\n", x, y, sum);
}

int main() {
    calculateSum(5, 3);
    calculateSum(8, 2);
    calculateSum(15, 15);
    return 0;
}
```

Shënime mbi parametrat

Vini re se kur jeni duke punuar me parametra të shumtë, **thirrja** e funksionit duhet të ketë të **njëjtin numër argumentesh** si **parametrat** dhe argumentet duhet të kalojnë në **të njëjtin rend**.

Kaloni vargjet si parametra funksioni

Ju gjithashtu mund të kaloni vektorë numerike ose [grupe](#) në një funksion:

Shembull

```
void myFunction(int myNumbers[5]) {
    for (int i = 0; i < 5; i++) {
        printf("%d\n", myNumbers[i]);
    }
}

int main() {
    int myNumbers[5] = {10, 20, 30, 40, 50};
    myFunction(myNumbers);
    return 0;
}
```

Shembulli i shpjeguar

Funksioni (**myFunction**) merr një vektor si parametër (**int myNumbers[5]**), dhe kalon nëpër elementët e vektorit (grup) me ciklin **for**.

Kur funksioni thirret brenda **main()**, ne kalojmë përgjatë vektorit **myNumbers**, i cili nxjerr elementët e vektorit/grupit.

Vini re se kur thërrisni funksionin, duhet vetëm të përdorni emrin e vektorit kur e kaloni atë si argument të **myFunction(myNumbers)**. Megjithatë, **deklarimi i plotë i vektorit** është i nevojshëm në parametrin e funksionit (**int myNumbers[5]**)

Vlerat e kthimit

Fjala kyçe **void**, e përdorur në shembujt e mëparshëm, tregon se funksioni nuk duhet të kthejë një vlerë. Nëse ju dëshironi që funksioni të kthejë një vlerë, mund të përdorni një lloj të dhënash (të tilla si **int** ose **float**, etj.) në vend të **void**, dhe të përdorni fjalën kyçe **return** brenda funksionit:

Shembull

```
int myFunction(int x) {  
    return 5 + x;  
}  
  
int main() {  
    printf("Result is: %d", myFunction(3));  
    return 0;  
}
```

Result is: 8

Ky shembull kthen shumën e një funksioni me **dy parametra**:

Shembull

```
int myFunction(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    printf("Result is: %d", myFunction(5, 3));  
    return 0;  
}
```

Result is: 8

Ju gjithashtu mund ta ruani rezultatin në një variabël:

Shembull

```
int myFunction(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int result = myFunction(5, 3);  
    printf("Result is = %d", result);  
    return 0;  
}
```

Result is: 8

Nëse marrim parasysh edhe një herë shembullin "**Llogaritni shumën e numrave**", mund t'i përdorim **return** dhe t'i ruajmë rezultatet në variabla të ndryshëm. Kjo do ta bëjë programin edhe më fleksibël dhe më i lehtë për t'u kontrolluar:

Shembull

```
int calculateSum(int x, int y) {
    return x + y;
}

int main() {
    int result1 = calculateSum(5, 3);
    int result2 = calculateSum(8, 2);
    int result3 = calculateSum(15, 15);

    printf("Result1 is: %d\n", result1);
    printf("Result2 is: %d\n", result2);
    printf("Result3 is: %d\n", result3);

    return 0;
}
```

```
Result1 is: 8
Result2 is: 10
Result3 is: 30
```

Këshillë: Nëse keni shumë "variabla rezultati", është më mirë t'i ruani rezultatet në një grup:

Shembull

```
int calculateSum(int x, int y) {
    return x + y;
}

int main() {
    // Create an array
    int resultArr[6];

    // Call the function with different arguments and store the results in the array
    resultArr[0] = calculateSum(5, 3);
    resultArr[1] = calculateSum(8, 2);
    resultArr[2] = calculateSum(15, 15);
    resultArr[3] = calculateSum(9, 1);
    resultArr[4] = calculateSum(7, 7);
    resultArr[5] = calculateSum(1, 1);

    for (int i = 0; i < 6; i++) {
        printf("Result%d is = %d\n", i + 1, resultArr[i]);
    }

    return 0;
}
```

```
Result1 is = 8
Result2 is = 10
```

```
Result3 is = 30
Result4 is = 10
Result5 is = 14
Result6 is = 2
```

Shembull i jetës reale

Për të demonstruar një shembull praktik të përdorimit të funksioneve, le të krijojmë një program që konverton një vlerë nga Fahrenheit në Celsius:

Shembull

```
// Function to convert Fahrenheit to Celsius
float toCelsius(float fahrenheit) {
    return (5.0 / 9.0) * (fahrenheit - 32.0);
}

int main() {
    // Set a fahrenheit value
    float f_value = 98.8;

    // Call the function with the fahrenheit value
    float result = toCelsius(f_value);

    // Print the fahrenheit value
    printf("Fahrenheit: %.2f\n", f_value);

    // Print the result
    printf("Convert Fahrenheit to Celsius: %.2f\n", result);

    return 0;
}
```

```
Fahrenheit: 98.80
Convert Fahrenheit to Celsius: 37.11
```

Fushëveprimi

Tani që e kuptoni se si funksionojnë funksionet, është e rëndësishme të mësoni se si veprojnë variablat brenda dhe jashtë funksioneve.

Në C, variablat janë të aksesueshme vetëm **brenda rajonit që janë krijuar**. Kjo quhet **shtrirje**.

Shtrirja lokale

Një variabël i krijuar brenda një funksioni i përket *fushës lokale* të atij funksioni dhe mund të përdoret vetëm brenda atij funksioni:

Shembull

```
#include <stdio.h>

void myFunction() {
    // Local variable that belongs to myFunction
    int x = 5;

    // Print the variable x
    printf("%d", x);
}

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

5

Një **variabël lokal** nuk mund të përdoret jashtë funksionit të cilit i përket.

Nëse përpiqeni ta përdorni jashtë funksionit, ndodh një gabim:

Shembull

```
void myFunction() {
    // Local variable that belongs to myFunction
    int x = 5;
}

int main() {
    myFunction();

    // Print the variable x in the main function
    printf("%d", x);
    return 0;
}
```

Shtrirja globale

Një variabël i krijuar jashtë një funksioni, quhet **variabël global** dhe i përket *shtrirjes globale*.

Variablat globale janë të disponueshme nga çdo fushëveprim, globale dhe lokale:

Shembull

Një variabël i krijuar jashtë një funksioni është global dhe për këtë arsye mund të përdoret nga kushdo:

```
// Global variable x
int x = 5;
```

```
void myFunction() {
    // We can use x here
    printf("%d", x);
}

int main() {
    myFunction();

    // We can also use x here
    printf("%d", x);
    return 0;
}
```

Emërtimi i variablave

Nëse operoni me të njëjtin emër variabli brenda dhe jashtë një funksioni, C do t'i trajtojë ato si dy variabla të veçanta; Një i disponueshëm në fushën globale (jashtë funksionit) dhe një i disponueshëm në fushën lokale (brenda funksionit):

Shembull

Funksioni do të printojë lokal dhe Pastaj kodi do të printojë globalin :xx

```
// Global variable x
int x = 5;

void myFunction() {
    // Local variable with the same name as the global variable (x)
    int x = 22;
    printf("%d\n", x); // Refers to the local variable x
}

int main() {
    myFunction();

    printf("%d\n", x); // Refers to the global variable x
    return 0;
}
```

22
5

Megjithatë, duhet të shmangni përdorimin e të njëjtit emër variabli si për variablat global ashtu edhe për atë lokal pasi mund të çojë në gabime dhe konfuzion.

Në përgjithësi, duhet të jeni të kujdesshëm me variablat globalë, pasi ato mund të jenë aksesuar dhe modifikuar nga çdo funksion:

Shembull

Ndryshoni vlerën x e nga myFunction:

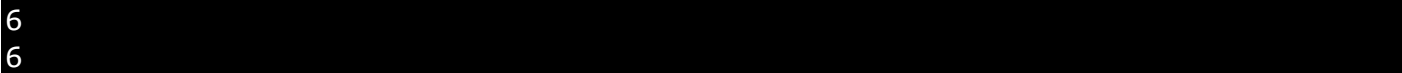
```
#include <stdio.h>

// Global variable
int x = 5;

void myFunction() {
    printf("%d\n", ++x); // Increment the value of x by 1 and print it
}

int main() {
    myFunction();

    printf("%d\n", x); // Print the global variable x
    return 0;
}
```



Përfundim

Për ta përmbledhur, përdorni variabla lokale (me emra të mirë variablash) sa më shumë që të mundeni. Kjo do ta bëjë kodin tuaj më të lehtë për t'u mirëmbajtur dhe më mirë për t'u kuptuar. Megjithatë, mund të gjeni variabla globale kur punoni në programet ekzistuese C ose ndërsa bashkëpunoni me të tjerët. Prandaj, është mirë të kuptoni se si funksionon fushëveprimi dhe si ta përdorni atë në mënyrë efektive për t'u siguruar që kodi juaj të jetë i qartë dhe funksional.

Deklarata dhe përkufizimi i funksionit C

Ju keni mësuar tashmë nga kapitujt e mëparshëm se mund të krijoni dhe thërrisni një funksion në mënyrën e mëposhtme:

Shembull

```
// Create a function
void myFunction() {
    printf("I just got executed!");
}

int main() {
    myFunction(); // call the function
    return 0;
}
```

Një funksion përbëhet nga dy pjesë:

- **Deklarata:** emri i funksionit, lloji i kthimit dhe parametrat (nëse ka)
- **Përkufizimi:** trupi i funksionit (kodi që do të ekzekutohet)

```
void myFunction() { // declaration
    // the body of the function (definition)
```

```
}
```

Për optimizimin e kodit, rekomandohet të ndani deklaratën dhe përkufizimin e funksionit.

