ELHUYAR

C PROGRAMAZIO-
-LENGOAIA

Iñaki Alegria - Nestor Garai

ELHUYAR

C PROGRAMAZIO-
-LENGOAIA

Egileak: *Iñaki Alegria*
Nestor Garai
UPV/EHUko Konputagailuen Arkitektura eta Teknologia Saila

Hezkuntza, Unibertsitate eta Ikerketa Sailaren dirulaguntza jaso du. (1995-XII-13)

© ELHUYAR, K.E. Asteasuain 14. Txikierdi. 20170 USURBIL (Gip.) (1995)
Lege-gordailua:
ISBN: 84-87114-03-2

Inprimatzailea: Lizarra inprimategia, S.L. Tafallarako bidea, 1. km. LIZARRA (Naf.)

AURKIBIDEA

HITZAURREA

"C programazio-lengoaia" izeneko liburu hau argitaratzeko ideia zaharra da. UEUn orain dela hamar bat urte Ander Basalduak azaldu zigun lengoaia honek zuen etorkizuna. Zenbait urte geroago, UEUn bertan, ikastaro bat prestatu genuen, liburu honetan azaltzen diren zenbait adibide ordukoak izanik. Orain dela bi urte, eta liburuaren prestakuntzaren lehen fase gisa, *Elhuyar Zientzia eta Teknika* aldizkarian 14 kapituluko ikastaro bat prestatu genuen hainbat zenbakitan Montse Maritxalarren laguntzarekin. Ikastaro haren materiala liburu honen oinarria izan arren, azken urte honetan liburua osatu da, testua birplanteatuz eta osatuz, adibideak gehituz eta programak konpilatuz eta egiaztatuz. Azkenik, C lengoaia ordezkatzen ari den C++ lengoaiari buruzko kapitulu bat gehitzea egoki iruditu zaigu, eta lan horretan Basilio Sierrak emandako laguntza eskertu nahi dugu. UPV/EHUko Konputagailuen Arkitektura eta Teknologia gure saileko hainbat lankiderekin taldean egindako lana oso lagungarria izan da liburuaren edukia gauzatzean.

Liburuaren testua ikastaroarena baino osatuagoa eta landuagoa bada ere, adibideen garrantzia mantendu egin da, C lengoaian programatu nahi duenarentzat laguntza-tresna bezala ulertu behar baita liburu hau. Balizko irakurleak, ikasleak zein profesionalak izan daitezke, baina programazioaren oinarri minimoak ezagutu beharko lukete liburuari etekina ateratzeko.

Liburuaren egitura nahikoa arrunta da. Lengoaiaren nondik-norakoak azaltzen dituen sarreraren ondoren oinarrizko osagaiak azaltzen dira lehen kapituluetan: datu-motak, eragileak, espresioak, sententziak, programazio--egiturak eta funtzioak. Seigarren kapitulutik hamabigarrenera arte kontzeptu

aurreratuagoak azaltzen dira: taulak, erakusleak, datu-egiturak, liburutegia, sarrera-irteera etab. Aurretik azaldutakoaren aplikazio praktiko pare bat erakusten da hamahirugarren kapituluan, C++ lengoaiaren sarrera batekin bukatzeko.

Hitzaurrea bukatu baino lehen, barkamena eskatu nahi dugu liburuan zehar ager daitezkeen akatsengatik, eta zuen iradokizunak ere eskertuko ditugu hutsuneak osatu eta gera daitezkeen akatsak zuzendu ahal izateko.

Donostia, 1995eko Uztaila.

1. SARRERA

1.1. C-REN HISTORIA

C lengoaia sortu eta lehenengo aldiz inplementatu zuena Dennis Ritchie izan zen 1972. urtean. AT&T Bell Labs-eko DEC PDP-11 batean jarri zuen martxan, UNIX* sistema eragilea idazteko lengoaia bezala.

Hasierako urteetan, C-ren estandarra UNIX sistema eragilearekin batera banatu zen bertsioa izan zen. C-ri buruzko erreferentziazko lehenengo testua D. Ritchie-ren "The C Reference Manual"-a izan zen, eta honetan oinarriturik, Brian Kernighan eta D. Ritchie-k, 1978. urtean "The C Programming Language" argitaratu zuten, bertan K&R estandarra (UNIX makinek eskaintzen dutena, alegia) bezala ezagutu ohi dena deskribatzen delarik.

Urteak pasa ahala, mikrokonputagailuen hedapenaren ondorioz, C inplementazio desberdin asko sortu zen. Konpiladore berriak azaldu ziren, makina berrietan eta sistema eragile ezberdinekin exekutatzen zirenak. Orduan sortutako arazo bat C lengoaia "txikia" izatearen ondorioa zen: konpiladore-garale bakoitzak bere gustuko ziren eraikuntzak gehitzeko joera zuen. C-k zuen garraiagarritasuna galtzeko zorian zegoenez, ANSI erakundeak C-ren estandarra definitzeari ekin zion, emaitza liburu batean argitaratuz. ANSI C, gaur egun, guztiek (IBM, DEC, Microsoft eta AT&T erakundeak barne daudelarik) onartua dagoen estandarra da, K&R estandarra UNIX sistema duten makinetan mantendu arren.

* UNIX Bell Laborategietako marka erregistratua da.

1.2. C-REN EZAUGARRI OROKORRAK

C erdi-mailako programazio-lengoaiatzat onartua izan da. Honek zera esan nahi du: goi-mailako lengoaien elementuak behe-mailako lengoaien funtzionaltasunarekin konbinatzen dituela. Goi-mailako lengoaia bezala, datu-motak kudeatzen ditu. Datu-mota batek aldagai batek eduki dezakeen balio-multzoa eta aldagai horren gainean egin daitezkeen eragiketak definitzen ditu. Behe-mailako lengoaia bezala, bit-maneiua, byte, hitzak eta erakusleak (beraz, memoriako helbideak) maneia ditzake. Ezaugarri horiek direla medio, sistemen programaziorako lengoaia erabiliena da.

Bestalde, garraiagarria da C lengoaia; beraz, iturburu-lengoaiaz makina batean idatzitako programak besteetara eraman, eta aldaketarik gabe edo oso aldaketa gutxirekin, konpila eta exekuta daitezke. Honetan, merkatuan sistema askotarako C konpiladore asko egoteak ere laguntzen du.

Gainera, C lengoaia egituratua da; ondorioz, helburu baterako behar diren aldagaiak eta aginduak isola daitezke programaren beste atalek ikus ez ditzaten. Ezaugarri hau ondo datorkigu datu-mota berri bat eta berari dagozkion eragiketak definitzerakoan, zeren azken eragiketa hauetan soilik erabili nahi diren aldagaiak ezkutatuko baitira. Beste lengoaia egituratu batzuek ahalbidetzen badute ere, C-k, ordea, ez du uzten funtzio baten gorputzaren barnean beste funtzio bat definitzen.

C-ren egitura nagusia funtzioa da, C-ren azpiprograma independentea. Funtzioek programa baten eginkizunak definitzen dute. Funtzioak fitxategi desberdinetan gorde eta banaturik konpilatu daitezke, programazio modularra bultzatuz.

C sistemen programatzaileentzako lengoaia egokia da, programatzaile hauek behar dutena eskaintzen baitu: muga eta oztopo gutxi, bloke-egiturak, funtzio independenteak eta erreserbatutako hitz-multzo trinkoa; horrela beste

lengoaietan idazteko zailak diren programak erraz idatz baitaitezke C-n. Beraz, sistema eragilea osatzen duten programak zein interpretatzaileak, editoreak, konpiladoreak eta datu-baseen kudeatzaileak motako programak idazteko gehien aukeratzen den lengoaia da. Gaur egun, aplikazio-programetan ere erabiltzen da C, lortzen diren programa exekutagarriak azkarragoak baitira beste goi-mailako lengoaietan idatzitakoak baino.

Beste ezaugarri hau ere azpimarra daiteke: oso lengoaia sinplea izan arren, ahaltsua da, zeren eta bere sententzi multzo laburraz eta daukan liburutegiaz lagunduta, mota guztietako problemak erraz ebatz baitaitezke. Halaber, esan den bezala C-z idatzitako programak, konpilatu ondoren, oso eraginkorrak gertatzen dira; oso trinkoak (memori zati txikia hartuz) eta azkarrak baitira.

Dena den, kontrako ezaugarriak ere baditu: alde batetik, programatzaileek oso zorrotz jokatzen ez badute behintzat, programak oso kriptikoak eta interpretaezinak izan daitezke. Bestetik, konpiladoreak ez du ezaugarri asko egiaztatzen eta PASCAL eta antzeko lengoaia oso egituratuek ez bezala, ez du egiaztatzeko kode exekutagarria sortzen. Horrela, adibidez, hogei osagaiko taula batean hogeita batgarren osagaia erreferentzia daiteke. Eta ehungarrena edo –2.a ere bai. Gainera taularen osagaiak hitzak badira, karaktereak edo hitz bikoitzak erreferentzia daitezke, programatzaileari askatasun handia ematen diolarik. Programatzaileak egindakoa norberak dakiela "suposatzen" du konpiladoreak, programatzaile esperientziadunentzat diseinatua baitago C lengoaia.

Beraz, erroreak detektatzea eta zuzentzea zaila gertatzen da. Azken honetarako, programa araztaile edo zorri-kentzaileek (*debugger*) gero eta gehiago laguntzen dutenez, C programatzaileei horrelako programak ezagutzea eta erabiltzea gomendatzen zaie, programa hauen helburua erroreak detektatzen eta zuzentzen laguntzea baita.

1.3. C-REN HITZ ERRESERBATUAK

Programazio-lengoaia batean badago hitz-multzo bat esanahi berezia duena, eta multzo horretan dauden hitzak erreserbatuak dira; beraz, hitz horiek ezin dira ez funtzio, ez aldagai, ezta konstanteen izen bezala erabili. C--ren hitz erreserbatuak 1.1 Taulan azaldutakoak dira.

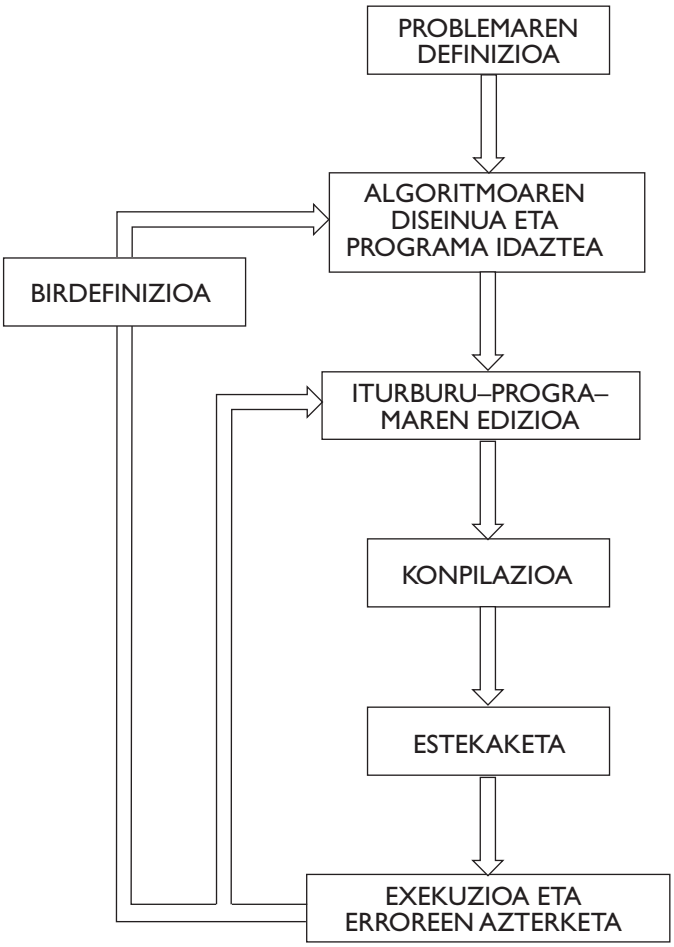
`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

1.1 Taula. C-ren hitz erreserbatuak.

Azpimarratzekoa da konpiladoreak maiuskulak eta minuskulak bereizten dituela, eta adibidez, `izena` eta `IZENA` aldagai desberdinak liratekeenez, kode osoa minuskulaz idaztea gomendatzen da, konstante parametrizatuak eta datu-mota berriak salbu, hauek maiuskulaz idatzi ohi dira eta.

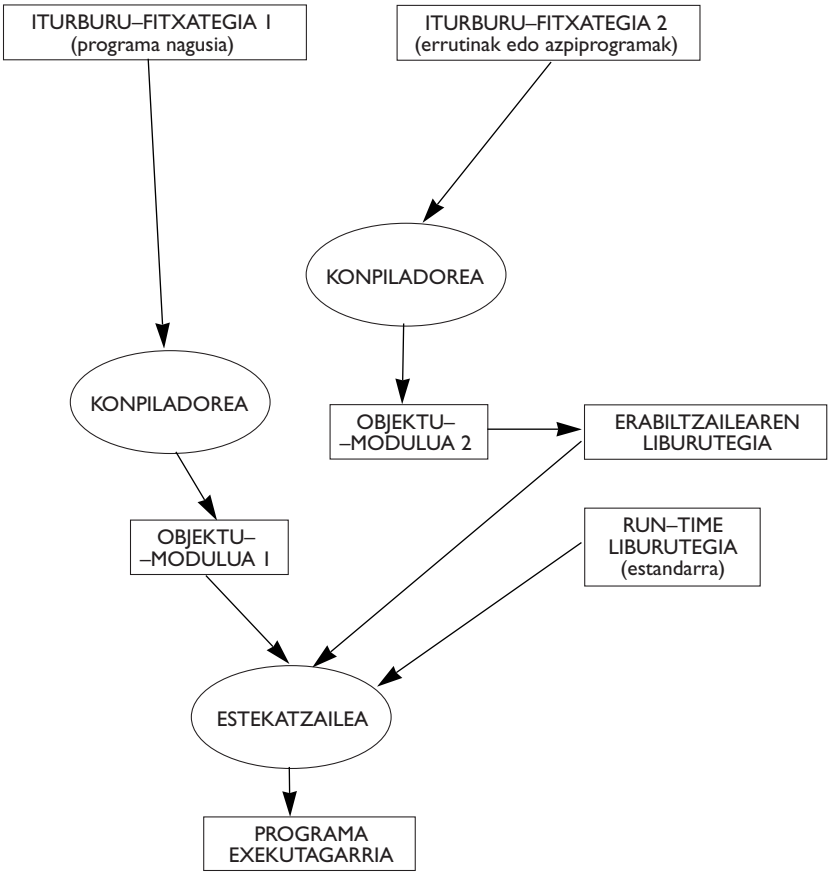
1.4. PROGRAMEN GARAPENA

Liburu honetan C lengoaia adibideen laguntzaz azalduko dugu, adibide hauek ANSI C arauak errespetatuz idatzi ondoren, konpiladorearen bidez itzuli eta probatuta direlarik. Dena den, lengoaiarekin hasi baino lehen komenigarria da 1.1 irudian azaltzen diren faseak gogoratzea: problemaren definizioa, algoritmoaren diseinua, iturburu-programaren edizioa, konpilazioa, estekaketa (*link*) eta azkenik proba.



1.1. Irudia. *Programa baten garapen-faseak*

Aurretik esandakoa gogoratuz C lengoaia sinplea izan arren oso liburutegi handia eskaintzen du, bertan sarrera/irteerako prozedurak, funtzio aritmetiko eta grafikoak, eta beste errutina asko baitaude. Horrez gain, programazio egituratuan laguntzen duen konpilazio banatua bultzatuko dugu. Konpilazio--mota hau erabiltzean, programatzaileak liburutegia erabiliko du, bere objektu-moduluak bertan gordez. Ondorioz, kontuan hartu behar den konpilazio--estekaketaren eskema 1.2 irudian agertzen dena da.



1.2. Irudia. *Konpilazioa eta estekaketa.*

1.5. PROGRAMEN EGITURA

C-ren programa guztiak funtzio batez edo gehiagoz daude osatuta. Derrigorrez azalduko den funtzioa `main()` izenekoa da, programa nagusia definitzen duena. `main()` funtzioan beste funtzio-deiak egin daitezke eta funtzio horiek hiru motakoak izan daitezke: moduluan bertan definitutako barne--funtzioak, beste moduluren batean definitutako kanpo-funtzioak eta liburutegi estandarrean definitutakoak.

C programa baten egitura orokorra 1.3 irudian azaltzen da; `f1, f2, ..., fN` erabiltzaileak definitutako funtzioak dira.

```
erazagupen globalak

main()
{    aldagai lokalak
     sententzi fluxua
}

f1()
{    aldagai lokalak
     sententzi fluxua
}

f2()
{    aldagai lokalak
     sententzi fluxua
}

...
...
...

fN()
{    aldagai lokalak
     sententzi fluxua
}
```

1.3. irudia. *C programa baten egitura.*

`{` eta `}` ikurrek funtzio edo bloke baten gorputzaren hasiera eta bukaera adierazten dute eta gorputzaren sententzia guztiak ; ikurraz bukatzen dira. 1.1 programan ondoren aztertuko dugun programa erraz bat aurkezten da.

```
#include <stdio.h>

main ( )
/* lehen programa */
{
char izena [20];

printf ("Sakatu zure izena: \n");
scanf ("%s", izena);
printf ("Elhuyar irakurriz %s C ikasten ari da\n",izena);
}
```

1.1. programa. *Gure lehen programa.*

`char izena [20]` erazagupen-sententzia da. Bertan `izena` aldagaia definitzen da 20 karaktereko taula bezala. Datu-motak eta dagozkien erazagupenak hurrengo kapituluan aztertuko ditugu.

`/*` eta `*/` karaktere-pareek iradokizun edo ohar bat mugatzen dute. Beraz, tartean dagoena ez da konpilatuko, baina programa dokumentatzen dute eta beste edonork ulergarriagoa aurkituko du. Ahalik eta ohar gehien egitea gomendatzen da, C lengoaia kriptiko bihurtzeko arriskua dago eta.

1.6. LIBURUTEGI ESTANDARRA

Programatzaileak definitutako sententziez bakarrik osatutako programa bat exekuta badaiteke ere, hau oso gutxitan ematen da, sarrera/irteerako eragiketak ez baitaude definiturik C lengoaiaren baitan. Normalean, programek C lengoaiaren liburutegi estandarrean dauden funtzioak erabiltzen dituzte, sarrera/irteerako eragiketak burutzeko besteak beste.

Konpiladore batzuetan liburutegia fitxategi handi bakarrean dagoen arren, beste batzuetan fitxategi txiki anitzetan azaltzen da, eraginkortasuna eta praktikotasuna lortuz azken era honetan.

Gure hasierako programetatik erabiliko ditugun `printf` eta `scanf` funtzioekin lortuko dugu oinarrizko sarrera/irteera. `printf` eta `scanf` dira irteera eta sarrera bideratzen dituzten liburutegi estandarreko funtzioak —parentesiek parametroak mugatzen dituzte—. Funtzio hauek erabili ahal izateko, funtzio horiek `stdio.h` fitxategian erazagutzen direla esan behar zaio C-ren aurrekonpiladoreari `#include`-aren bitartez. Argumentu-kopuru ez-finkoa dute, komatxoen artean azaltzen diren karaktere-sekuentzien araberakoa izanik. Karaktere horien barruan `%s` karaktere-katea (string) formatuaren adierazlea da. Beste formatuak ere onartzen dira; adibidez, `%d` formatu

hamartarra, `%c` karakterea, `%f` erreala eta `%e` idazkera zientifikoa. `\n` sekuentziak lerro-bukaera adierazten du.

1.7. Nola konpilatu

Programa editatu ondoren honako urrats hauek eman behar dira programa probatu baino lehen: programa konpilatuko da, ondoren estekatuko eta, azkenik, exekutatuko.

Microsoft-eko konpiladoreaz PC batean, editatu ostean honako komando burutuko dira:

```
$ msc programa
$ link programa
$ programa
```

UNIX Sistema Eragileaz berriz, edizioaren osteko pausuak hauek dira:

```
$ cc —o programa programa.c
$ programa
```

Kasu honetan lehenengo sententziak konpilazio zein estekaketa burutzen du.

1.8. Hurrengo kapituluak

Ondoren liburuaren plangintza azalduko dugu.

2. kapituluan oinarrizko datu-motak aztertuko dira, beraien ezaugarriak azalduz. 3. kapitulua eragile eta espresioei buruzkoa da, zeintzuk diren eta beraien arteko lehentasun eta asoziatibitatea aipatuz. 4. kapituluan kontrol-

-egiturak komentatuko dira kontrol-fluxua nola alda daitekeen baldintzazko edo errepikapen sententziak agertuz. 5. kapituluan funtzioak eta makroak aipatuko dira, liburutegien erabilpena eta konpilazio banatuaren oinarriak azalduz.

6. kapituluan oinarrizko datuak ez diren taulak eta erakusleak azaltzen dira, 7. kapituluan oinarrizko datuetatik abiatuz lortzen diren datu-egituren ezaugarriak biltzen dira eta 8.ean aldagaien ezaugarriak eta memoria dinamikoa adierazten dira. Konpilazio banatua finkatu asmoz, 9., 10. eta 11. kapituluek sarrera/irteerari dagozkion kontzeptuak, C-ren aurrekonpiladorearen sasiaginduak, mihiztadura-lengoaiarekiko lotura eta liburutegi estandarra azaltzen dituzte.

Kapitulu horiek ikusi eta gero, *software*-ingeniaritza C erabiliz 12. kapituluan azalduko da, nolabaiteko konplexutasuna duten proiektuak garatzeko eman behar diren pausuak adieraziz. Ondoren, hainbat aplikazio ikusiko dira 13. kapituluan, pila, ilara, lista eta zuhaitzen tratamendua barne.

Objektuei zuzendutako programazio-metodologiaren hedapena eta gaurkotasuna direla eta, azken kapitulua C-ren "oinordeko" bezala azaltzen den C++ lengoaiari buruzkoa da, C-rekiko dituen berezitasunei gainbegirada bat emanez.

A eranskinean, C-ren sintaxia aurki daitekeen bitartean, B eranskinean ANSI C eta K&R C-ren arteko bereizketak azalduko dira.

2. OINARRIZKO DATU-MOTAK

Beste lengoaietan bezala, C lengoaia erabiliz programak idazten dituguneean, datuak definitu egin behar dira. Konstanteak edo aldagaiak izanik, dagokien mota esleitzen zaie, konpiladoreak memori zatia eta bit-segida egokia eslei diezazkien.

Jakina denez, konstanteak beren balioa aldatzen ez duten datuak dira, hau da, balio finkoak; beraz ez dute identifikatzaile berezirik. Aldagaiek, ordea, ez dute beti balio bera, baina identifikatzaile batez ezagutzen dira eta balioaren aldaketak esleipenaren bitartez izaten dira. Aldagaiak erabili baino lehen definitu edo erazagutu behar diren bitartean —konpiladoreak identifikatzailea ezagut dezan—, konstanteak ez, balioa bera baita haien identifikatzailea.

2.1. DATU-MOTAK

C-k ondoko oinarrizko datuak ezagutzen ditu:

- karaktereak `(char)`
- osoko zenbakiak `(int)`
- zenbaki errealak `(float)`
- doitasun bikoitzeko zenbaki errealak `(double)`
- baliorik gabekoak `(void)` —ez zen aurrikusten lehen konpiladoreetan—

Karakterea normalean byte batekoa da, osokoa hitz batekoa, zenbaki errealа lau bytekoa eta doitasun bikoitzekoak 8 bytekoak. 2.1. taulan mota bakoitzari dagozkion ezaugarriak ikus daitezke.

MOTA	GAKO-HITZA	BIT-KOPURUA	ADIERAZPIDEA (aldagaien definizioan)
karakterea	`char`	8	ASCII
osokoa	`int`	16 edo 32	koma finkoa
erreala	`float`	32	koma higikorra
doitasun bikoitzekoa	`double`	64	koma higikor erreala
baliorik gabekoa	`void`	0	baliorik gabe

2.1. taula. Oinarrizko datuen ezaugarriak.

Mota errealen arteko desberdintasuna doitasunean datza; `float` motakoek 7 zifra hamartar dituzten bitartean `double` motakoek 15 zifrako doitasuna baitute. Dena den, konpiladore bakoitzean doitasun eta adierazpide-barrutia desberdinak izan daitezkeenez, `<limits.h>` fitxategi estandarrean aurki daitezke beraien balioak.

Beste mota batzuk ere erabil daitezke, baina beti aipatutako motak oinarritzat hartuz. Horrela karaktere-kateak karaktereen taulak izango dira, boolearrak bi balio baino erabiltzen ez dituzten osoak edo karaktereak, eta abar (ikus 6. eta 7. kapituluak). Oinarrizko datu-mota ez bada ere, erakuslea zer den azalduko dugu orain —sakonki 6. kapituluan ere azalduko da—, bitarteko kapituluetan erabilia izango baita. Erakuslea bi modutan defini daiteke: datu baten memoriako helbidea —behe-mailako lengoaien terminologia erabiliz—, edo aldagai bat erreferentziaz adierazten duen mekanismoa —goi-mailako lengoaien terminologia erabiliz—. Beti mota bati lotuta joango da erakuslea, helbideratzen edo erreferentziatzen duen datuaren mota hain zuzen.

Mota nagusien aurretik hainbat aldarazle ipin daitezke. Aldarazle bat oinarrizko motaren esanahia une bakoitzeko beharretara zehatzago egokitzeko erabiltzen da. Aldarazle posibleak `signed` (zeinuduna), `unsigned` (zeinurik gabe), `long` (luzea) eta `short` (laburra) dira. 2.2. taulak ANSI C-k onartzen dituen konbinazio guztiak azaltzen ditu, beraien bit-kopurua eta adierazpide-barrutia barne direlarik.

MOTA	BIT-KOPURUA	ADIERAZPIDE-TARTEA
`char`	8	ASCII karaktereak
`unsigned char`	8	0/255
`signed char`	8	–128/127
`int`	16 edo 32*	
`unsigned int`	16 edo 32*	
`signed int`	16 edo 32*	
`short int`	16	–32768/32767
`unsigned short int`	16	0/65535
`signed short int`	16	–32768/32767
`long int`	32	$-2^{31}/2^{31}-1$
`unsigned long int`	32	$0/2^{32}-1$
`signed long int`	32	$-2^{31}/2^{31}-1$
`float`	32	7 zifra hamartarreko doitasuna
`double`	64	15 zifra hamartarreko doitas.
`long double`	128	31 zifra hamartarreko doitas.

* Ordenadoreen arabera, beraz, ez da zeharo trukagarria.

2.2. taula. ANSI C-k onartutako azpimotak.

Aldarazle guztiak osoko zenbakiekin eta karaktereekin erabil daitezkeen bitartean, `long` aldarazlea doitasun bikoitzeko errealekin ere erabil daiteke. Hala ere, konbinazio batzuk alferrikakoak dira, adibidez `signed` osokoekin, osoko zenbakiek zeinua dutela suposatzen baita. `unsigned` edo `long` hitzak bakarrik erabil daitezke, beraien esanahia `unsigned int` edo `long int`-en berbera izanik.

2.2. KONSTANTEAK

Konstanteak ez dira erabili baino lehen erazagutu behar, zeren eta beren idazkeraren arabera dagokien mota konpiladoreak asmatzen baitu. Idazkeratik mota lortzeko konpiladoreak, besteak beste, ondoko irizpideak hartzen ditu kontuan: . hamartarra da errealen ezaugarria, ' karaktereena, " karaktere-kateena, L edo I atzizkiak long (osoko zenbaki luzea) motakoena. Konstanteak adieraztean aurrizki batzuk ere erabiltzen dira memoriaratu nahi dugun balioaren idazkera zehazteko. Adibidez, 0 aurrizkiak balioa digitu zortzitarrez osatuta dagoela dio; 0x aurrizkiak, aldiz, digitu hamaseitarrez. 2.3. taulan adibide batzuk ikus daitezke.

ADIBIDEA	ESANAHIA
'a'	a karakterea
25	25 int
25L	25 long
25.	25 float
12.5	12.5 float
12.5EØ	12.5 double
1.15E8	$1.15x10^8$ double
Ø123	123 zortzitar = 83 int
Øx53	53 hamaseitar = 83 int
"C ikastaroa"	11 karaktereko katea

2.3. taula. Konstanteen definizioa.

Idazkera zientifikoak doitasun bikoitzeko `double` mota behartzen du. Idazketa zientifikoaren ezaugarria hauxe da: bi zenbaki zehazten dira `E` edo `e` karaktereaz bereizturik, lehena, mantisa, osokoa edo erreala izan daiteke, eta bigarrenak, berretzaileak, osoa izan behar du. Horrela, `3e2` koma higikorreko konstanteari $3*10^2$ edo 300 balioa dagokio. 2.4. taulan onartzen diren eta onartzen ez diren idazkera zientifikoak ikus daitezke:

ONARTUTAKOAK	ONARTZEN EZ DIRENAK	ARRAZOIA
3.141	35	Ez du puntu dezimalik ez berretzailerik
.3333	3,500.45	Komak ez dira onartzen
0.3	4E	E letraren ondorengo zenbakia falta da
3e2	4e3.6	Berretzaileak osokoa izan behar du
5E–5		
3.7e12		

2.4. Taula. Idazkera zientifikoaren adibideak

Karaktereei dagokienean, komatxo bakunen artean jartzea nahikoa da inprimagarrientzat (adibidez, 'a', 'b', 'c'...). Karaktere ez-inprimagarriak (orga-itzulera adibidez), ordea, ezin dira teklatuaren karaktere bakunen bidez adierazi. Hau dela eta, C-k alderantzizko barra (\ karakterea) duten konstante bereziak sortu ditu, 2.5. taulan ikus daitekeenez. Alderantzizko barra duen kodea beste edozein karaktere bezala erabil daiteke.

KODEA	ESANAHIA
\a	Alerta seinale bat (normalean soinu bat) sortzen du.
\b	Atzera-karakterea
\f	Orri-saltoa
\n	Lerro-saltoa
\r	Orga-itzulera
\t	Tabulazio horizontala
\v	Tabulazio bertikala
\’	Komatxo bakuna
\"	Komatxoak
\\	Alderantzizko barra
\0	Karaktere nulua
\?	Galde-zeinua
\ddd*	Konstante zortzitarra (karaktereen ASCII kodeak jartzeko)
\xdd*	Konstante hamaseitarra (karaktereen ASCII kodeak jartzeko)

* d letrak zenbakien (zortzitarrak edo hamaseitarrak) adierazgarri bezala daude

2.5. taula. Konstante berezien kodeak.

Konstanteen idazkera ikusi ondoren, programazio egituratuak gomendatzen duen irizpide bat gogoratu nahi dugu; konstante parametrizatuak erabiltzearena hain zuzen. Irizpide honen arabera, konstanteak zuzenean erabili beharrean, konstanteei hasieran izen edo identifikatzaile bat ematen zaie eta erabilpen guztietan izen hori erabiliko da. Horrela, programak irakurgarritasuna irabaziko du, eta balioa aldatzekotan izen-ematean baino ez da aldatu behar. Konstanteei izena emateko C programetan `#define` sasiagindua erabiltzen da programen hasieran (ikus 2.1. programa). Konstante parametrizatuak erabiltzeko beste era bat badago —konstanteari izen batez gain mota bat esleitzen diona— ANSI C-ren barnean, `const` hitz erreserbatua erabiliz, jarraian azaldutako eran:

```
const mota ident;
```

C-ren konpiladoreek maiuskulak eta minuskulak ezberdintzen dituztenez, konstanteen izenak maiuskulaz eta aldagaienak minuskulaz idazteko ohitura dago.

```
#include <stdio.h>

#define PI   3.14159
main ( )
{
float r, azalera, perimetroa;

printf ("sakatu erradioaren neurria\n");
scanf ("%f", &r); /* &->erreferentziaz */
perimetroa = 2 * PI * r;
azalera = PI * r * r;
printf ("azalera: %f\n", azalera);
printf ("perimetroa: %f\n", perimetroa);
}
```

*2.1. **programa.** Konstante parametrizatuen erabilpena.*

2.3. ALDAGAIAK

Aldagaiek, lehen aipatu bezala, balioaren aldaketak jasan ditzakete. 2.1. programan ikus daitekeenez aldagai guztiak, erabili baino lehen, erazagutu egin behar dira, 2.3. taulan ikusitako gako-hitzak erabiliz, dagokien mota aurretik zehaztuko delarik. Horrela, konpiladoreak, aldagaiari programa exekutagarrian zenbat bit gorde beharko dizkion jakiteaz gain, aldagaia agertzen den eragiketetan mota-egiaztapena buru dezake.

Aldagaien erazagupena, beraz, horrela egiten da:

```
mota ident;
```

Modu honetan mota bereko aldagai bat baino gehiago defini daiteke, horretarako identifikatzaileak komen bidez bereiziz:

```
mota ident1 [, identN];
```

Ohitura ona izaten da mota bereko erazagupenak batera egitea, erreferentziak erraz gogoratzeko eta irakurgarritasuna lortzeko.

Aldagaiak edo konstanteak erazagutzean hasierako balio bat eslei dakieke, horrela egiten ez bada konpiladoreak 0 balioa —konpiladore gehienetan, ez guztietan— esleituko dielarik. Hasieraketa esplizitu hori burutzeko erazagupen-sententzian izenaren ondoren = karakterea eta ezarri nahi den balioa idatziko du, era honetan:

```
mota izena=balioa;
```

Adibidez, karaktere motako `datua` aldagaiari `a` balioa esleitzeko, ondoko sententzia erabiliko da:

```
char datua = 'a';
```

Identifikatzaileen eraketari buruz hitz egiterakoan, zera komentatu behar da: karaktere bat edo gehiago eduki behar dituzte, karaktere bakoitza alfabetikoa, numerikoa edo azpimarra izan daiteke, baina beti karaktere alfabetiko batez hasita —konpiladore batzuek azpimarraz hastea onartzen dute. Identifikatzaileek edozein luzera eduki dezakete, baina estekatze-prozesuan zera gerta daiteke: izeneko lehen sei karaktereak bakarrik hartuko dira kontutan eta, gainera, maiuskulak eta minuskulak ezberdindu gabe.

3. ERAGILEAK ETA ESPRESIOAK

Aurreko kapituluan erazagupena eta identifikazioa aztertu ondoren, aipatutako aldagai eta konstanteak espresioak osatzeko nola konbina daitezkeen ikustea izango da kapitulu honen helburua.

3.1. ESPRESIO-MOTAK

C lengoaian eragiketa bat adierazteko eragile bat erabiltzen da. Aldagaiak edota konstanteak eragileekin konbinatuz espresioak sortzen dira, eta espresio batek edo gehiagok, eragileen bidez konbinaturik, agindu bat osatzen du.

Espresio bat eragigai bat edo gehiagoz eta zero edo eragile gehiagoz osatuta dago, beti elkartuak balio bat itzultzeko helburuarekin. Honela, *a+2* espresioa zilegia da, *a*-k zuen balioa gehi bi emaitza itzultzen duena. *a* aldagaia bera bakarrik espresio bat da, 2 konstantea beste bat den bezala, biok balio bana errepresentatzen baitute. Espresio bakoitzari mota bat dagokio beti, mota hori konpiladoreak inplizituki edo programatzaileak esleitzen dutelarik.

Aurreko kapituluetako programetan erabili ditugun **=** (esleipena) **+** (batuketa) **–** (kenketa) eta ***** (biderkaketa) eragileak baino askoz ere gehiago daude. Banan-banan aztertu baino lehen, eragileen bi ezaugarri nagusi aztertuko dira: lehentasuna eta elkargarritasuna, hain zuzen.

3.2. ERAGILEEN EZAUGARRIAK

Espresio batean eragile bat baino gehiago dagoenean ebaluaketa-ordena eragileen lehentasunaren arabera gertatzen da. Lehentasun handieneko eragiketak burutzen dira lehen, lan-emaitzak lortuz. Geroago, eta lehentasunaren ordena beheranzkorrari jarraituz, ondorengo eragileak aplikatzen dira aurreko lan-emaitzen gainean. Adibidez, `2+3*4` espresioaren ebaluaketak 14 sortzen du, biderkaketak batuketak baino lehentasun-maila handiagoa duelako. Parentesiek, beste lengoaietan bezala, ebaluaketaren ordena aldatzen dute. Honela, `(2+3)*4` espresioaren ebaluaketak 20 balioa sortzen du.

Beste ezaugarriak, elkargarritasunak hain zuzen, ebaluaketa nola burutzen den esaten digu, ezkerretik eskuinera ala alderantziz. Adibidez, **=** eragileak eskuin-ezker elkargarritasuna duen bitartean, eragile aritmetiko arruntek (+, –, *) ezker-eskuinekoa dute.

`a = b = c` eskuinetik ebaluatzen da. Beraz, *b*-k lehenik eta *a*-k gero *c*--ren balioa hartuko dute.

`a + b – c` aldiz, ezkerretik eskuinera ebaluatuko da lehentasun-maila berean, lehenik `a + b` batuketa kalkulatuz eta, ondoren, lortutako emaitzari c kenduko zaiolarik. 3.1. taulan eragile garrantzitsuenak eta dagokien elkargarritasuna agertzen dira, lehentasunaren ordenan.

Eragile batek karaktere bat baino gehiago badu, ezin da inolako karaktererik sartu horien artean. 3.1 taulan ondoren azalduko diren eragileak eta dagozkien ezaugarriak agertzen dira.

ERAGIKETA-MOTA	ERAGILEA	ELKARGARRITASUNA
parentesiak eta makoak	`( ) [ ]`	→
egituraren eremua	`→ .`	→
zeinua	`+ –`	←
erakusleak	`* &`	←
cast	`( )`	←
ukapena	`~ !`	←
inkrementua, dekrementua	`++ --`	←
biderkaketa, zatiketa, modulua	`* / %`	→
batuketa, kenketa	`+ –`	→
desplazamendua	`<< >>`	→
baldintza matematikoak	`< <= >= >`	→
	`== !=`	→
logikoak (and, or, xor)	`& ^ |`	→
baldintzen konposaketa	`&& ||`	→
baldintzapeko egitura	`? :`	←
esleipena	`=`	←
	`+= -= *= /= %=`	←
	`&= ^= |= <<= >>=`	←

3.1. taula. Eragileak eta dagozkien lehentasuna eta elkargarritasuna

3.3. ESLEIPEN-ERAGILEA

Esleipena aldagai baten balioa aldatzeko erabiltzen da, eta C bezalako programazio-lengoaia agintzaileen funtsezko eragiketa da. Esleipen espresio baten forma orokorra honakoa da:

```
aldagai_izena=espresioa
```

Bertan azaltzen den espresioa konstante, aldagai eta eragileez osaturiko edozein konbinazio izan daiteke.

3.4. ERAGILE ARITMETIKOAK

3.2. taulan ikusten denez, ohizko eragiketez —batuketa, kenketa, biderkaketa eta zatiketa— aparte beste bi agertzen dira; inkrementua eta dekrementua hain zuzen. Bietan esleipena gertatzen da eta beste era batez adieraz badaitezke ere, erosotasun eta eraginkortasunari begira sartu dira C lengoaiaren barruan, batzuetan programak irakurtzeko zailak gertatu arren.

ERAGIKETA	ERAGILEA	FORMATUA	AZALPENA
ukapena	-	-x	x-en balioa zeinuz aldatuta
batuketa	+	x + y	x gehi y
kenketa	-	x - y	x ken y
biderkaketa	*	x * y	x bider y
zatiketa	/	x / y	x zati y
modulua	%	x % y	hondarra x/y egitean
inkrementua	++	x++ ++x	x = x + 1
dekrementua	--	x-- --x	x = x - 1

3.2. taula. Eragile aritmetikoak

`x=x+1`, `x++` eta `++x` espresioek ondorio berbera dute, `x=x-1`, `x-` eta `--x` espresioek duten bezala. Inkrementua eta dekrementua espresio konplexuago baten barruan jarriz gero, aldagaia eragilearekiko non jartzen den aztertu behar da, `++x` eta `x++` espresioek ondorio desberdina ekar baitezakete zenbait espresioren barruan. Nolanahi ere, `x`-en balioa inkrementatua izango bada ere, albo-ondorioak desberdinak izan daitezke. Inkrementu edo dekrementu eragileak eragigaiaren aurretik jarriz gero, C-k eragigaiaren balioa erabili

aurretik burutzen du eragiketa. Eragilea eragigaiaren ondoren badago, aldiz, C-k bere balioa erabiliko du inkrementatu edo dekrementatu aurretik.

Adibidez, demagun `x` aldagaiaren balioa 6 dela. `y=++x` agindua burututakoan `y`-k 7 balioa hartuko du. `y=x++` burututakoan, ordea, `y`-k 6 balioa hartuko du

Inkrementua eta dekrementua bezalako eragileak kontu handiz erabili beharko dira beste espresioen barruan, albo-ondorio nahigabeak gerta baitaitezke.

Konpiladore gehienek objektu-kode azkar eta eraginkorra lortzen dute inkrementu eta dekrementu eragileentzat, dagozkien esleipenarena baino hobea; baina hala eta guztiz ere, espresio sinpleetan baino ez da aholkatzen haiek erabiltzea, arazteko oso zailak gertatzen diren akatsak sortzea oso erraza baita.

Aipatzekoa da zatiketarako `/` eragile bakarra egotea, bai zatiketa osorako bai errealerako —PASCALez aldiz, `div` eta `/` eragileak bereiztu egiten dira. Eragigaiak guztiak osokoak badira, emaitza zatiketa osoarena izango da eta bestelakoetan, aldiz, erreala (ikus 3.1. programa).

```
#include <stdio.h>

main()
{
int x1, y;
float x2;

x1=10;
x2=10.0;
y=4;
printf("%d %d", x1/y, x2/y);
}


Emaitza: 2  2.5
```

3.1. programa. *Zatiketa osoa eta erreala*

Modulua, `%` eragilea, osoko eragigaiekin baino ezin da erabili eta emaitza osokoa izango da. Osokoekin `/` eta `%` eragileen erabilpenak argitzeko 3.2 programa proba daiteke.

```
#include <stdio.h>

main()
{
int x, y;

x=10;
y=3;
printf("%d %d", x/y, x%y);
}


Emaitza: 3  1
```

3.2. programa. / eta % eragileen erabilpena.

`/` eta `%` eragileen eragigaiak osokoak eta positiboak direnean, konpiladore guztietan emaitza berberak lortzen dira. Zatiketa egitean, eragigairen bat negatiboa bada, aldiz, emaitza biribildu edo moztua gerta daiteke eta moduluaren zeinua zein izango den konpiladore edota konputagailuaren arabera desberdina izan daiteke. Hau gerta ez dadin, liburutegi estandarrean azaltzen den `abs` funtzioa erabil daiteke eragigai guztien balio absolutua behartzeko.

3.5. BIT-MANEIURAKO ERAGILEAK

C lengoaiaren ezaugarrien artean bitak atzitzeko gaitasuna aipatu dugu eta horretarako 3.3. taulan azaldutako eragileak dauzkagu. Jakina denez AND eragiketa zenbait bit 0 egoeran jartzeko erabiltzen da, OR eragiketa 1 egoeran jartzeko eta XOR-a (bitak) egoeraz aldatzeko. Horretarako karaktere (8 bit), *short* (16 bit) edo *long* (32 bit) motako aldagai bat hamaseitarrez (*0x* aurrizkia) adierazitako maskara batekin parekatzen da. Mantendu nahi diren bitei *0* balioa egokitzen zaie maskaran eta batean jarri (OR) edo aldatu (XOR) nahi

direnei bat balioa. AND eragiketan alderantziz egin behar da, hau da, mantendu nahi diren bitei 1 balioa egokitzen zaie eta horretarako eragilea, osagarri bezala, erabil daiteke.

ERAGIKETA	ERAGILEA	FORMATUA	AZALPENA
AND	&	x & y	x AND y
OR	|	x | y	x OR y
XOR	^	x ^ y	x XOR y
ukapen logikoa	~	x ~ y	x-en bit guztiak aldatu
ezkerrerako desplazamendua	<<	x << y	x-en bitak y aldiz desplazatu ezkerrera
eskuinerako desplazamendua	>>	x >> y	x-en bitak y aldiz desplazatu eskuinera

3.3. taula. Bit-maneiurako eragileak.

Adibidez, **n** aldagaia 8 bitez osatuta egonda, (haien balioa jakin gabe) eta ezkerreko bita batean jarri, eskuinekoa zeroan jarri eta erdiko biak aldatu nahi baditugu ondokoa egin behar da:

```
                    /*   n   ←    xxxx xxxx    */
n = n | Øx8Ø        /*   n   ←    1xxx xxxx    */
n = n &(~ØxØ1)      /*   n   ←    1xxx xxxØ    */
n = n^ Øx18         /*   n   ←    1xxx̄ xx̄xØ    */
```

Eragile hauen erabilpena argitzeko, 3.4. taulan adibide batzuk azaltzen dira, marraren gainean dauden balioak eragigaiak eta behean daudenak emaitzak izanik.

```
   10101010      10101010      11110000
   01010101      01010101      10101010      01010101
&  ________   |  ________   ^  ________   ~  ________
   00000000      11111111      01011010      10101010
```

3.4. taula. Bit-maneiurako eragileen adibideak.

Desplazamenduek bit-maneiua egitura errepikakorretan erabiltzeko balio dute batez ere, 4. kapituluan ikusiko dugunez. Adibide bezala, ondokoak jar daitezke:

```
m = 0xf0    /* m<— 11110000 */
k = m >> 2  /* k<— 00111100 */
j = m << 2  /* j<— 11000000, bi bateko galdu egingo dira */
```

3.6. ERLAZIO-ERAGILEAK

Erlazio-eragileak baldintzak adierazteko erabiltzen dira batipat (ikus 3.5. taula). Dena den, C lengoaiaz edozein espresio baldintza izan daiteke; esleipena, espresio aritmetikoa, etab.

ERAGIKETA	ERAGILEA	FORMATUA	AZALPENA
handiago	>	a > b	baldin a > b EGIAZKO (1) bestela FALTSUA (Ø)
txikiago	>	a < b	baldin a < b 1 bestela Ø
berdin	==	a == b	baldin a == b 1 bestela Ø
handiago edo berdin	>=	a >= b	baldin a >= b 1 bestela Ø
txikiago edo berdin	<=	a <= b	baldin a <= b 1 bestela Ø
ezberdin	!=	a != b	baldin a != b 1 bestela Ø
eta	&&	e1 && e2	baldin e1 eta e2 1 bestela Ø
edo	||	e1 || e2	baldin e1 edo e2 1 bestela Ø
ez	!	! e1	baldin e1 Ø bestela 1

3.5. taula. Erlazio-eragileak

Erlaziozko ez den espresio bat baldintza denean, espresioaren ebaluaketa zero denean faltsutzat jotzen da, eta gainerako kasuetan, ordea, egiazkotzat. Hori dela eta, kontu handia eduki behar da berdintasuneko erlazioarekin; oso erraza baita == eragilea jarri beharrean = (esleipena) jartzea, beste lengoaietan egiten denaren legez. Ondorioz, egiaztatuko den baldintzaren emaitza esleitutako balioaren araberakoa izango da.

Adibide batzuk azter ditzagun:

```
int n = 0
if(n=0)   /* beti faltsua, esleipena baita */
if(n==0)  /* kasu honetan egiazkoa, n 0 baita */
a<b       /* egiazkoa a b baino txikiagoa bada */
a>b       /* egiazkoa a b baino handiagoa bada */
a<=b      /* egiazkoa a b baino txikiagoa edo berdin bada */
a>=b      /* egiazkoa a b baino handiagoa edo berdin bada */
a==b      /* egiazkoa a eta b berdinak badira */
a!=b      /* egiazkoa a eta b desberdinak badira */
```

`&&` (bi baldintzak batera) eta `||` (baldintzetako bat nahikoa) baldintza-konposaketarako eragileak interesgarriak dira. Ezkerretik eskuinera ebaluatzen dira modu honetan elkarturik dauden espresioak, eta ebaluaketa honek egiazko edo gezurrezko balioa itzuliko du. Konposatutako espresioetan asignaziorik ez jartzea gomendatzen da, konpiladorearen arabera gerta baitaiteke balioa itzultzea eragiketa guztiak burutu gabe.