Shpesh do të shihni programe C që kanë deklaratë funksioni më sipër , dhe përkufizimi i funksionit më poshtë `.main()main()`

Kjo do ta bëjë kodin Më mirë i organizuar dhe më i lehtë për t'u lexuar:

Shembull

```
// Function declaration
void myFunction();

// The main method
int main() {
    myFunction(); // call the function
    return 0;
}

// Function definition
void myFunction() {
    printf("I just got executed!");
}
```

Po në lidhje me parametrat

Nëse përdorim shembullin nga [kapitulli i parametrave të funksionit](#) në lidhje me parametrat dhe vlerat e kthimit:

Shembull

```
int myFunction(int x, int y) {
    return x + y;
}

int main() {
    int result = myFunction(5, 3);
    printf("Result is = %d", result);
    return 0;
}
// Outputs 8 (5 + 3)
```

Konsiderohet praktikë e mirë ta shkruani kështu:

Shembull

```
// Function declaration
int myFunction(int x, int y);
```

```
// The main method
int main() {
    int result = myFunction(5, 3); // call the function
    printf("Result is = %d", result);
    return 0;
}

// Function definition
int myFunction(int x, int y) {
    return x + y;
}
```

Funksionet që thërrasin funksione të tjera

Për sa kohë që deklaroni funksionet së pari, është gjithashtu e mundur të përdorni funksione për të thirrur funksione të tjera:

Shembull

Përdorni një funksion për të thirrur një funksion tjetër:

```
// Declare two functions, myFunction and myOtherFunction
void myFunction();
void myOtherFunction();

int main() {
    myFunction(); // call myFunction (from main)
    return 0;
}

// Define myFunction
void myFunction() {
    printf("Some text in myFunction\n");
    myOtherFunction(); // call myOtherFunction (from myFunction)
}

// Define myOtherFunction
void myOtherFunction() {
    printf("Hey! Some text in myOtherFunction\n");
}
```

```
Pak tekst në myFunction
Hej! Disa tekste në myOtherFunction
```

Funksioni inline

Ndonjëherë mund të shihni fjalën kyçe të përdorur në funksionet e njerëzve të tjerë. Nuk është diçka që duhet ta përdorni shpesh si fillestar, por është mirë të dini se çfarë do të thotë **inline**

Një **funksion inline** është një funksion i vogël që i kërkon përpiluesit të fusë kodin e tij direkt aty ku thirret, në vend që të kërcejë në të.

Kjo mund t'i bëjë funksionet e shkurtra, të përdorura shpesh pak më të shpejta, sepse heq vonesën e vogël të një thirrjeje funksionale normale.

Le të krahasojmë një funksion të rregullt me një funksion në linjë:

Funksioni i rregullt

```
int square(int x) {  
    return x * x;  
}
```

Funksioni inline

```
inline int square(int x) {  
    return x * x;  
}
```

Të dy funksionet funksionojnë në të njëjtën mënyrë. Dallimi i vetëm është se versioni i *sugjeron* përpiluesit të kopjojë kodin e funksionit direkt aty ku përdoret. **inline**

Shembull i funksionit në linjë

Këtu është një shembull i thjeshtë që përdor një funksion: **inline**

Shembull

```
#include <stdio.h>  
  
inline int add(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    printf("%d", add(5, 3));  
    return 0;  
}
```

Këtu, përpiluesi mund të zëvendësojë **add(5, 3)** me kodin **5 + 3** drejtpërdrejt - duke kursyer kohën që do të duhej për të thirrur një funksion të veçantë.

Kur të mos përdorni Inline:

Funksionet në linjë janë më të mirat për funksione **të vogla dhe të thjeshta**. Shmangni përdorimin e tyre për:

- Funksione të mëdha (ato e bëjnë programin tuaj më të madh)
- Funksionet rekursive
- Funksione që rrallë thirren

Shumë funksione **inline** /në linjë mund ta bëjnë programin tuaj më të ngadalshëm dhe më të madh, një problem i njohur si *fryrje e kodit*.

Funksioni i brendshëm vs i rregullt

Regular Function	Inline Function
Code jumps to the function each time it's called	Code is inserted directly where it's called
Slightly slower (small delay)	Slightly faster
Good for large functions	Good for small functions

Dallimet kryesore:

⚡ Funksionet inline (në C/C++)

- Deklarohen me fjalën kyçe `inline`.
- Kompajleri përpiqet t'i zëvendësojë thirrjet e funksionit me kodin e tij direkt.
- Rrit shpejtësinë e ekzekutimit duke shmangur thirrjet e funksioneve, por mund të rrisë madhësinë e kodit (code size).
- Përdoren zakonisht për funksione shumë të vogla dhe të thjeshta.

Rekursioni

Rekursioni është teknika e bërjes së një thirrjeje funksioni. Kjo teknikë ofron një mënyrë për t'i ndarë problemet e ndërlikuara në probleme të thjeshta që janë më të lehta për t'u zgjidhur.

Rekursioni mund të jetë pak i vështirë për t'u kuptuar. Në mënyra më e mirë për të kuptuar se si funksionon është të eksperimentoni me të.

Shembull i rekursionit

Shtimi i dy numrave së bashku është i lehtë për t'u bërë, por shtimi i një diapazoni numrash është më shumë e komplikuar. Në shembullin e mëposhtëm, rekursioni përdoret për të shtuar një gamë numrash së bashku duke e zbërthyer atë në detyrën e thjeshtë të shtimit të dy numrave:

Shembull

Përdorni rekursionin për të shtuar të gjithë numrat nga 1 në 10:

```
int sum(int k);

int main() {
    int result = sum(10);
    printf("%d", result);
    return 0;
}

int sum(int k) {
    if (k > 0) {
        return k + sum(k - 1);
    } else {
        return 0;
    }
}
```

Shembulli i shpjeguar

Kur thirret funksioni `sum()`, ai shton parametrin `k` në shumën e të gjithë numrave më të vegjël sesa `k` dhe kthen rezultatin. Kur `k` bëhet 0, funksioni thjesht kthen 0. Kur Duke ekzekutuar, programi ndjek këto hapa:

```
10 + shumë(9)
10 + ( 9 + shumë(8) )
10 + ( 9 + ( 8 + shumë(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + shumë(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Megjithatë, funksioni nuk e thërret veten kur `k` është 0, programi ndalon atje dhe kthen rezultatin.

Zhvilluesi duhet të jetë shumë i kujdesshëm me rekursionin pasi mund të jetë mjaft e lehtë të rrëshqasësh në shkrimin e një funksioni që **nuk përfundon kurrë**, ose një që përdor **sasi të tepërta të memories ose fuqisë së procesorit**. Megjithatë, kur shkruhet saktë, rekursioni mund të jetë një qasje shumë efikase dhe matematikisht elegante ndaj programimit.

Shembull

Përdorni rekursionin për të numëruar mbrapsht nga 5:

```
void countdown(int n);

int main() {
    countdown(5);
    return 0;
}

void countdown(int n) {
    if (n > 0) {
        printf("%d ", n);
        countdown(n - 1);
    }
}
```

Funksioni thërret veten me `n - 1` derisa `n` bëhet `0`.

Llogaritni faktorialin me rekursion

Ky shembull përdor një funksion rekursiv për të llogaritur faktorialin e 5:

Shembull

```
int factorial(int n);

int main() {
    printf("Factorial of 5 is %d", factorial(5));
    return 0;
}

int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

Faktorial do të thotë shumëzimi i një numri me çdo numër poshtë tij, deri në 1. Për shembull, faktoriali i 5 është: $5 * 4 * 3 * 2 * 1 = 120$. Sipas përkufizimit, $0!$ është gjithashtu 1.

Pointeri /shënjeshti i funksionit

Një **pointer funksioni** është si një [shënjeshti](#) normal, por në vend që të tregojë një variabël, ai tregon një **funksion**.

Kjo do të thotë se ruan *adresën e një funksioni*, duke ju lejuar ta thërrisni atë funksion duke përdorur pointerin.

Pointerat e funksionit ju lejojnë të vendosni se cilin funksion të ekzekutoni ndërsa programi është në funksion, ose kur dëshironi të kaloni një funksion si argument në një funksion tjetër.

Mendoni si të ruani një numër telefoni – treguesi/pointer e di se *ku* jeton funksioni në memorie, kështu që mund ta "telefononi" më vonë.

Deklarimi i një pointer funksioni

Sintaksa e përgjithshme për deklarimin e një treguesi funksioni është:

```
returnType (*pointerName)(parameterType1, parameterType2, ...);
```

Për shembull:

```
int (*ptr)(int, int);
```

Kjo do të thotë se është një tregues **ptr** për një funksion që merr dy vlera **int** dhe kthen një **int**.

Caktimi i një funksioni një treguesi

Ju mund t'i caktoni një funksion një treguesi në dy mënyra:

```
ptr = add;  
ptr = &add;
```

Të dyja janë të njëjta, sepse emri i funksionit tashmë përfaqëson adresën e tij në memorie.

Thirrja e një funksioni përmes një pointer

Pasi të caktohet treguesi, mund ta thërrisni funksionin në dy mënyra:

```
ptr(5, 3);  
(*ptr)(5, 3);
```

Të dyja janë të vlefshme dhe bëjnë të njëjtën gjë.

Shembull i pointer të funksionit

Tani që dini si të deklaroni dhe caktoni një tregues funksioni, le të shohim një shembull të plotë:

Shembull

Një tregues për një funksion që shton dy numra:

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int (*ptr)(int, int) = add;
    int result = ptr(5, 3);
    printf("Result: %d", result);
    return 0;
}
```

Këtu:

- `ptr` është një tregues për funksionin `.add()`
- `ptr(5, 3)` thërret funksionin duke përdorur treguesin.

Kjo funksionon saktësisht njësoj si thirrja `.add(5, 3)`

Shënim: Një emër funksioni në vetvete tregon fillimin e kodit të tij në memorie. Kjo do të thotë se një emër funksioni tashmë vepron si një tregues! Deklarimi i një treguesi funksioni ju jep vetëm një variabël që mund ta mbajë atë adresë - kështu që mund ta ndryshoni ose ta kaloni.

Kalimi i një funksioni si argument

Treguesit e funksionit mund të kalojnë në funksione të tjera - kjo quhet **callback**.

Kjo lejon një funksion të thërrasë një funksion tjetër që ju jepni si hyrje.

Shembull

Kalimi i një treguesi funksioni në një funksion tjetër:

```
void greetMorning() { printf("Good morning!\n"); }
void greetEvening() { printf("Good evening!\n"); }

void greet(void (*func)()) {
    func();
}

int main() {
    greet(greetMorning);
    greet(greetEvening);
    return 0;
}
```

Këtu, `greet()` merr një funksion tjetër si parametër dhe e thërret atë. Kjo përdoret zakonisht në programet ose bibliotekat e drejtuara nga ngjarjet që kanë nevojë për funksione të kthimit të ofruara nga përdoruesi.

Grupi /vargu i treguesit pointer të funksionit

Ju gjithashtu mund të ruani disa tregues funksionesh në një grup, kështu që mund të zgjidhni se cilin funksion të ekzekutoni ndërsa programi është në funksion.

Ky shembull ekzekuton tre funksione të ndryshme duke përdorur një sërë treguesish funksionesh:

Shembull

```
void add() { printf("Add\n"); }
void subtract() { printf("Subtract\n"); }
void multiply() { printf("Multiply\n"); }

int main() {
    void (*operations[3])() = { add, subtract, multiply };
    for (int i = 0; i < 3; i++) {
        operations[i]();
    }
    return 0;
}
```

Kjo përdoret shpesh për menu të thjeshta, lista komandash ose kalkulatorë - kudo që dëshironi të thërrisni funksione të ndryshme bazuar në inputin e përdoruesit.

Shembull

Një kalkulator i thjeshtë që përdor një varg treguesish funksionesh:

```
void add(int a, int b) { printf("Result: %d\n", a + b); }
void subtract(int a, int b) { printf("Result: %d\n", a - b); }
void multiply(int a, int b) { printf("Result: %d\n", a * b); }

int main() {
    int choice, x = 10, y = 5;

    void (*operations[3])(int, int) = { add, subtract, multiply };

    printf("x = %d, y = %d\n\n", x, y);
    printf("Choose an operation:\n");
    printf("0: Add\n1: Subtract\n2: Multiply\n");
    scanf("%d", &choice);

    if (choice >= 0 && choice < 3) {
        operations[choice](x, y);
    } else {
        printf("Invalid choice!\n");
    }
}
```

```
return 0;  
}
```

Në këtë program:

- Të tre funksionet matematikore (`add()`, `subtract()`, dhe `multiply()`) ruhen në një varg.
- Përdoruesi zgjedh cilin të ekzekutojë duke futur një numër.
- Funksioni i saktë thirret më pas duke përdorur vargun e treguesve.

Ky është një shembull i thjeshtë se si vargjet e treguesve të funksioneve i bëjnë programet më fleksibël dhe të ripërdorshme.

Treguesi i funksionit vs funksioni normal

Normal Function	Function Pointer
Called directly by its name	Called using a pointer
The function is decided before the program runs	You can choose which function to call while the program is running
Good for simple code	Good for flexible and reusable code

Përmbledhje

- Një tregues funksioni ruan adresën e një funksioni.
- Ju mund të deklaroni, caktoni dhe thërrisni funksione përmes tij.
- Lejon kalimin e funksioneve si argumente në funksione të tjera.
- E dobishme për thirrje, menutë dhe dizajnin fleksibël të programit.

Funksioni i thirrjes mbrapsht

Një **funksion callback** është një funksion që *kalon si argument* te një funksion tjetër.

Funksioni marrës mund ta **thërrasë** përsëri (ta ekzekutojë) sa herë që ka nevojë.

Kjo është një mënyrë e fuqishme për ta bërë kodin tuaj **fleksibël dhe të ripërdorshëm** - mund të vendosni se cili funksion duhet të ekzekutohet, pa ndryshuar logjikën kryesore.

Në C, funksionet callback zakonisht implementohen duke përdorur [tregues funksionesh](#).

Shembull i thjeshtë i thirrjes mbrapsht

Ja një shembull bazë që tregon se si një funksion mund të "thërrasë mbrapsht" një tjetër:

Shembull

Kalimi i një funksioni si argument te një funksion tjetër:

```
void sayHello() {
    printf("Hello from the callback!\n");
}

void runCallback(void (*callback)()) {
    printf("Before calling the callback...\n");
    callback();
    printf("After calling the callback.\n");
}

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

Në këtë shembull:

- **sayHello()** është një funksion normal që printon një mesazh.
- **runCallback()** merr një tregues funksioni si parametër.
- Kur **runCallback(sayHello)** ekzekutohet, thërret **sayHello()** brenda vetes.

Shënim: Fjala "callback" vjen nga fakti që një funksion "thërret mbrapsht" një tjetër që i është dhënë më parë.

Callback me Parametra

Gjithashtu mund të kalosh funksione që marrin parametra - vetëm sigurohu që Përputhjet e tipit të treguesve të funksioneve:

Shembull

Kalimi i një funksioni me parametra si thirrje prapa:

```
void addNumbers(int a, int b) {
    printf("The sum is: %d\n", a + b);
}

void calculate(void (*callback)(int, int), int x, int y) {
    callback(x, y);
}

int main() {
    calculate(addNumbers, 5, 3);
    return 0;
}
```

Këtu, `calculate()` merr një funksion dhe dy numra, pastaj thërret funksionin (në këtë rast, `addNumbers()`) me ato numra si argumente.

Thirrje të shumta për rikthim

Mund të përdorësh callback-e për të lejuar që një funksion të sillet ndryshe në varësi të funksionit që kalon. Kjo është e zakonshme në renditje, filtrim dhe menaxhim të kodeve të ngjarjeve.

Shembull

Përdorimi i funksioneve të ndryshme callback për të kryer veprime të ndryshme:

```
void greetMorning() { printf("Good morning!\n"); }
void greetEvening() { printf("Good evening!\n"); }

void greet(void (*callback)()) {
    callback();
}

int main() {
    greet(greetMorning);
    greet(greetEvening);
    return 0;
}
```

Në varësi të funksionit që kalon, programi prodhon dalje të ndryshme - pa ndryshuar `greet()` vetveten.

Shembull nga Jeta Reale: Përdorimi i Callback-eve në `qsort()`

Shumë funksione të bibliotekës standarde C përdorin callback.

Për shembull, funksioni `qsort()` në `<stdlib.h>` përdor një callback për të krahasuar elementet gjatë renditjes.

Ju ofroni funksionin e krahasimit dhe `qsort()` e thërret sipas nevojës. Kjo do të **rendisë** elementet:

Shembull

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

int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

int main() {
    int numbers[] = { 5, 2, 9, 1, 7 };
    int size = sizeof(numbers) / sizeof(numbers[0]);

    qsort(numbers, size, sizeof(int), compare);

    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }
    return 0;
}
```

Këtu `compare()` është funksioni i rikthimit që përdoret nga `qsort()` për të vendosur se si të renditen numrat.

Përmbledhje

- Një funksion callback është një funksion që kalon si argument te një funksion tjetër.
- Lejon që një funksion të thërrasë një tjetër pa e ditur emrin e tij paraprakisht.
- Callback-et e bëjnë kodin më fleksibël dhe më të ripërdorshëm.
- Ato përdoren në Bibliotekën Standarde C (si `qsort()`).

C Files- Menaxhimi i Skedarëve dhe Si Të Krijoni Skedarë

Trajtimi i Skedarëve

Në C, mund të krijosh, hapësh, lexosh dhe shkruash në skedarë duke deklaruar një tregues të tipit `FILE`, dhe të përdorësh funksionin: `fopen()`

```
FILE *fptr;
fptr = fopen(filename, mode);
```

`FILE` është në thelb një tip të dhënash, dhe ne kemi nevojë për të krijuar një variabël tregues për të punuar me të (`fptr`). Për tani, kjo linja nuk është e rëndësishme. Është thjesht diçka që të duhet kur punon me skedarë.

Për të hapur realisht një skedar, përdorni funksionin `fopen()`, i cili merr dy parametra:

Parameter	Description
<i>filename</i>	The name of the actual file you want to open (or create), like filename.txt
<i>mode</i>	A single character, which represents what you want to do with the file (read, write or append): w - Writes to a file a - Appends new data to a file r - Reads from a file

Krijohet një Skedar

Për të krijuar një skedar, mund të përdorni modalitetin **w** brenda funksionit **fopen()**.

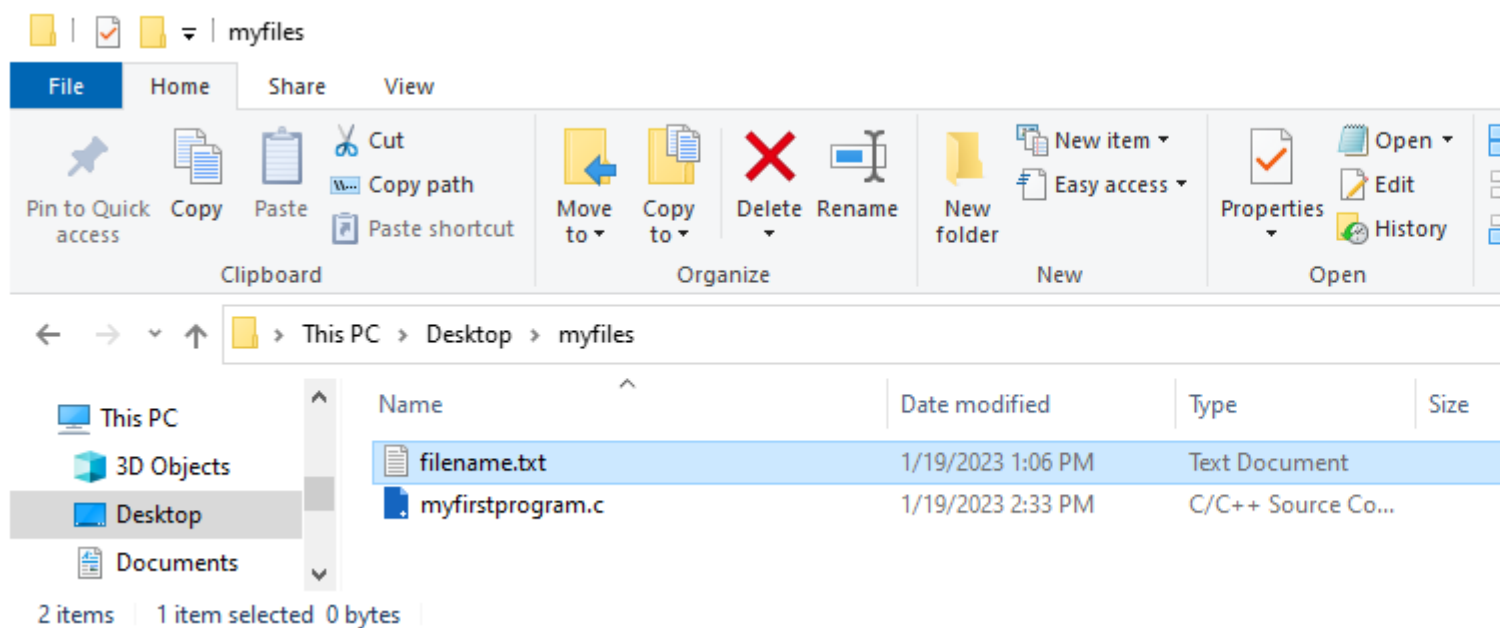
Ky modalitet përdoret për të shkruar në një skedar. **Megjithatë**, nëse dosja e bën nuk ekziston, do të krijojë një për ju: **w**

Shembull

```
FILE *fptr;  
  
// Create a file  
fptr = fopen("filename.txt", "w");  
  