Adibidez, `espr1 && espr2` espresioan, `espr1` gezurrezkoa bada, espresio guztiaren balioa gezurrezkoa izango da, eta `espr2` ez da, agian, ebaluatuko.

Aipatzekoa da karaktere-kateak ikusitako eragileekin ezin direla konbinatu, karaktere-kateak oinarrizko datuak ez direlako (ikus 6. kapitulua).

3.7. BESTE ERAGILEAK

Aurreko eragiketetako bat eta esleipena konbinatzen dute eragile konbinatuek, emaitza eta lehen eragigaia beti aldagai berbera direlarik. Horrela, `x=x+a` jarri beharrean `x+= a` jar daiteke, `x=x>>3` jarri ordez `x >>=3` etab.

Eragile konbinatuak ondokoak dira:

```
+=      -=      *=      /=
%=      &=      |=      ^=
<<=     >>=
```

`sizeof` eragigai bakarreko eragilea da, eta konpilazio-denboran eragigai bezala eman zaion aldagaia edo motaren byte-kopurua itzultzen du. Adibidez, osoko balioak `(int)` 4 bytekoak badira, `sizeof(int)` espresioak 4 balioa itzuliko du. Errealak `(float)` 4 bytekoak badira, `sizeof(float)` espresioak 4 itzuliko du. `erre` aldagaia `float` bezala erazagututa balego, `sizeof(erre)` espresioak ere 4 itzuliko luke. Aldagaien izenekin edo espresioekin, parentesia hautazkoa den artean, moten izenekin, parentesia derrigorrezkoa da. Hala ere, irakurgarritasunaren aldetik parentesiak erabiltzea komenigarria da. `sizeof` eragilearen erabilpen nagusia C-ren datu-moten tamainen menpe dagoen kode garraiagarria sortzea izango da.

`?` eragilea 4. kapituluan ikusiko dugun `if-else` erako egituraren ordezkoa da. Bere sintaxia ondokoa da: `esp1 ? esp2 : esp3`

Beraz, hiru argumentu ditu, hirurak espresioak. `?` eragileak `esp1` baldintza-espresioa ebaluatzen du. Egiazkoa bada, `esp2` ebaluatu eta honen balioa itzuliko du, bestela, `esp3` ebaluatu eta bere balioa itzuliko du.

Eragile bezala, komak hainbat espresio kateatzen du. Koma eragilearen ezkerraldea dagokion mota galtzen da *void* motaz ordezkaturik. Ondorioz, komaren eskuinaldea komaz bereizitako espresioaren balio bilakatzen da.

Koma eragilea ez da komenigarria, irakurgarritasuna galtzen da eta. Normalki, koma eragilea 4. kapituluan ikusiko den `for` egituraren hasieraketan (lehen espresioa) eta inkrementuan (azken espresioan) bakarrik erabiltzen da bakoitzean esleipen anitz egiteko. Adibidez,

```
for (j=0, k=100; k-j>0; j++, k--);
```

aginduan bi hasieraketa egiten da. `j` eta `k` aldagaietan, eta bigizta guztietan beste bi, `j`-ren inkrementua eta `k`-ren dekrementua.

ERAGIKETA	ERAGILEA	FORMATUA	AZALPENA
erreferentzia	`&`	`& x`	x-en helbidea
erakuslea	`*`	`* p`	p-k helbideratzen duen datua
taula	`[ ]`	`x [i]`	taularen i. elementua
egituraren eremua	`.`	`x . y`	x egituraren y eremua
	`->`	`p -> a`	p-k helbideratzen duen egituraren a eremua

3.6. taula. Bestelako eragileak

3.8. BATERAGARRITASUNA ETA BIHURKETAK

Pascal bezalako lengoaietan ez bezala, C-n idatzitako programetan eragigaiek, motari dagokionean, ez dute bateragarriak izan behar.

Datu-bihurketa inplizituak gertatzen dira honako kasu hauetan:

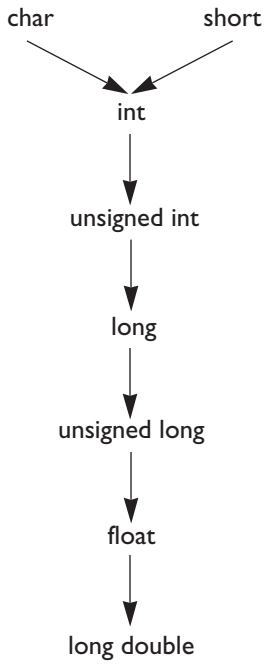
1. Esleipenetan, eskuinaldeko balioa ezkerraldeko aldagaiaren motara bihurtuko da mota berekoak ez badira.

2. `char` edo `short int` motako zerbait azaltzen bada espresio batean, `int` motara bihurtzen da. Era berean, `unsigned char` edo `unsigned short` motakoak `int`-era bihurtzen dira, `int` motak beraien balioak errepresenta baditzake; bestela, `unsigned int`-erako bihurtzen dira.

3. Espresio aritmetikoetan, eragigaiek zenbait eragilek eskatzen duten motak har ditzaten.

4. Kasu batzuetan, funtzioen argumentuak ere bihurtzen dira.

Lehen bi puntuetan aipatutako bihurketek 3.1 irudiko eskemari jarraitzen diote.



3.1. irudia. *Bihurketa inplizituak.*

Hau dela eta, karaktere bat maiuskula bihurtzeko (minuskulaz dagoela suposatuz) 3.3. programaren bidez egin daiteke.

```
#include <stdio.h>

main ( )
{
char kar_min, kar_maj;

printf ("sakatu karaktere bat minuskulaz \n");
scanf ("%c", &kar_min);
kar_maj = kar_min - 'a' + 'A';
printf ("%c", kar_maj);
}
```

3.3. programa. *Minuskula maiuskula bihurtzea.*

Programan ikusten denez, karaktereen batuketa eta kenketa gertatzen da, baina benetan exekutatuko dena ondokoa da: karaktereak osoko zenbaki bihurtu, osokoen arteko batuketa eta kenketa, eta azkenik emaitza karaktere bihurtu.

Dena den, komenigarria da bihurketak esplizitatzea, erroreak saihestu eta irakurgarritasuna irabazteko; gainera, C konpiladore batzuetan, esplizituki egiten ez bada abisu-mezuak sortzen direla kontuan hartu behar da.

Bihurketa esplizitua adierazteko *cast* —izenburua— izeneko elementua erabiltzen da; parentesi artean datu-motaren gako-hitza zehazten duena. Adibidez, karaktere bat zenbaki oso bihurtzeko `(int)` izenburua jarriko da karaktere-aldagaiaren erreferentziaren aurrean.

Adibidez, 3.3. programan maiuskulazko karakterea izateko behar diren bihurketak esplizituki jarriz gero, honako kodea agertuko litzateke:

```
kar_maj = (char) ((int)kar_min - (int)'a' + (int)'A');
```

Edozein motatako bihurketan kontutan eduki behar dira beste puntu hauek:

1. `int`-etik `float`-erako edo `float`-etik `double`-rako bihurketek ez dute ez zehaztasuna ezta doitasuna handitzen. Bihurketa hauek adierazpidea baino ez dute aldatzen.

2. C konpiladore batzuek `char` aldagaiak beti positibotzat hartzen dituzten bitartean, beste batzuek 127 baino balio handiagoa duten karaktereak negatibotzat hartzen dituzte. Ohizkoa da `char` motako aldagaiak soilik karaktereentzat erabiltzea. Bestela, `int`, `short int` edo `signed char` motak erabiliko dira, garraiagarritasun-problemak saihesteko asmoz.

3.7 taulan informazio-galera izan dezaketen moten bihurketak azaltzen dira. Bertan agertzen ez diren bihurketak egin nahi izanez gero, pausoka egin daitezke. Honela, `double`-tik `int`-era bihurketa egitea nahi bada, lehenengo `double`-tik `float`-era eta, ondoren, `float`-etik `int`-era bihurketak burutuko dira. Taulan azaltzen diren bihurketen alderantzizko bihurketak ere burutu daitezke.

XEDE-MOTA	ESPRESIO-MOTA	INFORMAZIO-GALERA POSIBLEA
`signed char`	`char`	Balioa>127 bada, xedea negatiboa da.
`char`	`short int`	8 bit adierazgarrienak
`char`	`int`	8 edo 24 bit adierazgarrienak
`char`	`long int`	24 bit adierazgarrienak
`short int`	`int`	0 edo 16 bit adierazgarrienak
`short int`	`long int`	16 bit adierazgarrienak
`int`	`long int`	0 edo 16 bit adierazgarrienak
`int`	`float`	frakziozko zatia eta beharbada gehiago
`float`	`double`	doitasuna, emaitza biribildua
`double`	`long double`	doitasuna, emaitza biribildua

3.7. Taula. Bihurketa arrunt batzuk.

3.9. BESTELAKOAK

Programak hobeto irakurri ahal izateko, zuriguneak eta tabulazioak gehi daitezke C-ren espresioetan, beti kontuan hartuz ezin direla karaktere anitzeko eragileen artean zein identifikatzaileen barruan jarri.

Parentesiak erabiltzea komenigarria da, ebaluaketaren ordena argitzeko, bai norberari, baita programa irakurriko duen edonori ere. Erredundanteak izan daitezkeen parentesiak erabiltzeak ez du errore-abisurik sortzen ezta espresio baten egikaritzapenaren abiadura moteltzen ere.

Edozein espresio sententzia bihurtzen da baldin eta ondoan **;** karaktere bat (puntu eta koma karakterea) badute, C lengoaian **;** karaktereak sententziaren amaiera adierazteko balio baitu.

Sententzi multzo bat giltzen artean —**{** eta **}** karaktereen artean— bil daiteke, sententzia konposatu edo bloke bat lortuz. Sintaxiaren aldetik, bloke bat sententzia bakun baten baliokidea da. Sententzia konposatuen adibideak orain arte ikusitako programetan aurkitzen dira, edozein funtzioren gorputza horrelakoa baita. Bloke-amaierako giltzaren ondoren ez da beharrezkoa **;** karakterea jartzea.

Edozein blokeren barruan aldagaiak erazagutu daitezke. Aldagai bat bloke batean erazagutzean, bloke horretako aldagai lokala izango da —hau da, bloke horretatik kanpo ezin izango da erabili. Ohizkoa izaten da aldagai lokalak pilan gordetzea baina honetaz 8. kapituluan sakonduko dugu.

4. KONTROL-EGITURAK

Kontrol-egituren bidez, exekutatuko diren espresioak hautatzeko edo errepikatzeko aukera dago. Aurreko kapituluan azterturiko espresioak erabiliz programa guztiz linealak eta sekuentzialak idatz daitezke; baina programa gehienetan datuen arabera zenbait agindu exekutatu ala ez erabaki behar da baldintzazko egituren bidez, eta agindu-multzoen egikaritzapena errepikatu egitura errepikakorrak erabiliz. Egitura hauek C lengoaiaz nola idatzi eta erabili da kapitulu honen helburua.

Baldintzazko egituren artean bi dira erabilienak eta C lengoaiak bi hauek dauzka: *baldin* egitura `(if)` eta *aukera* `(switch)`. Lehenengoan espresio logiko baten balioaren arabera bi adarren artean aukeratzen da. Bigarrenean, aldiz, aukeran nahi adina adar jar daitezke.

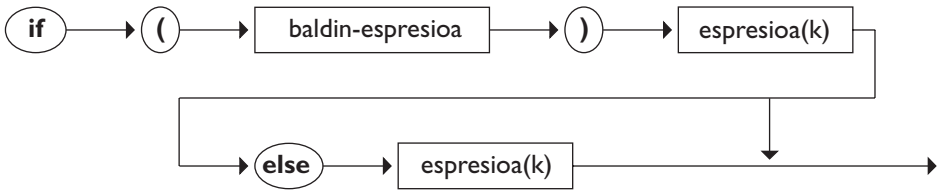
Sententzia bat edo sekuentzia-multzo bat hainbat alditan errepikatzea nahi bada, egitura errepikakorrak erabiliko dira. Errepikapenen kopurua aurretik jakina edo ez-jakina izan daiteke. Jakina bada, errepikapenen kopurua zenbatuko duen kontrol-aldagai bat erabiliko da, soilik errepikapen horiek egikari daitezen. Ez-jakina bada, baldintza baten menpe egotea edo bigizta infinitua izatea gerta daiteke. Egitura errepikakorretan C oso aberatsa da; hiru motakoak baitauzka: *bitartean* `(while)`, *errepika* `(do-while)` eta *aldatuz* `(for)`. Beren ezaugarriak ere aztertuko ditugu. Egitura hauekin salbuespen-kasuetarako erabiltzen diren `break` eta `continue` sententziak ere azaltzen dira. Aurkezten den azken sententzia `goto` sententzia da, gaur egungo programazio-metodologiek baztertzen dutena.

Azkenik, egitura hauen bidez egindako programak azaltzen dira, ezaugarriak ezagutzeaz gain egitura hauei dagozkien aplikazioak ere erakusteko asmoz.

4.1. *IF* EGITURA

if —ingeleseko baldin hitza— egitura, baldintzapeko kasuetan sententzi multzo bat egikaritzeko balio du. Egitura edo sententzia hau edozein lengoaiatan dago, eta C-n ezaugarri orokorrak mantendu egiten ditu. Baldintza espresio logiko baten bidez adierazten da, eta baldintza betetzen bada (hau da, zero balioa ez badagokio), segidan duen multzoa egikarituko da. Baldintza bete ezean, *else* adarra baldin badago, adarra honi dagokion sententzi multzoa egikarituko da. Bestela, *else* adarra ez badago alegia, ez da egitura honetako ezer egikarituko.

Egiturari dagokion sintaxia:



***if* egituraren sintaxia.**

Ikus daitekeenez *bestela* adarra aukerakoa da, bere agerpenaren arabera egitura osoa edo sinplea bereiziz. Adibidez, ondoko programa zatian agertzen den *if* egitura sinplea da.

```
int n;
...
if (n < 0)
     n = (–n); /* balio absolutua*/
...
```

Aldiz, honako zati honetan egitura osoa dugu:

```
int n, em;
...
if (n < 0)
   printf ("errorea: negatiboa");
else
   {
   em = n % 5;
   printf ("%d modulu 5 = %d", n, em);
   }
...
```

Ondoko ñabardura hauek hartu behar dira kontuan:

- Baldintzazko espresioan berdintasuna adierazteko == eragilea erabili behar da eta ez =, zeren kasu honetan errorerik ez bada sortzen ere, esleipen bat gertatzeaz gain, esleitzen den balioa zeroa bada, baldintzaren emaitza faltsutzat hartuko da eta gainerakoetan egiazkotzat.

- `(j!=0)` eta `(j)` baldintzak guztiz baliokideak dira, `(j==0)` eta `(!j)` diren bezala.

- Baldintzazko espresioa `&&`, `||` eta `!` eragileez osaturiko baldintza konposatua izan daiteke.

- *if* baten barruan beste *if* bat dagoenean —*if* kabiatua— arazoa dago bestela adarrekin, zeren *else* bat aurkitutakoan lehenengo edo bigarrenari ote dagokion zalantza egon baitaiteke. Honelakoetan dagoen araua honakoa da: *else* bat idatzitako azken *if*-ari dagokio beti, marjinak kontutan hartu gabe. *else* adar bat azkena ez den *if* bati lotzea nahi bada, giltzak erabiliko dira joera normala saihesteko asmoz. Adibidez:

```
if (x)
   {
   if (y)
     printf("Lehena");
   }
else
   printf("Bigarrena");
```

- Kasu batzuetan, *if* egitura `baldintza? espresio1: espresio2;` sententziaren bidez ordezka daiteke 4.1. programan ikus daitekeenez. Joera hau *if* egituraren baldintzapeko sententziak bakunak direnean ematen da batez ere.

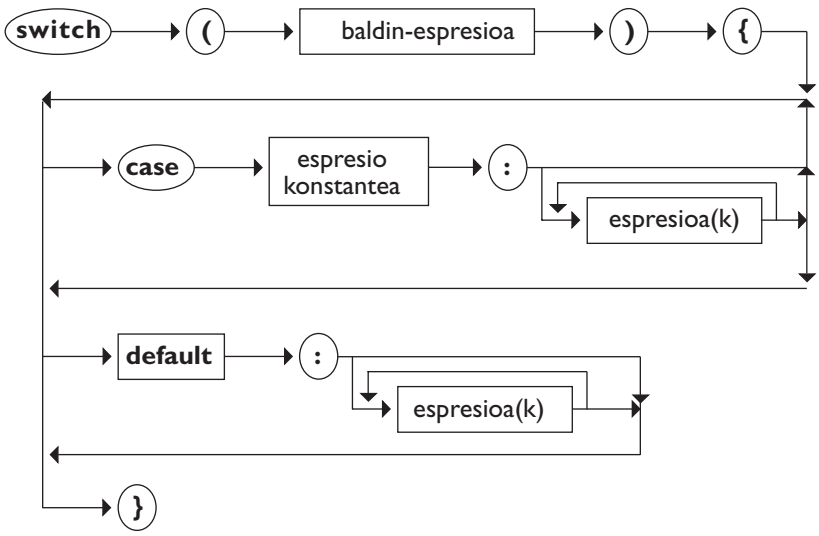
```
#include <stdio.h>

main ( ) /* bi zenbakien arteko handiena */
{
int a, b, hand;

printf ("sakatu bi zenbaki \n");
scanf ("%d", &a);
scanf ("%d", &b);
if (a > b)
  hand = a;
else
  hand = b;

/* honela ere egin daiteke :
  hand = (a > b) ? a : b; */

printf ("%d eta %d artean handiena: %d", a, b, hand);
}
```

*4.1. **programa**. Baldintzazko egituraren adibideak.*

4.2. *SWITCH* EGITURA

Egitura honen bidez espresio osoko[1] baten balioaren arabera nahi dugun adina adar sor daiteke, adar bakoitza *case* gako-hitz batez hasten delarik (azkenekoa izan ezik; hau *default* gako-hitzaz has baitaiteke). Egitura honi dagokion sintaxia ondoko irudian azaltzen da:

[1] ANSI Cn edozein osoko izan daitekeen bitartean —int, short, long etab.— K&R-koan int motakoa izan behar zuen



***Switch* egituraren sintaxia.**

Ikus daitekeenez, adarretan espresioak aukerakoak dira, *default* adarra den bezala. Adar batean sententzia bat baino gehiago jartzen bada, ez da *{* eta *}* jarri behar, baina *;* karaktereaz bukatu behar du bakoitzak.

Exekuzioan *switch* bat aurkitzen denean, baldin-espresioa ebaluatzen da aurretik, eta balioaren arabera bilatzen da adarretako zein espresio konstantek balio berbera duen. Horrelako adarrik aurkitzen bada, adar horri eta gainerako hurrengoei dagozkien sententziak burutuko dira. Espresioak duen balioa adarretan aurkitzen ez bada, *default* adarraren espresioak burutuko dira, adar hau idatzi bada noski.

Beste lengoaietako antzerako egiturekin antza badu ere, ezberdina da aukeratutako adarrari eta ondokoei dagozkien sententzia guztiak exekutatzen baitira, kontrakoa esaten ez bada behintzat. Adibidez, Pascal-eko *case* egituran, adarrari dagozkion sententziak baino ez dira burutzen. Dena den, C-z horrelakorik nahi bada, adar bakoitzeko azken sententzia aurrerago sakonduko den *break* sententzia izango da. *default* adarraren azken sententzia *break* izatea derrigorrezkoa ez den arren, jartzea komenigarria da.

switch sententzia erabiltzerakoan honako ideiak eduki behar dira kontuan:

1. Baldin-sententziaren balioa eta adarretakoen artean berdintasuna baino ezin da egiaztatu, *if* sententziak erlaziozko edo logikoak diren espresioak ebalua ditzakeen artean.

2. Karaktere motako konstanteak erabiltzen baldin badira aukeretan, automatikoki osoko balio bihurtuko dira.

3. Balio berbera duten espresio konstante bi ezin dira *switch* baten barruan egon. Gerta daitekeena, aldiz, *switch* sententzia baten barnean kabiatuta dagoen beste *switch* sententzia baten barnean espresio konstante bera egotea da.

```
#include <stdio.h>

main ( ) /* eragilea sakatuta eragiketa burutu */
{
char c;
int a, b;

a = 10;
b = 5;
printf ("sakatu +, -, * edo / :\n");
scanf ("%c", &c);

switch (c)
   {
   case '+': printf ("%d\n", a+b);
             break;
   case '-': printf ("%d\n", a-b);
             break;
   case '*': printf ("%d\n", a*b);
             break;
   case '/': printf ("%d\n", a/b);
             break;
   default:  printf ("gaizki sakatu dugu\n");
             break;
   }
}
```

4.2. programa. *Switch eta break konbinatuak.*

4.2. programan *switch*-aren erabilpen konbentzionala (*break* erabiltzen duena) azaltzen da. 4.3.ean aldiz, erabilpen bereziagoa.

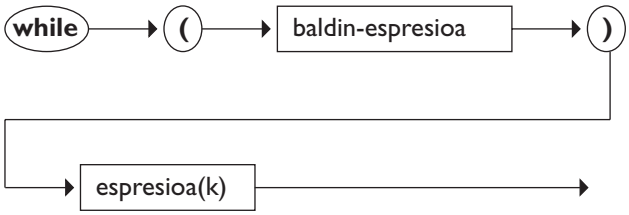
```
#include <stdio.h>

main ( ) /* karakterea bokala den ala ez esan */
{
char c;

printf("Karaktere bat sakatu, ea bokala den asmatzen dudan: \n");
scanf ("%c", &c);
switch (c)
   {
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u':
   case 'A':
   case 'E':
   case 'I':
   case 'O':
   case 'U': printf ("bokala da \n");
             break;
   default:  printf ("ez da bokala \n");
             break;
   }
}
```

4.3. programa. *Switch-aren erabilpen ez-konbentzionala.*

4.3. programan ikus daitekeenez, *case* adarrak mihiztadura-lengoaietako etiketa gisa ibiltzen dira, beraz, ondoan dagoen oro korrituko da, sekuentzia eten bat (*break* adibidez) aurkitu arte.

4.3. *WHILE* EGITURA

Egitura errepikakor orokorrena da Pascal-eko *while-do* sententzia bezalakoa. Sintaxiaren aldetik ondoko egitura du:



while egituraren sintaxia.

Baldin-espresioa —espresio logiko bezala— ebaluatu ondoren, egiazkoa (ez zero) bada egiturari dagozkion sententzia edo sententzia-multzoa exekutatuko da, eta baldin-espresioa ebaluatzera itzuliko da, bigizta osatuz.

Baldin-espresioaren ebaluazioa faltsua denean, *while*-aren ondoko sententziara pasatuko da, egituraren gorputza exekutatu gabe.

Parentesien arteko espresioa da errepikatzeko baldintza eta sententzia edo sententzi multzoa errepikatzen den gorputza. Sententzia hutsa ere azal daiteke gorputz bezala.

4.4. programan faktoriala kalkulatzen duen adibide bat ikus daiteke. Bertan, baldintza `(i<=n)` da, eta makoen arteko bi sententziek osatzen dute gorputza.

```
#include <stdio.h>

main ( ) /* n faktoriala */
{
int n, i;
long fakt;

printf("Saka ezazu zenbaki bat, eta bere faktoriala esango\
      dizut:\n");
scanf ("%d", &n);
if (n < 0)
  printf ("errorea: zenbakia < 0 \n");
```

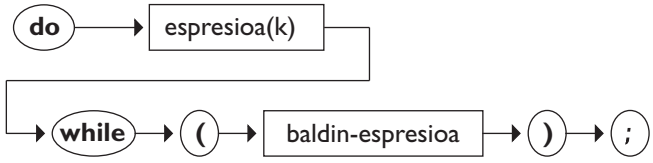
```
else
   {
   fakt = 1;
   i = 2;
   while (i <= n)
      {
      fakt = fakt * i;
      i++;
      }
   printf ("%d faktoriala = %ld \n", n, fakt);
   }
}
```

4.4. programa. *n faktoriala while-ren bidez.*

4.4. *DO-WHILE* EGITURA

While egituraren aldaketa honetan, baldintzaren ebaluazioa bigiztaren bukaeran gertatzen da. Beraz, *do-while* egitura honetan gorputza gutxienez behin exekutatuko da. Aldiz, *while* egituran behin ere ez burutzea gerta daiteke baldintzaren lehen ebaluazioa faltsua baldin bada. Gainerakoan berdin-berdinak dira.

Egituraren sintaxia honako hau da:



do-while egituraren sintaxia.

Sententzia honen ohizko erabilera menuaren bidezko errutinekin egiten dena da. Menu baten aukerak aurkeztu ondoren, horietako bat aukeratu arte errepikatuko da bigizta, hautapen okerren aurrean berriro aukeratzeko utziz.

4.5. programan faktoriala kalkulatzeko programan *do-while* erabiltzeko egindako aldaketak ikus daitezke. *if* berria sartzen da 0 eta 1-en faktorialak ondo buru daitezen.

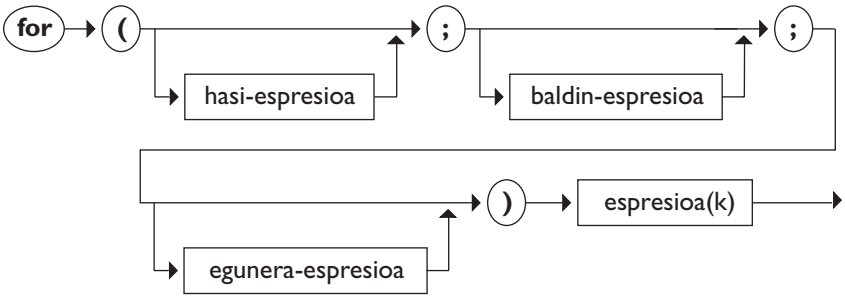
```
#include <stdio.h>

main ( ) /* n faktoriala */
{
int n, i;
long fakt;

printf("Saka ezazu zenbaki bat, eta bere faktoriala esango\
     dizut:\n");
scanf ("%d", &n);
if (n < 0)
  printf ("errorea: zenbakia < 0 \n");
else
  {
  fakt = 1;
  if (n > 1)
    {
    i = 2;
    do { fakt = fakt * i;
      i ++;
    } while (i <= n );
  }
  printf ("%d faktoriala = %ld \n", n, fakt);
  }
}
```

4.5. programa. *n faktoriala do-whileren bidez.*

4.5. *FOR* EGITURA

Egitura errepikakor konplexu honetan, bigiztaren gorputza eta errepikatzeko baldintzaren gain hasieraketarako eta ziklo bakoitzeko eguneratzeko espresioak onartzen dira, kasu batzuetarako oso egitura erosoa gertatuz.

Hona hemen dagokion sintaxia:



for sententziaren sintaxia.

Egitura honen funtzionamendua ondokoa da:

1. hasi-espresioaren exekuzioa (hasieraketa).

2. baldin-espresioaren ebaluaketa. Faltsua bada *for*-aren ondoko sententziara pasatu (amaiera).

3. baldin-espresioa egiazkoa bada, gorputzaren sententzi multzoa exekutatuko da.

4. egunera-espresioaren exekuzioa (eguneratzea), eta 2. puntura itzuli.

Hasieraketa normalki esleipen-sententzia bat da eta bertan bigizta kontrolatuko duen aldagaia hasieratuko da. Normalean, eguneratze-espresioak kontrol-aldagai hori urrats bakoitzean gaurkotuko du. Lehenengo aldian baldin-espresioa faltsua bada, bigizta ez da sekula exekutatuko, ***while*** egituran bezala.

Beraz, parentesien arteko lehen espresioa (hasieraketa) behin exekutatzen da, bigarrena ziklo edo urrats bakoitzaren hasieran, eta hirugarrena (eguneratzea), ziklo bakoitzaren bukaeran. Dena den *for* egituraz egiten dena *while* eta esleipenaren bidez egin daiteke ondoan ikus daitekeenez.

```
for (e1;e2;e3)                    e1;
   {                              while(e2)
   esp1;                             {
   esp2;                             esp1;
   }                                 esp2;
                                     e3;
                                     }
```

4.6. programan faktorialaren kalkulua ikus daiteke *for* egitura erabiliz.

```
#include <stdio.h>

main () /* n-faktoriala */
{
int n,i;
long fakt;

printf("Saka ezazu zenbaki bat, eta bere faktoriala esango\
     dizut:\n");
scanf ("%d", &n);

if (n<0)
   printf ("errorea: zenbakia < 0 \n");
else
   {
   fakt = 1;
   for (i = 2; i <= n; i++)
      fakt = fakt * i;
   printf ("%d faktoriala = %ld\n", n, fakt);
   }
}
```

4.6. programa. *Faktoriala for erabiliz.*

Egituraren espresioak aukerazkoak badira ere, puntu eta koma sinboloek beti agertu behar dute espresioen funtzioa posizioaren araberakoa da eta. Baldin-espresioa agertuko ez balitz, bigizta infinitua gertatuko litzateke. Lehen eta hirugarren espresioak agertzen ez direnean *for* eta *while* egiturak guztiz baliokideak dira.

Gorputzaren sententzia bakarra, hutsa edo konposatua izan daiteke. Sententzia hutsa bigiztaren gorputzean, atzerapenak sortzeko erabiltzen da. Adibidez, ondoko sententziarekin zenbait denbora ezer egin gabe gelditu ko da programa:

```
for (i=0; i<10000; i++);
```

for egiturako kontrol-sententziak —hasierakoan eta eguneratzekoan erabili ohi da— konposatuak izan daitezke, horretarako koma eragilea erabiltzen delarik. Adibidez:

```
for (x=0, y=10; x<y; x++, y--)
```

Kasu horretan, lehenik bi aldagaiak hasieratzen dira. Ondoren, baldintza egiazkoa dela egiaztatu eta, gorputza egikaritu ondoren, `x`-en balioa inkrementatu eta `y`-rena dekrementatzen dira, `x`-en balioa –`y`-rena baino txikiago den bitartean.

Baldintza-espresioan baldintza konposatu bat jartzea badago, beste egituretan bezala, horretarako `&&` edo `||` eragileak erabiliz. 4.7. programan ikus daitekeenez, *for*-aren gorputza gehienez 10 alditan errepikatuko da, lehenago ezkutuko karakterea, `N`, asmatzen ez bada.

```
#include <stdio.h>

main()
{
char kar;
int konta;

printf("Ea ezkutuko karakterea asmatzen duzun.\n");
scanf("%c", &kar);
for (konta=0; konta<10 && kar!='N'; konta++)
  scanf("%c", &kar);
if (konta==10)
  printf("Ez duzu karakterea asmatu!\n");
else
  printf("Zorionak!!! Asmatu duzu!!!\n");
}
```

4.7. programa. *Baldin-espresioa konposatua denean.*

Hasieraketa *for* sententziatik kanpo egitea gertatzen den gehienetan, kontrol-aldagaiaren balioa lehenago esleitu delako izaten da. 4.8. programan honen adibide bat ikus daiteke, bertan ***kon*** aldagaiaren hasieraketa eta eguneratzea kontrol-sententzietatik kanpo gertatzen delarik.

```
#include <stdio.h>

/* Zenbakiak batu Ø eta 9 artean dauden bitartean */
main()
{
int kon, batura=0;

printf("Batu nahi diren zenbakiak sakatu\n");
scanf("%d", &kon);
for (; kon>=0 && kon<=9;)
   {
   batura+=kon;
   scanf("%d", &kon);
   }
printf("%d", batura);
}
```

4.8 programa. *Hasieraketarik gabeko for-a.*

Bigizta infinituak idazteko era normalena `for(;;)` idaztea bada ere, `while(1)` egitura ere erabil daiteke. Dena den, honek ez du ziurtatzen amaigabeko bigiztak izatea, bere gorputzaren barruan ***break*** sententzia azal baitaiteke.

4.6. BREAK SENTENTZIA

break sententziak bigiztaren bat-bateko amaiera eragiten du, bigiztaren ondorengo sententziara joanez. 4.9. programan B letra sakatu arte bukatuko ez den bigizta ikus daiteke.

```
#include <stdio.h>

main()
{
char k='\0';

printf("Ihes-karakterea sakatu arte ez da ondoko bigizta\
     bukatuko.\n");
for (;;)
   {
   scanf("%c", &k);
   if (k=='B')
     break;
   }
printf("B letra idatzi da\n");
}
```

***4.9. programa**. Bigizta infinitu batean break sententziaren erabilpena.*

Sententzia honen erabilpenak bi dira:

1. *switch* sententzia bateko *case* baten egikaritzapena amaitzeko eta ondorengo adarrak ez exekutatzeko, lehenago ikusi den bezala.

2. Bigizta bateko egikaritzapena momentuan eteteko, bigiztaren baldin-espresioa ebaluatu gabe. Normalean salbuespenen tratamendurako erabiltzen da, salbuespen guztiak baldin-espresioan kontuan hartzea konplexuegia izango bailitzateke. Bigizta batean *break* bat aurkitzean, bigizta berehala bukatzen da eta bigiztaren ondoko sententziara jauziko da.

4.10. programan ikus daiteke honen adibide bat. Bertan, 80 karaktere irakurtzen dira eta zenbat maiuskula dagoen kontatzen da. Dena den, puntua (. karakterea) irakurtzen bada, ez da karaktere gehiagorik irakurri behar.

```
#include <stdio.h>

main () /* maiuskulak kontrolatu */
{
int i, kont = 0;
char c;

printf ("sakatu 80 karaktere =>\n");
for (i = 0; i < 80; i++)
   {
   scanf ("%c", &c);
   if (c >= 'A' && c <= 'Z')
      kont ++;
   if (c =='.')
      break;
   }
printf ("%d maiuskula kontatu ditut\n", kont);
}
```

4.10. programa. *Break sententziaren erabilpena.*

Egitura errepikakorrak kabiatuak egon daitezkeenez, hau da, baten gorputzean beste egitura errepikakor bat egon daitekeenez, *break* aginduak barrurago dagoen bigiztaren amaiera eragiten du. Era berean, egitura errepikakor baten barruan *break* sententzia duen *switch* egitura erabiltzen bada, *break*-k *switch* horrengan bakarrik du eragina.

4.7. CONTINUE SENTENTZIA

Bigiztaren amaiera behartu ordez, bigiztaren errepikapen edo urrats berri bat eragiten du, oraingo urratsari dagokion gainerako kodea exekutatu gabe. Egitura errepikakorretako gorputzean erabili ohi da kontuan hartu behar ez diren iterazioak saihesteko. *while* eta *do-while* bigiztetan, *continue* sententziak behartzen du kontrola baldintzaren ebaluaketara pasatzera eta bigiztaren prozesuarekin jarraitzera. *for* bigiztetan, aldiz, lehenengoz eguneratzea gertatuko da eta gero baldintza ebaluatuko.

4.11. programan 20 zenbaki irakurtzen dira eta beren arteko biderkadura kalkulatu behar da, baina zenbakia zeroa bada zenbakia ez da kontutan hartuko.

```
#include <stdio.h>

main () /* biderkaketa */
{
int i = 0, zenb;
float em = 1.;

printf ("sakatu  0 ez diren 20 zenbaki\n");
while (i < 20)
   {
   scanf ("%d", &zenb);
   if (zenb ==0)
     continue;
   em = em * zenb;
   i++;
   }
printf("Zeroak izan ezik, esandako zenbakien biderkadura %lf\
     da.\n", em);
}
```

***4.11. programa**. continue sententzia erabiltzen duen adibidea.*

4.8. GOTO SENTENTZIA ETA ETIKETAK

goto sententziari buruz ondokoa esan behar da: programazio-metodologiaren ikuspuntutik bere erabilera debekatzea da egin daitekeen onena, programen irakurgarritasuna eta zuzentasuna bultzatzearren. Sententzia hau, izan ere, ez da beharrezkoa, bere eragina beste kontrol-egituren bidez fluxu-kontrola osoa lor baitaiteke. Hala ere, *goto* sententzia komenigarria izan daiteke salbuespeneko egoera batzuetan, atal honetatik kanpo erabiliko ez dugun arren.

goto sententziak etiketa bat behar du eragigai bezala. Etiketa bat C-k onartzen duen sinbolo bat da eta aurretik edo ondoren definitu behar da kode-puntu bat izendatuz, horretarako etiketari: karakterea (bi puntu karakterea) eransten zaiola.

goto sententziaren sintaxia honoko hau da: `goto etiketa;`

Aurreko sententziaren ondorioz programaren exekuzio sekuentziala eten egiten da eta exekutatzen den ondoko sententzia etiketa ondorengo lehen sententzia izango da. Erabilpen zilegi bat 4.12. programa-zati baliokideetan ikus daiteke:

```
egina=0;
for (i=0;i<500;i++)
{
   for (j=0;j<20;j++)
      {
      while (x[i]*y[j]>0 && x[i]%2!=0)
         {
         if (x[i]*y[j]>1000)
            {
            egina=1;
            break;
            }
         batura+=x[i]*y[j];
         x[i]--;
         }
      if (egina)
         break;
      }
   if (egina)
      break;
}
printf("1000 baino handiagoa den balioa!!!");
```

4.12.a) goto sententzia erabili gabe salbuespenen tratamendua

```
for (i=0;i<500;i++)
   {
   for (j=0;j<20;j++)
      {
      while (x[i]*y[j]>0 && x[i]%2!=0)
         {
         if (x[i]*y[j]>1000)
            goto gelditu;
         batura+=x[i]*y[j];
         x[i]--;
         }
      }
   }
gelditu:
      printf("1000 baino handiagoa den balioa!!!");
```

4.12 b) goto sententzia salbuespenak tratatzeko

goto sententzia salbuespen gisa baino ez da erabili behar. Baina kodea bestelakoetan ere irakurgaitza (kontrol-aldagai askoren agerpena dela eta) edo exekuzio-abiadura kritikoa bada, orduan *goto*-aren erabilpena onar daiteke.

4.9. ADIBIDEAK

1. enuntziatua:

Irakurri hiru zenbaki oso eta handiena idatzi (soluzioa 4.13. programan).

```
#include <stdio.h>

main () /* 3 zenbakiren artean handiena */
{
int a, b, c;

printf("Sakatu hiru zenbaki:\n");
scanf ("%d %d %d", &a, &b, &c);
if (a > b)
   if (a > c)
     printf ("handiena: %d \n", a);
   else
     printf ("handiena: %d \n", c);
else
   if (b > c)
     printf ("handiena: %d \n", b);
   else
     printf ("handiena: %d \n", c);
}
```

4.13. programa. *If kabiatuak.*

Adibide honetan *if* baten adarretan *if* egiturak agertzen dira (if kabiatuak) eta egitura bakoitza adierazpen bakuntzat jotzen denez ez da giltzarik behar.

2. enuntziatua:

Irakurri karaktere bat eta kalkulatu zero egoeran zenbat bit dagoen bere adierazpidean. (soluzioa 4.14. programa).

```
#include <stdio.h>

main ()
{
char datu, maskara = 0x01;
int i, kont = 0;

printf("Sakatu karaktere bat:\n");
scanf ("%c", &datu);
for (i = 0; i < 8; i++)
   {
   if ((datu & maskara) ==0) kont++;
   maskara = maskara << 1;
   }
printf ("%c karakterean, %d bit 0 egoeran daude\n", datu,
      kont);
}
```

4.14. programa. *Bit-maneiuaren adibidea.*

Bit-maneiua oso zaila gertatzen da goi-mailako lengoaietan, baina C-n aldiz, izugarri erraza da & (and), | (or), ^ (xor), << (ezker-desplazamendua) eta >> (eskuin-desplazamendua) eragileei esker. Adibidean, *and* eragiketaren bidez bit bat aztertzen da, zeren *and* eragiketan bit bakar bat 1 egoeran duen maskara batekin datu bat parekatzen dugunean, 1ari dagokion datuaren bita zeroa bada emaitza zero izango baita, eta bata bada, emaitza desberdin zero. Eragiketa 8 aldiz errepikatzen den bigizta batean dago (*for* egituraz) baina bit desberdinak aztertzeko, bigiztaren barruan maskarako bitak desplaza daitezke (programan egin den legez), edo datu bera desplazatu.

3. enuntziatua:

Irakurri 40 karaktere eta kontatu zenbat bit duten zero egoeran.

Adibide honen aurrean 2 galdera sortzen dira:

– Karaktereak batera irakur al daitezke ala banan-banan? Posible da denak batera irakurtzea, baina horretarako karaktere-katea datu-mota erabili behar da; orain arte aztertu ez duguna.

– Posible al litzateke hau ebazteko 2. enuntziatuan erabilitako kodeaz baliatzea? Erantzuna baiezkoa da, baina horretarako 4.14. programa nagusia (`main`) izan beharrean ekintza ez-primitibo (funtzioa, prozedura edo errutina izenaz ere ezagutzen da) gisa definitu behar da, hurrengo kapituluan aztertuko dugunez.

5. *FUNTZIOAK ETA MAKROAK*

5.1. FUNTZIOAK

Goi-mailako beste lengoaietan bezala, C-k azpiprogramak idazteko aukera ematen digu. Lengoaiaren arabera azpiprogramei izen desberdinak ematen zaizkie, *prozedura*, *funtzioa*, *errutina* eta *azpirrutina* arruntenak direlarik. Izendapen desberdinak garrantzi txikiko ezaugarri diferenteek desberdintzen dituzte. Horregatik funtzioa diogunean, C-ren kasuan behintzat, azpiprogramak inplizituki emaitza bat itzuliko duela suposatzen da. Prozedurak aldiz, emaitza inpliziturik ez du itzultzen.

Funtzio baten izena programa-zati baten laburdura dela pentsa daiteke. Funtzioa behin definitzen da, baina sarritan deitua izan daiteke. Maiz exekutatzen den sententzi multzoa funtzio gisa jarri ohi da edozein programazio-metodologia erabiliz. Funtzioen erabilerak, irakurgarritasuna bultzatzeaz gain, handitzen du programak idazteko eta aldatzeko erraztasuna, malgutasuna eta fidagarritasuna.

C-ren funtzioen helburua, azpiprograma guztiena bezala, programan diseinua eta idazketa erraztea da; beraien bidez posible baita problema konplexu bat errazagotan banatzea, horietako bakoitza programa errazagoaz ebatziko delarik. Behe-mailako funtzioek eragiketa sinpleenak burutzen dituzte eta goi-mailakoek behe-mailakoak erabiltzen dituzte. Teknika honi beheranzko programazioa edo programazio modularra deitzen zaio, eta gaur egun oso hedapen handia du programatzaileen eraginkortasuna (garapen zein zuzenketaren garaian) hobetzen duelako. Beheranzko programazioa sakonago ikusiko da 12. kapituluan.

Ikus dezagun adibide bat; faktorialaren kasuarena esaterako. Aurreko bi kapituluetan faktoriala kalkulatzen duten programa desberdinak egin ditugu, baina beti programa bakar batean. Hori ez da aukera bakarra; zeren faktorialaren kalkulua azpiprograma edo funtzio batean programa baitaiteke behin betirako, eta programa nagusi desberdinetatik erabili 5.1 programan azaltzen den legez.

```
#include <stdio.h>

/* funtzioaren definizioa */
long faktoriala (n)
int n;
{
long em = 1;
int i;

for (i = 1; i<=n; i++)
   em = em * i;
return (em);
}

/* faktorialaren 1. erabilpena */
main ( )
{
int zenb;
long fakt;

printf("Sakatu zenbaki bat\n");
scanf ("%d", &zenb);
if (zenb < 0)
   printf ("errorea: negatiboa da!\n");
else
   {
   fakt = faktoriala (zenb);
   printf ("%d-en faktoriala %ld da\n", zenb, fakt);
   }
}
```

5.1. programa. *Faktoriala funtzio baten bidez.*

5.2. programan funtzio beraren erabilpena ikus daiteke, $\frac{m!}{n!(m-n)!}$ formulaz $\binom{m}{n}$ espresioa kalkulatzeko.

```
#include <stdio.h>

main ()     /* beste modulu batean egon daiteke */
{
extern long faktoriala (int n);
int m, n;
float em;

printf("Sakatu bi zenbaki, lehena bigarrena baino handiagoa:\
   \n");
scanf ("%d %d", &m, &n);
em = faktoriala(m) / (faktoriala(m-n) * faktoriala(n));
printf ("Emaitza %f da.\n", em);
}
```

*5.2. **programa**. Faktorial funtzioaren 2. erabilpena.*

Bigarren programa horretan ez dugu faktorialaren kodea idatzi, eta funtzioa aparte konpilatuko dela adierazi beharko zaio konpiladoreari. Horregatik faktoriala zein motatakoa den adierazi behar da, `extern` bereizgarria zehaztuz funtzioa erazagutzen denean. Eskema honekin, faktoriala funtzioaren definizioa konpilatzean sortzen den objektu-modulua estekatuko da programa nagusiak sortutakoarekin batera programa exekutagarri bakarra lortuz.

Adibideetan ikusitakoaren arabera, C prometan funtzioak erabiltzen badira hiru moduko aipamenak egin daitezke: definizioa, deia eta erazagupena. *Definizioa* funtzioaren izena eta mota, argumentu edo parametro formalak eta gorputza —dagokion kodea— zehazten da. *Deia* aipamen hutsa da, eta beraren bidez erreferentziatzen da funtzioa, hau da, bere exekuzioa eskatzen da. Deian izenaz gain parametro formalen balioak zehaztuko dituzten parametro efektiboak idazten dira. *Erazagupen* batean, azkenik, beste modulu batean definitutako funtzio baten izena, mota, eta, kasuaren arabera, parametroak definitzen dira baina gorputza zehaztu gabe.

5.2. FUNTZIO-MOTAK

void motakoak ez diren funtzio guztiek balioren bat itzultzen dute. Funtzioari egokitu behar zaion mota itzultzen duen balioarena da. Funtzioaren emaitzan itzultzen denaren arabera hiru multzotan bana daitezke funtzioak:

1. Konputazioaren emaitza itzultzen duten funtzioak. Funtzio hauek beren argumentuekin eragiketak burutzen dituzte, lortzen duten emaitza itzuliz. Azpirrutina hauek funtzio hutsak dira. Emaitzaren bat itzultzen dutela ere, ez da derrigorrezkoa balio hori erabiltzea, alde batera utz baitaiteke. Adibide bat 5.3. programan ikus daiteke.

```
#include <stdio.h>

int batu(a,b)
int a, b;
{    return(a+b);
}

main()
{
int x, y, z;

x=10;
y=20;
z=batu(x, y);
/* itzulitako balioa z-ri esleitu */
printf("%d", batu(x, y));
/* itzulitako balioa printf-ean erabili */
batu(x, y);
/* itzulitako balioa galduko da */
}
```

*5.3. **programa.** Konputazio helburua duen funtzioaren erabilpenak.*

2. Funtzioaren kalkulua ondo ala gaizki joan den kode bat itzultzen dutenak. Ohizkoa izaten da zerbait gaizki joan denean –1 balioa itzultzea.

3. Esplizituki inolako baliorik itzultzen ez dutenak. Kasu honetan, prozedura izena jaso ohi dute eta *void* motakoa izan behar du emaitza-motak.

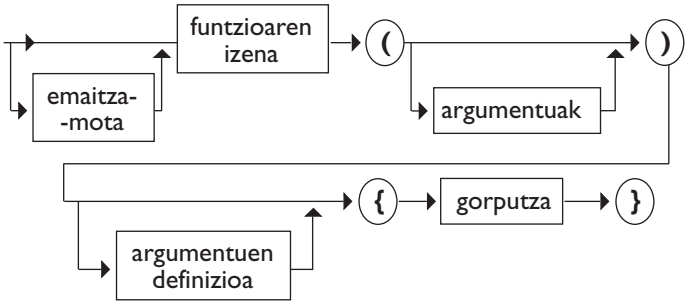
2. eta 3. motak C terminologian funtzio izenak jaso arren, logikoki prozedurak dira eta bestelako emaitzak, baleude, aldagai globaletan —salbuespen gisa erabiltzea gomendatzen da— edo erreferentziaren bidezko argumentuetan itzuli ohi dira.

5.3. FUNTZIOAREN DEFINIZIOA

Funtzioak egiten duena adierazteko balio du definizioak, bertan funtzioaren emaitza-mota, izena, argumentuak, argumentuen definizioa, eta gorputza zehazten direlarik.

Argumentu bat baino gehiago badago, koma karaktereaz bereizten dira. 5.1 programaren funtzioak ondoko osagaiak dauzka: izena, *faktoriala*, funtzio-mota, *long*, argumentu bat, *n*, argumentuaren erazagupena `int n;` eta gorputza, programa nagusiraino hartuz ondoan dagoen guztia. Kontuan hartu beharrekoa da programa nagusia —oraingoz behintzat— argumenturik gabeko *main* izeneko funtzioa dela.

Definizioari dagokion formatua ondokoa da:



Funtzioen definizioaren sintaxia.

Funtzioak emaitza-motarik aipatzen ez badu, osoko zenbakia itzuliko duela suposatuko da, baina beti aipatzea gomendatzen da.

Argumentuen definizioa datuen definizioa bezala egiten da. Diferentzia bakarra zera da: datu hauek beste funtzioetatik —edo programa nagusitik— jasoko dira. Horrexegatik argumentu hauei *formalak* deitzen zaie eta funtzio--deiaren datu zehatzek ordezkatuko dituzte.

Funtzioaren gorputzean azpiprogramaren kodea doa, funtzioaren datuen definizioa zein aginduak bertan adierazten direlarik.

Goazen bi osoko zenbakietan handiena itzultzen duen definizioa idaztera: izena asmatuko dugu, *handien* adibidez; parametroak bi dira eta gainera biak osoko zenbakiak (*a* eta *b* izenekoak hain zuzen) eta emaitza-mota bi zenbakietako bat izango denez, osokoa ere izango da. Beraz, funtzioaren burua honako hau izango da:

```
int handien (a, b)
int a, b;        /* funtzioaren parametroak */
```

Gorputza oso sinplea da, **{** karaktereaz hasi eta **}** karaktereaz bukatuko da eta adibidean, beste gauzen artean, *emaitza* izeneko aldagai lokala erabiliko da.

```
{
int emaitza;     /* aldagai lokala */

if (a > b)
     emaitza = a;
else
     emaitza = b;
return (emaitza);
}
```

`return` gako-hitzaren ondoko espresioaren balioa izango da emaitza.

5.4. FUNTZIOAREN DEIA

Funtzio bat exekutatu nahi denean dei edo erreferentzia bat egingo zaio, horrela deian aipatutako datuekin —parametro efektiboekin— funtzioaren definizioari dagokion kodea exekutatu eta emaitza itzuliko da eta. Dei bat egitean, datu-bihurketa automatikoak gertatzen dira. Horrela, *char* eta *short* motako argumentuak *int* motara bihurtuko dira eta *float* motakoak, *double*-ra.

Funtzioak emaitzarik itzultzen badu, deia normalean esleipen baten eskuinaldean agertuko da seguruen. Hala ere, espresio baten erdian edo beste funtzio baten parametro gisa ere ager daiteke.

Pentsa dezagun bi zenbaki irakurri ditugula *z1* eta *z2* aldagaietan eta bietako handiena idatzi nahi dugula. Ondoko bi aukera hauek dauzkagu:

```
a) int em;
   ...
   em = handien (z1, z2);
   printf ("%d", em);

b) printf ("%d", handien (z1, z2));
```

Hasieran azaldutako 5.2. programan, *faktoriala* funtzioari programa bereaan hiru aldiz deitzen zaiola ikus daiteke, baina parametro desberdinak erabiliz.

Azpiprograma batean emaitza bat baino gehiago lortu behar denean, ezin da aipatutako *return* sententziaz egin —honen bidez emaitza bakarra itzul baitaiteke—. Hau erreferentziaren bidezko parametroen bidez egingo da (ikus kapitulu honetan funtzioen argumentuei buruzko atala eta 6. kapitulua).

5.5. FUNTZIOAREN ERAZAGUPENA

Programa batean beste modulu batean definituta dagoen funtzio bati deitzen diogunean funtzioa erazagutu egin behar da, definizio faltagatik konpiladoreak akatsik eman ez dezan. Hori 5.2. programan egin da eta erazagupenean zehaztu behar dena ondokoa da: *extern* gako-hitza funtzioa kanpoan definitzen dela adierazteko, funtzio-mota, funtzioaren izena eta parentesiak funtzioa dela bereizteko. Ari garen adibidean erazagupena honako hau litzateke:

```
extern int handien ( );
```

Argumentuen motak eta kopurua definitzea behar-beharrezkoa ez bada ere ANSI C-n badago aukera hori, parentesien artean parametroen motak zehaztuz. Honela, konpiladoreak egiazta dezake funtzioaren dei guztietan zuzenak diren motak eta kopuruak azaltzea. Gai honi buruz aurrerago sakonago hitz egingo da. Beraz, erazagupena honela jarriko litzateke:

```
extern int handien (int, int);
```

Funtzioa eta deia modulu berean daudenean ez da funtzioaren erazagupenik behar, funtzioaren definizioa deia baino lehenago idazten baldin bada, bestela erazagupena beharrezkoa da baina *extern* gako-hitza jarri gabe.

Funtzioa erazagutzen ez bada, *int* motako emaitza itzuliko duela suposatuko du konpiladoreak eta horrek arazoak sor ditzake. Hori dela eta, funtzioak beti erazagutzea da ohiturarik onena. Aurrekoa frogatzeko ikus dezagun 5.4. programa. *batu* funtzioak *float* motako balioa itzultzen duela jakin behar du programa nagusiak, bestela frakziozko aldea galduko bailitzateke.

```
#include <stdio.h>

float batu(float, float); /*funtzioa erazagutu beharra */

main()
{
float lehen, bigarren;

lehen = 123.23;
bigarren = 99.09;
printf("%f", batu(lehen, bigarren));
}

float batu(float a, float b)
{
return a+b;
}
```

5.4. programa. *funtzioak erazagutzeko beharra.*

Funtzioaren erazagupenak konpiladoreari *batu*-k koma higikorreko emaitza itzuliko duela esaten dio. Erazagupenaren laguntzaz konpiladoreak objektu-kode zuzena sortuko du eta ez da errorerik gertatuko.

Erazagupenik gabe, ordea, errore bat sortuko da seguruen, deia egiten duenak espero duen datu-mota eta funtzioak itzultzen duena desberdinak direlako. Funtzio biak fitxategi berean baldin badaude, konpiladoreak moten arteko errorea nabarituko du, programaren konpilazioa etenez. Fitxategi berean ez badaude, aldiz, konpiladoreak ez du errorea aurkituko, moten egiaztapena konpilazio-garaian bakarrik gertatzean, estekaketa-prozesuan zehar ez baitira detektatuko. Beraz, kontu handiz ibili beharko da funtzioen motekin, eta funtzioen motak beti erazagutzea da irtenbiderik onena ustekabeko erroreak saihestu nahi badira.

5.6. *RETURN* SENTENTZIA

Sententzia honek bi helburu ditu: azaltzen den funtzioaren amaiera behartzea —hau da, programaren egikaritzapena funtzio-deia egin duen kodera itzultzea—, eta, hala egin behar denean, itzuli beharreko balioa zehaztea.

Funtzioaren amaierari dagokionez, *return* sententzia ez da ezinbestekoa, konpiladoreak funtzioaren azken sententziaren ondoren behartu baitezake itzulera hori, funtzioaren bukaerako ixteko giltza aurkitzean. Ondoko funtzioak, adibidez, N karaktereko taula alderantziz inprimatuko du pantailan.

```
void inprimatu_alderantziz(tau)
char tau[N];
{
int i;

for (i=N-1; i; i--)
  printf(tau[i]);
}
```

N karaktereak inprimatu ondoren, funtzioak ez du zeregin gehiago eta bukatuko da, kontrola programa deitzaileari itzuliz.

return sententzia ezinbestekoa da ordea zero ez den balio esplizitu bat itzuli behar baldin bada. Kodea errazteko edo laburtzeko ere balio du, funtzioa eraginkorrago bihur dezake, irteera-puntu bat baino gehiago zehaztuz. Funtzio batek, beraz, *return* bat baino gehiago eduki dezake.

return sententziaren sintaxi posibleak honakoak dira:

```
return espresioa;
return (espresioa);
return;
```

Lehenengo biak baliokideak dira eta espresioaren ebaluazioaz sortzen den emaitza itzuliko zaio deitzaileari, itzuli behar den motara bihurturik hala behar bada. Hirugarrena erabiliko da kontrola itzuli besterik egin nahi ez denean.

5.7. *MAIN* FUNTZIOA

main funtzioa C programa bat hastean exekutatuko den lehen funtzioa da, hau da, C-ren programa nagusien izena *main* da beti. *main* funtzioa bukatzean, beraz, programaren exekuzioa bukatuko da. Beste funtzioen sintaxi-arauak jarraitzen ditu *main* funtzioak ere.

Orain arteko adibideetan, *main* funtzioak itzultzen duen balio-mota ez dugu adierazi. Normalki, zero osoko balioa izango da itzuliko dena, programa zuzen exekutatu bada. Bestelako baliorik itzultzea nahi izanez gero, *return* sententzia erabil daiteke, lehenago ikusi bezala.

Funtzio bat denez gero, *main* funtzioak argumentuak ere eduki ditzake, orain arte erabili ez baditugu ere. 6. kapituluan, taulak eta erakusleak ikusi ondoren, sakonean ikusiko dira *main* funtzioak eduki ditzakeen bi argumentuak nolakoak eta nola erabiliko diren.

5.8. FUNTZIOEN ARGUMENTUAK

Funtzioen definizioetan agertzen diren argumentu formalak funtzioaren aldagai lokalak bezalakoak dira: funtzioan sartzean sortu eta irteterakoan desagertzen dira.

Beste goi-mailako lengoaia askok ez bezala, egiaztapen gutxi egiten du C-k eta, errore logikoak egon arren, exekuzioarekin jarraitzen du. Argumentu formalek eta funtzio-deian erabilitako argumentuek mota berekoak izan behar dute. Moten arteko erroreren bat badago, konpiladoreak ez du errore-mezurik erakutsiko argumentuen motak zehazten ez badira funtzioaren erazagupenean, baina lortuko diren emaitzak, askotan, ez dira espero direnak izango. Funtzioen prototipoak erabiliz gero, aldiz, lehen aipatu bezala argumentuen motak eta kopurua zuzenak diren ala ez egiaztatzen da.

Aldagai lokalak bezala, parametro formalak C-k onartutako edozein espresiotan azal daitezke funtzioaren gorputzean, baita esleipenetan ere. Aldagai hauen zeregina funtzioari emandako argumentuen balioa jasotzea den arren, beste edonolako aldagaiak bezala erabiliak izan daitezke.

Argumentuak bi modutan pasa daitezke.

1. Balioaren bidez. Era honetan argumentuari dagokion balioa kopiatzen da funtzioaren parametro formalean beraz, ondoren aldagaian gertatutako aldaketek ez dute eraginik izango programa deitzailean, albo-ondorioak saihestuz.

2. Erreferentziaren bidez. Hemen, argumentuaren helbidea da parametro formalaren balioa. Azpirrutinaren barruan balioaren helbidea —erreferentzia ere dei daiteke— erabiliko da, deian pasatutako argumentua kopiatu gabe zegoen tokitik erabiliko delarik. Honela, funtzioa deitzeko erabili diren aldagaiek ere funtzioan zehar gertatutako aldaketak jaso ditzakete. Bide hau erabiliko da emaitza anitz lortu nahi direnean, edo egitura luzeen tratamendua eraginkorragoa egiteko.

Kasu gehienetan, C-k balioaren bidezko deia erabiltzen du argumentuak pasatzerakoan eta, beraz, funtzio-deian erabilitako aldagaiak ez dira aldatuko. 5.5 programan adibide bat azter daiteke.

```
#include <stdio.h>

int karratua(int j)
{
j = j*j;
return(j);
}

main()
{
int i=10;

printf("%d da %d zenbakiaren karratua",karratua(i),i);
}
```

5.5. programa. *Balioaren bidezko deiak.*

Adibide honetako hasieran, *j* argumentuari 10 balioa emango zaio. Aldiz, karratua funtzioan biderkaketa egiterakoan, aldaketak jasango dituena *j* aldagai lokala izango da, programa nagusiko *i* aldagaiak 10 balioa mantenduko duelarik. Beraz, programa honek inprimatuko duena ondoko mezua izango da: `"100 da 10 zenbakiaren karratua"`.

Balioaren bidezko parametroetan gogoratu behar dena zera da: argumentuaren kopia bat bidaltzen da, programa deitzailean jatorrizko balioa geldituz. Funtzioaren gorputzean gertatutakoak ez du eraginik deian erabilitako aldagaian.

Erreferentziaren bidezko parametroak erakusleen bidez erabiltzen direnez, 6. kapituluan aztertuko dira sakonean.

5.9. FUNTZIOEN PROTOTIPOAK

Funtzioen prototipoak AT&T-ko Bjarne Stroustrup-ek proposatu zituen, eta, geroago, ANSI C komiteak onartu zituen. C++-en osagai bat dira ere, 14. kapituluan azalduko den bezala. Aipatu den bezala, funtzioen prototipoak argumentuen datu-motak jartzea onartzen duten funtzioen erazagupenak dira.

Horrek bi abantaila dakar:

a) Funtzioaren argumentu formalak eta funtzio-deian azaldutako argumentuak bateragarriak diren eta kopuru berdina duten egiazta dezake konpiladoreak, akatsak konpilazio-denboran detektatuz.

b) Ez ditu bihurketa automatikoak egiten uzten. Koma higikorreko motak ez dira *double*-ra bihurtuko, ezta osoko zenbaki laburrak (*char* edo *short*-ak) *int* motara ere. Honela, bihurketa horiek ez gertatzean, osoko zenbaki

laburrak edo koma higikorreko datuak erabiltzen dituzten algoritmoak asko azkartuko dira.

Funtzioak erazagutzeko sintaxia lehengoa da, baina parentesien artean argumentuen motak idatz daitezke oraingoan koma karaktereaz bereiziz. Adibidez, eta beti ANSI C konpiladoreez ari garela kontuan hartuz:

```
void funtzioa(int, float, long);
```

Honela, hiru argumentu —lehena *int* motakoa, bigarrena *float*-a eta *long* hirugarrena— onartzen dituen funtzioa erazagutuko da. Parametro-trukean egiaztapenik ez egotea nahi izango bagenu, orduan `void funtzioa();` erazagupena egingo genuke. Argumentu-motekin, argumentuen izenak ere idatz daitezke. Adibidez, aurreko erazagupena honela ere jar daiteke:

```
void funtzioa(int a, float higi, long b);
```

Argumentuen izenen zeregina erazagupenen irakurketa erraztea da soilik; ez baita memoriarik gordeko beraientzat, ezta gatazkarik izango ere izen bera duten aldagaiekin. Argumenturik ez duen funtzioaren prototipoan, *void* mota erabiliko da.

Funtzio-dei bat egiten bada `funtzioa(x,y);` idatziz, konpiladoreak errore bat itzuliko du, deiak bi argumentu bakarrik dituelako, erazagupenean hiru azaltzen diren artean. Argumentuen motak prototipoen motetara ezin baldin badira bihurtu, beste konpilazio-errore bat gertatuko da. Argumentuen bihurketen erregelak esleipenarenak berberak dira (ikus 3. kapitulua). Beraz, prototipoak erabiliz egiten den mota-egiaztapenak sortutako erroreak datu-mota batzuentzat, erakusleetarako batez ere, dira interesgarri, baina ez osoko edo koma higikorreko kasuetan, hauetan datu-bihurketa inplizituak gertatzen baitira. Adibidez, ondorengo adibidean *j float* motara eta *x short*-era bihurtuko dira, argumentu bezala pasatu aurretik.

```
{
...
void g(float, short);
double x;
long j;
...
g(j,x);
...
}
```

Prototiporik gabe, adibide honek emaitza okerra emango luke, `j` *float* eta `x` *short* bezala tratatuko bailituzke.

Prototipoak erabiltzeak funtzioak eraginkorrago bihur ditzakeela esan dugu. 5.6 programan ez dira automatikoak liratekeen *float*-etik *double*-rako bihurketak egingo, *main* funtzioak *float* zenbakiak pasatzen baitizkio *karratuen_batura* funtzioari, argumentu bakoitzeko bi bihurketa bazterraraziz. Honela, guztira sei bihurketa aurrezten dira.

```
#include <stdio.h>

main()
{
float karratuen_batura(float, float, float);
float x, y, z;

printf("Sakatu koma higikorreko hiru zenbaki: ");
scanf("%f %f %f", &x, &y, &z);
printf("%f, %f eta %f zenbakien karratuen batura %f da.\n",
   x, y, z, karratuen_batura(x, y, z));
}

float karratuen_batura(float a, float b, float c)
{
return(a*a + b*b + c*c);
}
```

*5.6. **programa.** Funtzioaren prototipoen adibidea.*

5.10. ALDAGAIEN EZAUGARRIAK

Aldagaiek bi funtsezko ezaugarri dute: esparrua eta iraupena. Lengoaia bateko esparru-arauak kode-bloke batek beste kode-bloke batean dagoena atzi dezakeena kontrolatzeko dira.

C-n funtzioak kode-bloke diskretuak dira, hau da, funtzio baten gorputza funtzio horrentzat pribatua da eta ezin du beste funtzio bateko inolako espresiorik atzitu, aurreko funtzioari dei bat egiten ez zaion bitartean. Adibidez, ez da posible *goto* sententzia erabiltzea beste funtzio baten gorputzera jauzi egiteko. Funtzioaren gorputza ezkutatuta dago programaren beste edozein atalentzat, eta atal horiek ezin diote inolako aldaketarik eragin bere aldagaietan —erreferentziaren bidezko parametroen bidez ez bada noski—. 3. kapituluan ikusi den bezala, sententzia konposatuan aldagaiak erazagut daitezke. Aldagai hauek aldagai lokalak dira eta existituko dira funtzioa exekutatzen den bitartean.

Beste aldetik, horretarako propio den mekanismo bat erabiltzen ez den bitartean, aldagai lokalek ezin dute beren balioa gorde bi funtzio-deien artean. Erregela hau hausteko modu bakarra, aldagaiaren iraupena aldatzen duen *static* modukoa definitzea da, 8. kapituluan sakonago ikusiko den bezala. Aldagai globalak —global izateak esparru eta iraupen berezia edukitzea besterik ez da— erabiltzeko modua ere 8. kapituluan ikusiko da.

5.11. ERREKURTSIBITATEA

Programazio-lengoaia errekurtsiboa baldin bada, funtzio batek bere buruari dei diezaioke. C-ren funtzioek beren buruari dei diezaiokete. Funtzioaren definizio batean funtzio berari deia egiten bazaio, funtzioa errekurtsiboa da.

Errekurtsibitatearen adibide bezala, *n*-ren faktorialaren definizio errekurtsiboa erabiliko da. *n* zeroa bada, *n*-ren faktorialaren balioa batekoa da. *n* zero baino handiagoa bada, aldiz, *n*-ren faktorialaren balioa (*n–1*)-en faktorialaren balioa bider *n* izango da. Faktorialaren bertsio errekurtsiboa 5.7 programan azaltzen dena da.

```
long fakterre(int n)
/* faktorial errekurtsiboa */
{
int erantzuna=1;

if (n<0)
   {
   printf("Zenbaki negatiboek ez dute\faktorialik\n");
   return(0);
   }
if (n==0)
   return(1);
erantzuna=fakterre(n–1)*n;
return(erantzuna);
}
```

5.7. ***programa.*** *Errekurtsibitatearen bidezko faktorialaren definizioa.*

fakterre funtzioari `0` balioa iritsiz gero, funtzioak `1` itzuliko du. Bestela, *fakterre*-k `(n–1)*n` itzuliko du. Espresio hori ebaluatzeko *fakterre*-ri `(n–1)` argumentuaz deituko zaio. Hau *n* `0` izan arte gertatuko da errekurtsiboki. Une horretan, ordurarteko funtzio-deiak itzuliz joango dira azken balioa lortu arte.

Funtzio batek bere buruari deitzean, pilan aldagai lokal eta parametroetarako lekua gorde egiten da, eta aldagai berri horiekin funtzioaren kodea exekutatuko da hasieratik. Dei errekurtsibo batek ez du funtzioa kopiatzen, aldagai berriak sortzen baizik. Dei errekurtsibo batetik itzultzean, dei horretako aldagai lokalak eta parametroak desagertuko dira pilatik eta funtzioaren egikaritzapenak bere buruari deitu zion puntutik aurrera jarraituko du.

Errutina errekurtsiboek abiadura-galera dakarte gehienetan. Gainera, errekurtsioan urrats asko ematen baldin badira pilaren gainezkada gerta daiteke. Funtzio errekurtsiboak, berriz, era errepikakorrean idatzitako bertsioa baino argiagoa eta errazagoa izan daiteke programatzeko orduan, adibidez, ordenazio-algoritmoak edo adimen artifizialaren problema batzuk ebazteko.

Funtzio errekurtsiboak idaztean, errekurtsibitatetik alde egitea bideratzen duen *if* espresioren bat eduki behar da. Hau egin ezean, funtzioak ez du sekula kontrola itzuliko eta programa etengabe arituko da pila erabat bete arte. Zeharkako errekurtsioa ere eman daiteke. Adibidez, *f1* funtzioak *f2*-ri, *f2*-k *f3*-ri eta *f3*-k *f1*-i deitzean gertatuko da errekurtsioa: azken finean, *f1*-k bere buruari deitu diola esan daiteke eta.

5.12. Moduluak eta liburutegiak

C-k konpilazio banatua onartzen du. Gehienbat, hau gertatuko da garatzen ari den programa oso handia bada. Honela, kode guztia fitxategi edo modulu batean gorde beharrean, modulu desberdinetan bilduko da, konpiladore eta estekatzailearen artean loturak ebatziko dituztelarik. Modulu horietako batzuk *liburutegi*etan antolatzen dira. Funtzioen liburutegiak eta banaka konpilatutako eta funtzioez osaturiko fitxategiak ezberdinak dira. Liburutegiko funtzioak erabiltzean, estekatzaileak behar dituen funtzioak hautatuko ditu liburutegitik, banaturik konpilatutako fitxategian, errutina guztiak kargatu eta estekatuko dira programarekin. Programatzaileak sortutako eta funtzioez osatutako fitxategi gehienetan, fitxategiaren funtzio guztiak erabiltzea nahi izango da. C-ren liburutegi estandarraren kasuan, aldiz, funtzio guztiak programarekin ez dira estekatuko, objektu-kodearen tamaina izugarria izango bailitzateke.

Bi motako liburutegiak bereizten dira: *sistemarenak* edo *estandarrak* batetik, sistema eragilearekin batera aurreprogramaturik datoz, erabiltzaile guz-

tiek erabil ditzaten; eta *erabiltzailearenak* edo *programatzailearenak* bestetik, norberak programatutakoak dira programa desberdinetan erabiltzen diren funtzioak aurreprogramaturik edukitzeko. Programatzailearen liburutegiak sortzeko sistemaren komandoren bat erabili beharko da, 12. kapituluan ikusiko denaren arabera. Bi mota hauen artean beste maila bat bereiz daiteke: software-ekoizleek saltzen dituzten helburu orokorreko liburutegiak. Liburutegien eta bestelako moduluen arteko ezberdintasunak aurrerago azalduko dira.

Metodologiaren aldetik, komenigarria litzateke, alde batetik, kontzeptualki erlazionaturik dauden funtzioak modulu berean sartzea. Adibidez, testu-editore bat osatuko duten funtzioak programatzerakoan, testua ezabatzeko funtzioak modulu edo fitxategi batean, testu-bilaketak egiteko funtzioak beste batean, eta abar. Bestalde, ondo definitutako datu-multzo baten gainean oinarrizko eragiketak burutzen dituzten funtzio guztiak liburutegi batean joatea ere oso interesgarria izango da.

Beste kapitulutan honetaz luzatuko bagara ere, liburutegiak sortzen direnean liburutegi bakoitzeko buru-fitxategi bat sortzen da, bertan liburutegiari dagozkion datuak eta funtzioak erazagutzen direla. Horrela liburutegi bateko funtzioak erabiliko dituzten programek dagokion izenburu-fitxategia integratuko dute, konpilazio-prozesua modurik onenean buru dadin. Orain arteko programa batzuetan ikusi izan den `#include <stdio.h>` sententziaren bidez sarrera/irteerako liburutegi estandarraren *stdio* izenburu-fitxategia sartzen da programan.

5.13. Makroak

Mihiztadura-lengoaietan hain arrunt den programazio-tresna hau goi-mailako lengoaia gutxi batzuetan agertzen da. C-n aukera hau dugu mihiztadura-lengoaiaz egiten dena goi-mailan egiteko tresna baita C.

Makroak azpiprograma sinpleak definitzeko eta erreferentziatzeko erabiltzen dira, azpiprograma konplexuetarako ez baitira egokiak, baina funtzioekiko funtsezko desberdintasuna dute: makroen erreferentziek edo deiek ez dute azpiprogramaren exekuzioa eragiten; aldiz, konpiladoreak makro-erreferentzia bat ikustean definizioaren gorputzaz ordezkatzen du. Beraz, konpilazio-garaian makroen deiak edo erreferentziak desagertu eta kode berri bat konpilatu egiten da.

Makroen parametroetan parentesiak erabiltzea gomendatzen da, bestela arazoak egon daitezke eta.

5.14. MAKROEN DEFINIZIOA ETA ERREFERENTZIA

Makroak definitu eta erreferentziatu (deitu) egiten dira. Definizioaren formatua honako hau da:

makroen definizioaren sintaxia

`#define` sasiagindua konstanteak definitzeko ere erabiltzen da 2. kapituluan ikusi den legez. Era horretan definitutako konstanteak argumenturik gabeko makrotzat ere jo daitezke.

Zenbakirik handiena itzuliko duen makroa definitzeko, honelaxe egingo genuke:

```
#define handien (x, y) ((x) > (y)? (x) : (y))
```

if egitura erabili beharrean, `? :` eragilea erabili izan da; bestela emaitza itzultzerik ez baitago, makroetan *return* erabiltzerik ezin da eta. Erreferentziak funtzio-deietan bezala egiten dira, eragina lortzeko bidea oso diferentea izan arren.

Adibidez, horrelako erreferentzia jarrita:

```
em = handien (zenb1, zenb2);
```

konpiladoreak, sententzia hori makina-lengoaia bihurtzean, ordezkapen bat egiten du, ondoko emaitza lortuz:

```
em = ((zenb1) > (zenb2))? (zenb1) : (zenb2));
```

Horrela da, zeren eta konpiladoreak makro izen bat ikustean makroaren gorputzaz ordezkatzen baitu, argumentu formalen ordez erreferentzi parametroak idazten dituelarik. Ikusten denez mekanismoa funtzioena baino murritzagoa da, eta horregatik gutxitan erabiltzen da.

Gorputza lerroaren bukaeraraino dagoen kodea izaten da. Lerro batean sartzen ez bada, lerroaren bukaeran alderantzizko barra (\ karakterea) jarriko da, gorputzak hurrengo lerroan jarraitzen duela adieraziz. Alderantzizko barrak karaktere-kate bat hurrengo lerroan jarraitzen duela adierazteko ere balio du.

Makroen argumentuak ez dira aldagaiak, ez dute motarik ezta memori zatirik. Kontuz erabiliz gero, tresna ahaltsua eta eraginkorra da hau. Adibidez, *handien* izeneko makroak argumentu gisa dituen balio bietatik handiena itzuliko du, mota kontutan izan gabe. Makro honen baliokidea den funtzioa idaztea oso zaila da, mota guztiak kontuan hartzeko ez baitaude pentsatuta funtzioak.

Zirkuluaren azalera kalkulatzeko makroa ondokoa izan daiteke:

```
#define PI 3.1416
#define azalera (r)         PI*(r)*(r)
```

Makro baten izenak bere esanahia iturburu-fitxategiaren amaiera arte gordeko du, 10. kapituluan sakonago ikusiko den *#undef* sasiagindua erabiltzen ez bada behintzat.

5.15. AURREDEFINITUTAKO MAKROAK

ANSI estandarrean aurrekonpiladorearentzat aurredefinituriko bost makro daude. Makro hauek ezin dira birdefinitu eta ezin zaie *#undef* sasiagindua aplikatu. Konpiladore zaharretan, makro hauek guztiak ez egotea gerta daiteke. Beraien izenak bi azpimarraz hasi eta beste bi azpimarraz bukatzen dira, honako hauek izanik: `__LINE__`, `__FILE__`, `__TIME__`, `__DATE__` eta `__STDC__`.

LINE eta *FILE* makroak diagnosi tresnak dira, erroreen tratamendurako erabiliko direnak, uneko lerroa eta fitxategia zehazteko balio dutelako. *TIME* eta *DATE* makroak fitxategi bat konpilatu zen azken aldia jakiteko aproposak dira. Adibidez, programa baten bertsioa jakiteko funtzioa honakoa izan daiteke:

```
void bertsioa_inprimatu()
{
printf("Programa hau konpilatutako azken aldia %s—eko \
     %s—etan izan zen\n",__DATE__, __TIME__);
}
```

STDC makroak konpiladoreak ANSI estandarra jarraitzen duenentz jakiteko balio du. Aldagai logiko gisa erabiltzen da eta bat ez den balioa itzultzen badu edo ez badago, konpiladoreak ANSI estandarrari ez diola jarraitzen adierazten du.

5.16. MAKRO ETA FUNTZIOEN ARTEKO KONPARAKETA

Makroek, funtzioekin konparatuz, ondoren azaltzen diren abantailak eta eragozpenak dituzte.

Abantailak:

1. Makroak funtzioak baino azkarragoak dira, funtzio-deiek dakarten gainkarga ez dutelako.

2. Makroen argumentuen kopurua definizioan azaltzen dena ote den egiaztatuko da. Hala ere, ANSI C-ren funtzioen prototipoen sintaxia segitzen bada, prototipoak erabiliz konpiladoreak ere funtzioen argumentu-kopurua egiazta dezake.

3. Makroak ez daude mota bati egokituak. Edonolako motarentzat balio dute.

Eragozpenak:

1. Makroen argumentuak gorputzean agertzen diren aldi bakoitzean ebaluatuko dira. Honek espero ez diren emaitzak eman ditzake, argumentuetariko batek albo-ondorioak baditu (`i++` edo `--i` kasuetan bezala).

2. Funtzioen gorputzak behin konpilatzen dira eta funtzio-deiek kode berbera konpartitzen dute, kode exekutagarrian gorputza behin bakarrik azaltzen delarik. Makroen kasuan, ordea, azaltzen diren bakoitzean makroen kodea azalduko da. Horrela, programa batean asko erabiltzen den azpiprograma bat makroaren bidez eraikitzen bada, programak kode luzeagoa edukiko du funtzioaren bidez eraikita baino.

3. Makroek argumentuen kopurua egiaztatzen duten artean, ez dituzte argumentuen motak egiaztatzen. ANSI C-n definitutako funtzioen prototipoak argumentuen kopurua eta motak egiaztatzen ditu.

4. Makroak erabiltzen dituzten programak araztea zailagoa da.

6. TAULAK ETA ERAKUSLEAK

Taulak dira datu egituratu arruntenak eta erabilienak. C-n, taulen izenak dagozkien erakusleen edo memoriako helbideen bidez maneiatzen dira. Taulak maneiatzea ez da erakusleen funtzio bakarra, erreferentziaren bidezko parametroak ere erakusleak dira.

6.1. TAULAK

Taula da edozein lengoaiatan gehien erabiltzen den datu-egitura. Bektore, matrize, array, katea eta string hitzak erabili ohi dira taularen sinonimo gisa, batzuetan taula-motaren azpimotak baino ez badira ere.

Elkarrekin gorde eta izen berarekin ezagutzen den mota bereko aldagai--multzo batek osatzen du taula. Multzoaren osagai bat atzitzeko, indize bat behar da. Taulak dimentsio bakarra edo anitza eduki dezake, lehen kasuan bektore ere deitzen zaiolarik. Bi dimentsiokoei edo gehiagokoei matrize deitzen zaie, eta osagaiak bektorearen barruan identifikatzeko indize bat erabiltzen den bitartean, matrizetan bi edo indize gehiago erabiliko dira. Taularen osagaiak karaktereak direnean, taulari karaktere-katea, katea edo *string* deitzen zaio, mota honek ezaugarri desberdinak dituelarik.

C-k ez du egiaztatzen taulen indizeak dimentsioa gainditzen duenentz. Ondorioz, taularen mugetatik kanpo irtetea eta bestelako aldagairen batean edo programa-kodearen barne idaztea gerta daiteke. Programatzailearen zeregina da mugen egiaztapenak egitea, ustekabeko errore detektagaitzak ez gertatzea nahi baldin bada behintzat.

6.2. TAULEN DEFINIZIOA ETA ERABILPENA

C-n taulak esplizituki erazagutu behar dira, konpiladoreak beraientzako behar den memoria gorde dezan. Taulak definitzeko osagaien datu-mota, taularen izena eta dimentsioa edo osagai-kopurua mako artean zehaztu egin behar dira. Taulak dimentsio bakarra badu, erazagupena honela egingo da:

```
mota izena[dimentsioa];
```

`mota` taularen elementu bakoitzeko mota da, `dimentsioa` elementuen kopurua den artean. Dimentsio bakarreko taula batek okupatzen duena, `sizeof(mota)*dimentsioa` da, emaitza bytetan dagoelarik.

Taulak bi dimentsio baditu, dimentsio bakarreko taulen taula bezala ikus daiteke. Honelako taulak erazagutzeko, jarraikoa zehaztuko da:

```
mota izena[lerroak][zutabeak];
```

Taula honen byte kopurua jakin nahi bada, `lerroak*zutabeak* sizeof(mota)` egingo da.

Bi baino dimentsio gehiago dituzten taulak ager daitezke C lengoaian. Dimentsioen kopuru maximoa konpiladorearen menpe dago. Mota honetako taulak ez dira maiz erabiltzen baina zenbait kalkulu zientifikotan funtsezkoak dira. Erazagupena honelakoa da:

```
mota izena[a][b]...[l];
```

Erabilpenean, taula bere osotasunean aipatzeko taularen izena soilik zehazten da. Osagai bat adierazteko aldiz, izena eta indizea mako artean aipatu egin behar dira. Indizeak osagaiaren taula barruko posizioa adierazten du, baina lengoaia gehienetan indizearen balio-tartea batetik osagai-kopururaino (*n*-raino) den bitartean, C-n zerotik (*n–1*)eraino erabiltzen da. Kontutan hartzekoa da mako artean zehazten den indizea edozein espresio oso izan daitekeela; konstantea, aldagaia edo beren arteko konbinazio aritmetikoa.

1. programan adibide bat ikus daiteke, non, 12 hilabeteko salmentak taula batera irakurri ondoren, urteko salmenta eta hileko portzentaiak kalkulatu eta inprimatzen diren. Taula irakurtzeko 6.2. programan definituko dugun funtzio bat erabiliko da.

```
#include <stdio.h>

main ()
{
long salmen [12]; /* taularen definizioa */
int i;
float ehuneko;
long osora=0;
void irakur_taula (); /* funtzioaren erreferentzia */

irakur_taula (salmen, 12); /* funtzioaren deia */
for (i=0; i<12; i++)
  osora = osora + salmen[i];
for (i=0; i<12; i++)
  {
  ehuneko=salmen[i]*100.0/osora;
  printf ("%d\t %ld\t %lf\n", i+1, salmen[i], ehuneko);
  }
printf ("guztira:\t %ld\n", osora);
}
```

***6.1 programa**. Taularen definizioa eta erabilpena.*

6.3. TAULEN HASIERAKETA

Definizioan taulen hasieraketa egin daiteke, horretarako osagaien balioak komen bidez banandu eta **{** hasieran eta **}** bukaeran zehaztu behar delarik. Adibidez, hilabete bakoitzak zenbat egun duen adierazten duen taularen definizioa honelakoa litzateke C-z:

```
int hil-egun[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Ezer ez jartzekotan zeroen bidez hasieratzen dira taulak normalean.

Taula globalak hasieratzea konpiladore guztiek onartzen duten bitartean, taula lokalak hasieratzeko posibilitatea konpiladorearen menpe egoten da, ANSI C konpiladore berrietan, hala ere, posiblea da mota guztietako taulak hasieratzea.

Dimentsio anitzeko taulak dimentsio bakarrekoak bezala hasiera badaitezke ere, errenkada desberdinak giltzen artean mugatzea gomendatzen da irakurgarritasunari begira. Adibidez, lehen 10 zenbaki naturalak eta dagozkien karratuak taula batean gordetzeko hauxe litzateke definizioa:

```
int karratuak[10][2]= {{1,1},{2,4},{3,9},{4,16},{5,25},
          {6,36},{7,49},{8,64},{9,81},{10,100}};
```

Mugarik gabeko taulak ere hasiera daitezke eta C-k, hasieraketa horri esker, beraien mugak kalkulatuko ditu automatikoki; konpiladoreak zerrendan azaldutako balio guztiak gorde ditzakeen taula sortuko baitu taularen tamaina azaldu ezean. Horrela, luzera aldakorreko taulak erazagut daitezke, konpiladoreak beraientzako tokia gordeko duelarik. Taulen dimentsioak luzatu edo moztuko dira, erazagupeneko dimentsioak aldatzea beharrezkoa ez delarik.

Adibidez, aurreko erazagupenak honela ere egin daitezke:

```
int i[]={1,2,3,4,5,6,7,8,9,10};
```

6.4. TAULAK FUNTZIOEN PARAMETRO GISA

Dimentsio bakarreko taulak funtzioen argumentu bezala pasatzean, argumentuan taularen izena agertuko da, eta taularen lehenengo osagaiaren helbidea pasatuko zaio funtzioari. Ez da taula osoa argumentu bezala pasatzen, bere helbidea baizik.

Funtzio batek dimentsio bakarreko taula jaso behar badu, argumentu hori hiru eratara defini daiteke: erakusle bezala, mugatua dagoen taula bezala edo mugarik gabeko taula bezala. Erakusleei buruz geroago hitz egingo da. Zein erabili berdin da, hiru motok erakusle bat jasoko dela esaten baitiote konpiladoreari. Mugak jartzea ala ez berdin da, C-k ez baititu mugak egiaztatzen, aurrerago esan bezala. Posiblea da benetakoak ez diren mugak pasatzea.

```
a) funtzioa(katea)          b) funtzioa(katea)          c) funtzioa(katea)
     char *katea;                char katea[10];             char katea[];
     {...                        {...                        {...
     }                           }                           }
```

6.1. irudia. Taula argumentu bezala erazagutzeko hiru erak: a) erakusle bezala; b) mugatua dagoen taula bezala; c) mugarik gabeko taula bezala.

Lehen programan irakur-taula funtzioa definitu gabe utzi dugu. Taula bat erreferentziatzeko —balioaren bidez ez dira pasatzen oso motela litzatekeelako— parametro gisa bi aukera daude erabilpenari begira: array bezala edo erakusle bezala, 6.2 eta 6.3 programetan ikus daitezkeenez (biak baliokideak dira).

```
#include <stdio.h>

void irakur_taula (taula, os_kop)
long taula [];
int os_kop;
{
int i;

printf("Sakatu taularen osagaien kopuruak:\n");
for (i= 0; i < os_kop; i++)
   scanf ("%ld", &taula[i]);
}
```

6.2 programa. *Taula funtzio baten parametroa*

```
void irakur_taula (p, os_kop)
long *p;
int os_kop;
{
long * paux;

printf("Sakatu taularen osagaien kopuruak:\n");
for (paux = p; paux < (p + os_kop); paux ++)
  scanf ("%ld", paux);
}
```

6.3 programa. *Taulak erakusleen bidez*

Ondoko desberdintasunak daude aurreko bi programen artean:

– Lehenengo soluziobidean indizeak erabil daitezke eta, ondorioz, laneko aldagaia *i* da. Bigarrenean aldiz, indizerik gabe burutuko da prozesua osokoa eta laneko aldagaia *paux* erakuslea izango da.

– *scanf* funtzio estandarrak datua erreferentziaz eskatzen duenez, lehenengo ebazpidean `&` (helbidea) erabiltzen da, baina bigarrenean ez, *paux* berez erakuslea —eta beraz erreferentzia edo helbidea— baita. Bigarren ebazpidean `paux++` egiten denean *paux*-i lau gehitzen zaio (*long*-i dagokion luzera) eta `p+os_kop` espresioa ebaluatzean erakusleari *os_kop* bider lau gehitzen zaio.

– `irakur_taula(salmen, 12)` eta `irakur_taula(&salmen[0], 12)` baliokideak lirateke programa nagusian.

Dimentsio bat baino gehiago badu taulak, lehenengo elementuari erakusle bat pasatuko zaio taularen izena zehazten bada parametro gisa. Dena den, lehenengo dimentsioa izan ezik, beste guztiak erazagutu behar dira, konpiladoreak taula era zuzenean indexatu ahal izateko. Hala nahi bada, lehen dimentsioa ere erazagut daiteke.

6.5. KARAKTERE-KATEAK

Karaktere-katea (*string* ingelesez) karaktereez osatutako taula da, non azken karakterea `'\0'` (0 bitarra) konstantea den. Katearen bukaerako karaktere honen bidez katearen amaiera detektatzen da, katea luzera aldakorrekoa denean.

```
#include <stdio.h>

/* lerro bat irakurri eta kontatu zenbat aldiz agertzen den
   bokal bakoitza */
main ()
{
int kont_bok[5], i;
char lerroa[80], bokalak[6] = "aeiou";

for (i = 0; i < 5; i++)
   kont_bok[i]=0;
printf("Sakatu zurigunerik ez duen 80 karaktereko\
   testua.\n");
scanf ("%s", lerroa); /* ez da & behar taula denean */

for (i = 0; lerroa [i]!= '\0'; i++)
   switch (lerroa [i])
      {
      case 'a': kont_bok[0]++;
                break;
      case 'e': kont_bok[1]++;
                break;
      case 'i': kont_bok[2]++;
                break;
      case 'o': kont_bok[3]++;
                break;
      case 'u': kont_bok[4]++;
                break;
      }

/* karaktereak bukatutakoan idatzi */
for (i = 0; i < 5; i++)
   printf ("%c-en kopurua: %d da.\n", bokalak[i], kont_bok[i]);
}
```

6.4. programa. *Karaktere-kateen erabilpena taulen bidez.*

Karaktere-kateek gainerako taulekiko ondoko diferentziak dituzte:

– String motako konstante bakar batez hasiera daiteke, karaktereak banan banan komen bidez bereizi gabe. Adibidez:

```
char izenburua [20] = "URTEKO SALMENTAK";
```

C konpiladoreak automatikoki eransten dio bukaerako karakterea kateari.

– `'\0'` karaktereaz bukatuko direnez, karaktere-kateak gordeko dituzten taulek gutxienez gorde dezaketen taula luzeenaren luzera gehi bat osagai eduki behar dute. Adibidez, 10 karaktereko kateak gordetzeko `char string[11]` jarriko da.

– Funtzioetara eta makroetara dimentsioa pasatzea ez da beharrezkoa, bukaera detektatzeko `'\0'` karakterea bilatzea nahikoa baita.

– *scanf* eta *printf* liburutegiko funtzioetan *%s* formatu berezi bat dago kateetarako, baina ez gainerako tauletarako.

– Liburutegi estandarrean funtzio-multzo bat dago kateen maneiurako, 11. kapituluan sakonduko direnak. Horietako erabilienetarikoak hurrengoak dira:

  - `strlen(s)`: s-ren luzera itzuliko du.
  - `strcpy(s1,s2)`: s1-en s2 kopiatuko du.
  - `strcmp(s1,s2)`: s1 eta s2 berdinak badira, 0 itzuliko du; baldin s1<s2 zenbaki negatiboa, eta baldin s1>s2 zenbaki positiboa.
  - `strcat(s1,s2)`: s1-en amaieran s2 kateatuko du.

6.5 programan karaktere bat katea batean zenbat aldiz agertzen den kalkulatzeko funtzioa azaltzen da.

```
#include <stdio.h>

int kont_kar (katea, kar)
char katea[], kar;
{
int kont = 0, i;

for (i = 0; katea [i] != '\0'; i++)
  if (katea [i] == kar)
    kont++;
return (kont);
}
```

6.5. programa. `kont_kar` *funtzioa taulen bidez.*

Karaktere-kateen taulak osatzeko, bi dimentsioko taulak erabiltzen dira. Definiziorako aukera bat honetan datza: lehenengo indizean karaktere-kateen kopurua zehaztea eta bigarrenean katea bakoitzaren luzera. Adibidez, gehienez 80 karaktereko 15 katea erazagutzeko ondoko sententzia erabil daiteke:

```
char kateen_taula[15][80];
```

Horietako katea bat atzitzeko, lehen indizea baino ez da jarriko. Adibidez, `kateen_taula`-ren laugarren katea erabili nahi bada, `kateen_taula[3]` —gogoratu indizeak erabiltzean 0tik hasi behar dela— soilik jar daiteke, `&kateen_taula[3][0]` jartzearekin erabat baliokidea dena. Lehenengo motako idazkera normalagoa da C programetan.

6.6. ERAKUSLEAK

Taulekin erabiltzeko joera handia badago ere, erakusleak datu-mota guztiekin erabil daitezke. Edozein aldagai erreferentziaren bidez zehazteko erabiltzen dira erakusleak. Erakusleak aldagaiak dira; beraz, definitu egin behar dira eta memorian kokatuko dira, memoriaren zati bat hartuko dutelarik.

Erakusleak (*pointer* ingelesez) datu baten memoriako helbidea adierazten du eta beti lau bytetan kokatzen da. Definizioan ondoko datuak zehaztu behar dira: erakusten duen datu-mota, * karakterea eta erakusle-aldagaiaren izena. Aldagai batek beste aldagai baten helbidea zehazten badu, orduan lehenak bigarrena apuntatzen edo erakusten duela esango da. Erazagupenak honelako itxura du:

```
mota *izena;
```

Pascalezko erakusleekin alderatzen baditugu, desberdintasun handiena hauxe da: Pascalezkoak ez dira definitu behar, *new* egitean dagozkien memoria konpiladoreak dinamikoki eskuratzen baitu —memoria dinamikoa deritzo mekanismo honi—; baina C-n estatikoak dira, besterik zehazten ez den bitartean. 6.5 programan azaldutakoa erakusleen bidez egiten bada 6.6 programako kodea lortzen da. 6.3 programan ere ikusi da taulak erakusleen bidez nola erabil daitezkeen.