// Close the file  
fclose(fptr);
```

Shënim: Dosja është krijuar në të njëjtin direktorium me skedarët e tjerë të C-së, nëse asgjë tjetër nuk është e specifikuar.

Në kompjuterin tonë, duket kështu:



Këshillë: Nëse dëshiron të krijosh skedarin në një dosje specifike, thjesht jep një rrugë absolute (Mos harro të përdorësh double backslash për të krijuar një backslash të vetëm (\), siç specifikuam në [Strings Special Personazhet](#)):

```
fptr = fopen("C:\\directoryname\\filename.txt", "w");
```

Mbyllja e skedarit

A e vutë re funksionin `fclose()` në shembullin tonë më sipër?

Kjo do ta mbyllë dosjen kur të përfundojmë me të.

Konsiderohet si praktikë e mirë, sepse siguron që:

- Ndryshimet ruhen siç duhet
- Programe të tjera mund të përdorin skedarin (nëse dëshiron)
- Pastroni hapësirën e panevojshme të memories

Në kapitujt e ardhshëm, do të mësoni si të shkruani përmbajtje në një skedar dhe të lexoni prej tij.

Shkruaj në një skedar

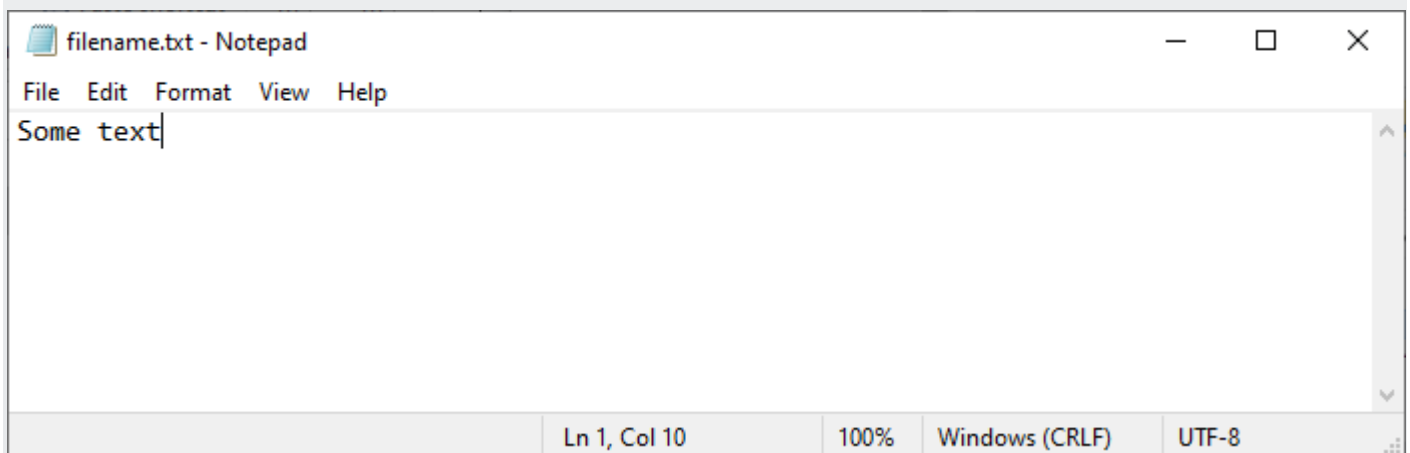
Le të përdorim mënyrën `w` nga kapitulli mëparshëm, e përsëri, shkruaj diçka në skedarin që sapo krijuam.

Modaliteti `w` do të thotë që skedari hapet për **shkrim**. Për të futur përmbajtje në të, mund të përdorni funksionin `fprintf()` dhe të shtoni variablin tregues (`fptr` në shembullin tonë) dhe pak tekst:

Shembull

```
FILE *fptr;  
  
// Open a file in writing mode  
fptr = fopen("filename.txt", "w");  
  
// Write some text to the file  
fprintf(fptr, "Some text");  
  
// Close the file  
fclose(fptr);
```

Si rezultat, kur hapim skedarin në kompjuterin tonë, duket kështu:



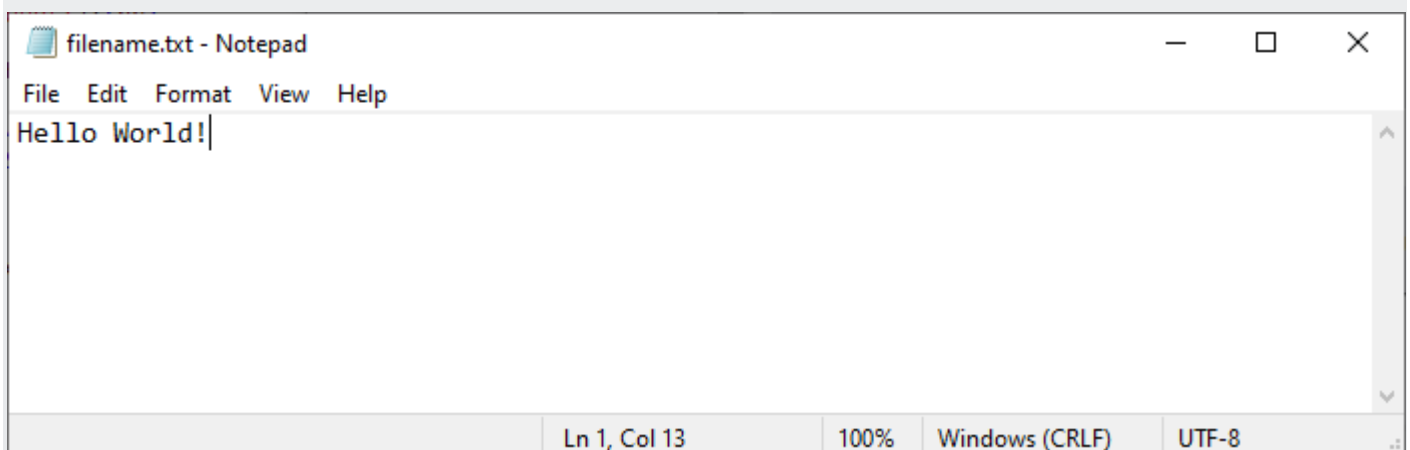
Shënim: Nëse shkruan në një skedar që ekziston tashmë, përmbajtja e vjetër fshihet dhe futet përmbajtja e re. Kjo është e rëndësishme për t'u ditur, pasi mund ta fshish aksidentalisht Përmbajtje ekzistuese.

Për shembull:

Shembull

```
fprintf(fptr, "Hello World!");
```

Si rezultat, kur hapim skedarin në kompjuterin tonë, thotë "Hello World!" në vend të "Disa tekste":



Shtoni përmbajtje në një skedar

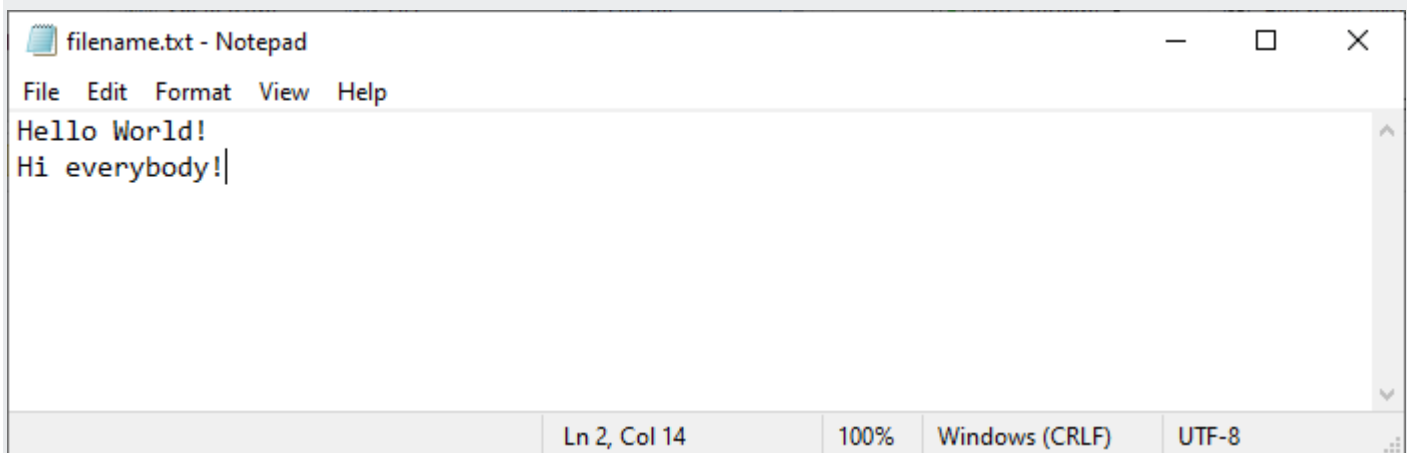
Nëse dëshiron të shtosh përmbajtje në një skedar pa fshirë përmbajtjen e vjetër, mund të përdorësh këtë mënyrë. **a**

Modaliteti **a** shton përmbajtje në fund të skedarit:

Shembull

```
FILE *fptr;  
  
// Open a file in append mode  
fptr = fopen("filename.txt", "a");  
  
// Append some text to the file  
fprintf(fptr, "\nHi everybody!");  
  
// Close the file  
fclose(fptr);
```

Si rezultat, kur hapim skedarin në kompjuterin tonë, duket kështu:



Shënim: Ashtu si me modalitetin **w**; Nëse skedari nuk ekziston, modaliteti **a** do të krijojë një skedar të ri me përmbajtjen "të shtuar".

Lexo një Skedar

Në kapitullin e mëparshëm, shkruam në një skedar duke përdorur modalitetet **w** dhe **a** brenda funksionit **fopen()**.

Për të **lexuar** nga një skedar, mund të përdorni modalitetin **r**:

Shembull

```
FILE *fptr;
```

```
// Open a file in read mode  
fptr = fopen("filename.txt", "r");
```

Kjo do ta bëjë **filename.txt** të hapur për lexim.

Kërkon pak punë për të lexuar një skedar në C. Qëndro i fortë! Ne do t'ju udhëzojmë hap pas hapi.

Më pas, duhet të krijojmë një varg që duhet të jetë mjaftueshëm i madh për të ruajtur përmbajtjen e skedarit.

Për shembull, le të krijojmë një varg që mund të ruajë deri në 100 karaktere:

Shembull

```
FILE *fptr;  
  
// Open a file in read mode  
fptr = fopen("filename.txt", "r");  
  
// Store the content of the file  
char myString[100];
```

Për të lexuar përmbajtjen e **filename.txt**, mund ta përdorim funksionin (**fgets**).

Funksioni **fgets()** merr tre parametra:

Shembull

```
fgets(myString, 100, fptr);
```

1. Parametri i parë specifikon ku të ruajmë përmbajtjen e skedarit, e cila do të jetë në vargun **myString** që sapo krijoi.
2. Parametri i dytë përcakton madhësinë maksimale të të dhënave për t'u lexuar, e cila duhet të përputhet me madhësinë e **myString** (100).
3. Parametri i tretë kërkon një tregues skedari që përdoret për të lexuar skedarin (**fptr** në shembullin tonë).

Tani, mund të printojmë vargun, i cili do të nxjerrë përmbajtjen e skedarit:

Shembull

```
FILE *fptr;  
  
// Open a file in read mode  
fptr = fopen("filename.txt", "r");  
  
// Store the content of the file  
char myString[100];
```

```
// Read the content and store it inside myString
fgets(myString, 100, fptr);

// Print the file content
printf("%s", myString);

// Close the file
fclose(fptr);

Hello World!
```

Shënim: Funkzioni `fgets` lexon vetëm rreshtin e parë të skedarit. Nëse e mbani mend, kishte dy rreshta teksti në `filename.txt`.

Për të lexuar çdo rresht të skedarit, mund të përdorni një cikël: [while](#)

Shembull

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];

// Read the content and print it
while(fgets(myString, 100, fptr)) {
    printf("%s", myString);
}

// Close the file
fclose(fptr);

Hello World!
Hi everybody!
```

Praktika e Mirë

Nëse përpiqeni të hapni një skedar për lexim që nuk ekziston, funksioni `fopen()` do të kthejë [NULL](#).