```
#include <stdio.h>

int kont_kar (p, kar)
char *p, kar;
{
int kont = 0;
char *paux;

for (paux=p;*paux!='\0';paux++)
   if (*paux == kar) kont++;
return (kont);
}
```

6.6. programa. `kont_kar` *funtzioa erakusleen bidez.*

Erabilpenerako * eta **&** eragileak erabiltzen dira. * karaktereak erakusleak erakutsitako aldagaiaren balioa adierazten du (Pascal-eko ↑ eragilearen funtzio bera du); *&* eragilea aldiz, erakusle ez diren aldagaien aurrean zehazten da, beraien erreferentzia edo helbidea eta ez balioa edo edukia aipatzeko. `&` eta `*` eragileek argumentu bakarra dute. Adibidez, *kontua* aldagaia 2000. hel-

bidean badago eta bere balioa 100 bada, `helbi=&kontua` egin ondoren, `helbi` aldagaian 2000 balioa egongo da. Geroago, `berri=*helbi;` egiten bada, `berri`-ren balioa 100 izango da.

Funtzio baten emaitza, eta ondorioz funtzio-mota, erakuslea ere izan daiteke, horretarako funtzioaren erazagupenean izenaren ezkerraldean erakusten den datu-mota eta * karakterea zehaztu behar delarik (ikus 6.7. programa).

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

/* karaktere-katea baten kar karakterearen lehen
   agerpenaren helbidea itzultzen duen funtzioa */
char *bila_kar (str, kar)
char str[];
char kar;
{
int i;

for (i=0; str [i]!= '\0'; i++)
   if (str[i] == kar)
      return (&(str[i]));
return (NULL);
}
```

6.7. programa. *Erakusle-motako funtzio bat.*

Erakusleen interesa puntu hauetan zehatz daiteke:

1. Erreferentzia bidezko deiak egin ahal izateko balio dute.

2. Datu-egiturak modu eroso eta eraginkorrean kudeatzeko interesgarriak dira.

3. C-ren memoriaren erreserba dinamikoa beraien bitartez egingo da.

Oso baliabide ahaltsua dira erakusleak, baina oso arriskutsuak ere bai. Hasieratu gabeko edo kontrolik gabeko erakusleek sistemaren babesa kolo-

kan jar dezakete —ondorioa beste programetara edo sistemara jauzi egitea izan baitaiteke—, edota programan aurkitzeko gaitzak diren akatsak sor ditzakete. Beraz kontu handiz erabili behar dira.

Erakusleek zuzeneko mota apuntatzen dutela ziurtatu behar da. Adibidez, *int* motako aldagaien erakusleak apuntatzen duen helbidea osoko aldagai batena dela suposatuko da.

Erakusleak ez dira bateragarriak osoko zenbakiekin, zero zenbakiarekin izan ezik. Zero zenbakia erakusle bati eslei dakioke eta erakusleak zeroarekin konpara daitezke. Sarritan, `NULL` konstantea erabiltzen da zero zenbakiaren ordez, erakusleen balio berezia dela adierazteko asmoz. `NULL` konstantea `<stdio.h>` liburutegi estandarrean definituta dago.

Esan den bezala C-n erreferentziaren bidezko argumentuak pasatzeko erakusleak erabiltzen dira. Funtzioari argumentuaren helbidea pasatzen zaionez, funtziotik kanpo dagoen balioa alda daiteke, emaitzak itzultzeko erabil daitekeelarik.

6.8. programan bi aldagaien arteko balioak trukatzeko funtzio bat definitzen da. Bi parametroak datuak eta emaitzak dira aldi berean. Beraz, ezin da balioaren bidez trukatu erreferentziaz baizik (beste edozein lengoaiatan bezala).

```
#include <stdio.h>

void trukatu (x,y)
int *x, *y; /* parametroak bi erakusle dira */
{
int tmp; /* bertako aldagaia */

tmp = *x; /* x-k erakutsitako balioa->tmp */
*x = *y; /* y-k erakutsitakoa x-k erakutsitakoaren ordez */
*y = tmp; /* tmp-n gordetakoa y-k erakutsitakoaren ordez */
}
```

```
main ()
{
int a, b;

printf("Sakatu trukatu beharreko aldagaiak.\n");
scanf ("%d %d", &a, &b); /* irakurtzeko
      erreferentziaren bidez */
trukatu (&a, &b); /* erreferentziaren bidez */
printf ("%d %d\n", a, b);
}
```

6.8. programa. *trukatu funtzioaren erabilpena.*

6.7. ERAKUSLEEN HASIERAKETA

Erakusle bat definitu ondoren, balioren bat esleitzen ez zaion bitartean balio ezezagun bat du. Balioren bat esleitu aurretik erabili nahi bada, programak eta, beharbada sistema eragileak ere, huts egingo du.

Printzipioz, ***NULL*** balioa emango zaio inora apuntatzen ez duen erakusleari. Dena den, kontuz ibili beharko da erakusle nuluekin, esleipen baten xede-aldagaian erabiltzen bada, programak huts egin baitezake. Erakusle nuluak erakusleen funtzioak programatzeko errazago eta eraginkorrago bihurtzeko balio dezake, adibidez, erakusleen taula bat erakusle nuluaz amaitzen bada.

Karaktere-kateak hasieratzean ere erakusleak hasiera daitezke. Adibidez, honako hasieraketa egitean `era` aldagaiari `"Kaixo denoi"` katearen hasierako helbidea emango zaio:

```
char *era="Kaixo denoi";
```

6.8. ERAKUSLEEN ARITMETIKA

Erakusleekin erabil daitezkeen eragileen artean, aritmetikoak daude alde batetik: ++, —, + eta – , eta erlaziozkoak bestetik: <, >, ==, <=, >= eta !=. Kontu handia izan behar da erakusleek eragiketa aritmetikoetan parte hartzen dutenean, zeren gehitzen edo kentzen zaien unitatea, bat izan beharrean, erakutsitako aldagaiaren luzera baita.

Suposa dezagun osoko aldagaiek memoriako 4 byte okupatzen dutela eta *poso*, osoko aldagaien erakuslea dena, hasieran 1000 balioa duela (1000. helbideko balioa apuntatzen du). `poso++;` egiten bada, *poso*-ren balioa 1004 da eta ez 1001. *poso* inkrementatzen den bakoitzean memoriako hurrengo osokoari apuntatuko dio. Era berean, *poso*-k 1000 balioa edukiz gero eta `poso--;` egiten bada, bere balioa 996 izango da. Inkrementuaz eta dekrementuaz aparte, osoko zenbakiekin batuketak eta kenketak ere egin daitezke. Adibidez, `p2=poso+9;` egitean *poso*-ren ondorengo 9. osagaiari apuntatuko dio *p2*-k.

Horregatik ondoko programa-zatian *p1* aldagaiari bat (karaktere baten luzera) *p2*-ri bi eta *p3*-ri lau gehitzen zaio:

```
char *p1; short *p2; long *p3;
...
p1++; p2++; p3++;
```

Bestetik erakusleen arteko batuketak eta kenketak onartzen dira, erakutsitako datu-motak berdinak badira. Baldintza beretan erakusleak erlaziozko espresioetan ere erabil daitezke.

Bestelako eragiketak ezinezkoak dira erakusleekin: erakusleen arteko biderkaketak edo zatiketak, erakusleen bit-maneiua, *float* edo *double* motako zenbakiekin batuketak edo kenketak, ...

Erakusleetarako erakusleak maila anitzeko zeharkako helbideratzeak lortzeko modua da: lehen erakusleak bigarrenaren helbidea erakusten du, eta bigarrenak, aldiz, gordetako balioa. Zeharkako helbideratze anitza nahi den mailaraino eraman daiteke, baina oso kasu gutxitan da beharrezkoa 2. mailatik gora.

Erakusleetarako erakusleak `*` eragile anitzen bidez maneiatzen dira.

Erakusle batek erreferentziatzen duena beste erakusle bat izatea da ondorioa. Adibidez:

```
int r, *p, **q;
r = 10;
p = &r;        /* *p -> 10 */
q = &p;        /* **q -> 10 */
```

6.9. ERAKUSLEAK ETA TAULAK

C-n biziki erlazionatuak daude taulak eta erakusleak. Taula baten izena lehen osagaia helbideratzen duen erakusle bat da. Beraz, taula bat definitzen denean taula osorako espazioa eta erakusle baterako lau byte erreserbatzen dira.

Esan dugunez, taula baten izena indizerik gabe erabiltzen denean taula osoa aipatzen da, baina horrek ez du osagai guztien balioa esan nahi; baizik eta taula memorian kokatzen deneko tokiaren erreferentzia. Horrela *taula* definitu ondoren —`char taula[10];`— `taula` eta `&taula[0]` zehaztea gauza bera da. Erlazio honetaz ondo jabetzeko 6.2 eta 6.3 zein 6.5 eta 6.6 programak berraztertzea interesgarria da. Aurretik esandakoa kontuan hartuz, bi taulen arteko asignazio edo esleipena burutzeko —karaktere-kateak ere honetan sartzen dira— osagaiz osagai egin behar da, liburutegiko funtzio bat erabiltzen ez bada behintzat; eragiketa hori ezin baita asignazio soil batez burutu.

Erakusleen aritmetika taulen indexazioa baino azkarragoa izatea da taulen elementuak atzitzeko erakusleak erabiltzeko arrazoietako bat. Abiadurako irabazpena tauletako atzipen sekuentzialean lortzen da; ausazko atzipenean aldiz, ez dago diferentziarik.

Erakusleak tauletan jar daitezke, beste edonolako datu-motak bezala. Adibidez:

```
char * hil_izen[12];
```

definizioaren bidez `hil_izen` taulak —erakusle batez definiturik— 12 erakusle dauzka, bakoitzak karaktere bat erreferentziatzen duelarik. Definizio honetan 12 erakusletarako tokia gorde da, baina ez hilabeteen izenetarako tokia. Beste definizio alternatiboa eta arras desberdina ondokoa da:

```
char hil_izen[12][20];
```

non erakusle bat eta 20 karaktereko (zutabe matrizeen terminologian) 12 osagai duen taula bat definitu den. `hil_izen` aldagaia mota desberdina da bi kasuetan, lehenean karaktere baten erakuslea den bitartean, bigarrenean erakusle baten erakuslea.

Erakusleen taulak sarritan erabiltzen dira errore-mezuen taulak gordetzeko. Adibidez, 6.9 programan azaltzen den `errore_esan` funtzioak errore--kodea duen mezua erakutsiko du. Bertan zehazten zen *static* gako-hitzaren esangura 8. kapituluan azalduko da.

```
#include <stdio.h>

void errore_esan(zen)
int zen;
{
static char *errore[]=  {
                        "ezin da fitxategia zabaldu\n",
                        "irakurtzerakoan errorea\n",
                        "idazterakoan errorea\n",
                        "euskarriaren errorea\n"
                        };

printf("%s", errore[zen]);
}
```

```
main()
{
int i;

for (i=0; i<4; i++)
   errore_esan(i);
}
```

6.9. programa. `errore_esan` *funtzioa.*

6.10. FUNTZIOEN ERAKUSLEAK

Funtzioen erakusleak C-ren ezaugarri baliagarriak dira, baina korapilatsuak dira. Korapiloa honengatik sor daiteke: funtzio batek erakusle bati eslei dakiokeen memoriako helbide batean dago, baina funtzioa ez da aldagaia. Funtzioaren helbidea funtzioaren sarrera-puntua da; beraz, erakusle batek funtzioari deitzeko balio dezake.

Funtzioen erakusleak nola dabiltzan ulertzeko, C-n nola konpilatzen eta funtzio-deiak nola egikaritzen diren ulertu behar da. Lehenik, funtzioa konpilatzean iturburu-kodea objektu-kode bihurtuko da, sarrera-puntua azalduz. Programa exekutatzen den artean funtzioa deitua izatean, makina-lengoaiako dei bat egingo zaio sarrera-puntu horri. Beraz, erakusle batek funtzioaren sarrera-puntuaren helbidea gordetzen badu, funtzioa deitzeko balio izango zaigu.

Funtzioaren helbidea funtzioaren izenaren bitartez (parentesi eta argumenturik gabe) lortzen da, taulen helbidea taulen izenen bidez lortzen den bezala.

Ondoko programan *fun* funtzioaren helbidea *pfun* aldagaian kokatzen dugu asignazio baten bidez:

```
extern int fun ( );
int (* pfun) ( );
pf = f;
```

ondo idatzitako testua da, eta bertan, *fun* funtzioaren helbidea *pfun* aldagaian kokatzen da. Funtzioen erakusleen erabilpena ikusteko 6.10 programa azter daiteke.

```
#include <stdio.h>

void egiaztatu(a, b, egi)
char *a, *b;
int (*egi)();
{
printf("Berdinak diren egiaztatzen...\n");
if (!(*egi)(a, b))
  printf("%s\n eta\n%s\n berdinak dira\n", a, b);
else
  printf("%s\n eta\n%s\n ez dira berdinak\n", a, b);
}

main()
{
int strcmp(); /* funtzioaren erazagupena */
char kate1[80], kate2[80];

strcpy(kate1, "Hura zen gizona, hura!");
strcpy(kate2, "Hura zen gizona, hura!");
egiaztatu(kate1, kate2, strcmp);
strcpy(kate2, "Ura nahi zuen gizonak, ura!");
egiaztatu(kate1, kate2, strcmp);
}
```

6.10 programa. *Funtzioen erakusleen erabilpena.*

strcmp liburutegiko funtzioa erazagutzen da, konpiladoreari funtzio bat dela abisatzeko, eta itzuliko duen balioa zein motatakoa den jakinarazteko. C-n ez dago zuzenean aldagai bat funtzio baten erakusle bezala erazagutzeko biderik, beraz, karaktereen erakuslea bezala erazagutzen da, eta bihurketa baten bidez funtzioaren helbidea emango zaio.

`egiaztatu` funtzioaren argumentuak karaktereen erakusle bi eta funtzio baten erakuslea dira, beraz, deia egiterakoan, mota horietako parametroak pasatzen zaizkio. Funtzioen erakusleak argumentu gisa jasotzeko egin behar den erazagupen mota `egiaztatu` funtzioan azaltzen denaren antzerakoa da, itzuliko den mota alda daitekeelarik. `*egi` espresioaren parentesiak beharrezkoak dira, konpiladoreak erazagupena modu zuzenean uler dezan. `egiaztatu` funtzioaren barruan `(*egi)(a,b)` egiteak *egi* aldagaiak apuntatzen duen funtzioa —*strcmp* kasu honetan— deitzen du. Sententzia horrek funtzioaren erakusle bat apuntatzen duen funtzioari deitzeko era orokorra adierazten du.

Funtzioetarako erakusleak oso interesgarriak gerta daitezke, datu edo kode baten arabera funtzio edo errutina desberdina exekutatzeko aukera ematen baitigute. Horren adibide argia eten-bektorea edo 6.11 programan azaltzen den errore-maneiurako modulua dugu.

```
#include <stdio.h>

void (* erroreak [2]) ( );/* ezer itzultzen ez duten
                             funtzioetarako erakusleez
                             osatutako taula da erroreak */

void errore0 ()
{
char c;

printf ("errore arina, jarraitzeko tekla bat sakatu \n");
scanf ("%c", &c);
}

void errore1 ()
{
printf ("errore larria, ezin da jarraitu \n");
exit (); /* programa bukaera UNIX-ean */
}

void hasi_erroreak ( )
{
erroreak [0] = errore0;
erroreak [1] = errore1;
}
```

```
void errore_maneiua (kod)
int kod;
{
(* erroreak [kod]) ( );
}

main()
{
hasi_erroreak();
errore_maneiua(0);
errore_maneiua(1);
}
```

6.11 programa. *Erroreak tratatzeko modulua.*

6.11. ERAKUSLEEKIN GERTATZEN DIREN ARAZOAK

Erakusleekin akatsik gertatzean, oso zaila gertatzen da akatsa aurkitu eta konpontzea. Erakusle akastun bat erabiliz irakurketa edo idazketa bat egitean, memoriako leku ezezagun batean gertatzen dira eragiketa horiek, okerrak eraginez. Lortzen diren emaitza horiek pentsaraz dezakete akatsa programaren beste atal batean dagoela eta akatsa konpondu arte denbora luzea pasatzea sarritan gertatzen da, lan horretan *debugger*-ak edo arazleak oso tresna garrantzizkoak izanik.

Akats asko erakusleak hasieratu ezaren ondorioa dira eta hau gerta ez dadin erakuslea erabili aurretik balio duten datuetara apuntatzen duela ziurtatu behar da.

Beste akats-mota bat erakusleak nola erabili ez jakitetik dator. *&* eta *** eragileen erabilera menperatzen ez bada, nahi ez diren emaitzak lortuko dira, 6.12 programan bezala.

```
#include <stdio.h>

main() /* programa okerra */
{
int x, *era;

x=10;
era=x;
printf("%d", *era);
}
```

6.12 programa. *Erakusleen erabilpen okerra.*

6.12. programak ez du 10 balioa idatziko, bukatzea lortzen bada; ezezagu-na den balio bat inprimatuko da, `era=&x` esleipena egin beharrean `era=x` esleipena egin baita.

Horrez gain, bestelako akats mota bat aldagaiak memorian nola kokatzen diren ez jakitearen ondorioa da. Ezinezkoa da datuen memoriako helbidea aurrez jakitea eta, beraz, erakusleen arteko konparaketa batzuek ez dute zen-tzurik (erakusleen konparaketa zentzurik gabeko eragiketa da, kokapena, eta ondorioz helbidea, konpiladorearen menpe baitago).

6.12. *MAIN* FUNTZIOAREN ARGUMENTUAK

Batzuetan, programei argumentuak pasa behar zaizkio programa ahal-tsuago eta malguago izan dadin. Programei argumentuak pasatzeko era, exe-kutagarriaren ondoren argumentuak idaztea izango da. Adibidez, UNIXen konpiladoreari programaren izena emateko, hauxe zehaztuko da:

```
$ cc programaren_izena
```

`programaren_izena` konpiladorearen argumentua izanik.

`argc` eta `argv` argumentu bereziak erabiltzen dira argumentu horiek jaso-tzeko. *main* funtzio nagusiak eduki ditzakeen argumentuak dira. *argc* osoko

zenbakia da eta komando-interpretatzailetik —*shell* deitu ohi zaio UNIXen— pasatzen den argumentuen kopurua adierazten du. Gutxienezko balioa 1 da, programa exekutagarriaren izena kontuan hartzen baita.

argv zehaztutako argumentuen erakusleen taularen erakuslea da. Taularen osagai bakoitzak argumentu bat erakusten du. Argumentu guztiak karaktere--kateak dira eta zenbakiren bat azalduko balitz, formatu egokira bihurtu beharko da programaren kodean.

Adibidez, 6.13 programak programaren ondoren erabiltzailearen izena zehaztea espero du, ondoren "`Kaixo`" eta jasotako izena idatzi ahal izateko.

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
if (argc==1)
   {
   printf("Izena ez duzu idatzi\n");
   return;
   }
printf("Kaixo %s!\n", argv[1]);
}
```

6.13 programa. `main` *funtzioa argumentuekin.*

Argumentuak banaka atzitu nahi badira, `argv` taula indexatu beharko da. Honela, `argv[0]` programaren izena da; `argv[1]` lehen argumentua, eta abar.

Programaren hasieran parametroak jasotzeko erabili ohi dira *argc* eta *argv*. Argumentuek fitxategi baten izena edo aukera bat adierazten dute askotan.

7. DATU-EGITURAK

Bigarren kapituluan aztertutakoaren arabera, karaktereak, zenbaki osoak eta zenbaki errealak dira C-ren oinarrizko datuak. Horiez gain, C programak idaztean, ondoko datu-motak erabil daitezke: aurreko kapituluan aztertutako taulak eta erakusleak, egiturak, bildurak, bit-eremuak eta enumeratuak. Bestetik, goi-mailako PASCAL bezalako lengoaietan datu-mota berriak defini daitezkeen bitartean, C-k ez du horrelakorik onartzen, `typedef` espresioaren bidez datu-moten sinonimoak defini badaitezke ere. Beraz, kapitulu honetan aipatutako datu-mota zein sinonimoen definizioa eta erabilpena azalduko dugu.

7.1. EGITURAK

Tauletan datu guztiak mota berekoak diren bitartean, egitura edo erregistroan datu-multzo bat dago, baina ez dute mota berekoak izan behar. Egitura osatzen duten datuak edo *eremuak* unitate edo elementu bati dagozkio, diskoan gordetzen den datu-mota arruntena izanik. Liburutegiaren fitxategian adibidez, liburu bakoitzeko erregistro (edo egitura) bat dago, bertan kodea, izenburua, idazlea, orri-kopurua, etab. (mota desberdineko datuak) kokatzen direlarik.

Taulekin konparatuz, datu bat egituran zehazteko indize bat erabiltzea ezinezkoa denez, datu edo eremu bakoitzari izen bat egokitzen zaio, eta horrela datu bat identifikatzeko egitura eta eremuaren izenak zehaztu beharko dira.

7.2. EGITUREN DEFINIZIOA

Beste datu-motetan bezala egituretan definizioa eta erabilpena bereiztuko ditugu, baina oraingo honetan definizioa bi urratsetan egingo da, zeren lehenengoan egitura abstraktua dagokien eremuekin definitzen den bitartean (egituraren definizioa), bigarrenean mota honetako aldagaia(k) definituko baitira.

Egituraren definizioa *struct* gako-hitzaz hasitako eta puntu eta komaz bukatutako sententzia bat da, bertan egituraren izena eta eremuen identifikatzaileak zehazten direlarik. Behin egitura definitu ondoren —konpilazio banatua erabiltzean askotan definizioak goiburuko fitxategi batean daude— egitura horri jarraitzen dioten aldagaiak defini eta erazagut daitezke; horretarako lehenengo *struct* hitza, gero egituraren izena eta, azkenik, aldagaiaren izena zehaztuko da. Konpiladoreak, aldagaiak definitzen direnean, beraientzako behar den memoria erreserbatuko du, egituraren definizioan oinarrituz. Egituraren definizioa egitean aldagaiak ere defini daitezke, kasu honetan bezala:

```
struct liburu
{
int kode;
char izenburua[30], idazlea[30]:
int orri_kop, salneurria;
char arg_etxea[30];
}     lib, lib_nire, lib_zure;
```

Sententzia horren bidez *liburu* egitura definitzeaz gain *lib*, *lib_nire* eta *lib_zure* mota horretako aldagaiak definitzen dira.

Izena esleitzea komenigarria izan arren egitura-aldagai bat definitzeko ez da derrigorrezkoa egiturak izena eduki dezala. Honela idatz liteke aurreko *lib* aldagaiaren definizioa:

```
struct
{
int kode;
char izenburua[30], idazlea[30];
int orri_kop, salneurria;
char arg_etxea[30];
} lib;
```

ANSI C-n edozein egitura hasiera daitekeen artean, K&R eta konpiladore zaharretako C-n bakarrik egitura globalak eta *static* erakoak hasiera daitezke. Hasieraketa honela egingo da:

```
struct liburu lib={"C lengoaia", "XXX, YYY", 230, 2000,
          "Elhuyar"};
```

7.3. EGITUREN ERABILPENA

C lengoaiaz egituren eremuak erreferentziatzeko • (puntu) eta –> eragileak erabiliko dira. Puntua egitura-aldagaiekin zehazten den bitartean, gezia egitura-erakusle baten ondoren erabiltzen da eta ondoko pasartean aztertuko da. Puntu eragilearen bidez osagai bat erreferentziatzeko honakoa egiten da:

```
egitura_aldagaia.osagai_izena
```

Aurreko atalean egindako definizioan lib aldagaiaren —konturatu *lib* aldagaia dela eta *liburu* egitura baten izena— salneurria eremua aldatu nahi badugu ondokoa egin daiteke:

```
lib.salneurria = 2200;
```

7.1 programan liburutegi bateko fitxategiaren erregistroa definitu eta erabiltzen dugu.

```
#include <stdio.h>

#include "liburu.h"
/* liburu egitura definitutzat hartzen da */

/* prezio-tarte bateko liburuak lortzen dituen funtzioa */
print_lib (min, max)
int min, max; /* prezio-tarterako */
{
struct liburu lib; /*liburu-motako aldagaia*/
int kod_buk;

kod_buk = irakur_lib (& lib);
while (kod_buk > = 0)
   {
   if ((lib.salneurria <= max) && (lib.salneurria >=min))
      printf ("%d %s %s %d \n", lib.kod, lib.izenburua,
        lib.idazlea, lib.salneurria);
   kod_buk = irakur_lib (&lib);
   /*geroago programatzeko*/
   }
```

*7.1 **programa**. Liburu egitura inprimatzen duen funtzioa.*

Bertan ikus daitekeenez egituraren definizioan eremuen izenak eta motak agertzen dira **{** eta **}** artean, `struct` eta izena aurretik eta **;** atzetik azaldu behar delarik. Erabilpenean aldiz, hasieran mota horretako aldagaia definitutakoan eremuak adierazteko aldagaia, • eragilea eta eremuaren izena zehazten dira.

Egitura-mota bereko bi aldagai badaude haien arteko esleipena konpiladore batzuekin asignazio bakarrez posible den bitartean, beste batzuekin eremuz eremu burutu behar da. Dena den, taulekin gertatzen zenaren kontra, egitura motako aldagai baten izenak bere balioa adierazten du eta ez erreferentzia. Beraz, egitura bat parametro gisa zehaztean —erreferentziaz egitea gomendatzen da beti eraginkortasuna dela eta— **&** zehaztuko zaio aurretik, eta trukatuko dena bere erakuslea izango da.

7.4. EGITURAK, TAULAK ETA ERAKUSLEAK

Egiturak tauletan biltzea oso erabilpen ohizkoa da. Adibidez, liburutegiaren liburu guztien informazioa memoriako taula batean sar daiteke.

Egiturak eta erakusleak konbinatzea oso arrunta da. Batetik, egitura baten osagaia erakusle bat izan daiteke, eta bestetik, aurretik esan dugunez, egituren erakusleak erabiltzen dira; funtzio-deietan batez ere.

Mota hauetako erazagupenak segidan agertzen dira:

```
struct liburu lib[100]; /* egituren taularen erazagupena */
struct liburu *era_lib; /* egitura baten erakuslearen
                           erazagupena */
```

Lehenengo kasuan liburu baten eremu bat aipatzeko taula, indizea eta eremua zehaztu behar da, eremuaren aurrean puntu eragilea zehaztuz.

Bigarren kasuaren adibide gisa pentsa dezagun aipatutako egitura idazteko, liburutegiarena hain zuzen, funtzio bat dugula eta parametro gisa *liburu* egituraren erreferentzia. Funtzio hori 2. programan definitzen da, eta • eragilea erabili beharrean -> eragilea erabiliko da.

```
#include <stdio.h>

void print_lib_bat (plib)
struct liburu *plib;
{
printf ("%d %s %s %d \n", plib -> kod,
     plib -> izenburua, plib -> idazlea,
     plib -> salneurria);
}
```

*7.2 **programa**.* `liburu` *egitura inprimatzen duen funtzioa.*

Gezi eragile berria puntuaren eboluzioa da, ondorengo bi espresioak zeharo baliokideak baitira —baina bigarrena erosoago lehena baino.

```
(* egitura_erakusle).eremua
egitura_erakusle->eremua
```

7.5. EGITURA KABIATUAK

Egitura baten osagai bat beste egitura bat izan daiteke. Kasu honetan, osagaia den egitura bestean kabiatuta dagoela esaten da. Adibidez, *liburu* egitura definiturik badago, honako egitura eta aldagaia erazagut daitezke:

```
struct lana
{
struct liburu lib;
int urtea;
};
struct lana lan1;
```

lana egiturak bi osagai ditu, bata *egitura* motakoa eta bestea osokoa, liburua idatzi zeneko urtea, alegia. Honela erabil daiteke aldagaia:

```
lan1.urtea=1994;
strcpy(lan1.lib.izenburua, "C lengoaia"):
```

Egituraren osagaiak erreferentziatzeko ordena kanpokoenetik barruragokoenera doa. Teorikoki, egiturak edozein maila arte kabia daitezke, baina konpiladore gehienek kopurua mugatua dute.

Egituren erakusleak erabiliz lor daiteke egitura batean egitura bereko beste aldagai bat erreferentziatzea. Hau oso erabilia da zerrendak, ilarak, pilak, eta antzeko datu-motak definitu eta erabiltzeko, batez ere tratamendu errekurtsiboei begira.

7.3 programan string-en arbola bitarra definitzeko erabiltzen den egitura eta funtzioak azaltzen dira. Funtzioak arbola ezkerretik eskuinera idazten du, algoritmo errekurtsiboa erabiliz.

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

#define MAX 20
#define NMAX 30

struct nodo
{
char izen [MAX];
struct nodo *ezker, *eskuin;
};

struct nodo arbola [NMAX]; /* egituren taula*/

void print_arbola (pnodo)
struct nodo *pnodo;
{
if (pnodo!=NULL)
   {
   print_arbola (pnodo->ezker);
   printf ("%s\n", pnodo->izen);
   print_arbola (pnodo->eskuin);
   }
}
```

***7.3 programa**. Egitura errekurtsiboaren adibidea.*

7.6. EGITURAK FUNTZIOEN ARGUMENTU

Hasierako K&R C-k ez zuen uzten funtzio batek egitura bat itzul zezan, ezta egituren arteko esleipenak egin zitezen ere. ANSI C-k egituren arteko esleipenak onartzen ditu.

Egitura bat parametro gisa erabiltzean, argumentu formalaren eta pasatzen den parametroaren motek bateragarriak edo berdina izan behar dute. Hau egi-

teko modu normala egitura orokorki definitzea da, eta gero bere izenak aldagaiak eta parametroak erazagutzeko balio izango du, 7.4 programan ikusiko den bezala. Honela mota berekoak direla ziurtatzen da.

```
#include <stdio.h>

/* egitura-mota bat definitu */
struct data_mota
{
int eguna;
char hilabetea[15];
int urtea;
};

inpri_urte_hilabetea(ppara)
struct data_mota *ppara;
{
printf("hilabetea: %dko %s\n",ppara->urtea,ppara->hilabetea);
}

main()
{
struct data_mota data; /*arg erazagutu */

data.urtea=1995;
strcpy(data.hilabetea,"Iraila");
inpri_urte_hilabetea(&data);
}
```

7.4 programa. *Egituraren erazagupen orokorra.*

Bertan *data-mota* egitura definitzen da, programa nagusian mota horretako *data* aldagaia definituz, eta azpiprograman mota honetarako *ppara* erakuslea.

7.5 irudian liburutegi baten inguruko aplikazio txiki bat erakusten da, bertan aurretik azaldutako adibide batzuk erabiltzen direlarik.

```
#include <stdio.h>

/* egituraren definizioa*/
struct liburu
{
int kod;
char izenburua [30], idazlea [30];
int orri_kop, salneurria;
char arg_etxea [30];
};

/* liburu bati buruzko datuak hartzeko funtzioa */
int irakur_lib(plib)
struct liburu *plib;
{
char aux2[30];
int aux1;

printf("\n\nliburuaren kodea:\n");
scanf("%d", &aux1);
plib->kod=aux1;
if (plib->kod>=0)
   {
   printf("liburuaren idazlea:\n");
   scanf("%s", aux2);
   strcpy(plib->idazlea, aux2);
   printf("liburuaren argitaletxea:\n");
   scanf("%s", aux2);
   strcpy(plib->arg_etxea, aux2);
   printf("liburuaren orri-kopurua:\n");
   scanf("%d", &aux1);
   plib->orri_kop=aux1;
   printf("liburuaren izenburua:\n");
   scanf("%s", aux2);
   strcpy(plib->izenburua, aux2);
   printf("liburuaren salneurria:\n");
   scanf("%d", &aux1);
   plib->salneurria=aux1;
   }
return(plib->kod);
}
```

```
/* prezio-tarte batean dauden liburuen zerrenda lortzen duen
   funtzioa */
extern print_lib (int min, int max);/*aurretik definiturik*/
int min, max; /* prezio-tarterako */
{
struct liburu lib; /*liburu-motako aldagaia*/
int kod_buk;

kod_buk = irakur_lib (&lib);
while (kod_buk >= 0)
   {
   if ((lib.salneurria <= max) && (lib.salneurria >=min))
     printf("%d %s %s %d \n", lib.kod,lib.izenburua,
       lib.idazlea, lib.salneurria);
   kod_buk = irakur_lib (&lib);
   }
}
/* programa nagusia */
main()
{
int handi, txiki, aux;

printf("Sakatu kontutan edukiko den tartearen mugak:");
scanf("%d %d", &txiki, &handi);
if (txiki>handi)
   {
   aux=txiki;
   txiki=handi;
   handi=aux;
   }
print_lib(txiki, handi);
}
```

7.5 programa. Egituren definizioa eta erabilpena.

7.7. BIT-EREMUAK

Programazio-lengoaia gehienek aukera ematen ez duten artean, C-k byte baten barruko bit bat atzitzea onartzen du. Hau interesgarria izan daiteke ondoko kasuetan:

- Memoria oso mugatua dagoenean edo aurreratu nahi denean, byte batean zenbait aldagai boolear (true edo false) gorde daitezke.

- Hainbat gailuren interfazeetarako bytearen barruko bitetan kodetutako informazioa maneiatu behar delako.

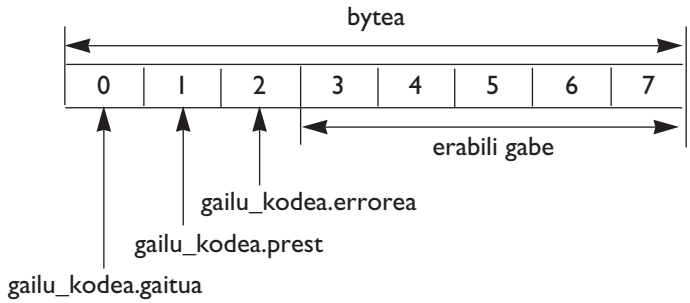
- Zifratzeko errutina batzuek eta antzekoek bytearen barruko bitak atzitu behar dituztenean.

Kasu hauetan guztietan byteak eta bit-maneiurako eragiketak erabil badaitezke ere, bit-eremuek egituraketa eta kodearen eraginkortasuna eta ulergarritasuna hobetzen dute, garraiagarriago eginez. C-ren bitak atzitzeko era egituretan oinarritzen da: bit-eremuak bere tamaina bitetan adierazten duen egiturak dira. Bit-eremu bat definitzeko era orokorra honakoa da:

```
struct egitura_izena
{
mota izena1: bit-luzera;
mota izena2: bit-luzera;
...
}
```

Bit-eremuak *int*, *unsigned* edo *signed* bezala erazagutuko dira. Bateko luzera duten eremuek *unsigned* bezala erazagutuko dira, bit bakarrak zeinurik ezin duelako eduki. Halaber, konpiladore batzuek *unsigned* motako bit-eremuak baino ez dituzte onartzen. Bit-eremu bakoitza ez dago izendatu beharrik. Erabili gabeko bitak izendatu gabe utziz, benetan erabiliko direnak atzi daitezke. 7.1 irudian ikus daiteke ondoko bit-egituraren definizioari dagokion memori hartzea:

```
struct gailu
{
unsigned gaitua:1;
unsigned prest:1;
unsigned errorea:1;
} ;
struct gailu gailu_kodea;
```



7.1 irudia. `gailu_kodea` *nola gordetzen den memorian.*

Egiturak diren aldetik bit-eremuak atzitzeko puntu eragilea erabiliko da, egitura erakusle baten bidez erreferentziatzen ez bada, azken kasu honetan -> eragilea erabiliko delarik.

7.6 programako *gai_idatzi* errutinak bit-eremuak nola erabil daitezkeen argitzeko balio du. *lortu* errutinak gailuaren egoera itzuliko du eta `gailuan_idatzi` errutinak idazketa fisikoa.

```
#include <stdio.h>

struct gailu
{
unsigned gaitua:1;
unsigned prest:1;
unsigned errorea:1;
} gailu_kodea;

void lortu(kode)
struct gailu *kode;
{
printf("Errutina honek gailuak duen kodea lortzen du\n");
switch (kode->prest)
   {
   case 1:kode->gaitua=0;
          kode->prest=0;
          break;
   case 0:kode->prest=1;
          break;
   }
}
```

```
void gailuan_idatzi(kar)
char kar;
{
printf("%c karakterearen idazketa fisikoa egikaritzen da\n",
   kar);
gailu_kodea.prest=0;
gailu_kodea.gaitua=1;
}

void gai_idatzi(k)
char k;
{
while (!gailu_kodea.prest)
   lortu(&gailu_kodea); /* itxaron */
gailuan_idatzi(k); /* idazketa fisikoa */
while (gailu_kodea.gaitua)
   lortu(&gailu_kodea);
   /* idazketa fisikoa burutu arte itxaron */
if (gailu_kodea.errorea)
   printf("Idazketakoan errorea!\n");
}

main()
{
char karak;

gailu_kodea.prest=0;
gailu_kodea.gaitua=0;
gailu_kodea.errorea=0;
printf("Sakatu idatzi nahi den lehen karakterea\n");
scanf("%c", &karak);
gai_idatzi(karak);
gailu_kodea.errorea=1;
printf("Sakatu idatzi nahi den bigarren karakterea\n");
scanf("%c", &karak);
gai_idatzi(karak);
}
```

7.6 programa. bit-eremuak erabiltzen.

Bit-eremuek honako mugak dituzte:

1. Ezin da bit-eremu baten aldagaiaren helbidearekin lan egin.

2. Bit-eremuak ezin dira tauletan antolatu.

3. Ezin da bat bestearen gainean teilakatu.

4. Ezin da jakin ea bitak eskuinetik ezkerrera edo ezkerretik eskuinera egongo diren.

Amaitzeko, egituren osagai normalak eta bit-eremuen osagaiak nahas daitezke, hurrengo sententzia horren adibidea izanik:

```
struct langile
{
struct hel helbide;
float soldata;
unsigned lanean:1;
unsigned orduko:1;
unsigned murrizketak:3;
};
```

Horrela, azken hiru eremuak byte bakarrean gordeko dira, beste modutan hiru byte beharko lituzkeenak.

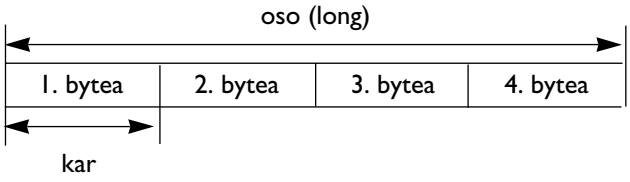
7.8. BILDURAK

Bildurak definitzeko eta erabiltzeko egituretan aztertutako eragile berberak (. ->) erabili arren, datu-mota honen helburua zeharo desberdina da, zeren eta bietan deskribatutako eremuak mota desberdinekoak izan badaitezke ere, bilduretan eremu horiek baitira eremu bakar bati dagokion motaren interpretazio desberdinak, egituretan multzo baten osagaiak diren bitartean. Beraz, bildura motako datu bat irakurtzean bilduran definitutako eremuen artean bat baino ezingo da erabili. Mota hau gutxitan erabili arren, fitxategi batetik irakurritakoa datu-mota desberdin gisa tratatzeko memoria gutxiago gastatzea da duen erabilpen hedatuena. Adibidez, 7.7. programan 80 byteko memori

zati bat inprimatzen da, baina kode baten arabera 80 karaktere dira edo 20 zenbaki oso.

Bildurak definitzeko eta erazagutzeko sintaxia egiturenaren antzerakoa da, `struct` hitzaren ordez `union` agertuko delarik. Bildura bat definitzean, konpiladoreak bilduraren mota handiena gordetzeko gai den aldagaiarentzat erreserbatuko du memoria. Beraz, ondoko bildura definituta badago eta bilduraren hasierako helbidea ezaguna denez, 7.2 irudian azaltzen dira bilduraren interpretazio posibleak.

```
union bildura
{
char kar;
long oso;
} alda;
```



7.2 irudia. `bildura`*ren interpretazio posibleak.*

7.7 programan ikusten denez, bildurak erakusle eta taulekin konbina daitezke. Horrela bildura baten erakuslea bidali egin da funtziora eta bilduraren eremuak taulak dira. Bildurak egiturekin ere konbina daitezke; posible baita egitura baten barruan bilduraren bat egotea, eta alderantziz, bilduraren barruan egiturak egotea.

```
#include <stdio.h>

union kar_zen
{
char kate [80];
long zenb [20];
};

print_bildu (p, kod)
union kar_zen *p; /*bilduraren erakuslea*/
int kod;
```

```
{
int i;

if (kod ==0)
   for (i = 0; i < 80; i++)
      printf ("%c", p->kate[i]);
else
   for (i = 0; i < 20; i++)
      printf ("%ld\t", p->zenb[i]);
}

void sar_bildu(bil, kod)
union kar_zen *bil;
int kod;
{
int i;

if (kod==0)
   {
   printf("Sakatu 80 karaktere:\n");
   for (i=0; i<80; i++)
      {
      bil->kate[i]='\0';
      scanf("%c", &(bil->kate[i]));
      }
   }
else
   {
   printf("Sakatu 20 mega-zenbaki:\n");
   for (i=0; i<20; i++)
      {
      bil->zenb[i]=0L;
      scanf("%li", &(bil->zenb[i]));
      }
   }
}

main()
{
int kodea;
union kar_zen bildu;

kodea=0;
sar_bildu(&bildu, kodea);
printf("\n");
print_bildu(&bildu, kodea);
kodea++;
print_bildu(&bildu, kodea);
sar_bildu(&bildu, kodea);
print_bildu(&bildu, kodea);
}
```

7.7 ***programa.*** *Bildura baten erabilera.*

ANSI C-k bildurak hasieratzea onartzen du, K&Rkoak aldiz, ez. ANSI C-n, bildurak hasieratzeko modu bakarra bere lehen osagaiaren motako balio batekin lortzen da. Adibidez ondoko hasieraketa egin daiteke:

```
union kar_zen
{
char kate [80];
long zenb [20];
};

union kar_zen zerrenda={"Hau adibidea da"};
```

7.9. ENUMERATUAK

Datu-mota hau, aldagai baten balio posiblea multzo labur baten osagaietako bat denean erabiltzen da. Mota honetan egiten dena, karaktere edo zenbaki osoez egin daiteke (balio bakoitzari kode bat emanez), baina programa irakurterraza izan dadin, oso datu-mota aproposa gertatzen da.

Enumeratuak definitzeko aurreko egituretarako erabilitako formato bera erabiltzen da baina *enum* gako-hitza erabiliz *struct* edo *union* erabili beharrean.

Enumeratuen zerrendako osagai bakoitza osoko zenbaki bat dela pentsa daiteke. Bestelakorik esaten ez bada, lehenengo osagaiaren balioa 0 da, bigarrenarena 1, hirugarrenarena 2, eta abar. Horren ondorioz zerrendako elementuak erabil daitezke dagokion espresioan egiturari erreferentziarik egin gabe; eta beraz, puntu edo bestelako eragile berezirik erabili gabe ere bai. Hona hemen adibide bat:

```
enum aste_eg {astele, astear, asteaz, ostegu, ostira,
              larunb, igande};
enum aste_eg eguna;
eguna=asteaz;
if (eguna==ostira)
     printf("Ostirala da!!!\n");
```

Osagaien balioak alda daitezke hasieraketen bidez. Hau lortzeko modua = zeinua eta ondoren osoko balio bat jarriz emango da. Hasieraketa erabiltzen bada, hurrengo osagaiei hasieratutako balioa baino handiagoak diren balioak emango zaizkie. Adibidez, honakoa egiten bada:

```
enum aste_eg {astele, astear, asteaz, ostegu, ostira=50,
              larunb, igande};
```

ondorio zera da: *ostira* konstanteari 50 balioa esleitzeaz gain *larunb* eta *igande* konstanteei dagozkien balioak 51 eta 52 izango dira.

7.8 programan bere aplikazio bat ikus daiteke, asteko egunaren arabera tarifa desberdina aplikatzen duen funtzio batekin:

```
#include <stdio.h>

enum aste_eg {astele, astear, asteaz, ostegu, ostira,
              larunb, igande};

long tarifa (eguna, tar1, tar2)
enum aste_eg eguna;
long tar1, tar2;
{
switch (eguna)
   {
   case astele:
   case astear:
   case asteaz:
   case ostegu:
   case ostira: return (tar1);
   default: return (tar2);
   }
}


main()
{
enum aste_eg noiz;
long merke, garesti, zenbat;
```

```
merke=50;
garesti=200;

for (noiz=astele;noiz<=igande; noiz++)
   {
   zenbat=tarifa(noiz, merke, garesti);
   printf("\nOrdaintzen dena %ld da.\n", zenbat);
   }
}
```

***7.8 programa**. Enumeratuak erabiltzen dituen programa.*

7.10. SINONIMOAK

Aurrekoan esan genuenez, C-k ez du datu-mota berriak definitzeko ahalmenik, baina hala ere antzeko zerbait egin daiteke *typedef* eragileak sinonimoak definitzeko ematen duen aukeraz.

Metodo honen bidez programek irakurgarritasuna hobetzen dute, baina ez da benetako datu-mota berria sortzen, ondorio bakarra konpilazio-garaian string-en itzulpen soila baita (makroen kasuan bezala).

Horrela, adibidez, datu-mota boolearra defini daiteke, 7.9 programan ikus daitekeenez:

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0
typedef short bool; /* bool short-en sinonimoa da */

bool eta (a,b)
bool a, b;
{
return (a&b);
}
```

***7.9 programa**. `typedef`-aren erabilpena mota boolearra simulatzeko.*

7.11. ERAZAGUPEN KONPLEXUAK

C-z idatzitako programa batzuetan erazagupen konplexuak agertzen dira, deklaratutako datua zer den jakitea zaila gertatuz. Horregatik, ondorengo sententzian azaltzen den erroreak aldagaiaren erazagupena ez da berehala ulertzen.

```
void (* erroreak [2]) ();
```

Zailtasunaren arrazoi nagusia honetan datza: * eragilea aurretik idazten den bitartean [ ] eta ( ) atzetik idazten dira, taula eta funtzioaren eragileek lehentasun handiagoa dute erakuslearenek baino, eta elkarketa ezkerretik eskuinera da [ ] eta ( ) eragileen kasuan, baina alderantziz erakusleenean.

Definizioa osatzeko ondoko erregelei jarraitzea da errazena:

1) ezkerretatik hasi eta izenean bukatu.

2) lehentasun-erregelak alderantziz aplikatuz, ondoko itzulpena egin:
   - `( )` ... itzultzen duen funtzio
   - `[ ]` ...-z osatutako taula bat
   - `*` ...-rako erakusle

Horrela *erroreak*-en definizioa horrela irakur daiteke: ezer itzultzen ez duten funtzio-etarako erakusle-ez osatutako taula (2 osagaiekin) da. Beste adibideak azter ditzagun:

```
char * x [ ];
```

karaktere-etarako erakusle*z osatutako taula bat da x*.

```
struct S (* (* (* x) [ ]) ( )) [ ];
```

S egitura-z osatutako taula-*rako e*rakuslea itzultzen duen funtzio-rako erakusle-ez osatutako taula bat*erako erakuslea da x*

8. ALDAGAIEN EZAUGARRIAK

Aurreko kapituluetan C lengoaiak maneiatzen dituen aldagaiak eta dagozkien datu-motak aztertu baditugu ere, aldagaiei dagozkien zenbait ezaugarri aztertzeke daude, garrantzitsuenak iraupena eta esparrua (*scope*) direlarik. Definizioaren kokapena eta `auto`, `static`, `extern` eta `register` gako-hitzak konbinatuz C-ren biltegiratze-mota desberdinak bereizten dira.

Gogora ditzagun aldagai globalen eta lokalen arteko desberdintasunak. Aldagai globalak programaren edozein ataletan erabil daitezke (hau da, edozein lekutatik atzitu eta aldatzea gerta daiteke), memoriaren helbide finko batean kokatuak daudelarik, programaren exekuzioaren hasieratik amaieraraino. Aldagai lokalak, aldiz, programaren atal batzuetan (adibidez, funtzio baten barnean) baino ez dira ezagutzen eta bertan baino ezin dira erabili. Aldagai lokalak sortzen eta desagertzen joan daitezke programa exekutatzen den bitartean. Gainera, bloke baten barnean aldagai global baten izena duen aldagai lokal bat definitzen bada, bloke horretan izen hori azaltzen den guztietan aldagai lokalaz ari dela azpimarratu behar da.

Ahalik eta aldagai global gutxien erabili behar dira, alde batetik memoria hartzen baitute programaren exekuzio-denbora guztian zehar (eta ez bakarrik funtzioa aktiboa denean); bestetik, funtzio orokorrak egin nahi badira aldagai lokalek gainontzeko moduluekiko independentzia ematen dute; eta azkenik, aldagai globalak erabiltzen direnean albo-ondorioak direla eta, errore ezezagun eta detektatzeko zailak gertatu ohi dira. Hala eta guztiz ere batzuetan egokia izan daiteke aldagai globalen bat erabiltzea ondoko bi arrazoiengatik: abiadura batetik, errutinak aktibatzean bertako aldagaiak sortzea eta zenbait

parametro kopiatzea saihets daitekeelako; eta erosotasuna bestetik, parametro-truke asko saihets baitaiteke kodearen idazketa laburtuz.

8.1. IRAUPENA: FINKOAK VERSUS AUTOMATIKOAK

Aldagai *finko* bat programaren hasieran sortzen da eta programaren exekuzioa amaitu arte *bizirik* irauten du, bere memori zatia etengabe hartzen edo mantentzen duelarik. Aldagai *automatiko* bat aldiz, programaren funtzio edo bloke zehatz batekin lotuta dago. Beraz, funtzio edo bloke hori exekutatzen hasten denean baino ez da sortuko, eta blokearen exekuzioa bukatzean desagertuko da. Horren ondorioz, aldagai automatiko bat behin baino gehiagotan sor daiteke, agian memoria, zati ezberdinak hartuz aldi bakoitzean; eta aldagaiari dagokion bloke edo funtzioa exekuzio batean erabiltzen ez bada, aldagaia ez da sortuko.

Aurrekoak eragin handia du hasieraketan, zeren finkoek hasieraketa bakarra duten bitartean, automatikoak aldiz, funtzioa edo blokea kanpotik erreferentziatzen den bakoitzean hasieratuko baitira. Gainera, aldagai automatikoen hasieraketetan edozein espresio azal daitekeen artean (espresio horretako aldagai guztiak aurretik definituta badaude), aldagai finkoen hasieraketetan espresio konstanteak baino ezin dira azaldu (hau da, ezin dira aldagaien izenak azaldu).

Halaber, aldagai automatikoak hasieratzea beharrezkoa da, ezezagunak diren balioak har ez ditzaten nahi bada behintzat. Aldagai finkoei aldiz, sistemak berak balioak eman ohi dizkie (adibidez, zenbakizkoak badira, 0 balioa emango die). Geroxeago ikusiko dugunez, aldagai globalak finkoak izango dira eta lokalak automatikoak.

8.2. ESPARRUA

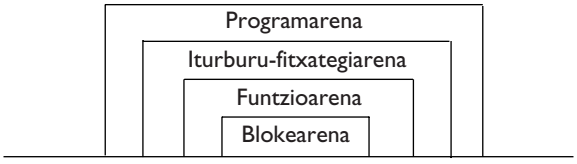
Esparrua edo *scope*-a iturburu-programari dagokion kontzeptua da eta aldagai baten izenaren ezagutza-eremua adierazten du. C lengoaian ondoko lau ezagutza-eremu edo esparru-motak bereizten dira:

1) programarena: aldagaiaren izena globala da; beraz, edozein funtzio, bloke edo agindutatik erreferentzia daiteke.

2) iturburu-fitxategiarena: aldagaiaren izena definitzen den iturburu-fitxategian baino ez da ezagutzen. Beraz, konpilazio banatua erabiltzen denean beste modulu batetik erreferentziatzen bada ez da ezagutuko, beste aldagaitzat hartuko baitu konpiladoreak.

3) funtzioarena: aldagaiaren izena funtzio edo azpiprograman bertan baino ez da ezagutzen.

4) blokearena: bloke bat "`{`" eta dagozkion "`}`" artean dagoena da eta aldagaia definitzen den blokean baino ez da ezagutzen, honela albo-ondorioak saihesten direlarik.



8.1. iruđia. *Esparruen arteko erlazioak.*

Aldagaiaren esparrua non erazagutzen denaren araberakoa da, `static` gako-hitzak alda badezake ere. Funtzioaren esparrua duten bakarrak `goto` sententziaren etiketak baino ez dira.

Adibide batzuk ikusiko ditugu aurrekoa argitzeko asmoz:

A)
```
int funtz1( )                    int funtz2()
  {                              {
  int i;                         int i;
  ...                            ...
  }                              }
```

- Bi aldagai automatiko hauek (`i` izenekoak) desberdinak dira, blokearen esparrua baitute.

- Arrazoia definizioen kokapenean datza, funtzio barruan egotean aldagaia, besterik esaten ez den bitartean, automatiko baita eta bere esparrua blokearena.

B) 8.1 programako j aldagaia.

```
#include <stdio.h>

int j = 10; /* programan ezaguna */

int f1(void)
{
int j=0; /* blokean ezaguna */

for (;j<3;j++)
  printf("j: %d \t",j);
}

main ()
{
f1();
printf ("J: %d",j);
}
```

8.1. programa. *Izen bereko bi aldagaien erabilpena.*

- `j` izeneko bi aldagai ditugu, bata globala, kanpoan definitutakoa, (finkoa) eta bestea lokala, `f1` funtzioaren barruan erazagututakoa.

- Exekuzioaren ondorioa ondokoa litzateke:

```
j: 0      j: 1      j: 2    J: 10
```

Lehenengo hiru emaitzak `f1` funtzioaren exekuzioaren eragina baitira (hasieraketa 0 da besterik ez zehaztean) eta azkena programa nagusiarena.

8.3. BILTEGIRATZE-MOTAK

Goi-mailako lengoaia batzuetan bi biltegiratze-mota bereizten dira: globala eta lokala. C-n aldagaien iraupena eta esparrua konbina daitezke, biltegiratze--mota desberdinak lortuz, horretarako aldagaiaren definizioaren kokapena eta `extern`, `static`, `auto` eta `register` gako-hitz edo adierazleen erabilpena kontutan hartuko direlarik.

Ikus ditzagun adierazle hauen esanahia:

`extern`: aldagai global bat definituta dagoen modulutik kanpo erabili nahi denean, aldagaia berriro erazagutzen da, baina `extern` adierazlea aurretik zehaztuta, definizioa beste nonbait dagoela adieraziz.

`static`: funtzio bikoitza du. Kanpoan definitutako aldagaiei (globalei) esparrua edo ikustaldea murrizten die programatik fitxategira. Funtzioen barruan definitutakoei aldiz, iraupena aldatzen die automatikoak finko bihurtuz.

`auto`: bloke-esparrua duten aldagaiei egokitzen zaie baina funtzioen barruan definitutakoei. Aldagai hauek, adierazlerik egokitzen ez bazaie, `auto` adierazleak eragindako ezaugarri berberak hartzen dituztenez, adierazle hau ez da ia inoiz erabiltzen.

`register`: bloke-esparrua duten aldagaiekin baino ezin da erabili adierazle hau, eta duen eragina exekuzio-denboran aldagaia Prozesatzeko Unitate Zentralaren barnean dagoen erregistro orokor batean

kokatzea da. Normalki, `int` eta `char` motako aldagaiekin erabiliko da. Horren bidez, programaren exekuzioa azkar daiteke, asko erabiltzen diren aldagaiei egokituz gero, baina gerta daiteke konpiladoreak adierazle honi kasurik ez egitea. Aipatzekoa da mota honetako aldagaien erakusleak ezin direla erabili, ez baitagokie helbiderik.

8.2. irudian gako-hitz horiei guztiei dagozkien esparru eta bizi-iraupena laburtzen da.

DEFINIZIOAREN KOKAPENA	–	*extern*	*static*	*auto edo register*
KANPOAN	iraupena: finkoa esparrua: programa	iraupena: finkoa esparrua: programa	iraupena: finkoa esparrua: fitxategia	_____
BARRUAN	iraupena: automatikoa esparrua: blokea	iraupena: finkoa esparrua: programa	iraupena: finkoa esparrua: blokea	iraupena: automatikoa esparrua: blokea
FUNTZIO ARGUMENTU BEZALA	iraupena: automatikoa esparrua: blokea	_____	_____	iraupena: automatikoa esparrua: blokea

8.2. irudia. *Aldagaien iraupena eta esparrua adierazlearen arabera.*

8.4. EZAUGARRIEN ERABILPENA

Ezaugarri guztiak aztertu ondoren, ikus dezagun azaldutako aldagai-mota desberdinek nolako erabilpena duten. Ondoko lau erabilpen desberdin hauek bereiz ditzakegu:

1) **aldagai lokal automatikoak**: gehien erabiltzen direnak dira, blokearen esparrua eta iraupen automatikoa baitute. Definizioa funtzioen barruan egiten da, inolako adierazlerik gabe edo `register` adierazlearekin.

2) **aldagai lokal finkoa**: funtzio edo bloke bakar batean erabiltzen den aldagairen bat mota honetakoa izango da balioa mantendu behar baldin badu

dei batetik bestera. Definizioa aurrekoa bezala egiten da, baina `static` gako-hitza erabiliz. Hauen erabilpenaren adibide bat 7.2. programan ikus daiteke; bertan hilabeteen izena gordetzen duen taula lokala den bitartean, —beste funtzioetatik ez baita ikusi behar— eraginkortasuna dela eta, dei bakoitzean taula sortzea galarazten da.

```
#include <stdio.h>

char * hil_izena(int n)
{
static char *izenak[13]=
   {"errore","urtarrila","otsaila","martxoa", "apirila",
   "maiatza","ekaina", "uztaila", "abuztua", "iraila",
   "urria", "azaroa", "abendua"};

return((n>0 && n<13) ? izenak[n] : izenak[0]);
}


main()
{
int i;

for (i=0; i<=13; i++)
   printf("%d. hilearen izena %s da.\n", i, hil_izena(i));
}
```

8.2. programa. *Aldagai lokal finkoen erabilera.*

3) **aldagai global arrunta**: funtzio edo bloke desberdinetatik ezagutu eta balioa mantentzea dute ezaugarri nagusia. Definitzean adierazlerik ez da jarri behar, baina funtzioetatik bereiztuta zehazten dira. Dena den, beste modulu edo fitxategi batetik erreferentziatu behar direnean, aldagaiaren aipamena egin behar da, `extern` adierazlea zehaztuz. Funtzio gehienak (funtzioei mota eta ezaugarri bat egokitzen baitzaie), mota honetakoak dira normalean. K&R eta ANSI C artean garraiagarritasuna lortzeko asmoz, `extern` azaltzen den kasuetan hasieraketarik ez egitea gomendatzen da.

4) **aldagai global isolatua**: datu-mota berriak definitzean oso interesgarri gertatzen da aldagaiak datu-mota definitzen duen moduluan dauden funtzioetatik atzigarriak izatea, baina ez beste moduluetatik. Horretarako iraupena finkoa eta fitxategiaren esparrua interesgarria da oso. Definizioa aurrekoa bezala egiten da, baina `static` adierazlea aurretik jarriz. 8.3. programan adibide bat ikus daiteke, bertan definitzen den pila datu-mota berria kanpotik *push*, *pop* eta *clear* funtzioen bidez baino ezin delarik atzitu.

```
#include <stdio.h>

#define MAX 100

static int sp = 0;

static long pila [MAX];

void push(long a)
{
if (sp < MAX)
   {
   pila [sp] = a;
   sp++;
   }
else
   printf ("errorea: pila betea\n");
}

long pop (void)
{
if (sp >0)
   {
   --sp;
   return (pila [sp]);
   }
else
   printf ("errorea: pila hutsa \n");
}

void clear (void)
{
sp = 0;
}
```

8.3. programa. *Datu-mota berri baten definizioa.*

Horietako funtzio bat erabiltzen den moduluan aipamen bat egingo da `extern` adierazlea zehaztuz. Adibidez *push* erabiltzeko: `extern void push();`.

8.5. DEFINIZIOA ETA ERAZAGUPENA

Aurreko kapituluetan esan den bezala, erazagupen batek aldagai baten ezaugarriak (batez ere motari dagozkionak) erazagutzen ditu. Definizio batek, aldiz, aldagai horrentzako memoria erreserbatzen du.

Adibidez, honako definizioak zehazten badira:

```
char karak;
int tau[MUGA];
```

aldagaientzat memoria erreserbatzen da, bide batez iturburu-fitxategi horretarako aldagaiak erazagutuz.

Beste sententzia hauen bidez aldiz,

```
extern char karak;
extern int tau[];
```

iturburu-fitxategiarentzat erazagutzen da `karak char` motakoa eta `tau int` motako taula direla, bigarrenaren tamainaren definizioa eta bien memori erreserba beste nonbait egingo delarik. Azken hau, beraz, erazagupen hutsa da.

Aldagai global batek definizio bakar bat eduki behar du, baina nahi adinako erazagupen onartzen du. Taulen tamainak definizioarekin azaltzea beharrezkoa den artean, ez da gauza bera gertatzen erazagupenetan. Aldagai globalen hasieraketek, baldin badaude, definizioarekin batera joan behar dute.

8.6. BESTELAKO EZAUGARRIAK: *CONST* ETA *VOLATILE*

ANSI estandarreko `const` hitz erreserbatuak ziurtatzen du aldagai bat ez dela aldatuko bere hasieraketaren ondoren. Helburu nagusia irakurketa-soilik motako datuak aldatzen ez direla ziurtatzea da. Horrela, definizio honen ostean:

```
const char kate[20]="Liburu hau ona da!";
```

ezin da aldatu *kate* katearen osagairik. Zentzu horretako sententziaren bat azalduko balitz, konpiladoreak errore-mezu bat azalduko luke.

Erakusle bat ere konstante bezala erazagut daiteke; horrela erakuslearen balioa ezin izango da aldatu, erakusten duen objektua, aldiz, alda daitekeela. Horretarako *const* gako-hitza * sinboloaren ondoren agertuko da definizioan, ondoren egiten den legez:

```
int alda=19;
int *const era=&alda;
```

`volatile` gako-hitza ANSI estandarrekoa da, eta dagozkion aldagaiak konpiladoreak ezagutzen ez dituen erako aldaketak eduki ditzakeela adierazten du. Normalean, hau erabiltzen da memoriako helbide bati egokituak dauden aldagaiekin, gailuen erregistroekin adibidez. Beste modu batean esanda, erazagupen hau erabiliko da programatzaileak idatzitako espresio bat edo sententzi multzo bat adierazitako ordenan exekuta dadin, konpiladoreak kode hori optimizatu gabe (ohizkoa baita konpiladoreek kodea hobetzea), bestela ondorio kaltegarriak gerta baitaitezke.

Bi gako-hitz hauek ezin dira erabili K&R motako C konpiladoreetan, kasu horietan erabiltzailearen sinbolotzat jotzen baitira.

8.7. MEMORIA DINAMIKOA

Orain arte ikusitako aldagaiak estatikoak dira, hau da, zenbat memoria hartzen duten konpiladoreak jakingo du eta ondorioz kokapena aurrikusiko du.

Memoria dinamikoa interesgarria da maneiatzen ditugun datuen luzera exekuzioaren arabera aldakorra denean. Honela, programak bere beharren arabera eskura dezake memoria, behar ez duenean askatuz. Kontuan eduki behar da programa bakoitzari egokitutako memoria mugatua dela eta, beraz, berari dagokiona baino gehiago nahi izanez gero, sistema eragilearekin arazoak gerta daitezke.

Memoria dinamikoarekin lan egitearen abantaila behar den memoria baino gehiago ez erreserbatzea da. Desabantaila, aldiz, konplexutasunaren handitzea da, memoria estatikoarekin konparatuz gero.

Pascalez erakusleak dinamikoak dira, `new` eta `dispose` funtzioen bidez eskatu eta askatzen baitira. C-n badago memoria dinamikoa eskuratu eta libratzeko mekanismo bat, 10. kapituluan azalduko diren `malloc`, `calloc`, `realloc` eta `free` liburutegiko funtzioak erabiltzea hain zuzen.

9. SARRERA/IRTEERA

C-z egin daitekeen sarrera/irteera liburutegi estandarra erabiliz egiten da, helburu horretarako sententzia bereziak ez daude eta. Irizpidearen arabera ondoko sarrera/irteera-motak bereiz daitezke:

1) Fitxategiari begira: estandarra/esplizitua. Sarrera/irteera estandarra erabiltzeko funtzio bereziak daude funtzio orokorrez gain. Estandarra ez den sarrera/irteera erabili nahi izanez gero, esplizituki bere izena aipatu beharko da funtzio orokorrak erabiltzen direnean.

2) Formatuari begira: formaturik gabekoa/formatuduna. Funtzioaren argumentu bezala formatua zehatz badaiteke, funtzioa formatudun bat erabiltzen ari da.

3) Atzipen-motari begira: sekuentziala/zuzenekoa. Datuak bata bestearen ondoren jarraian tratatu nahi badira, atzipena sekuentziala izango da. Bestalde, jarraian dauden datuak bata bestearen ondoren tratatzen ez direnean, zuzeneko atzipena beharko da. Bestelako atzipen-motetarako (atzipen sekuentzial indexatua adib.) liburutegi estandarrak ez ditu funtzioak eskaintzen eta, beraz, liburutegi partikularrak erosi edo programatu beharko dira.

Funtzioak azaltzean beren ezaugarri garrantzitsuenak baino ez dira aipatuko, sakontasun osoan azaltzea oso nekagarria litzatekeelako. Izan ere, funtzio horien guztien zehaztasunak eskuliburuetan agertzen dira, eta UNIX sistemetan kontsulta daitezke *man* komandoaren bidez.

9.1. SARRERA/IRTERRA ESTANDARRA

Orain arte ikusitako *scanf* eta *printf* funtzioek fitxategi estandarrekin lan egiten dute, formatudunak dira eta atzipen sekuentziala egiten dute. Funtzio hauek azalduko dira ondoren, eskaintzen dituzten aukera guztiak zeintzuk diren zehaztuz.

printf funtzioa

Funtzio honek irteera formatuduna egiten du irteera estandarretik. Formatua honako hau da:

```
int printf(char * formatu, arg1, arg2, ...);
```

Funtzio honek, karaktere-katea den lehen argumentuak adierazten duen formatuaren arabera, karaktere bihurtzen eta inprimatzen ditu gainerako argumentuen balioak. Itzulitako balioa inprimatutako karaktere-kopurua izango da.

Formatua zehazten duen karaktere-kateak objektu anitz adierazten du eta bi motakoak izan daitezke: dauden eran kopiatuko diren konstanteak, eta datuen formatuak, bakoitzak *printf*-aren hurrengo argumentuaren bihurketa eta inprimaketa eragiten duena. Formatu-zehaztapen bakoitza % karaktereaz hasi eta bihurketa-mota adierazten duen karaktere batez bukatzen da.

9.1. taulan ikus daitezke bihurketa-karaktere desberdinak.

Askotan aurretik aipatutako bi karaktere hauek baino zehazten ez badira ere, datu bakoitzeko beste karaktere batzuk azal daitezke aipatutako bien artean jarraian aipatzen den ordenan:

- Flag den hauetako karaktere bat: – ezker-justifikazioa behartzeko, + zeinuaren agerpena behartzeko zenbakietan eta # erabilpen berezietarako.

- Luzera minimoa adieraziko duen zenbaki bat.

- Puntu bat, luzera minimoa eta doitasuna ezberdintzeko.

KARAKT.	ARGUM--MOTA[1]	INPRIMATZEKO ERA
`d, i`	`int`	zenbaki hamartarra.
`o`	`int`	zeinurik gabeko zenbaki zortzitarra (hasierako zeroa gabe).
`x, X`	`int`	zeinurik gabeko zenbaki hamaseitarra (hasierako `0x` edo `0X` gabe),
`u`	`unsigned`	zeinurik gabeko zenbaki hamartarra.
`C`	`char`	karaktere bakarra. Esangura txikieneko bytea hartzen da karaktere motako ez diren datuetan.
`s`	`char*`	karaktere-katea inprimatuko du bukaerako `'\0'` edo doitasunean adieraziko karaktere-kopurua aurkitu arte
`f`	`double`	zenbaki hamartar erreala, doitasunean puntu ondorengo zifra-kopurua zehaztuko delarik (balio lehenetsia 6 da)
`e, E`	`double`	idazkera zientifikoa, doitasunean puntu ondorengo zifra-kopurua zehaztuko delarik (balio lehenetsia 6 da)
`g, G`	`double`	`%e` edo `%f` formatuak. Berretzailea -4 baino txikiagoa denean `%e` erabiltzen da, bestela, `%f`
`p`	`void*`	erakuslearen balioa (inplementazioaren menpeko adierazpena)
`%`		ez da argumenturik inprimatuko: `%` karakterea baizik (bestela % inprimatzea ez baitago).

9.1. taula. *printf funtzioan zehatz daitezkeen formatuak*

[1] Argumentu-mota arruntena aipatzen da. `int` aipatzen denean edozein osoko (`long`, `short`, `char`) ere onartzen da.

- Doitasuna izeneko zenbakia. Karaktere-katea batean luzera maximoa adierazten du, koma hamartarraren ostean azalduko den zifra-kopurua koma higikorreko zenbakietan, eta osokoetan zifra-kopuru minimoa.

- osoko zenbakietan `h` bat `short` motakoa edo `l` edo `L` bat `long` motakoa behartzeko datuetan.

Luzera edo doitasuna aldagai batez ordezkatzeko `*` sinboloa zehatz daiteke formatuan. Kasu horretan, balioa kalkulatzen da hurrengo argumentuaren bitartez (derrigorrez `int` motakoa). Adibidez, `kate` katearen gehienez *max* karaktere inprimatzeko honakoa egingo da:

```
printf("%.*s", max, kate);
```

scanf funtzioa

printf-aren ezaugarriak dituen sarrera-funtzioa da *scanf*, formatuak adierazten dituen bihurketetan zenbait aldaketak badaude ere. Formatua ondokoa da:

```
int scanf(char * formatu, arg1, arg2, ...);
```

scanf funtzioak karaktereak irakurtzen ditu sarrera estandarretik, *formatu*-aren arabera interpretatzen ditu, eta emaitzak ondorengo argumentuetan gordetzen ditu. Argumentuen zerrendako guztiek erakusleak izan behar dute, erreferentziaz baino ezin baitira emaitzak jaso funtzioetatik. Argumentu bakoitzari sarrerako datu bat esleituko zaio.

Parekatu edo esleitu diren balioen kopurua izango da emaitza. Horrela, zenbat elementu aurkitu diren jakingo da. Fitxategi-bukaera aurkitzean `EOF` itzuliko du.

Formatua adierazten duen karaktere-katean honako karaktereak aurki daitezke:

- formatu-zehaztapenak. `%` karaktereaz bereizten dira eta irakurtzen diren datuak nola interpretatu adierazten dute. 9.1 taulan adierazitakoak balio du baina "inprimatzeko era" zehazten duen zutabea, "irakurtzen den informazioaren formatua" da kasu honetan. `%s` formatua adierazten denean karaktere-segida bat irakurtzen da, zurigune edo bere baliokide bat aurkitu arte. `%c` formatua karaktere bat edo luzera finkoko karaktere-katea bat irakurtzeko erabil daiteke azken kasu horretan luzera `%` eta `c` karaktereen artean jarriz. `%*s` formatua erabiltzen da sarrerako kateak jauzteko aldagairik esleitu gabe.

- zuriguneak, tabulazioak, lerro-bukaerak, orga-itzulerak, tabulazio bertikalak eta orri-jauziak zehazten badira, karaktere hauek ez dira kontuan hartuko aldagaiei esleipenak egitean. Karaktere hauek guztiak zurigunearen baliokide dira `%s` formatuari begira.

- karaktere normalak (`%` karakterea aurretik ez dutenak), sarreran datozen ez-zurigune karaktereekin parekatzea espero direnak. Parekatzen ez badira errorea itzuliko da.

Beste funtzioak fitxategi estandarren gainean

C-k maneiatzen dituen fitxategi estandarrak hiru dira: *stdin* sarrerarako, *stdout* irteerarako eta *stderr* errore-mezuetarako. Fitxategi hauek sistema eragileak prestatuak (irekiak) ematen dizkio programa nagusiari, honek zuzenean erabil ditzan eta, amaieran, automatikoki itxi egingo ditu. Fitxategi estandarrak teklatua eta pantaila dira besterik ezean, baina sistema eragilearen funtzioen bidez berbidera daitezke. UNIX-en eta DOS-en < eta > karaktereak sarrera eta irteera estandarrak berbideratzeko erabiltzen dira. C-ren liburutegian aipatutako *scanf* eta *printf* funtzioez gain badaude sarrera/irteera fitxategi estandarretan egiteko beste funtzio batzuk, eta garrantzitsuenak honako hauek dira: *getchar*, *putchar*, *gets* eta *puts*. Lauak formaturik gabekoak dira eta lehenengo biek karaktere bakar bat irakurri edo idazten duten

bitartean, besteek lerro oso bat maneiatzen dute. Haien erazagupena `stdio.h` fitxategian dago, gainerako sarrera/irteerako funtzio guztiena bezala, eta hauexek dira dagozkien prototipoak:

```
extern int getchar (void);
extern int putchar (int);
extern char * gets (char *);
extern int puts (char *);
```

9.1. programan *getchar* funtzioa nola erabiltzen den ikus daiteke. Lerro bat irakurri eta lerroak duen luzera kalkulatzen du bertan azaltzen den funtzioa. Ikusten denez *getchar* funtzioak itzultzen duen balioa, *int* motakoa izan arren, irakurritako karakterea da. Bihurketa inplizituak gertatzen dira, karakterea osoko bihurtuz lehenengoz eta osokoa karaktere ondoren.

```
#include <stdio.h>

int irakur_lerro_luz (lerroa)
char lerroa []; /*irakurtzeko bufferra*/
{
int c, i = 0;

do{
   c = getchar ( );
   lerroa [i]= c;
   i++;
   }
while (c!= '\n' && c!= EOF);
lerroa [i] = '\0'; /*string bukaera */
return (i);
}
```

9.1. programa. *getchar funtzioaren erabilpena.*

9.2. programan 9.1ean agertzen zen funtzio bera programatu da baina *gets funtzioa* erabiliz. Bigarren ebazpide honetan arazo bat dago, *gets* funtzioan ezin da karaktere-kopuru maximoa zehaztu eta gerta liteke kateena baino karaktere gehiago irakurtzea, horrek ekar ditzakeen albo-ondorio kaltegarriekin —lehen kasuan hori saihets daiteke kodea osatuz.

```
#include <stdio.h>

int irakur_lerro_luz (lerroa)
char lerroa [];
{
int i ;

gets (lerroa);
for (i = 0; lerroa [i]!= '\0'; i++);
     /*ez du gorputzik edukiko */
return (i+1);
}
```

*9.2. **programa.** gets funtzioaren erabilpena.*

9.2. SARRERA/IRTEERA FITXATEGIAK ERABILIZ

C-rako fitxategi-kontzeptua orokorra da; bertan dispositiboak (teklatua, pantaila, inprimagailua, ...) zein memoria lagungarriko (disko, diskete, zinta, ...) fitxategiak sartzen baitira. Irakurtzen edo idazten den karaktere-bektore bat bezala ikusten baita UNIXek sarrera/irteerako edozein eragiketa.

Beste lengoaiatan bezala, fitxategi bat erabiltzeko fitxategi hori ireki egin behar da, dagokion kontrol-informazioa memorian karga dadin eta sistema eragilearen sarrera/irteerako oinarrizko funtzioek kontsulta dezaten. Kontrol--informazio hori *FILE* motako egitura batean metatzen da (`stdio.h` fitxategian dago definituta) fitxategia irekitzen denean, eta erakusle batez erabiliko dugu, zeren eta fitxategi baten gaineko funtzioak deitzean bere *FILE* egituraren erreferentzia hori argumentu gisa aipatu egingo da.

Beraz, fitxategiak erabiltzeko, C programetan `FILE *` motako aldagai bat definituko da, irekitzean (*fopen* funtzioaz) emaitza jaso dezan eta gainerako sarrera/irteerako funtzioetan parametro gisa erabili ahal izateko.

Ondoren FILE egitura maneiatzen duten sarrera/irteerako funtzio nagusien zerrenda azaltzen da, bertan parametro eta emaitzaren datu-mota zein azalpen motz bat agertzen direlarik. Funtzio horiek erabiltzeko `#include <stdio.h>` sasiagindua jarri beharko da iturburu-fitxategiaren hasieran. Bestela esaten ez den bitartean eragiketa ondo bukatuz gero zero balioa itzuliko dela suposatuko da, eta zero ez den balioa erroreren bat gertatu bada.

```
FILE * fopen(const char *izena, const char *modua)
```

`izena` deituriko fitxategia zabaldu eta berari dagokion `FILE` egituraren erakuslea itzultzen du. Bigarren argumentua fitxategia atzitzeko modua azaltzen duen karaktere-katea da (ikus 9.3 taula). Erroreren bat gertatuko balitz (adibidez, existitzen ez den fitxategia irakurketa moduan zabaldu nahi izatea), `NULL` erakuslea itzuliko du.

```
int fclose(FILE *fp)
```

`fp`-ri egokitutako fitxategia itxi eta `fp` erakuslea eta fitxategiaren arteko erlazioa eten egiten da.

```
int fgetc(FILE *fp)
```

`fp`-ren hurrengo karakterea lortzen du, ondoren uneko posizioa batean inkrementatzen du eta, azkenik, lortutako karakterea itzuliko du `int` motara bihurtuz. Fitxategi-bukaera aurkitzean edo erroreren bat gertatzean, EOF itzuliko du eta `feof` edo `ferror` (ikus 9.4 atala) funtzioak erabili beharko dira zer gertatu den jakin ahal izateko.

```
int getc(FILE *fp)
```

`fgetc` funtzioaren helburu bera du baina funtzioa izan beharrean makroa da.

```
int fputc(int kar, FILE *fp)
```

`kar` karakterea `fp`-ko uneko posizioan idatzi eta, ondoren, posizioa inkrementatu eta idatzitako karakterea itzuliko du. Karakterea `int` bezala pasatu behar zaio, bestela bihurketa inplizitua gertatuko da eta. Erroreren bat gertatuz gero, `EOF` itzuliko du.

```
int putc(int kar, FILE *fp)
```

`fputc` funtzioaren helburu bera du baina funtzioa izan beharrean makroa da.

```
char * fgets(char *buf, int luz, FILE *fp)
```

`fp`-tik karaktere-katea bat irakurtzen du lerro-bukaera edo fitxategi-bukaera aurkitu arte edo `luz-1` osagaiko katea osatu arte. Karaktere-katea horri karaktere nulua (`\0`) gehituko dio bukaeran. Gauzak zuzen badoaz, `buf` erakuslea itzuliko du; bestela, `NULL` erakuslea.

```
int fputs(const char *buf, FILE *fp)
```

`buf` erakusleak adierazten duen katea `fp`-ren barneko bufferrean idazten du, lerro-bukaera aurkitzean irteeran idatz dadin.

```
int fread(void *buf, int osa_luz, int osa_kop, FILE *fp)
```

`fp` fitxategitik `osa_luz` luzerako `osa_kop` osagai irakurriko ditu, emaitza `buf` erakusleak adierazten duen taulan utziz. Funtzio hau bukatuko da irakurketa horien ondoren, edo fitxategi-bukaera azaltzean, edo irakurketa-errore bat gertatzean. Kasu guztietan, irakurritako osagaien kopurua itzuliko du. Hau `osa_kop` baino txikiagoa bada, `feof` edo `ferror` erabili beharko dira zer gertatu den jakiteko. Amaieran, uneko posizioa azken irakurketaren osteko lekuan gelditzuko da.

```
int fwrite(const void *buf, int osa_luz, int osa_kop, FILE *fp)
```

`buf` erakusleak adierazten duen taulatik, `osa_luz` luzerako `osa_kop` osagai idatziko ditu `fp` fitxategian. Ondoren, uneko posizioa azken idazketaren ondoko posizioan geldituko da. Itzuliko duena idatzitako osagaien kopurua izango da (errorerik gertatu ezean `osa_kop`-en balioa)

Aurreko funtzioez gain fitxategien gaineko sarrera/irteera formatuduna bideratzen duten bi funtzioak daude: *fscanf* eta *fprintf* liburutegi estandarreko funtzioak *scanf* eta *printf* funtzioen aldaerak direnak, bertan estandarrak ez diren fitxategiak erreferentziatu ahal izateko.

```
int fscanf(FILE *fp, const char *formatu, ...)
```

`scanf` egiten du fitxategi batetik. Bihurketekin hasi aurretik fitxategi-bukaera edo erroreren bat gertatuz gero, `EOF` itzuliko du. Bestela, bihurtutako eta esleitutako sarrera osagaien kopurua itzuliko du.

```
int fprintf(FILE *fp, const char *formatu, ...)
```

`printf`-aren antzerakoa da, fitxategi batean formatudun datuak idazteko. Idatzitako karaktereen kopurua itzuliko du, edo zenbaki negatibo bat erroreren bat gertatu bada.

Azaldutako funtzio guztiak atzipen sekuentzialean erabiltzen dira, karaktereak (*fgetc*, *fputc*), lerroak (*fgets*, *fputs*), erregistroak (*fread*, *fwrite*) edo datu formatudunak (*fscanf*, *fprintf*). *fp* jarri behar den tokian *stdin* edo *stdout* zehazten bada, funtzioa sarrera edo irteera estandarraren gainean burutuko da. Irakurketak eta idazketak uneko posizioan egiten dira posizio hori *fopen* funtzioaz hasieratu eta atzipen bakoitzean eguneratu egiten delarik.

9.3. programan sarrera/irteerako funtzioak erabiltzen dira, UNIXen erabiltzen den *cat* izeneko programa idazteko. Programa honen bidez argumentu gisa pasatzen diren fitxategiak idazten dira bata bestearen atzean irteera estandarrean. Argumentuak *argc* eta *argv* aldagaien bidez maneiatzen dira, 6. kapituluan ikusi genuen bezala.

```
#include <stdio.h>

void erantsi_fitxat (file)
FILE * file;
{
char c;

do
   {
   c = fgetc (file);
   fputc (c, stdout);
   }
while (c!= EOF);
}

main (argc, argv)
int argc;
char *argv [ ];
{
FILE *fp;
int i;

if(argc < 2)
   erantsi_fitxat(stdin);
else
   for (i=1; i<argc; i++)
      {
      fp = fopen (argv [i], "r");
      if(fp == NULL)
         {
         fprintf (stderr, "errorea irekitzean\n");
         exit (1);
         }
      else
         erantsi_fitxat (fp);
      fclose (fp);
      }
exit (0);
}
```

9.3. programa. *Fitxategien kateaketa burutzen duen cat programa.*

9.3. IREKITZEKO MODUAK

fopen funtzioaren parametroa nolakoa denaren arabera, uneko posizioa aukeratzeaz gain buru daitezkeen eragiketak definitzen dira, 9.2 taulan azaltzen den legez.

modua	onartutako eragiketak	uneko posizioa
"r"	irakurketak	hasiera
"w"	idazketak (fitxategi berria)	hasiera
"a"	idazketak (gehikuntzak)	fitxategiaren bukaera
"r+"	irakurketak+idazketak (eguneratzea)	hasiera
"w+"	irakurketak + idazketak (fitxat. berria)	hasiera
"a+"	irakurketak + idazketak (gehikuntzak)	fitxategiaren bukaera

9.2 taula. fopen funtzioa erabiltzeko moduak.

Aipatu beharra dago ANSI estandarrean datuak testu edo bitar formatuetan atzi daitezkeela. Fitxategi batean gordetzen dena testua dela suposatzen da bestela zehazten ez den bitartean. Horren ondorioz, karaktere batzuk interpretatuak izango dira; adibidez, `'\n'` karakterea orga-itzulera izango da, `'\t'` karakterea, tabulazioa, etab. Kontuz ibili behar da testu-fitxategiekin, karaktere inprimagarriak sistema guztietan era berean interpretatzen diren artean, hainbat kontrol-karaktere inplementazioaren arabera era batera edo bestera interpretatuko baitira. Esan beharra dago fitxategi estandarrak ere testu-fitxategi gisa zabaltzen direla.

Fitxategiak bitarrak definituz gero, aldiz, sistema eragileak ez du aipatutako byteen interpretazio berezirik egingo. Bitak irakurtzen edo idazten dira azaltzen diren eran. Fitxategi bitarrak testuak ez diren datuak gordetzeko eta fitxategiaren edukia dagoen bezala mantentzeko erabiliko dira.

9.2 taulan azaltzen diren moduak testu formatuko fitxategiak zabaltzeko dira. Formatu bitarrean zabaldu nahi izanez gero, moduaren ostean `b` letra gehituko da. Adibidez, fitxategi bitar bat eguneratzeko zabaltzen denean `"rb+"` modua zehaztuko da.

Ondoko bi funtzioek ere irekitzen dute fitxategi bat:

```
FILE *freopen(const char *izena, const char *modua, FILE *fp)
```

`fp`-ri dagokion fitxategia itxi egiten du eta *izena* deitzen den fitxategia zabaltzen du *modua* karaktere-kateak zehazten duen bezala (ikus 8.4. taula). Deia ondo burutuz gero, `fp` itzuliko da eta, bestela, `NULL`. Moduz aldatzeko erabil daiteke.

```
FILE *tmpfile(void)
```

Fitxategi bitar iragankor bat sortzen du eta berari dagokion `FILE *` egitura itzultzen du. Fitxategi hau idazteko moduan zabaltzen da. Fitxategia automatikoki ezabatuko da `fclose` bat egitean edo programa bukatzean. Funtzio honek `NULL` itzuliko du fitxategia ezin bada sortu.

9.4. ERROREEN MANEIUA ETA KONTROLA

Aipatu den bezala sarrera/irteerako funtzio bakoitzak balio berezi bat itzuliko du errore bat gertatzen bada. Batzuek zero balioa itzuliko dute errorea egotean, beste batzuek zeroa ez den balioa eta beste batzuek, `EOF` balioa.

`FILE` egituraren barnean erroreren bat edo fitxategi-bukaera gertatu deneko bi egoerak gordetzen dira. Beraien balioak `feof` edo `ferror` funtzioen bidez atzituko dira. Fitxategi-bukaerako egoera fitxategi-bukaera eta gero dauden datuak irakurtzen saiatzean gertatzen da.

Hauez gain, `errno` izeneko aldagai globala erabiltzen dute sarrera/irteerako funtzio batzuek erroreak gordetzeko. Batez ere UNIX sistemetarako sortutako `stddef.h` fitxategi estandarrean erazagututako `errno` osoko aldagaia gehienbat matematikazko funtzio estandarretan erabiltzen da, eta oso sarrera/irteerako funtzio gutxik erabiltzen dute. `clearerr` funtzioaren bidez aipatutako balioak ezaba daitezke.

Ondoren fitxategien gaineko funtzio laguntzaileen prototipoak eta deskribapen laburra azaltzen da.

```
int ferror(FILE *fp)
```

Aurreko irakurketa edo idazketa batean erroreren bat gertatu bada, zeroa ez den balio bat itzuliko du; bestela, zeroa. Ez du errore-kodea ezabatzen, horretarako `clearerr` funtzioa baitago.

```
int feof(FILE *fp)
```

Aurreko irakurketa edo idazketa batean uneko posizioa fitxategi-bukaeraraino heldu bada, zeroa ez den balio bat itzuliko du; bestela, zeroa. Ez du errore-kodea ezabatzen, horretarako `clearerr` funtzioa baitago.

```
void clearerr(FILE *fp)
```

`fp` fitxategiari egokitutako fitxategi-bukaera edo errore-adierazlea garbitzen ditu. Ez du emaitzarik itzultzen.

`exit` liburutegi estandarreko funtzioa ere errore-maneiurako erabil daiteke. Funtzio honek programaren exekuzioa bukatzen du deitua izatean. `exit` funtzioa zehaztu duen prozesua sortu zuen prozesuak atzi dezake `exit` funtzioaren argumentua `wait` funtzioaren bidez; modu horretan bere menpe da-

goen programa zuzen ala oker bukatu den jakin daiteke. Ohizkoa izaten da zero balioak dena ondo joan dela azaltzea; bestelako balioek zerbait ondo ez dela joan esan nahi dute. `main` funtzioan `return espr` egitea eta `exit(espr)` egitea guztiz baliokideak dira, benetan egiten dena bigarrena da eta.

9.5. ZUZENEKO ATZIPENA

Adibideetan ikusi dugunez, liburutegiko oinarrizko sarrera/irteerako funtzioek (*fgetc*, *fputc*, *fread*, *fwrite*, ...) atzipen sekuentziala bideratzen dute, zeren fitxategia irekitzean hartzen duen uneko posizioa erabili eta eguneratzen baitute.

Zuzeneko atzipena burutzeko aipatutako funtzioak *fseek*-arekin konbinatu behar dira; azken honek posizio jakin batean kokatzeko ahalmena eskaintzen baitu. *fseek* funtzioaren parametroak fitxategiaren deskribatzailea (`FILE *` motakoa), desplazamendua eta modua dira. Hiru modu bereizten dira, bakoitzak desplazamendua aplikatzeko puntu desberdina aukeratzen duelarik, lehenago ikusi bezala. `SEEK_SET` fitxategiaren hasiera aukeratzen du, `SEEK_CUR` uneko posizioa eta `SEEK_END` fitxategiaren bukaera.

`ftell` funtzioarekin uneko posizioa ezagut daiteke. Hau interesgarria izaten da zuzeneko atzipena atzipen sekuentzialarekin konbinatu nahi denean.

Ondoren aipatutako bi funtzioak eta fitxategien posizioa kontrolatzeko liburutegi estandarreko beste zenbait funtzio aurkezten dira.

```
int fseek(FILE *fp, long pos, int modua)
```

Zuzeneko atzipena bideratzen du. `fp` fitxategian `modua`-ren arabera eta `pos` kontutan hartuz, uneko posizioa aldatuko du. `modua`-ren balioa `SEEK_SET` bada, posizio-aldaketak fitxategiaren hasierarekikoak dira; `SEEK_CUR` bada, oraingo uneko posizioarekikoak eta `SEEK_END` bada, azken posizioarekikoak. `pos` negatiboa izan daiteke.

```
long ftell(FILE *fp)
```

Uneko posizioaren balioa itzuliko du gauzak ondo badoaz. Erroreren bat gertatu bada (adibidez, fitxategia gailu bati badagokio edo emaitza `long int` bezala adieraztea ezinezkoa bada), `–1` balioa itzuliko du.

```
int ungetc(int kar, FILE *fp)
```

`kar` karakterea (printzipioz, *getc* azken irakurketan irakurritakoa) `fp` fitxategira itzuliko da. Hurrengo irakurketak `kar` karakterea berriz irakurriko du. Bakarrik karaktere bat itzul daiteke fitxategi batera.

```
void rewind(FILE *fp)
```

`fp` fitxategiaren uneko posizioa fitxategiaren hasiera izango da hemendik aurrera.

9.4. programan zuzen_atzi funtzioa definitzen da, zeinak parametro baten arabera posizio jakin batean dagoen luzera finkoko eremu bat irakurtzen edo idazten duen.

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

#define IRAKUR 'I'

int zuzen_atzi(FILE *fitx,long pos, char buffer[],intluz,
       char eragik)
{
int stat;

stat = fseek (fitx, pos, SEEK_SET);
if(stat!= 0)
   {
   printf("fseek-a ez da ondo joan\n");
   return (stat);
   }
```

```
switch (eragik)
   {
   case IRAKUR:
     if ((stat = fread (buffer, luz, 1,fitx))<1)
       printf("Irakurketa gaizki joan da\n");
     break;
   default:
     if ((stat = fwrite (buffer,luz, 1,fitx))<1)
       printf("Idazketa gaizki joan da\n");
     break;
   }
return (stat);
}
```

9.4. programa. *Zuzeneko atzipenaren erabilera.*

9.5. programan editoreek duten bilaketa-laguntza nola programatzen den ikustearekin batera, `ftell` funtzioaren erabilpenaren adibidea dugu. Esan beharra dago *strstr* funtzioak bere bigarren argumentua den karaktere-katea lehen argumentua den katean azaltzen denentz begiratzen duela —11. kapituluan sakonago ikusiko den bezala—; aurkitzen badu, bere posizioa itzuliko du; bestela, NULL erakuslea itzuliko du.

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

#define EZ 0
#define BAI 1
#define MAX 100

int aurkitu_al (FILE *fitx, char st[MAX])
{
char buff[MAX];
long pos;

while (fgets (buff, MAX-1, fitx)!= NULL)
   if (strstr (buff, st)!= NULL) /*bilatu st buff-ean */
     return (BAI);
return (EZ);
}
```

```
int bilatu (FILE *fitx, char *string)
{
long uneko_pos;

uneko_pos = ftell (fitx);
if (aurkitu_al (fitx, string) == EZ)
   {
   fseek (fitx, uneko_pos, SEEK_SET);
   /* zegoen posiziora itzuli*/
   return (EZ);
   }
else
   return (BAI);
}
```

*9.5. **programa.** fseek eta ftell funtzioen erabilpena.*

9.6. BUFFER-EN ERABILPENA

Memoriarekin konparatuz, memoria lagungarriak oso biltegigailu motelak dira. Hortaz, fisikoki gertatzen diren irakurketa eta idazketen kopuruak ahal den heinean gutxitu behar dira. Buffer-ak erabiltzea hau lortzeko modu errazena eta hedatuena da.

Duten helburua gauzatu baino lehen datuak pilatzeko memori zatia da bufferra. Buffer-en erabilpen egokiak datu-transferentzia azkartzen du, sarrera/irteerako gailuen atzipenak gutxituz.

Sistema eragile guztiek buffer-ak erabiltzen dituzte sarrera/irteerako gailuetan irakur eta idazteko. Hau da, sistema eragileak tamaina finkoko bloketan (ohizkoa izaten da blokeok sektoreen luzera edukitzea) atzitzen ditu sarrera/irteerako gailuak. Horrela, bloke bat irakurriko da, nahiz eta karaktere bakarra izan irakurri nahi dena. Gehiago nahi izanez gero, blokearen barnean dauden artean blokea memorian mantenduko da eta bertatik irakurriko dira, benetako irakurketak gutxituz.

C-ren liburutegiak buffer-ak kontrolatzeko erak eskaintzen ditu. Modu bat lerrokako buffering-a da: kasu honetan, sistemak karaktereak gordetzen ditu lerro-bukaerako karaktere bat aurkitu edo buffer-a bete arte, ondoren lerro osoa prozesatuz. Hau gertatzen da, adibidez, datuak terminaletik irakurtzean. Blokekako buffering-ean, aldiz, blokea bete arte gordeko dira karaktereak, jarraian prozesatzeko asmoz. Alderantzizkoa esaten ez bada, fitxategi guztietan buffer-en bidez egiten dira irakurketa eta idazketak.

Lerrokako eta blokekako buffering-ak karaktereak banaka tratatzea baino eraginkorragoak diren artean, karaktere bakoitza azaltzen den une berean prozesatzea galarazten dute. Adibidez, erabiltzailearengandik jasotako karaktereak idazten diren une berean tratatu nahi izango balira.

C-k buffer-aren tamaina aldatzeko edo buffer-a erabat deuseztatzeko moduak badauzka. Baina tamainaren aldaketek sarrera/irteerako eragiketen abiadura moteltzea eragin dezakete, gehienetan balio lehenetsia optimoa izaten baita. Buffer-a deuseztatzeko edo tamaina aldatzeko, `setbuf` edo `setvbuf` funtzioak erabil daitezke. Buffer-a erabiltzean, kontuz ibili behar da programa bat baino gehiagok fitxategi berarekin lan egiten badute.

`fflush` funtzioarekin buffer batean dagoen guztia batera bere helburura bidaliko da. Interesgarria izaten da funtzio hau erabiltzea irteerako datuak denbora luzez egon ez daitezen gailuetara joan gabe, makina-errore bat gertatuz gero informazioa benetan gorde gabe baitago.

Ondoren `setbuf`, `setvbuf` eta `fflush` funtzioen prototipoak eta deskribapen laburra ikus daitezke.

```
void setbuf(FILE *fp, char *buffer)
```

Fitxategi horren `buffer`-aren tamaina taularena bera izango da. Taularen erakuslea `NULL` bada, buffering-a deuseztatuta gelditzuko da. Funtzio honek ez du baliorik itzultzen.