Këshillë: Si praktikë e mirë, mund të përdorim një deklaratë `if` për të testuar për `NULL`, dhe të printojë pak tekst në vend të tij (kur skedari nuk ekziston):

Shembull

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("loremipsum.txt", "r");

// Print some text if the file does not exist
if(fptr == NULL) {
    printf("Not able to open the file.");
}

// Close the file
fclose(fptr);
```

Nëse skedari nuk ekziston, printohet teksti i mëposhtëm:

```
Not able to open the file.
```

Duke pasur këtë parasysh, mund të krijojmë një kod më të qëndrueshëm nëse përdorim "lexim skedari " Shembulli më sipër përsëri:

Shembull

Nëse skedari ekziston, lexo përmbajtjen dhe printoje. Nëse skedari nuk ekziston, printoni një mesazh:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];

// If the file exist
if(fptr != NULL) {

    // Read the content and print it
    while(fgets(myString, 100, fptr)) {
        printf("%s", myString);
    }

// If the file does not exist
} else {
    printf("Not able to open the file.");
}

// Close the file
fclose(fptr);
```

```
Hello World!
Hi everybody!
```

Shënim: Do të mësoni më shumë në [kapitullin tonë NULL](#).**NULL**

Strukturat

Strukturat (të quajtura struct) janë një mënyrë për të grupuar disa variabla të lidhur në një vend.

Çdo variabël në strukturë është i njohur si **anëtar** i strukturës.

Ndryshe nga një [varg](#), një strukturë mund të përmbajë shumë lloje të ndryshme të të dhënave (**int**, **float**, **char**, etj.).

Krijo një Strukturë

Mund të krijoni një strukturë duke përdorur fjalën kyçe **struct** dhe të deklaroni secilin prej anëtarëve të saj me kllapa të dredhura:

```
struct MyStructure {    // Structure declaration
    int myNum;           // Member (int variable)
    char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

Për të aksesuar strukturën, duhet të krijosh një variabël të saj.

Përdor fjalën kyçe **struct** brenda metodës **main()**, e ndjekur nga emri të strukturës dhe pastaj emri i variablit të strukturës:

Krijoni një variabël strukturore me emrin "s1":

```
struct myStructure {
    int myNum;
    char myLetter;
};

int main() {
    struct myStructure s1;
    return 0;
}
```

Anëtarët e Strukturës së Aksesit

Për të aksesuar anëtarët e një strukture, përdorni sintaksën me pikë (.):

Shembull

```
// Create a structure called myStructure
struct myStructure {
    int myNum;
    char myLetter;
};

int main() {
    // Create a structure variable of myStructure called s1
    struct myStructure s1;

    // Assign values to members of s1
    s1.myNum = 13;
    s1.myLetter = 'B';

    // Print values
    printf("My number: %d\n", s1.myNum);
    printf("My letter: %c\n", s1.myLetter);

    return 0;
}
```

Tani mund të krijoni lehtësisht disa variabla strukture me vlera të ndryshme, duke përdorur vetëm një strukturë:

Shembull

```
// Create different struct variables
struct myStructure s1;
struct myStructure s2;

// Assign values to different struct variables
s1.myNum = 13;
s1.myLetter = 'B';

s2.myNum = 20;
s2.myLetter = 'C';
```

Po për vargjet ose fjet në struktura?

Mbani mend që vargjet në C janë në fakt një varg karakteresh, dhe Fatkeqësisht, nuk mund t'i caktoni një vlerë një vargu të tillë:

Shembull

```
struct myStructure {
    int myNum;
    char myLetter;
    char myString[30]; // String
};

int main() {
    struct myStructure s1;
```

```
// Trying to assign a value to the string
s1.myString = "Some text";

// Trying to print the value
printf("My string: %s", s1.myString);

return 0;
}
```

Do të ndodhë një gabim:

```
prog.c:12:15: error: assignment to expression with array type
```

Megjithatë, ka një zgjidhje për këtë! Mund të përdorni funksionin `strcpy()` dhe t'i caktoni vlerën `s1.myString`, si kjo:

Shembull

```
struct myStructure {
    int myNum;
    char myLetter;
    char myString[30]; // String
};

int main() {
    struct myStructure s1;

    // Assign a value to the string using the strcpy function
    strcpy(s1.myString, "Some text");

    // Print the value
    printf("My string: %s", s1.myString);

    return 0;
}
```

Rezultati:

```
My string: Some text
```

Sintaksa më e thjeshtë

Gjithashtu mund t'u caktoni vlera anëtarëve të një variabli strukture në kohën e deklarimit, në një rresht të vetëm.

Thjesht fut vlerat në një listë të ndarë me presje brenda kllapave `{}`. Vini re që nuk përdoret funksioni `strcpy()` për varg me këtë teknike:

Shembull

```
// Create a structure
struct myStructure {
    int myNum;
    char myLetter;
```

```
char myString[30];
};

int main() {
    // Create a structure variable and assign values to it
    struct myStructure s1 = {13, 'B', "Some text"};

    // Print values
    printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

    return 0;
}
```

Shënim: Renditja e vlerave të futura duhet të përputhet me rendin e llojet e variablave të deklaruara në strukturë (13 për int, 'B' për char, etj).

Strukturat e Kopjimit

Gjithashtu mund t'i caktoni një strukturë një tjetre.

Në shembullin vijues, vlerat e s1 kopjohen në s2:

Shembull

```
struct myStructure s1 = {13, 'B', "Some text"};
struct myStructure s2;

s2 = s1;
```

Modifiko Vlerat

Nëse dëshiron të ndryshosh/modifikosh një vlerë, mund të përdorësh sintaksën dot (.).

Dhe për të modifikuar një vlera e vargut, funksioni është përsëri i dobishëm: **strcpy()**

Shembull

```
struct myStructure {
    int myNum;
    char myLetter;
    char myString[30];
};

int main() {
    // Create a structure variable and assign values to it
    struct myStructure s1 = {13, 'B', "Some text"};

    // Modify values
    s1.myNum = 30;
```

```
s1.myLetter = 'C';
strcpy(s1.myString, "Something else");

// Print values
printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

return 0;
}
```

Modifikimi i vlerave është veçanërisht i dobishëm kur kopjoni vlerat e strukturës:

Shembull

```
// Create a structure variable and assign values to it
struct myStructure s1 = {13, 'B', "Some text"};

// Create another structure variable
struct myStructure s2;

// Copy s1 values to s2
s2 = s1;

// Change s2 values
s2.myNum = 30;
s2.myLetter = 'C';
strcpy(s2.myString, "Something else");

// Print values
printf("%d %c %s\n", s1.myNum, s1.myLetter, s1.myString);
printf("%d %c %s\n", s2.myNum, s2.myLetter, s2.myString);
```

Ok, pra, si janë të dobishme strukturat?

Imagjinoni që duhet të shkruani një program për të ruajtur informacione të ndryshme rreth makinave, si marka, modeli, dhe vitin. Ajo që është e shkëlqyer për strukturat është se mund të krijosh një "Model makine" dhe përdore për çdo makinë që prodhon. Shih më poshtë për një shembull nga jeta reale.

Shembull nga jeta reale

Përdorni një strukturë për të ruajtur informacion të ndryshëm rreth makinave:

Shembull

```
struct Car {
    char brand[30];
    char model[30];
    int year;
};

int main() {
    struct Car car1 = {"BMW", "X5", 1999};
```

```
struct Car car2 = {"Ford", "Mustang", 1969};
struct Car car3 = {"Toyota", "Corolla", 2011};

printf("%s %s %d\n", car1.brand, car1.model, car1.year);
printf("%s %s %d\n", car2.brand, car2.model, car2.year);
printf("%s %s %d\n", car3.brand, car3.model, car3.year);

return 0;
}
```

Sfida: Ruaj moshën e një personi

Testoni kuptimin tuaj të **strukturave C** duke përfunduar një sfidë të vogël kodimi.

Udhëzime

Përfundoni hapat më poshtë (ndiqni rendin):

1. Mbi `main()`, përcaktoni një strukturë të quajtur `Person`
2. Struktura duhet të përmbajë një anëtar `int` të quajtur `age`
3. Brenda `main()`, deklaro një variabël të tipit `struct Person` të quajtur `p`
4. Vendosur `p.age` në `25`
5. Printoni `p.age` duke përdorur specifikuesin e formatit të saktë

```
#include <stdio.h>

// Define the structure here
struct Person{
    int age;
};

int main() {
    // Declare the struct variable here
    struct Person p;
    // Set the age here
    p.age=25;
    // Print the age here
    printf("%d",p.age);
    return 0;
}
```

🎉 Great job! You nailed it!

Checklist



Include stdio.h	✓
Start struct Person	✓
Add int age;	✓
End struct with };	✓
Define struct Person above main()	✓
Have int main()	✓
Declare struct Person p	✓
Set p.age to 25	✓
Print p.age using %d or %i	✓

```
End with return 0;      ✓  
End with a closing }
```

Struktura të Ndërlidhura

Një strukturë mund të përmbajë edhe një strukturë tjetër si anëtar. Kjo quhet **strukturë e folezuar**, dhe është e dobishme kur dëshironi të gruponi të dhënat e lidhura në shtresa:

Shembull