```
void setvbuf(FILE *fp, char *buffer, int modua, int tam)
```

Fitxategi baten buffering-politika aldatzen da moduaren arabera: `_IOFBF` bada blokeka, `_IOLBF` bada lerroka eta `_IONBF` bada buffer-ik gabe. Zehazten den bufferra erabiliko da sistemak erabiltzen duenaren ordez.

```
int fflush(FILE *fp)
```

Bufferretan gordetakoa bere helburura bidaltzen da.

9.7. BESTELAKO SARRERA/IRTEERA FUNTZIOAK

`<stdio.h>` fitxategi estandarraren barnean azaltzen diren eta aurretik aipatu ez diren funtzio batzuk azaltzen dira ondoren labur-labur.

```
void perror(const char *kate)
```

`stderr`-en errno aldagai globalari dagokion mezua idatziko du.

```
int remove(const char *izena)
```

Izen hori duen fitxategia ezabatuko du. Honen ostean fitxategia irekitzeko saioek huts egingo dute.

```
int rename(const char *izen_zahar, const *char izen_berri)
```

`izen_zahar` deituriko fitxategiaren izena `izen_berri` izatera pasako da. `izen_berri` izeneko fitxategi bat jada baldin bazegoen funtzio honi deitu aurretik arazoak egon daitezke.

```
int sprintf(char *buffer, const char *formatua, ...)
```

Bere argumentuak `formatua`-n azaltzen diren moduan eraldatzen dira (ikus `printf`-aren barneko eraldaketak) eta `buffer`-ek zehazten duen karaktere--katean gordetzen dira. `printf`-ak bezala funtzionatzen du gainontzean.

```
int sscanf(char *buffer, const char *formatua, ...)
```

`buffer` kateako karaktereak `formatua`-k zehaztutako eran gordetzen dira bere argumentuetan. Bestela, `scanf`-ak bezala funtzionatzen du.

```
char *tmpnam(char *izen)
```

Fitxategi baterako izen unibokoa eraikitzen du `izen` aldagaian gordez. `izen` aldagaiaren helbidea itzuliko du.

10. AURREKONPILADOREA ETA BESTE LENGOAIAK

C lengoaiaz programatzen dugunean, konpiladoreek zein estekatzaileek eskaintzen dizkiguten laguntzak nola erabil ditzakegun azaltzea da kapitulu honen helburua.

Konpilazio-prozesua bi fasetan gertatzen da: lehenean (aurrekonpilazioa deitutakoan) aurrekonpiladoreari zuzendutako sententziak edo sasiaginduak tratatzen diren bitartean, bigarrenean konpiladore guztiek burutzen duten itzulpen-prozesua gertatzen da eta, ondorioz, objektu-programa lortzen da.

Estekatzaileak (*linker* ingelesez) objektu-modulu bat edo gehiago lotzen ditu (*link*), haien arteko erreferentzia gurutzatuak ebatziz eta programa exekutagarria lortuz. Lotzen diren moduluak erabiltzailearenak edota estandarrak izan daitezke. Azken hauen ezaugarriak honako hauek dira: C liburutegian daude, aukera asko eskaintzen dute eta C-k duen ahalmenaren atal nagusietako bat da. Lengoaia desberdinetako moduluak aparte konpilatu ondoren, liburutegian biltegiraturik ala ez, objektu-moduluak direnez estekatzailearen bidez bil daitezke.

Aurkeztutako elementu hauek azaltzeko, kapitulu honetan honako gai hauek jorratuko ditugu: aurrekonpiladorea, erabiltzailearen liburutegiak eta sistema-deien erabilpena. Hurrengo kapituluan liburutegi estandarra azalduko dugu.

10.1. AURREKONPILADOREA

Pentsa daiteke C-ren aurrekonpiladorea konpiladorea baino lehen exekutatzen den programa dela. Aurrekonpiladorearen lana gidatzeko, programatzaileak *sasiaginduak* izeneko aurrekonpiladorerako sententziak idatz ditzake.

Aurrekonpiladoreak eskaintzen dituen aukera nagusiak honako hiru hauek dira:

- Konstante parametrizatu eta makroen definizioa, `#define` sasiaginduaren bidez. Bostgarren kapituluan azaldu diren arren hemen zertxobait gehiago sakonduko dugu.

- Beste iturburu-modulu bat edo gehiago programan sartzea, `#include` sasiaginduaren bidez.

- Baldintzapeko konpilazio-zatiak ezartzea, `#ifdef` eta `#endif` bezalako sasiaginduen bidez.

Aurrekonpiladoreari zuzendutako sasiagindu guztiak # karaktereaz hasten dira. Karaktere horrek zurigune edo tabulazioa ez den lerroko lehenengo karakterea izan behar du konpiladorea ANSI estandarrari egokitua badago. Konpiladore zaharrenetan, aldiz, lerroko lehenengo karakterea izan behar du eta ezin da zurigunerik egon karakterearen eta sasiaginduaren artean.

Sasiaginduak lerro-bukaerako karakterearekin bukatzen dira, eta ez puntu eta komaz, bestelako C sententziak bezala. Sasiagindu batek lerro bat baino gehiagoko luzera edukitzea nahi bada, lerro-bukaeraren aurreko karaktereak alderantzizko barra (\ karakterea, alegia) izan behar du.

10.2. *#define* ETA *#undef* SASIAGINDUAK

`#define` sasiagindu honen bidez konstante eta makroen definizioa egiten da 2. eta 5. kapituluetan ikusi genuenez. Beraz, izen bati balio edo agindu-sekuentzia bat egokitzen zaio, eta ondorioz, konpiladoreak izena aurkitzean, izen horren ordez dagokion balioa edo agindu-sekuentzia jarriko du konpilatu aurretik. 9.1. programan bi aplikazio arruntenak daitezke: alde batetik `MAX` konstantea definitzen da eta, bestetik, `ABSOL(x)` makroa.

Aurretik aipatu zen bezala, konstante eta makroen izenak maiuskulaz idazteko joera oso zabala da, aldagaien izenetatik bereizi ahal izateko. ANSI estandarrean beraien izenak 31 karakteretarainokoak izan daitezke.

Nahiz eta `#define` sasiaginduak iturburu-fitxategien edozein tokitan ager daitezkeen, hobe da hasieran edo `#include` baten bidez sartutako goiburuko fitxategi batean azaltzea, programaren edozein ataletatik erreferentziatu ahal izateko.

```
#include <stdio.h>

#define MAX 32767
#define ABSOL(x) ((x < 0)? - x: x)

int min_abs (taula, luz) /* taula baten minimo absolutua */
int taula [], luz;
{/* taula baten balio absolutu txikiena */
int i, txiki = MAX;

for (i=0; i < luz; i++)
  if (ABSOL (taula [i]) < txiki)
    txiki = ABSOL (taula [i]);
return (txiki);
}
```

10.1 programa. `#define` *erabiltzen deneko adibidea.*

Definizioaren esanahiak moduluaren bukaeraraino irauten du `#undef` sasiagindua erabiltzen ez bada. Sasiagindu berri hau interesgarri gertatzen da definizioari esanahi berri bat emateko; konpiladore gehienek eta ANSI estandarrak berak birdefinizioa baino lehen hala egitea eskatzen baitute.

Makro baten definizioan azaltzen den parametro bat # karakterearen ondoren idatzia badago (bitarteko inolako karaktererik gabe), orduan izen hori komatxoen artean eta uneko parametroaz ordezkaturik azalduko da makroa zabaltzerakoan. Adibidez, arazketarako erabiltzen den ondorengo inprimatze-makroa definitzen badugu:

```
#define arainpri(esp) printf(#esp " = %g\n", esp)
```

eta programan zehar honelako deia egiten bazaio:

```
arainpri(x/y);
```

orduan, makroaren zabaldutakoan honokoa agertuko da:

```
printf("x/y" " = %g\n", x/y);
```

Konstanteetan ager daitezkeen komatxo karaktereak `\"` kateagatik ordezkatzen dira, alderantzizko barra karaktereak `\\` kateagatik ordezkatzen diren bezala, karaktere horiek soilduak agertzen direnean bestelako funtzioa baitute.

Bestalde, argumentuak katea daitezke ## eragileari esker. Bere aurretik eta ondoren dauden izenak kateatzen dira, erdian dauden zuriguneak eta ## eragilea desagertuz. Adibidez, hurrengo makroari deitzerakoan:

```
#define elkartu(lehena, bigarrena) lehena ## bigarrena
```

bere bi argumentuen izenak elkartuko dira: `elkartu(hi, bai)` egiteak `hibai` emango du emaitza bezala.

10.3. *#include* SASIAGINDUA

Sasiagindu honen bidez iturburu-programa bat beste baten barruan sar daiteke, liburutegiaren kontzeptua iturburu-mailara hedatuz.

Sasiagindua idazteko erak hauek dira:

```
#include <fitxategi_izena>
```

edo

```
#include "fitxategi_izena"
```

`fitxategi_izena`-k barneratu nahi den goiburuko fitxategiak duen izena zehazten du eta `.h` luzapena izatea gomendatzen da; horrela bereizten baitira goiburuko (*header*) deitutako fitxategiak. < eta > edo komatxoak erabiltzea desberdina da, goiburuko fitxategia katalogo desberdinetan bilatzen baita. C-ren goiburuko fitxategi estandarra bada, < eta > artean jarriko da konpiladoreak ezagutzen duen katalogoan bila dezan (adibidez, UNIXen ohizkoa da /usr/include katalogoan bilatzea). Komatxoen artean egoteak, aldiz, erabiltzailearen goiburuko fitxategia dela adierazten du eta normalean uneko katalogoan (iturburu-fitxategia dagoen katalogoan bertan askotan) bilatuko da. Hor aurkitzen ez bada, sistemak goiburuko fitxategiak gordetzen dituen katalogoan bila daiteke.

Goiburuko fitxategiak liburutegiekin eta konpilazio banatuarekin batera erabiltzen dira. Funtzio batzuk modulu banatu batean konpilatzen badira estekatzailea arduratuko da moduluen arteko loturaz, baina modulu banatuan erabiltzen diren parametroak, datu-motak, aldagai globalen definizioa zein funtzioen motak beste moduluetan erazagutu behar direnez, erosoena goiburuko fitxategi batean aipaturiko definizioak sartzea da. Fitxategi hori modulu banatuaren funtzioek erabiltzen dituzten programetan sartuko da `#include` baten bidez. 9.2. programan honen adibide bat ikus daiteke:

```
/* logika.h fitxategiaren edukia */
/* eta, edo eta ala makroak modulu banatuan */

#define TRUE 1
#define FALSE 0

#define eta(x,y) (x&&y)
#define edo(x,y) (x||y)
#define ala(x,y) ((x&&!y)||(!x&&y))

typedef int bool;

/* probalog.c programa */

#include "logika.h"

main ( )
{
bool a,b,c;

a = TRUE; b = FALSE;
c = edo (a,b);
printf("Emaitza %d da.\n", c);
c = eta (a,b);
printf("Emaitza %d da.\n", c);
c = ala (a,b);
printf("Emaitza %d da.\n", c);
}
```

10.2. programa. `#include` *sasiagindua nola erabili.*

Goiburuko fitxategi batean aldaketaren bat egiten bada, fitxategi hori erabiltzen duten fitxategi guztiak birkonpilatu behar dira.

10.4. BALDINTZAPEKO KONPILAZIOA

Baldintzapeko konpilazioa oso baliagarria da zenbait kasutan, ondoko biak ohizkoenak izanik:

- programa bat makina eta plataforma desberdinetarako egiten denean, arazoak egoten dira erabateko garraiagarritasuna ziurtatzeko. Horrelako kasuetan makinaren araberako kodea ezar daiteke baldintzapeko konpilazioari esker. Hau egin ohi da ANSI ez diren karaktereekin (`ñ` adibidez).

- programa baten arazketa-garaian inprimatze bereziak egin nahi direnean, konstante baten balioaren arabera programa arazteko kodea eranstea posiblea izango baita.

Baldintzapeko konpilazioa egiteko modu bat `#if`, `#else`, `#elif` eta `#endif` sasiaginduak erabiltzea da. Baldintzazko sasiagindu hauek ondoko erregelei jarraitzen diete:

- `#if` eta `#elif` baten ostean azaldutako baldintzazko espresioak konstantea izan behar du. Bihurketa inplizituak gertatzen direla kontuan edukiko da (adibidez, ANSI estandarrean baldintzazko espresioetan azaltzen diren konstanteak `long int` motara bihurtzen dira).

- `#define` sasiaginduaren bidez definitutako makro eta konstanteak beren balioengatik ordezkatzen dira baldintzazko espresioa ebaluatu aurretik.

- Baldintzazko espresio batean definituta ez dagoen izen bat azaltzen bada, zeroak ordezkatuko du.

- Baldintzazko sasiaginduak kabia daitezke, `if` sententziak kabiatzen diren bezala.

Aipatu beharra dago sasiagindu hauen eta C lengoaiaren baldintzazko egituren arteko ezberdintasunak:

a. `#if` eta `#elif` sasiaginduetan azaldutako baldintzapeko espresioek ez dute parentesi artean egon beharrik (parentesi artean egotea aukerakoa da).

b. `#elif` sasiagindua (K&R estandarraren barne ez dagoena) C lengoaiaren `else if` aginduaren analogoa da.

c. sasiagindu baten menpean dauden blokeak eta sententziak ez dira giltzen artean idazten. Horren ordez, `#elif`, `#else` edo `#endif` sententziek mugatuta daude.

d. `#if` bakoitzak edozein `#elif`-en kopurua eduki dezake, baina `#else` bakarra onartzen da.

e. `#if` bakoitzak `#endif` batean bukatu behar du.

Badaude beste bi sasiagindu baldintzapeko konpilazioa egiteko, `#ifdef` eta `#ifndef` izenekoak. `#if` sasiaginduak bezala, biok `#endif` sasiagindua-rekin bukatu behar dute. Biok `#else` adarra onartzen duten artean, ez dute `#elif` adarrik onartzen.

`#ifdef`-aren bidez agindu-sekuentzia bat programaren barruan sartuko da definizioa indarrean badago, eta konpiladoreak ez du kontuan hartuko, definizio hori ez badago. Definizioa kontrolatzeko `#define` eta `#undef` sasiaginduez gain konpilazio-komandoaren D aukera erabil daiteke. `#ifndef` sasiaginduak, ordea, agindu-sekuentzia sartuko du definizioa indarrean ez badago.

Aurrekoaz gain ANSI estandarrak `defined` izeneko eragilea ere badauka definituta. Horrela ondoko bi sasiaginduak guztiz baliokideak dira:

```
#if defined izena
#ifdef izena
```

eta beste bi hauek ere:

```
#if !defined izena
#ifndef izena
```

Esan dugun bezala baldintzapeko konpilazioa interesgarria suertatzen da makina desberdinetarako programa bat garatzen ari garenean. Adibidez, programa batek PC bateko inprimagailuaren erregistroa zuzenean maneiatu behar badu, segidako kodean azaldutakoa egin daiteke makina desberdinetako helbideak kontuan har daitezen.

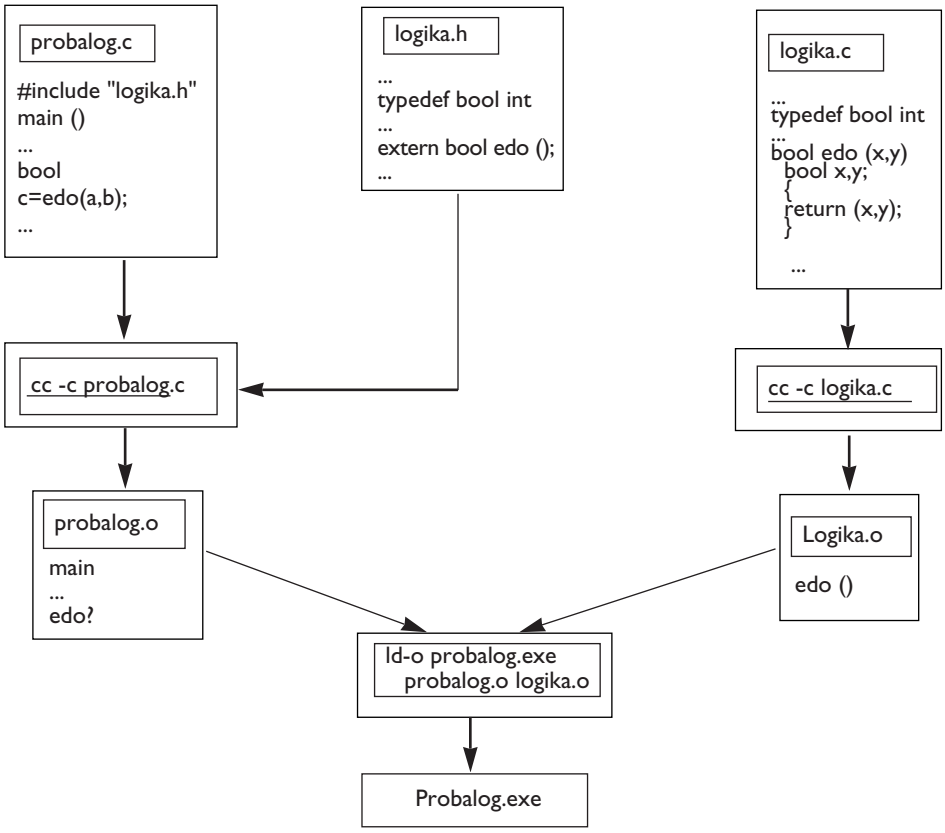
```
#ifdef IBM
   int reg_dat_inp = 0x378
#endif
#ifdef BESTE
   int reg_dat_inp = 0x3B8
#endif
...
```

Konpilatzeko komandoaren `D` aukeraren bidez edo 12. kapituluan azalduko den *make* programa erabiliz definituko da sinboloetako bat, IBM edo beste.

10.5. ERABILTZAILEAREN LIBURUTEGIAK

Programazio egituratuak eta softwarearen ingeniaritzak agindu bezala, programak luzeak direnean funtziotan banatu behar dira, funtzio-multzoak osatuz, horietako bakoitza bere aldetik konpilatu eta probatuko delarik. Horrela sortutako moduluak liburutegi batean kataloga daitezke, estekatzaileak erabil ditzan erreferentziatzen dituzten programekin. Horretarako 8. kapituluan aztertutako aldagaien ezaugarriak hartu behar dira kontuan.

Bide honetan oso komenigarria da objektu-modulu edo liburutegi bakoitzeko goiburuko fitxategi bat edo gehiago egitea, bertan konstante eta datu-moten definizioa zein funtzioen erazagupena egingo delarik. 10.1 eskeman, ikusitako 10.2 programari dagokion konpilazio-estekaketaren eskema azaltzen da.



10.1 eskema. *konpilazio banatuaren eskema*

10.1. eskema horretan logika.o aparteko modulua da, baina ez da liburutegi batean katalogatu. Horretarako ar (UNIXekin) programa erabili behar da eta estekatzeko komandoa aldatu. Gelditukо litzatekeen komando-multzoa honako hau litzateke:

```
$ cc -c logika.c
$ ar rc liblogika.a logika.o
$ cc -o probalog.exe probalog.c -l logika
```

10.6. SISTEMA-DEIAK C-TIK

Nahiz eta C lengoaia oso ahaltsua izan, batzuetan sistema eragilearen edota sistemaren ingurunearen baliabideak atzitzea nahi izango edo beharko da. Sarritan, gailu zehatz bat edo sistema eragilearen funtzio bat (hau da sistema-dei bat) erabili behar da, liburutegi estandarreko funtzio arruntak erabiliz halako zerbait egitea ezinezkoa baita. Sistema-deiak sistema eragilearen jatorrizko eragiketak dira, beraz, sistema eragilearen arabera sistema-dei desberdinen aurrean egongo gara. Hortaz, beraien funtzio guztiak zehaztasun guztiekin deskribatzen dituzten eskuliburuak erabiltzea derrigorrezkoa izango da.

Dena den, sistema-deiak erabili aurretik, honako ideia hauek kontutan hartu behar dira:

1) C-ren funtzio batzuen ordez sistema-deiak erabiliz, azkarragoak diren eta memoria gutxiago erabiltzen duten programak lor daitezke. Gainera, C liburutegi estandarraren bitartez atzi ez daitezkeen funtzioak atzi daitezke.

2) Arazo asko sor daitezke sistema-deiak liburutegi estandarraren funtzioen ordez erabiltzean: programa ez baita jadanik garraiagarria izango, era horretan sistema eragilea eta C konpiladorearen bertsioen menpeko kodea lortuko baita. Norberaren erabakia da sistema eragilea eta makinarekiko menpekotasuna duen kodea noiz eta nola sortu.

Ondoren azalduko ditugun sistema-deiak bi sistema eragile erabilienetakoak dira: MS-DOS eta UNlXenak.

*MS-DOS*eko liburutegia

dos.h fitxategian sistema-deiak erabiltzen dituzten liburutegiko funtzioak erazagutzen dira. Honako hauek dira esanguratsuenak:

funtzioa	*esanahia*
`absread`	irakurketa
`abswrite`	idazketa
`disable`	etenak galaraztea
`getdfree`	erabili gabeko tokia diskoan
`inport`	portu batetik irakurtzea
`outport`	portu batean idaztea
`keep`	bukatu eta egoiliar mantentzea
`int86`	eten bat sortzea
`setveet`	eten-bektorea aldatzea
`bdos`	sistema-dei desberdinak parametroaren arabera
`intdos`	sistema-dei desberdinak parametroaren arabera

UNIXeko liburutegia

UNIXen sistema-dei asko daude, sistema eragilearen zerbitzuak zuzenean erabiltzeko asmoz. Goiburuko fitxategi desberdinetan dago sistema-deiei dagozkien funtzioen erazagupena; definizioa liburutegian konpilaturik baitago.

Honako hauek dira dei garrantzitsuenak:

prozesuen gaineko deiak	
`exec`	programa berri baten karga
`fork`	prozesu berri baten sorrera
`wait`	sortutako prozesu baten bukaera arte gelditzea
`exit`	programaren bukaera
`sleep`	zenbait segundo gelditzea

fitxategi eta gailuen gaineko deiak	
`open`	fitxategi bat irekitzea
`read`	fitxategi batean irakurtzea
`write`	fitxategi batean idaztea
`close`	fitxategi bat ixtea
`lseek`	fitxategiaren uneko posizioa aldatzea
`creat`	fitxategi bat sortzea
`link`	fitxategi bat konpartitzea
`mknod`	azpikatalogo bat sortzea
`chmod`	babes-kodea aldatzea
`utime`	denbora-irakurketa
`fstat`	fitxategiaren egoera
`ioctl`	gailuen ezaugarri-aldaketa

prozesu arteko komunikazioa	
`pipe`	pipe (komunikazio-bide sekuentziala) bat sortzea
`dup`	fitxategiaren deskribatzailea bikoiztea
`semget`	semaforo bat eskuratzea (System V)
`semop`	semaforoaren gaineko eragiketa (System V)

Sistema-dei hauen erabilpena nahikoa konplexua eta bertsioaren araberakoa da. Horrexegatik ez dugu hemen gehiago sakonduko.

10.7. C ETA MIHIZTADURA-LENGOAIA

C-k behe-mailako lengoaietako funtzionalitate asko izan arren, batzuetan errutina bat mihiztadura-lengoaian idazteko beharra ere izaten da. Hori egiteko arrazoiak honakoak dira:

a) Abiadura eta eraginkortasuna handitu nahia. Modu horretan, programatzaile batek programa baten errutina kritikoak konpiladore onenak baino hobeto optimizatzeko gai baldin bada abiadura handi dezake.

b) Makinari espezifikoa den eta C-n eskura ez dagoen funtzio bat eraiki nahia, makina askok C konpiladoreak dituen aginduekin zuzenean exekuta ez daitezkeen aginduak baititu (adibidez, datu-segmentuak aldatzen dituztenak edo konputagailuaren sarrera/irteerako atakak edo portuak zuzenean irakur/idatz ditzaketenak).

c) Xede orokorreko mihiztadura-lengoaiako errutina-paketeak erabili nahia. Ohizkoa izaten da objektu-kodean idatzitako errutinaz osaturik dauden liburutegiak erostea (adibidez, grafikoak marrazten dituzten azpirrutinekin). Batzuetan estekaketa baino gehiago behar ez den artean, bestetan interfaze-modulu bat programatu beharra dago mihiztadura-lengoaiaz, erositako errutina horiek erabili ahal izateko.

Prozesadore bakoitzak bere mihiztadura-lengoaia duenez, honela idatzitako programak garraiagarriak ez dira izango, sistema-deiak dituztenak bezala. Hala ere, oso gai aurreratua denez, bakarrik derrigorrezkoa denean mihiztadura-lengoaian idaztea gomendatzen da.

PCetako mihiztadura-lengoaia

PCetan exekutatzen diren programak memoriako 6 eredu desberdinetan antola daitezke konpiladorearen arabera. Hurrengo pasartean aztertuko ditugun eredu hauen arabera, erakusleak ondoko hiru mota honetakoak izan daitezke: `near`, `far` eta `huge`.

`near` motako erakusleak 16 bitekoak diren bitartean, gainerakoak 32 bitekoak dira; lehen kasuan helbidea segmentu berean dagoela suposatzen baita. `far` eta `huge` motako erakusleak antzekoak dira (biak 32 bitetan gordetzen

baitira) eta haien artean dagoen desberdintasun bakarra desplazamenduan adieraz daitekeen neurrian datza.

`near`, `far` eta `huge` gako-hitzak dira. PCetako C konpiladoreetarako eta erakusle, taula zein funtzioen definizioetan erabil daitezke. C programa batek mihiztadura-lengoaiaz idatzitako errutina bat erabiltzeko, ez du ezer berezirik egin behar; funtzio bezala erazagutu eta erreferentziatzea besterik ez baitu behar. Dena den, funtzioaren erazagupenean `near`, `far` edota `huge` erabil daiteke memoriako ereduaren arabera.

Aldiz Ctik deitutako errutina bat mihiztadura-lengoaiaz idazten badugu, ondoko arauak hartu behar dira kontuan:

- parametroak pilan kokatuko dira, jeneralean alderantzizko ordenan, hau da, azken parametrotik hasiko da pila betetzen. Parametroek hartuko duten luzera 2. kapituluan aztertutako datuen ezaugarrien araberakoa izango da.

- errutinan, dagozkion ekintzak baino lehen, honako beste hauek burutu egin behar ditu: l) erabiliko diren erregistroen edukia (BP beti, eta SI, DI, SS, DS edota CS beharrezkoa bada) pilan gordetzea. 2) BP erregistroan SPren balioa gordeko da, horrela BPren bidez parametroak eskuratuko baitira. Lehen parametroa (BP + 4) helbidean egongo da funtzioa `near` bada eta (BP + 6)-an gainerako kasuetan.

- aldagai lokalak erabiltzeko SP erregistroaren balioa txikiagotuz joango da.

- errutinari dagozkion ekintzak burutu ondoren, itzul-emaitza AX erregistroan kokatu behar da eta ondoren gordetako erregistroen balioa berreskuratu. Batzuetan, eta datuen tamainaren arabera, DX erregistroan ere kokatuko da emaitza.

10.3. programan bi zenbaki biderkatzen dituen mihiztadura-lengoaiaz idatzitako bider izeneko errutina azaltzen da, segmentuei dagozkien sasiaginduak azaldu gabe. Aipatzekoa da mihiztadurazko errutinen izenari azpimarra karakterea aurretik jartzen zaiola.

```
          PUBLIC      _bider
_bider    PROC        NEAR
          push        bp
          mov         bp, sp
          mov         ax [bp + 4]
          cmul        WORD PTR [bp + 6]
          mov         sp, bp
          pop         bp
          ret
_bider    ENDP
```

10.3. programa. *bider errutina mihiztadura-lengoaiaz.*

C konpiladorea 16 biteko ordenadoreetan

Esan dugunez PCetako programetan memoriako 6 eredu desberdinen artean aukera egin daiteke konpilatzeko garaian. PCetako memoria sistema segmentatua da eta bertan lau segmentu bereizten dira (aginduetakoa, datuetakoa, pilakoa eta estra), bakoitzari erregistro bat dagokiolarik; CS, DS, SS eta ES hain zuzen. Erregistro hauen balioa (eta ondorioz, programaren luzera) aukeratzeko ereduak daude.

Small ereduaz bi segmentu baino ez dira erabiltzen: aginduetarakoa eta datuetarakoa. Beraz, programek gehienez 128 K edukiko dute. Programa txikietarako erabiltzen da. *Medium* ereduaz datuetarako segmentu bat eta aginduetarako segmentu anitz adierazten da. Beraz, datuek 64 K baino gehiago hartzen ez duten bitartean, aginduek Mega bateraino irits daitezke. Datu gutxi maneiatzen duten programa luzeetarako da. *Compact* ereduaz, aurrekoaren kontrakoa egiten da: aginduek 64 K baino gutxiago hartu eta datuek Mega bateraino. Datu asko maneiatzen duten programetarako egokia da. *Large* ereduaz, agindu zein datuak Mega bateraino irits daitezke, bietan seg-

mentu anitz egokitzen delako. Beraz, programa handietarako egokia da. *Huge* eredua, *Large* ereduaren antzekoa da, baina datu estatikoak segmentu bakarrean kokatzen dira.

Eredua handiagotu ahala, programa luzeagoa eta motelagoa gertatzen da, erregistro eta pilaren arteko trukeak direla eta.

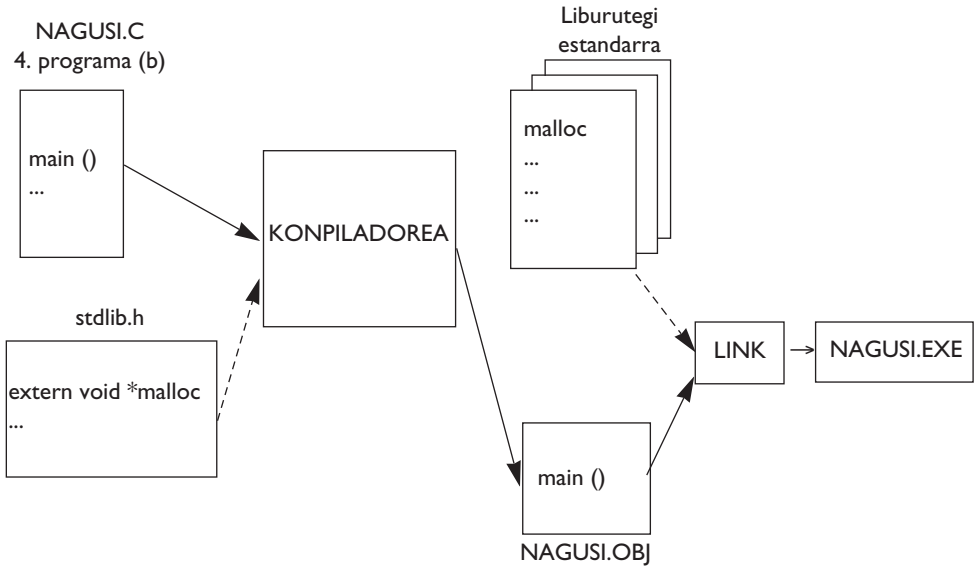
Seigarren eredua *Tiny* izenekoa da eta Turbo C eta Borland C bezalako konpiladoreek baino ez dute eskaintzen. Eredu honen arabera programa osoa segmentu bakarra da eta 64 K baino gehiago ezin du hartu.

Eredua konpilatzeko komandoan zehazten da, eta konpiladore gehienetan honela erabiltzen da: `AS` aukera *small* eredurako, `AM` aukera *medium* eredurako, `AC` aukera *compact* eredurako eta `AL` aukera *large* eredurako.

11. LIBURUTEGI ESTANDARRA

5. kapituluan azaldu genuenez, funtzioak erabili baino lehen definitu egin behar dira, baina badago aurredefinituta dagoen funtzio-multzo bat; funtzio estandarren multzoa, edo, gauza bera dena, liburutegi estandarreko funtzioak. Sarrera/irteerakoak 9. kapituluan aztertu baditugu ere, besteak ez.

Kapituluan zehar funtzio garrantzitsuenak agertzen dira; sarrera/irteerakoak salbu. Funtzio bakoitzari dagokion izenburu-fitxategiaren izena zehazten da, funtzio estandarrak erabiltzeko dagokien izenburua `#include` batez zehaztu behar baita. Konpilazio- eta estekaketa-prozesua irudikatzen da 11.1 eskeman.



11.1 eskema. *Konpilazio- eta estekaketa-prozesua liburutegi estandarrekin.*

11.1. KARAKTEREEN GAINEKO FUNTZIOAK

Makro bezala inplementatzen dira eta bi motakoak daude: karakterearen mota probatzen dutenak eta bere mota aldatzen dutenak. Guztiak erabiltzeko, aurretik `#include <ctype.h>` sasiagindua egin beharko da. Karaktereak `int` motako parametroak izaten dira, baina bihurketa inplizituak gertatuko dira karaktere-motakoak erabiltzen badira.

Ondoren mota honetako funtzio erabilienak azaltzen dira, beren prototipoa eta deskribapen laburra emanez.

```
int tolower(int kar)
```

`kar` karakterea maiuskulaz dagoen letra bada, berari dagokion minuskula-karakterea itzuliko du. Bestela, ez du aldaketarik egingo.

```
int toupper(int kar)
```

`kar` karakterea minuskulaz dagoen letra izanik, dagokion maiuskula-karakterea itzuliko du. Bestela, ez da aldatuko.

```
int islower(int kar)
```

`kar` karakterea minuskulaz dagoen letra bat bada, zeroa ez den balio bat (egiazkoa) itzuliko du; bestela, zeroa (faltsua).

```
int isupper(int kar)
```

`kar` karakterea maiuskulaz dagoen letra bat bada, zeroa ez den balio bat itzuliko du; bestela, zeroa.

```
int isalpha(int kar)
```

`kar` karakterea letra bat bada, zeroa ez den balio bat itzuliko du; bestela, zeroa.

```
int isdigit(int kar)
```

`kar` karakterea digitu hamartar bat bada, zeroa ez den balio bat itzuliko du; bestela, zeroa.

```
int isalnum(int kar)
```

`kar` alfanumerikoa bada (hau da, 0-9, a-z edo A-Z tarteetako batean badago) zeroa ez den balio bat itzuliko du; bestela, zeroa.

```
int iscntrl(int kar)
```

`kar` kontrol-karaktere bat bada, zeroa ez den balio bat itzuliko du; bestela, zeroa. Kontrol-karaktereak inprimatu ezin daitezkeen karaktereak dira. ANSI karaktereen barnean, 0x00-0x1F tartekoak, biak barne, eta 0x7F karaktereak kontrol-karaktereak dira.

11.1 programan batzuen erabilpena argitzen da. Programa honek karaktereak jasotzen ditu terminaletik eta, zenbakizkoak badira, mezu bat itzuliko du. Letrak ez badira ere mezu bat itzuliko du. Letrak badira, maiuskulak minuskula eta minuskulak maiuskula bihurtuko ditu. Hau guztia hamar aldiz burutuko du.

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

main()
{
char kar;
int zenb, i;

for (i=0; i<10; i++)
  {
  printf("Sakatu karaktere bat:");
  scanf("%c", &kar);
  if (isdigit((int)kar))
    printf("%c zenbakizkoa da\n", kar);
  else
    if (!isalpha((int)kar))
      printf("%c ez da letra ezta zenbakizkoa\n",kar);
    else
      if (islower((int)kar))
        {
        zenb=toupper((int)kar);
        printf("%c karakterearen maiuskula %c da\n",
          kar, (char)zenb);
        }
      else
        {
        zenb=tolower((int)kar);
        printf("%c karakterearen minuskula %c da\n",
          kar, (char)zenb);
        }
  }
}
```

11.1 programa. *Karaktere-motei buruzko funtzioen erabilpena.*

11.2. FUNTZIO MATEMATIKOAK

Eragiketa matematikoetan laguntzea dute helburu. Alderantzizkoa esaten ez bada behintzat, guztiek aurretik `#include <math.h>` sasiagindua behar dute. Esan beharra dago funtzio hauetan bi motako erroreak gerta daitezkeela:

- Eremu-erroreak: Argumentu baten balioa argumentuentzat legezkoa den barrutitik kanpo dagoenean (adibidez, zenbaki negatibo baten erro karratua lortu nahi izanez gero).

- Hein-erroreak: Emaitzaren balioa `double` batek adieraztea ezinezkoa denean.

Funtzio matematiko gehienek `double` motako datuekin lan egiten dute. Dena den, ANSI komiteak etorkizunean `float` edo `long double` motako datuekin ere lan egin dezatela pentsatzen ari da.

Ondoren mota honetako funtzio matematiko erabilienak azaltzen dira, beren prototipoa eta deskribapen laburra emanez.

```
int abs(int zenb)
```

`zenb` zenbakiaren osoko balio absolutua itzuliko du. Emaitza `int` motako balio batean adieraztea ezinezkoa bada, orduan ezezaguna izango da. Funtzio honek `<stdlib.h>` fitxategi estandarraren `#include`-a eskatzen du (eta ez lehen aipatutako `<math.h>` fitxategiarena).

```
double pow(double t, double u)
```

`t` ber `u` balioa itzuliko du. Eremu-errore bat gertatuko da `t` zeroa eta `u` zeroa baino txikiagoa edo berdina bada, edo `t` negatiboa eta `u` osokoa ez bada. Emaitza `double` batean sartzerik ez badago, lehen aipatutako hein-errorea gertatuko da.

```
double sqrt(double t)
```

`t` zenbakiaren erro karratu positiboa itzuliko du. `t` negatiboa bada, eremu-errore bat gertatuko da.

```
double fabs(double t)
```

`t` zenbakiaren balio absolutu erreala itzuliko du.

```
double exp(double t)
```

`t`-ren esponentziala itzuliko du.

```
double log(double t)
```

`t`-ren logaritmo nepertarra itzuliko du. `t` negatiboa bada, eremu-errore bat gertatuko da. `t` zeroa bada, hein-errore bat gertatuko da.

```
double log10(double t)
```

`t`-ren logaritmo hamartarra itzuliko du.

```
double sin(double t)
```

`t` radianetan adierazita dagoela, bere sinua itzuliko du. `t` oso handia bada, emaitzak esanahirik ez edukitzea gerta daiteke.

```
double cos(double t)
```

`t` radianetan adierazita dagoela, bere kosinua itzuliko du. `t` oso handia bada, emaitzak esanahirik ez edukitzea gerta daiteke.

```
double tan(double t)
```

`t` radianetan adierazita dagoela, bere tangentea itzuliko du. `t` oso handia bada, emaitzak esanahirik ez edukitzea gerta daiteke.

Hiru funtzio trigonometriko hauez gain beste asko daude: `asin` (arku sinua), `acos` (arku kosinua), `atan` (arku tangentea), `sinh` (sinu hiperbolikoa), `cosh` (kosinu hiperbolikoa), `tanh` (tangente hiperbolikoa).

11.2. programa segidan azaltzen diren funtzio batzuen erabilpenaren adibide bat da, –1 eta +1 arteko eta 0.1-eko tartea duten zenbakien sinua, kosinua eta tangentea inprimatzen baititu. Zenbakiak positiboak badira, beraien erro karratuak ere inprimatzen ditu.

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

main()
{
double aldagai=—1.0;

do{
   printf("%lf-ren sinua %lf da, kosinua %lf eta tangentea\
     %lf\n", aldagai, sin(aldagai), cos(aldagai),
     tan(aldagai));
   if (aldagai>=0.0)
     printf("eta bere erro karratua %lf da\n",
       sqrt(aldagai));
   aldagai+=0.1;
   } while (aldagai<1.1);
}
```

11.2. programa. *Funtzio matematikoen erabilpena.*

11.3. MEMORIA DINAMIKOA KUDEATZEKO FUNTZIOAK

Funtzio hauek memoria dinamikoki eskuratzea eta libratzea bideratzen dute. Erabiltzeko `<stdlib.h>` goiburuko fitxategia aipatuko da.

Lau funtzio erabiltzen dira memoria dinamikoa kudeatzeko: *malloc* eta *calloc* tokia eskuratzeko, *free* eskuratutako tokia itzultzeko eta *realloc* eskuratutako tokia berrantolatzeko. Lau funtzio hauek azaltzen dira ondoren.

```
void * malloc(int tam)
```

`tam` tamaina duen objektu batentzako lekua erreserbatzen du. Lehenengo byte-rako erakuslea itzuliko du. Lekua erreserbatzea ezinezkoa bada, edo `tam`-ek zero balioa badu, orduan `NULL` erakuslea itzuliko du. Erreserbatutako lekuan ez da hasierako balio berezirik jarriko.

```
void free(void *erak)
```

`free` funtzioak `erak` erakusleak adierazten duen lekua libratzen du. `erak`-en balioak aurretik burututako `calloc`, `malloc` edota `realloc` funtzioen bidez itzulitako balio bat izan behar du. `erak` `NULL` erakuslea bada, `free`-k ez du ezer egingo. Memoriaren area liberatzen denean, bertan zeuden balioak galtzen dira eta, printzipioz, ezin izango da area hori berriro erabili.

```
void * calloc(int osa_kop, int osa_luz)
```

`osa_luz` bytetako tamaina duten `osa_kop` osagaientzako jarraian dagoen lekua erreserbatzen du. Leku horretako bit guztiek zero balioa hartuko dute. `calloc`-ek espazio horretako lehenengo byte-a helbideratzen duen erakuslea itzuliko du. Espazioa ezin bada erreserbatu, edo `osa_kop` edo `osa_luz`-en balioa zero bada, orduan NULL erakuslea itzuliko du.

```
void * realloc(void *erak, int tam)
```

`calloc`, `malloc` edo `realloc` batez erreserbatuta zegoen tokia berrantolatzen du, `erak`-ek helbideratzen duen objektuaren tamaina aldatuz. Espazio berriari apuntatuko dion erakuslea itzuliko du, edo `NULL` eskaera ezin bada bete.

11.3 programan memoria dinamikoa erabiltzen duen modulu bat azaltzen da, memoria estatikoarekin gauza bera egingo lukeen programarekin kontrajarria, eta 11.1 eskeman dagozkion konpilazio eta estekaketa-programak.

```
/* lista bat memoria estatikoaz osatzea */

  #include <stdio.h>

  #define MAX 1000

  main ( )
  {
  int n,i,zenb;
  int list [MAX];

  printf ("zenbat balio?:");
  scanf ("%d",&n);
  if (n>MAX)
     {
     printf("errorea!\n")
     return;
     }
  for (i=0; i<n; i++)
     scanf ("%d",&list[i]);
  }

/* lista bat memoria dinamikoaz osatzea */

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

  main ( )
  {
  int n,i,zenb;
  int * list;

  printf ("zenbat balio?:");
  scanf ("%d",&n);
  list = (int *) malloc (n*sizeof (int));
  for (i=0; i<n; i++)
      scanf ("%d", list+i);

  }
```

11.3 programa. *Memoria dinamikoaren erabilpena.*

11.3 programan azaltzen denez, memoria estatikoa erabiltzen bada maximo bat erabili behar da listarako memori zatia definitzerakoan, eta benetako

balioa hori baino handiagoa bada errorea sortuko da (memoria dinamikoarekin konbinatzen ez bada behintzat). Beraz, luzera aldakorreko memori zati handi samarrak erabili behar direnean, memoria dinamikoa askoz hautapen egokiagoa da estatikoa baino; eraginkortasunari begira motelxeago izan badaiteke ere.

11.4. KARAKTERE-KATEEI BURUZKO FUNTZIOAK

Karaktere-kateak ez dira C-ren oinarrizko datuak eta, horren ondorioz, oinarrizko eragiketak burutzeko —katea bat esleitzeko edo bi katea konparatzeko adib.— ezin dira C-ren eragileak zuzenean erabili, ez bada egitura errepikakor baten barruan. Lana errazteko kateen gaineko liburutegiko funtzioak eskaintzen ditu C-k. Ondoren ikusiko denez C-ren liburutegiak karaktere-kateak tratatzeko funtzio erabilgarri asko ditu, eta funtzio horiek erabiltzeko `<string.h>` goiburuko fitxategia aipatu beharko da. Gogoratu behar da karaktere-katea bat karaktere nuluaz bukatutako karaktere-bektorea dela.

```
char * strcpy(char *kate1, const char *kate2)
```

`kate2` katearen osagaiak, bukaerako karaktere nulua barne, `kate1` erakusleak adierazten duen taulan kopiatzen ditu. `kate1` eta `kate2` teilakatzen badira, emaitza ezezaguna da. Funtzioaren emaitza bezala, `kate1`-en balioa itzuliko da.

```
char * strncpy(char *kate1, const char *kate2, int zenbat)
```

Funtzio honek `kate2`-ko gehienez `zenbat` karaktere kopiatzen ditu `kate1`-en barnean. `kate2` `zenbat` karaktere baino laburragoa bada, bukaeran karaktere nuluak gehituko dira, `zenbat` karaktere izan arte. `kate2`-k `zenbat` karaktere baino gehiago baditu, `kate1`-en geldituko den kopia ez da karaktere nuluaz amaituko.

```
char * strcat(char *kate1, const char *kate2)
```

kate2-ren kopia bat erantsiko da kate1-en bukaeran. kate1-en bukaerako karaktere nuluaren ordez kate2-ko lehen karakterea idatziko da. kate2-ren bukaerako karaktere nuluraino (hau ere barne egonik) kopiatuko dira.

```
char * strncat(char *kate1, const char *kate2, int zenbat)
```

kate2-ren lehen zenbat karaktereen kopia bat kateatuko zaio kate1-i.

```
int strcmp(const char *kate1, const char *kate2)
```

kate1 eta kate2 kateak konparatzen ditu. Bi kateak berdina badira 0 itzuliko da, kate1 kate2 baino handiagoa bada (ordena alfabetikoaren arabera eta ezkerretik eskuinera), zero baino handiagoa den osoko bat, eta kate1 kate2 baino txikiagoa baldin bada, aldiz, zenbaki negatibo bat itzuliko du.

```
int strncmp(const char *kate1, const char *kate2, int zenbat)
```

kate1 eta kate2 kateetako lehenengo zenbat karaktereak konparatzen ditu, strcmp-ren antzera.

```
int strlen(const char*kate)
```

kate-ren karaktereen kopurua itzultzen du.

```
char * strchr(const *kate, int kar)
```

Funtzio honek kar (char motara bihurtuta) karakterearen kate katean azaldutako lehen agerpenaren posizioa itzultzen du, osagaietariko bat baldin bada. Bestela, NULL erakuslea itzultzen du. Bukaerako karaktere nulua ere kate-ren osagaitzat hartzen da.

```
char * strrchr(const *kate, int kar)
```

Funtzio honek `kar` (`char` motara bihurtuta) karakterearen katea katean azaldutako azken agerpenaren posizioa itzultzen du, osagaietariko bat bada. Bestela, `NULL` erakuslea itzultzen du.

```
char * strstr(const char *kate1, const char kate2)
```

Funtzio honek `kate2` katearen (bukaerako karaktere nulua salbu) `kate1` katean azaldutako lehen agerpenaren posizioa itzultzen du, `kate1`-en parte bada. Bestela, `NULL` erakuslea itzultzen du.

```
char * strerror(int n)
```

`n` erroreari dagokion errore-mezurako erakuslea itzultzen du. Erabilgarria da `errno` aldagaiaren balioen esanahiak aurkitzeko.

11.4 programan beraietako batzuen erabilpena erakusten da. Programa honek katea batzuei balioak ematen dizkie. Ondoren, beraien luzerak lortzen ditu eta, azkenik, konparatzen ditu, konparaketari dagokion emaitzari egokitutako mezu bat inprimatuz.

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

main()
{
char kate1[40], kate2[40];
int luz;

strcpy(kate1, "Hau lehenengo katea da.\n");
strcpy(kate2, "Hau bigarrena.\n");
```

```
luz=strlen(kate1);
printf("%s katearen luzera %d da\n", kate1, luz);

luz=strlen(kate2);
printf("%s katearen luzera %d da\n", kate2, luz);

luz=strcmp(kate1, kate2);
if (luz>0)
  printf("%s alfabetikoki %s baino geroagokoa da\n", kate1,
    kate2);
else
  if (luz<0)
    printf("%s alfabetikoki %s baino lehenagokoa da\n",
     kate1, kate2);
  else
    printf("%s eta %s berdinak dira\n", kate1, kate2);
}
```

11.4. programa. Karaktere-kateei buruzko funtzioen erabilpena.