```
struct Owner {  
    char firstName[30];  
    char lastName[30];  
};  
  
struct Car {  
    char brand[30];  
    int year;  
    struct Owner owner; // Nested structure  
};  
  
int main() {  
    struct Owner person = {"John", "Doe"};  
    struct Car car1 = {"Toyota", 2010, person};  
  
    printf("Car: %s (%d)\n", car1.brand, car1.year);  
    printf("Owner: %s %s\n", car1.owner.firstName, car1.owner.lastName);  
  
    return 0;  
}
```

```
Car: Toyota (2010)  
Owner: John Doe
```

Këtu, struktura **Car** përmban një strukturë tjetër (**Owner**) brenda saj. Kjo e bën të lehtë organizimin e të dhënave komplekse, për shembull, një makinë dhe pronari i saj.

Strukturat dhe treguesit

Mund të përdorësh pointer-a me struktura për ta bërë kodin tënd më efikas, sidomos kur kalon struktura te funksionet ose kur ndryshon vlerat e tyre.

Për të përdorur një tregues për një strukturë, thjesht shto simbolin, siç do të bëje me tipe të tjera të dhënash.*

Për të aksesuar anëtarët e saj, duhet të përdorni operatorin në vend të sintaksës së pikës: **->**.

Shembull

```
// Define a struct
struct Car {
    char brand[30];
    int year;
};

int main() {
    struct Car car = {"Toyota", 2020};

    // Declare a pointer to the struct
    struct Car *ptr = &car;

    // Access members using the -> operator
    printf("Brand: %s\n", ptr->brand);
    printf("Year: %d\n", ptr->year);

    return 0;
}
```

Kalimi i treguesve të strukturës për funksionet

Ja si mund të kaloni një tregues strukturor te një funksion dhe të ndryshoni vlerat e tij:

Shembull

```
struct Car {
    char brand[30];
    int year;
};

// Function that takes a pointer to a Car struct and updates the year
void updateYear(struct Car *c) {
    c->year = 2025; // Change the year
}

int main() {
    struct Car myCar = {"Toyota", 2020};

    updateYear(&myCar); // Pass a pointer so the function can change the year

    printf("Brand: %s\n", myCar.brand);
    printf("Year: %d\n", myCar.year);

    return 0;
}
```

Pse të përdorni Struct Pointers?

Përdorimi i treguesve me struktura është i dobishëm kur:

- **Dëshiron të shmangësh kopjimin e sasisë së madhe të të dhënave.**
Në vend që të kopjosh një strukturë të tërë, mund të kalosh thjesht një tregues. Kjo e bën programin tuaj më të shpejtë dhe përdor më pak memorie.

- **Dëshiron të ndryshosh vlerat brenda një funksioni.**

Nëse kalon një tregues të një strukture në një funksion, funksioni mund të ndryshojë vlerat origjinale.

- **Dëshiron të krijosh struktura në mënyrë dinamike duke përdorur alokimin e memories.**

Me tregues, mund t'i përdorësh për të krijuar struktura ndërkohë që programi është në funksion.

Do të mësoni më shumë rreth [menaxhimit të memories në një kapitull të mëvonshëm](#). `malloc()`

Këshillë: Nëse po punoni me programe të mëdha ose shumë vlera, treguesit e strukturës mund të ndihmojnë që kodi juaj të jetë më i pastër dhe më efikas.

C Unions

Unioni është i ngjashëm me strukturën `struct` në atë që mund të ruajë anëtarë të llojeve të ndryshme të të dhënave.

Megjithatë, ka disa ndryshime:

- Në një **struct**, çdo anëtar ka memorien e vet.
- Në një **union**, të gjithë anëtarët **ndajnë të njëjtën memorie**, që do të thotë se mund të përdorni vetëm një nga vlerat në të njëjtën kohë.

Shumicën e kohës, do të përdorni `struktura` në vend të unioneve, pasi mund të ruajë dhe aksesojë **vlera të shumta në të njëjtën kohë**, gjë që është më e zakonshme në programet e përditshme.

Megjithatë, unionet janë të dobishme kur duhet të ruani vetëm një nga disa lloje të mundshme në të njëjtën kohë, dhe dëshironi të **kurseni memorie**.

Deklaro një union / Bashkim

Për të krijuar një union, përdor fjalën kyçe **union**, dhe pastaj krijo një variabël prej saj (ashtu si me strukturat):

Shembull

```
union myUnion {           // Union declaration
    int myNum;              // Member (int)
    char myLetter;          // Member (char)
    char myString[30];      // Member (char array)
};
```

```
int main() {  
    union myUnion u1; // Create a union variable with the name "u1":  
    return 0;  
}
```

Anëtarët e Access Union

Dhe ashtu si me strukturat, për të aksesuar anëtarët e një unioni, përdorni sintaksën `.pikë`.

E rëndësishme: Meqenëse të gjithë anëtarët ndajnë të njëjtën kujtesë, ndryshimi i njërit ndikon edhe te të tjerët. Vetëm **anëtari i fundit i caktuar** mban një vlerë të vlefshme:

Shembull

```
union myUnion {  
    int myNum;  
    char myLetter;  
    char myString[30];  
};  
  
int main() {  
    union myUnion u1;  
  
    u1.myNum = 1000;  
  
    // Since this is the last value written to the union, myNum no longer holds 1000  
    - its value is now invalid  
    u1.myLetter = 'A';  
  
    printf("myNum: %d\n", u1.myNum); // This value is no longer reliable  
    printf("myLetter: %c\n", u1.myLetter); // Prints 'A'  
  
    return 0;  
}
```

Madhësia e një unioni

Madhësia e një **union** do të jetë gjithmonë e njëjtë me madhësinë e anëtarit të saj më të madh:

Shembull

```
union myUnion {  
    int myNum;  
    char myString[36];  
};  
  
int main() {  
    union myUnion u1;  
    printf("Size of union: %zu bytes\n", sizeof(u1));  
}
```

```
return 0;
}
```

Duke qenë se anëtari më i madh ka 36 bajte, i gjithë **union** do të jetë gjithashtu **36 bajte**.

Nëse do të ishte **struct**, madhësia do të ishte 40 bajte:
myNum (4 bajtë) + **myString** (36 bajtë) = **gjithsej 40 bajtë**

Sqarim me shembull:

Shembull

```
union Data {
    int numri;
    float decimal;
    char shkronja;
};
```

Mund të krijosh një variabël:

```
union Data d;
```

```
d.numri = 10;
printf("%d\n", d.numri);
```

Por nëse më pas bën:

```
d.decimal = 3.14;
```

vlera e d.numri nuk ruhet më në mënyrë të vlefshme, sepse numri dhe decimal përdorin **të njëjtën hapësirë memorie**.

```
struct VS union
struct A {
    int x;
    float y;
};
```

```
union B {
    int x;
    float y;
};
```

Te struct A, x dhe y ekzistojnë njëkohësisht.

Te union B, zakonisht përdoret **vetëm njëri prej tyre në një moment të caktuar**.

Për shembull, nëse int është 4 byte dhe float është 4 byte:

```
sizeof(struct A) // zakonisht 8 byte
sizeof(union B)  // zakonisht 4 byte
```

Kur përdoren unionet

Përdorni unionet kur:

- Duhet të ruash lloje të ndryshme në të njëjtin vend
- Përdor vetëm një lloj në të njëjtën kohë
- Ruajtja e kujtesës është shumë e rëndësishme

Pra, ideja kryesore: union përdoret kur do që disa lloje të dhënash të **ndajnë të njëjtën hapësirë memorie**, zakonisht për të kursyer memorie ose për të interpretuar të njëjtat të dhëna në mënyra të ndryshme.

C typedef

Fjala kyçe **typedef** ju lejon të krijoni një emër të ri (një **pseudonim**) për një lloj ekzistues. Kjo mund ta bëjë më të lehtë leximin e deklaratave komplekse dhe mirëmbajtjen e kodit tuaj.

Për shembull, në vend që gjithmonë të shkruajmë **float**, mund të krijojmë një tip të ri të quajtur **Temperature** për ta bërë kodin më të qartë:

Shembull

```
#include <stdio.h>

typedef float Temperature;

int main() {
    Temperature today = 25.5;
    Temperature tomorrow = 18.6;

    printf("Today: %.1f C\n", today);
    printf("Tomorrow: %.1f C\n", tomorrow);

    return 0;
}
```

Këtu, **Temperature** është thjesht një emër tjetër për **float**. Megjithatë, kodi është më shprehës: na tregon se këto numra janë **temperatura**, jo thjesht numra **float** me presje lëvizëse.

Pse të përdorni typedef?

- **Thjeshton kodin:** Emra shkronjash më të shkurtër dhe më të lehtë për t'u lexuar.
- **Përmirëson qartësinë:** Shpreh më mirë qëllimin (për shembull, **AGE** në vend të thjesht **int**). Gjithashtu ndihmon të shmanget konfuzioni kur shumë variabla ndajnë të njëjtin lloj baze (si **float** ose **double**).

Mirë të dihet: Në C-në moderne, **typedef** përdoret shpesh së bashku me **struct**, **enum** dhe **pointer**/tregues funksionesh për të mbajtur kodin të pastër dhe më të lehtë për t'u lexuar.

Typedef me strukturë

typedef mund të jetë i dobishëm me **struct**, sepse ju lejon të shmangni shkrimin çdo herë: **struct**

Shembull

```
#include <stdio.h>

// Without typedef:
struct Car {
    char brand[30];
    int year;
};

// With typedef:
typedef struct {
    char brand[30];
    int year;
} Car;

int main() {
    struct Car car1 = {"BMW", 1999}; // needs "struct"
    Car car2 = {"Ford", 1969}; // shorter with typedef

    printf("%s %d\n", car1.brand, car1.year);
    printf("%s %d\n", car2.brand, car2.year);
    return 0;
}
```

Ky shembull tregon se si **typedef** e bën më të lehtë punën me shumë variabla strukturore, si modelet e ndryshme të makinave:

Shembull

```
#include <stdio.h>

// With typedef
typedef struct {
    char brand[30];
    char model[30];
    int year;
} Car;

int main() {
    Car car1 = {"BMW", "X5", 1999};
    Car car2 = {"Ford", "Mustang", 1969};
    Car car3 = {"Toyota", "Corolla", 2011};

    printf("%s %s %d\n", car1.brand, car1.model, car1.year);
    printf("%s %s %d\n", car2.brand, car2.model, car2.year);
    printf("%s %s %d\n", car3.brand, car3.model, car3.year);

    return 0;
}
```

Këtu mund të shihni se si **typedef** mban kodin të pastër kur strukturat janë të vendosura brenda strukturave të tjera:

Shembull