Ondoren azalduko diren `mem...` funtzioek objektuak karaktere-kateak balira bezala erabiltzen dituzte. Karaktere-kateen funtzioak ez bezala, memoriako zati batekin lan egiten dute eta ez dute karaktere nulua kontutan hartzen.

```
void * memchr(const void *alder, int kar, int n)
```

`alder` alderdiko lehen `n` karaktereetan `kar` karakterea azaltzen denentz begiratzen du. Hala bada, lehen agerpenari apuntatzen dion erakuslea itzultzen du; bestela, `NULL`.

```
int memcmp(const void *ald1, const void *ald2, int n)
```

`ald1` eta `ald2` alderdien lehenengo `n` karaktereak konparatzen ditu karakterez karaktere `strcmp`-en azaldutako emaitzak itzuliz.

```
void * memcpy(void *ald1, const void *ald2, int n)
```

`ald2`-ko lehenengo `n` karaktereak `ald1`-ean kopiatzen ditu. `ald1`-eko lehen karaktereari zuzendutako erakuslea itzultzen du.

```
void * memmove(void *ald1, const void *ald2, int n)
```

`ald2`-ko lehenengo `n` karaktereak `ald1`-an kopiatzen ditu. `memcpy`-k ez bezala, `memmove`-k gauza bera egingo du bi alderdiak teilakatzen badira ere. `ald1`-eko lehen karaktereari zuzendutako erakuslea itzultzen du.

```
void * memset(void *alder, int kar, int n)
```

`alder` erakusleak adierazten duen alderdiko lehen `n` karaktereak `kar` karakterearen kopiekin betetzen dira. `alder`-erako erakuslea itzultzen du.

11.5. BESTELAKO FUNTZIOAK

`<stdlib.h>` izenburu-liburutegian erazagutzen diren erabilpen handiko funtzio heterogeno batzuk biltzen dira hemen.

Bihurketak

Hasteko karaktere-katea formatuan dauden zenbakiak formatu bitarrera bihurtzeko funtzioak ditugu: `atoi`, `atof`, `atol`.

```
int atoi(const char *kate)
```

`kate` erakusleak helbideratzen duen karaktere-katea zenbakiez osaturik badago, dagokion osoko zenbakia itzuliko du. Adibidez, atoi("1937"); sententziaren emaitza 1937 balioa duen osoko bat da.

```
long int atol(const char *kate)
```

`kate` erakusleak helbideratzen duen karaktere-katea zenbakiez osaturik badago, dagokion osoko zenbaki luzea itzuliko du.

```
double atof(const char *kate)
```

`kate` erakusleak helbideratzen duen karaktere-katea zenbakiez osatuta badago, dagokion double motako zenbakia itzuliko du.

Ausazko zenbakiak

Ondoren ausazko zenbakiak sortzeko funtzioak azaltzen dira.

```
int rand(void)
```

`0` eta `RAND_MAX` balioen arteko osoko bat itzultzen du. `RAND_MAX` `<stdlib.h>` fitxategiaren barnean definitutako konstantea da. `rand` funtzioari jarraian egindako deiek balio desberdinak itzultzen dituztenez, sasiausazko zenbakiak sortzeko balio du.

```
void srand(unsigned int hasi)
```

`rand` funtzioak sasiausaz sortzen dituen zenbakietarako hasierako balio bat ematen du. rand erabili aurretik `srand`-i deirik egiten ez bazaio, hasierako balioa 1 izango da.

Bukaera bereziak

Bukaera ez-ohizkoa eragiteko funtzioak ere badaude.

```
void abort(void)
```

Programaren bukaera ez-ohizkoa eragiten du, SIGABRT seinalea gertatu izan balitz bezala. Inplementazioaren arabera dago bufferrak gordetzea eta fitxategiak ixtea.

```
int atexit(void (*funtzio)(void))
```

Honela, funtzio funtzioa exekutatuko da programa bukatzean, bertako eragiketak egikarituz. atexit-en bitartez erregistratzen diren funtzioak alderantzizko ordenan egikarituko dira. 32 funtzio erregistra daitezke.

Bilaketa eta sailkapena

Tauletan bilatzeko eta taulak sailkatzeko erabiltzen dira ondoko bi funtzioak.

```
void * bsearch (const void *gako, const void *oinarri, int n, int
     tam, int (*knp)(const void *arg1, const void *arg2))
```

gako-ak erakusten duen balioa bilatzen du n elementuko oinarri-taulan. Taularen elementuek goranzko ordenan sailkaturik egon behar dute eta tam taulako elementuen tamaina da. knp funtzioa erabiltzen da gakoa eta taulako osagaiak konparatzeko eta negatiboa, zeroa edo positiboa itzuli behar du hurrenez hurren lehen argumentua bigarrena baino txikiagoa, berdina edo handiagoa denean (askotan strcmp erabiliko da). bsearch-ek egokitzen den taularen elementuari zuzendutako erakuslea itzuliko du egokitzapena badago; bestela, NULL.

```
void qsort(void *oin, int n, int tam, int (*knp) (const void *,
     const void *))
```

`tam` tamainako `oin[0], ..., oin[n–1]` taularen elementuak goranzko ordenan sailkatzen ditu. `knp` funtzioa erabiltzen da sailkapenerako eta negatiboa, zeroa edo positiboa itzuli behar du hurrenez hurren lehen argumentua bigarrena baino txikiagoa, berdina edo handiagoa denean (askotan `strcmp` erabiliko da).

Sistemarekin lotutakoak

Sistemak mantentzen duen ingurunea (zenbait aldagairi balioak esleitzeko erabili ohi dena) atzitzeko eta komandoak egikaritzeko funtzioak azaltzen dira.

```
char * getenv(const char *izen)
```

Inguruneak mantentzen duen informazioa lortzeko balio du. Informazio hau inplementazioaren menpekoa den katea-bikoteetan dagoen inguruneko zerrenda batean da atzigarri, bikotea izenaz eta balioaz osaturik dagoela. `getenv` funtzioak zerrenda horretan begiratzen du `izen` izeneko bikotea aurkitu nahian. Aurkitzen badu, balioa definitzen duen karaktere-katearen erakuslea itzuliko du; bestela, `NULL` erakuslea.

```
int system(const char *prog)
```

`prog` izeneko exekutagarria bidaliko dio dagokion komando-interpretatzaileari exekuta dezan.

11.6. DENBORAREKIN LOTUTAKO FUNTZIOAK

<time.h> goiburuko fitxategiak eguna eta orduaren balioak tratatzeko motak eta funtzioak eskaintzen ditu. Bi denbora-mota bereizten dira: orokorra eta lokala.

Ondoko motak erabiltzen dira funtzio hauetan:

clock_t: sistemaren denbora adierazten duen zenbakia. Unitatea (*tick*) segundoaren frakzioa da.

time_t: denbora adierazteko balio aritmetikoa.

tm: denboraren osagaiak dituen egitura:

```
struct tm
{
int tm_sec; /* segundoak 0–59 */
int tm_min; /* minutuak 0–59 */
int tm_hour; /* eguneko ordua 0–23 */
int tm_mday; /* hileko eguna 1–31 */
int tm_mon; /* hilabetea urtarriletik hasita 0–11 */
int tm_year; /* urtea, 1900.ean hasita */
int tm_wday; /* asteko eguna, igandetik hasita 0–6 */
int tm_yday; /* urtarrilaren 1etik hasita, urteko eguna
                  0–365 */
int tm_isdst;
};
```

```
clock_t clock(void)
```

Programa deitzaileak erabili duen prozesadore-denboraren hurbilketa bat itzultzen du. Itzulitako balioa segundotara itzul daiteke <time.h>-ren barnean definituta dagoen CLK_TCK konstanteaz zatituz.

```
time_t time(time_t *derak)
```

Uneko denbora orokorra itzultzen du, balio aritmetiko bezala funtzioaren emaitza gisa eta, `derak` `NULL` ez bada, derak-k helbideratutako egituran ere.

```
char * asctime(const struct tm *derak)
```

Egutegi-denbora adierazten duen `*derak` karaktereez osatutako katea bihurtzen du, honako formatuarekin: `EEE HHH ee oo:mm:ss UUUU\n\0`

EEE	asteko eguna (alfabetikoa)
HHH	hilabetea (alfabetikoa)
ee	hileko eguna (zenbakitan)
oo:mm:ss	ordua:minutuak:segundoak (zenbakitan)
UUUU	urtea (zenbakitan)

```
double difftime(time_t denb1, time_t denb2)
```

`denb1` eta `denb2` artean dagoen diferentzia segundotan itzultzen du.

```
struct tm * gmtime(const time_t *derak)
```

`*derak` denbora orokorra `GMT` formatura bihurtzen du. Denbora ezin bada atzi, `NULL` itzultzen du.

```
char * ctime(const time_t *derak)
```

`*derak` denbora orokorra ASCII karaktere-katea baten bidez adierazten den denbora lokalera bihurtzen du.

```
struct tm * localtime(const time_t *derak)
```

`*derak` denbora orokorra denbora lokal bihurtzen du, emaitza `tm` egitura batean utziz eta egitura horri apuntatzen dion erakuslea itzuliz.

```
time_t mktime(struct tm *derak)
```

`*derak`-en dagoen denbora lokala egutegi-denborara itzultzen du, `localtime` eta `gmtime`-ren alderantzizkoa eginez.

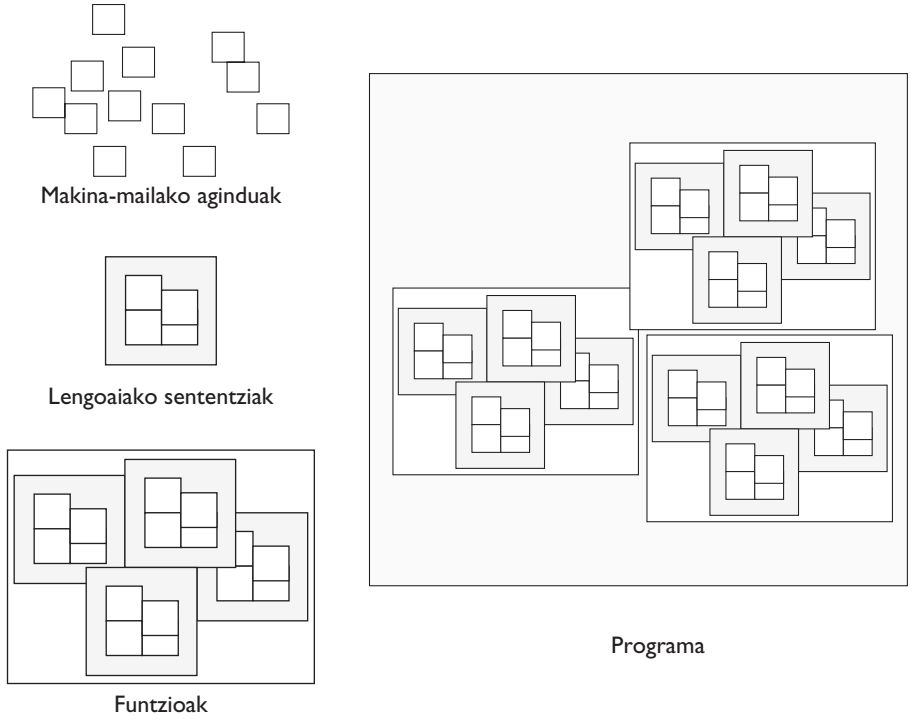
12. SOFTWARE-INGENIARITZA C-N

Gaur egungo programak gero eta sofistikatuagoak dira. Erabiltzaile partikular baten neurrira egindako programetatik erabiltzaile *generikoei* —behar amankomuneko erabiltzaile-multzoa— zuzendutako programa-multzo edo *paketee*tara pasa gara azken urteetan. Hortaz, leihoetan oinarritutako sarrera/irteera atseginagoa, aukera anitz eta norberaren beharretara parametrizatzeko aukera eskaintzen dute merkatuan agertzen doazen programa gehienak. Bilakaera horrek aldaketa asko dakar, programen garapenean duen eragina aldaketa garrantzitsuenetarikoa izanik.

Software-proiektu bat planteatzen denean, gaur egun, honako ezaugarri hauek kontuan hartu ohi dira:

- Modulartasuna: programak ezin dira monolitikoak izan ezta pertsona bakar batek garatuta izan. Programatzean erabiltzen diren unitateak azaltzen dira 12.1 irudian. Modulartasuna bultzatzeko aplikazioak modulutan banatzen dira, modulu bakoitzak besteekiko duen elkarrekintza edo interfazea oso ondo definituz; horrela, konplexutasuna banatzeaz gain, modulu bakoitza programatzaile desberdin batek gara dezakeelako eta zenbait modulu aplikazio desberdinetan aplika daitekeelako. Programatzeko modu honi *beheranzkoa* esaten zaio. Aipatutako konpilazio banatua oinarria izango da helburu hau lortzeko.

- Garraiagarritasuna: Programak ordenadore-mota edo *plataforma* desberdinetarako garatu nahi ohi dira marka edo sistema desberdinetatik independenteak izan daitezen (gainera bezero gehiago edukitzeko abantailarekin). Baldintzapeko konpilazioa oso lagungarria da bide horretan.



12.1 irudia. Programa bateko unitateak.

- Irakurgarritasuna eta mantenu erraza: Programen dimentsio handiagoak eta lan-taldean aritzeak areagotzen du programak idazterakoan argitasuna eta ulergarritasuna bultzatzeko beharra. Betidanik arazoak sortu dituzte programetan egin beharreko aldaketek, programen arazketa edo eguneratzea dela eta. Arazo hauek gutxiagotzeko oinarrizko teknikak zaharrak dira: iruzkin asko, konstante parametrizatuak, aldagaien izen esanguratsuak, dokumentazio aberats eta eguneratua, etab.

- Erabilera erraza eta interfaze atsegina: merkatu zabalaren bila joan ohi denez, ezin da suposatu balizko erabiltzailea informatikan espezializaturik dagoenik, ezta erabiltzaileak eskuliburu lodikoteak irakurtzera ohituta daudenik, beraz programaren erabilerak intuitiboa izan behar du eta erabiltzailearekiko hartu-emanak oso ondo diseinatu behar dira ahalik eta sinpleenak eta atseginenak izan daitezen.

C lengoaiaz programatzen dugunean aurreko ezaugarriak kontuan hartzen baditugu, C lengoaia egituratuegia ez bada ere, posiblea da software-proiektu konplexuak garatzea gehiegizko oztoporik gabe. Horretarako konpiladoreek, estekatzaileek (*linker* izenekoek) zein sistemaren software-ak eskaintzen dizkiguten laguntzak ahalik eta etekin handienaz erabili behar dira. Laguntza hauen aukerak azaltzea da kapitulu honen helburua.

Aipatu den bezala, eta oso erabilia izan arren, C, berez, ez da lengoaiarik egokiena software-proiektu handietarako. Eskaintzen duen malgutasuna eta eraginkortasuna bere alde egon arren, faltan dituen egituraketa, hierarkia eta kodearen auto-egiaztapena bere aurka daude gaur egun hedatzen ari diren objektuei zuzendutako lengoaien aurrean. Halere, merkatuan dauden software-pakete asko eta asko garatu dira C lengoaiaz. Gabezia hauek zuzendu nahian sortu zen C-ren eboluzioz sortu zen C++ izeneko lengoaia. 14. kapituluan lengoaia horren sarrera egiten da.

12.1. PROIEKTUEN GARAPENA

Software-proiektu bat garatzen denean ondoko urratsak bereizi ohi dira:

- Produktuaren definizioa eta diseinua. Produktuaren helburua eta erabiltzaileei eskainiko zaizkien funtzioak eta aukerak zehazten dira zehatz-mehatz. Egiteko modua ondoko urratsetarako uzten da, baina erabiliko diren sarrera-datuak eta datu-baseak eta lortuko diren emaitzak aurrikusi behar dira. Programa eta erabiltzailearekin gertatuko diren elkarrekintzak aurrikusi ohi dira urrats honetan. Hasierako definizioa ez da oso sakona izan behar eta birfindu daiteke balizko erabiltzaileekin kontsultatuz.

- Analisi zehatza. Produktuaren garapena nola egingo den erabakitzen da urrats honetan. Aplikazioa modulutan banatzea eta modulu guztien espezifikazio zehatza —parametroak, sarrera/irteera, etab.— egiten da. Modu-

luak egituratzen direnean eta lengoaiaren ahalmenaren arabera *datu-mota abstraktu* edo *objektu* izeneko unitateak lortzen dira. Moduluen arteko elkarrekintza, haien arteko egituratzea eta egitura zein aldagai globalen definizio zehatza funtsezkoa da lan-banaketa eraginkorra izan dadin. Horrez gain, funtsezko eragiketetarako erabiliko diren algoritmoak erabakiko dira, horretan etzango baita produktuaren eraginkortasuna eta konplexutasuna.

Urrats honetan moduluak garatzeko tresnak ere aukeratzen dira: programazio-lengoaia(k) —moduluaren arabera desberdin izan daiteke—, sistemaren softwareko tresnak —UNIXek, adib., oso tresna ahaltsuak eskaintzen ditu: *script*-ak egiteko aukera, *awk*, *pearl*, *lex* eta *yacc* etab., eta tresna horien bidez idatzi beharko litzatekeen kode asko aurrera daiteke— erosi beharreko liburutegiak edo tresnak —batzuetan norberak egin beharrean merkatuan daudenak erabiltzea da egokiena. Lan-banaketa egiteko garaia da ere, modulu batzuetarako kanpoko batzuekin elkarrekin lan egitea —azpikontratazioaren bidez edo kolaborazio-politika baten barruan— interesgarri gerta daiteke. Kostuak eta epeak zehaztuko dira ere funtsezkoa den urrats honetan. Ez gara honetaz gehiago luzatuko liburu honen esparrutik kanpoko gaia delako.

- Programazioa, arazketa eta probak. Kodea idazten hasi aurretik aurreko urratsaren emaitza eta dokumentazioa eduki behar da. Behin analisi zehatza edukiz gero programatzeari ekiten zaio analisiari ahalik eta kasu gehien eginez eta irakurgarritasuna eta mantenu erraza buruan edukiz. Honetan laguntzeko *CASE* (Konputagailuz lagunduriko software-ingeniaritza) izeneko tresnak sortu dira. Puntu honetaz luzatuko gara ondoko pasarteetan.

- Dokumentazioa eta mantenua. Analisia zein programazioa egiten den bitartean funtsezkoa da dokumentazio argi, oso eta koherentea sortzea, hauxe izango baita tresnarik garrantzitsuena proiektua garatzean sortzen

diren eragozpenak ebazteko, garatzen duten pertsonekin erabateko menpekotasuna saihesteko eta produktuaren balizko hobekuntzak garatzeko. Mantenurako dokumentazioa funtsezko tresna izan arren *CASE* tresnen erabilpena ere lagungarri da.

12.2. PROGRAMAZIO-ESTILOA

Programatzeko orduan aurretik aipatutako ezaugarriak funtsezkoak dira: *modulartasuna*, *mantenu erraza*, *irakurgarritasuna* eta *garraiagarritasuna*. Bide horretan oso inportantea da aurreko diseinuan ondo definitzea zeintzuk izango diren moduluak, zeintzuk haien arteko elkarrekintzak eta zein erabiltzailearekiko komunikazioa.

Software-proiektu bat garatzean funtsezkoa da programatzaileek estilo amankomun bat zehaztea eta erabiltzea, produktibitatea handitzeko, gataz-kak saihesteko eta mantenua errazteko. C lengoaian programatzen denean ondoko gomendioak eman ohi dira programazio-estilo zuzena lortzeko asmoz:

- Modulartasuna dela eta, ahalik eta aldagai global gutxien erabiltzea, parametroen bidezko datu-trukea bultzatuz. Aldagai globalen erabilerak detektatzeko zailak diren erroreak (albo-ondorioak) sortarazten dituzte.

- Aldagai globalak erabili behar direnean, iturburu-moduluaren esparrua gordetzea datu-mota abstraktu bat eraikiz. Datu-mota abstraktu batean datu bat edo datu-multzo bat eta horren gaineko eragiketak definitu egiten dira, eta, handik aurrera, definiziotik kanpo aipatutako datuak erabiltzeko definitutako eragiketak baino ez dira erabiliko.

- Programa eta funtzio luzeak saihestea funtzio laburrez osatutako moduluak lehenetsiz. Funtzioak mailaka antolatzen dira, zeren funtzio bat

definitzean algoritmo sinpleari konplexutasuna ekartzen dizkioten ekintzak zuzenean definitu beharrean funtzio batez ebaztuko dira, funtzio hauek geroago eta beheragoko mailan ebatziz. Honi *beheranzko programazioa* deritzo.

- "ad-hoc" soluzioetatik alde egitea eta ahalik eta funtzio orokorrenak eraikitzea, beste moduluetatik erreferentziatzeko aukera emanez. Behin funtzio bat programatzen hasiz gero, pentsatu behar da ea beste modulu edo programaren baterako interesgarri izan daitekeen, eta hala bada irtenbide orokorra bultzatuko da. Funtzioak orokortzeko konplexutasuna pittin bat handitzen da parametro gehixeago jarri behar ohi delako.

- Moduluen arteko loturak bideratzen dituzten erazagupenak goiburuko fitxategien bidez egitea. Honekin modulartasuna eta irakurgarritasuna irabazten da.

- Konstanteak ez dira zuzenean erabili behar, konstante parametrizatuek programaren parametrizazioa eta mantenua errazten baitituzte, kodea esanguratsuago bihurtuz. Oso inportantea da garraiagarritasunari begira ASCII-7 ez diren karaktere ez-estandarrak (haien artean ñ eta Ñ letrak) zuzenean ez erabiltzea, eta horien ordez izen bat jartzea, izen hori ordenadore bakoitzean dagokion karakterearekin parekatuz baldintzapeko konpilazioaren bidez.

- Programen irakurgarritasuna bultzatzeko, ahalik eta izen esanguratsuenak erabiltzea aldagaietarako. Hasiera batean detaile txikia iruditzen bada ere, funtsezkoa da oso, eta egiteko zaila disziplina hartzen ez bada.

- Aipatutako irakurgarritasunaren onerako ere, saihestu egin behar dira C-k bideratzen dituen espresio konplexuak. Saihesteko erabilera tipikoak bi hauek dira: asignazioak egitea baldintzetan eta autoinkrementuak erabiltzea esleipenetan.

- Kontrol-fluxu sinplea eraikitzea, egitura superkonplexuak saihestuz eta *goto* sententzia erabili gabe. Aipaturiko beheranzko programazioa erabil daiteke sinpletasuna lortzeko.

- Funtzioek gerta daitezkeen erroreak egiaztatu behar dituzte, eta itzulera-balioa erabili errorea gertatu den ala ez adierazteko.

UNIX sistemetako *lint* programak konpiladorearen lana sakontzen du, programazio-estiloaren aldetiko hankasartze larriak detektatuz.

12.3. KONPILAZIO BANATUA

Aurretik esandakoari jarraitzeko badago funtsezko bide bat: konpilazio banatua. Moduluak banan banan konpilatzeak abantaila besterik ez dakar. Ondorengo puntuetan labur daitezke konpilazio banatuaren abantailak:

- Modulu bakoitzean agertzen diren erroreak detekta daitezke beste moduluekin lotu gabe. Konpilazioarenak konpiladorearen esku dauden bitartean, exekuzioarenak detektatzeko proba berezi bat prestatu behar da modulu bakoitzeko. Honek, neketsua irudi badezake ere, benetan merezi du, ondorengo faseetako probak eta arazketa laburtuko dituelako.

- Programa osatzean erroreak detektatzen direnean ez dira modulu guztiak birkonpilatu behar, soilik errorea eragiten dutenak eta programa nagusia. Gainera, modulu bat programa desberdinetatik erabiltzen bada behin bakarrik konpilatuko da.

- Proiektuaren moduluak liburutegietan eta azpikatalogoetan egituratuta egoteko laguntzen du.

Konpilazio banatuaren nondik-norakoak aurreko kapituluetan azaldu dira. Funtsezko elementuak hauexek dira:

- Modulu bakoitzari dagozkion aldagai globalen eta funtzioen erazagupenak gordetzen dituzten goiburuko fitxategiak. Horrekin batera goiburuko fitxategiak txertatzeko `#include` sasiagindua.

- Konpiladorearen aukerak programak konpilatzeko exekutagarririk lortu gabe.

- Konpiladorearen edota estekatzailearen aukera objektu-moduluak biltzeko programa exekutagarri bakar bat lortzeko.

- Sistema eragilearen programaren baten laguntza (*ar* programa UNIX sistemetan) objektu-liburutegiak sortzeko eta kudeatzeko.

12.4. Programen garapenerako tresnak

Aipatutako programazio-estiloa egiaztatzeko eta liburutegiak maneiatzeko, programez gain gaur egungo sistema eragile gehienetan CASE motako tresna lagungarriak eskaintzen dira programen garapenerako.

Software-proiektu batean oso zaila izaten da programatzaileentzat modulu guztien arteko menpekotasunak ondo kontrolatzea eta eguneratze bakoitzak dakarren birkonpilatzeko beharrak zein moduluk eragiten dituen jakitea. Gainera, C lengoaian bereziki, goiburuko fitxategien erabilpenak zailtzen du are gehiago menpekotasun-erlazio hau. Lan hauez arduratzen da UNIX eta beste sistematan dagoen *make* programa.

Programa hau menpekotasunak definitzen dituen *make-fitxategi* batean oinarritzen da. Menpekotasunez gain, exekutagarrien izenak, moduluak aur-

kitzeko eta gordetzeko katalogoak, konpilazio-aukerak eta beste zenbait informazio islada daiteke aipatutako fitxategi horretan.

make programaz gain, sistema eragileek tresna interesgarri gehiago eskaintzen dute software-garapenerako. UNIXekoen artean hauek azpimarra daitezke.

- *grep* programa: fitxategietan funtzioak, espresioak edo aldagaiak aurkitzeko balio du. Programen arazketan erabili ohi da.

- *prof*: eraginkortasunaren azterketarako programa interesgarria, exekuzioan zehar funtzioek kontsumitzen duten denbora erakusten baitu.

- *flow*: Programen arazketan zehar funtzioen arteko erlazioak isladatzeko erabiltzen den tresna.

12.5. Arazketa

Aurretik azaldutako laguntzak oso inportanteak izanda ere, programaren exekuzioa zehatz-mehatz aztertzea bideratzen duen programak, programa *araztaileak* edo *zorri-kentzaileak*, ezinbestekoak dira exekuzio-erroreak detektatu ahal izateko arrazoizko denboran. Programa hauek funtsezkoak dira edozein programazio-lengoaiatan, baina, are gehiago, mihiztatzaileen ezaugarriak dituen eta hain malgua den C lengoaian. C-n erakusleak maneiatzean sortzen diren erroreak dira horren lekuko.

Programa araztaile desberdinak daude konpiladorearen arabera. Guztiek dituzten aukera nagusiak honako hauek dira:

- Sententziaz sententzia moduko exekuzioa, kontrol-fluxua zehatz-mehatz jarraitzeko eta aginduaren ondorioz gertatutako aldaketaz jabetu ahal izateko.

- Geratze-puntuak ezartzea programan zehar, programa puntu horretatik pasatzen denean edo baldintzaren bat betetzen denean gera dadin. Momentu horretan gainerako aukerak ditu eskuragarri programatzaileak.

- Programa-zatiak pantailaratzea exekuzio-puntua erreferentziatzat har daitekeela.

- Aldagaien balioa azalaraztea momentu batean edo programaren exekuzioa gelditzen denean. Aldagai horien balioa aldatzeko aukera ere badago.

- Exekuzioa puntu batetik abiatzea edo jarraitzea.

Programen arazketa "artea" da eta laguntza-programez gain esperientzia eta pazientzia funtsezko tresnak dira lan horretan. Arazketan lau urrats hauek bereizi ohi dira, bakoitzaren barruan proba-zuzenketa bigiztan aritzen delarik:

1) Modulu bakoitzaren egiaztapena, horretarako modulu bakoitzerako programa nagusi txiki gehigarri bat eginez.

2) Programa osoaren lehen proba sinplea egitea orokorrean ondo funtzionatzen duenentz egiaztatzeko. Urrats hau gainditu ondoren *alfa bertsioa* lortzen dela esaten da.

3) Programaren froga sakona eta exhaustiboa prestatu eta egin. Fase honek analisi-garaian pentsaturik egon behar du funtsezkoa baita programaren kalitatea ziurtatu nahi bada. Urrats hau gainditu ondoren *beta bertsioa* lortzen dela esaten da.

4) Zenbait erabiltzaile potentzialen eskuetan jartzen da programa, erabil ondoren akatsen zein balizko hobekuntzen berri eman dezaten. Erabiltzaileen aukeraketa funtsezkoa da urrats honetan.

13. APLIKAZIOAK

Kapitulu honetan aurreko kapituluetan aztertutako kontzeptu, espresio eta egitura gehienak praktikan jarri nahian bi adibide aukeratu dugu: 1) liburutegi (liburuena) baterako aplikazio txiki bat eta 2) liburutegi (programena) batean gordetzen den *lista* datu-mota berriaren definizioa.

13.1. LIBURUTEGI TXIKI BATEN KUDEAKETA

Hasteko, liburutegi txiki baten kudeaketa burutuko duen aplikazioa azalduko da.

Lehenengo, datuak (liburuei buruzkoak kasu honetan) nola adieraziko diren erabaki behar da. Ondoren, datu horiekin burutu daitezkeen eragiketak zehaztuko dira, honela datu-mota osoa zehaztuz. Azkenik, datu eta eragiketa horiek erabiliz, liburutegia kudeatuko duten funtzioak (funtzio nagusia barne) adieraziko dira. Beraz, adibide honen programazioa goranzkoa da hasiera batean, nahiz eta ondoren beheranzkoa izan (programa nagusia zehaztu ondoren, bertan azaltzen diren funtzio lagungarriak geroago garatuko baitira).

Interesatzen zaigun aurreneko erabakia erregistroari buruzkoa da, hau da, liburu bakoitzari buruz mantenduko dugun informazioa zein den zehaztea. Pentsa dezagun, sinplifikazio bat dela kontuan hartuz, ondoko informazioa gordeko dela: liburu-kodea, izenburua, idazlea, urtea eta liburu-motaren kodea. Erregistro hau funtzio gehienetan erabiliko denez, `liburu.h` fitxategian erregistroaren definizioa egingo dugu (ikus 13.1. programa).

```
/* liburu. h */

struct liburu
{
int kod;
char izen[40], idazle[40];
int urte, mota;
};
```

13.1. programa. *Erregistroaren definizioa.*

Erregistro honekin oinarrizko bi eragiketa egingo dira: informazioa teklatutik irakurtzea eta pantailan idaztea. Funtzio hauek aparte konpilatuko ditugu; beraz, fitxategi batean kokatuko dira, `lib_oinarri.c` izeneko fitxategian gure kasuan, 13.2. programan ikusten den kodea edukiz.

```
/* lib_oinarri.c – erregistro baten irakurketa eta idazketa */

#include <stdio.h>
#include "liburu.h"

void garbi_pantaila(void); /*geroago definituta. Erazagupena */

void irakur (struct liburu *plib)
{
int aux1;
char aux2[80], c;

garbi_pantaila ( );
printf ("liburuaren kodea: \n");
scanf ("%d", &aux1);
plib->kod = aux1;
scanf ("%c", &c);
printf ("\nliburuaren izenburua: \n");
gets (aux2);
strncpy (plib->izen, aux2, 40);
if (strlen(aux2)>=40)
  plib->izen[39]='\0';
printf ("\nliburuaren idazlea: \n");
gets (aux2);
strncpy (plib->idazle, aux2, 40);
if (strlen(aux2)>=40)
  plib->idazle[39]='\0';
```

```
printf ("\nliburuaren urtea: \n");
scanf ("%d", &aux1);
plib->urte = aux1;
printf ("\nliburu-motaren kodea: \n");
scanf ("%d", &aux1);
plib->mota = aux1;
}

void idatz (struct liburu *plib)
{
char c[2];

garbi_pantaila ( );
printf ("liburuaren kodea: %d\n", plib->kod);
printf ("liburuaren izenburua: %s\n", plib->izen);
printf ("liburuaren idazlea: %s\n", plib->idazle);
printf ("liburuaren urtea: %d\n", plib->urte);
printf ("liburu-mota: %d\n", plib->mota);
printf ("jarraitzeko sakatu edozein tekla!");
scanf ("%c", &c[0]);
}

void garbi_pantaila (void)
{
int i;

for (i = 0; i < 24; i++) printf ("\n");
}
```

13.2. programa. *Oinarrizko funtzioak*

Pantailaren maneiua hobe daiteke. Horretarako sistema berrietan leiho-sistema (windows) erabili ohi da, datuak teklatu zein saguaren (mouse) bidez sar daitezkeelarik. Zoritxarrez sistema hauek programatzeko liburutegiak ez dira estandarrak eta horrexegatik ez ditugu hemen azalduko.

Pentsa dezagun orain gure liburuei buruzko datu-basean zein eragiketa mota egin daitezkeen. Hasiera batean gutxienez ondoko eragiketak izan beharko lirateke posible: liburuen datu-sarrera, liburuen listatua eta liburuei buruzko kontsulta. Liburuen kontsultari dagokionean ondoko irizpideak hartu beharko lirateke kontutan: kodea (horrela liburu bakarraren informa-

zioa lortuz), idazlea, urtea (daten arteko tarteak onartuz) eta mota. Kontsulta hauek azkar egiteko bi aukera daude: batetik liburuen datu-basea txikia izan eta memoriara ekarri hasieran, horrela kontsultak azkarrago burutuz, eta bestetik, indizeak eraiki beste fitxategi batzuetan datu-basea eraiki ahala, eta kontsultak egitean indize hauek erabiliz datu-basea irakurri. Bigarren hau orokorragoa da, eta horrexegatik erabiltzen da datu-base sistemetan, baina liburu honen helburuetatik kanpo dago; oso programa-multzo konplexua gertatzen baita. Beraz, lehenengoari ekingo diogu. Dena den, liburuaren kodea sekuentziala dela kontsideratuko dugu. Horrela kodearen araberako kontsulta berehalakoa izango da; irakurketa sekuentziala, memorian edo fitxategian, ez baita egin beharko. Beraz, programa nagusiak egingo duena honako hau izango da: datu-basea duen fitxategia ireki, fitxategia memoriara ekarri, pantailan menu bat azaldu aukera desberdinak emanez, aukera bat irakurri eta aukeraren arabera funtzio berezitu bati deitu. 13.3. programan hori burutzen duen programa dugu.

```
/* nagusi.c – programa nagusia: menu */
# include <stdio.h>
# include "liburu.h"

# define MAX 1000

void main (int argc, char *argv[])
/* argumentua datu-basearen izena da*/
{
/* ondoren beste modulutako funtzioen erazagupena */
extern void sarrera (struct liburu tau[], char ize[]),
   listatu (struct liburu t[]),
   kon_kod (struct liburu lib[]),
   kon_beste (struct liburu taula[], int auke);

void irakur_osoa (FILE * fd, struct liburu ta[]);

FILE *fd;
int auk;
struct liburu lib[MAX]; /*memoria estatikoa*/

if (argc<2)
   {
   printf("Ez duzu argumenturik eman\n");
   return;
   }
```

```
for (i=0; i<MAX; i++) /* hasieraketa */
   {
   lib[i].kod=-1;
   lib[i].izen[0]='\0';
   lib[i].idazle[0]='\0';
   }
fd = fopen (argv[1], "r");
irakur_osoa (fd, lib);
while (1) /* etengabe jarraitu exit gertatu arte */
   {
   garbi_pantaila ( );
   printf ("1- sarrera \n2- listatua \n");
   printf ("3- kontsulta kodearen arabera \n");
   printf ("4- kontsulta idazlearen arabera \n");
   printf ("5- kontsulta dataren arabera \n");
   printf ("6- kontsulta motaren arabera \n");
   printf ("0- bukatu \n \n \n ZURE AUKERA:");
   scanf ("%d", &auk);
   switch (auk)
     {
     case 0: return;
     case 1: sarrera (lib, argv[1]);
             break;
     case 2: listatu (lib);
             break;
     case 3: kon_kod (lib);
             break;
     case 4:
     case 5:
     case 6: kon_beste (lib, auk);/* hiru kasuetan */
             break;
     default: printf("ERROREA-aukeratu: 0-6");
   scanf("%d", &auk); /* edozer sakatu */
     }
   }
}

void irakur_osoa (FILE *fd, struct liburu taula[])
{
void sorrera(char ler[], struct liburu *p);
int i;
char lerro[120];

lerro[119]='\0';
for (i=0; i<MAX ; i++)
   if(fread (lerro, sizeof(lerro)-1, 1, fd)== EOF)
     break;
sorrera(lerro, &taula[i]);
fclose (fd);
}
```

```
void sorrera(char lerro[], struct liburu *plib)
{
/*lerrotik irakurritakoa liburu egituran kokatzen du */
}
```

13.3. programa. Programa nagusia.

Konturatu `sarrera`, `listatu`, `kon_kod` eta `kon_beste` funtzioak programa nagusian definitu gabe daudela, baina `extern` bezala erazagutu ditugula. Funtzio hauen definizioa lau fitxategi desberdinetan egin daiteke, 13.4., 13.5., eta 13.6. programetan azaltzen den bezala; `listatu` funtzioa ez baitugu azalduko. `garbi_pantaila` funtzioa ere `extern` bezala erazagutu da; `lib_oinarria.c` fitxategian baitago. Beraz, ez da definitu behar. Ezta fitxategi horren `#include` egin behar ere; estekatzaileak (linker) lotura egingo baitu.

```
/* sarrera.c */

#include <stdio.h>
#include "liburu.h"
#define MAX 1000

/* lerroak 120 karakterekoak izango dira eta bai idazlea eta
bai liburuaren izena, 39 karaktere gehi bukaerakoa ('\0')
izango dira; zenbakiak (kodea, urtea eta mota- gehienez 6
digitukoak izango dira */

void itoa (int zen, char kate[], int oin)
/* zenbaki bat string bihurtzea.*/
{
char s[6];
int i, j;

for (i=0; (zen)>0; i++)
   {
   s[i]=zen%oin+'0';
   zen/=oin;
   }
s[i]='\0';
if (!i)
   {
   kate[0]='0';
   kate[1]='\0';
   return;
   }
```

```
for (j=i-1; j>=0; j—)
  kate[j]=s[i-j-1];
kate[i]=’\0’;
}

void eratu (struct liburu lib, char lerro[])
{
char zenba[6];
int i;

lerro[0]='\0';
itoa(lib.kod, zenba, 10);
strcat(lerro, zenba);
strcat(lerro, " ");
strcat(lerro, lib.izen);
for (i=strlen(lib.izen); i<39; i++)
  strcat(lerro, " ");
strcat(lerro, lib.idazle);
for (i=strlen(lib.idazle); i<39; i++)
  strcat(lerro, " ");
itoa(lib.urte, zenba, 10);
strcat(lerro, zenba);
strcat(lerro, " ");
itoa(lib.mota, zenba, 10);
strcat(lerro, zenba);
for (i=strlen(lerro); i<118; i++)
  strcat(lerro, " ");
lerro[118]='\n';
lerro[119]='\0';
}

void sarrera (struct liburu taula[], char izena[])
{
FILE * fitx2;
struct liburu berri;
char lerroa[120];
extern void irakur (struct liburu *);
void errore (int);

irakur (&berri);
if (berri.kod<0)
  printf("Kodeak positiboa izan behar du!!!\n");
if (berri.kod >= MAX)
  errore (berri.kod);
else
  {
  fitx2 = fopen (izena, "r+");
  eratu(berri, lerroa);
  fseek (fitx2, berri.kod *strlen (lerroa), SEEK_SET);
```

```
   fwrite (lerroa, strlen (lerroa), 1, fitx2);
   fclose (fitx2);
   taula [berri.kod] = berri; /* taulan sartu */
   }
}

void errore (int kod)
{
char c[2];

printf("kodea 1000 baino txikiagoa, beraz ez %d\n", kod);
      scanf("%c", &c[0]);
}
```

13.4. programa. Liburuen sarrera.

```
/* kon_kod.c */

#include <stdio.h>
#include "liburu.h"
#define MAX 1000

void kon_kod (struct liburu taula[])
{
int kod;
extern void idatz (struct liburu *), errore (int);

printf ("liburuaren kodea: \n");
scanf ("%d", &kod);
if (kod >= MAX)
  errore (kod);
else
  if (kod>=0 && taula[kod].kod == kod)
    idatz (&taula[kod]);
}
```

13.5. programa. Kontsulta kodearen arabera.

```
/* kon_beste.c */

#include <stdio.h>
#include "liburu.h"
#define MAX 1000

void kon_beste (struct liburu taula[], int aukera)
{
int i, mota, data1, data2;
char idazle[40], idaz[80], c;
extern void idatz (struct liburu *);
```

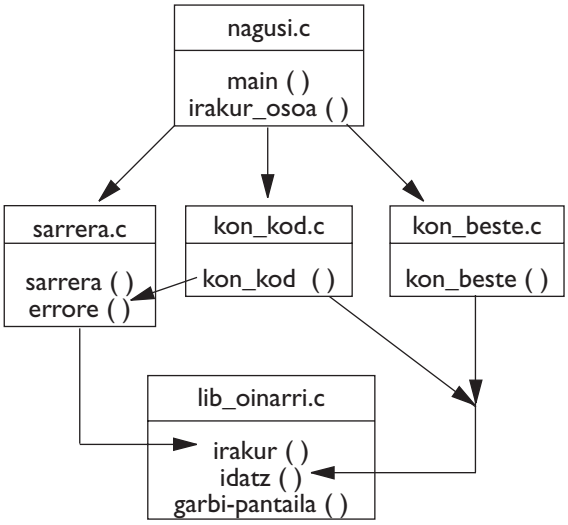
```
switch (aukera)
   {
   case 4: printf ("idazlea: \n");
           scanf("%c", &c);
           gets (idaz);
           strncpy(idazle, idaz, 40);
           if (strlen(idaz)>=40)
           idazle[39]='\0';
           break;
   case 5: printf("lehen urtea eta azkena:\n");
           scanf ("%d %d", &data1, &data2);
           break;
   case 6: printf ("mota: \n");
           scanf ("%d", &mota);
           break;
   }
for (i = 0; i < MAX; i++)
   if((aukera == 4 && strlen(idazle) &&
     (!strcmp (idazle,taula[i].idazle) ||
     !strncmp(idazle, taula[i].idazle, strlen(idazle))))
     || (aukera == 6 && mota == taula[i].mota &&
     taula[i].kod>=0) || (aukera == 5 &&
     data1<=taula [i].urte && data2>=taula[i].urte &&
     taula[i].kod>=0))
     idatz (&taula[i]);
}
```

13.6. programa. *Kontsultarako errutina orokorra.*

Erroreen maneiua askoz ere konplexuago egin zitekeen, baina ahalik eta laburren egin dugu garrantzia ekintza nagusiei ematearren.

Beraz, 6 modulu ditugu: `liburu.h`, `lib_oinarri.c`, `nagusi.c`, `sarrera.c`, `kon_kod.c` eta `kon_beste.c`. `listatu.c` modulua ez da azaltzen `idatz` funtzioarekin antza handia duelako, funtzio hori erabiltzeaz gain inprimagailua maneiatzea eta `for` egitura batez kontrola eramatea besterik egin gabe.

Modulu hauen dependentzia hierarkikoa 13.1. irudian azaltzen da. Bertan ez da agertzen `liburu.h` modulua goiburuko fitxategia delako eta, beraz, beste moduluetan `#include` sasiaginduaren bidez txertatzen delako.



13.1. irudia. Funtzioen arteko dependentziak.

Konpilatzeko, lehenbizi lau modulu ez-nagusiak banan-banan konpilatzen dira, dagozkien `.o` objektu-moduluak lortuz. Ondoren, programa nagusia konpilatu eta estekatu egiten da, azken honetarako aurretik lortutako objektu-moduluen izenak aipatu egin behar direlarik. Estekaketaren ondorioz moduluen arteko erreferentzia gurutzatuak ebatzi eta programa exekutagarri bakarra lortzen da.

13.2. LISTEN KUDEAKETA

Segidan erabilpen orokorreko funtzio-multzo bat definituko dugu, funtzio--multzo hau liburutegi batean gorde eta edozein aplikaziotatik erabil daitekeelarik. Definituko dugun funtzio-multzoak listak maneiatzeko datu-mota berria osatuko du, eta aplikazio batek listak erabili nahi baditu, datu-mota berri horren bidez egin dezake C liburutegi estandarreko funtzioak balira bezala. Beraz, has gaitezen listak eta dagozkien funtzioak definitzen.

Lista bat bi erakuslek definituko dute, bata lehen osagaia adierazteko eta bestea azkena adierazteko. Listaren osagai bakoitzak badu erakusle bat listaren ondoko osagaia adierazteko, osagaiaren informazioaren erakuslea eta lehentasuna, zeren osagaiak lehentasunaren arabera txertatzeko aukera baitago. 13.7. programan `lista_def.h` fitxategiaren edukia azaltzen da, bertan lista eta osagaiaren egiturak definiturik utziz.

```
/* lista_def.h */

struct elem
{
struct elem *ondoko;
int lehentasuna;
char *infor;
};

struct lista
{
struct elem *lehena;
struct elem *azkena;
};

#define NIL (struct elem *) 0 /* lista hutserako */
#define BOOL short /* funtzio boolearretarako */
#define TRUE 1
#define FALSE 0
```

13.7. programa. `lista_def.h` *definizioak.*

Ondoren aplikazioek beharko dituzten listen gaineko funtzioak diseinatu eta definitu egin behar dira.

Diseinuaren aldetik, eta goranzko programazioa erabiliz, funtzioak zeintzuk eta zein motatakoak izango diren eta zein parametro izango dituzten da zehaztu behar dena. Hasteko, lista huts bat sortzeko funtzioa beharko dugu; `sortu` izenekoa eta parametro bakartzat listaren erakuslea izango duena. Lista hutsa den ala ez jakiteko, funtzio boolear bat izango dugu parametro bakarrarekin; `huts` funtzioa alegia. Listetan osagaiak erantsi eta kendu egi-

ten direnez, hauexek izango dira funtzio nagusiak: bukaeran eranstea (lehentasuna kontuan hartu gabe) edo lehentasunaren arabera eranstea, eta lehen osagaia kentzea zein lehentasun jakin baten lehen osagaia kentzea. Lau funtzio hauei `lotu`, `txertatu`, `lehena` eta `atera` izenak dagozkie eta ondoko ezaugarriak egokituko dizkiegu:

- `lotu` eta `txertatu` eransteko funtzioak direnez, bi parametro izango dituzte, lista eta osagaia (erakusleen bidez), eta ez dute baliorik itzuliko.

- `lehena` eta `atera` funtzioek osagai bat listatik kentzen dutenez, osagai horren erakuslea itzuliko dute, bietan parametro bakartzat listaren erakuslea agertuz.

- `atera` funtzioan listaren gain beste parametro bat zehaztu behar da, lehentasuna hain zuzen; lehentasun horretako lehen osagaia kendu behar baita listatik.

```
/* lista_fun.h */

extern void sortu(struct lista *);
extern BOOL huts(struct lista *);
extern void lotu(struct lista *, struct elem *);
extern void txertatu(struct lista *, struct elem *);
extern struct elem * lehena(struct lista *);
extern struct elem * atera(struct lista *, int);
```

13.8. programa. *Funtzioen prototipoak.*

Diseinatutako funtzioen prototipoak 13.8 programan azaltzen dira eta `lista_fun.h` fitxategian isladatuko dira; aplikazioetan fitxategi hau listen gaineko funtzioak erazagutzeko erabiliko baita.