```
#include <stdio.h>

// Define three structs with typedef
typedef struct {
    char firstName[20];
    char lastName[20];
} Owner;

typedef struct {
    char brand[20];
    int year;
    Owner owner;
} Car;

typedef struct {
    char name[30];
    Car featuredCar;
} Dealership;

int main() {
    Owner person = {"John", "Doe"};
    Car car1 = {"Toyota", 2010, person};
    Dealership d = {"City Motors", car1};

    printf("Dealership: %s\n", d.name);
    printf("Featured Car: %s (%d), owned by %s %s\n",
        d.featuredCar.brand,
        d.featuredCar.year,
        d.featuredCar.owner.firstName,
        d.featuredCar.owner.lastName);

    return 0;
}
```

Është në dorën tënde nëse do ta përdorësh **typedef** apo jo. Kodi yt do të funksionojë njësoj edhe pa të. Megjithatë, në C-në moderne përdoret shpesh për ta bërë kodin më të shkurtër, më të qartë dhe më të lehtë për t'u mirëmbajtur.

Mbushja e strukturës

Kur krijoni një strukturë në C, kompilatori mund të shtojë disa **bajte shtesë mbushjeje** midis anëtarëve.

Kjo bëhet për ta bërë programin të funksionojë më shpejt në kompjuterin tuaj, sepse shumica e CPU-ve lexojnë të dhëna më efikas kur ato janë **të përshtatura siç duhet** në memorje.

Rrallëherë ke nevojë të shqetësohesh për këtë, përveç nëse punon me memorie ose formate skedarësh të nivelit të ulët.

Në pak fjalë: Mbushja do të thotë që kompilatori ndonjëherë shton hapësira bosh brenda një strukture për të mbajtur gjërat të shpejta dhe të rreshtuara siç duhet në memorie.

Le të shohim një strukturë të thjeshtë:

Shembull

```
struct Example {  
    char a; // 1 byte  
    int b;  // 4 bytes  
    char c; // 1 byte  
};  
  
int main() {  
    printf("Size of struct: %zu bytes\n", sizeof(struct Example));  
    return 0;  
}
```

Size of struct: 12 bytes

Mund të prisni që madhësia të jetë - por zakonisht printon **12 bajtë!**

1 + 4 + 1 = 6 bytes

Pse?

Kompilatori shton **bajte mbushëse** në mënyrë që anëtari (**int**) të fillojë nga një adresë memorie që është shumëfish i 4. Kjo ndihmon CPU-në ta lexojë më shpejt. **b**

Ja si është në të vërtetë e organizuar kujtesa:

Member	Bytes	Notes
a	1	Stored first
padding	3	Added so b starts at a multiple of 4

b	4	Aligned to 4-byte boundary
c	1	Stored next
padding	3	Added to make total size a multiple of 4

Totali = 1 + 3 + 4 + 1 + 3 = **12 bajte**.

Si të Reduktohet Mbushja

Mbushja varet nga **renditja** e anëtarëve në strukturë. Nëse gruponi fillimisht llojet më të mëdha, mund ta bëni strukturën më të vogël:

Shembull

```
struct Example {  
    int b; // 4 bytes  
    char a; // 1 byte  
    char c; // 1 byte  
};  
  
int main() {  
    printf("Size of struct: %zu bytes\n", sizeof(struct Example)); // Usually 8  
    bytes  
    return 0;  
}
```

Tani struktura është vetëm **8 bajtë** në vend të 12, sepse nevojitet më pak mbushje.

Pse Ka Rëndësi

- Mbushja ndihmon programet të funksionojnë më shpejt duke mbajtur të dhënat të përshtatura në memorje.
- Mund të bëjë strukturat më të mëdha se sa pritej.
- Riorganizimi i anëtarëve mund të zvogëlojë madhësinë totale.
- **Shënim:** Rrallëherë ke nevojë të shqetësohesh për këtë, përveç nëse punon me memorie ose formate skedarësh të nivelit të ulët.

Strukturat kundrejt unioneve

Deri tani, kemi folur për mbushjen e brendshme **të strukturave**. Por çfarë ndodh me [unionet ose sindikatat](#)?

Unionet / bashkues ruajnë të gjithë anëtarët në **të njëjtin vend memorieje**, kështu që **nuk ka mbushje midis anëtarëve**. Megjithatë, si strukturat ashtu edhe unionet ndjekin rregullat e **përshtatjes** - të dhënat duhet të fillojnë në adresat e memories që përputhen me madhësinë e tipit të tyre.

Shembull

```
struct S {
    char a;
    int b;
    char c;
};

union U {
    char a;
    int b;
    char c;
};

int main() {
    printf("Struct size: %zu\n", sizeof(struct S));
    printf("Union size: %zu\n", sizeof(union U));
    return 0;
}
```

Rezultati tipik:

```
Struct size: 12
Union size: 4
```

Shpjegim:

- **Struktura** - anëtarët ruhen njëri pas tjetrit, kështu që ndërmjet tyre shtohet mbushje.
- **Bashkimi** - të gjithë anëtarët ndajnë të njëjtën memorie, kështu që vetëm anëtari më i madh vendos madhësinë totale të tij.

Feature	Struct	Union
Members stored	One after another	All share the same memory space

Padding between members	Yes	No
Aligned to	Each member's type	Largest member's type
Total size	Sum of members + padding	Size of largest member

Këshillë: Zakonisht nuk ke pse të shqetësohesh për këtë, por ndihmon të dish pse [strukturat](#) ndonjëherë marrin më shumë memorie se sa pritej, ndërsa [unionet/bashkuesit/](#) jo.

Përmbledhje

- Anëtarët e strukturës janë të rreshtuar bazuar në madhësinë e llojit të të dhënave të tyre.
- Kompilatori mund të shtojë bajte mbushëse për të përshtatur të dhënat siç duhet.
- Madhësia totale e strukturës shpesh është më e madhe se shuma e anëtarëve të saj.
- Riorganizimi i anëtarëve mund të reduktojë mbushjen dhe të kursejë memorie.

C Enums – Lista specifike

Një **enum** është një lloj i veçantë që përfaqëson një grup konstantash (vlera të pandryshueshme).

Për të krijuar një enum, përdor fjalën kyçe **enum**, pasuar nga emri i enum-it, dhe ndarja e enum-it me presje, **jashtë main()**, por qe thirret nga main:

```
enum Level {
    LOW,
    MEDIUM,
    HIGH
};
```

Vini re se artikulli i fundit nuk ka nevojë për presje.

Nuk është e detyrueshme përdorimi i shkronjave të mëdha, por shpesh konsiderohet si praktikë e mirë.

Enum është shkurtim për "enumerations", që do të thotë "të listuara në mënyrë specifike".

Për të aksesuar enum-in, duhet të krijosh një variabël të tij.

Brenda metodës `main()`, specifikoni fjalën kyçe `enum`, e ndjekur nga emri të enumit (`Level`) dhe pastaj emrin e variablës `enum` (`myVar` në këtë Shembull):

```
enum Level myVar;
```

Tani që keni krijuar një `enum` variabël (`myVar`), mund të caktoni një vlerë për të.

Vlera e caktuar duhet të jetë një nga elementet brenda enumit (`LOW`, `MEDIUM` ose `HIGH`):

```
enum Level myVar = MEDIUM;
```

Në mënyrë të paracaktuar, elementi i parë () ka vlerën 0, i dyti () ka vlerën 1, etj. `LOW=0MEDIUM=1`

Nëse tani përpiqesh të printosh `myVar`, do të dalë , që përfaqëson :`1MEDIUM`

```
int main() {  
    // Create an enum variable and assign a value to it  
    enum Level myVar = MEDIUM;  
  
    // Print the enum variable  
    printf("%d", myVar);  
  
    return 0;  
}
```

jep 1

Ndryshimi i Vlerave

Siç e dini, objekti i parë i një enumi ka vlerën 0. E dyta ka vlerën 1, dhe kështu me radhë.

Për të kuptuar më mirë vlerat, mund t'i ndryshoni lehtësisht:

```
enum Level {  
    LOW = 25,  
    MEDIUM = 50,  
    HIGH = 75  
};  
  
printf("%d", myVar); // Now outputs 50
```

Vini re se nëse i caktoni një vlerë një artikulli specifik, artikujt e ardhshëm do të përditësojnë numrat e tyre në përputhje me këtë:

```
enum Level {  
    LOW = 5,  
    MEDIUM, // Now 6  
    HIGH // Now 7  
};
```

Enum në një deklaratë çelës / switch

Enum-et përdoren shpesh në deklaratat e çelësit për të kontrolluar vlerat përkatëse:

```
enum Level {
    LOW = 1,
    MEDIUM,
    HIGH
};

int main() {
    enum Level myVar = MEDIUM;

    switch (myVar) {
        case 1:
            printf("Low Level");
            break;
        case 2:
            printf("Medium level");
            break;
        case 3:
            printf("High level");
            break;
    }
    return 0;
}
```

Typedef me Enum

Mund ta përdorni **enum** edhe me **typedef**. Kjo e bën më të lehtë deklarinimin e variablave të tipit enum, pa pasur nevojë të shkruhet **enum** çdo herë:

Shembull

```
// Without typedef
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
enum Day today = WED;

// With typedef
typedef enum {MON, TUE, WED, THU, FRI, SAT, SUN} Day;
Day today = WED;
```

Të dy versionet funksionojnë njësoj, por versioni **typedef** është më i shkurtër dhe më i lehtë për t'u lexuar:

Shembull

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

```
typedef enum {MON, TUE, WED, THU, FRI, SAT, SUN} Day;
```

```
int main() {
    Day today = WED;
    if (today == WED) {
        printf("It is Wednesday!\n");
    }
    return 0;
}
```

Pse dhe kur të përdorni enums?

Enum-et përdoren për t'u dhënë emra konstantave, gjë që e bën kodin më të lehtë për lexim dhe mirëmbajtje.

Përdor enums kur ke vlera që e di se nuk do të ndryshojnë, si ditë mujore, ditë, ngjyra, pako kartash, etj.

Sfida: Ruani një ditë jave duke përdorur një Enum

Testoni kuptimin tuaj për **C enums** duke përfunduar një sfidë të vogël kodimi.

Udhëzime

Përfundoni hapat më poshtë (ndiqni rendin):

1. Brenda `main()`, deklaro një variabël të tipit `enum Day` të quajtur `today`
2. Vendos `today` në `MON`
3. Printoni `today` duke përdorur specifikuesin e formatit të saktë

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

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
```

```
int main() {
    // 1) Declare the enum variable here
    // 2) Set the value here
    // 3) Print the value here
```

```
    return 0;
}
```