Ondoko programetan 13.9.etik 13.14.era arte, funtzioen definizioari ekingo diogu; definizioak denak fitxategi batean edo fitxategi bereizietan egon daitezke. Fitxategi bakoitzeko hasieran `#include lista_def.h` sententzia zehaztu beharko da.

```
/* sortu.c */
#include "lista_def.h"

void sortu (struct lista *plista)
{
plista -> lehena = NIL;
plista -> azkena = NIL;
}
```

13.9. programa. `sortu` *funtzioaren definizioa.*

```
/* huts.c */
#include "lista_def.h"

BOOL huts (struct lista *plista)
{
return (plista -> lehena == NIL); /* balio boolearra */
}
```

13.10. programa. `huts` *funtzioaren definizioa.*

```
/* lotu.c */
#include "lista_def.h"

void lotu (struct lista *plista, struct elem *pelem)
{  /* elementu bat listaren azkenari lotzen zaio */
struct elem * paux;

paux = plista -> azkena;
if (paux == NIL)
  plista -> lehena = pelem;
else
  paux -> ondoko = pelem;
pelem -> ondoko = NIL;
plista -> azkena = pelem;
}
```

13.11. programa. `lotu` *funtzioaren definizioa.*

```
/* txertatu.c */
#include "lista_def.h"

void txertatu (struct lista *plista, struct elem *pelem)
{  /* elementu bat lehentasunaren arabera lotzen da */
struct elem *paux, *paurreko;

paurreko = NIL;
for (paux = plista -> lehena; paux!= NIL;
     paux = paux -> ondoko)
  {
```

```
/* lista korritzen da txertatu behar den tokia aurkitu arte */
  if (pelem->lehentasuna <= paux->lehentasuna)
    paurreko = paux;
  else
    break; /* hemen txertatu */
  }
/* paurrekoa-n aurrekoa edo NIL lehena txertatzeko;
  paux-n ondoko edo NIL azkena txertatzeko */
pelem -> ondoko = paux;
if (paurreko!= NIL)
  paurreko -> ondoko = pelem;
else
  plista -> lehena = pelem;
if (paux == NIL)
  plista -> azkena = pelem;
}
```

13.12. programa. txertatu *funtzioaren definizioa.*

```
/* lehena.c */
#include "lista_def.h"

struct elem *lehena (struct lista *plista)
{
struct elem * paux;

paux = plista -> lehena;
if (paux!= NIL)
  {
  plista -> lehena = paux -> ondoko;
  if (plista -> azkena == paux)
    plista -> azkena = NIL;
  }
return (paux);
}
```

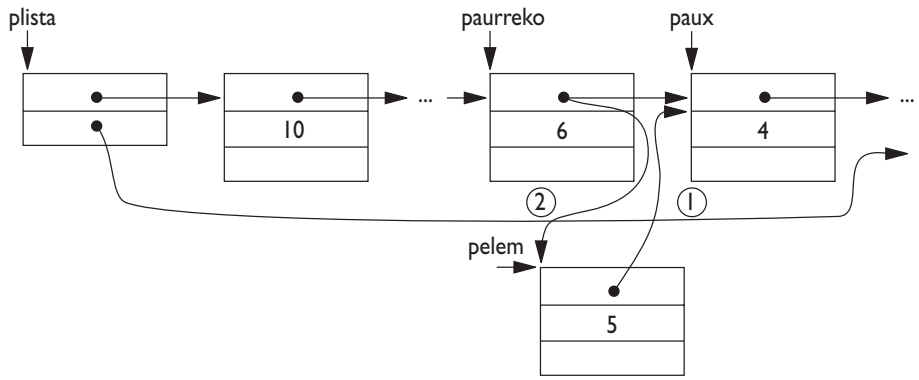
13.13. programa. lehena *funtzioaren definizioa.*

```
/* atera.c */
#include "lista_def.h"

struct elem *atera (struct lista *plista, int lehent)
{
struct elem *paux, *paurreko;

paurreko = NIL;
for (paux = plista -> lehena; paux!= NIL;
     paux = paux -> ondoko )
  {
```

```
  if (paux -> lehentasuna > lehent)
    paurreko = paux;
  if (paux -> lehentasuna == lehent)
    { /* listatik kendu */
    if (paurreko != NIL)
      paurreko -> ondoko = paux -> ondoko;
    else
      plista -> lehena = paux -> ondoko;
    if (paux -> ondoko == NIL)
      plista -> azkena = paurreko;
    return (paux);
    }
  if (paux -> lehentasuna < lehent)
    return (NIL);
  }
return (NIL);
}
```

13.14. programa.. `atera` *funtzioaren definizioa.*

13.2. irudian egoera arrunta isladatzen da, hau da, txertatzen den elementua lehena edo azkena ez denekoa.



13.2. irudia. `txertatu` funtzioaren ondorioa.

Kodean ikus daitekeenez `lotu` eta `txertatu` funtzioen alderantzizkoak dira `lehena` eta `atera` funtzioak; elementu bat lotu beharrean kendu egiten baitute.

Funtzio hauek konpilatu ondoren, liburutegi batean gordetzea komeni da; horrela aplikazioetatik erreferentziatzen direnean estekatzeko prozedura

erraztu egiten baita. Aurreko adibideko liburuei buruzko funtzioak ez ziren liburutegi batean kokatzen, aplikazio jakin baterako idatzita zeudelako, baina adibide honetako funtzioen helburua edozein aplikaziotarako datu-mota berria eskaintzea denez, liburutegi batean kokatzea da prozedura estandarra.

Objektu-moduluak liburutegietan gordetzeko sistema eragile bakoitzak komandoren bat dauka, aurreko kapituluan esandakoaren arabera. Liburutegia osatuta edukiz gero, ikus dezagun liburutegian gordetako aipatutako funtzioak nola erabil daitezkeen.

Adibide batez azalduko dugu. Pentsa dezagun aurreko kapituluko datu-basean liburuak publikazio-dataren arabera listatu nahi ditugula. 13.15. programan azaltzen denaren arabera, lista bat definituko dugu eta listaren osagaiak memoria dinamikoan kokatuko dira. Ondokorako erakusleak, lehentasun gisa erabiliko den publikazio-datak eta dagokion *liburu* erregistrorako erakusleak osatzen dute listaren osagai edo elementu bakoitza. Liburu guztiak korrituz lista osatu ondoren, osagaiak ordenan atera eta inprimatuko dira.

```
/* sailkatu.c */

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "lista_def.h"
#include "lista_fun.h"
#include "liburu.h"
#define MAX 1000

void main (int argc, char *argv[])
{
extern void irakur_osoa (FILE * fp, struct liburu taula[]);
struct liburu lib [MAX];
struct lista lis;
struct elem *p;
int i;
FILE * fd;
```

```
if (argc < 2)
   {
   printf("Argumentua pasatzea ahaztu zaizu\n");
   return;
   }
for (i=0; i<MAX; i++)
   {
   lib[i].kod=-1;
   lib[i].izen[0]='\0';
   lib[i].idazle[0]='\0';
   }
fd = fopen (argv[1], "r");
irakur_osoa (fd, lib); /* memorian edukitzeko */
sortu (&lis); /* lista hutsa */
for (i = 0; i < MAX; i++) /* listan txertatzeko */
   {
   if (lib[i].kod >= 0)
      {
      p=(struct elem *)malloc(sizeof(struct elem));
      p -> infor = (char *) &lib[i];
      p -> lehentasuna = lib[i].urte;
      txertatu (&lis, p);
      }
   }
p = lehena (&lis);
while (p != NIL) /* ordenan atzitzeko */
   {
   idatz (p -> infor);
   p = lehena (&lis);
   }
}
```

13.15. programa. *Listen gaineko funtzioen erabilpena liburuak sailkatzeko.*

Konturatu lista eta elementuak, orain arte memoria asignaturik ez zeukatenez, programa nagusian definitu direla, bata memoria estatikoan eta memoria dinamikoan besteak. Bestetik, listei dagozkien funtzioen erazagupena ez da egiten, `lista_fun.h` fitxategiaren bidez txertatzen dira eta.

Bukatzeko, esan beharra dago programa nagusia estekatzean listei dagokien liburutegia aipatu beharko dela.

14. C++: LEHEN URRATSAK

Basilio Sierrak idatzia

Hasi baino lehen esan behar da C eta C++ arteko desberdintasun nagusia ez dagoela lengoaiaren sintaxian, filosofian baizik. Sintaktikoki antzerakoak izan arren, zerikusi gutxi dute erabiltzerako orduan. Idazle askok eta askok C++ C lengoaiaren oinordekotzat hartzen du, baina, analisi sakona eginez gero, lengoaiaren bilakaera baino filosofiaren moldaketa izan dela esan dezakegu.

14.1. OBJEKTUEI ZUZENDUTAKO PROGRAMAZIOA (OZP)

Ohiko programazio-estiloan algoritmoa da garrantzi gehien duena, eta programak funtzionatu dezan aldagaiak erazagutzen dira. OZPk filosofi aldaketa dakar: orain ez dago algoritmorik, objektuak (*klaseak*) eta objektuen arteko komunikazioak (*mezuak*) baizik. Ez dago programaren jarraipen lineala egiterik.

OZPa ez da ideia berria. Bere kontzeptu nagusiak —datu-abstrakzioa, herentzia eta polimorfismoa— aspaldi erabiliak izan dira Simula67 eta Smal--Talk bezalako lengoaietan. Programatzaileen artean sortutako interes handikorra da berri bihurtzen duena.

Datu-abstrakzioa da OZPren elementu bereizgarrietako bat, eta datu--abstrakzioa C++ lengoaian klaseen bidez burutzen da. Klase bat (*class* ingelesez) jatorrizko C lengoaiaren struct datu-motaren hedapena da. Bi gehiketa adierazgarriak egin zaizkio *struct*-ari:

- klasearen barnean aldagaiez gain funtzioak ere onartzen dira, hurrenez hurren *aldagai kideak* eta *funtzio kideak* deitzen direnak.

- kideak *publikoak* edo *pribatuak* izan daitezke.

Kide pribatuak klasearen funtzioen bidez baino ezin dira atzitu, publikoak aldiz kanpotik —hau da, beste klaseetatik— ere ikusten diren artean. Aurrakoa esaten ez den bitartean kideak pribatuak dira, horrela informazioa ezkutatzen baita.

C jakinez gero erraz ikas daiteke C++aren sintaxia. Izan ere, diseinatzeko eta programatzeko era berbideratzea da zailena OZP teknika gure programetan erabili nahi badugu. OZPko kontzeptuen laburpena egingo dugu datozen orrietan, eta segidan C++erako sarrera.

14.2. DATU-ABSTRAKZIOA: OBJEKTUAK, KLASEAK ETA METODOAK

Datu-abstrakzioa zer den azaltzeko, C-ren fitxategi-maneiua hartuko dugu adibide bezala. Programatzaileek fitxategiak "stream" (datu-korrontea edo fluxua) bezala ikusten dituzte, eta *stream* horren gainean burutzen diren eragiketak (fopen, fgetc,...) erabiltzen dituzte. Hori da fitxategien eredu abstraktua, `stdio.h` iturburu-liburutegian egindako *FILE* Datu-Mota Abstraktua (DMA) definizioa oinarritzat hartzen delako. Programatzaileek ez dute zertan jakin behar fitxategien barruko tratamendua *FILE* mota erabiltzeko. Lan egiteko metodo honi datuen *kapsulatzea* deitzen zaio.

Datu-abstrakzioa datu-mota bat isolatzean eta definitzean datza (jeneralki datu-mota abstraktua izendatuta), datuen kapsulaketa erabiliz. DMA bateko definizioak ez du bere barneko adierazpena bakarrik zehazten, programaren beste moduluek datu horiek maneiatzeko erabili behar dituzten funtzioak ere

aipatzen baitira. Datuen kapsulatzeak DMA bateko barne-adierazpena aldatzea erabiltzen duten programak aldatu gabe bideratzen du (datuak erabiltzeko funtzio berdinak erabiliz, hauen gorputzak egokitu eta gero).

OZPn DMA batetik objektu bat sortzen da. Funtsean, aldagai-multzoak eta aldagai horiekin lan egiteko funtzioek osatzen dute DMA bat. Aldagaiek objektuan gordetzen den informazioa adierazten dute eta funtzioek objektu horrekin egin daitezkeen eragiketak definitzen dituzte. Objektu bat instantzia edo balio desberdinak eduki ditzakeen eskema dela pentsa daiteke. Eskema edo txantiloi hori definitzeko **klase** izena erabili ohi da. Klasea DMAren sinonimoa da eta C++en *class* motaren erazagupena erabiltzen da DMAk definitzeko.

Objektuak tratatzen dituzten funtzioei **metodo** deitzen zaie. Izen hori SmallTalk lengoaiatik hartu da eta C++ lengoaian "klaseen funtzio kideak" deitzen zaie.

Objektu baten gainean eragiketa bat buru dadin **mezu** bat bidaltzen zaio eta horren ondorioz metodo bat erreferentziatzen da. Hau ere SmallTalk-etik hartutako beste kontzeptu bat da. C++n hori egiteko funtzio kide egokia deitu behar da.

14.3. Herentzia eta polimorfismoa

DMAek ez dute egiazko munduko objektuen ezaugarri garrantzitsu bat betetzen: objektuen arteko erlazioa hain zuzen. Munduan objektuak ez daude isolatuak, objektu bakoitza beste objektu batzuekin erlazionatuta dago eta OZPan hau bideratzen da.

Bide horretan, objektu bat deskriba daiteke dauden objektuetatik nola desberdintzen den aipatuz. Hori gertatzen da, adibidez, "X Y bezalakoa da, baina Xk ... dauka eta Xk ... egiten du" esaten denean.

OZPn, beraz, erabat normala da dauden objektuetatik abiatuz beste objektu berri bat definitzea. **Herentzia** hitza erabiltzen da hau adierazteko, klase batek bere ezaugarriak beste klase batetik heredatzen dituela pentsa dezakegulako. Bide horretatik klaseen hierarkia bat finkatzen da, *klase umea klase gurasoe*ngandik heredatzen duelarik. C++eko terminologian, klase gurasoa *oinarrizko klasea* da, eta klase umea *klase eratorria* da.

Objektu askok klase desberdinetatik heredatutako ezaugarriak eduki ditzakete. Adibidez, lehoiak haragijaleak dira jateko ohituren ikuspegitik, eta ugaztunak bere sailkapen biologikoaren aldetik. Honi **herentzia anitza** deitzen zaio, eta nahiz eta lengoaia askotan horretarako aukera ez egon, C++ek bideratu egiten du herentzia anitza.

Polimorfismoa forma bat baino gehiago izateko koalitatea da. Eragile berak objektu desberdinekin modu desberdinetan eragiteko ahalmena izendatzen du polimorfismoak OZPn, hau da, objektu desberdinek modu diferentean erantzuten diotela mezu berberari.

Adibidez:

- Gehitzeko eragilea (+) elementu desberdinak gehitzen ditu: osoko zenbakiak, errealak, konplexuak, ... Polimorfismoari esker eragile bera erabil dezakegu beste datu-motetan, adib. bi string kateatzeko.

- Irudi geometriko desberdin guztiek (triangelua, errektangelua, zirkunferentzia, ...) erantzuten diote 'marraztu' mezuari pantailan beren irudiak marraztuz.

Polimorfismoak objektuen bilduma batean egin daitezkeen eragiketaren sintaxia errazten du.

Adibidez: Irudi geometrikoen azalera kalkulatzeko:

```
/* "irudiak" irudi geometrikoen taula bat dela suposatzen dugu */
for (i=0; i<irudien_kopurua; i++)
     azal = irudiak[i].azalera_kalkulatu();
```

Programatzaileak aurrekoa egin dezake irudi bakoitzak *azalera_kalkulatu* funtzioari eusten diolako. Irudi bakoitzeko, beraz, beraren azalera kalkulatzen duen funtzioa definituko da (triangeluetan oina*altuera/2 adib.).

14.4. C-REN MUGAK OZPRAKO

Objektuak inplementatzen dituzten datuen egiturak C-n adieraz daitezkeen arren, arazo batzuk azaltzen zaizkigu OZPa egiterakoan.

- OZPko oinarrizko legea honakoa dugu: objektu bateko kideak atzitzeko eta maneiatzeko objektuaren funtzio kideak erabili behar dira. Honek ziurtatzen du objektuaren barne-inplementazioren xehetasunak ezkutaturik gelditzen direla beste moduluentzat. Horrela posible da barne-inplementazioa aldatzea beste moduluak konturatu gabe. Beste OZP lengoaiek (OZPL) datu-kapsulaketa egiteko erraztasunak eskaintzen dituzten artean, C lengoaian programatzailearen beraren ardura da datu-egituren kideen atzipen zuzenekoa gerta ez dadin beharrezko diren neurriak hartzea.

- Oinarrizko klasetik herentzia egokia soportatzea programatzailearen ardura da. Funtzioak idatzi behar dira objektu berri batek oinarrizko kideak bereak egin ditzan. Eragozpenak egon ez daitezen, eskema argi bat diseinatu behar da objektuen metodoak (funtzio kide publikoak) deitzeko erabiltzen diren mezuei (funtzio-deiak) ondo erantzun ahal izateko.

14.5. DATU-ABSTRAKZIOA C++N

Aurretik aipatutako adibidearekin jarraituz C lengoaian, fitxategiak definitzen dituen datu-mota honela erazagutzen da `stdio.h` fitxategian:

```
typedef struct
{
char *buff :     // S/Iko bufferra
unsigned flags : // egoera adierazteko
...              // Beste barne-aldagaiak
} FILE;          // Hau da FILE datu-mota
```

FILE egiturarekiko eragiketak aparte azaltzen diren funtzioak dira, bakoitzak parametroren batean FILE erakusle bat erabiltzen duelarik.

C++eko klaseak erabiliz, 14.1 programan azaltzen den bezala defini daiteke FILE datu-mota berbera.

```
/* Fitxategia : File.h
 * fitxategien S/I-rako klase bat
*/
#if !defined(FILE_H)
#define FILE_H

#include <stdio.h> // C-ren S/Irako funtzioen erazagupenak

class Fitxategi
{
  FILE *fp; // C-ren fitxategia
  ... // beste barne aldagaiak
  ...

public:
Fitxategi (const char *izena, // Eraikitzailea const char
         *ireki_modua);
~Fitxategi(); // Suntsitzaile
size_t read(const size_t zenbat, // Irakurri fitxategitik
           const size_t elementuen_luzera,
           void *buffer);
```

```
size_t write(const size_t zenbat, // Idatzi fitxategira
             const size_t elementuen_luzera,
             const void *buffer);
}; // Kontuan hartu ';' jarri behar dela hemen
#endif
```

14.1 programa. *Lehen adibidea C++ erabiliz*

Lehen kode-zati honen aurrean honako hau hartu behar da kontuan:

- Oharrak edo iruzkinak: C++eko konpiladoreak C estandarreko oharrak onartzen ditu (/* eta */ artean), baina, horrez gain, // karaktere-bikotetik lerro-bukaera arte etortzen dena ohar bezala hartzen du.

- *const*aren esanahia: *const* gako-hitza izen baten aurrean jarririk aldagai konstantea dela adierazten da, eta ezin dela bere balioa aldatu programaren bidez. Modu berean, funtzio baten argumentua erakusle bat bada, eta *const* bezala erazagutzen bada, funtzioak ezin izango du erakusleak adierazten duen datua aldatu.

Azaldu dugun 14.1 programako *Fitxategi* klaseak *FILE* motak baino abstrakzio-maila altuagoa emango badu ere, eragiketetarako Cren funtzioak erabiliko ditu. Hori dela eta, *FILE* motaren erakusle bat definitzen da klasean. Geroago ikusiko dugunez, *Fitxategi* klaseko funtzio eraikitzaileak erakusle hori gaurkotzen du klasearen instantzia berri bat sortzen denean.

Fitxategi klasearen erazagupena, *class*, C-ren *struct* motaren antzerakoa da. *public:* hitz erreserbatua garrantzizkoa da; beren atzetik jarritako aldagai eta funtzio kide guztiak publikoak dira programaren edozein lekutatik atzi baitaitezke. Klasearen lehenengo kideak, *public:* hitzaren aurretik azaltzen direnak hain zuzen, pribatuak dira. Kide pribatuak funtzio kideak erabiliz baino ez dira atzitzen. *struct* eta *class* arteko desberdintasun bakarra ondokoa da: *struct*-aren barruan azaltzen dena guztiz publikoa dela.

C++en klase berri bat definitzen dugunean, datu-mota berri bat definitzen ari gara. Konpiladoreak datu horren barruko xehetasunak ezkutatzen ditu, eta funtzio kide publikoak dira beste moduluek datuak atzitzeko eduki ohi duten modu bakarra. Honela, klaseak definitzeak datu-abstrakzioa bideratzen du, eta modulartasuna bultzatu.

Klasearen definizioa

Klase bat erazagutzen denean, bere funtzio kideak ere erazagutzen dira, baina ez dira definitzen, jeneralean beste fitxategi batean definitzen baitira. Horrela, klasearen espezifikazioa goiburuko fitxategi batean egin ohi da (`'.h'` motakoa) eta klasearen inplementazioa izen berdineko `'.cpp'` motako beste batean. Abstrakzioa ondo eginez gero, programatzaileek ez dute klasearen inplementazioaren xehetasunik ezagutu behar, talde-lana erraztuz. Beraz, C++ lengoaian fitxategi-mota berri bat agertzen da: klaseen definizioei dagokien *cpp* mota (*cxx* mota ere erabil daiteke C++en).

C++en eragile berri bat ere agertzen da, esparrua mugatzeko erabiltzen den '::' eragilea hain zuzen. Funtzio kideak definitzerakoan, '::' erabiltzen da definitzen ari garen funtzioa zer klasetako kidea den esateko. Hori dela eta, `Fitxategi::read` idaztean *Fitxategi* klasearen funtzio kide bezala identifikatzen da *read*. Hala ere, "::" eragilea klase-izenik gabe erabil daiteke, funtzio edo aldagai zeharo globala erabiltzen dela adierazteko.

Adibidez:

```
int edozein;// fitxategi guztian ikus daitekeen aldagai
            //globala
void edozer (void)
{
int edozein; //funtzioaren aldagai lokala

edozein=1; //aldagai lokalaren balioa jartzen da
if (::edozein) //aldagai globalaren balioa begiratzen dugu
   zerbait_egin();
}
```

Gauza bera egin daiteke funtzio globalak adierazteko, hobesten den lokala saihestuz.

Funtzio eraikitzaileak eta suntsitzaileak

Fitxategi klasearen adibidean, klasearen izeneko funtzio kide bat dago (`Fitxategi();`), *klasearen funtzio eraikitzailea* deitua, eta beste bat *~Fitxategi()* , klasearen funtzio suntsitzailea dena.

C++eko konpiladoreak klasearen eraikitzaileari deitzen dio (definituta baldin badago) klasearen instantzia berri bat sortzen den bakoitzean. Klase bateko objektuak hasieratzeko erabil daiteke funtzio eraikitzailea (adibidez, objektu baten erakusleren batek adierazten duen memoria gorde eta helbideratzeko). Funtzio eraikitzaileek beti dute klasearen izen bera.

Funtzio suntsitzaile bat ere defini daiteke klase bakoitzeko, zerbait egiteko objektu bat suntsitzerakoan (adibidez, erabilitako memoria dinamikoa itzultzeko). Tilde "~" karakterea eta klasearen izenarekin osatzen da funtzio suntsitzaileen izena. Fitxategien adibidean, funtzio suntsitzaileak fitxategia itxi egiten du.

14.6. Herentzia eta polimorfismoa C++en

Herentzia

Klase bat erazagutzen denean, ezaugarriak beste klase batetik heredatzen dituela adieraz daiteke. Horretarako erazagupenaren lehenengo lerroan ":" sinboloa jartzen da eta jarraian oinarrizko klasearen izenak.

Adibidez, "borobil" klasea "irudia" klasea oinarritzat hartuko duela definitu nahi badugu, horrela jarriko dugu lehenengo lerroa:

```
class borobil: public irudia
{
// aldagai eta funtzio kideak
...
}
```

Kasu honetan, *irudia* da oinarrizko klasea, eta *borobil* klase eratorria. *Public* gako-hitza jartzen da irudi-motaren edozein aldagai edo funtzio publiko atzituko dela adierazteko.

Polimorfismoa eta lotura dinamikoa

C++en oinarrizko klasearen funtzio bat alda dezake klase eratorriaren funtzio batek. Horrez gain, klase eratorriaren objektu bat adierazteko oinarrizko klaseari zuzendutako erakuslea ere erabil daiteke. Bi aukera hauekin C++eko klaseetan polimorfismoa antola daiteke.

Ikus dezagun berriro "irudia" klasearen adibidea: oinarrizko klasearen datuak eta funtzio arruntak kapsulatuta daude. Irudi konkretu baten klasea "irudia" klasearen eratorria izango da (triangelu klasea, borobil klasea, etab.). Funtzio amankomun bat "marraztu" da, irudiaren marrazkia pantailan jartzen duena. Irudi bakoitzaren marrazkia desberdina denez, oinarrizko klasean "marraztu" funtzioa "virtual" gako-hitzarekin definitzen da:

```
class irudia
{
// datu pribatuak
...
public:
   virtual void marraztu(void) const {};
   // Beste funtzio kideak
};
```

virtual hitz erreserbatuak C++eko konpiladoreari honakoa esaten dio: oinarrizko klaserako definitzen den "marraztu" funtzioa erabiliko da soilik baldin eta klase eratorriek birdefinitzen ez badute.

Klase eratorri batean "marraztu" funtzioa birdefini daiteke:

```
class borobil: public irudia
{
// datu pribatuak
...
public:
virtual void marraztu(void) const;
// Beste funtzio kideak
};
```

Geroago *borobil* klasearen marraztu funtzioaren inplementazioa burutu behar da. Gauza bera egin daiteke "triangelu" eta "errektangelu" klaseekin. Horrela, funtzio kide bera klase desberdineko instantziei aplika dakieke eta C++eko konpiladoreak "marraztu" funtzio egokiari deituko dio.

```
// borobil eta errektangeluen instantziak sortu
borobil B1 (100,100,50);
errektangelu L1(10.,20.,30.,40.);

B1.marraztu(); // borobil klasearen marraztu funtzioa deitzen da
L1.marraztu(); // errektangelu klasearen marraztu funtzioa
               //deitzen da
```

Honekin polimorfismo estatikoa lortzen dugu, konpilazio-denboran zein funtziori deitu behar zaion jakina baita. Dena den, polimorfismo dinamikoa (lotura dinamikoa) interesgarriagoa da guretzat. Hau gertatzen da alegiazko (birtual) funtzio bati objektu baten erakuslearen bidez deitzen diogunean, eta objektuaren mota ezezaguna denean konpilazio-denboran.

C++en hau posible da C++ek oinarrizko klasearen erakuslea erabiltzea onartzen duelako eratorritako klasearen objektu bat erakusteko.

Adibidez, eta irudiekin jarraituz, irudi batzuk sortu nahi baditugu, eta geroago marraztu, honako hau egin daiteke.

```
int i;
irudia *irudiak[2]; // oinarrizko klaseei erakusleak

// irudi sortu eta erakusleak gorde :
irudiak[0] = new borobil(100, 100, 50);
irudiak[1] = new errektangelu(10, 10, 20, 30);

// irudiak marraztu :
for (i=0; i<2; i++)
   irudiak[i]->marraztu();
```

Erakusleen bidez objektu bakoitzaren funtzio kideari dei diezaiokegu. "marraztu" alegiazko funtzioa denez, exekuzio-denboran klase eratorriaren funtzioa aktibatzen da.

Adibidea: Irudien klase berri bat eransten

Hauek dira irudi klasearen klase eratorri berri bat eransteko eman behar diren urratsak:

1) klasearen definizioa goiburuko fitxategian jarri, normalean bere klase gurasoaren fitxategi berean (adib. triangelu klasea).

2) triangelu klasearen funtzio kideak beste fitxategi batean definitu (adibidez triangelu.cpp)

Bi urrats hauetan sortzen den kodea 14.2 programako a) eta b) ataletan azaltzen dena izan liteke.

```
/*********************************************************
*** Fitxategia:Irudia.h                                 ***
*** triangelu klasearen erazagupena                     ***
*********************************************************/
/* aurretik irudia klasea definiturik egongo da */
class triangelu: public irudia
{
```

```
private:
   double x1,y1; // erpien koordenatuak
   double x2,y2;
   double x3,y3;
public:
   triangelu (double x1, double y1, // klasearen eraikitzailea
              double x2, double y2,
              double x3, double y3);
   double azalera_kalkulatu(void) const;
   void marraztu(void) const;
};
```

(a) goiburuko fitxategia

```
/*************************************************************
*** Fitxategia:triangelu.cpp                                ***
*** triangelu irudia klasearen definizioa C++en             ***
*************************************************************/
#include "irudi.h" //oinarrizko klasearen definizioa
triangelu::triangelu( double xa, double ya, double xb,
                      double yb, double xc, double yc)
{
x1 = xa; y1 = ya;
x2 = yb; y2 = yb;
x3 = xc; y3 = yc;
}

double triangelu::azalera_kalkulatu(void) const
{
double azalera, x21, y21, x31, y31;

x21= x2 - x1;
y21 = y2 - y1;
x31 = x3 - x1;
y31 = y3 - y1;
azalera = fabs (y21 * x31 - x21 * y31) / 2.0;
return (azalera);
}

void triangelu::marraztu(void) const
{ // hobe zitekeen liburutegi grafiko bat erabiliz
printf("Marrazkia: triangeluaren erpinak \n"
   " (%f , %f) (%f , %f) (%f , %f) \n", x1,y1,x2,y2,x3,y3);
}
```

(b) cpp motako fitxategia

```
int i;
irudi *irudiak[3];
// irudi batzuen sorrera
irudiak[0] = new borobil(100, 100, 50);
irudiak[1] = new errektangelu(10, 10, 20, 30);
irudiak[2] = new triangelu(100, 100, 200, 100, 150, 50);

// azalerak kalkulatzea
for (i=0; i<3; i++)
  printf(" %d Irudiaren azalera = %f\n", i,
    irudiak[i].azalera_kalkulatu());
                        (c) erabilera
```

14.2 programa. *triangeluen klasea*

Aurreko urratsak emanez gero gure programan triangelu klasea erabiltzen has gaitezke. Beharrezkoa da, edozein kasutan, programa *triangelu.cpp* fitxategiarekin konpilatzea kode hau erabili ahal izateko.

Exekuzio-denboran klase baten instantziak sortzeko aukera da C++en ezaugarri garrantzitsu bat. C lengoaian *calloc* eta *malloc* funtzioak erabiltzen dira dinamikoki memoria erreserbatzeko, eta gero hartutako memoria *free* errutinaren bitartez askatzen da. Objektuak sortzeko eta suntsitzeko *new* eta *delete* eragileak dauzka C++ek.

new eragileak sortutako klasearen erakusle bat itzultzen du, eta berehala klasearen eraikitzaileari deitzen dio instantzia berria hasieratzeko. *delete* eragilearen bidez objektuak suntsitzen dira, eta beraiei egokitutako memoria askatzen da.

14.7. C++ DESKRIBATZEN

C++eko programek C antzerako funtzioak erabil baditzakete ere, betebehar berriak jartzen ditu funtzio horien erabilpena eraginkor eta ziurrago izan

dadin[1]. Beste gauzen artean C++en ezinbestekoa da funtzioen prototipoa (hau da, funtzioen erazagupena) jartzea erabili aurretik.

Argumentu lehenetsiak

C++eko funtzioei besterik ezean erabiliko diren argumentuak jar dakizkieke prototipoa idazterakoan. Adibidez, pantailan leiho bat idazten duen *leiho_ipini* funtzioa definitzen dugunean, zehatz daitezke leihoaren neurrirako, kokapenerako eta kolorerako balio lehenetsiak. Ondoko definizioan:

```
window idatz_leihoa( int x = 0, int y = 0, int altuera = 50,
          int zabalera = 100, int kolorea = 0);
```

esleipenak ez dira hasierako balioak, balio lehenetsiak baizik. Erabilpena egitean aukera desberdinak daude, besteren artean hurrengoak:

```
window w; // Leiho bat
w = idatz_leihoa(); // idatz_leihoa(0, 0, 50, 100, 0)
w = idatz_leihoa(100, 20); // idatz_leihoa(100, 20, 50, 100, 0)
```

Jarritako parametroak sekuentzian elkartzen dira prototipoan azaltzen direnekin. Parametroren bat zehazten ez bada, dagokion argumentu lehenetsia aukeratzen da.

Gainkargatutako funtzioak

C lengoaian funtzio bat definitzen denean parametroen mota zehaztu behar da. Parametro-mota desberdinetarako, nahiz eta eragiketa berbera definitu nahi, funtzio-izen desberdinak erabili behar dira. C++en, berriz, izen berdineko funtzio desberdinak eduki ditzakegu parametroen kopuru eta motaren arabera. Horrelakoa gertatzen denean, funtzioen izena gainkargatuta dagoela

[1] OZParen ikuspegitik eraginkorragoak eta ziurragoak dira, baina programatzeko ahalmena galtzen da, eta, nolabait esateko, Ck eskaintzen duen sormen-atal bat galdu egiten da, programatzailearen iniziatiba murriztuz.

(overloaded) esaten da. Antzerako lana egiten duten eta erlazionatuta dauden funtzioei izen adierazgarri bat jarri nahi diegunean gainkargatu ohi dira funtzioak. Adib. C++en posible da osokoen biderkaketa eta errealen biderkaketa burutzea bietan *biderkaketa* izeneko funtzioa erabiliz (ikus 14.3 programa).

Lineako funtzioak

Lineako funtzioak aurrekonpiladoreen makroak bezalakoak dira, konpiladoreak funtzioaren gorputza txertatzen baitu dei bakoitzean. Horrelako funtzioak erabil daitezke eragiketak eraginkorragoak izan daitezen. Makroak bezala lineako funtzioak erabili ohi dira funtzioak laburrak direnean eta funtzio deiek dakarten gainkarga saihestu nahi denean. Lineako funtzioak makro antzerakoak izan arren, bien artean bi desberdintasun nabari daude:

a) Makroek ordezkapenetan sortzen dituzten arazoak (parentesi asko jartzera behartzen dutenak) saihesten dira lineako funtzioetan.
b) Lineako funtzioak beste motako funtzioak bezala erabiltzen dira, eta horrela ez dute albo-ondoriorik ematen (makroek ematen dituzten bezala) eta gainkargatuta egon daitezke.

```
#include <stdio.h>
// zuzeneko funtzio bat definitu bi osoko zenbaki biderkatzeko
inline int biderkaketa( int x, int y)
{
return (x*y);
}

// gainkargatutako funtzioa bi double biderkatzen dituena
inline double biderkaketa(double x, double y)
{
return (x*y);
}

main ()
{
printf("5 eta 6 arteko biderketa = %d\n", biderketa(4+1,6));
printf("3.1 eta 10.0 arteko biderketa = %f\n",
   biderketa(3.0+0.1,10.0));
}
```

Programa hau konpilatu eta exekutatzen dugunean honelako emaitzak lortzen ditugu:

```
5 eta 6 arteko biderketa = 30
3.1 eta 10.0 arteko biderketa = 31.000000
```

14.3 programa. *Lineako funtzioak*

14.3 programan biderkaketa burutzen duen lineako funtzioa azaltzen da. Funtzio hori makroen bidez horrela definitu izan balitz lehen emaitza okerra litzateke, ordezkatzean 4+1*6 zehaztuko bailitzateke.

Funtzio adiskideak

C++ek gako-hitz berri bat eskaintzen digu OZPko teknikak erabiltzea errazteko: *friend* adierazlea. Datuen kapsulaketarako arauek diotenez, klase baten datu pribatuak funtzio kide publikoek baino ezin dituzte atzitu. Datu pribatuak itzultzen dituzten funtzioak programa ditzakegu, baina kasu batzuetan ez da batere eraginkorra izaten. Kanpoko funtzio batzuei datu pribatuak atzitzeko aukera eskaintzen die C++ek, OZPren legeak apurtuz. Horretarako funtzioari "adiskide" ezaugarria eslei dakioke, klasearen izenaren aurrean *friend* gako-hitza jarriz.

```
#include <stdio.h>

class konplexu
{
float erreala, irudikaria;
public:
   friend konplexu batu(konplexu a, konplexu b);
   friend void idatzi(konplexu a);
   konplexu() {
              erreala = 0.0;
              irudikaria = 0.0;
              }
   konplexu(float a, float b)  {
                               erreala = a ;
                               irudikaria = b;
                               }
};
```

```
konplexu batu (konplexu a, konplexu b)
{
konplexu z;

z.erreala = a.erreala + b.erreala;
z.irudikaria = a.irudikaria + b.irudikaria;
return z;
}

void idatzi (konplexu a)
{
printf(" (%f+ i* %f)",a.erreala, a.irudikaria);
}

main()
{
konplexu a, b, c;

a = konplexu(1.5, 2.1);
b = konplexu(1.1, 1.4);
idatzi(a);
printf(" eta ");
idatzi(b);
printf("\n konplexuen arteko batuketa =\n");
c = batu(a, b);
idatzi(c);
}
```

Programa honek bi funtzio adiskide erabiltzen ditu, baina kanpotik erabiltzeko ez da klasearen izena aipatu behar. Programaren emaitza hau da:

```
(1.500000 + i*2.100000) eta (1.100000 + i*1.400000)
konplexuen arteko batuketa =
(2.600000 + i* 3.500000)
```

14.4 programa. *Funtzio adiskideak*

14.4 programan, zenbaki konplexuak batzeko eta idazteko funtzioak azaltzen dira eta bertan erabiltzen da *friend* ezaugarria.

Erreferentziaren bidezko datu-motak argumentu bezala

C-n, normalean, argumentuak balioaren bidez trukatzen dira, hau da, funtzio bat deitzerakoan, argumentuaren balioa kopiatzen da memoriako zati batean, "parametroen pila" denekoa, eta funtzioek memoriaren zati horrekin burutzen dituzte eragiketak. Adibide bezala ikus dezagun hurrengo funtzioa:

```
void bikoitza(int a)
{
a = a*2;
}
...
int x = 5;
// dei bikoitza funtzioari
bikoitza(x);
printf("x = %d\n",x);
```

Programa honek *x = 5* idatziko digu pantailan baina trukea erreferentziaren bidez eginez *x = 10* idatziko zuen. Azken hori egiteko C-n "*" eta "&" eragileak erabili behar dira derrigorrez, eta horrek sintaxia zailtzen du. C++ek beste aukera bat ematen digu argumentuak erreferentzia bidez pasatzeko: "alias" edo sinonimo bat jar diezaiokegu edozein datu-motari. Sintaktikoki, ampersand bat ("&" karakterea) gehitzen zaio datuaren definizioari.

Adibidez:

```
int i = 5;
int *punti = &i;// hasieran i aldagaia erakusten du punti
int &errei = i;  // i aldagaiaren erreferentzia da errei
```

errei erabili dezakegu i edo **punti* jarri nahi dugun edozein lekutan. Honela idazten badugu

```
errei  = errei+10; // i aldagaiari 10 batzen zaio
```

i aldagaiak orain 15 balioa izango du, *errei* i-ren beste izena baita.

Datu-motaren erreferentzia erabiliz, *bikoitza* funtzioa modu errazago batean idatz daiteke, funtzioaren definizioan * eragilea ez baita zergatik erabili behar:

```
void bikoitza(int &a)
{
a = a*2;
}
...
int x = 5;
// Argumentua erreferentzia bidez pasatzen da automatikoki
bikoitza(&x);
printf("x = %d\n",x);
```

Programak `"x = 10"` idazten du pantailan.

Klaseak balioaren bidezko parametro bezala pasatzen direnean, datuak kopiatzeko denbora handia izan daiteke. Hori da erreferentzia bidezko parametroak erabiltzeko beste arrazoia.

Gainkargatutako eragileak

C++ek izen berdineko funtzioak erazagutzea onartzen duen modu berean, izen berdineko eragileak erabil daitezke eragiketa desberdinak burutzeko —edo hobeki esanda, eragile berberarekin gauza desberdinak egiten uzten digu—. Horrela, eragilearen esanahia aldatzea posible da, beste datu-mota batekin egingo diren eragiketen definizioa jarriz. Klaseak DMA berriak izanik, eragileak gainkargatzean klasearen datu-motarekin lan egiten duten eragiketak defini ditzakegu.

Adibidez, bi zenbaki konplexu gehitzeko '+' eragilea gainkarga daiteke 14.5 programan azaltzen den bezala.

```
class konplexu
{
float erreala, irudikaria;
public:
   friend konplexu operator+(const konplexu &a,const
   konplexu &b);
   konplexu() {
              erreala = 0.0;
              irudikaria = 0.0;
              }
   konplexu(float a, float b) {
                              erreala = a ;
                              irudikaria = b;
                              }
};

konplexu operator+(const konplexu &a, const konplexu &b)
{
konplexu z;

z.erreala = a.erreala + b.erreala;
z.irudikaria = a.irudikaria + b.irudikaria;
return z;
}
```

14.5 programa. *Gainkargatutako eragilea*

Ikus dezakegunez, eragile bat definitzea funtzio bat definitzea bezalakoa da, funtzioaren izena jartzen den tokian aldaketa sintaktiko bat azaltzen delarik. Aldaketa hau eragilearen adierazlea *operator* gako-hitzaren atzetik idaztea da.

Orain + eragilea bi konplexu batzeko beste zenbakizko motetan bezala erabil dezakegu:

```
konplexu a, b, c;
a = konplexu (1.5, 2.1);
b = konplexu (1.1, 1.4);
c = a + b; // Honela gehitzen dira bi zenbaki konplexuak
```

Datuen erazagupena C++en

ANSI Cn ezin dira nahastu datuen erazagupenak programaren aginduekin. Bloke baten hasieran aldagai guztiak erazagutu behar dira, eta ondoren aldagai horiekin lan egiten duten aginduak idatzi.

C++ek ez ditu erazagupenak bestelako programaren sententzietatik bereizten. Horregatik datu erazagupenak programaren edozein lekutan idatz ditzakegu.

C++eko ezaugarri hau interesgarria da, edozein lekutan aldagaiak erazagutzeko eta hasieratzeko aukera ematen baitigu. Honela, gure programak irakurgarriago bihur daitezke, erabili behar ditugun lekuetan aldagaiak erazagutuko direlako.

C++ek duen beste ezaugarri bat da *struct*-en izena definizioan bertan erabiltzeko aukera. Adibidez, 13. kapituluan agertzen den listaren definizioa (`lista_def.h` fitxategian) horrela geldituko litzateke:

```
/* lista_def.h */

struct elem
{
elem *ondoko; // Erazagututako datu-motarako erakuslea
int lehentasuna;
char *infor;
};

struct lista
{
elem *lehena;
elem *azkena;
};

lista *lehenengoa; // Lista motako elementu baten definizioa
```

Egitura baten izena erabil daiteke, beraz, *struct* gako-hitza erabili gabe.

C eta C++en arteko beste desberdintasunak

C++ eta Cren artean dauden desberdintasun gehienak aipatu dira. Sintaxiaren ikuspuntutik dauden desberdintasunen laburpena azalduko da datozen lerroetan.

Gako-hitz berriak

C++ek 15 gako-hitz berri gehitzen dizkio gako-hitzen zerrendari. ANSI Cko programek hauetakoren bat erabiltzen badute aldatu egin beharko lirateke hitz horiek C++eko gako-hitz bezala interpreta ez daitezen.

Hitz berrien zerrenda honakoa da:

```
asm         catch       class       delete      friend
inline      new         operator    private     protected
public      template    this        throw       virtual
```

Zerrenda honen hitz batzuk azaldu dira aurreko orrietan. Beste batzuk (*throw* eta *template,* alegia) ez dira gaur egun erabiltzen, baina etorkizunean funtzionaltasun berriak sartzeko asmoarekin erreserbatu egin dira jadanik. Adibidez, *template* gako-hitza funtzioen familiak edo motak definitzeko erabiliko da, eta *catch* eta *throw* hitzekin salbuespenak tratatzeko mekanismo bat burutu nahi da.

Funtzioen prototipoak

ANSI Cn, funtzio baten prototipoa ez bada jartzen eta erabiltzen bada definitu aurretik, konpiladoreak *int* motako emaitza ematen duela suposatzen du. C++ek prototipoen erabilpena behartzen du eta errore bat ematen du funtzio bat erazagutu gabe erabiltzen bada.

Adibidez, C++ek errore bat ematen du programa hau konpilatzerakoan:

```
main()
{
printf("Kaixo, mundua!\n");
}
```

ANSI Cn errorea konpontzen da `stdio.h` txertatuz, fitxategi horren barruan *printf* funtzioa erazagutzen baita. Hala ere, C++eko konpiladoreak errore batzuk eman ditzake Cn ondo dauden programak konpilatzerakoan, funtzioen prototipoak ez baziren jartzen *int* datu-mota itzultzen zutenean.

Adibidez, Cko programa askok *malloc* funtzioa honela erazagutzen eta erabiltzen dute:

```
char *malloc();
int *datuak;
datuak = (int *) malloc(1024);
```

Kode honek errore bat ematen du C++eko konpiladoreak itzultzen duenean, hemen argumentu-zerrenda hutsen tratamendua desberdina baita. ANSI Cn argumentu-zerrenda hutsik badago, funtzioak 0 argumentu edo gehiago eduki ditzake, eta C++en berriz, *void* bezala interpretatzen da. *malloc* funtzioaren deia aurkitzen duenean, C++eko konpiladoreak errore bat ematen du ez baitu argumenturik espero.

const aldagaiak

C++en *const* aldagaiak hasieratu behar dira erazagutzerakoan. ANSI Cn, berriz, ez da horrelakorik gertatzen. Adibidez:

```
const buffer;         // Zuzena ANSI Cn baina ez C++en
const buffer = 512    // Zuzena bietan
```

const aldagaien beste berezitasun adierazgarri bat honakoa da: edozein adierazpen konstantetan erabili ahal izatea, konpiladoreak aldagaiaren balioak edozein momentutan ezagutzen baititu. Horrela posible da hau egitea:

```
const luzera = 512;
const buffer[luzera]; // Onartuta C++en baina ez ANSI Cn
```

void erakusleak

ANSI C-n *void ** datu-motari erakusleak erabiltzea uzten zaigu segidan edozein erakusleri esleitzeko aukerarekin. C++ek behar den *casting* edo bihurketa esplizitua erabiltzera behartzen du. Ondoko adibide honek dagoen desberdintasuna erakusten digu:

```
void *p_void;
int i, *p_i;
p_void = &i;          // Onartuta bietan
p_i = p_void;         // Onartuta Cn, baina ez C++en
p_i = (int *)p_void;  // Onartuta bietan
```

"C programazio-lengoaia" izeneko liburu hau argitaratzeko ideia zaharra da. UEUn orain dela hamar bat urte Ander Basalduak azaldu zigun lengoaia honek zuen etorkizuna. Zenbait urte geroago, UEUn bertan, ikastaro bat prestatu genuen, liburu honetan azaltzen diren zenbait adibide ordukoak izanik. Orain dela bi urte, eta liburuaren prestakuntzaren lehen fase gisa, Elhuyar Zientzia eta Teknika *aldizkarian 14 kapituluko ikastaro bat prestatu genuen hainbat zenbakitan Montse Maritxalarren laguntzarekin. Ikastaro haren materiala liburu honen oinarria izan arren, azken urte honetan liburua osatu da, testua birplanteatuz eta osatuz, adibideak gehituz eta programak konpilatuz eta egiaztatuz. Azkenik, C lengoaia ordezkatzen ari den C++ lengoaiari buruzko kapitulu bat gehitzea egoki iruditu zaigu.*

ELHUYAR