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

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
```

```
int main() {
```

```
enum Day today;// 1) Declare the enum variable here
today=MON;// 2) Set the value here
printf("%d",today);// 3) Print the value here
```

```
return 0;
}
```

Del 1

Checklist

- ✓ Include stdio.h
- ✓ Përcaktimi enum Day
- ✓ Përcaktimi enum Day before main()
- ✓ int main()
- ✓ Deklarimi enum Day today
- ✓ vendos today to MON
- ✓ Print today using %d or %i
- ✓ Fundi return 0;
- ✓ Mbyllja me }

Menaxhimi i Memories C

Menaxhimi i memories është procesi i trajtimit të sasisë së memories që përdor një program përmes operacioneve të ndryshme.

Memoria në C

Të kuptosh se si funksionon memoria në C është e rëndësishme. Kur krijon një variabël bazë, C automatikisht rezervon hapësirë për atë variabël. Një variabël **int**, për shembull, zakonisht zë 4 bajtë memorie, ndërsa një variabël **double** zë 8 bajtë memorie.

Mund të përdorni operatorin për të gjetur madhësinë e llojeve të ndryshme: **sizeof**

Shembull

```
int myInt;
float myFloat;
double myDouble;
char myChar;

printf("%zu\n", sizeof(myInt)); // 4 bytes
printf("%zu\n", sizeof(myFloat)); // 4 bytes
printf("%zu\n", sizeof(myDouble)); // 8 bytes
printf("%zu\n", sizeof(myChar)); // 1 byte
```

Pse është e rëndësishme të dihet?

Nëse krijon një program që zë shumë ose memorie të panevojshme, mund të rezultojë në performancë të ngadaltë dhe të dobët.

Në C, duhet të menaxhosh vetë memorien. Është një detyrë komplekse, por gjithashtu mjaft e fuqishme kur përdoret si duhet: Menaxhimi i duhur i memories së kompjuterit optimizon performancën e programit, kështu që është e dobishme të dish si të lirosh memorien kur nuk është më e nevojshme dhe të përdorësh vetëm aq pak sa është e nevojshme për detyrën.

Në kapitujt e mëparshëm mësove për [adresat e memories](#) dhe [treguesit ose pointer](#).

Të dyja kanë rëndësi kur bëhet fjalë për menaxhimin e memories, pasi është e mundur të punohet drejtpërdrejt me memorien përmes [treguesve / pointer](#).

Por ki kujdes; **treguesit pointer duhet të trajtohen me kujdes**, pasi është e mundur të dëmtohen të dhënat e ruajtura në adresa të tjera të memories.

Menaxhimi i memories

Menaxhimi i memories është procesi i trajtimit të sasisë së memories që përdor një program përmes alokimit, rishpërndarjes dhe dealokimit (shpesh i referuar si "çlirim"). Ne do të prezantojmë secilën nga këto tema në kapitujt në vijim.

Procesi i rezervimit të memories quhet alokim. Mënyra e alokimit të memories varet nga lloji i memories.

C ka dy lloje memorieje: memorie statike dhe memorie dinamike.

Memoria Statike

Memoria statike është memorie që rezervohet për variabla **para** se programi të ekzekutohet. Alokimi i memories statike njihet gjithashtu si alokimi i *memories në kohën e kompilimit*.

C alokon automatikisht memorien për çdo variabël kur programi kompilohet.

Për shembull, nëse krijoni një varg integer me 20 studentë (p.sh. për një semestër vere), C do të rezervojë hapësirë për 20 elementë që zakonisht janë 80 bajtë memorie (20 * 4):

Shembull

```
int students[20];  
printf("%zu", sizeof(students)); // 80 bytes
```

Por kur fillon semestri, rezulton se vetëm 12 studentë marrin pjesë. Atëherë ke humbur hapësirën e 8 elementeve të përdorura.

Pasi nuk mund të ndryshoni madhësinë e vargut, mbeteni me memorie të rezervuar të panevojshme.

Vini re se programi do të vazhdojë të ekzekutohet dhe nuk është dëmtuar në asnjë mënyrë. Por nëse programi yt përmban shumë nga ky lloj kodi, mund të funksionojë më ngadalë se sa do të funksiononte në mënyrë optimale.

Nëse dëshironi kontroll më të mirë të memories së alokuar, shikoni Memorien Dinamike më poshtë.

Memoria Dinamike

Memoria dinamike është memoria që alokohet **pasi** programi fillon të ekzekutohet. Alokimi i memories dinamike mund të quhet gjithashtu alokim i memories *në kohë ekzekutimi*.

Ndryshe nga memoria statike, ke kontroll të plotë mbi sa memorie po përdoret në çdo moment. Mund të shkruash kod për të përcaktuar sa memorie të duhet dhe ta alokosh atë.

Memoria dinamike nuk i përket një variabli, ajo mund të aksesohet vetëm me tregues.

Për të alokuar memorien dinamike, mund të përdorni funksionet or. Është e nevojshme të përfshihet titulli për t'i përdorur ato. Funksionet dhe caktojnë pak memorie dhe kthejnë një tregues në adresën e tij. `malloc()` `calloc()` `<stdlib.h>` `malloc()` `calloc()`

```
int *ptr1 = malloc(size);
int *ptr2 = calloc(amount, size);
```

Funksioni ka një parametër, *madhësinë*, i cili përcakton sa memorie duhet të alokohet, i matur në bajte. `malloc()`

Funksioni ka dy parametra: `calloc()`

- *Sasia* - Specifikon sasinë e artikujve që do të alokohen
- *madhësia* - Specifikon madhësinë e çdo artikulli të matur në bajte

Shënim: Të dhënat në memorien e alokuar nga janë të paparashikueshme. Për të shmangur vlerat e papritura, sigurohuni të shkruani diçka në memorie para se ta lexoni. `malloc()`

Ndryshe nga , funksioni shkruan zero në të gjithë memorien e alokuar. Megjithatë, kjo e bën pak më pak efikas. `malloc()` `calloc()` `calloc()`

Mënyra më e mirë për të alokuar sasinë e duhur të memories për një lloj të dhëne është përdorimi i operatorit: **sizeof**

```
int *ptr1, *ptr2;
ptr1 = malloc(sizeof(*ptr1));
ptr2 = calloc(1, sizeof(*ptr2));
```

Kujdes: i thotë C-së të matë madhësinë e të dhënave në adresë. Nëse harron dhe shkruan në vend të tij, ai do të matë madhësinë e treguesit vetë, që zakonisht janë 8 bajtët që nevojiten për të ruajtur një adresë memorie. **sizeof(*ptr1)*sizeof(ptr1)**

Shënim: Operatori nuk mund të matë sa memorie dinamike i është alokuar. Kur mat memorien dinamike, të tregon vetëm madhësinë e llojit të të dhënave të memories. Për shembull, nëse rezervoni hapësirë për 5 vlera, operatori do të kthejë 4, që është numri i bajtëve të nevojshëm për një vlerë të vetme. **sizeoffloatsizeoffloat**

Le të përdorim memorien dinamike për të përmirësuar shembullin e mësipërm të nxënësve.

Siç u përmend më parë, nuk mund të përdorim për Mat sa memorie është alokuar, duhet ta llogarisim duke shumëzuar Sasia e artikujve sipas madhësisë së tipit të të dhënave: **sizeof**

Shembull

```
int *students;
int numStudents = 12;
students = calloc(numStudents, sizeof(*students));
printf("%d", numStudents * sizeof(*students)); // 48 bytes
```

Shënime

Kur punoni me alokim dinamik të memories, duhet gjithashtu **të kontrolloni për gabime** dhe **memorie të lirë** në fund të programit. Do të mësoni më shumë rreth kësaj në kapitujt e ardhshëm.

Memoria e grumbullit

Për plotësim, vlen të përmendet memoria stack. Memoria stack është një lloj memorie dinamike që rezervohet për variabla që deklarohen brenda funksioneve. Variablat e deklaruara brenda një funksioni përdorin memorien stack në vend të memories statike.

Kur thirret një funksion, memoria e grumbullit caktohet për variablat në funksion. Kur funksioni kthehet, memoria e stack-ut lirohet.

Është mirë të jeni të vetëdijshëm për memorien stack për të menaxhuar përdorimin e memories nga thirrjet e funksioneve të ndërlidhura dhe rekursionin. Rekursioni që përsëritet shumë herë mund të marrë shumë memorie stack. Kur ndodh kjo, quhet **mbingarkesë e stack-ut**.

Makro janë me te shpejte se funksionet

Në gjuhën C, direktivat (makrot e preprocesorit) janë më të shpejta në ekzekutim sesa funksionet e zakonshme, sepse ato zëvendësohen në kodin burimor para kompilimit. Megjithatë, *funksionet inline ofrojnë një kompromis: pothuajse të njëjtën shpejtësi si makrot, por me më shumë siguri dhe kontroll nga kompajleri*.

Direktiva (Makro)

- **Ekzekutim:** Nuk ka thirrje funksioni; kodi zëvendësohet direkt gjatë kompilimit → **shpejtësi maksimale**.
- **Kosto:** Mund të rrisë madhësinë e kodit (code bloat), sepse çdo përdorim zëvendësohet me tekstin e makros.
- **Rreziqe:** Nuk kontrollohen tipet e variablave, gjë që mund të sjellë gabime të fshehura.

Shembull:

```
c
#define SQUARE(x) ((x) * (x))
```

Funksionet Inline

- **Ekzekutim:** Kompajleri i vendos trupin e funksionit direkt në vendin e thirrjes → pothuajse pa overhead.
- **Avantazh:** Kontroll i tipave, debug më i lehtë, dhe optimizim më i mirë nga kompajleri.
- **Kosto:** Mund të mos inlinohen gjithmonë; varet nga vendimet e kompajlerit.
- **Shembull:**

```
c
inline int square(int x) {
    return x * x;
}
```

Krahësim i Shpejtësisë

Tipi	Shpejtësia	Siguria e tipave	Madhësia e kodit	Debugging
Makro	Shumë e lartë	Jo	Mund të rritet	Vështirë
Inline func.	Shumë e lartë (afër makros)	Po	Më e kontrolluar	Më e lehtë

Përfundim

- **Makrot** janë më të shpejta në teori, sepse nuk ka thirrje funksioni, por rrezikojnë gabime dhe rrisin madhësinë e kodit.
- **Funksionet inline** janë zakonisht zgjedhja më e mirë: ofrojnë shpejtësi të krahasueshme me makrot, por me siguri dhe optimizim më të mirë nga kompajleri.
- Në praktikë, **dallimi i shpejtësisë është minimal** në shumicën e rasteve moderne, sepse kompajlerët optimizojnë shumë mirë.